2017

AUTONOMOUS QUADROTOR COLLISION AVOIDANCE AND DESTINATION SEEKING IN A GPS-DENIED ENVIRONMENT

Thomas C. Kirven
University of Kentucky, thomasckirven@gmail.com
Author ORCID Identifier:
https://orcid.org/0000-0002-1992-679X
Digital Object Identifier: https://doi.org/10.13023/ETD.2017.473

AUTONOMOUS QUADROTOR COLLISION AVOIDANCE AND DESTINATION SEEKING IN A GPS-DENIED ENVIRONMENT

THESIS

A thesis submitted in partial fulfillment of the
requirements for the degree of Master of Science
in Mechanical Engineering in the College of
Engineering at the University of Kentucky

by

Thomas Cooper Kirven

Lexington, Kentucky

Director: Dr. Jesse B. Hoagg, Professor of Mechanical Engineering

Lexington, Kentucky

2017

ABSTRACT OF THESIS

AUTONOMOUS QUADROTOR COLLISION AVOIDANCE AND DESTINATION SEEKING IN A GPS-DENIED ENVIRONMENT

This thesis presents a real-time autonomous guidance and control method for a quadrotor in a GPS-denied environment. The quadrotor autonomously seeks a destination while it avoids obstacles whose shape and position are initially unknown. We implement the obstacle avoidance and destination seeking methods using off-the-shelf sensors, including a vision-sensing camera. The vision-sensing camera detects the positions of points on the surface of obstacles. We use this obstacle position data and a potential-field method to generate velocity commands. We present a backstepping controller that uses the velocity commands to generate the quadrotor's control inputs. In indoor experiments, we demonstrate that the guidance and control methods provide the quadrotor with sufficient autonomy to fly point to point, while avoiding obstacles.

KEYWORDS: quadrotor, quadcopter, autonomous, collision avoidance, destination seeking

Thomas Cooper Kirven
December 8, 2017

AUTONOMOUS QUADROTOR COLLISION AVOIDANCE AND DESTINATION SEEKING IN A GPS-DENIED ENVIRONMENT

by

Thomas Cooper Kirven

Dr. Jesse B. Hoagg

Director of Thesis

Dr. Haluk E. Karaca

Director of Graduate Studies

December 8, 2017

Acknowledgments

To all reading this thesis, thank you for your interest in a project I worked very hard on. I hope you find it interesting, if not informative and helpful.

I would like to thank my parents for their love and support, and the sacrifices they have made for my siblings and me growing up. I could not have accomplished this without them. I would like to thank my brothers and sister for their interest in my endeavors and their support throughout my years as a graduate student. Special thanks to my brother, Tucker Kirven, for helping me with my final experiment and providing good advice on how to be a better engineer.

My lab mates, Brandon Wellman, Zack Lippay, Shaoqian Wang, Roshan Chavan, Chris Heintz, Alireza Moosavi, and Zahra Abbasi made my time in the lab and the classroom worthwhile, and provided friendship, advice, and insightful discussions when I needed it. To them, I am very grateful.

I would like to thank Suzanne Smith for being a great advanced dynamics professor and serving on my thesis committee. She helped me derive the quadrotor equations of motion and believed in my abilities from the beginning to undertake this project. I would like to thank Michael Seigler for helping clarify a confusing aspect of quadrotor dynamics and serving on my thesis committee.

About a year and a half ago I began working on this thesis. With the help of my advisor, Jesse Hoagg, we decided upon this topic because it would provide him with a resource for future research, and was a focused consolidation of my desires to work on a tangible, navigation related project. I would like to thank Jesse Hoagg for the many hours he has dedicated to shaping me into a better writer and scientific thinker. In our weekly meetings he was always very patient with me, and taught me invaluable lessons that will serve me well in my professional career.

Lastly, I would like to thank Isaac Newton, Charles Babbage, Rudolf Kalman, and all of the scientists and engineers whose work made this project feasible.

Contents

List of Figures

Nomenclature

Physical Parameters

m	mass of the quadrotor (kg)
g	magnitude of acceleration due to gravity (m/s^2)
k	propeller thrust constant (kg·m/rad^2)
$\vec{I}_{\mathrm{q}}$	physical inertia matrix
I_{q}	$\vec{I}_{\mathrm{q}}$ resolved in B
I_{xx}, I_{yy}, I_{zz}	moments of inertia (kg·m^2)
l	thrust moment arm (m)
b	propeller drag constant (kg·m^2/rad^2)
I_{p}	propeller moment of inertia about axis of rotation (kg·m^2)
r_{p}	radius of propeller (m)

Quadrotor Equations of Motion

I	inertial frame
o_{I}	origin of I
$\hat{\imath}, \hat{\jmath}, \hat{k}$	orthogonal unit vectors of I
B	body frame
o_{B}	origin of B (quadrotor center of mass)
$\hat{b}_1, \hat{b}_2, \hat{b}_3$	orthogonal unit vectors of B
$\vec{p}_{o_{\mathrm{B}}/o_{\mathrm{I}}}$	position of o_{B} relative to o_{I} (m)

x, y, z	components of $\vec{p}_{o_\mathrm{B}/o_\mathrm{I}}$ resolved in I (m)
$\vec{a}$	acceleration of o_B relative to o_I with respect to I (m/s^2)
a	$\vec{a}$ resolved in I (m/s^2)
ψ, θ, ϕ	yaw, pitch, and roll Euler angles (rad)
$\mathbb{O}$	orientation matrix of B relative to I
$\vec{\omega}$	angular velocity of B relative to I (rad/s)
ω	$\vec{\omega}$ resolved in B (rad/s)
p, q, r	components of ω (rad/s)
ω_i	angular speed of ith propeller (rad/s)
T	magnitude of thrust (N)
$\vec{\tau}$	moment from propeller thrust and drag (N·m)
$\tau_{b_1}, \tau_{b_2}, \tau_{b_3}$	components of $\vec{\tau}$ resolved in B (N·m)
$\vec{H}_\mathrm{p}$	angular momentum of propellers (kg·m^2/s)
$\vec{H}$	total angular momentum of quadrotor (kg·m^2/s)

Attitude Control

$\phi_\mathrm{d}, \theta_\mathrm{d}, \psi_\mathrm{d}$	desired Euler angles (rad)
e_ϕ, e_θ, e_ψ	roll, pitch, and yaw errors (rad)
k_ϕ, k_θ, k_ψ	roll, pitch, and yaw proportional gains
ω_d	desired angular velocity (rad/s)
e_ω	angular velocity error (rad/s)
K_ω	positive-definite matrix of angular velocity gains
G_{ψ_d}	low-pass filter for desired yaw
ψ_c	yaw command (rad)

Velocity Control

u, v, w	components of $\dot{\vec{p}}_{o_\mathrm{B}/o_\mathrm{I}}$ resolved in I (m/s)
$u_\mathrm{d}, v_\mathrm{d}, w_\mathrm{d}$	desired velocities (m/s)
e_u, e_v, e_w	velocity errors (m/s)
k_u, k_v, k_w	velocity proportional gains
G_c	low-pass filter for velocity commands
$u_\mathrm{c}, v_\mathrm{c}, w_\mathrm{c}$	velocity commands (m/s)

Outer-Loop Guidance

$\vec{p}_i$	position of ith point on an obstacle relative to o_B (m)
p_i	$\vec{p}_i$ resolved in B (m)
n	number of points $p_1, ..., p_n$
r_c	collision radius (m)
$a_\mathrm{i}, b_\mathrm{i}, c_\mathrm{i}$	inner ellipsoid's semi-principle lengths (m)
$a_\mathrm{o}, b_\mathrm{o}, c_\mathrm{o}$	outer ellipsoid's semi-principle lengths (m)
E_i	inner ellipsoid
E_o	outer ellipsoid
U_max	maximum potential
U	ellipsoidal potential function
$\vec{p}_\mathrm{g}$	position of quadrotor's destination relative to o_B (m)
p_g	$\vec{p}_\mathrm{g}$ resolved in I (m)
s_d	desired speed for the quadrotor (m/s)
d_s	stopping distance (m)
V_g	goal velocity for the quadrotor (m/s)
n_e	number of points $p_1, ..., p_n$ contained in E_o

Implementation and Experiment

T_{s}	sample time (s)
$\mathcal{O}_{ij}$	entry in ith row and jth column of $\mathcal{O}$
$\hat{\dot{u}}, \hat{\dot{v}}, \hat{\dot{w}}$	estimated inertial accelerations (m/s^2)
s_{m}	measured speed (m/s)
h_{m}	measured height (m)
κ_{flow}	standard deviation of optical flow measurement
pw_i	pulse width sent to ith ESC-motor pair (ms)
l_{xx}	distance from o_{B} to axis of rotation (m)
P_{xx}	period of rotation (s)
$x_{\mathrm{d}}, y_{\mathrm{d}}, z_{\mathrm{d}}$	components of $p_{\mathrm{g}}(0)$ (m)

List of Files

(a) Thesis document (this document)

(b) Exp_2.mov: video example 1 of quadrotor obstacle avoidance and destination seeking experiment

(c) final_exp.mov: video example 2 of quadrotor obstacle avoidance and destination seeking experiment

(d) flight.cpp: flight guidance, navigation, and control script

(e) flight.hpp: header file for flight.cpp

Chapter 1 Introduction and Motivation

Improvements in sensing and computing have increased the capabilities of small unmanned aerial vehicles (UAVs). Quadrotors in particular have gained popularity because of their maneuverability, and their simple mechanical design as compared to traditional helicopters [1, 2]. Quadrotors are capable of vertical take off and landing, hovering, pivoting about a fixed point, and instantaneous acceleration in three dimensions. These capabilities make quadrotors ideal for developing aerial robotic applications and testing new control strategies [3, 4]. Traditionally, flight guidance and navigation is handled by a human pilot, who operates the quadrotor using either line of sight or a video feed from an onboard camera. However, recent computer and sensor advances (e.g., miniaturization, reduced cost) have made autonomous flight of quadrotors more feasible [5,6], particularly in GPS-denied environments [7,8]. Flight autonomy for quadrotors could simplify their use in current applications by removing the need for a pilot, and enable applications involving search and rescue [9, 10], distributed sensing, cooperative control [11–13], cooperative surveillance [14], and cooperative construction [15]. For example, a group of autonomous quadrotors could create a terrain map of a potential construction site, or gather high-resolution pressure or temperature data for atmospheric physics research. Similarly, autonomous quadrotors could provide agricultural monitoring or mail delivery.

1.1 Inner-Loop Attitude Control

The dynamics of a quadrotor can be derived from the Newton-Euler equations, which contain a translational subsystem and a rotational subsystem. The attitude control problem, described in [16], considers control of the rotational subsystem, which is a crucial prerequisite for controlling a quadrotor's translational states. Translational movement is driven by the thrust force, which acts along only one body-fixed axis. Therefore, a quadrotor must change its attitude (or equivalently, orient its thrust force) to achieve translation in three dimensions. The differential thrust and drag generated by the propellers induce three orthogonal body torques. The thrust force and body torques constitute a quadrotor's control inputs.

Linear control methods have been applied extensively to quadrotors because of their simplicity and robustness in the near-origin regime. For example, attitude-stabilizing controllers using PID and LQ techniques are presented in [17]. In, [17] the LQ controller is developed to stabilize the quadrotor about the origin (i.e., hover orientation) from initial angles up to $\pm 90°$ roll. In [18], the quadrotor's equations of motion are linearized near a desired attitude, and PID controllers stabilize each axis about this attitude. PID control is computationally simple and robust for small deviations from the desired attitude. However, at large deviations from the desired attitude, the linearized equations don't capture the true dynamics, and instabilities can arise while using PID control. If large rotations are expected, it can be beneficial to use nonlinear control methods. In [19], two nonlinear attitude-stabilizing controllers are presented that use PD and PD^2 feedback structures. Here, P (proportional) is quaternion feedback, D (derivative) is body angular rate feedback, and D^2 (second derivative) is body angular rate and quaternion velocity feedback. The first controller is model dependent and provides near exponential stability of the origin (hover orientation) using PD^2 feedback. The second controller is model independent and provides global asymptotic stability of the origin using PD feedback. Other nonlinear control methods for rotorcraft include geometric [11,20–22], sliding mode [23–25], and backstepping [24,26–29]. Sliding mode controllers are robust to model uncertainties, while backstepping relies on model information and is a constructive approach for designing recursively stabilizing controllers. Many backstepping controllers that use Euler angle parameterizations (e.g., [24,26,27,29]) linearize the quadrotor's rotational kinematics, that is, they assume the time derivative of the Euler angles are equal to the body angular rates. This approach simplifies the backstepping process, but does not capture all of the nonlinearities in the rotational dynamics.

In this thesis, we present an inner-loop attitude controller based on the Euler angle kinematics and the unsimplified rotational dynamics. The controller guarantees local exponential stability of any attitude equilibria with pitch between $\pm 90^\circ$.

1.2 Middle-Loop Control

Position controllers can stabilize a quadrotor about a fixed position or track a time-varying position (i.e., trajectory). A position or a trajectory are the inputs to a position controller, which either computes the desired attitude or desired angular rate (or both). Such controllers are demonstrated in [25, 28, 29] in simulation. Fewer position controllers are demonstrated in experiments indoors, because obtaining position feedback without GPS is nontrivial [30]. Image-base visual servo controllers are developed in [31, 32] that use visual features to provide position stabilizing feedback. In [32], feedback from an onboard camera is used to generate attitude set points. External sensors (e.g., motion-capture systems [33]) provide position feedback in [1,11,20,22,34]. In [1], a trajectory planning method and linear position controller are presented. The trajectory planner treats obstacles as constraints and generates a collision-free trajectory. Aerodynamic effects, the collision-free trajectory, and vehicle acceleration constraints are used to generate admissible control inputs. The trajectory tracking method presented in [1] is demonstrated in experiments indoors using motion capture, and in experiments outdoors using GPS. In [22], a trajectory tracking controller is developed that exploits the differential flatness of quadrotor dynamics, which is a property that facilitates trajectory tracking on certain outputs of the system. This controller is used to demonstrate aggressive maneuvers in experiments indoors using motion-capture feedback. In [11, 20], the control approach from [22] is used on multiple quadrotors for formation flying and trajectory tracking. In [11], each quadrotor plans its trajectory based on its neighbors state and planned trajectory. In [20], lightweight (73 g) custom quadrotors are developed for formation flight using the control approach from [22]. The small size of the quadrotors improve stability and agility of the formation as a whole. The success of these methods is largely due to the decentralized motion-capture systems, but without accurate and frequent position feedback trajectory tracking controllers can perform poorly. On the other hand, accurate and frequent velocity feedback is more accessible using only onboard sensors, which is demonstrated in [35, 36].

In this thesis, we present a middle-loop velocity controller that outputs the thrust and the desired attitude, which, in turn, drives the inner-loop controller. First, we design the thrust input to track a vertical velocity command. Then, we design the desired pitch and roll angles (using the thrust augmented translational dynamics) to track horizontal velocity commands. We utilize the camera module presented in [36] and sensor fusion to obtain inertial acceleration and velocity feedback.

1.3 Outer-Loop Guidance

An autonomous quadrotor should be capable of moving from one position to another while avoiding obstacles. To accomplish this, [37–44] use vision-based sensing or light detection and ranging (LIDAR) sensors to gather spatial and obstacle information about the environment. In [38, 39], simultaneous localization and mapping

(SLAM) algorithms build a two- or three-dimensional map of the environment, which is used by a trajectory planner to generate a collision-free trajectory. In [37] Google's Project Tango device (i.e. phone with vision sensing camera [45]) carries out onboard localization, mapping, and control. The controller uses the Tango's localization estimate and sensor fusion of the Tango and IMU pose estimates to follow trajectories in three dimensions. This approach relies on an onboard computer and the Tango's processor to run the estimation and fusion algorithms at 100 Hz. Though SLAM and trajectory planning methods achieve autonomous flight, as demonstrated in [37, 39], they are computationally intensive and can require a heavy sensor payload. Potential field methods [40, 42, 43] require fewer computations and a lightweight sensor payload as compared to SLAM methods. In [40], a potential field method is presented that assigns a minimum potential to a destination and larger potentials to obstacles. The quadrotor generates a force along the negative gradient of the potential function, and thus moves towards the destination and away from obstacles. A novel version of this method, demonstrated in [42, 43], uses a body-fixed potential field, which is a potential field that rotates and translates with the quadrotor. This body-fixed potential field also generates a reaction force that pushes the vehicle away from obstacles, but removes the need to keep track of the potentials of various obstacles in the flight space.

We approach the problem of collision avoidance using a quadrotor-fixed potential field method modified from the method presented in [42]. The method we present assumes no *a priori* information about the positions or shape of the obstacles. We use onboard vision-based sensing to measure the positions of points on the obstacles. We assume the obstacles are stationary and store the obstacle positions, similarly to [44], in order to avoid objects that aren't in the vision sensor's field of view. If an obstacle is detected within the quadrotor-fixed potential field, then an avoidance velocity is generated that directs the quadrotor away from the obstacle. In addition, we provide a constant speed destination seeking method. Together, the potential field and destination seeking methods constitute a guidance method that generates velocity commands that drive the middle-loop controller.

1.4 Summary of Contributions

In this thesis, we present an attitude tracking controller based on the Euler angle parameterization of attitude, and nonlinear rigid-body rotational dynamics. The attitude controller guarantees local exponential stability of any attitude equilibria with pitch between $\pm 90^\circ$. Next, we present a velocity controller that outputs the thrust and the desired attitude, which, in turn, drives the inner-loop attitude controller. We develop a quadrotor-fixed potential field method using the method presented in [42] that generates an avoidance component for the velocity commands. We also present a constant speed destination seeking method that provides the destination seeking component of the velocity commands. Together, the collision avoidance and destination seeking methods generate the velocity commands that drive the middle-loop velocity controller.

In addition, we use off-the-shelf components to build a quadrotor to test the guidance and control methods in experiment. The guidance method (i.e, collision avoidance and destination seeking) uses an onboard vision-sensing camera to detect the positions of obstacles, and dead reckoning to estimate the position of the destination. The middle-loop controller uses an onboard IMU, an optical flow sensor, and estimation methods (e.g., Kalman filters and sensor fusion) to provide acceleration and velocity feedback. The inner-loop attitude controller uses the IMU to provide angular rate and attitude feedback. An onboard Linux-based microcontroller communicates with the hardware and sensor components, and implements the guidance, estimation, and control algorithms. In indoor experiments, we demonstrate that the guidance and control methods achieve autonomous collision avoidance and destination seeking.

Chapter 2 Quadrotor Equations of Motion

2.1 Quadrotor Kinematics

Let I be an inertial frame, that is, a frame in which Newton's second law is valid. The origin $o_{\rm I}$ of I is any convenient point on the Earth's surface, and the orthogonal unit vectors of I are $\hat{\imath}, \hat{\jmath}$, and $\hat{k}$. Let B be the body frame, which is fixed to the quadrotor at its center of mass $o_{\rm B}$, and has orthogonal unit vectors $\hat{b}_1, \hat{b}_2$, and $\hat{b}_3$. Figure 2.1 shows the inertial and body frames. The position of $o_{\rm B}$ relative to $o_{\rm I}$ is

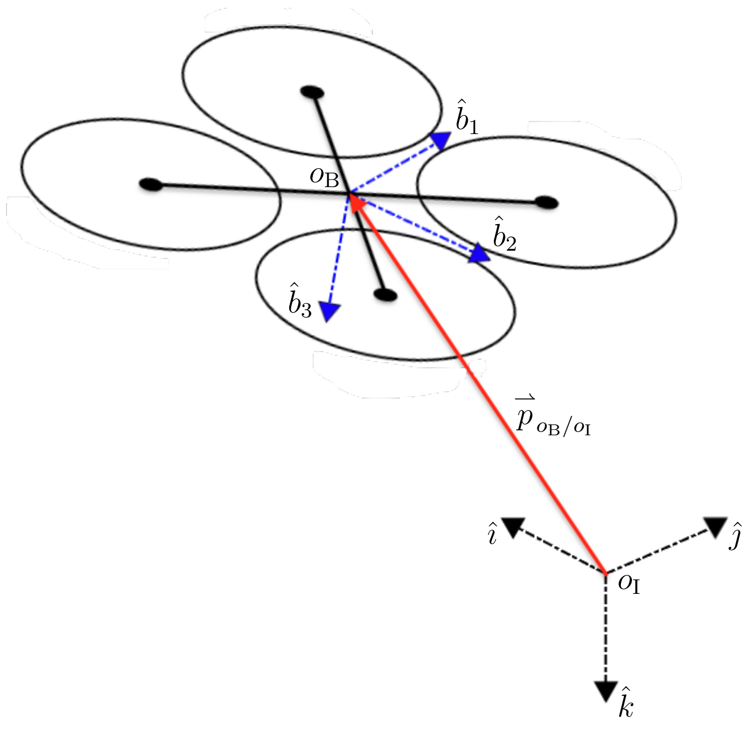

Figure 2.1: Inertial and Body Frames. The inertial frame is centered at $o_{\rm I}$, and the body frame is fixed to the quadrotor at its center of mass $o_{\rm B}$.

$$\vec{p}_{o_{\mathrm{B}}/o_{\mathrm{I}}} = x\hat{\imath} + y\hat{\jmath} + z\hat{k}.$$

Let ${}^{\mathrm{F}}(\dot{\cdot})$, and ${}^{\mathrm{F}}(\ddot{\cdot})$ denote the first and second time derivatives of a physical vector with respect to the frame F. Thus, the acceleration of o_{B} relative to o_{I} with respect to I is

$$\vec{a} \triangleq {}^{\mathrm{I}}(\ddot{\vec{p}}_{o_{\mathrm{B}}/o_{\mathrm{I}}}) = \ddot{x}\hat{\imath} + \ddot{y}\hat{\jmath} + \ddot{z}\hat{k}.$$

Let $\left[\,\cdot\,\right]_{\mathrm{F}}$ denote a physical vector or tensor resolved in the frame F. Therefore, $\vec{a}$ resolved in I is

$$a \triangleq \left[\vec{a}\right]_{\mathrm{I}} = \begin{bmatrix}\ddot{x} & \ddot{y} & \ddot{z}\end{bmatrix}^{\mathrm{T}}.$$

The angular velocity of B relative to I is $\vec{\omega}$, and we express $\vec{\omega}$ resolved in B as

$$\omega \triangleq \left[\vec{\omega}\right]_{\mathrm{B}} = \begin{bmatrix}p & q & r\end{bmatrix}^{\mathrm{T}}.$$

Let ψ, θ, and ϕ be Euler angles defined by a 3-2-1 rotation sequence as given in [46]. The orientation matrix of B relative to I is

$$\mathcal{O} \triangleq \begin{bmatrix} \mathrm{s}_\theta \mathrm{c}_\psi & \mathrm{c}_\theta \mathrm{s}_\psi & -\mathrm{s}_\theta \\ \mathrm{s}_\phi \mathrm{s}_\theta \mathrm{c}_\psi - \mathrm{c}_\phi \mathrm{s}_\psi & \mathrm{s}_\phi \mathrm{s}_\theta \mathrm{s}_\psi + \mathrm{c}_\phi \mathrm{c}_\psi & \mathrm{s}_\phi \mathrm{c}_\theta \\ \mathrm{c}_\phi \mathrm{s}_\theta \mathrm{c}_\psi + \mathrm{s}_\phi \mathrm{s}_\psi & \mathrm{c}_\phi \mathrm{s}_\theta \mathrm{s}_\psi - \mathrm{s}_\phi \mathrm{c}_\psi & \mathrm{c}_\phi \mathrm{c}_\theta \end{bmatrix}, \tag{2.1}$$

where for all $\gamma \in \mathbb{R}$, $\mathrm{c}_\gamma \triangleq \cos\gamma$ and $\mathrm{s}_\gamma \triangleq \sin\gamma$. Note that $\mathcal{O}^{-1} = \mathcal{O}^{\mathrm{T}}$ is the orientation matrix of I relative to B.

To represent ω in terms of the Euler angles ϕ, θ, and ψ and Euler angle rates $\dot{\phi}$, $\dot{\theta}$, and $\dot{\psi}$, we express each Euler angle rate as an angular velocity about the body frame axes, and sum these angular velocities. See [47] for more information. Thus,

$$\omega = \begin{bmatrix} \dot{\phi} \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 1 & 0 & 0 \\ 0 & \mathrm{c}_\phi & \mathrm{s}_\phi \\ 0 & -\mathrm{s}_\phi & \mathrm{c}_\phi \end{bmatrix} \begin{bmatrix} 0 \\ \dot{\theta} \\ 0 \end{bmatrix} + \begin{bmatrix} 1 & 0 & 0 \\ 0 & \mathrm{c}_\phi & \mathrm{s}_\phi \\ 0 & -\mathrm{s}_\phi & \mathrm{c}_\phi \end{bmatrix} \begin{bmatrix} \mathrm{c}_\theta & 0 & -\mathrm{s}_\theta \\ 0 & 1 & 0 \\ \mathrm{s}_\theta & 0 & \mathrm{c}_\theta \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ \dot{\psi} \end{bmatrix},$$

or equivalently,

$$\omega = Q \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix}, \tag{2.2}$$

where

$$Q \triangleq \begin{bmatrix} 1 & 0 & -\mathrm{s}_\theta \\ 0 & \mathrm{c}_\phi & \mathrm{s}_\phi \mathrm{c}_\theta \\ 0 & -\mathrm{s}_\phi & \mathrm{c}_\phi \mathrm{c}_\theta \end{bmatrix}.$$

Note that Q is singular if and only if $\theta = \pm\pi/2$. Thus, for all $\theta \neq \pm\pi/2$,

$$\begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = Q^{-1}\omega, \tag{2.3}$$

where

$$Q^{-1} = \begin{bmatrix} 1 & \mathrm{s}_\phi \mathrm{t}_\theta & \mathrm{c}_\phi \mathrm{t}_\theta \\ 0 & \mathrm{c}_\phi & -\mathrm{s}_\phi \\ 0 & \mathrm{s}_\phi/\mathrm{c}_\theta & \mathrm{c}_\phi/\mathrm{c}_\theta \end{bmatrix},$$

and for all $\gamma \in (-\pi/2, \pi/2)$, $\mathrm{t}_\gamma \triangleq \tan\gamma$. If $\theta = \pm\pi/2$, then Q is singular, and $\hat{b}_1$ and $\hat{k}$ are collinear. In this case ϕ and ψ are rotations about the same axis, that is, $\hat{b}_1 = \pm\hat{k}$. Thus, if $\theta = \pm\pi/2$, then the Euler angles ϕ and ψ correspond to a non-unique $\mathbb{O}$. We assume that for all $t > 0$, $\theta(t) \in (-\pi/2, \pi/2)$.

2.2 Quadrotor Dynamics

Let $m > 0$ be the mass of the quadrotor, and let $g > 0$ be the magnitude of acceleration due to gravity. Both m and g are assumed to be constant. Let $k > 0$ be the propeller thrust coefficient, and for $i = 1, ..., 4$, let $\omega_i > 0$ be the angular speed of the ith propeller, which are the four control inputs. The thrust vector is $-T\hat{b}_3$, where

$$T \triangleq k \sum_{i=1}^{4} \omega_i^2, \tag{2.4}$$

which is discussed further in [47]. Thus, Newton's second law yields

$$m\vec{a} = mg\hat{k} - T\hat{b}_3. \tag{2.5}$$

Note that (2.5) does not account for aerodynamic forces or flexible dynamics, which we assume to be negligible. Resolving (2.5) in I yields

$$m \begin{bmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{bmatrix} = \begin{bmatrix} -T(\mathrm{c}_\phi \mathrm{s}_\theta \mathrm{c}_\psi + \mathrm{s}_\phi \mathrm{s}_\psi) \\ -T(\mathrm{c}_\phi \mathrm{s}_\theta \mathrm{s}_\psi - \mathrm{s}_\phi \mathrm{c}_\psi) \\ mg - T\mathrm{c}_\phi \mathrm{c}_\theta \end{bmatrix},$$

or equivalently,

$$\ddot{x} = -\frac{T}{m}(\mathrm{c}_\phi \mathrm{s}_\theta \mathrm{c}_\psi + \mathrm{s}_\phi \mathrm{s}_\psi), \tag{2.6}$$

$$\ddot{y} = -\frac{T}{m}(\mathrm{c}_\phi \mathrm{s}_\theta \mathrm{s}_\psi - \mathrm{s}_\phi \mathrm{c}_\psi), \tag{2.7}$$

$$\ddot{z} = g - \frac{T}{m}\mathrm{c}_\phi \mathrm{c}_\theta. \tag{2.8}$$

Let $\vec{I}_\mathrm{q}$ be the quadrotor's physical inertia matrix relative to o_B. We assume that

$\hat{b}_1, \hat{b}_2$, and $\hat{b}_3$ are principle axes, which implies that

$$I_\mathrm{q} \triangleq \left[\overrightarrow{I_\mathrm{q}}\right]_\mathrm{B} = \begin{bmatrix} I_\mathrm{xx} & 0 & 0 \\ 0 & I_\mathrm{yy} & 0 \\ 0 & 0 & I_\mathrm{zz} \end{bmatrix},$$

where $I_\mathrm{xx}, I_\mathrm{yy}$, and I_zz are the moments of inertia.

Each propeller lies in the $\hat{b}_1$–$\hat{b}_2$ plane and is a distance l from $\hat{b}_1$ and a distance l from $\hat{b}_2$. Therefore, the thrust $-k\omega_i^2\hat{b}_3$ produced by the ith propeller creates a moment about $\hat{b}_1$ and $\hat{b}_2$.

As described in [47], we model the moment due to drag on the ith propeller as $(-1)^{i+1}b\omega_i^2\hat{b}_3$, where $b > 0$ is the propeller drag coefficient. Thus, the moment from propeller thrust and drag is

$$\overrightarrow{\tau} = \tau_{b_1}\hat{b}_1 + \tau_{b_2}\hat{b}_2 + \tau_{b_3}\hat{b}_3,$$

where

$$\tau_{b_1} \triangleq kl(\omega_1^2 - \omega_2^2 - \omega_3^2 + \omega_4^2), \tag{2.9}$$

$$\tau_{b_2} \triangleq kl(\omega_1^2 + \omega_2^2 - \omega_3^2 - \omega_4^2), \tag{2.10}$$

$$\tau_{b_3} \triangleq b(\omega_1^2 - \omega_2^2 + \omega_3^2 - \omega_4^2). \tag{2.11}$$

Let I_p be the moment of inertia of each propeller about its axis of rotation. Therefore, the total angular momentum is

$$\overrightarrow{H} = \overrightarrow{I_\mathrm{q}}\overrightarrow{\omega} + \overrightarrow{H}_\mathrm{p}, \tag{2.12}$$

where $\overrightarrow{H}_\mathrm{p} = I_\mathrm{p}(-\omega_1 + \omega_2 - \omega_3 + \omega_4)\hat{b}_3$ is the angular momentum of the propellers. Figure 2.2 shows the angular momentum vectors. Differentiating (2.12) with respect

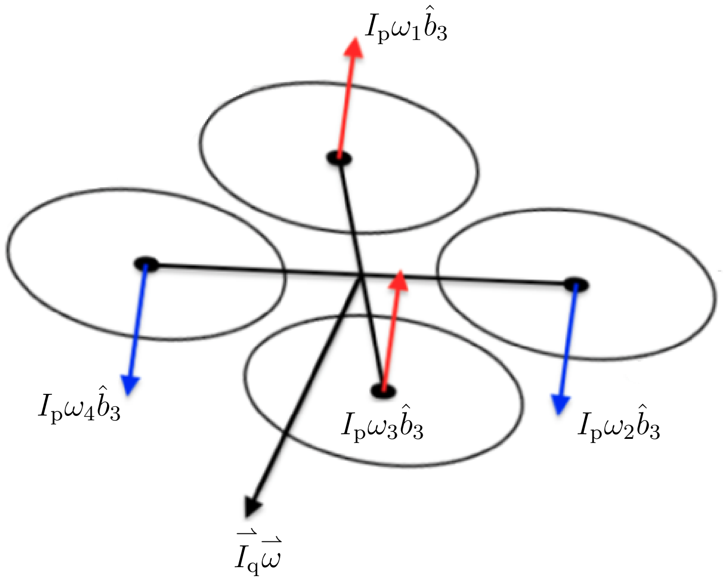
$I_\mathrm{p}\omega_1\hat{b}_3$
$I_\mathrm{p}\omega_4\hat{b}_3$
$I_\mathrm{p}\omega_3\hat{b}_3$
$I_\mathrm{p}\omega_2\hat{b}_3$
$\vec{I}_\mathrm{q}\vec{\omega}$

Figure 2.2: Angular Momentum. The total angular momentum $\vec{H}$ is the sum of the propellers' angular momentum (blue and red) and the quadrotor body's angular momentum (black).

to I yields

$$\begin{aligned} {}^{\mathrm{I}}(\dot{\vec{H}}) &= \vec{I}_\mathrm{q}{}^{\mathrm{I}}(\dot{\vec{\omega}}) + {}^{\mathrm{I}}(\dot{\vec{H}}_\mathrm{p}) \\ &= \vec{I}_\mathrm{q}{}^{\mathrm{B}}(\dot{\vec{\omega}}) + \vec{\omega} \times \vec{I}_\mathrm{q}\vec{\omega} + {}^{\mathrm{B}}(\dot{\vec{H}}_\mathrm{p}) + \vec{\omega} \times \vec{H}_\mathrm{p}. \end{aligned} \tag{2.13}$$

Thus, Euler's second law yields

$$\vec{I}_\mathrm{q}{}^{\mathrm{B}}(\dot{\vec{\omega}}) + \vec{\omega} \times \vec{I}_\mathrm{q}\vec{\omega} + {}^{\mathrm{B}}(\dot{\vec{H}}_\mathrm{p}) + \vec{\omega} \times \vec{H}_\mathrm{p} = \vec{\tau}, \tag{2.14}$$

and resolving (2.14) in B yields

$$I_\mathrm{q}\dot{\omega} + \Omega I_\mathrm{q}\omega + \left[{}^{\mathrm{B}}(\dot{\vec{H}}_\mathrm{p})\right]_\mathrm{B} + \Omega\left[\vec{H}_\mathrm{p}\right]_\mathrm{B} = \left[\vec{\tau}\right]_\mathrm{B},$$

where

$$\Omega \triangleq \begin{bmatrix} 0 & -r & q \\ r & 0 & -p \\ -q & p & 0 \end{bmatrix}.$$

Therefore,

$$\dot{\omega} = I_{\mathrm{q}}^{-1}\Big(\big[\vec{\tau}\big]_{\mathrm{B}} - \Omega I_{\mathrm{q}}\omega - \big[\dot{\vec{H}}_{\mathrm{p}}\big]_{\mathrm{B}} - \Omega\big[\vec{H}_{\mathrm{p}}\big]_{\mathrm{B}} \Big),$$

or equivalently,

$$\begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} = \begin{bmatrix} (\tau_{b_1} - qr(I_{zz} - I_{yy}) - qI_{\mathrm{p}}(-\omega_1 + \omega_2 - \omega_3 + \omega_4))/I_{xx} \\ (\tau_{b_2} + pr(I_{zz} - I_{xx}) + pI_{\mathrm{p}}(-\omega_1 + \omega_2 - \omega_3 + \omega_4))/I_{yy} \\ (\tau_{b_3} - pq(I_{yy} - I_{xx}) - I_{\mathrm{p}}(-\dot{\omega}_1 + \dot{\omega}_2 - \dot{\omega}_3 + \dot{\omega}_4))/I_{zz} \end{bmatrix}. \tag{2.15}$$

Chapter 3 Autonomous Collision Avoidance and Destination Seeking Control

In this chapter, we present a multi-loop guidance and control method for autonomous collision avoidance and destination seeking. The multi-loop guidance and control architecture is shown in Figure 3.1.

First, we present an inner-loop attitude controller that uses the desired Euler angles $\phi_{\rm d}, \theta_{\rm d}$, and $\psi_{\rm d}$, and the feedback ϕ, θ, ψ, p, q, and r to determine the control torques τ_{b_1}, τ_{b_2}, and τ_{b_3}. Next, we present a middle-loop velocity controller that uses the desired velocities $u_{\rm d}, v_{\rm d}$, and $w_{\rm d}$, and the feedback u, v, w, ϕ, θ, and ψ to determine the desired Euler angles $\phi_{\rm d}, \theta_{\rm d}$, and $\psi_{\rm d}$, which drive the inner-loop attitude controller. The velocity controller also determines the thrust control T. Finally, the outer-loop guidance method uses the position $p_{\rm g}$ of the quadrotor's desired destination, and the positions $p_1, ..., p_n$ of obstacles, and the feedback x, y, and z to generate the velocity commands $u_{\rm c}, v_{\rm c}$, and $w_{\rm c}$. These velocity commands are passed through low-pass filters $G_{\rm c}$, which output the desired velocities $u_{\rm d}, v_{\rm d}$, and $w_{\rm d}$, which drive the middle-loop velocity controller. We do not assume that the positions x and y are measured; however, we do assume that u and v are measured. Thus, we integrate the velocities u and v to estimate the positions x and y.

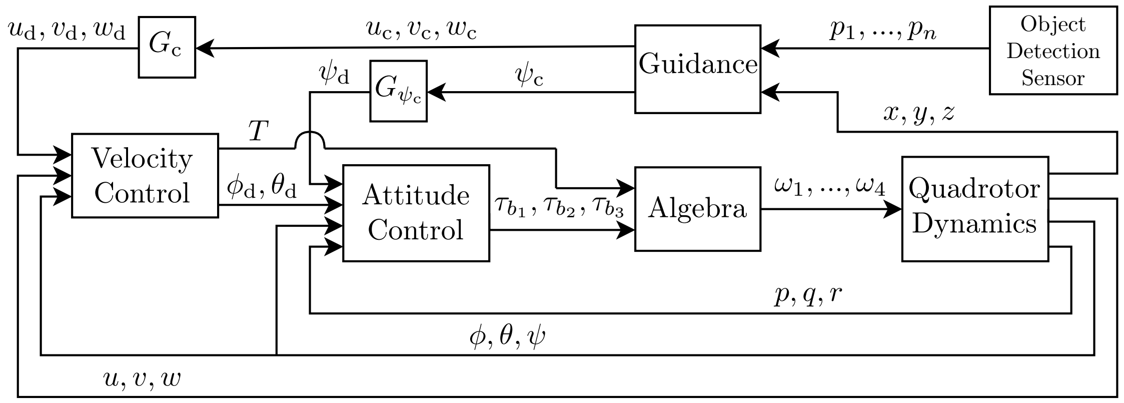

Figure 3.1: Multi-Loop Control Architecture. An onboard sensor (e.g., vision-based sensing) provides positions $p_1, ..., p_n$ of obstacles to the guidance algorithm, which outputs the velocity commands $u_\mathrm{c}, v_\mathrm{c}, w_\mathrm{c}$ and a yaw angle command ψ_c. The velocity commands are passed through low pass filters G_c, which output desired velocities $u_\mathrm{d}, v_\mathrm{d}$, and w_d, which drive the velocity controller. The yaw angle command ψ_c is passed through a low-pass filter G_{ψ_c}, which outputs the desired yaw angle ψ_d. The velocity controller outputs the desired roll and pitch angles ϕ_d and θ_d, which, along with the desired yaw angle ψ_d, drive the attitude controller. The velocity controller also outputs the thrust control T. The attitude controller outputs the control torques τ_{b_1}, τ_{b_2}, and τ_{b_3}, which, in addition to the thrust control T, are used to algebraically compute the propeller speeds $\omega_1, ..., \omega_4$. The positions x, y, and z are used as feedback for destination seeking (included in guidance) and the velocities u, v, and w, attitude ϕ, θ, and ψ, and angular rate p, q, and r are used as feedback to the velocity and attitude controllers.

3.1 Inner-Loop Attitude Control

Let $\phi_\mathrm{d} : [0, \infty) \to (-\pi, \pi]$, $\theta_\mathrm{d} : [0, \infty) \to (-\pi/2, \pi/2)$, and $\psi_\mathrm{d} : [0, \infty) \to (-\pi, \pi]$ be desired Euler angles, and define the roll, pitch, and yaw errors $e_\phi \triangleq \phi - \phi_\mathrm{d}$, $e_\theta \triangleq \theta - \theta_\mathrm{d}$, and $e_\psi \triangleq \psi - \psi_\mathrm{d}$. Define the desired angular velocity

$$\omega_\mathrm{d} \triangleq Q \begin{bmatrix} \dot{\phi}_\mathrm{d} - k_\phi e_\phi \\ \dot{\theta}_\mathrm{d} - k_\theta e_\theta \\ \dot{\psi}_\mathrm{d} - k_\psi e_\psi \end{bmatrix}, \tag{3.1}$$

where $k_\phi > 0$, $k_\theta > 0$, and $k_\psi > 0$ are the proportional gains associated with e_ϕ, e_θ, and e_ψ.

The following result shows that if the angular velocity ω is equivalent to the desired angular velocity ω_{d}, then e_ϕ, e_θ, and e_ψ satisfy unforced asymptotically stable linear time-invariant differential equations.

Proposition 1. *Consider* (2.3)*, and* ω_{d} *given by* (3.1)*. Assume* $\omega = \omega_{\mathrm{d}}$*, and for all* $t \geq 0$*,* $\theta(t) \neq \pm\pi/2$*. Then,* e_ϕ*,* e_θ*, and* e_ψ *satisfy*

$$\dot{e}_\phi + k_\phi e_\phi = 0, \quad \dot{e}_\theta + k_\theta e_\theta = 0, \quad \dot{e}_\psi + k_\psi e_\psi = 0.$$

Next, define the angular velocity error $e_\omega \triangleq \omega - \omega_{\mathrm{d}}$, and consider the control torques

$$\begin{bmatrix} \tau_{b_1} \\ \tau_{b_2} \\ \tau_{b_3} \end{bmatrix} = I_{\mathrm{q}}(\dot{\omega}_{\mathrm{d}} - K_\omega e_\omega) - I_{\mathrm{q}} Q^{-\mathrm{T}} \begin{bmatrix} e_\phi \\ e_\theta \\ e_\psi \end{bmatrix} + \begin{bmatrix} qr(I_{zz} - I_{yy}) + qI_{\mathrm{p}}(-\omega_1 + \omega_2 - \omega_3 + \omega_4) \\ -pr(I_{zz} - I_{xx}) - pI_{\mathrm{p}}(-\omega_1 + \omega_2 - \omega_3 + \omega_4) \\ pq(I_{yy} - I_{xx}) + I_{\mathrm{p}}(-\dot{\omega}_1 + \dot{\omega}_2 - \dot{\omega}_3 + \dot{\omega}_4) \end{bmatrix}, \tag{3.2}$$

where $K_\omega \in \mathbb{R}^{3\times 3}$ is positive definite.

To analyze the stability properties of the closed-loop attitude dynamics, consider the positive-definite Lyapunov candidate $V : (-\pi,\pi)\times(-\pi,\pi)\times(-\pi,\pi)\times\mathbb{R}^3 \to [0,\infty)$ defined by

$$V(e_\phi, e_\theta, e_\psi, e_\omega) \triangleq \frac{1}{2}(e_\phi^2 + e_\theta^2 + e_\psi^2 + e_\omega^{\mathrm{T}} e_\omega).$$

The following result provides the stability properties of (2.3), (2.15), (3.1), and (3.2).

Theorem 1. *Let* $\phi_{\mathrm{e}}, \psi_{\mathrm{e}} \in (-\pi, \pi]$ *and* $\theta_{\mathrm{e}} \in (-\pi/2, \pi/2)$*, and let* $\phi_{\mathrm{d}}(t) \equiv \phi_{\mathrm{e}}$*,* $\theta_{\mathrm{d}}(t) \equiv \theta_{\mathrm{e}}$*, and* $\psi_{\mathrm{d}}(t) \equiv \psi_{\mathrm{e}}$*. Then,* $(\phi, \theta, \psi, \omega) \equiv (\phi_{\mathrm{e}}, \theta_{\mathrm{e}}, \psi_{\mathrm{e}}, 0)$ *is a locally exponentially stable equilibrium of the closed-loop system* (2.3), (2.15), (3.1), *and* (3.2)*. Furthermore,* *define*

$$\omega_{\rm e}(\phi,\theta,\psi) \triangleq Q \begin{bmatrix} -k_\phi(\phi-\phi_{\rm e}) \\ -k_\theta(\theta-\theta_{\rm e}) \\ -k_\psi(\psi-\psi_{\rm e}) \end{bmatrix},$$

and

$$R_{\rm A} \triangleq \Big\{(\phi_0,\theta_0,\psi_0,\omega_0)\in\mathbb{R}^6 : V(\phi_0-\phi_{\rm e},\theta_0-\theta_{\rm e},\psi_0-\psi_{\rm e},\omega_0-\omega_{\rm e}(\phi_0,\theta_0,\psi_0)) < \frac{1}{2}\min\Big\{\Big(\frac{\pi}{2}-\theta_{\rm e}\Big)^2,\Big(\frac{\pi}{2}+\theta_{\rm e}\Big)^2\Big\}\Big\}.$$

Then, for all initial conditions $(\phi(0),\theta(0),\psi(0),\omega(0))\in R_{\rm A}$, $\lim_{t\to\infty}\phi(t)=\phi_{\rm e}$, $\lim_{t\to\infty}\theta(t)=\theta_{\rm e}$, $\lim_{t\to\infty}\psi(t)=\psi_{\rm e}$, *and* $\lim_{t\to\infty}\omega(t)=0$.

Proof. Define $e_\Phi \triangleq [e_\phi \quad e_\theta \quad e_\psi]^{\rm T}$ and $K_\Phi \triangleq {\rm diag}(k_\phi,k_\theta,k_\psi)$, and it follows from (2.3) that

$$\dot e_\Phi = A(e_\phi,e_\theta)e_\omega - K_\Phi e_\Phi,$$

where

$$A(e_\phi,e_\theta) \triangleq \begin{bmatrix} 1 & {\rm s}_{e_\phi+\phi_{\rm e}}{\rm t}_{e_\theta+\theta_{\rm e}} & {\rm c}_{e_\phi+\phi_{\rm e}}{\rm t}_{e_\theta+\theta_{\rm e}} \\ 0 & {\rm c}_{e_\phi+\phi_{\rm e}} & -{\rm s}_{e_\phi+\phi_{\rm e}} \\ 0 & {\rm s}_{e_\phi+\phi_{\rm e}}/{\rm c}_{e_\theta+\theta_{\rm e}} & {\rm c}_{e_\phi+\phi_{\rm e}}/{\rm c}_{e_\theta+\theta_{\rm e}} \end{bmatrix}.$$

Substituting (3.2) into (2.15) yields

$$\dot e_\omega = -K_\omega e_\omega - A^{\rm T}(e_\phi,e_\theta)e_\Phi.$$

Next, define

$$D \triangleq \big\{(e_\phi,e_\theta,e_\psi,e_\omega)\in\mathbb{R}^6 : e_\phi,e_\theta,e_\psi\in(-\pi,\pi),\ e_\theta+\theta_{\rm d}\in(-\pi/2,\pi/2),\ e_\omega\in\mathbb{R}^3\big\},$$

and it follows that for all $(e_\phi,e_\theta,e_\psi,e_\omega)\in D$, the derivative of V along the trajectories of (2.3), (2.15), (3.1), and (3.2) is

$$\begin{aligned}
\dot{V}(e_\phi, e_\theta, e_\psi, e_\omega) &\triangleq \frac{\partial V}{\partial e_\Phi}\dot{e}_\Phi + \frac{\partial V}{\partial e_\omega}\dot{e}_\omega \\
&= e_\Phi^{\mathrm{T}}(A(e_\phi, e_\theta)e_\omega - K_\Phi e_\Phi) + e_\omega^{\mathrm{T}}(-K_\omega e_\omega - A^{\mathrm{T}}(e_\phi, e_\theta)e_\Phi) \\
&= -e_\Phi^{\mathrm{T}} K_\Phi e_\Phi - e_\omega^{\mathrm{T}} K_\omega e_\omega \\
&\leq -\min\{k_\phi, k_\theta, k_\psi\} e_\Phi^{\mathrm{T}} e_\Phi - \lambda_{\min}(K_\omega) e_\omega^{\mathrm{T}} e_\omega \\
&\leq -\min\{k_\phi, k_\theta, k_\psi, \lambda_{\min}(K_\omega)\}(e_\Phi^{\mathrm{T}} e_\Phi + e_\omega^{\mathrm{T}} e_\omega) \\
&= -c_1 V(e_\phi, e_\theta, e_\psi, e_\omega), \qquad (3.3)
\end{aligned}$$

where $c_1 \triangleq 2\min\{k_\phi, k_\theta, k_\psi, \lambda_{\min}(K_\omega)\}$ is positive. Thus, $(\phi, \theta, \psi, \omega) \equiv (\phi_\mathrm{e}, \theta_\mathrm{e}, \psi_\mathrm{e}, 0)$ is a locally exponentially stable equilibrium.

To show the final statement of the result, define

$$R'_\mathrm{A} \triangleq \left\{(e_\phi, e_\theta, e_\psi, e_\omega) \in \mathbb{R}^6 : V(e_\phi, e_\theta, e_\psi, e_\omega) < \frac{1}{2}\min\left\{\left(\frac{\pi}{2} - \theta_\mathrm{e}\right)^2, \left(\frac{\pi}{2} + \theta_\mathrm{e}\right)^2\right\}\right\}.$$

Since $R'_\mathrm{A} \subseteq D$ and for all $(e_\phi, e_\theta, e_\psi, e_\omega) \in D \setminus \{0\}$, $\dot{V}(e_\phi, e_\theta, e_\psi, e_\omega) < 0$, it follows that R'_A is invariant with respect to (2.3), (2.15), (3.1), and (3.2).

Let $(\phi(0), \theta(0), \psi(0), \omega(0)) \in R_\mathrm{A}$, and it follows that $(e_\phi(0), e_\theta(0), e_\psi(0), e_\omega(0)) \in R'_\mathrm{A}$. Since R'_A is invariant, it follows that for all $t \geq 0$, $(e_\phi(t), e_\theta(t), e_\psi(t), e_\omega(t)) \in R'_\mathrm{A} \subseteq D$. Thus, it follows from (3.3) that

$$\begin{aligned}
0 &\leq \int_0^\infty V(e_\phi(t), e_\theta(t), e_\psi(t), e_\omega(t))\mathrm{d}t \\
&\leq -\frac{1}{c_1}\int_0^\infty \dot{V}(e_\phi(t), e_\theta(t), e_\psi(t), e_\omega(t))\mathrm{d}t \\
&= \frac{1}{c_1}\left[V(e_\phi(0), e_\theta(0), e_\psi(0), e_\omega(0)) - \lim_{t\to\infty} V(e_\phi(t), e_\theta(t), e_\psi(t), e_\omega(t))\right] \\
&\leq \frac{1}{c_1} V(e_\phi(0), e_\theta(0), e_\psi(0), e_\omega(0)),
\end{aligned}$$

which implies that $\int_0^\infty V(e_\phi(t), e_\theta(t), e_\psi(t), e_\omega(t))\mathrm{d}t$ exists. Since, in addition,

$\dot{V}(e_\phi(t), e_\theta(t), e_\psi(t), e_\omega(t))$ is bounded, it follows from Barbalat's lemma that $\lim_{t\to\infty} V(e_\phi(t), e_\theta(t), e_\psi(t), e_\omega(t)) = 0$, which confirms the last statement of the result. $\square$

Note that the control torques (3.2) require ϕ_d, θ_d, and ψ_d and their first and second derivatives for implementation. We generate ψ_d,$\dot{\psi}_\mathrm{d}$, and $\ddot{\psi}_\mathrm{d}$ by passing an external yaw command $\psi_\mathrm{c} : [0,\infty) \to (-\pi, \pi]$ through a third-order low-pass filter with unity gain at dc. More specifically, let ψ_d be the output of the asymptotically stable transfer function

$$G_{\psi_\mathrm{d}}(s) \triangleq \frac{\beta_0}{s^3 + \beta_2 s^2 + \beta_1 s + \beta_0},$$

where the input is ψ_c. Therefore, ψ_d satisfies

$$\begin{bmatrix} \dddot{\psi}_\mathrm{d} \\ \ddot{\psi}_\mathrm{d} \\ \dot{\psi}_\mathrm{d} \end{bmatrix} = \begin{bmatrix} -\beta_2 & -\beta_1 & -\beta_0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} \ddot{\psi}_\mathrm{d} \\ \dot{\psi}_\mathrm{d} \\ \psi_\mathrm{d} \end{bmatrix} + \begin{bmatrix} \beta_0 \\ 0 \\ 0 \end{bmatrix} \psi_\mathrm{c},$$

which can be solved to obtain ψ_d, $\dot{\psi}_\mathrm{d}$, and $\ddot{\psi}_\mathrm{d}$.

In contrast, we compute ϕ_d and θ_d, and their derivatives $\dot{\phi}_\mathrm{d}$, $\dot{\theta}_\mathrm{d}$, $\ddot{\phi}_\mathrm{d}$, and $\ddot{\theta}_\mathrm{d}$ (whose analytic expressions are provided in Appendix A) using a velocity controller, which is introduced in the next section.

3.2 Middle-Loop Velocity Control

Define the velocities $u \triangleq \dot{x}$, $v \triangleq \dot{y}$, and $w \triangleq \dot{z}$. Let $u_\mathrm{d}, v_\mathrm{d}, w_\mathrm{d} : [0,\infty) \to \mathbb{R}$ be the $\hat{\imath}$-, $\hat{\jmath}$-, and $\hat{k}$-direction desired velocities, which are assumed to be three-times continuously differentiable, and define the velocity errors $e_u \triangleq u - u_\mathrm{d}$, $e_v \triangleq v - v_\mathrm{d}$, and $e_w \triangleq w - w_\mathrm{d}$.

Consider the thrust control

$$T = \frac{m(g - \dot{w}_\mathrm{d} + k_w e_w)}{\mathrm{c}_\phi \mathrm{c}_\theta}, \tag{3.4}$$

where $k_w > 0$ is the proportional gain associated with e_w. Substituting (3.4) into (2.8) yields $\dot{e}_w + k_w e_w = 0$, which implies that e_w satisfies an unforced asymptotically stable linear time-invariant differential equation.

Define the desired pitch and desired roll

$$\theta_\mathrm{d} \triangleq \tan^{-1}\left[\frac{(\dot{u}_\mathrm{d} - k_u e_u)\mathrm{c}_\psi + (\dot{v}_\mathrm{d} - k_v e_v)\mathrm{s}_\psi}{\dot{w}_\mathrm{d} - k_w e_w - g}\right], \tag{3.5}$$

$$\phi_\mathrm{d} \triangleq \tan^{-1}\left[\frac{(\dot{u}_\mathrm{d} - k_u e_u)\mathrm{c}_{\theta_\mathrm{d}}\mathrm{s}_\psi - (\dot{v}_\mathrm{d} - k_v e_v)\mathrm{c}_{\theta_\mathrm{d}}\mathrm{c}_\psi}{\dot{w}_\mathrm{d} - k_w e_w - g}\right], \tag{3.6}$$

where $k_u > 0$ and $k_v > 0$ are the proportional gains associated e_u and e_v. The following result shows that if the pitch θ and roll ϕ are equivalent to the desired pitch θ_d and desired roll ϕ_d, then e_u and e_v satisfy unforced asymptotically stable linear time-invariant differential equations.

Proposition 2. *Consider* (2.6) *and* (2.7)*, where* T *is given by* (3.4)*, and consider* θ_d *and* ϕ_d *given by (3.5) and (3.6). Assume* $\theta = \theta_\mathrm{d}$*,* $\phi = \phi_\mathrm{d}$*, and for all* $t \geq 0$*,* $\dot{w}_\mathrm{d}(t) - k_w e_w(t) \neq g$*. Then,* e_u *and* e_v *satisfy*

$$\dot{e}_u + k_u e_u = 0, \quad \dot{e}_v + k_v e_v = 0.$$

The control torques and thrust control (3.2) and (3.4), where the desired roll and pitch are given by (3.5) and (3.6), require the desired velocities u_d, v_d, w_d and their first, second, and third derivatives for implementation. We generate the desired velocities and their derivatives by passing the external velocity commands $u_\mathrm{c} : [0, \infty) \to \mathbb{R}$, $v_\mathrm{c} : [0, \infty) \to \mathbb{R}$, and $w_\mathrm{c} : [0, \infty) \to \mathbb{R}$ through fourth-order low-pass filters with unity gain at dc. More specifically, let u_d be the output of the asymptotically stable transfer function

$$G_\mathrm{d}(s) \triangleq \frac{\alpha_0}{s^4+\alpha_3 s^3+\alpha_2 s^2+\alpha_1 s+\alpha_0},$$

where the input is u_c. Therefore, u_d satisfies

$$\begin{bmatrix} \ddddot{u}_\mathrm{d} \\ \dddot{u}_\mathrm{d} \\ \ddot{u}_\mathrm{d} \\ \dot{u}_\mathrm{d} \end{bmatrix} = \begin{bmatrix} -\alpha_3 & -\alpha_2 & -\alpha_1 & -\alpha_0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} \dddot{u}_\mathrm{d} \\ \ddot{u}_\mathrm{d} \\ \dot{u}_\mathrm{d} \\ u_\mathrm{d} \end{bmatrix} + \begin{bmatrix} \alpha_0 \\ 0 \\ 0 \\ 0 \end{bmatrix} u_\mathrm{c},$$

which can be solved to obtain u_d, $\dot{u}_\mathrm{d}$, $\ddot{u}_\mathrm{d}$, and $\dddot{u}_\mathrm{d}$. Similarly, we let v_d and w_d be the output of G_d with inputs v_c and w_c, respectively.

Recall that the quadrotor's physical controls are the propeller speeds ω_1, ω_2, ω_3, and ω_4. In order to implement the inner-loop and middle-loop controllers (3.1), (3.2), (3.4), (3.5), and (3.6), we determine the propeller speeds required to achieve the control torques and thrust control given by (3.2) and (3.4). Using (2.4) and (2.9)–(2.11), we obtain

$$\begin{bmatrix} T \\ \tau_{b_1} \\ \tau_{b_2} \\ \tau_{b_3} \end{bmatrix} = \begin{bmatrix} k & k & k & k \\ kl & -kl & -kl & kl \\ kl & kl & -kl & -kl \\ b & -b & b & -b \end{bmatrix} \begin{bmatrix} \omega_1^2 \\ \omega_2^2 \\ \omega_3^2 \\ \omega_4^2 \end{bmatrix}. \tag{3.7}$$

Since $k, b, l > 0$, it follows that the $4{\times}4$ matrix in (3.7) is invertible. Therefore,

$$\begin{bmatrix} \omega_1^2 \\ \omega_2^2 \\ \omega_3^2 \\ \omega_4^2 \end{bmatrix} = \begin{bmatrix} \frac{1}{4k} & \frac{1}{4kl} & \frac{1}{4kl} & \frac{1}{4b} \\ \frac{1}{4k} & \frac{-1}{4kl} & \frac{1}{4kl} & \frac{-1}{4b} \\ \frac{1}{4k} & \frac{-1}{4kl} & \frac{-1}{4kl} & \frac{1}{4b} \\ \frac{1}{4k} & \frac{1}{4kl} & \frac{-1}{4kl} & \frac{-1}{4b} \end{bmatrix} \begin{bmatrix} T \\ \tau_{b_1} \\ \tau_{b_2} \\ \tau_{b_3} \end{bmatrix}. \tag{3.8}$$

Note that (3.2) implies that τ_{b_1} and τ_{b_2} are linear functions of ω_1, ω_2, ω_3, and ω_4. Thus, (3.8) is a set of four coupled quadratic equations in ω_1, ω_2, ω_3, and ω_4. Numerical approaches (e.g., Newton's method) can be used to solve these equations approximately. In this work, we use (3.8) to solve for ω_1, ω_2, ω_3, and ω_4, where τ_{b_1} and τ_{b_2} are approximated by using the one-time-step-delayed value of ω_1, ω_2, ω_3, and ω_4, instead of the current values. For typical flight maneuvers, numerical simulations suggest that this approximation has a small impact because the terms $-qI_{\mathrm{p}}(-\omega_1 + \omega_2 - \omega_3 + \omega_4)$ and $pI_{\mathrm{p}}(-\omega_1 + \omega_2 - \omega_3 + \omega_4)$ are small relative to the remaining terms in (3.2).

We also note that (3.2) implies that τ_{b_3} is a linear function of $-\dot{\omega}_1 + \dot{\omega}_2 - \dot{\omega}_3 + \dot{\omega}_4$, which cannot be implemented with a causal control law. Thus, we approximate τ_{b_3} by using backward Euler approximations of $\dot{\omega}_1$, $\dot{\omega}_2$, $\dot{\omega}_3$, and $\dot{\omega}_4$, which are computed from the one- and two-time-step-delayed values of ω_1, ω_2, ω_3, and ω_4. For typical flight maneuvers, numerical simulations suggest that this approximation has a small impact.

3.3 Outer-Loop Guidance for Collision Avoidance and Destination Seeking

In this section we present an outer-loop guidance algorithm for autonomous collision avoidance and destination seeking. This algorithm generates the velocity commands $u_{\rm c}$, $v_{\rm c}$, and $w_{\rm c}$, which are used by the middle-loop controller. We generate the obstacle avoidance component of the velocity commands using a modified version of the artificial potential field method given in [42].

For $i = 1, ..., n$, let $\vec{p}_i$ be the position relative to $o_{\rm B}$ of the ith point on an obstacle. Let $r_{\rm p} > 0$ be the radius of each propeller, and let $r_{\rm c} > r_{\rm p} + l$ be the collision radius, which is the minimum acceptable length of $\vec{p}_i$. In other words, a collision occurs if and only if $||p_i|| < r_{\rm c}$, where $p_i \triangleq [\vec{p}_i]_{\rm B}$.

To generate the obstacle avoidance component of the velocity commands, we define a body-fixed potential field in $\mathbb{R}^3$. Let $a_{\rm i}, b_{\rm i}, c_{\rm i} > r_{\rm c}$, and define the inner ellipsoid

$$E_{\rm i} \triangleq \left\{ (x_{\rm e}, y_{\rm e}, z_{\rm e}) \in \mathbb{R}^3 : \frac{x_{\rm e}^2}{a_{\rm i}^2} + \frac{y_{\rm e}^2}{b_{\rm i}^2} + \frac{z_{\rm e}^2}{c_{\rm i}^2} \le 1 \right\}.$$

Note that $p_i \in E_{\rm i}$ if and only if $\vec{p}_i$ lies inside a physical ellipsoid whose geometric center is $o_{\rm B}$ and whose semi-principle axes are parallel to $\hat{b}_1, \hat{b}_2$, and $\hat{b}_3$.

Next, let $a_{\rm o} > a_{\rm i}, b_{\rm o} > b_{\rm i}$, and $c_{\rm o} > c_{\rm i}$, and define the outer ellipsoid

$$E_{\rm o} \triangleq \left\{ (x_{\rm e}, y_{\rm e}, z_{\rm e}) \in \mathbb{R}^3 : \frac{x_{\rm e}^2}{a_{\rm o}^2} + \frac{y_{\rm e}^2}{b_{\rm o}^2} + \frac{z_{\rm e}^2}{c_{\rm o}^2} \le 1 \right\}.$$

Note that $p_i \in E_{\rm o}$ if and only if $\vec{p}_i$ lies on or inside a physical ellipsoid whose geometric center is $o_{\rm B}$ and whose semi-principle axes are parallel to $\hat{b}_1, \hat{b}_2$, and $\hat{b}_3$.

Let $U_{\max} > 0$ be the maximum potential, and consider $U : \mathbb{R}^3 \to [0, U_{\max}]$ defined

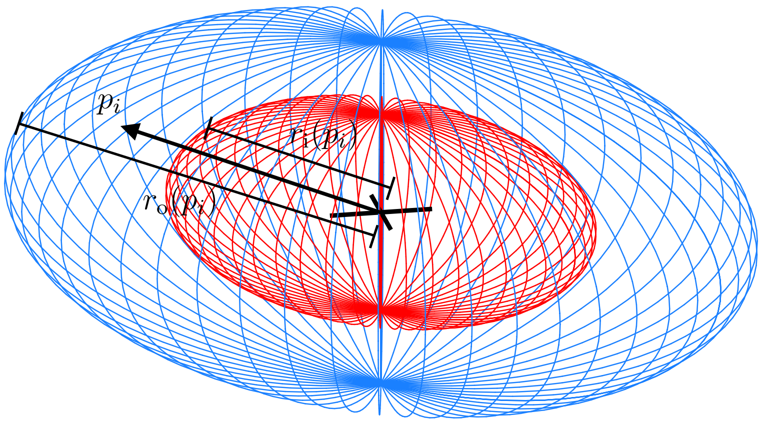

Figure 3.2: Quadrotor Fixed Potential Field. The inner (red) and outer (blue) physical ellipsoids are centered at o_B and have semi-principle axes that are parallel to $\hat{b}_1$, $\hat{b}_2$, and $\hat{b}_3$.

by

$$U(\sigma) \triangleq \begin{cases} 0, & \text{if } \sigma \in \mathbb{R}^3 \setminus E_o, \\ U_{\max}\left(1 - \dfrac{||\sigma|| - r_i(\sigma)}{r_o(\sigma) - r_i(\sigma)}\right), & \text{if } \sigma \in E_o \setminus E_i, \\ U_{\max}, & \text{if } \sigma \in E_i, \end{cases}$$

where $r_i : \mathbb{R}^3 \to [0, \max\{a_i, b_i, c_i\}]$ and $r_o : \mathbb{R}^3 \to [0, \max\{a_o, b_o, c_o\}]$ are defined by

$$r_i(\sigma) \triangleq \sqrt{a_i^2(\cos^2\beta_i(\sigma))\cos^2\alpha_i(\sigma) + b_i^2(\cos^2\alpha_i(\sigma))\sin^2\beta_i(\sigma) + c_i^2\sin^2\alpha_i(\sigma)},$$
$$r_o(\sigma) \triangleq \sqrt{a_o^2(\cos^2\beta_o(\sigma))\cos^2\alpha_o(\sigma) + b_o^2(\cos^2\alpha_o(\sigma))\sin^2\beta_o(\sigma) + c_o^2\sin^2\alpha_o(\sigma)},$$

where $\alpha_i, \alpha_o : \mathbb{R}^3 \to (-\pi/2, \pi/2)$ and $\beta_i, \beta_o : \mathbb{R}^3 \to\in (-\pi, \pi]$ are defined by

$$\alpha_i(\sigma) \triangleq \tan^{-1}\frac{\sigma_{(3)}/c_i}{\sqrt{\sigma_{(1)}^2/a_i^2 + \sigma_{(2)}^2/b_i^2}}, \quad \alpha_o(\sigma) \triangleq \tan^{-1}\frac{\sigma_{(3)}/c_o}{\sqrt{\sigma_{(1)}^2/a_o^2 + \sigma_{(2)}^2/b_o^2}},$$
$$\beta_i(\sigma) \triangleq \text{arctan2}(\sigma_{(2)}/b_i, \sigma_{(1)}/a_i), \quad \beta_o(\sigma) \triangleq \text{arctan2}(\sigma_{(2)}/b_o, \sigma_{(1)}/a_o),$$

where $\sigma_{(i)}$ is the ith entry of $\sigma \in \mathbb{R}^3$, and arctan2 is the 4-quadrant arctangent.

We use ellipsoids rather than spheres for the potential field because it allows us to elongate the potential field in the $\hat{b}_1$ direction, which is approximately the direction of flight. This feature creates a larger potential $U(p_i)$ if p_i is in the direction of flight, and a smaller potential $U(p_i)$ if p_i is not in the direction of flight.

We now introduce a method to generate the destination seeking component of the velocity commands. Let $\vec{p}_{\rm g}$ be the quadrotor's destination relative to the quadrotor's center of mass $o_{\rm B}$, and define $p_{\rm g} \triangleq [\vec{p}_{\rm g}]_{\rm I}$. Let $s_{\rm d} > 0$ be the desired speed, which is the desired quadrotor speed in the absence of any obstacles. Let $d_{\rm s} > 0$ be the stopping distance for the quadrotor, and define the goal velocity

$$V_{\rm g} \triangleq \begin{cases} \dfrac{s_{\rm d} p_{\rm g}}{||p_{\rm g}||}, & \text{if } ||p_{\rm g}|| \geq d_{\rm s}, \\[2ex] \dfrac{s_{\rm d} p_{\rm g}}{d_{\rm s}}, & \text{if } ||p_{\rm g}|| < d_{\rm s}. \end{cases}$$

Thus, if the quadrotor is farther than $d_{\rm s}$ from its destination, then the goal speed is constant, that is, $||V_{\rm g}|| = s_{\rm d}$. On the other hand, if the quadrotor is closer than $d_{\rm s}$ to its destination, then the goal speed is proportional to its distance to the destination. Let $n_{\rm e}$ be the number of points $p_1, ..., p_n$ that are contained in $E_{\rm o}$. Then, the velocity commands are given by

$$\begin{bmatrix} u_{\rm c} \\ v_{\rm c} \\ w_{\rm c} \end{bmatrix} \triangleq V_{\rm g} - \frac{1}{n_{\rm e}} \sum_{i=1}^{n} U(p_i) \frac{\mathcal{O}^{\rm T} p_i}{||p_i||}. \tag{3.9}$$

We use $n_{\rm e}$ rather than n to normalize the second term in (3.9) so that its magnitude does not depend on points outside of $E_{\rm o}$. This ensures that the second term in (3.9) is not negligible in situations where the number of points $n - n_{\rm e}$ outside of $E_{\rm o}$ is much larger than the number of points $n_{\rm e}$ contained in $E_{\rm o}$. Thus, the multi-loop guidance and control approach is given by (3.1), (3.2), (3.4)–(3.6), (3.8), and (3.9).

3.4 Numerical Examples

In this section, we present examples to demonstrate the multi-loop collision avoidance and destination seeking method. In the following simulations, we use physical parameters that match those of the quadrotor used in the experiments in Chapter 5. Specifically, $m = 1.01$ kg, $l = 0.1185$ m, $r_{\rm p} = 0.1016$ m, $k = 7.5 \times 10^{-6}$ kg·m/rad^2, $b = 1.4 \times 10^{-7}$ kg·m^2/rad^2, $I_{\rm xx} = 0.00585$ kg·m^2, $I_{\rm yy} = 0.0054$ kg·m^2, $I_{\rm zz} = 0.007$ kg·m^2, and $I_p = 0.00002$ kg·m^2.

The attitude controller gains are $k_\phi = k_\theta = k_\psi = 7$ and $K_\omega = \text{diag}(7, 7, 7)$, and the velocity controller gains are $k_u = k_v = k_w = 3.5$. These gains provide good stability and responsiveness to velocity commands. We place all poles of the transfer function $G_{\rm d}$ at -10, which yields $\alpha_3 = 40$, $\alpha_2 = 600$, $\alpha_1 = 4000$, $\alpha_0 = 10000$. We place all poles of the transfer function $G_{\psi_{\rm d}}$ at -6, which yields $\beta_2 = 9$, $\beta_1 = 27$, and $\beta_0 = 27$. We use a yaw angle command $\psi_{\rm c} = 0$ in all examples.

For all examples, we let $r_{\rm c} = 0.22$ m, $a_{\rm i} = 0.56$ m, $b_{\rm i} = 0.46$ m, $c_{\rm i} = 0.25$ m, $a_{\rm o} = 1.1$ m, $b_{\rm o} = 0.75$ m, $c_{\rm o} = 0.53$ m, $U_{\max} = 2.5$, $s_{\rm d} = 0.8$ m/s, and $d_{\rm s} = 0.8$ m.

Define $d_{\min} \triangleq \min\{||p_1||, ..., ||p_n||\}$, which is the distance between the quadrotor's center of mass $o_{\rm B}$ and the nearest point on an obstacle. The objective is to reach the destination subject to the constraint that for all $t \geq 0$, $d_{\min}(t) > r_{\rm c}$.

Example 1. The quadrotor's initial position is $[x(0) \quad y(0) \quad z(0)]^{\rm T} = [-2 \quad 0 \quad 0]^{\rm T}$ m, and the destination is $p_{\rm g}(0) = [2 \quad 0 \quad 0]^{\rm T}$ m. A single spherical obstacle with radius 0.3 m is centered at the origin. We model the obstacle with $n = 50$ evenly distributed points on the surface of the sphere. The positions of these points are $p_1, ..., p_n$.

Figure 3.3 shows an overhead view of the quadrotor's trajectory in the $\hat{\imath}$–$\hat{\jmath}$ plane. The quadrotor approaches the spherical obstacle and moves in the $-\hat{\jmath}$ direction as the obstacle enters the quadrotor's potential field. Recall that the avoidance velocity is in the direction opposite the position of the obstacle relative to the quadrotor, which is modeled with $p_1, ..., p_n$.

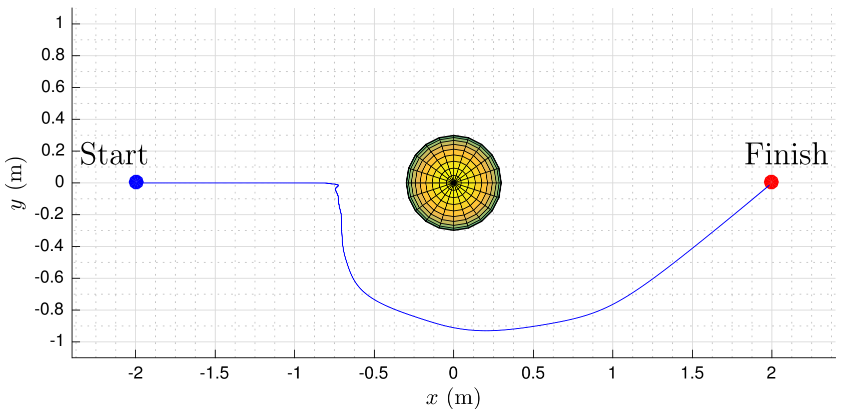

Figure 3.3: Quadrotor Trajectory and Obstacle. The quadrotor begins at the position $[-2 \quad 0 \quad 0]^{\mathrm{T}}$ m, flies towards its destination $p_{\mathrm{g}}(0) = [2 \quad 0 \quad 0]^{\mathrm{T}}$ m, and avoids the spherical obstacle.

Figure 3.4 shows that between $t = 3$ s and $t = 9$ s the quadrotor stops moving towards the destination, because at approximately $t = 3$ s, the obstacle is almost radially symmetric about $\hat{b}_1$ and lies between the quadrotor and its destination. In this case, it follows from (3.9) that u_{c}, v_{c}, and w_{c} are small. In this example, the

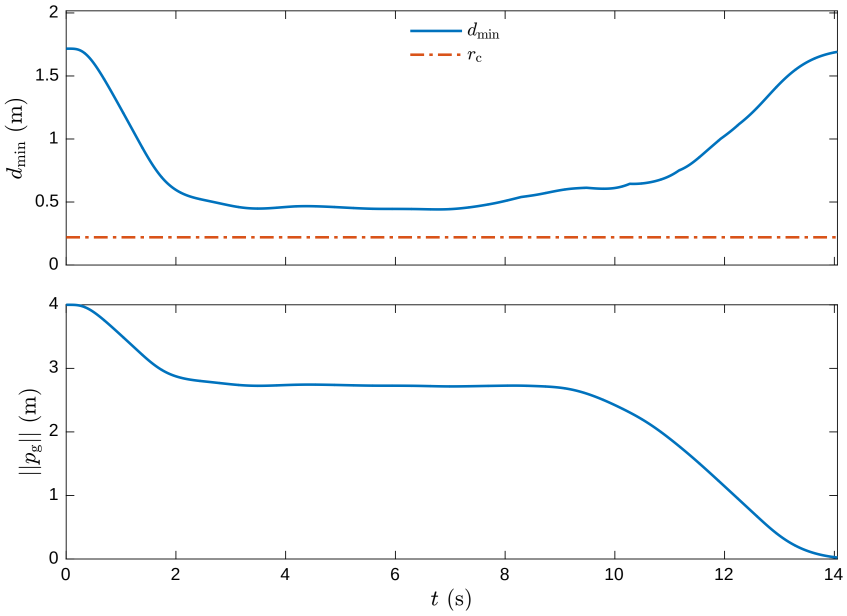

Figure 3.4: Distance to Obstacle and Distance to Destination. The top plot shows distance d_{min} to the nearest point on the obstacle. The orange dotted line is the collision distance $r_{\mathrm{c}} = 0.22$ m. The bottom plot shows the distance $||p_{\mathrm{g}}||$ to the destination.

obstacle's radial asymmetries about $\hat{b}_1$ generate a small velocity command in the $-\hat{\jmath}$ direction. As shown in Figure 3.5, the magnitude of v_c increases between $t = 3$ s and $t = 8$ s. After $t = 9$ s, the quadrotor resumes its progress towards the destination because the obstacle is no longer in the flight path.

Figure 3.5 shows the velocity commands u_c, v_c, and w_c, desired velocities u_d, v_d, and w_d, and the inertial velocities u, v, and w. The velocity commands are passed through low-pass filters G_d that generate the desired velocities, which results in time lag between the velocity commands and desired velocities. The quadrotor's velocities are approximately equal to the desired velocities, which shows that the middle-loop controller successfully tracks the desired velocities. These results agree with Proposition 2, which states that $\dot{e}_u + k_u e_u = 0$ and $\dot{e}_v + k_v e_v = 0$.

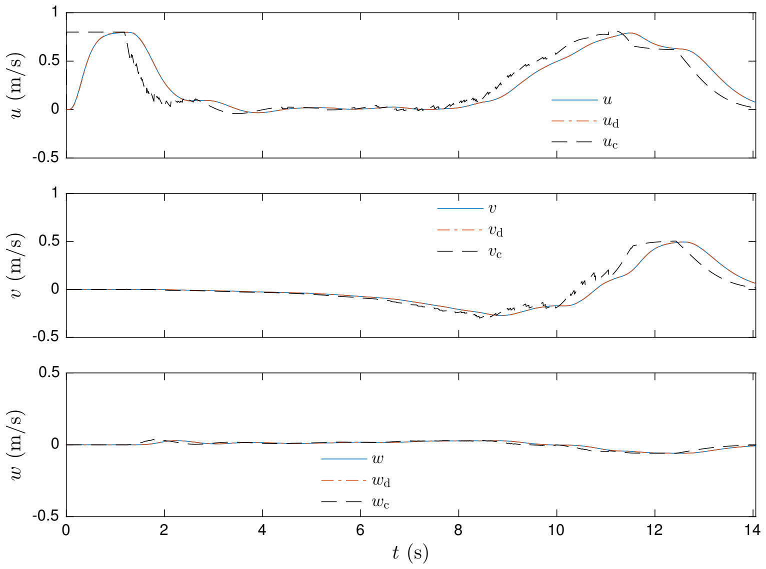

Figure 3.5: Commanded, Desired, and Inertial Velocities. The desired velocities u_d, v_d, and w_d and inertial velocities u, v, and w are approximately equal.

Figures 3.6 shows that the Euler angles are approximately equal to the desired Euler angles, which shows that the inner-loop attitude controller successfully tracks the desired Euler angles. These results provide evidence justifying the assumptions that $\phi = \phi_\mathrm{d}$ and $\theta = \theta_\mathrm{d}$, which are used in the design of the middle-loop velocity controller and in Proposition 2.

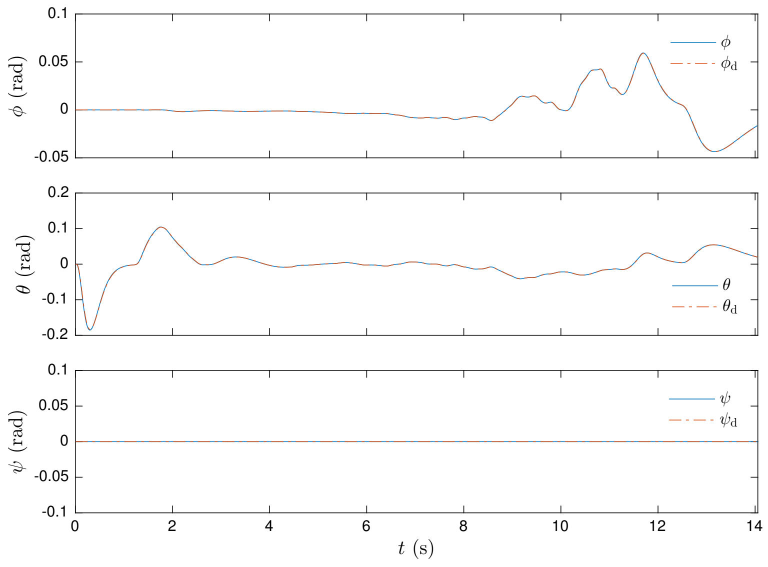

Figure 3.6: Euler Angles and Desired Euler angles. The Euler angles ϕ, θ, and ψ and the desired Euler angles ϕ_{d}, θ_{d}, and ψ_{d} are approximately equal.

Figure 3.7 shows the propeller speeds ω_1, ω_2, ω_3, and ω_4, which are the physical inputs to the quadrotor. These inputs are bounded between 550 and 600 rad/s, which is an acceptable range for the motors used in experiments in Chapter 5.

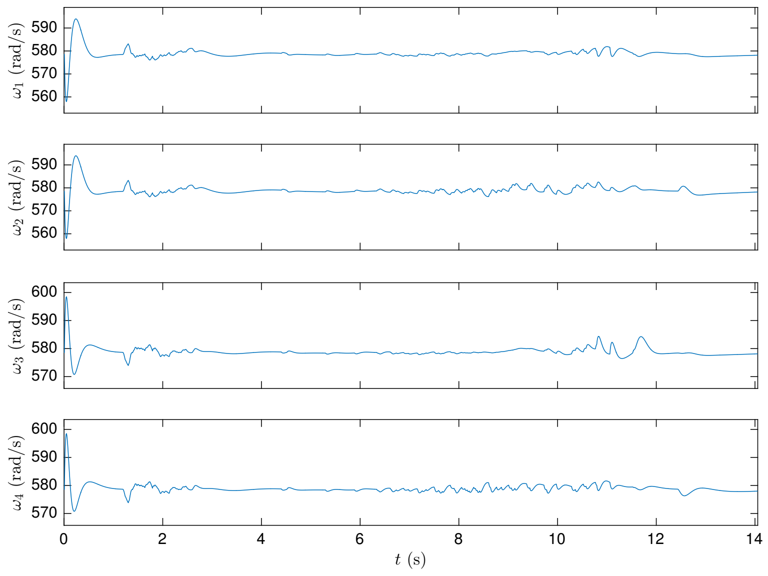

Figure 3.7: Propeller Speeds. The quadrotor's inputs are ω_1, ω_2, ω_3, and ω_4.

Example 2. In this example, we demonstrate collision avoidance and destination seeking with multiple randomly distributed obstacles. The quadrotor's initial position is $[x(0) \quad y(0) \quad z(0)]^{\mathrm{T}} = [-4 \quad 0 \quad 0]^{\mathrm{T}}$ m, and the destination is $p_{\mathrm{g}}(0) = [4 \quad 0 \quad 0]^{\mathrm{T}}$ m. We model 7 spherical obstacles (each with radius 0.3 m) using the same method as in Example 1. Each obstacle is made up of 50 evenly distributed points on its surface. Thus, $n = 350$. Figure 3.8 shows the quadrotor's trajectory in three dimensions, which shows that the quadrotor successfully avoids all of the obstacles and reaches the destination.

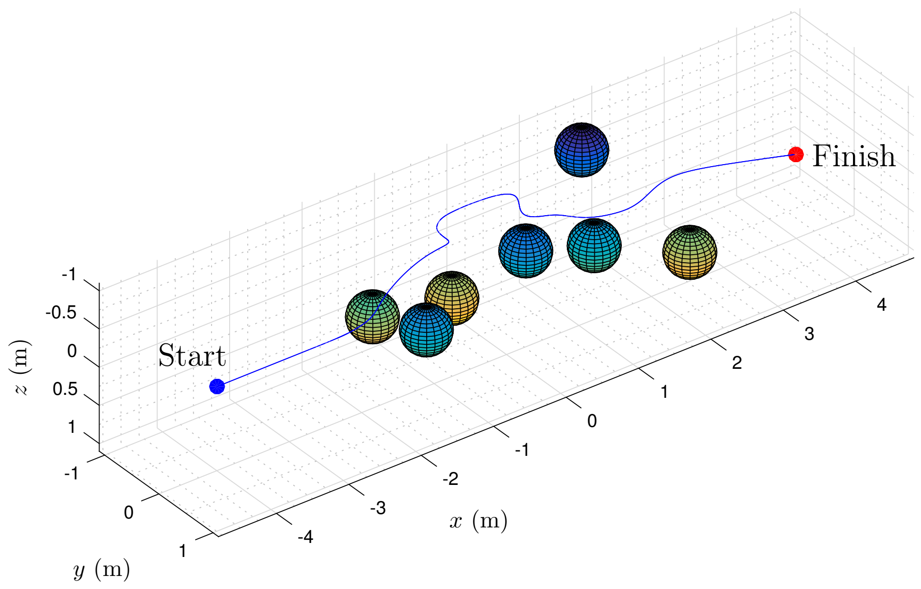

Figure 3.8: Quadrotor Trajectory and Obstacles. The quadrotor begins at the position $[-4 \quad 0 \quad 0]^{\mathrm{T}}$ m, flies towards its destination $p_{\mathrm{g}}(0) = [4 \quad 0 \quad 0]^{\mathrm{T}}$ m, and avoids all of the obstacles.

Unlike the previous example, the obstacles are placed randomly between the quadrotor's initial position and the destination. As a result, the quadrotor better maintains its destination seeking objective, which is shown by the bottom plot in Figure 3.9. Therefore, we see that obstacles that lie between the quadrotor and its destination do not significantly hinder the destination seeking objective if they are radially asymmetric about $\hat{b}_1$.

Figure 3.10 shows larger variations in the velocity commands as compared to Example 1, but the quadrotor's velocities are nonetheless approximately equal to the desired velocities.

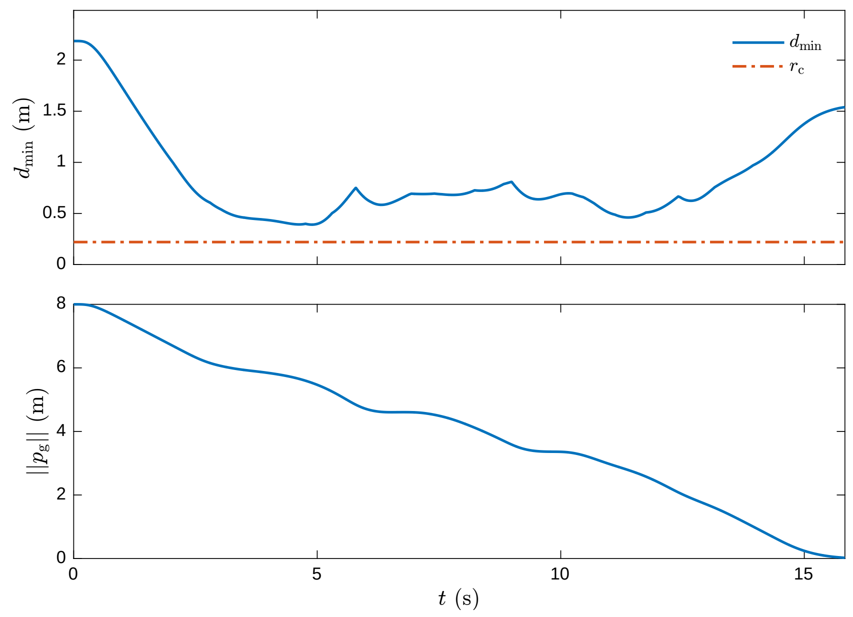

Figure 3.9: Distance to Obstacle and Distance to Destination. The top plot shows distance $d_{\min}$ to the nearest point on an obstacle. The orange dotted line is the collision distance $r_{\mathrm{c}} = 0.22$ m. The bottom plot shows the distance $||p_{\mathrm{g}}||$ to the destination.

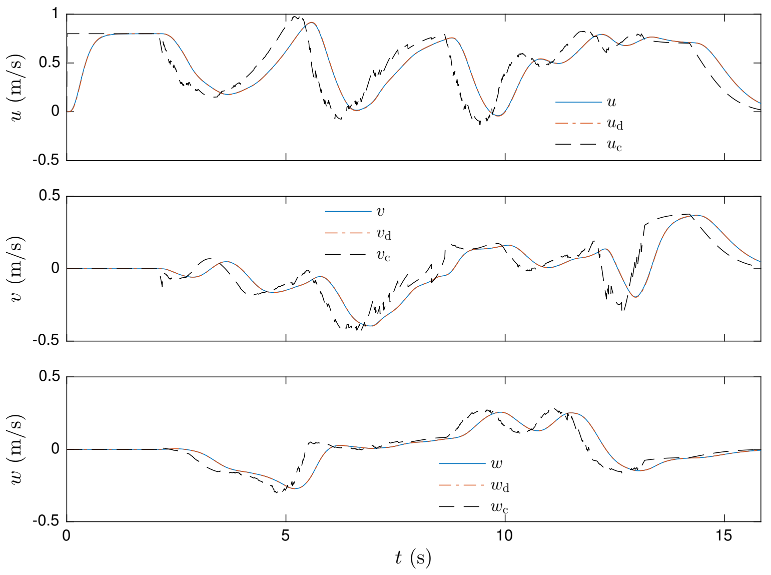

Figure 3.10: Commanded, Desired, and Inertial Velocities. The desired velocities u_{d}, v_{d}, and w_{d} and inertial velocities u, v, and w are approximately equal.

Figure 3.11 shows that the Euler angles are approximately equal to the desired Euler angles.

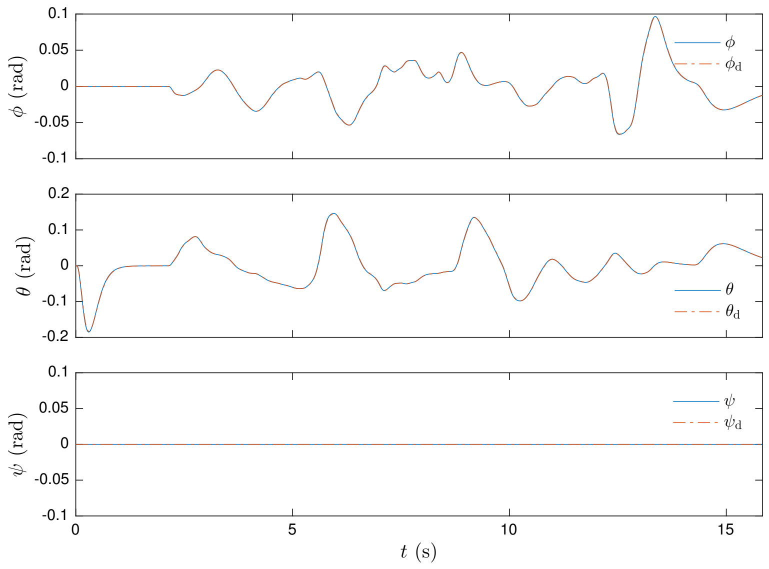

Figure 3.11: Euler Angles and Desired Euler angles. The Euler angles ϕ, θ, and ψ and the desired Euler angles ϕ_d, θ_d, and ψ_d are approximately equal.

Figure 3.12 shows the propeller speeds ω_1, ω_2, ω_3, and ω_4, which are bounded between 550 and 600 rad/s. Note that ω_1, ω_2, ω_3, and ω_4 have larger variations than in Example 1. These motor commands are needed to track the more aggressive attitude and velocity commands shown in Figures 3.10 and 3.11.

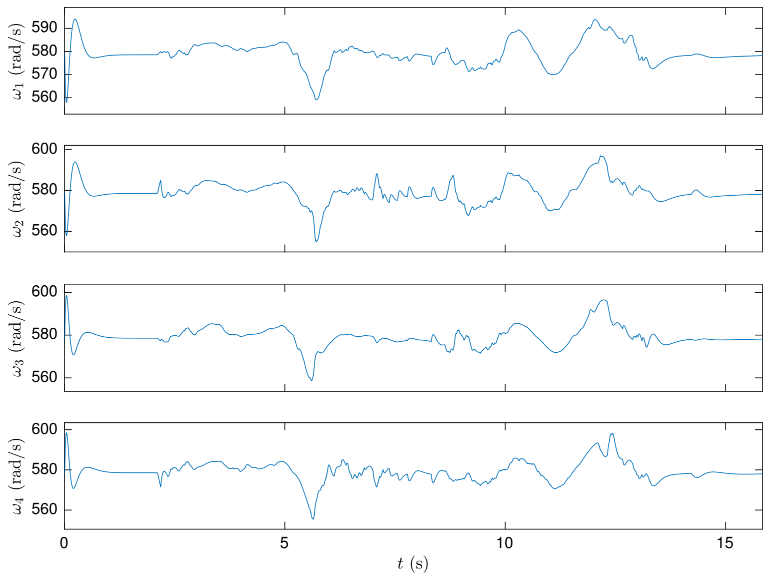

Figure 3.12: Propeller Speeds. The quadrotor's inputs are ω_1, ω_2, ω_3, and ω_4.

Chapter 4 Experimental Platform and Control Implementation

In this chapter, we describe the experimental quadrotor platform, including its hardware and software. The quadrotor is designed to fly in GPS-denied environments by using onboard sensors to determine its state as well as information describing its surrounding environment. We employ onboard vision-based sensing, specifically, the Intel Realsense R200, which is an active infrared stereoscopic camera, to determine the positions of obstacles nearby. The quadrotor is equipped with a single-board computer called the UP-Board, which processes sensor data and runs the multi-loop control algorithm (3.1), (3.2), (3.4)–(3.6), (3.8), and (3.9).

4.1 Hardware Platform

In this section, we describe the hardware components and the experimental quadrotor platform. Appendix B provides lists of the terms, hardware components, and software libraries that are used in this chapter.

We use the DJI F330 quadrotor frame, which consists of two platforms made from printed circuit board (PCB) material and four plastic arms. The PCB platforms hold the quadrotor's arms in place and provide flat surfaces to mount the hardware components. A brushless motor is mounted on the end of each arm and an electronic speed controller (ESC), which drives the motor, is taped to the underside of the same arm. The ESCs receive power through the bottom PCB, which connects to the battery and serves as a power distribution board. Each ESC has a three-wire servo lead that connects to a pulsewidth modulation (PWM) driver. The PWM driver sends signals to the ESCs to control the motors, and is controlled by the UP-Board's inter-integrated circuit (I2C) bus. The I2C bus allows the UP-Board to simultaneously communicate with multiple hardware devices, including the PWM driver. Figure 4.1 shows the motors, ESCs, and PWM driver mounted to the quadrotor frame.

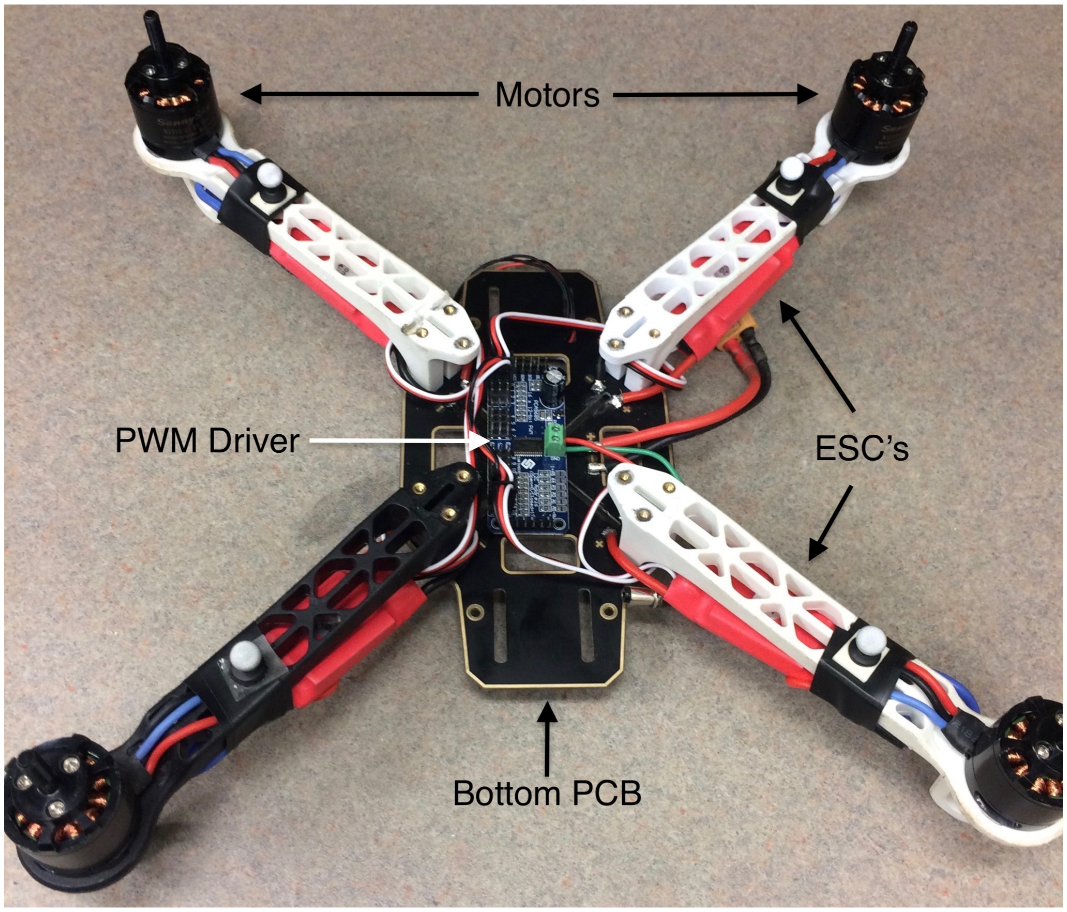

Figure 4.1: Top View of Partially Assembled Quadrotor Frame. Four brushless motors are mounted on the end of the quadrotor's arms and are connected to the ESCs, which are taped to the underside of the arms. The PWM driver is fixed to the bottom PCB and connects to each ESC via a three-wire (black-red-white) servo lead.

We mount the remaining components using 3D-printed parts that use the PCB's pre-drilled holes. On the top side of the top PCB, we use a 3D-printed part to mount the UP-Board. On the bottom side of the top PCB, we mount a 3D-printed part that supports a second part via three vibration isolating dampers. The isolated part holds the MPU-9250 inertial measurement unit (IMU), which provides feedback to the UP-Board via the I2C bus. A close-up view of the UP-Board mount and IMU enclosure is shown in Figure 4.2.

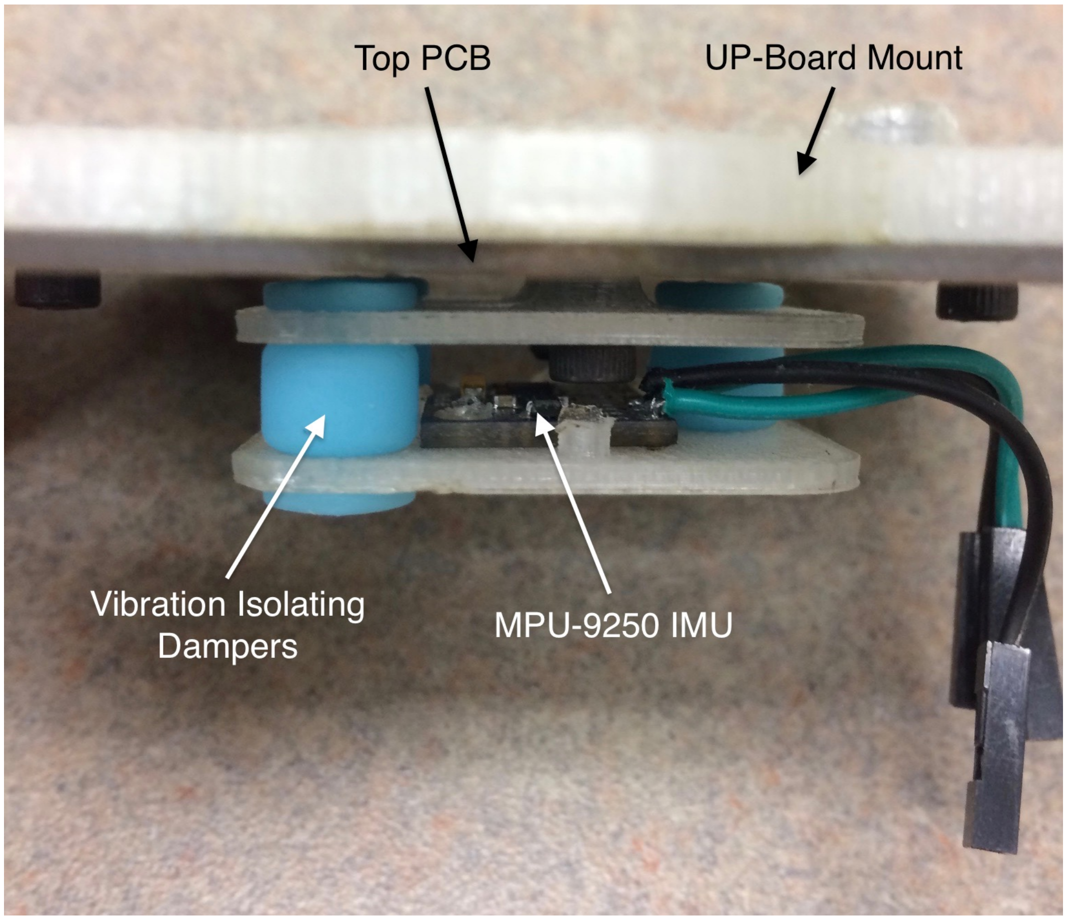

Figure 4.2: Close-Up View of IMU Enclosure. The IMU is fixed to the lower part of the enclosure, which is vibrationally isolated from the top PCB and the rest of the quadrotor frame.

On the underside of the bottom PCB, we mount a 3D-printed part that holds the battery and an optical flow sensor called the PX4flow. The PX4flow provides velocity and z-position feedback data to the UP-Board via the I2C bus. Figure 4.3 shows the UP-Board, IMU enclosure, battery slot, and PX4flow mounted to the quadrotor frame.

On the front end of the bottom PCB, we mount a 3D-printed part that holds the Realsense R200. The Realsense provides feedback to the UP-Board using the USB 3.0 bus. Figure 4.4 shows the fully assembled quadrotor, including the Realsense camera.

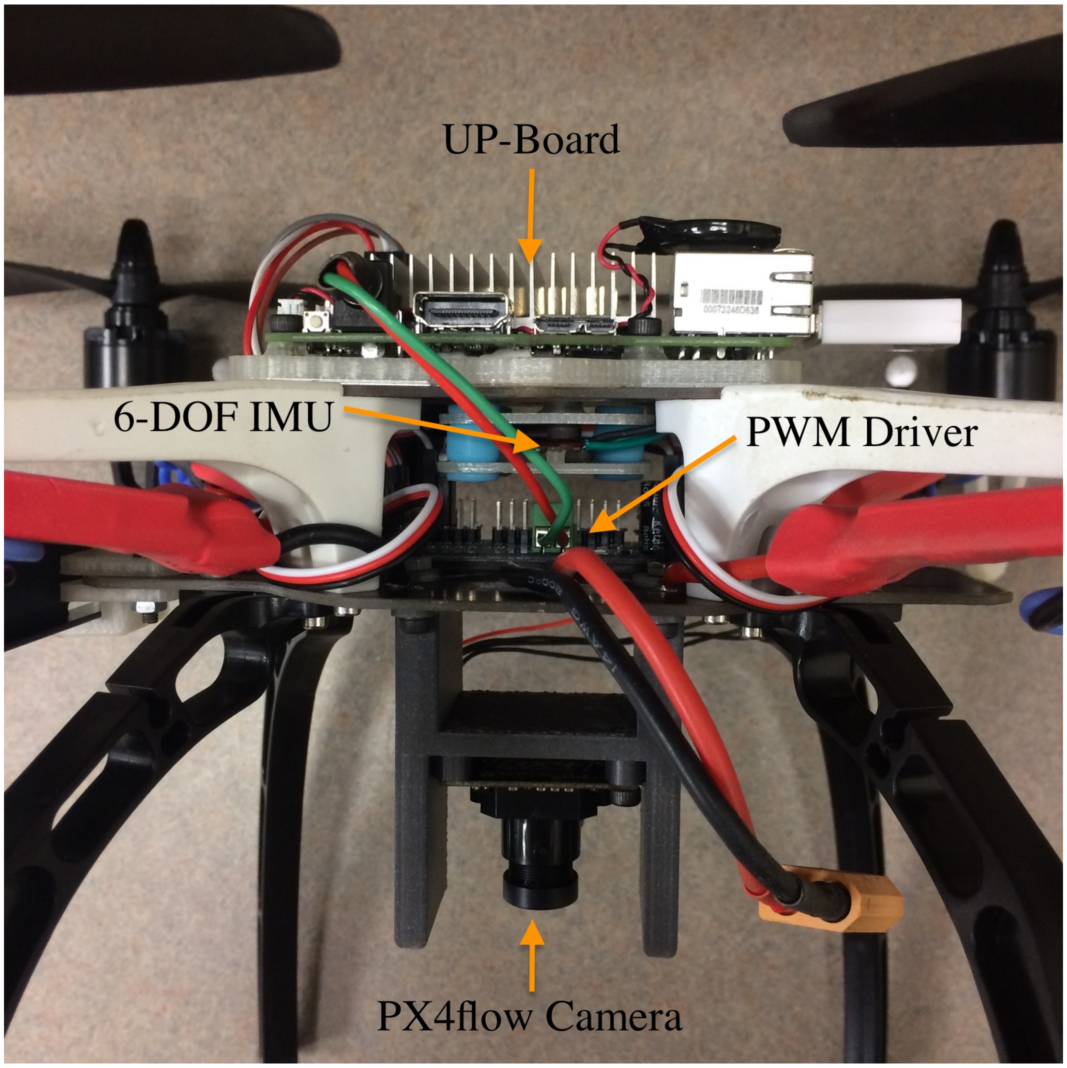

Figure 4.3: Side View of Fully Assembled Quadrotor Frame. The UP-Board is mounted to the top of the quadrotor frame, the IMU enclosure is mounted between the top and bottom PCBs, and the PX4flow camera is mounted below the battery slot.

The UP-Board is equipped with a Wi-Fi adapter that allows us to access the UP-Board's file system wirelessly from a PC or laptop on the same network. Thus, we can start and stop executable scripts, retrieve flight data, and make changes to the UP-Board's flight software remotely. We implement the multi-loop control and guidance algorithms in the script flight.cpp, which is provided in Appendix C and communicates with the hardware and sensors using the software libraries [48–51]. At every time step, flight.cpp gathers sensor data, filters the data, and implements the control inputs. Sections 4.2–4.3 describe the tasks performed at each time step.

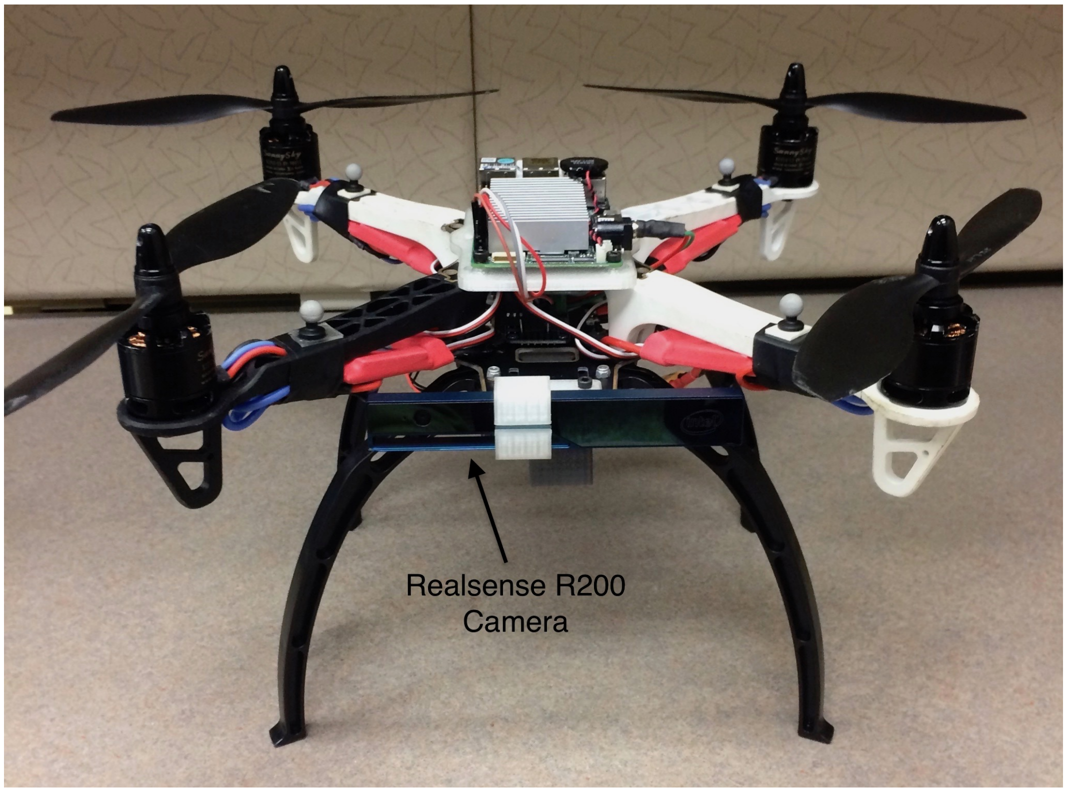

Figure 4.4: Front View of Fully Assembled Quadrotor. The Realsense camera is mounted to the front end of the bottom PCB.

4.2 Sensor Data Collection

Unless stated otherwise, we use floating point arrays to store the feedback data, which we access by array indexing. Each array is dedicated to a single data stream. For example, the yaw, pitch, and roll values are stored in three separate arrays, and each time step corresponds to a specific index common to all arrays. We use the sample time $T_\mathrm{s} \approx 0.02$ s, which is dictated by the IMU. If f is a function of time $t \geq 0$, then for all $k \in \mathbb{N} \triangleq \{0, 1, 2...\}$, define $f_k \triangleq f(kT_\mathrm{s})$, where k corresponds to the kth time step.

We obtain measurements of u and v from the PX4flow camera and a measurement of z from the Maxbotix Sonar module, which is on the PX4flow module. These measurements are made available via I2C packets by the PX4flow's stock firmware when the PX4flow is powered on. We retrieve these measurements using the UP-Board's I2C-1 bus and libmraa I2C library functions [48].

We obtain measurements of $\phi, \theta, \psi, p, q, r, \dot{u}, \dot{v}$, and $\dot{w}$ from the digital motion processor (DMP) onboard the MPU-9250 IMU. A software library called i2cdevlib [50] configures the DMP and retrieves orientation, angular rate, and acceleration estimates provided by the DMP. The DMP outputs the orientation in quaternion representation and we convert it to the 3-2-1 Euler angle representation. The DMP also compensates for gyroscope bias and thus provides non-biased estimates of p, q, and r.

We obtain measurements of $p_1, ..., p_n$ via USB 3.0 from the RealSense R200 camera using librealsense [51] library functions. The R200 provides native depth, color, and infrared streams, and librealsense computes "points" data from the depth stream and pixel positions. Each frame has a set of "points" data, which is the set of position vectors (relative to the center of the camera and resolved in B) of points on objects detected by the R200. In experiments, we use the closest point in each frame as the positions $p_1, ..., p_n$, where is n the number of frames. We limit the number of points stored to $n = 2000$, after which we replace the oldest points. Thus, we only keep track of the nearest point from each of the previous 2000 frames. We store the points $p_1, ..., p_n$ in a $3 \times n$ matrix using the matrix library called Matrix [52], and keep track of their positions relative to the quadrotor.

4.2.1 Acceleration Estimation

In this section, we describe the discrete-time extended Kalman filter used to estimate the accelerations $\dot{u}$, $\dot{v}$, and $\dot{w}$, which are used as feedback to the middle-loop velocity controller. We do not use the raw measurements of $\dot{u}$, $\dot{v}$, and $\dot{w}$, which are provided by the IMU, because they are noisy. Instead, we pass these measurements through a discrete-time extended Kalman filter. The following equations provide the information needed to implement the standard discrete-time extended Kalman filter algorithm, which is described in Appendix C.

Let $x \triangleq \begin{bmatrix} \dot{u} & \dot{v} & \dot{w} \end{bmatrix}^{\mathrm{T}}$ and $w \triangleq \begin{bmatrix} w_{\dot{\phi}} & w_{\dot{\theta}} & w_{\dot{\psi}} \end{bmatrix}^{\mathrm{T}}$, where $w_{\dot{\phi}}$, $w_{\dot{\theta}}$, and $w_{\dot{\psi}}$ are zero mean process noise terms. Let $\dot{x} = f(x, w)$ represent the system

$$\begin{aligned}
\ddot{u} &= \dot{u}((\dot{\phi} + w_{\dot{\phi}})T_\phi + (\dot{\theta} + w_{\dot{\theta}})T_\theta) - \dot{v}(\dot{\psi} + w_{\dot{\psi}}) + \frac{\ddot{w}_{\mathrm{d}} - k_w \dot{e}_w}{C_\phi C_\theta}\Theta_{31} + \frac{T}{m}(\dot{\phi} + w_{\dot{\phi}})\Theta_{21} \\
&\quad - (\dot{\theta} + w_{\dot{\theta}})C_\psi(g - \dot{w}_{\mathrm{d}} + k_w e_w) \\
\ddot{v} &= \dot{v}((\dot{\phi} + w_{\dot{\phi}})T_\phi + (\dot{\theta} + w_{\dot{\theta}})T_\theta) - \dot{u}(\dot{\psi} + w_{\dot{\psi}}) + \frac{\ddot{w}_{\mathrm{d}} - k_w \dot{e}_w}{C_\phi C_\theta}\Theta_{32} + \frac{T}{m}(\dot{\phi} + w_{\dot{\phi}})\Theta_{22} \\
&\quad - (\dot{\theta} + w_{\dot{\theta}})S_\psi(g - \dot{w}_{\mathrm{d}} + k_w e_w) \\
\ddot{w} &= \ddot{w}_{\mathrm{d}} - k_w \dot{e}_w,
\end{aligned}$$

where Θ_{ij} is the entry in the ith row and jth column of Θ. While the quadrotor is stationary, we spin the propellers to simulate in-flight vibrations, and record the values of p, q, and r provided by the IMU. We assume that these values approximate the process noise w, and use them to estimate the process noise covariance matrix, which is $W = \mathrm{diag}(0.01, 0.02, 0.01)$. The acceleration measurement obtained from the IMU's accelerometer is expressed as

$$z_k = x_k + v_k,$$

where v_k is the sensor noise. Thus, the output function h used by the discrete-time extended Kalman filter is $h(x) \triangleq x$. While the quadrotor is stationary, we spin the propellers, and assume the raw acceleration measurements from the IMU approximate the sensor noise v. Using these measurements, we estimate the sensor noise covariance matrix $R = \mathrm{diag}(2.2, 3.2, 1.6)$. We initialize the discrete-time extended Kalman filter with $x_0 = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^{\mathrm{T}}$ m/s^2, and the diagonal entries of the covariance matrix P_k (as given in Appendix C) are initialized with random values on the order of 10^{-2}.

4.2.2 Velocity and Altitude Estimation

In this section, we describe the discrete-time extended Kalman filter used to estimate the velocities u, v, and w, and the altitude z, which are used as feedback to the middle-loop velocity controller. We do not use the raw measurements of u, v, and z, provided by the PX4flow module, because they are noisy. Instead, we pass these measurements through a discrete-time extended Kalman Filter to estimate u, v, w, and z.

Let $\boldsymbol{x} \triangleq \begin{bmatrix} u & v & w & z \end{bmatrix}^{\mathrm{T}}$ and $\boldsymbol{w} \triangleq \begin{bmatrix} \boldsymbol{w}_{\dot{u}} & \boldsymbol{w}_{\dot{v}} & \boldsymbol{w}_{\dot{w}} & \boldsymbol{w}_{w} \end{bmatrix}^{\mathrm{T}}$, where $\boldsymbol{w}_{\dot{u}}$, $\boldsymbol{w}_{\dot{v}}$, $\boldsymbol{w}_{\dot{w}}$, and $\boldsymbol{w}_{w}$ are zero mean process noise terms. Let $\dot{\boldsymbol{x}} = f(\boldsymbol{x}, \boldsymbol{w})$ represent the system

$$\dot{u} = \hat{\dot{u}} - \boldsymbol{w}_{\dot{u}}, \quad \dot{v} = \hat{\dot{v}} - \boldsymbol{w}_{\dot{v}}, \quad \dot{w} = \hat{\dot{w}} - \boldsymbol{w}_{\dot{w}}, \quad \dot{z} = w - \boldsymbol{w}_{w},$$

where $\hat{\dot{u}}$, $\hat{\dot{v}}$, and $\hat{\dot{w}}$ are the acceleration estimates from section 4.2.1. Since $f(\boldsymbol{x}, \boldsymbol{w})$ is linear, it follows that the discrete-time extended Kalman filter algorithm described in Appendix C reduces to its linear form.

To characterize $\boldsymbol{w}$, we keep the quadrotor stationary, spin the propellers, and record the values of $\hat{\dot{u}}$, $\hat{\dot{v}}$, and $\hat{\dot{w}}$. Using these values, we estimate the process noise covariance matrix, which is $W = \mathrm{diag}(0.05, 0.08, 0.04)$. The measurements of u, v, and z obtained from the PX4flow module are expressed as

$$\boldsymbol{z}_k = C\boldsymbol{x}_k + \boldsymbol{v}_k,$$

where $\boldsymbol{v}_k$ is the sensor noise and

$$C \triangleq \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

The output function h used by the discrete-time Kalman filter is $h(\boldsymbol{x}) \triangleq C\boldsymbol{x}$. Define the measured speed $s_{\mathrm{m}} \triangleq \sqrt{\varkappa_{k(1)}^2 + \varkappa_{k(2)}^2}$ and the measured height $h_{\mathrm{m}} \triangleq -z_{k(3)}$. At each time step, we estimate the standard deviation of the output measurements as

$$\kappa_{\mathrm{flow}} \triangleq 0.04005232 h_{\mathrm{m}} - 0.00656446 h_{\mathrm{m}}^2 - 0.26265873 s_{\mathrm{m}}$$
$$+0.13686658 h_{\mathrm{m}} s_{\mathrm{m}} - 0.00397357 h_{\mathrm{m}}^2 s_{\mathrm{m}},$$

which is given by [53, lines 127–156]. The sensor noise covariance matrix is computed as $R = \mathrm{diag}(\kappa_{\mathrm{flow}}^2, \kappa_{\mathrm{flow}}^2, 0.01)$, where the first and second entries are the variances of the optical flow sensor's measurements of u and v, and the third entry is the variance of the sonar sensor's measurement of z. We initialize the discrete-time Kalman filter with $\boldsymbol{x}_0 = \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix}^{\mathrm{T}}$, and the diagonal entries of the covariance matrix P_k (as given in Appendix C) are initialized with random values on the order of 10^{-2}.

4.3 Propeller Speed Control

In this section, we describe the method used to actuate the brushless motors at approximately the speeds computed by (3.7). An ESC-motor pair is a single-input single-output system, where the input is the PWM signal and the output is the motor speed. We use the upm-pca9685 library [49] to command the PCA-9685 PWM driver to send a unique PWM signal to each ESC. A PWM signal is characterized by its pulsewidth and period, but only the pulsewidth determines the motor speed. The period of the PWM signal determines the maximum frequency at which the ESC can change its output, which is the maximum frequency at which the motor can change speeds.

Most quadrotor ESCs (including the ones used here) are designed to receive PWM signals with pulsewidths in the 1-to-2 ms range. Usually, a 1-ms pulsewidth corresponds to the minimum motor speed and a 2-ms pulsewidth corresponds to the maximum motor speed. We program the ESCs to accept PWM signals with a minimum pulsewidth of 0.65 ms and a maximum pulsewidth of 2.0 ms. The signals with 0.65-ms and 2.0-ms pulsewidths spin the propellers at speeds $\omega_{\min} > 0$ and $\omega_{\max} > \omega_{\min}$, respectively, which we measure using a digital tachometer. ESCs are designed to provide a near-linear relationship between pulsewidth and motor speed. Thus, given the desired propeller speeds $\omega_i \in [\omega_{\min}, \omega_{\max}]$ with $i = 1, ..., 4$ computed by (3.7), we compute the corresponding pulsewidth as

$$\mathrm{pw}_i \triangleq 0.65 + 1.35 \frac{\omega_i - \omega_{\min}}{\omega_{\max} - \omega_{\min}}.$$

We use upm-pca9685 library functions [49] to set the pulsewidth for each ESC-motor pair. The PCA-9685 driver generates PWM signals with the prescribed pulsewidths and continues to do so independently until we provide new pulsewidth values.

4.4 Estimation of Physical Parameters

Estimates of the physical parameters $m, l, k, b, I_{\mathrm{xx}}, I_{\mathrm{yy}}, I_{\mathrm{zz}}$, and I_{p} of the quadrotor platform are needed to implement the control given by (3.1), (3.2), (3.4)–(3.6), and (3.8). We estimate the parameters $m = 1.08$ kg and $l = 0.1185$ m using a scale and ruler. We estimate the moments of inertial I_{xx}, I_{yy}, and I_{zz} by treating the fully assembled quadrotor as a physical pendulum and performing a swing test. For example, to estimate I_{xx} we swing the quadrotor about an axis that is parallel to and a distance l_{xx} from $\hat{b}_1$, and measure the resulting period of oscillation P_{xx}. Then, I_{xx} is estimated as

$$I_{\mathrm{xx}} = \frac{mgl_{\mathrm{xx}}P_{\mathrm{xx}}^2}{4\pi^2} - ml_{\mathrm{xx}}^2,$$

which is derived using the parallel axis theorem and the equation for the period of a physical pendulum of mass m and inertia I_{xx} under small amplitude oscillations [54]. In this case we obtain $I_{\mathrm{xx}} = 0.00585$ kg·m^2. The moments of inertia $I_{\mathrm{yy}} = 0.0054$ kg·m^2 and $I_{\mathrm{zz}} = 0.007$ kg·m^2 are estimated in the same way. We estimate

$I_{\mathrm{p}} = 0.00002$ kg·m^2 by measuring the length and mass of the propeller and treating it as a uniform rod.

We estimate the propeller thrust coefficient k through experimental tuning. We adjust the value of k used in (3.8) until the quadrotor hovers at a desired height. Specifically, we command a constant height of 1 m using $p_{\mathrm{g}}(t) = [0 \quad 0 \quad z(t)+1]^{\mathrm{T}}$ m and observe the behavior of the quadrotor. If the quadrotor settles above 1 m, then the estimate of k is too low. If we underestimate the value of k, then the propeller speeds computed from (3.8) will be too large, and the resulting thrust will cause the quadrotor to settle above 1 m. Likewise, if the quadrotor settles below 1 m, then the estimate of k is too high. We adjust k accordingly until the quadrotor reaches 1 m altitude and remains there. For a constant pulsewidth command pw_i, the motor speeds, and consequently the thrust decrease as the battery voltage decreases. By adjusting k at various battery levels, we found that k can be expressed as $k = (e_{\mathrm{bat}} - 5.1) \times 10^{-6}$, where the battery voltage is $e_{\mathrm{bat}} \in [11.1, 12.6]$ volts. Therefore, we adjust the value of k used in the control before each flight, according to the battery voltage.

We also estimate the propeller drag coefficient b through experimental tuning. If the value of b used in (3.8) is the same order of magnitude as the value of k used in (3.8), then the quadrotor is unstable. Analogous to the case where k is overestimated, overestimating b causes the actual torque about $\hat{b}_3$ to be less than τ_{b_3}. We found that decreasing the value of b by 1 to 2 orders of magnitude stabilizes the quadrotor. Further decreasing b, we found that if $b \leq 5.0 \times 10^{-8}$ kg·m^2/rad^2, the quadrotor exhibits high-frequency, low-amplitude oscillations about the $\hat{b}_3$ axis and can become unstable. We reproduce this oscillation effect in simulation by using a smaller value for b in the control, which implies that a value $b \leq 5.0 \times 10^{-8}$ kg·m^2/rad^2 is an underestimate of the true value. Thus, we increase our estimate of b used in (3.8). The estimate is $b = 1.4 \times 10^{-7}$ kg·m^2/rad^2, which provides a stable response to yaw commands and no visible oscillations. Note that b could be better estimated by hard coding the input τ_{b_3}, and then measuring the torque produced about the $\hat{b}_3$ axis. Then, the value of b used in (3.8) could be adjusted until the torque produced about the $\hat{b}_3$ axis equals the hard-coded input τ_{b_3}. Similar to the thrust force, we expect the torque produced about $\hat{b}_3$ to decrease as the battery voltage decreases. Experiments could be conducted to determine b as a function of battery voltage.

Chapter 5 Experimental Results and Discussion

In this chapter, we present results from collision avoidance and destination seeking experiments using the quadrotor described in Chapter 4. In these experiments, all feedback data is provided by onboard sensors. In addition to the onboard position estimate, we measure the position of the quadrotor using an indoor motion-capture system (OptiTrack with Motive software). The motion-capture position measurements typically have sub-millimeter accuracy [33], and are used to validate the onboard position estimates used by the destination seeking method. We estimate the velocity using backward-Euler differentiation of the motion-capture position data, and compare those estimates to the onboard velocity estimates. The physical parameters of the quadrotor used in these experiments are $m = 1.08$ kg, $l = 0.1185$ m, $b = 1.4 \times 10^{-7}$ kg·m^2/rad^2, $I_{\text{xx}} = 0.00585$ kg·m^2, $I_{\text{yy}} = 0.0054$ kg·m^2, $I_{\text{zz}} = 0.007$ kg·m^2, and $I_p = 0.00002$ kg·m^2. The propeller thrust coefficient is computed as $k = (e_{\text{bat}} - 5.1) \times 10^{-6}$, where $e_{\text{bat}} \in [11.1, 12.6]$ volts is the battery voltage measured before takeoff.

Unless otherwise specified, the parameters that follow are those used in the experiments. We use the attitude controller gains $k_\phi = k_\theta = k_\psi = 7$ and $K_\omega = \text{diag}(7, 7, 7)$, and the velocity controller gains $k_u = k_v = 3.5$ and $k_w = 2.5$. These gains provided good stability and responsiveness to velocity commands as determined through experimental tuning. We modify the thrust control to include integral feedback of the error e_w. Specifically,

$$T = \frac{m(g - \dot{w}_\mathrm{d} + k_w e_w + k_{\mathrm{I}w} \int_0^t e_w(\tau)\mathrm{d}\tau)}{\mathrm{c}_\phi \mathrm{c}_\theta} \tag{5.1}$$

where we let $k_{\mathrm{I}w} = 1.0$. Thus, if the battery level decreases significantly during a single flight, then the integral gain on e_w will provide compensating thrust to keep the quadrotor at the desired velocity w_d. We also modify the desired pitch and roll as

$$\theta_\mathrm{d} = \tan^{-1}\left[\frac{(\dot{u}_\mathrm{d} - k_u e_u)\mathrm{c}_\psi + (\dot{v}_\mathrm{d} - k_v e_v)\mathrm{s}_\psi}{\dot{w}_\mathrm{d} - k_w e_w - k_{\mathrm{I}w} \int_0^t e_w(\tau)\mathrm{d}\tau - g}\right], \tag{5.2}$$

$$\phi_\mathrm{d} = \tan^{-1}\left[\frac{(\dot{u}_\mathrm{d} - k_u e_u)\mathrm{c}_{\theta_\mathrm{d}}\mathrm{s}_\psi - (\dot{v}_\mathrm{d} - k_v e_v)\mathrm{c}_{\theta_\mathrm{d}}\mathrm{c}_\psi}{\dot{w}_\mathrm{d} - k_w e_w - k_{\mathrm{I}w} \int_0^t e_w(\tau)\mathrm{d}\tau - g}\right]. \tag{5.3}$$

Analogous to Proposition 2, T is given by (5.1); $\theta(t) \equiv \theta_\mathrm{d}(t)$ and $\phi(t) \equiv \phi_\mathrm{d}(t)$, where $\theta_\mathrm{d}(t)$ and $\phi_\mathrm{d}(t)$ are given by (5.2) and (5.3); and for all $t \geq 0$, $\dot{w}_\mathrm{d}(t) - k_w e_w(t) - k_{\mathrm{I}w} \int_0^t e_w(\tau)\mathrm{d}\tau \neq g$, then e_u and e_v satisfy

$$\dot{e}_u + k_u e_u = 0, \quad \dot{e}_v + k_v e_v = 0.$$

We place all poles of the transfer function G_d at -5, which yields $\alpha_3 = 20$, $\alpha_2 = 150$, $\alpha_1 = 500$, $\alpha_0 = 625$. We place all poles of the transfer function G_{ψ_d} at -6, which yields $\beta_2 = 9$, $\beta_1 = 27$, and $\beta_0 = 27$. We use a yaw angle command $\psi_\mathrm{c} = 0$. We use the destination seeking parameters $s_\mathrm{d} = 0.5$ m/s and $d_\mathrm{s} = 1.0$ m. We break each flight into three consecutive segments: (*i*) the quadrotor takes off to an altitude of 1 m, (*ii*) flies to the position 1 m above the destination, and (*iii*) lands. When the quadrotor appears to have reached the end of each segment, we provide a key command that tells the quadrotor to continue to the next segment. We break the flights into these segments to avoid ground effects during segment (*ii*), which is where the quadrotor avoids obstacles, and because the optical flow and sonar sensors are inaccurate at altitudes below 0.3 m. In all experiments the quadrotor's initial position is taken to be o_{I} and all obstacles are fixed with respect to o_{I}. We do not have an onboard measurement of the destination p_{g}, so before takeoff we provide the value $p_{\mathrm{g}}(0)$. We use dead reckoning to approach the vicinity of p_{g}. The destinations in Experiments 1–3 are limited to distances between 3 and 4 m from o_{I} to ensure that the whole flight is recorded in the motion-capture volume. In Experiment 4, we fly the quadrotor in a hallway without the motion-capture system to test the collision avoidance and destination seeking methods over a larger distance.

Define $d_{\min} \triangleq \min\{||p_1||, ..., ||p_n||\}$, which is the distance between the quadrotor's center of mass o_{B} and the nearest point on an obstacle. The objective of these experiments is to reach the destination subject to the constraint that for all $t \geq 0$, $d_{\min}(t) > r_{\mathrm{c}}$.

Experiment 1. In this experiment, we let $r_{\mathrm{c}} = 0.22$ m, $a_{\mathrm{i}} = 0.56$ m, $b_{\mathrm{i}} = 0.46$ m, $c_{\mathrm{i}} = 0.25$ m, $a_{\mathrm{o}} = 1.1$ m, $b_{\mathrm{o}} = 0.75$ m, $c_{\mathrm{o}} = 0.53$ m, and $U_{\max} = 1.0$.

In this experiment there is a single obstacle. The quadrotor's initial position is the origin, and the destination is $p_{\mathrm{g}}(0) = [4 \quad 0 \quad 0]^{\mathrm{T}}$ m. The obstacle is a cardboard box (represented by the red square in Figure 5.1), which is placed between the quadrotor's initial position and its destination. In this experiment, we test obstacle avoidance in the $\hat{\imath}$–$\hat{\jmath}$ plane and thus compute w_{c} as a function of only V_{g}, that is, we do not use an avoidance velocity in the $\hat{k}$ direction.

Figure 5.1 shows an overhead view of the quadrotor's trajectory in the $\hat{\imath}$–$\hat{\jmath}$ plane, where the blue line is the onboard trajectory estimate and the black line is the offboard trajectory measurement. The onboard trajectory estimate varies locally from the offboard measurement, but the two trajectories remain close throughout the flight. The variation of the onboard estimate from the offboard measurement suggests that the onboard position estimate is subject to noise, but over time it does not accumulate a significant bias. Figure 5.2 shows the onboard estimate and offboard measurements of the quadrotor's trajectory in three dimensions.

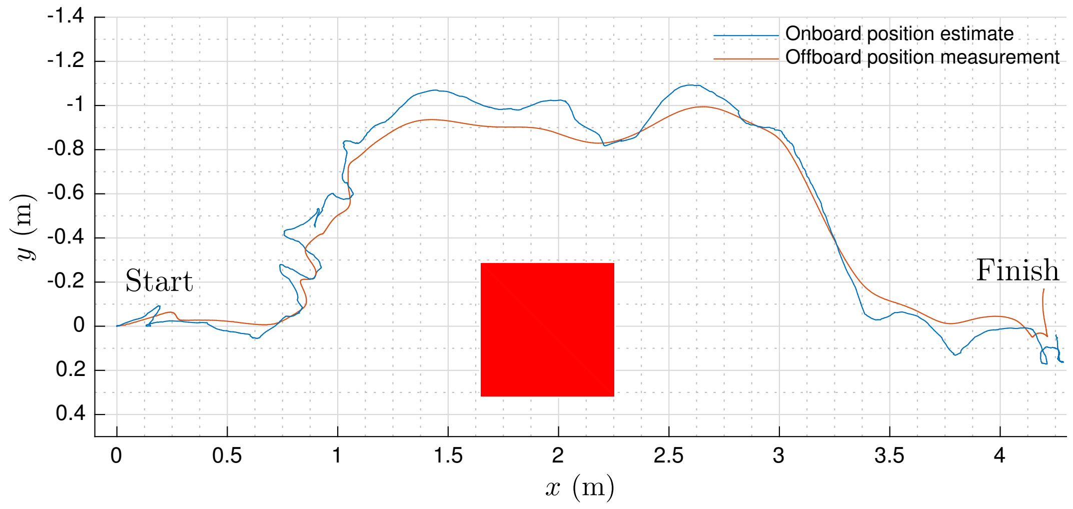

Figure 5.1: Onboard Estimate and Offboard Measurement of the Quadrotor's Trajectory in the $\hat{\imath}$–$\hat{\jmath}$ Plane.

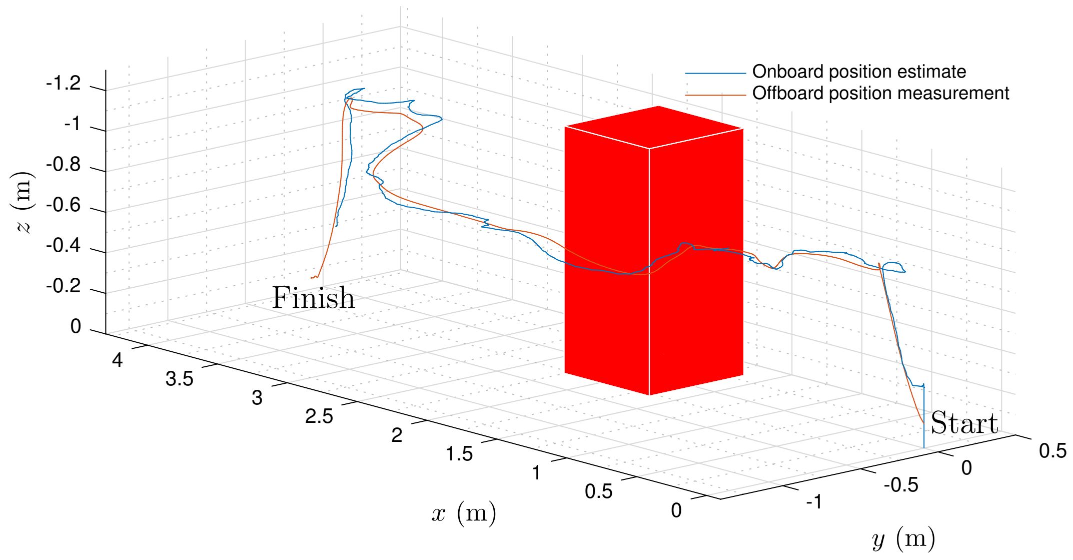

Figure 5.2: Onboard Estimate and Offboard Measurement of the Quadrotor's Trajectory in 3D.

Between $t = 5$ s and $t = 13$ s, the top plot in Figure 5.3 shows that the quadrotor slows its progress towards the destination as it approaches the obstacle. Then, the quadrotor moves in the $-\hat{\jmath}$ direction until there is a unobstructed path between o_{B} and the destination. Between $t = 13$ s and $t = 25$, the bottom plot in Figure 5.3 shows that the quadrotor resumes its progress towards the destination. Note that $||p_{\mathrm{g}}|| \neq 0$ at approximately $t = 25$ s. In this case, the quadrotor was not directly above the destination when we provided the key command to land the quadrotor.

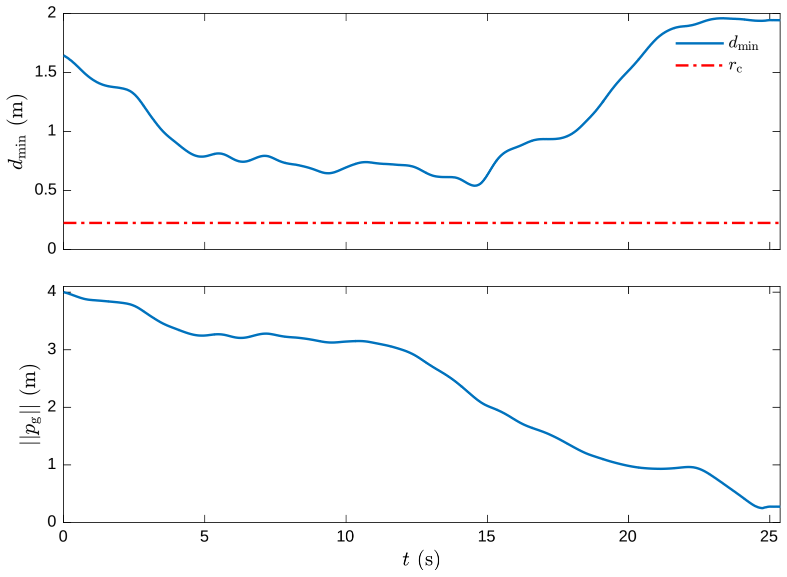

Figure 5.3: Distance to Obstacle and Distance to Destination. The top plot shows the distance $d_{\min}$ to the nearest point on the obstacle, and the orange dotted line is the collision distance $r_c = 0.22$ m. The bottom plot shows the distance $||p_g||$ to the destination.

Figure 5.4 shows the offboard position measurements and the onboard position estimates. The root mean squared (RMS) errors between the offboard measurements

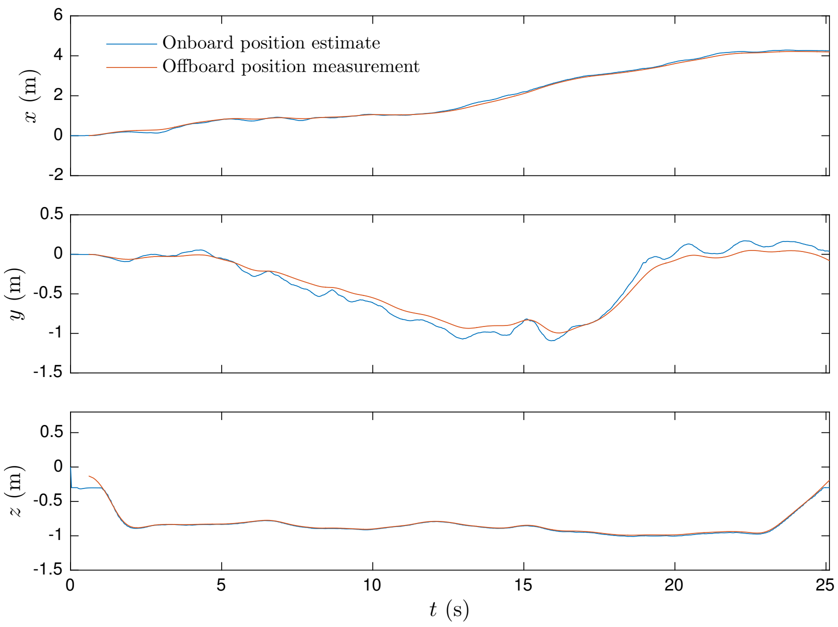

Figure 5.4: Onboard Position Estimates and Offboard Position Measurements.

and the onboard estimates of x, y, and z are 0.0648 m, 0.0815 m, and 0.0110 m respectively. The onboard z estimate is the most accurate because the sonar sensor directly measures z, while the onboard estimates of x and y are less accurate because they are obtained by integrating the estimates of u and v.

Figure 5.5 shows the offboard velocity estimates and the onboard velocity estimates.

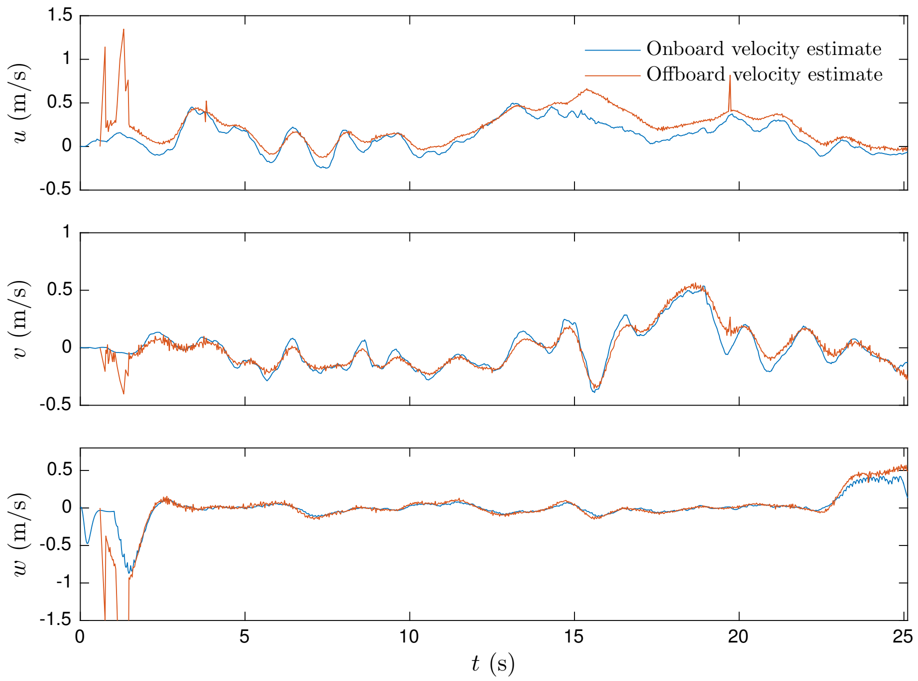

Figure 5.5: Onboard Velocity Estimates and Offboard Velocity Estimates.

The RMS errors between the offboard estimates and the onboard estimates of u, v, and w are 0.1156 m/s, 0.0522 m/s, and 0.0357 m/s respectively. Since the offboard velocity estimates are obtained by finite differentiation of the offboard position measurements we do not assume that they are velocity truths. However, we do assume that they are unbiased. The onboard velocity estimates stay close to the offboard velocity estimates, which means that the discrete-time Kalman filter described in section 4.2.2 works as expected. The sharp peaks in the offboard velocity estimates at approximately $t = 1.5$ s result from finite differentiation of noisy offboard position measurements. The noisy position measurements occur when the motion-capture system is unable to track one or more of the retro-reflective markers on the quadrotor.

Figure 5.6 shows that, on average, the velocities stay close to the desired velocities.

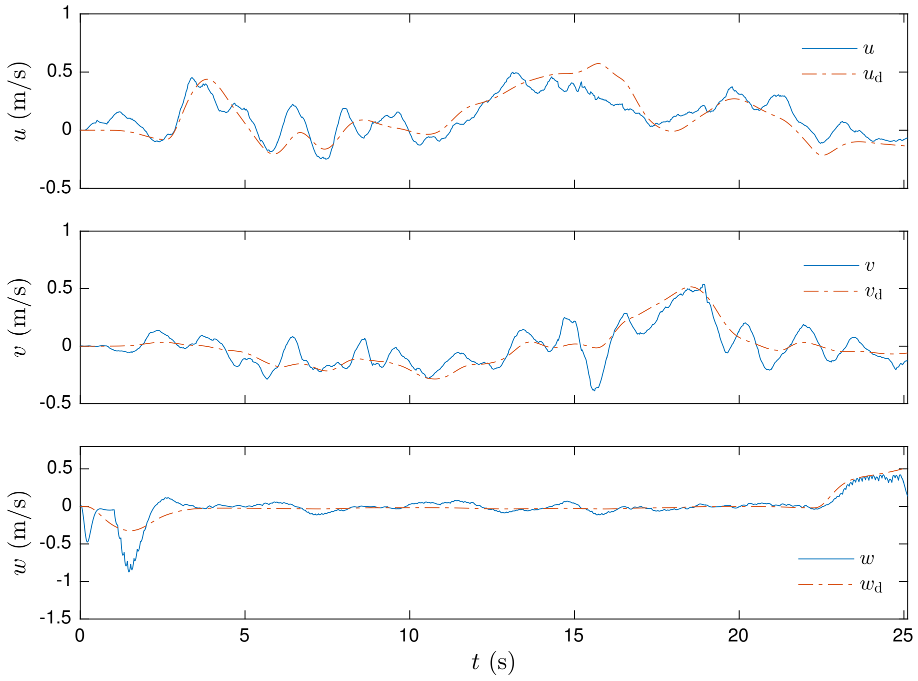

Figure 5.6: Velocities u, v, and w and Desired Velocities $u_{\mathrm{d}}, v_{\mathrm{d}}$, and w_{d}.

The velocities u and v oscillate about the desired velocities u_{d} and v_{d}, but w stays closer to the desired velocity w_{d}. We should expect better tracking of w_{d} because the physical control input T makes e_w go to zero exponentially. In contrast, we should expect larger errors e_u and e_v, since these depend on the attitude errors e_ϕ and e_θ being zero. The RMS errors between the velocities and desired velocities are 0.1146 m/s, 0.1007 m/s, and 0.0977 m/s respectively. Though these tracking errors appear to be significant in Figure 5.6, the speed of the quadrotor is small, and the destination seeking and collision avoidance objectives are met.

Figure 5.7 shows that the quadrotor roughly tracks the desired Euler angles. Similarly to the velocities, the values of the Euler angles are small, which is why the errors appear to be significant in Figure 5.7. The RMS errors between the Euler angles and desired Euler angles are 0.0386 rad, 0.0383 rad, 0.0198 rad respectively.

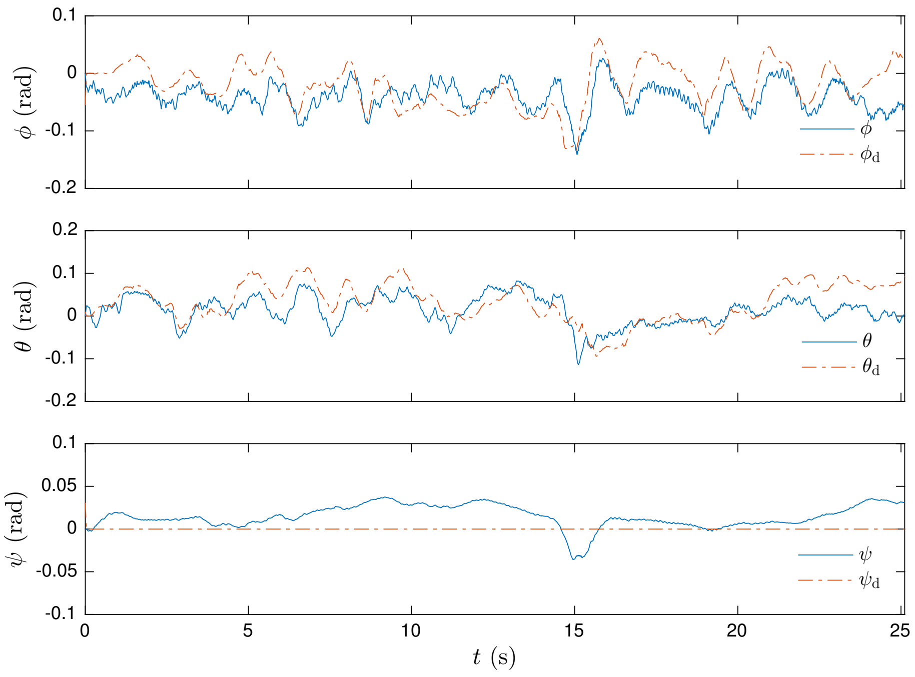

Figure 5.7: Euler Angles ϕ, θ, and ψ and Desired Euler Angles $\phi_{\mathrm{d}}, \theta_{\mathrm{d}}$, and ψ_{d}.

Figure 5.8 shows the quadrotor's commanded propeller speeds, which are not necessarily the true propeller speeds. Section 4.3 discusses the process we use to command

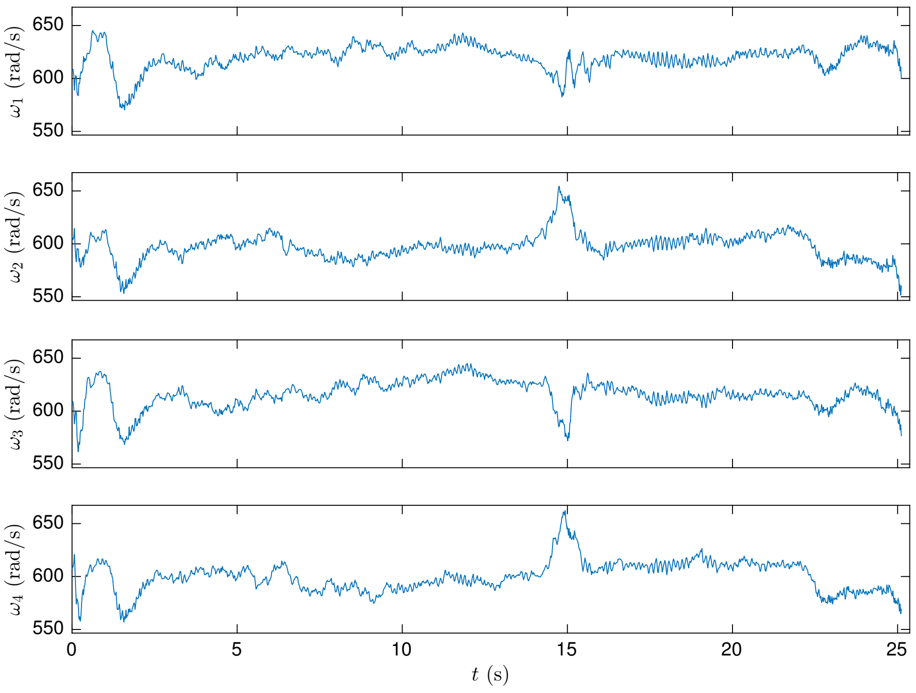

Figure 5.8: Desired Propeller Speeds $\omega_1, \omega_2, \omega_3$, and ω_4.

the propellers at the speeds in Figure 5.8. Unfortunately, there is no way to measure the true propeller speeds during a flight.

Experiment 2. In this experiment, we let $r_\mathrm{c} = 0.22$ m, $a_\mathrm{i} = 0.51$ m, $b_\mathrm{i} = 0.37$ m, $c_\mathrm{i} = 0.24$ m, $a_\mathrm{o} = 0.92$ m, $b_\mathrm{o} = 0.67$ m, $c_\mathrm{o} = 0.43$ m, and $U_\mathrm{max} = 1.1$.

In confined indoor spaces, a quadrotor could encounter multiple obstacles simultaneously. This experiment tests the behavior of the quadrotor in such a situation. The quadrotor's initial position is the origin, and the destination is $p_\mathrm{g}(0) = [3.5 \;\; 0 \;\; 0]$ m. Two obstacles are placed so that the quadrotor will detect one, and make an avoidance maneuver towards the second obstacle. Then, the quadrotor must fly in between both obstacles such that they pass through opposite sides of the quadrotor's potential field. As in Experiment 1, we compute w_c using only V_g, since there are no obstacles that the quadrotor must fly over or under.

Figure 5.9 shows an overhead view of the quadrotor's trajectory in the $\hat{\imath}$–$\hat{\jmath}$ plane.

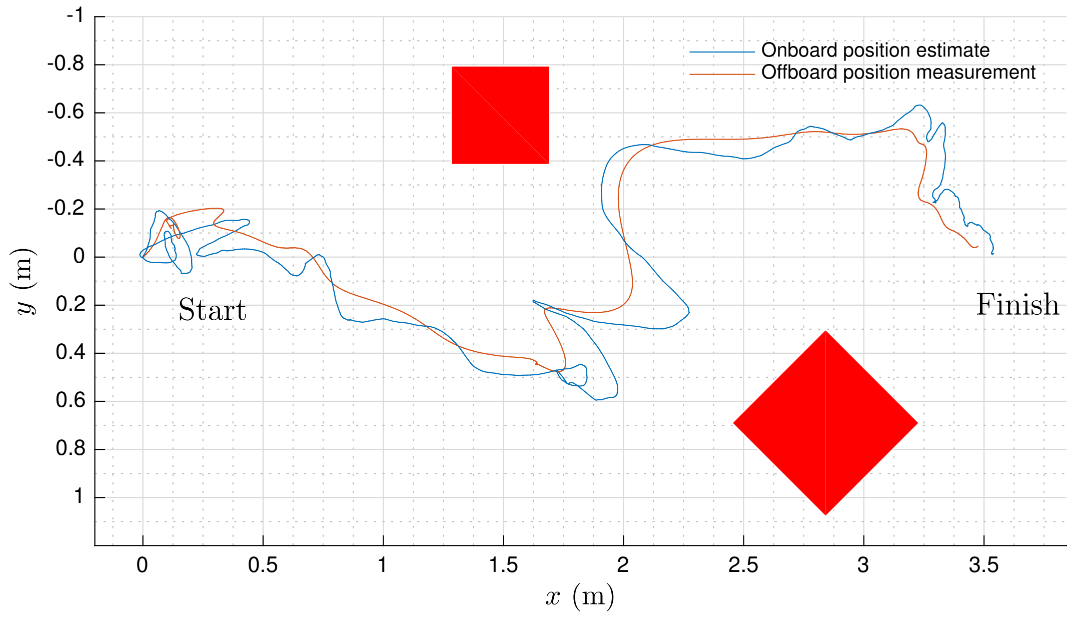

Figure 5.9: Onboard Estimate and Offboard Measurement of the Quadrotor's Trajectory in the $\hat{\imath}$–$\hat{\jmath}$ Plane. The red squares are the obstacles that the quadrotor must avoid.

Similarly to Figure 5.1, Figure 5.9 shows that the onboard position estimate varies locally from the offboard position measurement, but they remain close and are close at the end of the flight. Figure 5.10 shows the quadrotor's trajectory in three-dimensions as it avoids the obstacles and flies towards the destination. As the quadrotor moves

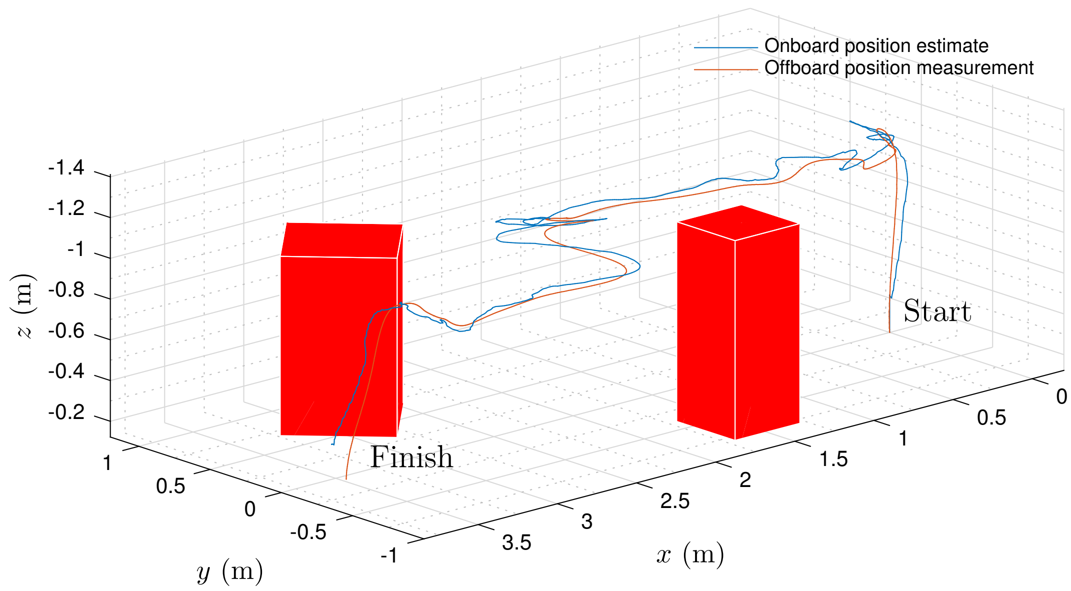

Figure 5.10: Onboard Estimate and Offboard Measurement of the Quadrotor's Trajectory in 3D. The red boxes are the obstacles that the quadrotor must avoid.

around the first obstacle, it flies towards the second obstacle, and then moves towards the back side of the first obstacle. At approximately $t = 16$ s, the top plot in Figure 5.11 shows that the quadrotor comes within 10 cm of the back side of the first obstacle. However, since we keep track of the positions $p_1, ..., p_n$ of obstacles, the quadrotor reacts to the first obstacle even though it is outside of the Realsense camera's field of view. The bottom plot in Figure 5.11 shows that the quadrotor approaches p_{g} and lands approximately 10 cm away from p_{g}.

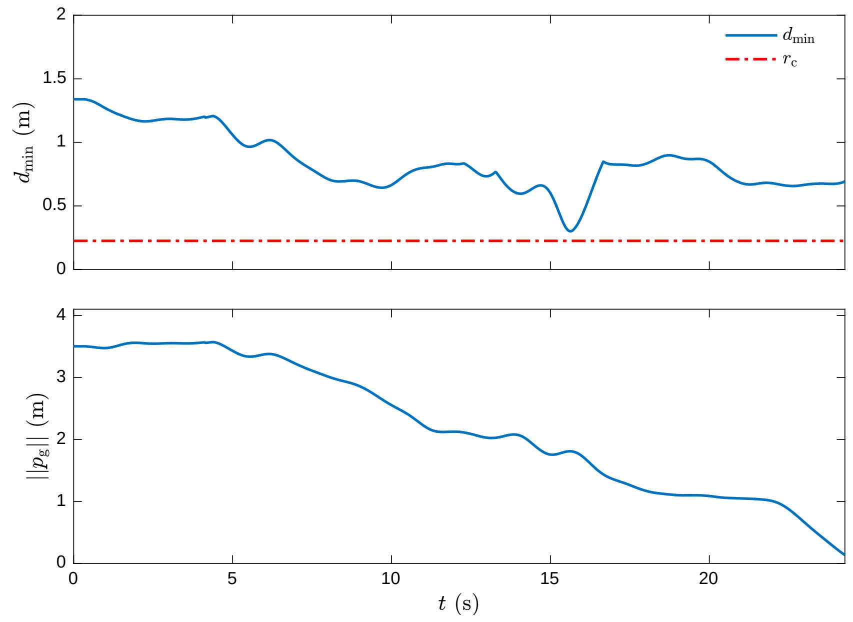

Figure 5.11: Distance to Obstacle and Distance to Destination. The top plot shows the distance $d_{\min}$ to the nearest point on the obstacle, and the orange dotted line is the collision distance $r_{\rm c} = 0.22$ m. The bottom plot shows the distance $||p_{\rm g}||$ to the destination.

Figure 5.12 shows that the onboard position estimates are close to the offboard measurements. The RMS errors between the offboard measurements and the onboard

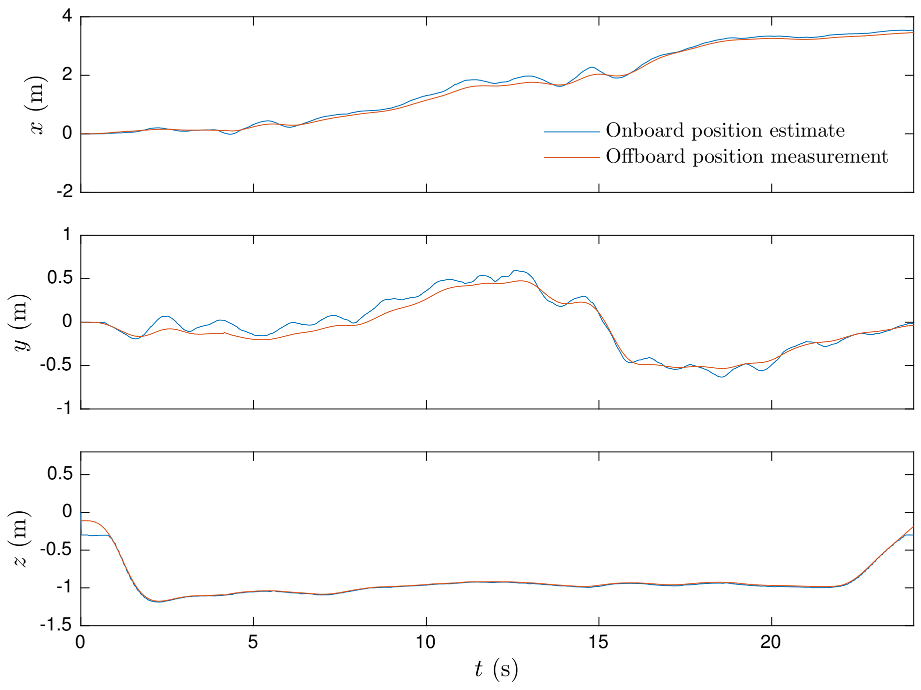

Figure 5.12: Onboard Position Estimates and Offboard Position Measurements.

estimates of x, y, and z are 0.1009 m, 0.0721 m, and 0.0099 m. The onboard estimate of z is the most accurate.

Figure 5.13 shows that the onboard velocity estimates stay close to the offboard velocity estimates. The RMS errors between the offboard estimates and the onboard

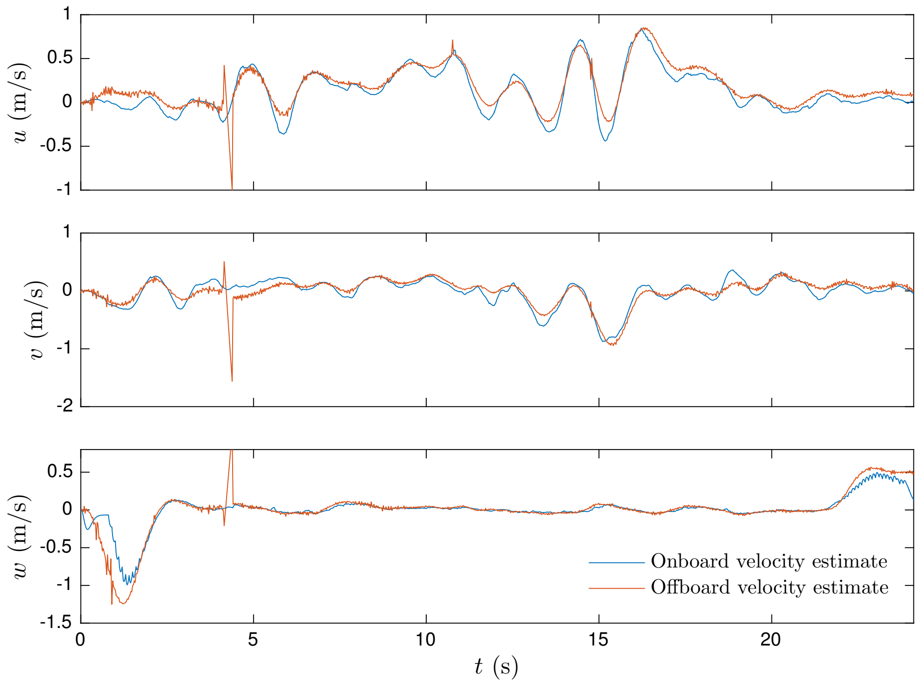

Figure 5.13: Onboard Velocity Estimates and Offboard Velocity Estimates.

estimates of u, v, and w are 0.0953 m/s, 0.1079 m/s, and 0.0995 m/s.

Figure 5.14 shows that, on average, the velocities u, v, and w stay close to the desired velocities $u_\mathrm{d}, v_\mathrm{d}$, and w_d. The RMS errors between the velocities and desired velocities are 0.1676 m/s, 0.1587 m/s, and 0.1112 m/s, which are larger than those in Experiment 1. Figure 5.15 shows that the Euler angles roughly follow the desired Euler angles. The RMS errors between the Euler angles and desired Euler angles are 0.0575 rad, 0.0394 rad, and 0.0260 rad. These RMS errors are larger than those in Experiment 1, which could explain why the RMS errors between the velocities and desired velocities are larger.

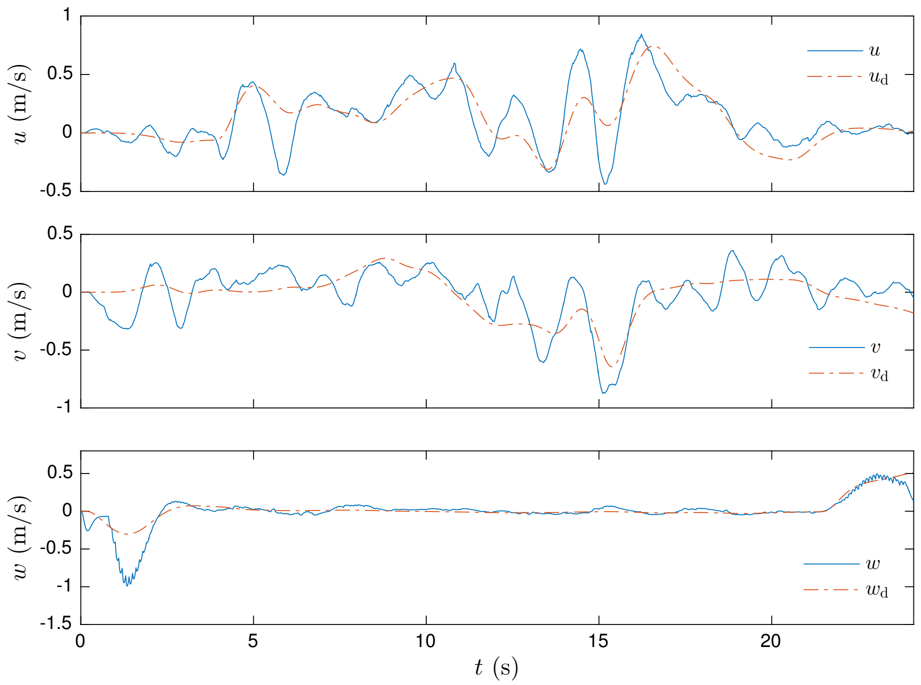

Figure 5.14: Velocities u, v, and w and Desired Velocities $u_\mathrm{d}, v_\mathrm{d}$, and w_d.

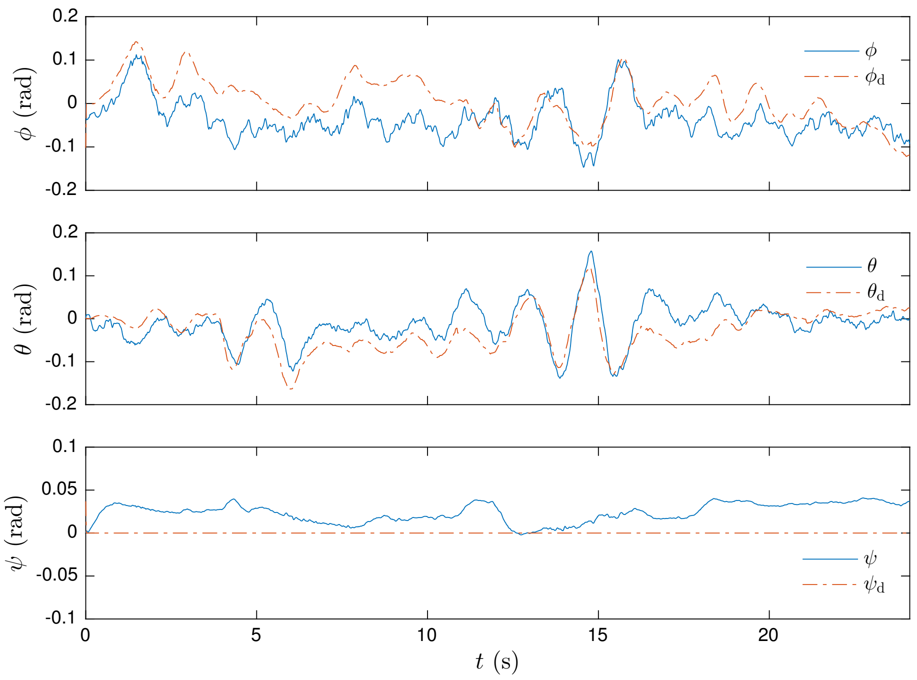

Figure 5.15: Euler Angles ϕ, θ, and ψ and Desired Euler Angles $\phi_\mathrm{d}, \theta_\mathrm{d}$, and ψ_d.

Figure 5.16 shows the commanded propeller speeds, which are the quadrotor's inputs. These inputs are bounded and do not appear to have large slopes, so they are acceptable inputs to the quadrotor's motors.

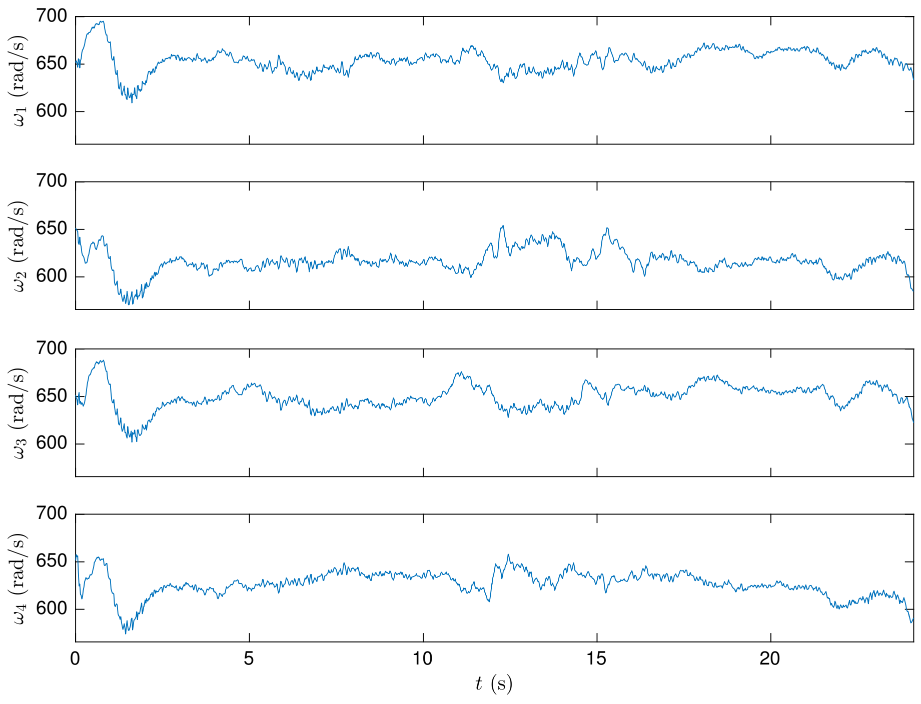

Figure 5.16: Desired Propeller Speeds $\omega_1, \omega_2, \omega_3$, and ω_4.

Experiment 3. In Experiments 1 and 2, we included avoidance velocities in the $\hat{\imath}$–$\hat{\jmath}$ plane only. In this experiment we include avoidance velocities in the $\hat{\imath}, \hat{\jmath}$, and $\hat{k}$ directions. The quadrotor's initial position is the origin, and the destination is $p_{\mathrm{g}}(0) = [3.5 \quad 0 \quad 0]^{\mathrm{T}}$ m. We use the same potential field and destination seeking parameters as in Experiment 2.

Figure 5.17 shows an overhead view of the quadrotor's trajectory in the $\hat{\imath}$–$\hat{\jmath}$ plane. Figure 5.18 shows a three-dimensional view of the quadrotor's trajectory. The quadrotor moves downwards and away from the tall part of the obstacle, then detects the low lying part of the obstacle, and proceeds to fly over it and towards the destination.

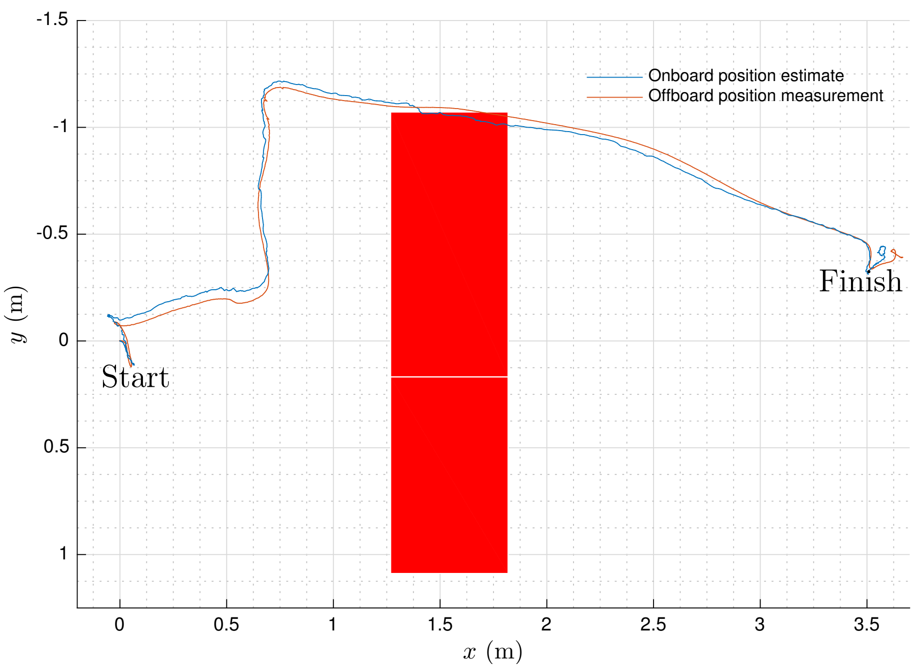

Figure 5.17: Onboard Estimate and Offboard Measurement of the Quadrotor's Trajectory in the $\hat{\imath}$–$\hat{\jmath}$ Plane.

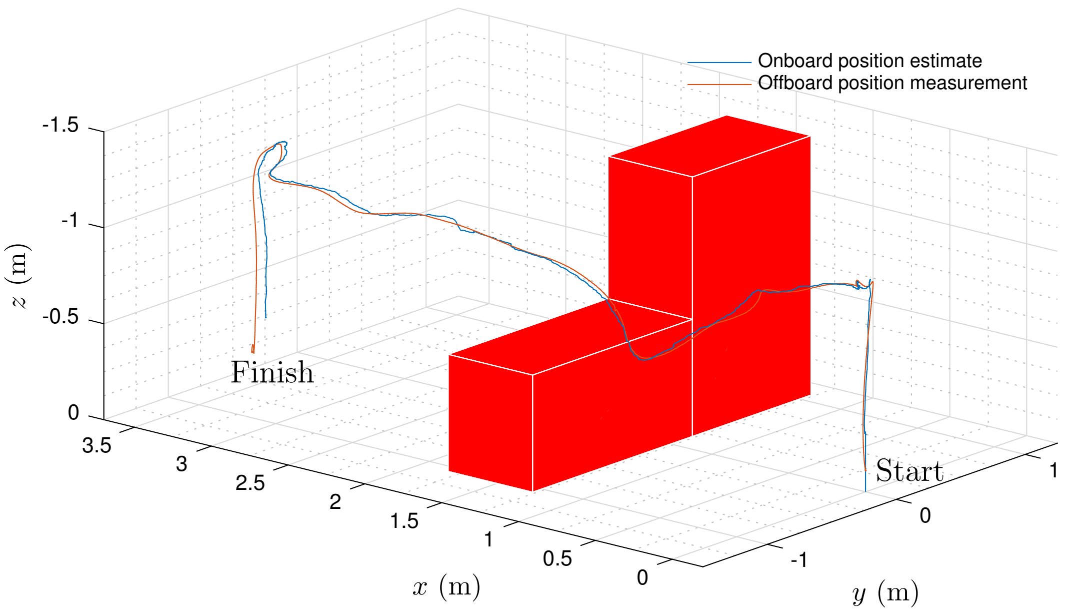

Figure 5.18: Onboard Estimate and Offboard Measurement of the Quadrotor's Trajectory in 3D.

The dip and rise of the trajectory indicates that the quadrotor follows avoidance velocity commands in three dimensions.

Figure 5.19 shows that the quadrotor reaches its destination while avoiding the obstacles. The top plot shows that the quadrotor avoids collision and remains farther

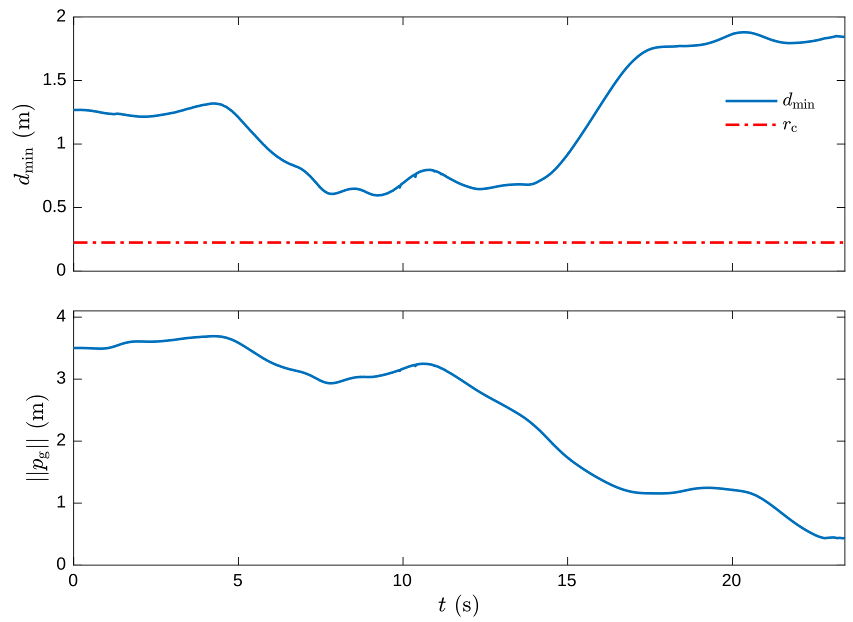

Figure 5.19: Distance to Obstacle and Distance to Destination. The top plot shows the distance d_{min} to the nearest point on the obstacle, and the orange dotted line is the collision distance $r_{\mathrm{c}} = 0.22$ m. The bottom plot shows the distance $||p_{\mathrm{g}}||$ to the destination.

than 39 cm from the obstacle. The bottom plot shows that the quadrotor lands approximately 47 cm away from its destination.

Figure 5.20 shows that the onboard position estimates are close to the offboard position measurements. The RMS errors between the offboard measurements and the onboard estimates of x, y, and z are 0.0429 m, 0.0319 m, and 0.0128 m. The onboard estimate of z is the most accurate.

Figure 5.21 shows the onboard and offboard velocity estimates. The RMS errors between the offboard estimates and the onboard estimates of u, v, and w are 0.1105 m/s, 0.1333 m/s, and 0.1237 m/s.

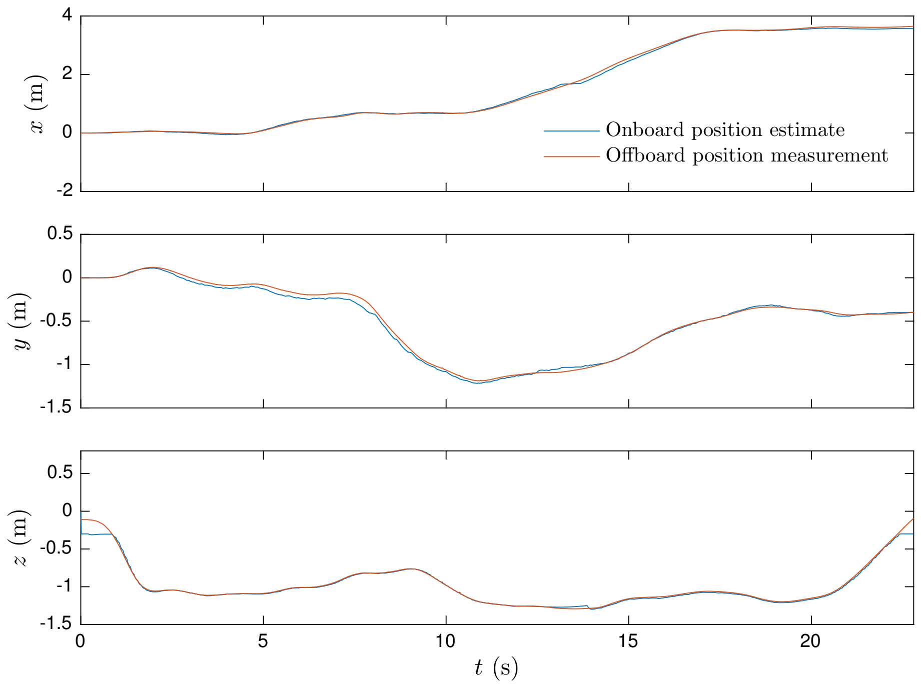

Figure 5.20: Onboard Position Estimate and Offboard Position Measurement.

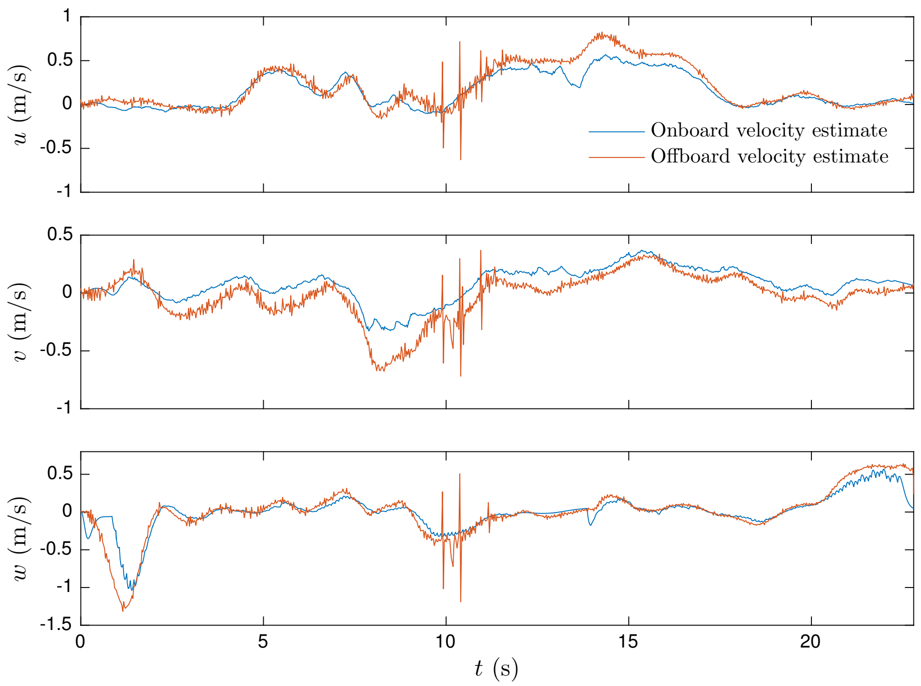

Figure 5.21: Onboard Velocity Estimates and Offboard Velocity Estimates.

Figure 5.22 shows that the quadrotor tracks the desired velocities with some occasional deviation from the desired velocities. The RMS errors between the velocities and the

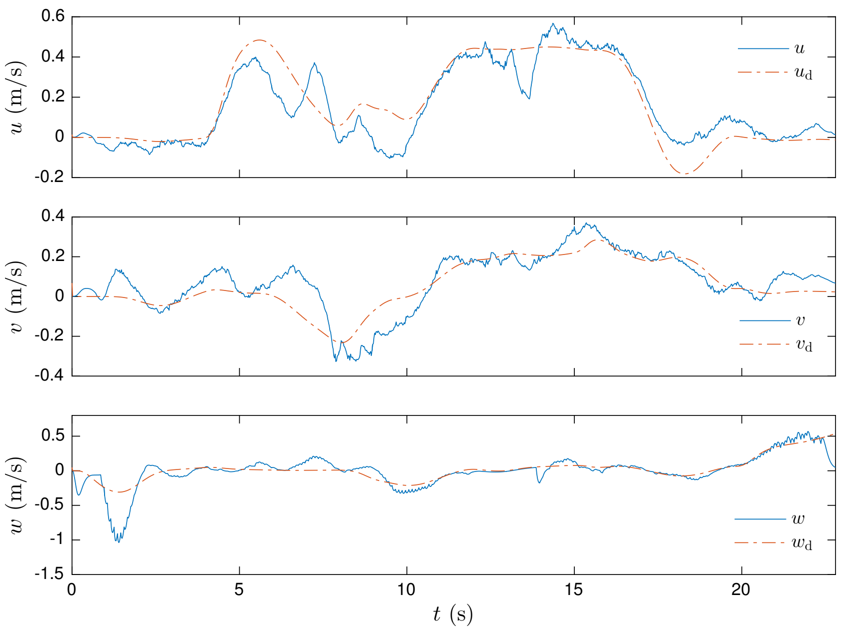

Figure 5.22: Velocities u, v, and w and Desired Velocities $u_{\mathrm{d}}, v_{\mathrm{d}}$, and w_{d}.

desired velocities are 0.0954 m/s, 0.0785 m/s, and 0.1303 m/s.

Figure 5.23 shows that the Euler angles roughly follow the desired Euler angles.

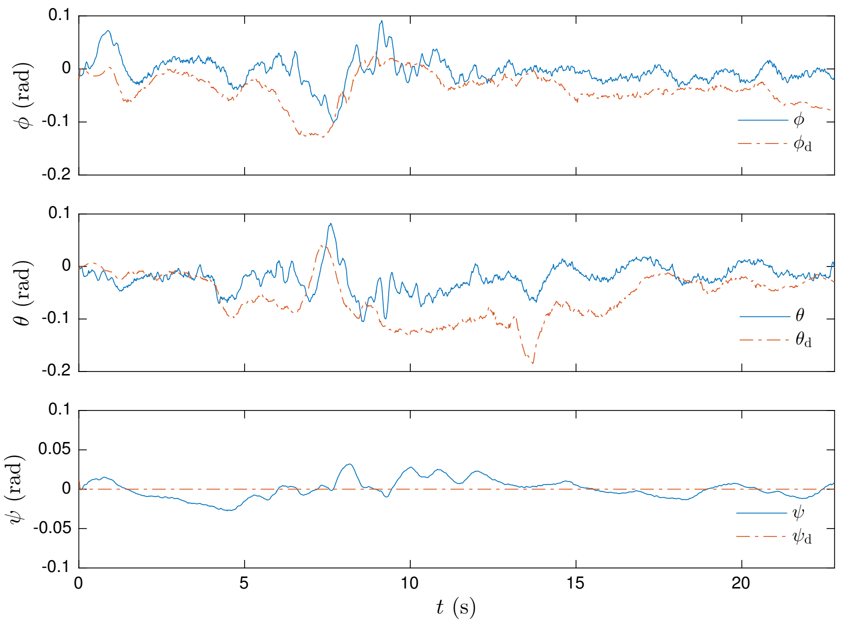

Figure 5.23: Euler Angles ϕ, θ, and ψ and Desired Euler Angles $\phi_{\mathrm{d}}, \theta_{\mathrm{d}}$, and ψ_{d}.

The RMS errors between the Euler angles and the desired Euler angles are 0.0421 rad, 0.0523 rad, and 0.0119 rad.

Figure 5.24 shows the quadrotor's desired propeller speeds.

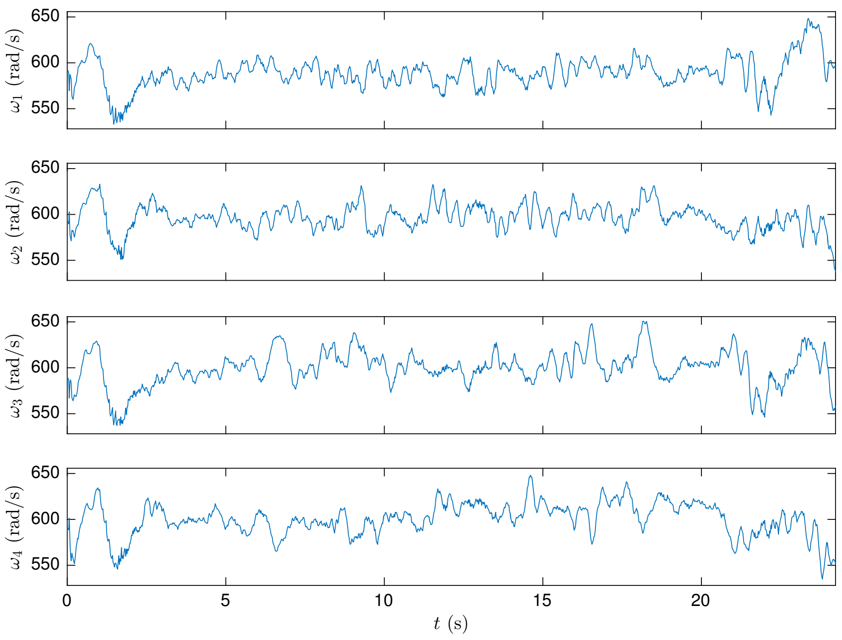

Figure 5.24: Desired Propeller Speeds $\omega_1, \omega_2, \omega_3$, and ω_4.

Experiment 4. In this experiment, we let $r_\mathrm{c} = 0.22$ m, $a_\mathrm{i} = 0.533$ m, $b_\mathrm{i} = 0.332$ m, $c_\mathrm{i} = 0.237$ m, $a_\mathrm{o} = 1.067$ m, $b_\mathrm{o} = 0.664$ m, $c_\mathrm{o} = 0.474$ m, and $U_\mathrm{max} = 1.25$. We use the destination seeking parameters $s_\mathrm{d} = 0.8$ m/s and $d_\mathrm{s} = 1.0$ m.

Experiments 1–3 were recorded in the motion-capture volume, which restricted the destination to a maximum distance of 4 m from the takeoff location. In this experiment we test the destination seeking and collision avoidance over a larger distance (10 m) through a hallway. Note that we increased the desired speed s_d. We found that a larger desired speed keeps the quadrotor on course over larger distances. Since the quadrotor will be traveling faster, we increase the maximum potential and the size of the potential field. The quadrotor's initial position is the origin and the destination is $p_\mathrm{g}(0) = [10 \quad -.2 \quad 0]$ m. As in Experiments 1 and 2, we compute w_c using only V_g

Figure 5.25 shows an overhead view of the quadrotor's trajectory in the $\hat{\imath}$–$\hat{\jmath}$ plane. Based on the previous experimental results, we assume that the trajectory shown in

Figure 5.25 (i.e., the onboard estimate) is close to the true trajectory. Figure 5.26

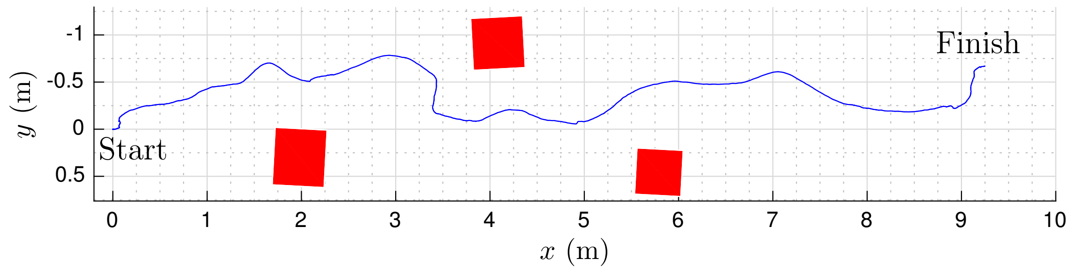

Figure 5.25: Onboard Estimate of the Quadrotor's Trajectory in the $\hat{\imath}$–$\hat{\jmath}$ Plane

shows a three-dimensional view of the quadrotor's trajectory. We visually confirmed that the quadrotor did not collide with any of the obstacles during flight, and that it landed at a distance of approximately 55 cm from the destination, which we obtained using a meter stick.

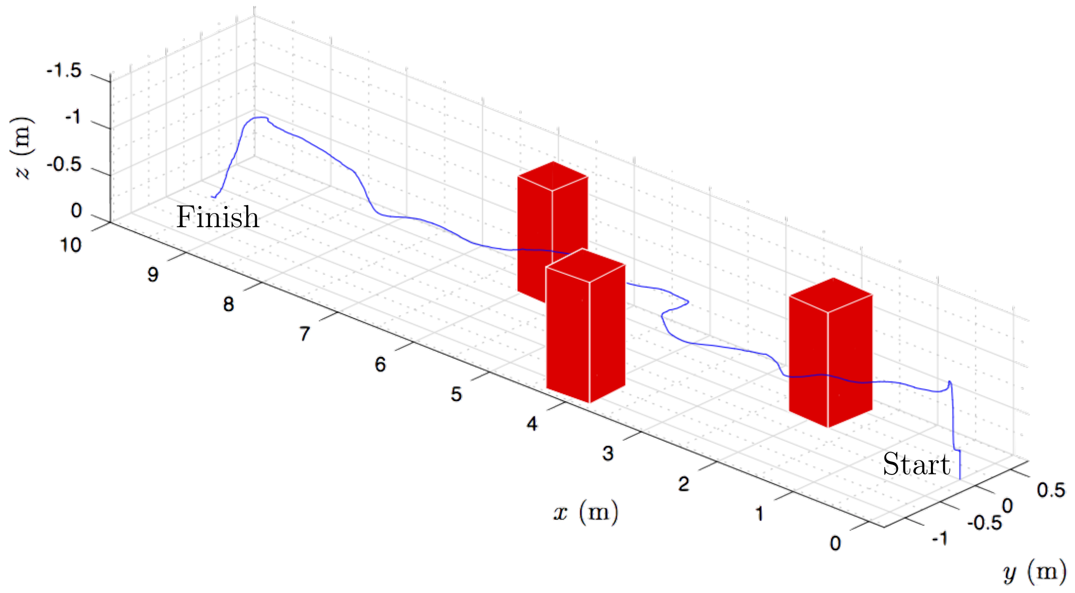

Figure 5.26: Onboard Estimate of the Quadrotor's Trajectory in 3D

The top plot in Figure 5.27 shows that the quadrotor's closest approach to an obstacle was 17 cm at approximately $t = 10$ s. The bottom plot in Figure 5.27 shows that the quadrotor approaches its destination between $t = 3$ s and $t = 25$ s. After landing, the onboard estimate of $||p_{\mathrm{g}}||$ was approximately 1 m, which was larger than the measured distance of 55 cm.

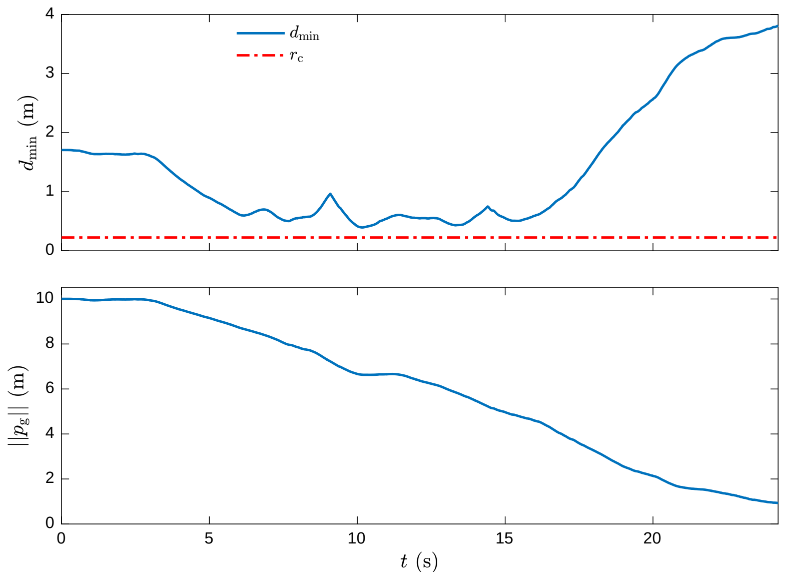

Figure 5.27: Distance to Obstacle and Distance to Destination. The top plot shows the distance $d_{\min}$ to the nearest point on the obstacle, and the orange dotted line is the collision distance $r_c = 0.22$ m. The bottom plot shows the distance $||p_g||$ to the destination.

Figure 5.28 shows that the quadrotor roughly tracks the desired velocities. The

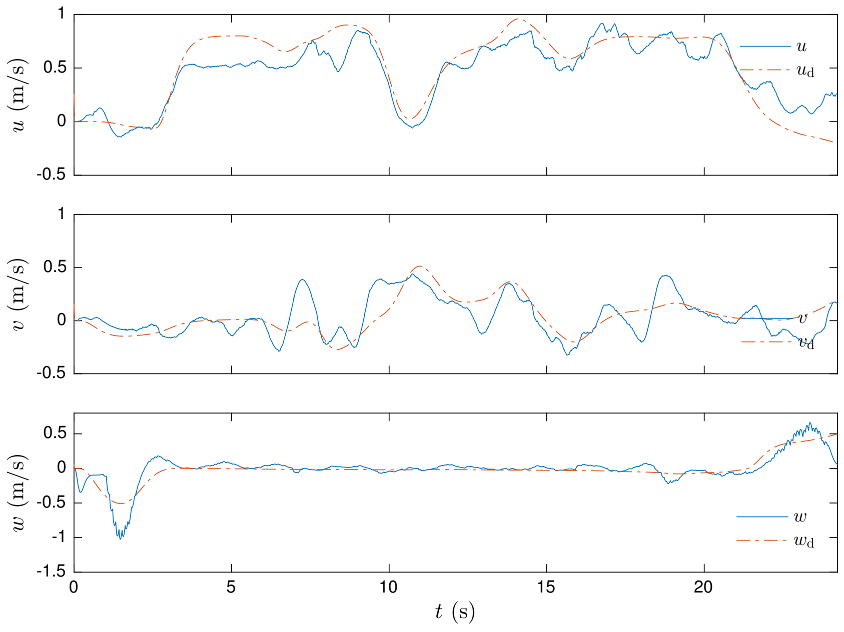

Figure 5.28: Velocities u, v, and w and Desired Velocities u_d, v_d, and w_d.

RMS errors between the velocities and the desired velocities are 0.1646 m/s, 0.1415 m/s, and 0.1095 m/s.

Figure 5.29 shows that the Euler angles roughly follow the desired Euler angles. The RMS errors between the Euler angles and the desired Euler angles are 0.0567 rad, 0.0874 rad, and 0.0149 rad.

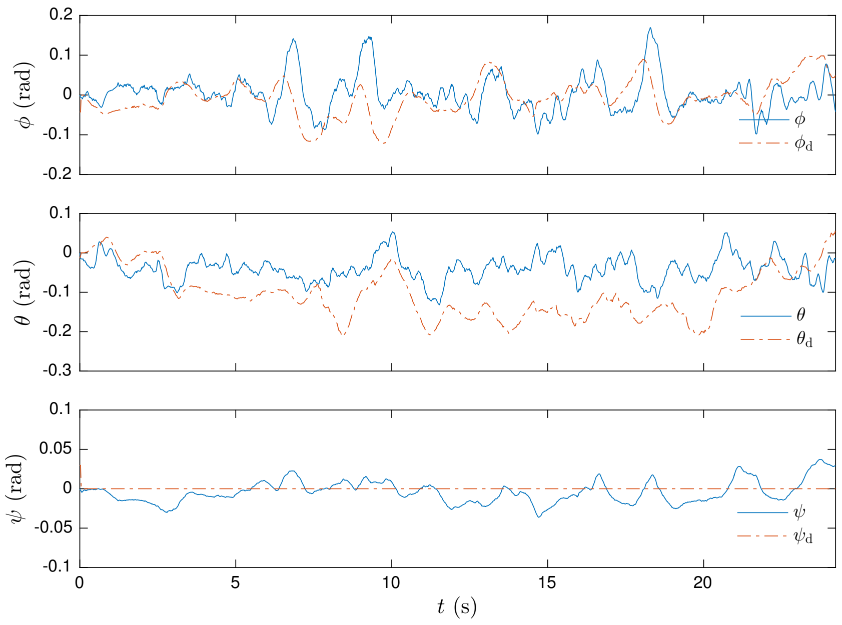

Figure 5.29: Euler Angles ϕ, θ, and ψ and Desired Euler Angles $\phi_{\mathrm{d}}, \theta_{\mathrm{d}}$, and ψ_{d}.

The quadrotor's desired propeller speeds are show in Figure 5.30.

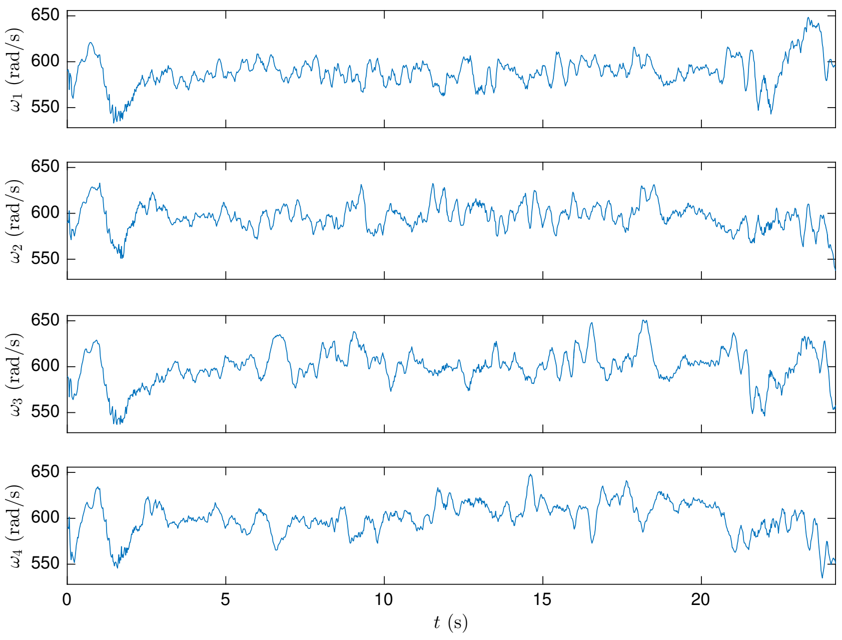

Figure 5.30: Desired Propeller Speeds $\omega_1, \omega_2, \omega_3$, and ω_4.

5.1 Discussion

In the experiments, we tested the collision avoidance and destination seeking methods using the experimental platform introduced in Chapter 4. First, we tested the collision avoidance in the i–j plane and destination seeking with one obstacle. In Experiment 2, we added an obstacle to constrain the quadrotor's movement as it avoided the first obstacle and flew towards the destination. In Experiment 3, we tested collision avoidance and destination seeking, where the avoidance velocities were generated in three dimensions. In Experiment 4, we made the destination farther than twice the distance of the previous destinations, and tested collision avoidance in the i–j plane with three obstacles. In all experiments the quadrotor avoided collision with all obstacles, and on average landed with approximately 8% (of $||p_{\mathrm{g}}(0)||$) error between its final position and its destination. For Experiments 1–3, we showed that the quadrotor's onboard position and velocity estimates stayed close to the offboard measurements, and did not accumulate significant bias. We also showed that the quadrotor tracked its desired velocities and Euler angles, but did so with errors that were larger than expected. The following discussion considers a possible source of these errors and an implementation strategy, for future work, that could reduce the controller's errors.

Recall that the desired propeller speeds are algebraic functions of the thrust T and the torques τ_{b_1}, τ_{b_2}, and τ_{b_3}, which are functions of the estimated mass and moments of inertia of the quadrotor. Our estimates of the moments of inertia are likely imperfect, which means the torque values computed by the controller are also imperfect. In addition, we made the simplifying aerodynamic assumptions (2.4) and (2.9)–(2.11) about the relationships between the thrust, torques, and propeller speeds. We also made the assumption that the ESCs drive the motors at speeds that are a linear function of the pulse-widths of the PWM signals. A significant source of the errors between the quadrotor's velocity and desired velocity, and attitude and desired attitude could be attributed to the discussed assumptions, since they are major features of implementation that were not considered in the numerical simulations. Though these assumptions were good enough to ensure that the quadrotor was stable, and that the objectives were met, other implementation strategies could be pursued to reduce the controller's errors. For example, we could measure the thrust force and drag torque produced by each motor-ESC pair as a function of pulse-widths, for a series of pulse-widths, and record the results. We could then fit a function to the results, or linearly interpolate between each recorded data point to obtain a continuous piecewise-linear function for each motor-ESC pair. Either way, the new functions would take the thrust and torques computed by (3.2) and (3.3) as inputs and output the pulse-widths of the signals that drive the ESCs. This implementation strategy could yield physical inputs that are closer to the computed inputs, since they would be supported by experiment rather than untested assumptions. As a result the inner- and middle-loop controllers could yield smaller errors between the actual and desired states.

Chapter 6 Conclusions and Future Work

We presented a multi-loop guidance and control method for a quadrotor in a GPS-denied environment. The outer-loop guidance method consists of a potential-field-based collision avoidance method and a constant speed destination seeking method, which generate desired velocities for the quadrotor. The middle-loop velocity controller uses the desired velocities to generate the thrust input and the desired attitude. The inner-loop attitude controller uses the desired attitude to generate the desired angular rates and the body-torque inputs. We proved exponential converge of the closed-loop attitude subsystem to certain Euler angle equilibria, which yields the conditions required for velocity tracking.

For the experiments, we presented a custom-built quadrotor, equipped with onboard vision sensing, an IMU, and an optical flow camera that provide feedback for the guidance and control algorithms. An onboard Linux-based single-board computer communicates the sensors and controls the motors. In Appendix D, we provide our own software that implements the guidance, navigation, and control methods. In four experiments, we showed that the quadrotor successfully achieved the collision avoidance and destination seeking objectives. We showed that the onboard estimates of position and velocity approximated the true values (provided by an offboard motion-capture system) within a reasonable margin of error.

In the future, we could refine the motor actuation strategy using measured input-output characteristics of the ESC-motor pairs. The current actuation method relies on assumptions about the ESCs and the aerodynamics of the propellers. We explained how we might discard these assumptions in favor of an approach that could provide more physically accurate thrust and torque inputs. Different estimation techniques (i.e. particle filter, unscented Kalman filter) could be implemented for comparison with the current onboard estimation techniques. Changes to the inner-loop controller could be made to make the controller more robust to parameter uncertainties, including moments of inertia, and propeller thrust and drag coefficients. In addition, the destination seeking method could be modified to prevent the well know "local-minima" problem described in [21]. For example, instead of seeking a fixed destination, the quadrotor could follow a constant speed command in the direction of known free space, which could be determined by the onboard vision sensor. Additional randomization or machine-learning heuristics could be applied to this seeking technique, while the collision avoidance method could remain the same.

Appendix A

Controller Derivatives

Recall that computation of (3.2) requires the first and second analytic derivatives of the desired pitch (3.4) and the desired roll (3.5). We re-write (3.4) and (3.5) using nested place holder definitions to make the differentiation process more feasible and reduce the number of repeated calculations in implementation. Define

$$\begin{gathered}\alpha_u \triangleq \dot{u}_\mathrm{d} - k_u e_u,\ \alpha_v \triangleq \dot{v}_\mathrm{d} - k_v e_v,\ \alpha_w \triangleq \dot{w}_\mathrm{d} - k_w e_w - g,\\ \beta_\theta \triangleq \frac{\alpha_u \mathrm{c}_\psi + \alpha_v \mathrm{s}_\psi}{\alpha_w},\ \beta_\phi \triangleq \frac{\alpha_u \mathrm{c}_{\theta_\mathrm{d}} \mathrm{s}_\psi - \alpha_v \mathrm{c}_{\theta_\mathrm{d}} \mathrm{c}_\psi}{\alpha_w}.\end{gathered} \tag{6.1}$$

Note that the following derivatives $\dot{\theta}_\mathrm{d}, \dot{\phi}_\mathrm{d}, \ddot{\theta}_\mathrm{d}, \ddot{\phi}_\mathrm{d}, \dot{p}_\mathrm{d}, \dot{q}_\mathrm{d}$, and $\dot{r}_\mathrm{d}$ where T, θ_d, and ϕ_d are given by (5.1)–(5.3), should be computed using

$$\alpha_{Iw} \triangleq \dot{w}_\mathrm{d} - k_w e_w - k_{\mathrm{I}w} \int_0^t e_w(\tau)\mathrm{d}\tau - g$$

and its appropriate derivatives in place of α_w and its derivatives.

Then

$$\begin{aligned}\theta_\mathrm{d} &= \tan^{-1}\beta_\theta,\\ \phi_\mathrm{d} &= \tan^{-1}\beta_\phi.\end{aligned} \tag{6.2}$$

Algorithmically θ_d must be computed before β_ϕ. Next we differentiate the expressions in (6.1), which yields

$$\dot{\alpha}_u = \ddot{u}_\mathrm{d} - k_u \dot{e}_u, \quad \dot{\alpha}_v = \ddot{v}_\mathrm{d} - k_v \dot{e}_v, \quad \dot{\alpha}_w = \ddot{w}_\mathrm{d} - k_w \dot{e}_w,$$

$$\dot{\beta}_\theta = \begin{bmatrix} \dfrac{\mathrm{s}_\psi}{\alpha_w} & \dfrac{\mathrm{s}_\psi}{\alpha_w} \end{bmatrix} \begin{bmatrix} \dot{\alpha}_u + \alpha_v \dot{\psi} \\ \dot{\alpha}_v - \alpha_u \dot{\psi} \end{bmatrix} - \beta_\theta \frac{\dot{\alpha}_w}{\alpha_w},$$

$$\dot{\beta}_\phi = \begin{bmatrix} \dfrac{\mathrm{s}_\psi}{\alpha_w} & \dfrac{\mathrm{c}_\psi}{\alpha_w} \end{bmatrix} \begin{bmatrix} (\dot{\alpha}_u + \alpha_v \dot{\psi})\mathrm{c}_{\theta_\mathrm{d}} - \alpha_u \dot{\theta}_\mathrm{d} \mathrm{s}_{\theta_\mathrm{d}} \\ (\alpha_u \dot{\psi} - \dot{\alpha}_v)\mathrm{c}_{\theta_\mathrm{d}} + \alpha_v \dot{\theta}_\mathrm{d} \mathrm{s}_{\theta_\mathrm{d}} \end{bmatrix} - \beta_\phi \frac{\dot{\alpha}_w}{\alpha_w}.$$

The first derivatives of the desired pitch and roll are

$$\begin{aligned} \dot{\theta}_\mathrm{d} &= \frac{\dot{\beta}_\theta}{1 + \beta_\theta^2}, \\ \dot{\phi}_\mathrm{d} &= \frac{\dot{\beta}_\phi}{1 + \beta_\phi^2}. \end{aligned} \tag{6.3}$$

Again, note $\dot{\theta}_\mathrm{d}$ must be computed before $\dot{\beta}_\phi$. Next we need the analytic equations for the inertial jerks $\ddot{u}, \ddot{v}$, and $\ddot{w}$. Differentiating (3.3) yields

$$\dot{T} = \frac{-m\dot{\alpha}_w}{\mathrm{c}_\phi \mathrm{c}_\theta} + T(\dot{\phi}\mathrm{t}_\phi + \dot{\theta}\mathrm{t}_\theta).$$

Differentiating (2.6)–(2.8) yields

$$\begin{aligned} \ddot{u} =& \frac{-\dot{T}}{m}(\mathrm{c}_\phi \mathrm{s}_\theta \mathrm{c}_\psi + \mathrm{s}_\phi \mathrm{s}_\psi) + \frac{T}{m}\big[\dot{\phi}(\mathrm{s}_\phi \mathrm{s}_\theta \mathrm{c}_\psi - \mathrm{c}_\phi \mathrm{s}_\psi) \\ & - \dot{\theta}\mathrm{c}_\phi \mathrm{c}_\theta \mathrm{c}_\psi + \dot{\psi}(\mathrm{c}_\phi \mathrm{s}_\theta \mathrm{s}_\psi - \mathrm{s}_\phi \mathrm{c}_\psi)\big], \\ \ddot{v} =& \frac{-\dot{T}}{m}(\mathrm{c}_\phi \mathrm{s}_\theta \mathrm{s}_\psi - \mathrm{s}_\phi \mathrm{c}_\psi) + \frac{T}{m}\big[\dot{\phi}(\mathrm{s}_\phi \mathrm{s}_\theta \mathrm{s}_\psi + \mathrm{c}_\phi \mathrm{c}_\psi) \\ & - \dot{\theta}\mathrm{c}_\phi \mathrm{c}_\theta \mathrm{s}_\psi - \dot{\psi}(\mathrm{c}_\phi \mathrm{s}_\theta \mathrm{c}_\psi + \mathrm{s}_\phi \mathrm{s}_\psi)\big], \\ \ddot{w} =& \ddot{\omega}_\mathrm{d} - k_\omega \dot{e}_\omega, \end{aligned} \tag{6.4}$$

where $\ddot{w}$ results from differentiating the closed loop dynamics of e_w. Taking the second derivative of (6.1) yields

$$\ddot{\alpha}_u = \dddot{u}_\mathrm{d} - k_u \ddot{e}_u, \quad \ddot{\alpha}_v = \dddot{v}_\mathrm{d} - k_v \ddot{e}_v, \quad \ddot{\alpha}_w = \dddot{w}_\mathrm{d} - k_w \ddot{e}_w,$$

$$\ddot{\beta}_\theta = \begin{bmatrix} \frac{c_\psi}{\alpha_w} & \frac{s_\psi}{\alpha_w} \end{bmatrix} \begin{bmatrix} \ddot{\alpha}_u + 2\dot{\alpha}_v\dot{\psi} - \alpha_u\dot{\psi}^2 + \alpha_v\ddot{\psi} \\ \ddot{\alpha}_v - 2\dot{\alpha}_u\dot{\psi} - \alpha_v\dot{\psi}^2 - \alpha_u\ddot{\psi} \end{bmatrix} - 2\dot{\beta}_\theta\frac{\dot{\alpha}_w}{\alpha_w} - \beta_\theta\frac{\ddot{\alpha}_w}{\alpha_w}$$

$$\ddot{\beta}_\phi = \begin{bmatrix} \frac{s_\psi}{\alpha_w} & \frac{c_\psi}{\alpha_w} \end{bmatrix} \begin{bmatrix} [\ddot{\alpha}_u + \alpha_v\ddot{\psi} + 2\dot{\alpha}_v\dot{\psi} - \alpha_u(\dot{\theta}_\mathrm{d}^2 + \dot{\psi}^2)]c_{\theta_\mathrm{d}} - [2\dot{\theta}_\mathrm{d}(\dot{\alpha}_u + \alpha_v\dot{\psi}) + \alpha_u\ddot{\theta}_\mathrm{d}]s_{\theta_\mathrm{d}} \\ [\alpha_u\ddot{\psi} - \ddot{\alpha}_v + 2\dot{\alpha}_u\dot{\psi} + \alpha_v(\dot{\theta}_\mathrm{d}^2 + \dot{\psi}^2)]c_{\theta_\mathrm{d}} + [2\dot{\theta}_\mathrm{d}(\dot{\alpha}_v - \alpha_u\dot{\psi}) + \alpha_v\ddot{\theta}_\mathrm{d}]s_{\theta_\mathrm{d}} \end{bmatrix}$$
$$- 2\dot{\beta}_\phi\frac{\dot{\alpha}_w}{\alpha_w} - \beta_\phi\frac{\ddot{\alpha}_w}{\alpha_w}.$$

The second derivatives of the desired pitch and roll are

$$\begin{aligned} \ddot{\theta}_\mathrm{d} &= \frac{\ddot{\beta}_\theta}{1+\beta_\theta^2} - \frac{2\beta_\theta\dot{\beta}_\theta^2}{(1+\beta_\theta^2)^2}, \\ \ddot{\phi}_\mathrm{d} &= \frac{\ddot{\beta}_\phi}{1+\beta_\phi^2} - \frac{2\beta_\phi\dot{\beta}_\phi^2}{(1+\beta_\phi^2)^2}, \end{aligned} \tag{6.5}$$

where $\ddot{\theta}_\mathrm{d}$ must be computed before $\ddot{\beta}_\phi$. Differentiating (3.1) yields

$$\begin{aligned} \dot{p}_\mathrm{d} =& (\ddot{\phi}_\mathrm{d} - k_\phi\dot{e}_\phi) - (\ddot{\psi}_\mathrm{d} - k_\psi\dot{e}_\psi)s_\theta - (\dot{\psi}_\mathrm{d} - k_\psi e_\psi)\dot{\theta}c_\theta, \\ \dot{q}_\mathrm{d} =& (\ddot{\theta}_\mathrm{d} - k_\theta\dot{e}_\phi)c_\phi - (\dot{\theta}_\mathrm{d} - k_\theta e_\phi)\dot{\phi}s_\phi + (\ddot{\psi}_\mathrm{d} - k_\psi\dot{e}_\psi)s_\phi c_\theta \\ &+ (\dot{\psi}_\mathrm{d} - k_\psi e_\psi)(\dot{\phi}c_\phi c_\theta - \dot{\theta}s_\phi s_\theta), \\ \dot{r}_\mathrm{d} =& -(\ddot{\theta}_\mathrm{d} - k_\theta\dot{e}_\phi)s_\phi - (\dot{\theta}_\mathrm{d} - k_\theta e_\phi)\dot{\phi}c_\phi + (\ddot{\psi}_\mathrm{d} - k_\psi\dot{e}_\psi)c_\phi c_\theta \\ &- (\dot{\psi}_\mathrm{d} - k_\psi e_\psi)(\dot{\phi}s_\phi c_\theta + \dot{\theta}c_\phi s_\theta). \end{aligned} \tag{6.6}$$

Appendix B

Hardware Terminology

- SBC – single board computer: a computer or microcontroller built on a single circuit board. An SBC usually has as an operating system, processor, memory, and input-output functionality.
- PCB – printed circuit board: single- or multi-layered platform that connects electrical components or integrated circuits.
- ESC – electronic speed controller: a single-input single-output microcontroller that controls the speed of a brushless motor.
- PWM – pulse-width modulation/modulated: a technique used by a voltage source that uses only discrete voltage levels to supply a continuous average voltage.
- I2C – inter-integrated circuit: a communication protocol used by microcontrollers and peripheral integrated circuits (sensors, hardware devices, etc).
- IMU – inertial measurement unit: a sensor that measures acceleration, rotation, and magnetic field.
- DMP – digital motion processor: proprietary sensor fusion hardware unit on Invensense IMUs.

- SDK – software development kit: library or code base, often associated with a hardware device, that is used to build applications.

Hardware Components

(a) DJI F330 quadrotor frame with universal landing gear

(b) Floureon 2200 mAh 3-cell Lipo battery

(c) Invensense MPU-9250 IMU

(d) 3DR PX4Flow optical flow camera and sonar

(e) Intel Realsense R200 depth camera

(f) UP-board SBC running Ubilinux operating system

(g) LB-LINK Wireless USB adapter

(h) NXP Semiconductors PCA-9685 PWM driver

(i) RCmall 20-ampere ESCs

(j) SunnySky 2212-13 980kv brushless motors

(k) 8x4.5 inch ABS plastic propellers

(l) 3D-printed mounts for hardware

Software Libraries

(a) I2Cdevlib – library for interfacing with I2C devices (also configures DMP)

(b) librealsense – open-source SDK for Intel Realsense cameras

(c) libupm-pca9685 – software driver for PCA-9685 PWM driver

(d) libmraa – linux library that implements I2C protocol

Appendix C

Discretized Extended Kalman Filter Review

Let $f : \mathbb{R}^n \to \mathbb{R}^n$ and consider the set of nonlinear differential equations in state space form, that is

$$\dot{x} = f(x, w)$$

where $x \in \mathbb{R}^n$ is the state, and $w \in \mathbb{R}^p$ is the zero-mean process noise vector. Let $T_\mathrm{s} > 0$ be the sample time. For all $k \in \mathbb{N} \triangleq \{0, 1, 2, ...\}$, define $x_k \triangleq x(kT_\mathrm{s})$. Let $h : \mathbb{R}^n \to \mathbb{R}^m$ be the output mapping, and define the measured output $\forall k \in \mathbb{N}$

$$z_k \triangleq h(x_k) + v_k,$$

where $v_k \in \mathbb{R}^m$ is the sensor noise vector. Let $\hat{x}_k$ be our estimate of x_k and let

$$F_k \triangleq \left.\frac{\partial f}{\partial x}\right|_{(x,w)=(\hat{x}_k,0)}, \quad G_k \triangleq \left.\frac{\partial f}{\partial w}\right|_{(x,w)=(\hat{x}_k,0)}, \quad H_k \triangleq \left.\frac{\partial h}{\partial x}\right|_{x=\hat{x}_k}$$

be the state transition matrix, process noise transition matrix, and measurement matrix respectively. Let the discrete state transition matrix be

$$\Phi_k \triangleq I + F_k T_\mathrm{s}, \tag{6.7}$$

where I is the $n \times n$ identity matrix. Let the process noise variance matrix be $W \triangleq \text{diag}(\sigma_1^2, ..., \sigma_p^2)$, and compute the discrete process noise matrix

$$Q_k = \int_0^{T_\text{s}} (I + F_k t) G_k W G_k^\text{T} (I + F_k t)^\text{T} dt. \tag{6.8}$$

We define the sensor noise variance matrix $R \triangleq \text{diag}(\kappa_1^2, ..., \kappa_m^2)$ and compute the Kalman gain K_k as follows

$$M_k = \Phi_k P_k \Phi_k^\text{T} + Q_k, \tag{6.9}$$

$$K_k = M_k H_k^\text{T} (H_k M_k H_k^\text{T} + R)^{-1}. \tag{6.10}$$

The new state estimate $\hat{x}_{k+1}$ and covariance P_{k+1} are obtained using the equations

$$\bar{x}_{k+1} = \hat{x}_k + f(\hat{x}_k, 0) T_\text{s}, \tag{6.11}$$

$$\hat{x}_{k+1} = \bar{x}_{k+1} + K_k(\varkappa_{k+1} - h(\bar{x}_{k+1})), \tag{6.12}$$

$$P_{k+1} = (I - K_k H_k) M_k. \tag{6.13}$$

Equations (6.9)–(6.13) are iterative and values for the initial state estimate $\hat{x}_0$ and initial covariance P_0 must be chosen.

Appendix D

flight.cpp

```
////////////////////////////////////////////////////////////////////////////////
// This file contains guidance, navigation, and control code implementing
    ↪ the methods described in the thesis
// "Autonomous Quadrotor Collision Avoidance and Destination Seeking in a
    ↪ GPS-Denied Environment"
// by Thomas Kirven
////////////////////////////////////////////////////////////////////////////////

/** ---- Information ---- **/
/*  - Most data structures and variables are defined in flight.h */
/*  - The ones defined here are parameters that affect the */
/*    dynamics and control, as well as hardware instances */

#include "flight.h" // flight.h contains all other #inlcudes, data structure
    ↪  inits, and functions

/** ---- Actuator, Guidance, and Control Configuration ---- **/
/*  - define PWM to idle props and fly */
/*  - comment out PWM to disable propellers (sensors and control computations
    ↪  still run)*/
/*  - define FLOW to enable optical flow and velocity control */
/*  - comment out FLOW to track zero attitude and altitude command z_g */
/*  - define REALSENSE to use obstacle avoidance */
#define PWM
#define FLOW
#define REALSENSE

#ifdef PWM
#define IDLE
#define FLY
#endif

#ifdef FLOW
#define VELOCITY_CTRL
#define DEST_SEEK // enable destination seeking, set x_g,y_g,z_g as
   ↪ destination
//#define FREE_FLY // enable keyboard control
               // i,k,j,l - commands velocity forward back left and right
               // z - zeros the velocity command
#endif

#ifdef REALSENSE
#define AVOIDANCE // comment out to remove avoidance velocity, but still
   ↪ record positions of obstacles
#endif

/*  -- enable/disable Integral action -- */
/*  - define VIC to use velocity integral control */
/*  - define AIC to use Euler angle integral control */
/*  - define RIC to use rate integral control */
```

```
//#define VIC
//#define AIC
//#define RIC

/*  -- Adaptive adjustments -- */
/*  Recommend use with zero velocity command and constant height command */
/*  - define ADPT_K to adapt thrust constant k based on e_w */
/*  - define ADPT_ATT to adapt reference quaternion based on e_u and e_v */
//#define ADPT_K
//#define ADPT_ATT

float g2r=.012383; // raw gyro units to radians (multiplier for 1000 deg/s)
//float g2r=.024766; // raw gyro units to radians (multiplier for 2000 deg/s
    ↪ )
float a2si=0.001133; // raw accel to m/s^2 (multiplier)

int main() {
  // Bash script which checks if this program (flight.cpp) is running
  // - if not, it execute 'kpwm' which shuts down the motors
  printf("Starting safe_check...\n");
  system("./safe_check &");

  // set up MPU9150/9250 IMU
  MPU9150 mpu=MPU9150();
  mpu.initialize();

  devStatus = mpu.dmpInitialize();

  if (devStatus == 0) {
    mpu.setDMPEnabled(true); // enable onboard motion processor
    mpuIntStatus = mpu.getIntStatus();

    // set our DMP Ready flag so the main while loop knows it’s okay to use
        ↪ it
    printf("DMP ready! Waiting for interrupt...");
    dmpReady = true;

    // get expected DMP packet size for later comparison
    packetSize = mpu.dmpGetFIFOPacketSize();
    printf("packetSize = %d \n",packetSize);

  } else {
    // ERROR!
    // 1 = initial memory load failed
    // 2 = DMP configuration updates failed
    // (if it’s going to break, usually the code will be 1)
    printf("DMP Initialization failed (code %d ) \n",devStatus);
  }

  // set up Realsense R200
#ifdef REALSENSE
  rs::log_to_console(rs::log_severity::warn);
  //rs::log_to_file(rs::log_severity::debug, "librealsense.log");

  rs::context ctx;
  if(ctx.get_device_count() == 0) throw std::runtime_error("No device
      ↪ detected. Is it plugged in?");
  rs::device & dev = *ctx.get_device(0);

  dev.enable_stream(rs::stream::depth, rs::preset::best_quality);
  dev.enable_stream(rs::stream::color, rs::preset::best_quality);
  dev.enable_stream(rs::stream::infrared, rs::preset::best_quality);
  try { dev.enable_stream(rs::stream::infrared2, rs::preset::best_quality);
      ↪ } catch(...) {}
  dev.start();

  // preset 5 minimizes the number of false positives (depth data where
      ↪ there is no object)
  apply_depth_control_preset(&dev,5);

  const rs::intrinsics depth_intrin = dev.get_stream_intrinsics(rs::stream::
      ↪ depth);
#endif

  /* Initial values */
  px[0]=10; py[0]=10; pz[0]=0;dist[0]=10;

  x_e[0]=0; y_e[0]=0; z_e[0]=0;
  z_o[0]=0;
  u_e[0]=0; v_e[0]=0; w_e[0]=0; // 1st intermediate frame velocities
  ud[0]=0; vd[0]=0; wd[0]=0; // 1st intermediate frame accelerations
  udb[0]=0; vdb[0]=0; wdb[0]=0; // body accelerations

  phi[0]=0; theta[0]=0; psi[0]=0; // Euler angles
  phid=0; thetad=0; psid=0; // Euler ange rates
  p[0]=0; q[0]=0; r[0]=0; // body rates
```

```
pd[0]=0; qd[0]=0; rd[0]=0; // rate derivatives

e_u[0]=0; e_v[0]=0; e_w[0]=0; // velocity errors
e_phi[0]=0; e_the[0]=0; e_psi[0]=0; // angle errors
e_p[0]=0; e_q[0]=0; e_r[0]=0; // rate errors

Ie_u[0]=0; Ie_v[0]=0; Ie_w[0]=0; // integral of velocity errors
Ie_ph[0]=0; Ie_th[0]=0; Ie_ps[0]=0; // integral of angle errors
Ie_p[0]=0; Ie_q[0]=0; Ie_r[0]=0; // integral of rate errors

alw=-10.0; alwd=.1;

psi_d=0; // desired yaw

Time[0]=0; // initial time
sample_freq[0]=50;

// reference model derivatives (previous value holders)
ud_cp=0; vd_cp=0; wd_cp=0; psid_cp=0;
u2_cp=0; v2_cp=0; w2_cp=0; psi2_cp=0;
u3_cp=0; v3_cp=0; w3_cp=0; psi3_cp=0;
u4_cp=0; v4_cp=0; w4_cp=0;

// refrence model commands
u_c[0]=0; v_c[0]=0; w_c[0]=0; psi_c[0]=0;
ud_c=0; vd_c=0; wd_c=0; psid_c=0;
u2_c=0; v2_c=0; w2_c=0; psi2_c=0;
u3_c=0; v3_c=0; w3_c=0; psi3_c=0;
```

```
// Kalman filter initializations
// acceleration
ph_k.setZero(); Q_k.setZero(); P_k.setZero(); R_k.setZero(); C_k.setZero()
    ↪ ;
P_k(0,0)=.01; P_k(1,1)=.01; P_k(2,2)=.01; // initial covariance
K_k.setZero();
x_k.setZero();
x_k(0)=.01; x_k(1)=.01; x_k(2)=.01; // initial state

R_k(0,0)=1; R_k(1,1)=1; R_k(2,2)=1; // accel sensor noise matrix
C_k(0,0) = 1; C_k(1,1) = 1; C_k(2,2) = 1;

// process noise variances (gyro std dev^2)
float phdv=.01*.01;
float thdv=.01*.01;
float psdv=.01*.01;

// velocity
x.setZero(); y.setZero();
A.setZero(); _Q.setZero();
A = matrix::eye<float,4>(); B.setZero();
C.setZero(); R.setZero(); _P.setZero();
S_I.setZero(); K.setZero();
_P(0,0) = .01; _P(1,1) = .01; _P(2,2) = .01; _P(3,3) = .01;
x(0)=.01; x(1)=.01; x(2)=.01; x(3)=.01;

C(0,0) = 1; C(1,1) = 1; C(2,3) = 1; // measurement matrix
_Q(0,0)=0.00001; _Q(1,1)=0.00001; _Q(2,2)=.05; _Q(3,3)=.0005; // process
    ↪ noise matrix
```

```
  R(2,2) = .01;

  Om.setZero();
  V_q.setZero();
  V_qI.setZero();
  V_qIp.setZero();

  // set up keyboard io
  struct termios ctty;

  tcgetattr(fileno(stdout), &ctty);
  ctty.c_lflag &= ~(ICANON);
  tcsetattr(fileno(stdout), TCSANOW, &ctty);

    // set up flow module on i2c-1
#ifdef FLOW
  mraa::I2c* flow;
  flow = new mraa::I2c(0);
  flow->address(0x42);
#endif

  // set up pwm driver
  int freq=500; // maximum frequency 500 hz
  float per=1000000/freq; // pwm period in microseconds
  int high=(int)(2000/per*4095+.5); // integer pulse value cooresponding to
      ↪ 2 ms
  int low=(int)(650/per*4095+.5); // integer pulse value cooresponding to
      ↪ .65 ms
  int inc=(int)((high-low)/200+.5); // pulse width increment
  float wmin=173.0; // min prop rate (rad/s)
  float wmax=911.0; // max prop rate (rad/s)

  int pulse=low;
  int min1=low; int min2=low; int min3=low; int min4=low;
  int curPwm=15;

  // set up PCA9685 driver on i2c-0
#ifdef PWM
  upm::PCA9685 *pwm = new upm::PCA9685(0,0x40);

  pwm->setModeSleep(true);
  pwm->setPrescaleFromHz(freq);
  pwm->setModeSleep(false);
  usleep(50000);

  printf("ESC frequency set to %d Hz \n",freq);

  pwm->ledOnTime(PCA9685_ALL_LED, 0); // pulse on
  pwm->ledOffTime(PCA9685_ALL_LED, low); // pulse off

  printf("c - Calibrate ESC's\n");
  printf("s - use saved EPROM calibration\n\n");

  // calibration procedure
  while(!cal) {
    if ((kcom = getUserChar()) != 0) {
```

```
   switch (kcom) {

   case 'c':
printf("1) Disconnect ESC's from battery\n");
printf("2) Key h to set pulsewidth to high (1.999 ms)\n");
printf("3) Reconnect ESC's to battery\n");
printf("4) Key l to set pulsewidth to low (.65 ms)\n");
printf("5) Key i to increment pulswidth until motors start to spin\n");
printf("6) Key x to save effective pulsewidth range\n");
break;

   case 's':
cal=true;
min1=1445; min2=1425; min3=1475; min4=1475;
printf(" - using EPROM saved throttle range \n");
    printf("min1=%d; min2=%d; min3=%d; min4=%d; \n",min1,min2,min3,min4);
printf(" - done \n");
    break;

   case 'h' :
incr=true;
pulse=high-1; // only works when slightly < 2.0 ms
printf(" - setting high (%f ms) \n",(float)pulse/4095.0*per/1000);
break;

   case 'l' :
    incr=true;
    pulse=low;
```

```
     printf(" - setting low (%f ms) \n",(float)pulse/4095.0*per/1000);
     break;

   case 'i' :
incr=true;
pulse=pulse+inc;
if (pulse>high) pulse=high;
printf(" - incremented to %f ms pulse \n",(float)pulse/4095.0*per/1000);
break;

   case 'd' :
     incr=true;
     pulse=pulse-inc;
if (pulse<low) pulse=low;
     printf(" - decremented to %f ms pulse \n",(float)pulse/4095.0*per
         ↪ /1000);
     break;

   case 'q' :
     incr=true;
     inc=inc+1;
printf(" - increment value changed to %d\n",inc);
break;

   case 'a' :
     incr=true;
      inc=inc-1;
if (inc<=0) inc=1;
     printf(" - increment value changed to %d\n",inc);
     break;

   case 'w':
min1=pulse;
pwm->ledOffTime(curPwm, low);
curPwm=1;
break;

   case 'e':
min2=pulse;
pwm->ledOffTime(curPwm, low);
curPwm=14;
break;

   case 'r':
min3=pulse;
pwm->ledOffTime(curPwm, low);
curPwm=0;
break;

   case 'x' :
min4=pulse;
pwm->ledOffTime(curPwm, low);
printf("min1=%d; min2=%d; min3=%d; min4=%d; \n",min1,min2,min3,min4);
cal=true;
break;
   }

   if (incr) {
```

```
   pwm->ledOffTime(curPwm,pulse);
   // uncomment to calibrate ESC's
/* pwm->ledOffTime(0,pulse);
   pwm->ledOffTime(1,pulse);
   pwm->ledOffTime(14,pulse);
   pwm->ledOffTime(15,pulse);
*/
   incr=false;
      }
    }
  }
#endif

  float rng=high-max(max(min1,min2),max(min3,min4)); // effective throttle
      ↪ range
  //printf("range = %f \n",rng);

  // physical parameters
  float m = 1.085; // mass of quadrotor 1.085 kg w/2200mah lip0 and 1.01 kg
      ↪ w/1300 mah lipo
  //m = 1.01;
  float g = 9.80665; // magnitude of acceleration due to gravity
  float ll = .1185; // moment arm
  float ls = .1185;
  float k = 7.2*pow(10,-6); // (7.5-6.0)*E-6 8045 thrust constant (wi^.5)
  kP[0]=k; // initial k
  float b = 1.4*pow(10,-7); // (7.0-10.0*E-8) 8045 // propeller drag
      ↪ constant (wi^.5)
  bP[0]=b; // initial b

  // principle inertias
  float Ixx=.00585; // .00585
  float Iyy=.00575; // .0054
  float Izz=.0072; // .01 .. .008
  float Ip=.00002; // .00002 propeller lengthwise inertia

  // without realsense
  /*
  Ixx=.00561; // .00533
  Iyy=.00551; // .0049
  Izz=.0066; // .00655
*/
  // velocity gains
  float ku=3.0;
  float kv=3.0;
  float kw=2.5;
  float kiw=1.0;

  // attitude gains
  float kphi=7.0;
  float ktheta=7.0;
  float kpsi=7.0;

  // rate gains
  float kp=7.0;
  float kq=7.0;
  float kr=7.0;
  // integral gains are zero unless integral controls are defined
  float kiu=0; float kiv=0;
  float kiph=0; float kith=0; float kips=0;
  float kip=0; float kiq=0; float kir=0;

#ifdef VIC // velocity integral control
  kiu=0.8;
  kiv=0.8;
#endif

#ifdef AIC // angle integral control
  kiph=3.5;
  kith=1.5;
  kips=1.5;
#endif

#ifdef RIC // rate integral control
  kip=7.0;
  kiq=7.0;
  kir=7.0;
#endif

  // reference model parameters
  float a0=625;
  float a1=500;
  float a2=150;
  float a3=20; // p=-5
```

```
// reference velocity command parameters // pole location
//a0=28561; a1=8788; a2=1014; a3=52; // p=-13
//a0=2401; a1=1372; a2=294; a3=28; //p=-7

// reference angle command for yaw
float b0=27;
float b1=27;
float b2=9; // p=-6

float sd=.8; // set speed of quadcopter
st_d=1.0; // stopping distance

// set destination p_g(0)
float x_g=0;
float y_g=-.2;
float z_g=-1.0;

// put destination in vector format
P_g(0) = x_g;
P_g(1) = y_g;
P_g(2) = z_g;

p_gn = P_g.norm();

// inner and outer ellipsoid semi-principles axes
float ai=ll*4.5; float bi=ll*2.8; float ci=ll*2.0;
float ao=2.0*ai; float bo=2.0*bi; float co=2.0*ci;

printf("outer ellipsoid semi principle axes: %f, %f, %f \n",ao,bo,co);
printf("inner ellipsoid semi principle axes: %f, %f, %f \n",ai,bi,ci);

// Ellipsoid radii and potential
float R_o,R_i,U_e;

float ep=1.25; // maximum potential

float d_min=2; // closest approach to obstacle (meters)
bool col=false; // collision boolean
int cnt = 0; // counter for position inside outer ellipsoids

pts.setAll(100); // initialize obstacle positions to > 100 meters
P_o.setZero();

// initialize data as a precaution
u_d[0]=0; v_d[0]=0; w_d[0]=0;

t1[0]=0; t2[0]=0; t3[0]=0;
T[0]=m*g;

om1[0]=T[0]/(4.0*k);
om2[0]=T[0]/(4.0*k);
om3[0]=T[0]/(4.0*k);
om4[0]=T[0]/(4.0*k);

om1[0]=sqrt(om1[0]); om2[0]=sqrt(om2[0]);
om3[0]=sqrt(om3[0]); om4[0]=sqrt(om4[0]);

Hx[0]=Ixx*p[0]; Hy[0]=Iyy*q[0]; Hz[0]=Izz*r[0];
```

```
Hp[0]=(om1[0]+om3[0]-om2[0]-om4[0])*Ip;

// set up loop timing
auto prev = std::chrono::high_resolution_clock::now();
auto now = std::chrono::high_resolution_clock::now();
long long microseconds = std::chrono::duration_cast<std::chrono::
    ↪ microseconds>(now-prev).count();

// rest orientation
float phi_r=0; float theta_r=0; float psi_r=0;
float arx=0; float ary=0; float arz=0;
// let dmp settle and get reference orientaion (refQ)
int acnt=0;
for (int j=0; j<1000; j++) {
  // reset interrupt flag and get INT_STATUS byte
  mpuInterrupt = false;
  mpuIntStatus = mpu.getIntStatus();

  // get current FIFO count
  fifoCount = mpu.getFIFOCount();

  // check for overflow (this should never happen unless our code is too
      ↪ inefficient)
  if ((mpuIntStatus & 0x10) || fifoCount == 1024) {
    // reset so we can continue cleanly
    mpu.resetFIFO();
    printf("fifo overflow!");
    // otherwise, check for DMP data ready interrupt (this should happen
        ↪ frequently)
  } else if (mpuIntStatus & 0x02) {
  // wait for correct available data length, should be a VERY short wait
    while (fifoCount < packetSize) fifoCount = mpu.getFIFOCount();

    // read a packet from FIFO
    mpu.getFIFOBytes(fifoBuffer, packetSize);

    // track FIFO count here in case there is > 1 packet available
    // (this lets us immediately read more without waiting for an interrupt
        ↪ )
    fifoCount -= packetSize;
    mpu.dmpGetQuaternion(&refQ,fifoBuffer); // obviously only last refQ is
        ↪ saved
    refQ.normalize();
    corrQ = (refQ.getConjugate()).getProduct(refQ);
    corrQ.normalize();

    if (j>699) {
      mpu.dmpGetEuler(ypr_r, &refQ);
 phi_r=phi_r+ypr_r[2];
 theta_r=theta_r+ypr_r[1];
 psi_r=psi_r+ypr_r[0];

 mpu.dmpGetAccel(&ar,fifoBuffer);
 mpu.dmpGetGravity(&gravity, &corrQ);
      mpu.dmpGetLinearAccel(&aaReal, &ar, &gravity);
 arx=arx+aaReal.x;

 ary=ary+aaReal.y;

 arz=arz+aaReal.z;

 acnt=acnt+1;

    }

  }

}

// get static accel and reference orientation

arx=arx/acnt;

ary=ary/acnt;

arz=arz/acnt;

phi_r=phi_r/acnt;

theta_r=theta_r/acnt;

psi_r=psi_r/acnt;


ypr_r[0]=psi_r;

ypr_r[1]=theta_r;

ypr_r[2]=phi_r;


printf("IMU orientation on ground : \n roll = %f pitch = %f yaw = %f \n",

  ypr_r[2],-ypr_r[1],-ypr_r[0]);


// hard code reference quaternion (orientation)

ypr_r[1]=0.01; // pitch = -0.013426 (switch sign)

ypr_r[2]=-.08; // roll = -0.082126

refQ=fromEuler(ypr_r);

refQ.normalize();


// Idle propellers at 1/15th full throttle
```

```
#ifdef IDLE
  pwm->ledOffTime(0,min1+rng/15);
  pwm->ledOffTime(1,min2+rng/15);
  pwm->ledOffTime(14,min3+rng/15);
  pwm->ledOffTime(15,min4+rng/15);
#endif

  // check DMP settings
  uint8_t rate=mpu.getRate();
  printf("sample rate set to %d hz\n",1000/(1+rate));
  uint8_t fsync=mpu.getExternalFrameSync();
  printf("Fsync config value set to %d \n",fsync);
  uint8_t mode=mpu.getDLPFMode();
  printf("DLPF mode %d \n",mode);

  // get takeoff command
  bool takeoff=false;
  while (!takeoff) {
    std::cout.flush();
    printf("Takeoff? y/n: \n");
    switch (tolower(getchar())) {
    case 'y' :
      takeoff=true;
      break;

    case 'n' :
#ifdef PWM
      setAllPWM(pwm,low);
#endif
```

```
#ifdef FLOW
      delete flow;
#endif
      printf(" - Exiting. \n");
      exit(1);
      break;
    }
  }
  prev = std::chrono::high_resolution_clock::now(); // start time

  while (1) {

    // reset interrupt flag and get INT_STATUS byte (resets after read)
    mpuInterrupt = false;
    mpuIntStatus = mpu.getIntStatus();

    // get current FIFO count
    fifoCount = mpu.getFIFOCount();

    // check for overflow (this should never happen unless our code is too
        ↪ inefficient)
    if ((mpuIntStatus & 0x10) || fifoCount == 1024) {
      mpu.resetFIFO(); // reset so we can continue cleanly
      printf("fifo overflow!");
      // otherwise, check for DMP data ready interrupt (this should happen
          ↪ frequently)
    } else if (mpuIntStatus & 0x02) {
      //printf("FFdmpIntStatus = %X\n",dmpIntStatus);
      //printf("FFmpuIntStatus = %X\n",mpuIntStatus);
```

```
   // wait for correct available data length, should be a VERY short wait
   while (fifoCount < packetSize) fifoCount = mpu.getFIFOCount();
   mpu.getFIFOBytes(fifoBuffer, packetSize); // read a packet from FIFO

   // track FIFO count here in case there is > 1 packet available
   // (this lets us immediately read more without waiting for an interrupt
      ↪ )
   fifoCount -= packetSize;

   mpu.dmpGetQuaternion(&Q,fifoBuffer);
   mpu.dmpGetGyro(gyro,fifoBuffer);

   now = std::chrono::high_resolution_clock::now();
   microseconds = std::chrono::duration_cast<std::chrono::microseconds>(
      ↪ now-prev).count();
   dt=(float)(microseconds)/1000000.0; // loop time (s)
   prev = now;
   Time[n+1] = Time[n]+dt;
   sample_freq[n+1]=1.0/dt;

   // quaternion with respect to reference quaternion
   corrQ = (refQ.getConjugate()).getProduct(Q);
   corrQ.normalize();

   /*
mpu.dmpGetYawPitchRoll(ypr,&corrQ,&gravity);
phi[n+1] = ypr[2];
theta[n+1] = ypr[1];
psi[n+1] = ypr[0];
```

```
*/

// convert quaternion to Euler angles
mpu.dmpGetEuler(ypr, &corrQ);
phi[n+1] = ypr[2];
theta[n+1] = -ypr[1];
psi[n+1] = -ypr[0];

// define current trig functions of Euler angles to save space later on
cph = cos(phi[n+1]); sph = sin(phi[n+1]); tph = tan(phi[n+1]);
cth = cos(theta[n+1]); sth = sin(theta[n+1]); tth = tan(theta[n+1]);
cps = cos(psi[n+1]); sps = sin(psi[n+1]);

// angular velocity (seem to have constant bias which are subtracted
   ↪ below)
p[n+1] = gyro[0]*g2r+.034;
q[n+1] = -gyro[1]*g2r-.0185;
r[n+1] = -gyro[2]*g2r-.0062;

// Euler rate kinematics
phid = p[n+1]+q[n+1]*sph*tth+r[n+1]*cph*tth;
thetad = q[n+1]*cph-r[n+1]*sph;
psid = q[n+1]*sph/cth+r[n+1]*cph/cth;

// angular momentum
//Hx[n+1]=Ixx*p[n+1]; Hy[n+1]=Iyy*q[n+1]; Hz[n+1]=Izz*r[n+1];

// if roll or pitch is too large, shut off propellers and write data
if ((abs(phi[n+1])>1.3)||(abs(theta[n+1])>1.3)) {
```

```
#ifdef PWM
   setAllPWM(pwm,low);
#endif

#ifdef FLOW
   delete flow;
#endif
   printf("FLIP\n");
   printf("phi[n+1] = %f \t theta[n+1] = %f\n",phi[n+1],theta[n+1]);
   printf("Closest object approach: %f \n",d_min);

   write_Params(n,Time,kP,bP);
   write_Map(i,pts);

   // write data to .txt files for gnuplot
   write_Freq(n,Time,sample_freq);
   write_Position(n,Time,x_e,y_e,z_e,z_o);
   write_Velocity(n,Time,u_e,v_e,w_e,u_c,v_c,w_c);
   //write_Vel_Int(n,Time,Ie_u,Ie_v,Ie_w);
   write_Vel_Int(n,Time,u_d,v_d,w_d);
   write_Acceleration(n,Time,ud,vd,wd);
   write_Angles(n+1,Time,phi,theta,psi,phi_d,theta_d,psi_c);
   write_Angle_Errors(n,Time,e_phi,e_the,e_psi);
   write_Rates(n,Time,p,q,r,p_d,q_d,r_d);
   write_Rate_Errors(n,Time,e_p,e_q,e_r);
   write_Momentum(n,Time,Hx,Hy,Hz,Hp);
   write_Inputs(n,Time,T,t1,t2,t3,om1,om2,om3,om4);
   write_Pos_Obj(n,Time,px,py,pz,dist);
```

```
  exit(1);
     }

#ifdef REALSENSE
     // get closest obstacle point (px,py,pz) from current frame
     if(dev.is_streaming()) dev.wait_for_frames();
     auto points = reinterpret_cast<const rs::float3 *>(dev.get_frame_data(
         ↪ rs::stream::points));
     //auto depth = reinterpret_cast<const uint16_t *>(dev.get_frame_data(rs
         ↪ ::stream::depth));
     min_dist = 30;
     for(int yy=0; yy<depth_intrin.height; ++yy) {
  for(int xx=0; xx<depth_intrin.width; ++xx) {
    if (points->z){
     dist[n+1] = points->x*points->x+points->y*points->y+points->z*points->
         ↪ z;
     if (dist[n+1]<min_dist) {
       min_dist = dist[n+1];
       px[n+1] = points->x;
       py[n+1] = points->y;
       pz[n+1] = points->z;
     }
    }
    ++points;
  }
     }

     dist[n+1] = sqrt(min_dist); // distance to (px,py,pz)
```

```
#endif

      // get linear acceleration
      mpu.dmpGetAccel(&aa,fifoBuffer);
      mpu.dmpGetGravity(&gravity, &corrQ);
      mpu.dmpGetLinearAccel(&aaReal, &aa, &gravity);
      aaReal.x=aaReal.x-arx;
      aaReal.y=aaReal.y+ary;
      aaReal.z=aaReal.z;
      mpu.dmpGetLinearAccelInWorld(&aaWorld, &aaReal, &corrQ);

      // convert acceleration to correct units
      ud[n+1] = (aaWorld.x)*a2si;
      vd[n+1] = -(aaWorld.y)*a2si;
      wd[n+1] = -(aaWorld.z)*a2si;

/** EKF for estimating acceleration **/

      // measurements for EKF
      y_k(0)=ud[n+1];
      y_k(1)=vd[n+1];
      y_k(2)=wd[n+1];

      // discrete transition matrix
      ph_k(0,0) = 1+alwd*dt/alw;
      ph_k(0,1) = -psid*dt;
      ph_k(0,2) = -kw*x_k(0)*dt/alw;

      ph_k(1,0) = psid*dt;
      ph_k(1,1) = 1+alwd*dt/alw;
      ph_k(1,2) = -kw*x_k(1)*dt/alw;

      ph_k(2,2) = 1-kw*dt;

      // discrete noise transition matrix
      Q_k(0,0)=(psid*(psid*(phdv*alw*alw*cps*cps+thdv*alw*alw*sps*sps+psdv*
          ↪ x_k(0)*x_k(0))+(alwd*(psdv*x_k(0)*x_k(1)+alw*alw*cps*phdv*sps-\\
          ↪ alw*alw*cps*sps*thdv))/alw)+(alwd*(psid*(psdv*x_k(0)*x_k(1) +
          ↪ alw*alw*cps*phdv*sps - alw*alw*cps*sps*thdv)+(alwd*(thdv*alw*alw
          ↪ *cps*cps+phdv*alw*alw*sps*sps+psdv*x_k(1)*x_k(1)))/alw))/alw)*dt
          ↪ *dt*dt/3+(2*psid*(psdv*x_k(0)*x_k(1)+alw*alw*cps*phdv*sps-alw*
          ↪ alw*cps*sps*thdv)+(2*alwd*(thdv*alw*alw*cps*cps+phdv*alw*alw*sps
          ↪ *sps+psdv*x_k(1)*x_k(1)))/alw)*dt*dt/2 + (thdv*alw*alw*cps*cps+
          ↪ phdv*alw*alw*sps*sps+psdv*x_k(1)*x_k(1))*dt;

      Q_k(0,1)=(psid*(psid*(psdv*x_k(0)*x_k(1)+alw*alw*cps*phdv*sps-alw*alw*
          ↪ cps*sps*thdv)+(alwd*(thdv*alw*alw*cps*cps+phdv*alw*alw*sps*sps+
          ↪ psdv*x_k(1)*x_k(1)))/alw)-(alwd*(psid*(phdv*alw*alw*cps*cps +
          ↪ thdv*alw*alw*sps*sps + psdv*x_k(0)*x_k(0))+(alwd*(psdv*x_k(0)*
          ↪ x_k(1)+alw*alw*cps*phdv*sps-alw*alw*cps*sps*thdv))/alw))/alw)*dt
          ↪ *dt*dt/3+(psid*(thdv*alw*alw*cps*cps+phdv*alw*alw*sps*sps+psdv*
          ↪ x_k(1)*x_k(1))-psid*(phdv*alw*alw*cps*cps+thdv*alw*alw*sps*sps+
          ↪ psdv*x_k(0)*x_k(0))-(2*alwd*(psdv*x_k(0)*x_k(1)+alw*alw*cps*phdv
          ↪ *sps-alw*alw*cps*sps*thdv))/alw)*dt*dt/2+(alw*alw*cps*sps*thdv-
          ↪ alw*alw*cps*phdv*sps-psdv*x_k(0)*x_k(1))*dt;

      Q_k(1,0)=(psid*(psid*(psdv*x_k(0)*x_k(1)+alw*alw*cps*phdv*sps-alw*alw*
          ↪ cps*sps*thdv)-(alwd*(phdv*alw*alw*cps*cps+thdv*alw*alw*sps*sps+
          ↪ psdv*x_k(0)*x_k(0)))/alw)+(alwd*(psid*(thdv*alw*alw*cps*cps +
          ↪ phdv*alw*alw*sps*sps + psdv*x_k(1)*x_k(1))-(alwd*(psdv*x_k(0)*
          ↪ x_k(1)+alw*alw*cps*phdv*sps-alw*alw*cps*sps*thdv))/alw))/alw)*dt
          ↪ *dt*dt/3+(psid*(thdv*alw*alw*cps*cps+phdv*alw*alw*sps*sps+psdv*
          ↪ x_k(1)*x_k(1))-psid*(phdv*alw*alw*cps*cps+thdv*alw*alw*sps*sps+
          ↪ psdv*x_k(0)*x_k(0))-(2*alwd*(psdv*x_k(0)*x_k(1)+alw*alw*cps*phdv
          ↪ *sps-alw*alw*cps*sps*thdv))/alw)*dt*dt/2+(alw*alw*cps*sps*thdv-
          ↪ alw*alw*cps*phdv*sps-psdv*x_k(0)*x_k(1))*dt;

      Q_k(1,1)=(psid*(psid*(thdv*alw*alw*cps*cps+phdv*alw*alw*sps*sps+psdv*
          ↪ x_k(1)*x_k(1))-(alwd*(psdv*x_k(0)*x_k(1)+alw*alw*cps*phdv*sps-
          ↪ alw*alw*cps*sps*thdv))/alw)-(alwd*(psid*(psdv*x_k(0)*x_k(1) +
          ↪ alw*alw*cps*phdv*sps - alw*alw*cps*sps*thdv)-(alwd*(phdv*alw*alw
          ↪ *cps*cps+thdv*alw*alw*sps*sps+psdv*x_k(0)*x_k(0)))/alw))/alw)*dt
          ↪ *dt*dt/3+((2*alwd*(phdv*alw*alw*cps*cps+thdv*alw*alw*sps*sps+
          ↪ psdv*x_k(0)*x_k(0)))/alw-2*psid*(psdv*x_k(0)*x_k(1)+alw*alw*cps*
          ↪ phdv*sps-alw*alw*cps*sps*thdv))*dt*dt/2+(phdv*alw*alw*cps*cps+
          ↪ thdv*alw*alw*sps*sps+psdv*x_k(0)*x_k(0))*dt;

      // compute Kalman gain
      P_k = ph_k*P_k*ph_k.transpose()+Q_k;
      SI_k = matrix::inv<float, 3>(C_k*P_k*C_k.transpose()+R_k);
      K_k = P_k*C_k.transpose()*SI_k; // Kalman gain

      // predict state
      x_k(0) = x_k(0) + (alwd*x_k(0)/alw-psid*x_k(1)+alw*(thetad*cps+phid*sps
          ↪ ))*dt;
      x_k(1) = x_k(1) + (alwd*x_k(1)/alw+psid*x_k(0)+alw*(thetad*sps-phid*cps
          ↪ ))*dt;
      x_k(2) = x_k(2) + alwd*dt;

      x_k += K_k*(y_k-x_k); // correct state
      P_k -= K_k*C_k*P_k; // update covariance

      // filter output
      ud[n+1]=x_k(0); // u dot
      vd[n+1]=x_k(1); // v dot
      wd[n+1]=x_k(2); // w dot

  /** Kalman filter for estimating velocity **/

      A(3,2) = dt; // discrete transition matrix (diagonal entries are always
         ↪ 1)

      // input matrix
      B(0,0)=dt;
      B(1,1)=dt;
      B(2,2)=dt;
      B(3,2)=dt*dt/2;

      // get optical flow data
  #ifdef FLOW
      flow->readBytesReg(0x00,frame,22);
      flow->readBytesReg(0x16,int_frame,26);
  #endif

      // sonar measurement of height
      z_e[n+1]=-(int16_t)(frame[20] | frame[21] << 8)/(1.0e3f);
      // if there's an instant change in z (there probably is an object below
          ↪ ), then estimate correct height
      if (abs(z_e[n+1]-z_e[n])>.28) z_e[n+1]=z_e[n]+w_e[n]*dt;

      h=-z_e[n+1]; // height

      // total measured optical flow
      flow_x_rad = (int16_t)(int_frame[2] | int_frame[3] << 8)*1.24f/1.0e4f;
      flow_y_rad = (int16_t)(int_frame[4] | int_frame[5] << 8)*1.24f/1.0e4f;

      // angular rotation since previous optical flow read
      gyro_x_rad = (int16_t)(int_frame[6] | int_frame[7] << 8)*1.24f/1.0e4f;
      gyro_y_rad = (int16_t)(int_frame[8] | int_frame[9] << 8)*1.24f/1.0e4f;

      // time since previous optical flow read
      dt_flow = (uint32_t)(int_frame[12] | int_frame[13] << 8 | int_frame[14]
          ↪  << 16 | int_frame[15] << 24)/1.0e6f;

      // if dt_flow is unreasonable use loop time
      if (dt_flow > 0.5f || dt_flow < 1.0e-2f) {
    dt_flow = dt;
      }

      // translational velocity measurements
      y(0) = (flow_y_rad - gyro_y_rad)*h/dt_flow;
      y(1) = -(flow_x_rad - gyro_x_rad)*h/dt_flow;

      // roll-pitch-yaw consecutive rotation (body to inertial)
```

```
O = matrix::Eulerf(phi[n+1],theta[n+1],psi[n+1]);
// estimate b3 direction velocity
y(2)=(u_e[n]+ud[n]*dt)*O(0,2)+(v_e[n]+vd[n]*dt)*O(1,2)+(w_e[n]+wd[n]*dt
    ↪ )*O(2,2);

V_q=y; // save quad velocity in body frame
V_qI = O * V_q; // inertial quad velocity
v=sqrt(y(0)*y(0)+y(1)*y(1)); // speed

y = V_qI; // measurements for Kalman filter

y(2)=z_e[n+1]; // height (with correct sign)

// limits on height and speed for stddev calculation
if (h > h_max) h = h_max;
if (h < h_min) h = h_min;
if (v > v_max) v = v_max;
if (v < v_min) v = v_min;

// optical flow standard dev. estimate
flow_vxy_stddev = P[0] * h + P[1] * h * h + P[2] * v + P[3] * v * h + P
    ↪ [4] * v * h * h;

// euler rate and body rate magnitudes squared
rot_sq = phi[n+1]*phi[n+1]+theta[n+1]*theta[n+1];
rotrate_sq = p[n+1]*p[n+1]+q[n+1]*q[n+1]+r[n+1]*r[n+1];

// sensor noise matrix
R(0,0) = flow_vxy_stddev * flow_vxy_stddev;//+7.0f*(rot_sq+rotrate_sq);
```

```
        R(1,1) = R(0,0);

        // compute Kalman gain
        _P = A*_P*A.transpose() + _Q;
        S_I = matrix::inv<float, 3>(C * _P * C.transpose() + R);
        K = _P * C.transpose() * S_I; // Kalman gain

        x = A * x + B * x_k; // predict state
        x += K * (y - C * x); // correct state
        _P -= K * C * _P; // update covariance

        // filter output
        u_e[n+1]=x(0); // u
        v_e[n+1]=x(1); // v
        w_e[n+1]=x(2); // w

        x_e[n+1] = x_e[n]+(V_qI(0)+V_qIp(0))*dt/2;
        y_e[n+1] = y_e[n]+(V_qI(1)+V_qIp(1))*dt/2;
        z_e[n+1]=x(3); // z (height)

        if (z_e[n+1]>-.3) z_e[n+1]=-.3; // if height estimate is bad, set z=.3
            ↪ meters

#ifdef DEST_SEEK
        // estimate destination using rotating reference frame technique
        /*
    Om(0,1) = -r[n+1];
    Om(0,2) = q[n+1];
    Om(1,0) = r[n+1];
    Om(1,2) = -p[n+1];

    Om(2,0) = -q[n+1];

    Om(2,1) = p[n+1];


    // enhance estimate of P_g using measured height

    P_gI = O * P_g; // body to inertial

    P_gI(2) = z_e[n+1] - z_g; // hard code height error since we have height
        ↪ measurement

    P_g = O.transpose() * P_gI; // put back into body frame


    // propogate P_g

    V_g = - V_q - Om*P_g;

    P_g = P_g + V_g*dt;


    // rotate new P_g into inertial frame for control

    P_gI = O * P_g;*/


        // destination vector relative to quadrotor

        P_gI(2) = z_e[n+1] - z_g;

        P_gI(0) = x_e[n+1] - x_g;

        P_gI(1) = y_e[n+1] - y_g;


        p_gn=P_gI.norm(); // distance to destination


        if (p_gn<.002||land) { // go straight down

    //land = true;

    sd=.8;

    P_gI(2)=-1;

        }
```

```
      // compute destination seeking velocity
      if (p_gn >= st_d) { // is distance to destination >= stopping distance
   u_d[n+1]=-sd*P_gI(0)/p_gn;
   v_d[n+1]=-sd*P_gI(1)/p_gn;
   w_d[n+1]=-sd*P_gI(2)/p_gn;
      } else { // distance to destination < stopping distance
   u_d[n+1]=-sd*P_gI(0)/st_d;
   v_d[n+1]=-sd*P_gI(1)/st_d;
   w_d[n+1]=-sd*P_gI(2)/st_d;
      }

#ifdef AVOIDANCE
      if (!land && avoid) {

   cnt=0; // reset counter for number of points in outer ellipsoid E_o

   // reset avoidance velocities
   u_a = 0;
   v_a = 0;
   w_a = 0;

   // update position of detected points relative to quadrotor
   for (int j=0; j<n_pts; j++) {
     pts(0,j) = pts(0,j)-(V_qI(0)+V_qIp(0))*dt/2;
     pts(1,j) = pts(1,j)-(V_qI(1)+V_qIp(1))*dt/2;
     pts(2,j) = pts(2,j)-(V_qI(2)+V_qIp(2))*dt/2;

     // set jth colum of pts matrix as current position vector
```

```
    P_oI(0) = pts(0,j);
    P_oI(1) = pts(1,j);
    P_oI(2) = pts(2,j);

    // position in the body frame
    P_o = O.transpose() * P_oI;

    //  is P_o inside E_o
    if (in_Elpsd(P_o,ao,bo,co)) {
      cnt++; // add to counter
      // get outer and inner ellipsoid radii along P_o
      R_o = ElpsdRad(P_o,ao,bo,co);
      R_i = ElpsdRad(P_o,ai,bi,ci);

      U_e = ep*(1-(P_o.norm() - R_i)/(R_o - R_i)); // compute potential of
          ↪ P_o
      //printf("U_e = %f\n",U_e);

      // sum avoidance velocities from all points
      u_a=u_a + U_e * P_oI(0)/P_oI.norm();
      v_a=v_a + U_e * P_oI(1)/P_oI.norm();
      w_a=w_a + U_e * P_oI(2)/P_oI.norm();
    }
  }
  // nearest position from most recent realsense frame
  P_o(0) = pz[n+1];
  P_o(1) = px[n+1];
  P_o(2) = py[n+1];
```

```
   if (!(P_o(0)==0||P_o(1)==0||P_o(2)==0)) { // is position data good?

P_oI = O * P_o; // rotate to inertial frame

// add to set of points (map)
pts(0,i) = P_oI(0);
pts(1,i) = P_oI(1);
pts(2,i) = P_oI(2);

if (++i == n_pts) i=0; // if number of points in map exceeds n_pts, then
    ↪ replace starting with oldest points

// is most recent P_o in E_o
if (in_Elpsd(P_o,ao,bo,co)) {
  cnt++;
  // get outer and inner ellipsoid radii along P_o
  R_o = ElpsdRad(P_o,ao,bo,co);
  R_i = ElpsdRad(P_o,ai,bi,ci);

  U_e = ep*(1-(P_o.norm() - R_i)/(R_o - R_i)); // compute potential of
      ↪ P_o
  //printf("U_e = %f\n",U_e);

  // add to avoidance velocities
  u_a=u_a + U_e * P_oI(0)/P_oI.norm();
  v_a=v_a + U_e * P_oI(1)/P_oI.norm();
  w_a=w_a + U_e * P_oI(2)/P_oI.norm();
}
   }
```

```
    // sometimes false positives show up, so make sure at least 12 points are
        ↪  in E_o before adding to desired velocity
    if (cnt>12) {
      float n_a = sqrt(u_a/cnt*u_a/cnt+v_a/cnt*v_a/cnt);
      if (n_a>1.5) {
              u_a=u_a/n_a;
         v_a=v_a/n_a;
      }
      u_d[n+1]=u_d[n+1] - u_a/cnt;
      v_d[n+1]=v_d[n+1] - v_a/cnt;
//    w_d[n+1]=w_d[n+1] - w_a/cnt;

      //printf("u_a = %f \n",-u_a/cnt);
      //printf("v_a = %f \n",-v_a/cnt);
      //printf("w_a = %f \n",-w_a/cnt);
    }
       }
#endif
#endif
       // remeber previous inertial u,v
       V_qIp(0) = V_qI(0);
       V_qIp(1) = V_qI(1);
       psi_d = 0; // set zero heading command

#ifdef FREE_FLY // fly with keyboard i,j,k,l keys

       P_gI(2) = z_e[n+1]-z_g; // relative k position of goal
       if (land) { // go straight down
    P_gI(2)=-1;
       }

       p_gn=abs(P_gI(2)); // distance to ground

       // compute w_d
       if (p_gn >= st_d) {
    w_d[n+1]=-sd*P_gI(2)/p_gn;
       } else {
    w_d[n+1]=-sd*P_gI(2)/st_d;
       }

       // keep previous desired velocities
       u_d[n+1]=u_d[n];
       v_d[n+1]=v_d[n];

#endif

       // put trajectories here
       if (traj) {
    // circle trajectory
    u_d[n+1]=1.0*cos(Time[n+1]*2.5);
    v_d[n+1]=1.0*sin(Time[n+1]*2.5);
       }

       // get key commands if any
       if ((kcom = getUserChar()) != 0) {
    switch (kcom) {
```

```
case 'a' : // avoid objects

  avoid = true;

  // set destination

  x_g = 10.0;

  //y_g = -1.0;

  break;


case 's' : // begin trajectory

  traj=true;

  break;


case 'f' : // yaw to 90 degrees

  psi_d=pi/2;

  break;


case 'd': // yaw to -90 degrees

  psi_d=-pi/2;

  break;


case ' ' : // land (space bar)

  kP[0]=k;

  traj=false;

  avoid=false;

  u_d[n+1]=0;

  v_d[n+1]=0;

  x_g = x_e[n+1];

  y_g = y_e[n+1];

  land=true;
  sd=.6;

  z_g = 1;

  break;


  // i,j,k,l -- sets u_d,-v_d,-u_d,v_d to .5 m/s respectively

  // must press z to set u_d,v_d back to zero

case 'l' :

  v_d[n+1]=2.0;

  break;


case 'j' :

  v_d[n+1]=-2.0;

  break;


case 'i' :

  u_d[n+1]=2.0;

  break;


case 'k' :

  u_d[n+1]=-2.0;

  break;


case 'z' : // zero the desired velocity and stop trajectory

  avoid = false;

  traj=false;

  u_d[n+1]=0;

  v_d[n+1]=0;

  //psi_d=0;

  break;
```

```
   case 'x' : // shutdown motors, write data, exit program
#ifdef PWM
     setAllPWM(pwm,low);
#endif

#ifdef FLOW
     delete flow;
#endif

     if (col) printf("Collision occured\n");
     printf("Closest object approach: %f \n",d_min);

     write_Params(n,Time,kP,bP);

     // get points relative to quadrotor's final position (for overlay with
         ↪ quadrotor trajectory)
     for (int i=0; i<n_pts; i++) {
       pts(0,i) = pts(0,i)+x_e[n];
       pts(1,i) = pts(1,i)+y_e[n];
       pts(2,i) = -pts(2,i)-z_e[n];
     }

     write_Map(i,pts);

     // write data to .txt files for gnuplot
     write_Freq(n,Time,sample_freq);
     write_Position(n,Time,x_e,y_e,z_e,z_o);
     write_Velocity(n,Time,u_e,v_e,w_e,u_c,v_c,w_c);
```

```
  //write_Vel_Int(n,Time,Ie_u,Ie_v,Ie_w);
  write_Vel_Int(n,Time,u_d,v_d,w_d);
  write_Acceleration(n,Time,ud,vd,wd);
  write_Angles(n+1,Time,phi,theta,psi,phi_d,theta_d,psi_c);
  write_Angle_Errors(n,Time,e_phi,e_the,e_psi);
  write_Rates(n,Time,p,q,r,p_d,q_d,r_d);
  write_Rate_Errors(n,Time,e_p,e_q,e_r);
  write_Momentum(n,Time,Hx,Hy,Hz,Hp);
  write_Inputs(n,Time,T,t1,t2,t3,om1,om2,om3,om4);
  write_Pos_Obj(n,Time,px,py,pz,dist);

  exit(1);
  break;
}
   }

   // generate reference commands
   u_c[n+1]=u_c[n]+ud_cp*dt;
   ud_c=ud_cp+u2_cp*dt;
   u2_c=u2_cp+u3_cp*dt;
   u3_c=u3_cp+u4_cp*dt;
   u4_c=-a3*u3_c-a2*u2_c-a1*ud_c-a0*(u_c[n+1]-u_d[n+1]);

   // set previous reference commands (used for solving refrence model
       ↪ with Euler method)
   ud_cp=ud_c; u2_cp=u2_c; u3_cp=u3_c; u4_cp=u4_c;

   v_c[n+1]=v_c[n]+vd_cp*dt;
   vd_c=vd_cp+v2_cp*dt;
```

```
v2_c=v2_cp+v3_cp*dt;

v3_c=v3_cp+v4_cp*dt;

v4_c=-a3*v3_c-a2*v2_c-a1*vd_c-a0*(v_c[n+1]-v_d[n+1]);


vd_cp=vd_c; v2_cp=v2_c; v3_cp=v3_c; v4_cp=v4_c;


w_c[n+1]=w_c[n]+wd_cp*dt;

wd_c=wd_cp+w2_cp*dt;

w2_c=w2_cp+w3_cp*dt;

w3_c=w3_cp+w4_cp*dt;

w4_c=-a3*w3_c-a2*w2_c-a1*wd_c-a0*(w_c[n+1]-w_d[n+1]);


wd_cp=wd_c; w2_cp=w2_c; w3_cp=w3_c; w4_cp=w4_c;


psi_c[n+1]=psi_c[n]+psid_cp*dt;

psid_c=psid_cp+psi2_cp*dt;

psi2_c=psi2_cp+psi3_cp*dt;

psi3_c=-b2*psi2_c-b1*psid_c-b0*(psi_c[n+1]-psi_d);


psid_cp=psid_c; psi2_cp=psi2_c; psi3_cp=psi3_c;


/** Velocity Control **/


// velocity errors

e_u[n+1]=u_e[n+1]-u_c[n+1];

e_v[n+1]=v_e[n+1]-v_c[n+1];

e_w[n+1]=w_e[n+1]-w_c[n+1];


// integral of velocity errors
```

```
      Ie_u[n+1]=Ie_u[n]+(e_u[n+1]+e_u[n])*dt/2;

      Ie_v[n+1]=Ie_v[n]+(e_v[n+1]+e_v[n])*dt/2;

      Ie_w[n+1]=Ie_w[n]+(e_w[n+1]+e_w[n])*dt/2;


      alu = ud_c-ku*e_u[n+1]-kiu*Ie_u[n+1];

      alv = vd_c-kv*e_v[n+1]-kiv*Ie_v[n+1];

      alw = wd_c-kw*e_w[n+1]-kiw*Ie_w[n+1]-g;


      alud = u2_c-ku*(ud[n+1]-ud_c)-kiu*e_u[n+1];

      alvd = v2_c-kv*(vd[n+1]-vd_c)-kiv*e_v[n+1];

      alwd = w2_c-kw*(wd[n+1]-wd_c)-kiw*e_w[n+1];


      // compute thrust

      T[n+1] = -m*alw/cph/cth;

      //if (z_e[n+1]>-.301) T[n+1]=.98*T[n+1];

      if (T[n+1]<0) T[n+1] = 0;

      if (T[n+1]>14) T[n+1] = 14;


      // zero attitude if VELOCITY_CTRL not defined

      theta_d[n+1] = 0;

      phi_d[n+1] = 0;

      thetad_d = 0;

      phid_d = 0;

      theta2_d = 0;

      phi2_d = 0;


      O = matrix::Eulerf(phi[n+1],theta[n+1],psi[n+1]);

#ifdef VELOCITY_CTRL
```

```
    Td = m*(-w2_c+kw*(wd[n+1]-wd_c)+kiw*e_w[n+1])/cph/cth
   +T[n+1]*(phid*tan(phi[n+1])+thetad*tth);
//     Td = m*(-w2_c+kw*(wd[n+1]-wd_c)+w_e[n+1])/cph/cth
  //       +T[n+1]*(phid*tan(phi[n+1])+thetad*tth);


    // Calculate inertial jerks (assholes to be honest)
    u2 = -Td/m*O(0,2)+T[n+1]/m*(phid*O(0,1)-thetad*cph*cth*cps+psid*O(1,2))
       ↪ ;
    /*-lambda*(om2[n]+om4[n]+om1[n]+om3[n])*(ud[n+1]*cps*cth-u_e[n+1]*psid*
       ↪ sps*cth-u_e[n+1]*cps*thetad*sth-wd[n+1]*sth-w_e[n+1]*thetad*cth+
       ↪ vd[n+1]*cth*sps-v_e[n+1]*thetad*sth*sps+v_e[n+1]*cth*psid*cps);
       ↪ */ // propeller drag?
    v2 = -Td/m*O(1,2)+T[n+1]/m*(phid*O(1,1)-thetad*cph*cth*sps-psid*O(0,2))
       ↪ ;
    /*-lambda*(om2[n]+om4[n]+om1[n]+om3[n])*(vd[n+1]*(cph*cps+sph*sps*sth)+
       ↪ v_e[n+1]*(-phid*sph*cps-psid*cph*sps+phid*cph*sps*sth+psid*sph*
       ↪ cps*sth+thetad*sph*sps*cth)-ud[n+1]*(cph*sps-cps*sph*sth)-u_e[n
       ↪ +1]*(-phid*sph*sps+psid*cph*cps+psid*sps*sph*sth-phid*cps*cph*
       ↪ sth-thetad*cps*sph*cth)+wd[n+1]*cth*sph-w_e[n+1]*thetad*sth*sph)
       ↪ +w_e[n+1]*phid*cth*cph;*/ //propeller drag?
    w2 = w2_c-kw*(wd[n+1]-wd_c)-kiw*e_w[n+1];


    alu2 = u3_c-ku*(u2-u2_c)-kiu*(ud[n+1]-ud_c);
    alv2 = v3_c-kv*(v2-v2_c)-kiv*(vd[n+1]-vd_c);
    alw2 = w3_c-kw*(w2-w2_c)-kiw*(wd[n+1]-wd_c);


    // math equations
    beth = (alu*cps+alv*sps)/alw;
    theta_d[n+1] = atan(beth);
```

```
cthd = cos(theta_d[n+1]); sthd = sin(theta_d[n+1]);

beph = (alu*sps-alv*cps)*cthd/alw;
phi_d[n+1] = atan(beph);

bthd = (cps*(alud+alv*psid)+sps*(alvd-alu*psid)-beth*alwd)/alw;
thetad_d = bthd/(1+beth*beth);

bphd = (sps*(cthd*(alud+alv*psid)-thetad_d*sthd*alu)
   +cps*(cthd*(alu*psid-alvd)+thetad_d*sthd*alv)
   -beph*alwd)/alw;

phid_d = bphd/(1+beph*beph);

bth2 = (cps*(alu2+2*alvd*psid-alu*psid*psid+alv*psi2)
   +sps*(alv2-2*alud*psid-alv*psid*psid-alu*psi2)
   -2*bthd*alwd-beth*alw2)/alw;

theta2_d = bth2/(1+beth*beth)-2*beth*bthd*bthd/pow(1+beth*beth,2);

bph2 = (sps*((alu2+alv*psi2+2*alvd*psid-alu*(pow(thetad_d,2)+pow(psid
    ↪ ,2)))*cthd
   -(2*thetad_d*(alud+alv*psid)+alu*theta2_d)*sthd)
   +cps*((alu*psi2-alv2+2*alud*psid+alv*(pow(thetad_d,2)+pow(psid,2)))*
       ↪ cthd
    +(2*thetad_d*(alvd-alu*psid)+alv*theta2_d)*sthd)-2*bphd*alwd-beph*
       ↪ alw2)/alw;
```

```
      phi2_d = bph2/(1+beph*beph)-2*beph*bphd*bphd/pow(1+beph*beph,2);

#endif

      // Euler angle errors
      e_phi[n+1] = phi[n+1]-phi_d[n+1];
      e_the[n+1] = theta[n+1]-theta_d[n+1];
      e_psi[n+1] = psi[n+1]-psi_c[n+1];

      // integral of Euler angle errors
      Ie_ph[n+1] = Ie_ph[n]+(e_phi[n+1]+e_phi[n])*dt/2;
      Ie_th[n+1] = Ie_th[n]+(e_the[n+1]+e_the[n])*dt/2;
      Ie_ps[n+1] = Ie_ps[n]+(e_psi[n+1]+e_psi[n])*dt/2;

      // desired rates
      p_d[n+1] = phid_d-kphi*e_phi[n+1]-kiph*Ie_ph[n+1]
   -(psid_c-kpsi*e_psi[n+1]-kips*Ie_ps[n+1])*sth;
      q_d[n+1] = (thetad_d-ktheta*e_the[n+1]-kith*Ie_th[n+1])*cph
   +(psid_c-kpsi*e_psi[n+1]-kips*Ie_ps[n+1])*sph*cth;
      r_d[n+1] = -(thetad_d-ktheta*e_the[n+1]-kith*Ie_th[n+1])*sph
   +(psid_c-kpsi*e_psi[n+1]-kips*Ie_ps[n+1])*cph*cth;

      // desired rate derivatives
      pd_d = phi2_d-kphi*(phid-phid_d)-kiph*e_phi[n+1]
   -(psi2_c-kpsi*(psid-psid_c)-kips*e_psi[n+1])*sth
   -(psid_c-kpsi*e_psi[n+1]-kips*Ie_ps[n+1])*thetad*cth;
      qd_d = (theta2_d-ktheta*(thetad-thetad_d)-kith*e_the[n+1])*cph
   -(thetad_d-ktheta*e_the[n+1]-kith*Ie_th[n+1])*phid*sph
   +(psi2_c-kpsi*(psid-psid_c)-kips*e_psi[n+1])*sph*cth
   +(psid_c-kpsi*e_psi[n+1]-kips*Ie_ps[n+1])*(phid*cph*cth-thetad*sph*sth);
      rd_d = -(theta2_d-ktheta*(thetad-thetad_d)-kith*e_the[n+1])*sph
   -(thetad_d-ktheta*e_the[n+1]-kith*Ie_th[n+1])*phid*cph
   +(psi2_c-kpsi*(psid-psid_c)-kips*e_psi[n+1])*cph*cth
   -(psid_c-kpsi*e_psi[n+1]-kips*Ie_ps[n+1])*(phid*sph*cth+thetad*cph*sth);

      // rate errors
      e_p[n+1] = p[n+1]-p_d[n+1];
      e_q[n+1] = q[n+1]-q_d[n+1];
      e_r[n+1] = r[n+1]-r_d[n+1];

      // integral of rate errors
      Ie_p[n+1] = Ie_p[n]+(e_p[n+1]+e_p[n])*dt/2;
      Ie_q[n+1] = Ie_q[n]+(e_q[n+1]+e_q[n])*dt/2;
      Ie_r[n+1] = Ie_r[n]+(e_r[n+1]+e_r[n])*dt/2;

      // torques inputs
      t1[n+1] = (pd_d-kp*e_p[n+1]-kip*Ie_p[n+1]-e_phi[n+1])*Ixx+q[n+1]*r[n
          ↪ +1]*(Izz-Iyy)+q[n+1]*Hp[n];
      t2[n+1] = (qd_d-kq*e_q[n+1]-kiq*Ie_q[n+1]-e_phi[n+1]*sph*tth
       -e_the[n+1]*cph-e_psi[n+1]*sph/cth)*Iyy-p[n+1]*r[n+1]*(Izz-Ixx)-p[n
           ↪ +1]*Hp[n];
      t3[n+1] = (rd_d-kr*e_r[n+1]-kir*Ie_r[n+1]-e_phi[n+1]*cph*tth
       +e_the[n+1]*sph-e_psi[n+1]*cph/cth)*Izz+p[n+1]*q[n+1]*(Iyy-Ixx);

      // try out different torque inputs
      /*
  +q[n+1]*r[n+1]*(Izz-Iyy)+q[n+1]*Hp[n]
  -p[n+1]*r[n+1]*(Izz-Ixx)-p[n+1]*Hp[n]
```

```
+p[n+1]*q[n+1]*(Iyy-Ixx)
      t1[n+1] = (pd_d-kp*(p[n+1]-p_d[n+1]))*Ixx+q[n+1]*r[n+1]*(Izz-Iyy)+q[n
          ↪ +1]*Hp[n];
      t2[n+1] = (qd_d-kq*(q[n+1]-q_d[n+1]))*Iyy-p[n+1]*r[n+1]*(Izz-Ixx)-p[n
          ↪ +1]*Hp[n];
      t3[n+1] = (rd_d-kr*(r[n+1]-r_d[n+1]))*Izz;

      t1[n+1] = (pd_d-kp*(p[n+1]-p_d[n+1]))*Ixx+q[n+1]*Hp[n];
      t2[n+1] = (qd_d-kq*(q[n+1]-q_d[n+1]))*Iyy-p[n+1]*Hp[n];
      t3[n+1] = (rd_d-kr*(r[n+1]-r_d[n+1]))*Izz;
      */

      // limit control torques to 0.4 N*m
      if (abs(t1[n+1])>.4) t1[n+1] = sgn(t1[n+1])*.4;
      if (abs(t2[n+1])>.4) t2[n+1] = sgn(t2[n+1])*.4;
      if (abs(t3[n+1])>.4) t3[n+1] = sgn(t3[n+1])*.4;

      // adaptively update thrust constant
#ifdef ADPT_K
      if (z_e[n+1]<-.301) {
   k=k-(8.0e-9f)*sigma(e_w[n+1],.03);
   if (k<.8*kP[0]) k=.8*kP[0];
   if (k>1.2*kP[0]) k=1.2*kP[0];

   kP[n+1]=k;
   //bP[n+1]=b;
      } else {
   kP[n+1]=kP[n];
   //bP[n+1]=bP[n];
```

```
        }
#endif

        // propeller mixing to implement thrust and torques
        om1[n+1] = T[n+1]/(4.0*k)+t1[n+1]/(4.0*k*ls)+t2[n+1]/(4.0*k*ll)-t3[n
            ↪ +1]/(4.0*b);
        om2[n+1] = T[n+1]/(4.0*k)-t1[n+1]/(4.0*k*ls)+t2[n+1]/(4.0*k*ll)+t3[n
            ↪ +1]/(4.0*b);
        om3[n+1] = T[n+1]/(4.0*k)-t1[n+1]/(4.0*k*ls)-t2[n+1]/(4.0*k*ll)-t3[n
            ↪ +1]/(4.0*b);
        om4[n+1] = T[n+1]/(4.0*k)+t1[n+1]/(4.0*k*ls)-t2[n+1]/(4.0*k*ll)+t3[n
            ↪ +1]/(4.0*b);

        om1[n+1] = sqrt(om1[n+1]); om2[n+1] = sqrt(om2[n+1]);
        om3[n+1] = sqrt(om3[n+1]); om4[n+1] = sqrt(om4[n+1]);

        // enforce propeller speed limits
        if (om1[n+1]<wmin) om1[n+1]=wmin;
        if (om2[n+1]<wmin) om2[n+1]=wmin;
        if (om3[n+1]<wmin) om3[n+1]=wmin;
        if (om4[n+1]<wmin) om4[n+1]=wmin;

        if (om1[n+1]>wmax) om1[n+1]=wmax;
        if (om2[n+1]>wmax) om2[n+1]=wmax;
        if (om3[n+1]>wmax) om3[n+1]=wmax;
        if (om4[n+1]>wmax) om4[n+1]=wmax;

        // protect math equations from nan
        if (isnan(om1[n+1])) om1[n+1]=om1[n];
```

```
      if (isnan(om2[n+1])) om2[n+1]=om2[n];
      if (isnan(om3[n+1])) om3[n+1]=om3[n];
      if (isnan(om4[n+1])) om4[n+1]=om4[n];

      // Angular momentum of propellers
      Hp[n+1]=(om1[n+1]+om3[n+1]-om2[n+1]-om4[n+1])*Ip;

      // compute pwm duty cycle based on effective pwm range and min/max
          ↪ propeller speeds
      duty1 = (int)(rng*(om1[n+1]-wmin)/(wmax-wmin)+.5f);
      duty2 = (int)(rng*(om2[n+1]-wmin)/(wmax-wmin)+.5f);
      duty3 = (int)(rng*(om3[n+1]-wmin)/(wmax-wmin)+.5f);
      duty4 = (int)(rng*(om4[n+1]-wmin)/(wmax-wmin)+.5f);

      // write pwm control values to PCA9685
#ifdef FLY
      pwm->ledOffTime(15,min1+duty1);
      pwm->ledOffTime(1,min2+duty2);
      pwm->ledOffTime(14,min3+duty3);
      pwm->ledOffTime(0,min4+duty4);
#endif
      n = n+1; // increment data arrays

      if (n==size-1) { // reset arrays (previous 6000 data points in all
          ↪ arrays will be lost!)
   x_e[0] = x_e[n]; y_e[0] = y_e[n]; z_e[0] = z_e[n];
   u_e[0] = u_e[n]; v_e[0] = v_e[n]; w_e[0] = w_e[n];
   ud[0] = ud[n]; vd[0] = vd[n]; wd[0] = wd[n];
   u_c[0] = u_c[n]; v_c[0] = v_c[n]; w_c[0] = w_c[n];
```

```
  u_d[0] = u_d[n]; v_d[0] = v_d[n]; w_d[0] = w_d[n];

  phi[0] = phi[n]; theta[0] = theta[n]; psi[0] = psi[n];
  p[0] = p[n]; q[0] = q[n]; r[0] = r[n];
  qd[0] = qd[n]; rd[0] = rd[n];

  Hx[0] = Hx[n]; Hy[0] = Hy[n]; Hz[0] = Hz[n];
  Hp[0] = Hp[n];

  e_u[0] = e_u[n]; e_v[0] = e_v[n]; e_w[0] = e_w[n];
  e_phi[0] = e_phi[n]; e_the[0] = e_the[n]; e_psi[0]=e_psi[n];
       e_p[0] = e_p[n]; e_q[0] = e_q[n]; e_r[0]=e_r[n];

  Ie_u[0]=Ie_u[n];Ie_v[0]=Ie_v[n]; Ie_w[0]=Ie_w[n];
  Ie_ph[0]=Ie_ph[n]; Ie_th[0]=Ie_th[n]; Ie_ps[0]=Ie_ps[n];
  Ie_p[0]=Ie_p[n]; Ie_q[0]=Ie_q[n]; Ie_r[0]=Ie_r[n];

  t1[0]=t1[n]; t2[0]=t2[n]; t3[0]=t3[n]; T[0]=T[n];

  om1[0] = om1[n]; om2[0] = om2[n];
  om3[0] = om3[n]; om4[0] = om4[n];
  n = 0; // reset index

  printf("Indices zero'd \n");
  cout.flush();
     }
   }
 }
}
```

flight.h

```
///////////////////////////////////////////////////////////////////////////
// This is the header file for flight.cpp. It contains initializations of
   ↪ data structures, and functions used by flight.cpp
///////////////////////////////////////////////////////////////////////////

#include "MPU9150_9Axis_MotionApps41.h"

#include "mraa.hpp"
#include "Matrix/matrix/math.hpp"

#include <librealsense/rs.hpp>

#include <chrono>
#include <vector>
#include <sstream>
#include <iostream>
#include <algorithm>

#include <unistd.h>
#include <inttypes.h>
#include <stdint.h>
#include <signal.h>
#include <chrono>
#include <fstream>
#include "/usr/include/upm/pca9685.h"
```

```
#include <stdio.h>
#include <string>

#include <termios.h>
#include <ctype.h>
#include <sys/wait.h>
#include <sys/ioctl.h>

#include<cstdlib>
#include<cstdio>
#include<ctime>
#include<time.h>

float pi=M_PI;

// MPU control/status vars
bool dmpReady = false; // set true if DMP init was successful
uint8_t dmpIntStatus; // holds actual interrupt status byte from DMP

uint8_t mpuIntStatus; // holds actual interrupt status byte from MPU
uint8_t devStatus;    // return status after each device operation (0 =
   ↪ success, !0 = e$
uint16_t packetSize=0; // expected DMP packet size (default is 42 bytes)
uint16_t fifoCount;   // count of all bytes currently in FIFO
uint8_t fifoBuffer[64]; // FIFO storage buffer

// orientation/motion vars
Quaternion refQ;      // reference quaternion container
Quaternion corrQ;     // corrected quaternion container
Quaternion Q;         // [w, x, y, z]       quaternion container
VectorInt16 aa;       // [x, y, z]          accel sensor measurements
VectorInt16 ar;
VectorInt16 aaReal;   // [x, y, z]          gravity-free accel sensor
    ↪ measurements
VectorInt16 aaWorld;  // [x, y, z]          world-frame accel sensor
    ↪ measurements
VectorFloat gravity;  // [x, y, z]          body gravity vector

float ypr[3],ypr_r[3]; // [psi, theta, phi] Euler angle containers
int16_t gyro[3];
int16_t a_x,a_y,a_z; // used for correcting accel for rotation (due to IMU
    ↪ offset)

volatile bool mpuInterrupt = false; // indicates whether MPU interrupt pin
    ↪ has gone high

const int n_pts = 1400; // number of points to map

Quaternion fromEuler(float ypr[]) {
  float cosX2 = cos(ypr[2] / 2.0f);
  float sinX2 = sin(ypr[2] / 2.0f);
  float cosY2 = cos(ypr[1] / 2.0f);
  float sinY2 = sin(ypr[1] / 2.0f);
  float cosZ2 = cos(ypr[0] / 2.0f);
  float sinZ2 = sin(ypr[0] / 2.0f);

  float qa[4];
  qa[0] = cosX2 * cosY2 * cosZ2 + sinX2 * sinY2 * sinZ2;
  qa[1] = sinX2 * cosY2 * cosZ2 - cosX2 * sinY2 * sinZ2;
  qa[2] = cosX2 * sinY2 * cosZ2 + sinX2 * cosY2 * sinZ2;
  qa[3] = cosX2 * cosY2 * sinZ2 - sinX2 * sinY2 * cosZ2;

  Quaternion q = Quaternion(qa[0],qa[1],qa[2],qa[3]);
  q.normalize();

  return q;
}

void dmpDataReady() {
  mpuInterrupt = true;
}

using namespace std;
using namespace matrix;

int sgn(float val) {
  if (val>0) {
    return 1;
  } else if (val<0) {
    return -1;
  } else {
    return 0;
  }
}

// linear function with deadzone
```

```
float sigma(float e_w,float ep) {
  if (abs(e_w)<ep/2) {return 0;}
  else {return e_w-sgn(e_w)*ep/2;}
}

char getUserChar() {
  int i;
  ioctl(0, FIONREAD, &i);
  if (i <= 0) return 0;
  return tolower(getchar());
}


float ElpsdRad(const Vector3f &P_o,float a,float b,float c) {
  float theta,phi,x,y,z;

  theta = atan((P_o(2)/c)/sqrt(pow(P_o(0),2)/(a*a)+pow(P_o(1),2)/(b*b)));
  phi = atan2(P_o(1)/b,P_o(0)/a);

  x=a*cos(theta)*cos(phi);
  y=b*cos(theta)*sin(phi);
  z=c*sin(theta);

  return sqrt(x*x+y*y+z*z); //radius of ellipsoid along p_i
}

// is object within ellipsoid
bool in_Elpsd(const Vector3f &P_o,float a,float b,float c) {
  if (P_o(0)==0||P_o(1)==0||P_o(2)==0) return false;
  return (pow(P_o(0)/a,2)+pow(P_o(1)/b,2)+pow(P_o(2)/c,2) <= 1);
}

// set all pwm values to val and close pwm connection
void setAllPWM(upm::PCA9685 *pwm, int val) {
  usleep(5000);
  //pwm->ledOffTime(PCA9685_ALL_LED, val);
  pwm->ledOffTime(15, val);
  pwm->ledOffTime(14, val);
  pwm->ledOffTime(1, val);
  pwm->ledOffTime(0, val);
  usleep(50000);
  printf(" - PWM off\n");
  pwm->setModeSleep(true);
  pwm->setModeSleep(false);
}

void write_Data(ofstream &file, float time[],float data[],int n) {
  if (file.is_open()) {
    for(int count = 0; count < n; count ++) {
   file << time[count] << "\t" << data[count] << "\n" ;
    }
    file << "\n";
  }
  else cout << "Unable to open file";
}

void write_Data(ofstream &file, float dat1[], float dat2[], float dat3[],int
    ↪  n) {
```

```
  if (file.is_open()) {
    for(int count = 0; count < n; count ++) {
   file << dat1[count] << "\t" << dat2[count] << "\t" << dat3[count] << "\n"
       ↪ ;
    }
    file << "\n";
  }
  else cout << "Unable to open file";
}
void write_Data(ofstream &file, const Matrix<float, 3, n_pts> &pts,int i) {
  if (file.is_open()) {
    for(int count = 0; count < i; count ++) {
      file << pts(0,count) << "\t" << pts(1,count) << "\t" << pts(2,count) <<
          ↪  "\n";
    }
    file << "\n";
  }
  else cout << "Unable to open file";
}

void write_Data(ofstream &file, float time[], float dat1[], float dat2[],
    ↪ float dat3[],float dat4[],int n) {
  if (file.is_open()) {
    for(int count = 0; count < n; count ++) {
      file << time[count] << "\t" << dat1[count] << "\t" << dat2[count] << "\
          ↪ t" << dat3[count] << "\t" << dat4[count] << "\n";
    }
    file << "\n";
  }
  else cout << "Unable to open file";
}

void write_Position(int n,float time[],float x[], float y[],float z[],float
    ↪ z_o[]) {
  ofstream xy ("xy.txt"); ofstream xyz ("xyz.txt"); ofstream zz ("z_e.txt");
  ofstream xx ("x_e.txt"); ofstream yy ("y_e.txt");
  write_Data(xy,y,x,n); write_Data(xyz,x,y,z,n); write_Data(zz,time,z,n);
  write_Data(xx,time,x,n); write_Data(yy,time,y,n);
  write_Data(zz,time,z_o,n);
  xy.close(); xyz.close(); zz.close(); xx.close(); yy.close();
}

void write_Map(int i, const Matrix<float, 3, n_pts> &pts) {
  ofstream map ("map.txt");
  write_Data(map,pts,i);
  map.close();
}

void write_Velocity(int n,float time[],float u[],float v[],float w[],float
    ↪ u_c[],float v_c[],float w_c[]) {
  ofstream uu ("u_e.txt"); ofstream vv ("v_e.txt"); ofstream ww ("w_e.txt");
  write_Data(uu,time,u,n); write_Data(vv,time,v,n); write_Data(ww,time,w,n);
  write_Data(uu,time,u_c,n); write_Data(vv,time,v_c,n); write_Data(ww,time,
      ↪ w_c,n);
  uu.close(); vv.close(); ww.close();
}
```

```
void write_Vel_Int(int n,float time[],float Ie_u[],float Ie_v[],float Ie_w
    ↪ []) {
  ofstream uu ("Ie_u.txt"); ofstream vv ("Ie_v.txt"); ofstream ww ("Ie_w.txt
      ↪ ");
  write_Data(uu,time,Ie_u,n); write_Data(vv,time,Ie_v,n); write_Data(ww,time
      ↪ ,Ie_w,n);
  uu.close(); vv.close(); ww.close();
}

void write_Acceleration(int n, float time[],float ud[],float vd[],float wd
    ↪ []) {
  ofstream uu ("ud.txt"); ofstream vv ("vd.txt"); ofstream ww ("wd.txt");
  write_Data(uu,time,ud,n); write_Data(vv,time,vd,n); write_Data(ww,time,wd,
      ↪ n);
  uu.close(); vv.close(); ww.close();
}

void write_Angles(int n,float time[],float phi[],float theta[],float psi[],
    ↪ float phi_d[],float theta_d[],float psi_c[]) {
  ofstream Phi ("phi.txt"); ofstream Theta ("theta.txt"); ofstream Psi ("psi
      ↪ .txt");
  write_Data(Phi,time,phi,n); write_Data(Theta,time,theta,n); write_Data(Psi
      ↪ ,time,psi,n);
  write_Data(Phi,time,phi_d,n); write_Data(Theta,time,theta_d,n); write_Data
      ↪ (Psi,time,psi_c,n);
  Phi.close(); Theta.close(); Psi.close();
}
```

```
void write_Ang_Int(int n,float time[],float Ie_ph[],float Ie_th[],float
    ↪ Ie_ps[]) {
  ofstream uu ("Ie_ph.txt"); ofstream vv ("Ie_th.txt"); ofstream ww ("Ie_ps.
    ↪ txt");
  write_Data(uu,time,Ie_ph,n); write_Data(vv,time,Ie_th,n); write_Data(ww,
    ↪ time,Ie_ps,n);
  uu.close(); vv.close(); ww.close();
}

void write_Angle_Errors(int n,float time[],float e_phi[],float e_the[],float
    ↪  e_psi[]) {
  ofstream eph ("eph.txt"); ofstream eth ("eth.txt"); ofstream eps ("eps.txt
    ↪ ");
  write_Data(eph,time,e_phi,n); write_Data(eth,time,e_the,n); write_Data(eps
    ↪ ,time,e_psi,n);
  eph.close(); eth.close(); eps.close();
}

void write_Rates(int n,float time[],float p[],float q[],float r[],float p_d
    ↪ [],float q_d[],float r_d[]) {
  ofstream pp ("p.txt"); ofstream qq ("q.txt"); ofstream rr ("r.txt");
  write_Data(pp,time,p,n); write_Data(qq,time,q,n); write_Data(rr,time,r,n);
  write_Data(pp,time,p_d,n); write_Data(qq,time,q_d,n); write_Data(rr,time,
    ↪ r_d,n);
  pp.close(); qq.close(); rr.close();
}
```

```
void write_Momentum(int n,float time[],float hx[],float hy[],float hz[],
   ↪ float hp[]) {
  ofstream Hb ("Hb.txt"); ofstream Hp ("Hp.txt");
  write_Data(Hb,time,hx,n); write_Data(Hb,time,hy,n); write_Data(Hb,time,hz,
     ↪ n);
  write_Data(Hp,time,hp,n);
  Hb.close(); Hp.close();
}

void write_Rate_Errors(int n,float time[],float e_p[],float e_q[],float e_r
   ↪ []) {
  ofstream ep ("ep.txt"); ofstream eq ("eq.txt"); ofstream er ("er.txt");
  write_Data(ep,time,e_p,n); write_Data(eq,time,e_q,n); write_Data(er,time,
     ↪ e_r,n);
  ep.close(); eq.close(); er.close();
}

void write_Inputs(int n,float time[],float T[],float t1[],float t2[],float
t3[],float om1[],float om2[],float om3[],float om4[]) {
  ofstream TT ("T.txt"); ofstream om ("om.txt");
  ofstream tt1 ("t1.txt"); ofstream tt2 ("t2.txt"); ofstream tt3 ("t3.txt");
  write_Data(TT,time,T,n); write_Data(om,time,om1,om2,om3,om4,n);
  write_Data(tt1,time,t1,n); write_Data(tt2,time,t2,n); write_Data(tt3,time,
     ↪ t3,n);
  TT.close(); om.close();
  tt1.close(); tt2.close(); tt3.close();
}

void write_Params(int n, float time[], float kp[],float bp[]) {
  ofstream kk ("k.txt"); ofstream bb ("b.txt");
  write_Data(kk,time,kp,n); write_Data(bb,time,bp,n);
  kk.close(); bb.close();
}

void write_Freq(int n, float time[],float clf[]) {
  ofstream freq ("freq.txt");
  write_Data(freq,time,clf,n);
  freq.close();
}

void write_Pos_Obj(int n,float time[],float px[], float py[],float pz[],
    ↪ float dist[]) {
  ofstream posx ("posx.txt"); ofstream posy ("posy.txt"); ofstream posz ("
      ↪ posz.txt");
  ofstream pts ("pts.txt");
  write_Data(posx,time,px,n); write_Data(posy,time,py,n); write_Data(posz,
      ↪ time,pz,n);
  write_Data(pts,time,dist,n);
  posx.close(); posy.close(); posz.close(); pts.close();
}

// Initialize all the data holders
uint8_t frame[22];
uint8_t int_frame[26];

// standard deviation height/velocity limits
const float h_min = 2.0f;
const float h_max = 8.0f;
const float v_min = 0.5f;
const float v_max = 1.0f;

// polynomial noise model, found using least squares fit
// h, h**2, v, v*h, v*h**2

const float P[5] = {0.04005232f, -0.00656446f, -0.26265873f, 0.13686658f
    ↪ ,-0.00397357f};
float flow_vxy_stddev=0;

float flow_x_rad;
float flow_y_rad;
float dt_flow;
float gyro_x_rad = 0;
float gyro_y_rad = 0;
uint16_t f_c;
uint8_t qual;
float rot_sq=0;
float rotrate_sq=0;

matrix::Vector3f delta_b;
matrix::Vector3f delta_n;

float h=0;
float v=0;

// acceleration estimation matrices
matrix::Vector3f x_k;
matrix::Vector3f y_k;
```

```
matrix::SquareMatrix<float, 3> ph_k;

matrix::SquareMatrix<float, 3> C_k;

matrix::SquareMatrix<float, 3> P_k;

matrix::SquareMatrix<float, 3> Q_k;

matrix::SquareMatrix<float, 3> R_k;

matrix::SquareMatrix<float, 3> K_k;

matrix::Matrix<float,3,3> SI_k;


matrix::Vector<float,4> x;

matrix::Vector3f y;

matrix::SquareMatrix<float, 4> A;

matrix::Matrix<float, 4, 3> B;

matrix::Matrix<float, 3, 4> C;

matrix::SquareMatrix<float, 4> _P;

matrix::SquareMatrix<float, 4> _Q;

matrix::SquareMatrix<float, 3> R;

matrix::Matrix<float,4,3> K;

matrix::Matrix<float,3,3> S_I;


matrix::Vector<float,2> xk;

matrix::Vector<float,2> yk;

matrix::SquareMatrix<float, 2> Ak;

matrix::SquareMatrix<float, 2> Ck;

matrix::SquareMatrix<float, 2> Pk;

matrix::SquareMatrix<float, 2> Qk;

matrix::SquareMatrix<float, 2> Rk;

matrix::SquareMatrix<float, 2> Kk;

matrix::SquareMatrix<float, 2> SIk;
```

```
// 1-2 attitude matrix
matrix::Dcmf Att;
matrix::Dcmf O;

// Velocity of quadrotor body-frame and inertial
matrix::Vector3f V_q;
matrix::Vector3f V_qI,V_qIp;

// Goal position and velocity
matrix::Vector3f P_g;
matrix::Vector3f P_gI;
matrix::Vector3f V_g;

// Object position in the body-frame
matrix::Vector3f P_o,P_oe;

// Object positions in the inertial frame
matrix::Vector3f P_oI;

// Object velocity in the body-frame
matrix::Vector3f V_o;

matrix::SquareMatrix<float, 3> Om; // skew symm. matrix corresponding to
    ↪ omega

matrix::Matrix<float, 3, n_pts > pts;

int i=0; //pts counter
float u_a=0;
float v_a=0;
float w_a=0;

// parameters for adjusting thrust constant
bool do_leveling=false;
int count=0; // counter

// initialize ESC's
bool cal=false; // initialized boolean
bool incr=false; // pulse increment boolean

int n=0;

const int size=6000;

float px[size],py[size],pz[size];
float dist[size];
float min_dist = 1;

float psi_d,phid_d,thetad_d,phi2_d,theta2_d,pd_d,qd_d,rd_d;
float x_e[size],y_e[size],z_e[size],z_o[size];
float u_e[size],v_e[size],w_e[size];
float u_d[size],v_d[size],w_d[size];

float ud[size],vd[size],wd[size];
float udb[size],vdb[size],wdb[size];
float u2,v2,w2;

float phi[size],theta[size],psi[size];
float p[size],q[size],r[size];
float pd[size],qd[size],rd[size];

float e_u[size],e_v[size],e_w[size];
float Ie_u[size],Ie_v[size],Ie_w[size];

float e_phi[size],e_the[size],e_psi[size];
float Ie_ph[size],Ie_th[size],Ie_ps[size];

float e_p[size],e_q[size],e_r[size];
float Ie_p[size],Ie_q[size],Ie_r[size];

float kP[size],bP[size];
float Hx[size],Hy[size],Hz[size],Hp[size];

float phid,thetad,psid,psi2;

float u_c[size],v_c[size],w_c[size];
float Td,ud_c,vd_c,wd_c,psid_c,u2_c,v2_c,w2_c,psi2_c,u3_c,v3_c,w3_c,psi3_c,
    ↪ u4_c,v4_c,w4_c;
float ud_cp,vd_cp,wd_cp,psid_cp,u2_cp,v2_cp,w2_cp,psi2_cp,u3_cp,v3_cp,w3_cp,
    ↪ psi3_cp,u4_cp,v4_cp,w4_cp;
float alu,alv,alw,alud,alvd,alwd,alu2,alv2,alw2,beth,beph,bthd,bphd,bth2,
    ↪ bph2;
float dt,cph,sph,tph,cth,sth,tth,cthd,sthd,cps,sps;

float phi_d[size],theta_d[size],psi_c[size];
float p_d[size],q_d[size],r_d[size];
float t1[size],t2[size],t3[size],T[size];
float om1[size],om2[size],om3[size],om4[size];


int duty1,duty2,duty3,duty4;

float Time[size],sample_freq[size];


float p_gn;

float st_d;


char kcom; // key command

bool traj=false;

bool land=false;

bool avoid=false;
```

Bibliography

[1] G. M. Hoffmann, H. Huang, S. L. Waslander, and C. J. Tomlin, “Quadrotor helicopter flight dynamics and control: Theory and experiment,” *Proceedings of the AIAA Guidance, Navigation, and Control Conference*, Hilton Head, South Carolina, August 2007.

[2] T. Dierks and S. Jagannathan, “Output feedback control of a quadrotor uav using neural networks,” *IEEE Transactions on Neural Networks*, vol. 21, no. 1, pp. 50–66, January 2010.

[3] D. Cabecinhas, R. Cunha, and C. Silvestre, “A nonlinear quadrotor trajectory tracking controller with disturbance rejection,” *Control Engineering Practice*, vol. 26, pp. 1–10, May 2014.

[4] R. Mahony, V. Kumar, and P. Corke, “Multirotor aerial vehicles: Modeling, estimation, and control of quadrotor,” *IEEE Robotics and Automation Magazine*, vol. 19, no. 3, pp. 20–32, September 2012.

[5] A. Bachrach, S. Prentice, R. He, and N. Roy, “Range-robust autonomous navigation in gps-denied environments,” *Journal of Field Robotics*, vol. 28, no. 5, pp. 644–666, August 2011. [Online]. Available: http://dblp.uni-trier.de/db/journals/jfr/jfr28.html#BachrachPHR11

[6] A. Bachrach, S. Prentice, R. He, P. Henry, A. S. Huang, M. Krainin, D. Maturana, D. Fox, and N. Roy, “Estimation, planning, and mapping for autonomous flight using an rgb-d camera in gps-denied environments,” *The*

International Journal of Robotics Research, vol. 31, no. 11, pp. 1320–1343, September 2012. [Online]. Available: https://doi.org/10.1177/0278364912455256

[7] X. Zhang, B. Xian, B. Zhao, and Y. Zhang, “Autonomous flight control of a nano quadrotor helicopter in a gps-denied environment using on-board vision,” *IEEE Transactions on Industrial Electronics*, vol. 62, no. 10, pp. 6392–6403, April 2010.

[8] R. He, S. Prentice, and N. Roy, “Planning in information space for a quadrotor helicopter in a gps-denied environment,” *IEEE International Conference on Robotics and Automation*, pp. 1814–1820, Pasadena, California, May 2008.

[9] T. Tomic, K. Schmid, P. Lutz, A. Domel, M. Kassecker, E. Mair, I. L. Grixa, F. Ruess, M. Suppa, and D. Burschka, “Toward a fully autonomous uav: Research platform for indoor and outdoor urban search and rescue,” *IEEE Robotics and Automation Magazine*, vol. 19, no. 3, pp. 46–56, September 2012.

[10] E. Altug, J. P. Ostrowski, and R. Mahony, “Control of a quadrotor helicopter using visual feedback,” *Proceedings 2002 IEEE International Conference on Robotics and Automation*, vol. 1, pp. 72–77, Washington, DC, May 2002.

[11] M. Turpin, N. Michael, and V. Kumar, “Trajectory design and control for aggressive formation flight with quadrotors,” *Autonomous Robots*, vol. 33, pp. 143–156, August 2012.

[12] M. Turpin, N. Michael, and V. Kumar, “Concurrent assignment and planning of trajectories for multiple robots,” *The International Journal of Robotics Research*, vol. 33, no. 1, pp. 98–112, January 2014. [Online]. Available: http://ijr.sagepub.com/content/33/1/98.abstract

[13] D. Zhou and M. Schwager, “Assistive collision avoidance for quadrotor swarm teleoperation,” *IEEE International Conference on Robotics and Automation*, pp. 1249–1254, Stockholm, Sweden, May 2016.

[14] M. Saska, V. Vonasek, J. Chudoba, J. Thomas, G. Loianno, and V. Kumar, "Swarm distribution and deployment for cooperative surveillance by micro-aerial vehicles," *Intelligent Robot Systems*, vol. 84, no. 2, pp. 469–492, December 2016.

[15] Q. Lindsey, D. Mellinger, and V. Kumar, "Construction with quadrotor teams," *Autonomous Robots*, vol. 33, no. 3, pp. 323–336, June 2012. [Online]. Available: http://dx.doi.org/10.1007/s10514-012-9305-0

[16] J. T.-Y. Wen and K. Kreutz-Delgado, "The attitude control problem," *IEEE Transactions on Automatic Control*, vol. 36, no. 10, pp. 1148–1162, October 1991.

[17] S. Bouabdallah, A. Noth, and R. Siegwart, "Pid vs lq control techniques applied to an indoor micro quadrotor," *Intelligent Robots and Systems*, pp. 2451–2456, Sendai, Japan, September 2004.

[18] S. J. Haddadi, O. Emamagholi, F. Javidi, and A. Fakharian, "Attitude control and trajectory tracking of an autonomous miniature aerial vehicle," *AI and Robotics*, Qazvin, Iran, April 2015.

[19] A. Tayebi and S. McGilvray, "Attitude stabilization of a vtol quadrotor aircraft," *IEEE Transactions on Control Systems Technology*, vol. 14, no. 3, pp. 562–571, May 2006.

[20] A. Kushleyev, D. Mellinger, C. Powers, and V. Kumar, "Towards a swarm of agile micro quadrotors." *Autonomous Robots*, vol. 85, no. 4, pp. 287–300, November 2013.

[21] T. Lee, M. Leok, and N. H. McClamroch, "Geometric tracking control of a quadrotor uav on se(3)," *49th IEEE Conference on Decision and Control*, pp. 5420–5425, Atlanta, Georgia, December 2010.

[22] D. Mellinger and V. Kumar, “Minimum snap trajectory generation and control for quadrotors,” *IEEE International Conference on Robotics and Automation*, pp. 2520–2525, Shanghai, China, May 2011.

[23] R. Xu and U. Ozguner, “Sliding mode control of a quadrotor helicopter,” *Proceedings of the 45th IEEE Conference on Decision and Control*, pp. 4957–4962, San Diego, California, December 2006.

[24] S. Bouabdallah and R. Siegwart, “Backstepping and sliding-mode techniques applied to an indoor micro quadrotor,” *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*, pp. 2247–2252, Barcelona, Spain, April 2005.

[25] M. O. Efe, “Robust low altitude behavior control of a quadrotor rotorcraft through sliding modes,” *2007 Mediterranean Conference on Control and Automation*, Athens, Greece, June 2007.

[26] S. Bouabdallah and R. Siegwart, “Full control of a quadrotor,” *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 153–158, San Diego, California, October 2007.

[27] T. Congling, W. Jingwen, Y. Zhaojie, and Y. Guohui, “Integral backstepping based nonlinear control for quadrotor,” *35th Chinese Control Conference*, pp. 10 581–10 585, Chengdou, China, July 2016.

[28] T. Madani and A. Benallegue, “Backstepping control for a quadrotor helicopter,” *Proceedings of the 2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 3255–3260, Beijing, China, October 2006.

[29] D. Matouk, O. Gherouat, F. Abdessemed, and A. Hassam, “Quadrotor position and attitude control via backstepping approach,” *8th International Conference*

on Modelling, Identification and Control, pp. 73–79, Algiers, Algeria, November 2016.

[30] S. Saeedi, A. Nagaty, C. Thibault, M. Trentini, and H. Li, "Perception and navigation for an autonomous quadrotor in gps-denied environments," *International Journal of Robotics and Automation*, vol. 31, no. 6, October 2016.

[31] T. Hamel and R. Mahony, "Visual servoing of an under actuated dynamic rigid-body system: An image based approach," *IEEE Transactions on Robotics and Automation*, vol. 18, no. 2, pp. 187–198, April 2002.

[32] O. Bourquardez, R. Mahony, N. Guenard, F. Chaumette, T. Hamel, , and L. Eck, "Image-based visual servo control of the translation kinematics of a quadrotor aerial vehicle," *IEEE Transactions on Robotics*, vol. 25, no. 3, pp. 743–749, June 2009.

[33] [Online]. Available: http://optitrack.com/products/prime-13/indepth.html

[34] D. Mellinger, N. Michael, and V. Kumar, "Trajectory generation and control for precise aggressive maneuvers with quadrotors," *The International Journal of Robotics Research*, vol. 31, no. 5, pp. 664–674, January 2012. [Online]. Available: http://ijr.sagepub.com/content/31/5/664.abstract

[35] D. Abeywardena, S. Kodagoda, G. Dissanayake, , and R. Munasinghe, "Improved state estimation in quadrotor mavs: A novel drift-free velocity estimator," *IEEE Robotics and Automation Magazine*, vol. 20, no. 4, pp. 32–39, December 2013.

[36] D. Honegger, L. Meier, P. Tanskanen, and M. Pollefeys, "An open source and open hardware embedded metric optical flow cmos camera for indoor and outdoor applications," *IEEE Conference on Robotics and Automation*, pp. 1736–1741, Karlsruhe, Germany, May 2013.

[37] G. Loianno, G. Cross, C. Qu, Y. Mulgaonkar, J. A. Hesch, and V. Kumar, "Flying smartphones: Automated flight enabled by consumer electronics," *IEEE Robotics and Automation Magazine*, vol. 22, no. 2, pp. 24–32, May 2015. [Online]. Available: http://dx.doi.org/10.1109/MRA.2014.2382792

[38] S. Grzonka, G. Grisetti, and W. Burgard, "A fully autonomous indoor quadrotor," *IEEE Transactions on Robotics*, vol. 28, no. 1, pp. 90–100, February 2012.

[39] S. Shen, N. Michael, and V. Kumar, "Autonomous multi-floor indoor navigation with a computationally constrained mav," *2011 IEEE International Conference on Robotics and Automation*, pp. 20–25, Shanghai, China, May 2011.

[40] T. T. Mac, C. Copot, A. Hernandez, and R. D. Keyser, "Improved potential field method for unknown obstacle avoidance using uav in indoor environment," *IEEE 14th International Symposium on Applied Machine Intelligence and Informatics*, pp. 345–350, Herlany, Slovakia, January 2016.

[41] S. Ahrens, D. Levine, G. Andrews, and J. P. How, "Vision-based guidance and control of a hovering vehicle in unknown, gps-denied environments," *2009 IEEE International Conference on Robotics and Automation*, pp. 2643–2648, Kobe, Japan, May 2009.

[42] M. Qasim and K. D. Kim, "Super-ellipsoidal potential function for autonomous collision avoidance of a teleoperated uav," *International Journal of Mechanical and Mechatronics Engineering*, vol. 10, no. 1, August 2016.

[43] F. Rehmatullah and J. Kelly, "Vision-based collision avoidance for personal aerial vehicles using dynamic potential fields," *12th Conference on Computer and Robot Vision*, pp. 297–304, Halifax, NS, Canada, June 2015.

[44] J. Park and Y. Kim, “Collision avoidance for quadrotor using stereo vision depth maps,” *IEEE Transactions on Aerospace and Electronic Systems*, vol. 51, no. 4, pp. 3226–3241, October 2015.

[45] [Online]. Available: https://get.google.com/tango/

[46] T. S. Alderete, “Simulator aero model implementation,” *NASA Ames Research Center, Moffett Field, California*, 2014.

[47] T. Luukkonen, “Modeling and control of quadcopter,” *Aalto University, School of Science*, August 2011.

[48] [Online]. Available: https://github.com/intel-iot-devkit/mraa

[49] [Online]. Available: https://github.com/intel-iot-devkit/upm/tree/master/src/pca9685

[50] [Online]. Available: https://github.com/jrowberg/i2cdevlib

[51] [Online]. Available: https://github.com/IntelRealSense/librealsense

[52] [Online]. Available: https://github.com/PX4/Matrix

[53] [Online]. Available: https://github.com/PX4/Firmware/blob/master/src/modules/local_position_estimator/sensors/flow.cpp

[54] G. R. Fowles, *Analytical Mechanics, 4th Edition.* NY, NY: Saunders, 1986.

Vita

Name: Thomas Kirven

Bachelor of Science in Physics, May 2014

Centre College, Danville, Kentucky

Place of Birth: Louisville, Kentucky