A Hardware-Minimal Unscented Kalman Filter Framework for Visual-Inertial Navigation of Small Unmanned Aircraft

Joshua Galen Eddy

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Aerospace Engineering

Kevin Kochersberger, Chair
Mazen Farhood
Craig Woolsey

May 5, 2017
Blacksburg, Virginia

Keywords: Unscented Kalman Filter, SLAM, State Estimation, Localization,
Visual-Inertial Navigation, Unmanned Aircraft

Abstract

This thesis presents the formulation and implementation of an Unscented Kalman Filter (UKF) framework for fusion of visual and inertial sensor data in unmanned aircraft navigation. Specifically, we fuse sensor readings from a 3-axis accelerometer, 3-axis gyroscope, and a Simultaneous Localization and Mapping (SLAM) algorithm to estimate the pose of an aircraft, such as a quadcopter, capable of hovering flight. We discuss the formulation of our UKF fusion algorithm, the development of a Robot Operating System (ROS) software package implementing the algorithm, and experiments in which this algorithm was used to track the motion of a physical vehicle simulating hovering flight in an indoor environment. In laboratory testing, the system was able to localize the test vehicle consistently with a mean total positional error of less than 7 cm and an overall maximum total error of less than 26 cm, as compared to the output of a Vicon motion capture system. We discuss possible applications of this system and future work that may build upon the results developed herein.

Abstract (General Audience)

This thesis presents the development and implementation of a software framework for estimating the position of a drone during flight. This framework is based on an algorithm known as the Unscented Kalman Filter (UKF), a recursive method of estimating the state of a highly nonlinear system, such as an aircraft. In this thesis, we present a UKF formulation specially designed for a quadcopter carrying an Inertial Measurement Unit (IMU) and a downward-facing camera. The UKF fuses data from each of these sensors to track the position of the quadcopter over time. This work supports a number of similar efforts in the robotics and aerospace communities to navigate in GPS-denied environments with minimal hardware and minimal computational complexity. The software framework explored in this thesis provides a means for roboticists to easily implement similar UKF-based state estimators for a wide variety of systems, including surface vessels, undersea vehicles, and automobiles. We test the system's effectiveness by comparing its position estimates to those of a commercial motion capture system and then discuss possible applications.

Acknowledgments

This work would not have been possible were it not for my graduate adviser and committee chair, Dr. Kevin Kochersberger. He has been a constant source of support since October 2012 when he provided my friends and me with lab space in which to design our first competition robots. Over the ensuing five years, Dr. Kochersberger has played a pivotal role in my development as an engineer and researcher. He demonstrates his commitment to nurturing students every day by giving tirelessly of his time and energy.

It is also my pleasure to thank Dr. Danette Allen of the NASA Langley Autonomy Incubator (now the Intelligent Flight Systems and Autonomy Innovation Laboratory), who provided me with two summer internships in her lab and taught me much of what I know today about Kalman filtering. Dr. Allen supplied me with both funding and equipment in order to perform this research, even going so far as to allow me to use her lab's motion capture facilities for my experiments.

I would also like to thank Dr. Loc Tran and Mr. James Neilan of NASA Langley and Ralph Williams of Analytical Mechanics Associates, who collectively provided me with hours of programming guidance during my time at the Autonomy Incubator.

Special thanks go to Samuel Rubenking, who stuck with me to the end. Thanks, Sam.

Contents

List of Figures

List of Tables

Abbreviations

EKF Extended Kalman Filter

GPS Global Positioning System

IMU Inertial Measurement Unit

KF Kalman Filter

PTAM Parallel Tracking and Mapping

ROS Robot Operating System

SLAM Simultaneous Localization and Mapping

UAS Unmanned Aircraft System

UAV Unmanned Aerial Vehicle

UKF Unscented Kalman Filter

UT Unscented Transform

VIN Visual-Inertial Navigation

VO Visual Odometry

Chapter 1

Introduction

Until recently, the sensing and localization capabilities of small unmanned aerial vehicles (UAVs) have been computationally constrained due to restrictions on both the size and weight of sensors and onboard computer systems. Recent advances in miniaturized desktop computers, as well as low-power Graphics Processing Units (GPUs), have enabled a new class of computationally intensive algorithms to run onboard small aircraft without degrading flight time or other performance metrics. Specifically, small drones now present a viable platform for high-fidelity Simultaneous Localization and Mapping (SLAM), visual object recognition, and state estimation algorithms employing numerous heterogeneous sensors. In addition, the rising popularity of drones among hobbyists and researchers has fueled tremendous growth in the markets for brushless motors, electronic speed controllers, and airframes. With high-quality flight hardware and lightweight computers readily available, the arsenal of the robotics researcher has never been better stocked. The work detailed in this thesis is, in many ways, the product of these advantageous market conditions. With small aircraft capable of lifting heavier payloads and computers now lighter and smaller in footprint, demand is high for outfitting high-performance vehicles with top-notch sensing capabilities.

This work centers on an algorithm known as the Unscented Kalman Filter (UKF), a sensor fusion method for estimating the state of systems such as small aircraft. For years, this algorithm was inaccessible to roboticists and aerospace researchers alike due to its computational complexity. Computers capable of running the algorithm were simply too large, too heavy, and too power-hungry to be viable onboard components of small UAVs. The advent of smaller computers has changed this state of affairs. In this thesis, we present a formulation of the Unscented

Kalman Filter aimed at estimating the pose of a quadcopter. This UKF formulation fuses outputs from only two sensors: an inertial measurement unit (IMU) and a downward-facing monocular camera. These two data streams are fused to estimate the vehicle's pose in real time. We propose this UKF framework as a hardware-minimal starting point for advanced multi-sensor fusion in UAV navigation.

1.1 System Applications

Applications for this system are wide-ranging, spanning virtually every conceivable use for a small drone. The following is a non-exhaustive list of possible applications:

1. Mapping and 3D reconstruction of structures and topography,
2. Emergency response,
3. Infrastructure inspection,
4. Autonomous package delivery, and
5. Military/defense solutions.

What all of these applications have in common is the need for robust localization. In each of the above scenarios, losing GPS could cause a crash. In such an event, the vehicle itself would pose a real threat to bystanders and property. Moreover, in the case of military aircraft, the vehicle could be captured by hostile forces.

Mapping and 3D Reconstruction

Growing demand for so-called "precision agriculture" has brought with it the need to map large areas of cropland at a high speed, with a low cost point. In the past, this need has been met by using satellite imagery and human-captured photography. With the arrival of low-cost drones, this work has been offloaded to aerial robots. Outfitted with specialized sensors such as hyperspectral cameras[1], small UAS are now able to provide minute-by-minute coverage of large expanses of land. Moreover, these robots can be outfitted with reservoirs of pesticide or fertilizer and deployed to spray individual plants autonomously. It is in such roles that high-accuracy

[1] https://en.wikipedia.org/wiki/Hyperspectral_imaging

robust localization becomes a necessity. The accuracy required for precision agriculture can be obtained by utilizing the sensor fusion techniques discussed in this thesis.

The surfaces of buildings and other structures can be mapped in a similar fashion, given the appropriate sensors and the right framework by which to fuse them. 3D reconstruction has become popular among real estate agents, safety inspectors, and other parties with a vested interest in precise 3D renderings of large geometries. In this case, the aircraft must be able to localize itself precisely in order to take full advantage of onboard sense-and-avoid technologies. The ability of the aircraft to estimate its position is crucial to its ability to map hard-to-reach areas and maintain stable flight. Sensor fusion systems shine when GPS and other sensors are inevitably compromised by various environmental factors.

Emergency Response

UAS technologies have become a desirable tool for first responders in many parts of the world. The ability to deploy a UAV to survey forest fires or search for missing persons has been a game changer for law enforcement and emergency response teams. In the case of looking for survivors of a natural disaster, the requirements for an effective vehicle system are accurate localization and overall robustness. Vehicles sent into disaster areas will have to be able to navigate in sensor-compromising environments, sometimes at substantial speed. This use case highlights the need for redundant sensors and intelligent, failure-resistant sensor fusion.

Infrastructure Inspection

Infrastructure inspection is similar in many respects to 3D mapping. In both cases, small drones are preferable to manned aircraft because of their low cost, high maneuverability, and ease of use. Governments and private corporations alike have taken an interest in using small aircraft to automate the inspection of many types of infrastructure, including the following:

1. Power lines,
2. Oil rigs,
3. Gas pipelines,
4. Railroads, and
5. Buildings.

UAVs are an obvious ally to companies servicing most types of static infrastructure. Inspecting miles of wires or railroad track is a task well-suited to today's UAVs, which commonly ship with intuitive flight planning software. Operators—even those who have no experience flying remote-control aircraft—are now able to program missions and deploy drones with ease. The aircraft are able to survey miles of infrastructure regardless of conditions on the ground, such as flooding, ill-maintained roads, or wild animals. Operators with First-Person View (FPV) hardware can watch video streaming live from the aircraft as if they were onboard themselves. The need for robust localization here is obvious: the UAV cannot inspect anything to which it cannot navigate. Operating in the wilderness may require flying below the forest canopy or under natural overhangs that could block GPS reception. The same is true in urban environments, where tall buildings create so-called "urban canyons" which compromise GPS effectiveness. Large metal structures can also distort magnetometry readings and thus undermine the aircraft's ability to maintain its heading. In any case, UAVs used for large-scale inspections will be dependent upon intelligently combined sensing modalities.

Autonomous Package Delivery

A number of organizations, most notably Amazon[2], have made big bets on drones as the future of delivery services. The task of delivering goods to individual homes is no trifling matter. American homes come in all shapes and sizes and are surrounded by numerous structures and natural obstacles that could endanger a delivery aircraft and, by extension, the people and property nearby. Moreover, any drone delivery service would have to be predicated upon the ability of the UAV to land with sub-meter accuracy. Delivery drones will have to land precisely in cluttered environments amid dynamic obstacles. Visual sensing will be extremely important for identifying safe landing zones, avoiding power lines and fences, and surmounting other environmental challenges. GPS alone will not be enough to guide the vehicle safely in every conceivable circumstance.

Military/Defense Solutions

Perhaps the most obvious application of a UKF fusion framework is in military operations, where precision navigation and robustness to sensor degradation are paramount for protecting soldiers

[2] https://www.amazon.com/Amazon-Prime-Air/b?node=8037720011

in the field. As of the time of this writing, UAVs have been used in combat for years. However, augmented navigation could still make revolutionary contributions in areas such as

1. Robust tracking of friendly and hostile elements,
2. Remote observation of roads and vehicle-related hazards,
3. Autonomous deliveries of materiel in-theater, and
4. Human-machine teaming.

Every combat zone will constitute a hostile environment for UAV operations, especially in the presence of GPS spoofing/jamming technologies. Combat UAVs will have to be able to tolerate multiple simultaneous sensor failures. Moreover, in-theater urban flight operations bring with them all of the same difficulties mentioned above, but with the added challenge of violent enemy interdiction. Robust sensor fusion could automate the task of operational overwatch and provide mission-critical intelligence in a timely manner—but only if the vehicle can localize itself reliably in sensor-hostile environments.

1.2 Personal Motivation

This section is largely comprised of background information regarding certain experiences leading up to, and motivating, the research developed in my thesis. This section does not contain any technical material, and is included only to provide context to the larger work.

I first took an interest in unmanned aircraft in the fall of 2012, my sophomore year of college. Several of my friends and I founded the Cooperative Autonomous Robotics Design (CARD) team at Virginia Tech in order to pursue our shared interest in autonomous vehicles. Our core team consisted of a dozen students devoted to designing and competing with drones and other robotic vehicles. Our team, guided by my future graduate adviser, Dr. Kevin Kochersberger, entered two design competitions and brought home two awards for the university. We were also one of four university robotics teams selected by the Smithsonian Institution to participate in the opening

Figure 1.1: The CARD team logo.

ceremony for Robotics Week 2015. These early experiences with the team brought me into contact with new skills such as microcontroller programming, Proportional-Integral-Derivative (PID) controller design, basic mechatronics, and computer-aided design (CAD) modeling.

After two years of involvement with the CARD team, I applied for an internship at the National Institute of Aerospace[3] (NIA) in Hampton, Virginia. In the summer of 2014, I was part of a team of NIA researchers working on the Flying Donkey Challenge[4], an international engineering competition centered around the idea of "flying donkeys," full-sized autonomous airplanes capable of quickly carrying cargo between small airports in rural Africa. This competition, unfortunately now defunct, was divided into a number of sub-challenges focusing on different technical objectives such as precision landing and collision avoidance. Our team's goal was to design an inexpensive navigation system that could reliably guide unmanned aircraft during a Global Positioning System (GPS) blackout. This project introduced me to many of the technologies and techniques that would later become my major research interests, particularly the Robot Operating System[5] (ROS), Kalman Filtering, and sensor fusion.

Through my internship at the NIA, I met Dr. Danette Allen, head of the NASA Langley Autonomy Incubator. During my 2014–15 academic year, Dr. Allen sponsored the CARD team to design and build two autonomous multirotor delivery drones. These aircraft were capable of delivering 5-lb packages to distances of up to 2.5 miles (or 5 miles, round trip). In addition, these vehicles were able to land precisely on 1 m^2 April tags, a type of visual marker used in augmented reality applications. Following the completion of this project, I worked as a summer intern at the Autonomy Incubator, thereby further advancing my interest in Kalman filtering, sensor fusion, and visual localization.

During the summer of 2015, I began the research that evolved into my thesis project, studying Visual-Inertial Navigation (VIN) and the Unscented Kalman Filter (UKF). As I read more and more on these subjects, I became interested in the design of the algorithm underlying the UKF. Unlike many other formulations of the Kalman Filter, the UKF has a notably limited dependence on information about the system under scrutiny (this *system agnosticism* is discussed in more detail in Chapter 3). The more I learned about the UKF, the more I became excited about the idea of taking advantage of this limited dependence trait to build a minimalistic software interface by which a wide variety of disparate systems could be tracked and studied in a ROS framework.

[3] http://www.nianet.org
[4] http://www.flyingdonkey.org
[5] http://wiki.ros.org

I envisioned a "one-stop shopping" experience for massively reusable and customizable filtering profiles that could fulfill the needs of researchers and roboticists who may have little knowledge of state estimation techniques. This vision eventually drove my development of the `kalman_sense` ROS package, cementing my interest in unmanned aerial vehicle (UAV) state estimation processes and controls.

1.3 Organization of this Document

Prior Work

In Prior Work, we explore recent contributions to loosely coupled filter-based navigation and state estimation processes. We focus primarily on a number of impactful publications coming from the Autonomous Systems Lab[6] (ASL) at ETH Zurich[7] and the University of Pennsylvania's GRASP Lab[8]. We define the current state of the art in filter-based navigation and establish the research context in which this thesis exists.

Algorithm Design and Implementation

Because of the algorithmic nature of state estimation processes, we explore in detail the design and implementation of the `kalman_sense` ROS package. We discuss plant model abstraction as well as code organization and data flow and then summarize the process by which one could extend `kalman_sense`'s functionality and the advantages of system-agnostic algorithm design.

Experimental Design

In this section, we first establish the goals of the testing regimen and then discuss the real-world execution of these goals. We discuss important statistical methods for characterizing the system's effectiveness as well as data collection procedures and post-processing. The system's physical testing infrastructure is explored in detail.

[6] `www.asl.ethz.ch`
[7] *Eidgenössische Technische Hochschule Zürich*, the Swiss Federal Institute of Technology in Zurich.
[8] `www.grasp.upenn.edu`

Experimental Results

In Experimental Results, we evaluate the system's performance during testing and seek out any limiting factors that influence estimation accuracy. We probe for possible improvements to the algorithm and provide a notional understanding of the system's theoretical effectiveness in real-world scenarios.

Conclusions and Future Work

We summarize the contributions made in this thesis, the effectiveness of the `kalman_sense` ROS package, and the insights acquired during programming and testing. We explore avenues for future research as well as possible improvements to the existing system.

Chapter 2

Prior Work

2.1 Development of the Unscented Kalman Filter

Since the late 1990s, the Unscented Kalman Filter (UKF) has been a frequent topic of interest in the field of guidance, navigation, and controls. This extension of the Kalman Filter (explored in detail in Chapter 3) appeals to roboticists and controls engineers because it allows the preservation of nonlinear behavior in both the state propagation and measurement prediction steps. Through its use of the unscented transform (discussed further below), the UKF retains not only the first and second moments of the state distribution—the mean and covariance—but also the third moment, the skew. It is this retention of higher order information, coupled with true nonlinear behavior, that makes the Unscented Kalman Filter such a desirable tool for motion tracking and localization.

In [1], Julier and Uhlmann presented a nonlinear estimation approach for the Kalman Filter, originally developed in Uhlmann's doctoral dissertation. Recognizing that most applications for autonomous navigation are fundamentally nonlinear in both their dynamics and their observation models, Julier and Uhlmann proposed the use of a set of discretely sampled "sigma points" to determine the mean and covariance of a probability distribution. By recasting the prediction and correction steps of the Kalman Filter in the form of unscented transforms (UTs), this new filter eliminates the need to calculate Jacobian matrices. Julier and Uhlmann argued that for this reason their formulation was easier to implement than the EKF and went on to suggest that its use could supplant the EKF in virtually all applications, linear or nonlinear.

In [2], Julier acknowledges that the (linear) Kalman Filter has been used successfully in many

nonlinear scenarios, but notes that the use of only the first two moments of the state estimate sigma points results in the neglect of all higher order information (that is, third-order moments, or "skew"), a potentially rich source of new and useful information relating to symmetry of the state estimate. By extending the sigma point selection scheme of the conventional unscented transform, Julier was able to present a tractable but computationally complex extension of the Kalman Filter that could predict not only the first two moments of a sigma point distribution but also the skew. Though formulated initially for unimodal distributions, Julier stated that the approach could, with additional mathematical considerations, be generalized for use with multimodal distributions. Julier's contention was that the use of higher order information could promote better performance levels in autonomous vehicle navigation. The utility of maintaining and utilizing higher order information through the use of skewed filtering was assessed in a realistic tracking scenario. However, the results were somewhat disappointing as the change in performance was only marginal, presumably due to the linearity of the filter's update rule. Accordingly, research in this area continues, including examination of the use of nonlinear update rules in the filtering process.

Julier describes a novel approach in [3] to modifying the UT state estimation method. In the new approach, Julier takes the additional step of introducing a framework for scaling sigma points as part of the state estimation process. The general framework of the new methodology allows preservation of the first two moments of any set of sigma points, thus providing a construct for limiting values to either the conventional unscented transform or the modified (scaled) transform. Providing detailed mathematical validations, Julier demonstrates that the new scaling algorithm is computationally manageable in that it is, in essence, little more than the conventional unscented transformation algorithm with the addition of a simple post-processing step, the only difference being the inclusion of an extra correction term. Thus, the new algorithm's computational and storage costs are similar to that of the non-scaled transformation. The performance level of the scaled UT is thus demonstrably superior to the unscaled UT for propagating the two lower-order moments of a sigma point distribution.

In [4], Julier and Uhlmann discuss the application of the EKF as an estimation algorithm and the associated difficulties in doing so. Because the EKF is fundamentally a linearizing approach to estimation, its effectiveness is thus tied to the veracity of the local linearity assumption for the system under scrutiny. These limitations led to the development of the UT for nonlinear applications. In this paper, Julier and Uhlmann describe the UT and its benefits, including

easier implementation and improved accuracy over the EKF. The UT offers greater accuracy and reliability by applying higher order information, using sigma point sampling, to the traditional mean and covariance information associated with linear applications. Julier and Uhlmann provide examples, which may be tailored to various process and observation models, that show how the UT overcomes the limitations of the EKF.

2.2 Visual-Inertial Navigation

Recent advancements in computing power have given rise to a plethora of once-inaccessible algorithms for visual localization and state estimation. Simultaneous Localization and Mapping (SLAM) algorithms have been one of the chief beneficiaries of this technological bounty. For example, the rise of Graphics Processing Units (GPUs) has allowed for tremendous increases in the speed of SLAM as well as Visual Odometry (VO) programs. Improved state estimation methods, such as the UKF, have grown in popularity at the same time. The convergence of these research avenues lies in Visual-Inertial Navigation (VIN), an approach to navigation using any number of methods to fuse either raw camera data or the outputs of vision-based algorithms with inertial sensor readings. In [5], Donavanik et al. provide a concise survey of the state of the art in VIN at the time of this writing. In this article, the researchers discuss many of the current challenges in robotic VIN, including those stemming from the use of particular SLAM algorithms and EKF-based frameworks, such as consistency of state estimates over time. As described in the article,

> This problem [of consistency] concerns the EKF as it uses linearization of a non-linear model, which introduces errors that in turn will lead to inconsistency. [...] Over time, this error can accumulate. A possible mitigation is to use the Unscented Kalman Filter (UKF).

At this juncture, we will briefly explore recent advancements in VIN, paying particular attention to filter-based methods and their accompanying sensor frameworks.

In [6], Klein and Murray proposed a method for tracking a handheld camera in unknown environments for use in small augmented reality (AR) workspaces. In contrast to many previous SLAM-based approaches to camera tracking, Klein and Murray split the tracking and mapping functions into two separate computational tasks. They performed these tasks on a dual-core computer utilizing parallel threads, with one thread directly tracking erratic motion of the

handheld camera and the other thread constructing a 3D map of the environment. Through the use of this Parallel Tracking and Mapping (PTAM) algorithm, Klein and Murray were able to take advantage of computationally expensive batch-optimization techniques for map reconstruction, techniques which were rarely used in real-time applications previously. This, in turn, allowed Klein and Murray to forego the common approach of creating a sparse map of high quality features in favor of a much denser map containing features that could vary widely in quality. The resulting system could produce detailed maps tracking thousands of features at frame-rate and could recover gracefully from a variety of intermittent tracking failures. That being said, the researchers made certain relaxing assumptions regarding the scenes to be tracked. PTAM, by nature of its orientation toward AR applications, operates best in small, static, planar environments (such as on the surface of a desk or the floor of an office). PTAM's value to the robotics community quickly became obvious due to its independence of *a priori* knowledge of the scene and its minimal initialization procedure (explored in Chapter 4).

Four years later, in [7], Weiss et al. presented a visual-inertial navigation system for autonomous UAV navigation which employed PTAM. The researchers presented the results of several experiments in which a UAV equipped with a single monocular camera and an IMU navigated through unknown environments without the aid of GPS satellites or other external sensing infrastructure. All calculations were performed in real time using an EKF framework, proving that this minimalist combination of sensors could be employed in real-world GPS-compromised flight scenarios to great effect. At approximately the same time, Shen et al. conducted experiments in [8] at the University of Pennsylvania GRASP Lab aimed at stable indoor flight and GPS-denied localization in constrained multi-floor environments with a similarly limited suite of purely onboard sensors. The research distinguishes itself by emphasizing the use of onboard sensors only, as well as fully autonomous, real-time internal computational capabilities, with no hands-on user interaction beyond basic high-level commands. The research extends to multi-floor UAV navigation with loop closure. It also addresses specially designed controllers to help compensate for sudden changes in wind velocity and air flow as the UAV traverses constrained low-clearance areas with potentially strong aerodynamic disturbances.

In [9], Weiss and Siegwart went on to tackle the problem of metric scale in monocular VIN systems. The researchers developed a general algorithm that provides metric scale to monocular visual odometry and monocular SLAM systems using inertial measurement unit (IMU) data. The authors accomplished the development of the metric scale by the addition of a 3-axis

accelerometer and 3-axis gyroscope to track 3D motion and to correct visual scale drift. Weiss and Siegwart created a modular solution based on their existing EKF framework and provided both simulated and empirical results. They also provided an in-depth analysis of their system's applications, versatility, and reliability for real-time visual odometry and SLAM.

In 2012, Weiss et al. built upon this metric scale algorithm to present a versatile sensor fusion framework for autonomous flight [10]. Due to latency, noise, and arbitrary scaling within the output of a UAV's sensors, it is both impractical and ill-advised to incorporate this sensor output for position control without calibration or post-processing. The researchers address these problems using an EKF-SLAM formulation which fuses pose measurements with inertial sensor data. In doing so, they not only estimate pose and velocity of the UAV, but also estimate the sensor biases, correct the scale of position measurements, and perform inter-sensor self-calibration in real time. Their research demonstrates that the proposed framework is capable of running entirely onboard a UAV and performing state prediction at a rate of 1 kHz. Their results illustrate that this approach is able to handle measurement delays (up to 500 ms), sensor noise (with positional standard deviation up to 20 cm), and slow update rates (as low as 1 Hz) while still allowing dynamic maneuvers. The researchers also present a detailed quantitative performance evaluation of the system under the influence of different disturbance parameters and different sensor setups to highlight the versatility of their approach. That same year, Weiss et al. further developed the VIN system in [11], adding a speed-estimation module to turn the monocular camera into a metric body-speed sensor. They then demonstrated how this module could be used for self-calibration of the UAV's onboard sensor suite in real time.

Shortly thereafter, Huang et al. presented solutions in [12] to two UKF limitations that exist in current state-of-the-art SLAM systems. Specifically, the researchers addressed the problems of cubic complexity in the number of state pose estimates, and the inconsistencies in those estimates caused by a mismatch between the observability properties of statistically-linearized UKF systems and the observability properties of nonlinear systems. To address the problem of cubic complexity, they introduced a novel sampling strategy which produces a constant computational cost. This sampling method, while linear in the prediction phase, is quadratic in the update phase. Although this new sampling strategy was primarily proposed for resolving the cubic complexity SLAM problem, the researchers stressed that it has potential usefulness in other nonlinear estimation applications. To address the problem of inconsistency in state estimations, Huang et al. proposed a new UKF algorithm which, due to the imposition of

observability constraints, ensures that the linear regression computations of the modified UKF system produce results similar to those of nonlinear SLAM systems and, in the process, provide improved accuracy and consistency in state estimations. Importantly, the researchers validated their results with both real-world and simulation experiments.

In 2013, Lynen et al. presented a generic framework in [13] based on the EKF-SLAM system developed in [10] which was shown to be more robust during sensor blackouts and to be self-correcting in scale. The researchers demonstrated that their Multi-Sensor Fusion EKF (MSF-EKF) framework was capable of processing measurements from an unlimited number of sensors, as well as sensor types, while simultaneously performing automatic self-calibrations of the overall sensor suite. It was the design of this software framework, which the researchers released as open source software shortly after publication, that inspired many of the design decisions behind the `kalman_sense` ROS package.

In [14], Rogers et al. presented a methodology for overcoming some of the constraining conditions encountered in a GPS-guided autonomous robotic system, such as occlusion (blocking of GPS signals) and multipath distortion (reception of indirect signals due to environmental reflections) and potentially to ameliorate the effects of jamming or spoofing resulting from adversarial activities. Specifically, the methodology incorporated GPS measurements into a feature-based mapping system, thus providing geo-referenced coordinates allowing for better execution of high-level missions and providing the ability to correct accumulated mapping errors over the course of long-term operations in both indoor and outdoor environments.

In [15], Faessler et al. reported on the development and demonstration of a low-cost, low-weight, vision-based quadrotor UAV with onboard sensing, computation, and control capabilities. These onboard capabilities eliminated reliance on external positioning systems such as GPS or motion capture systems. This development moved the UAV from its current line-of-sight control state to wireless communications with the ability to execute intricate processes autonomously and to transmit live feedback to a user. Reporting on both indoor and outdoor experiments, the researchers believe that such a vehicle potentially would be a great enhancement in search-and-rescue missions, disaster response, and remote inspection of terrain.

Chapter 3

Algorithm Design and Implementation

3.1 UKF Formulation

In this section, we develop a formulation of the Unscented Kalman Filter for estimating the state of a quadcopter based in large part on example MATLAB code from [16]. We begin with the two overarching phases of the algorithm, prediction and correction. Then we develop a kinematic process model which describes the vehicle's motion over time. After this, we develop an observation model which describes the relationship between the vehicle's state and the vehicle's sensor readings. Finally, we explore noise models which describe the Gaussian noise present in both the process and observation models.

3.1.1 Prediction Step

We begin by defining the following quantities:

$$\mathbf{p} = \{x,\ y,\ z\}^T \tag{3.1}$$

$$\mathbf{q} = \{q_x,\ q_y,\ q_z,\ q_w\}^T \tag{3.2}$$

$$\mathbf{v} = \{\dot{x},\ \dot{y},\ \dot{z}\}^T \tag{3.3}$$

$$\mathbf{\Omega} = \{\omega_x,\ \omega_y,\ \omega_z\}^T \tag{3.4}$$

$$\mathbf{a} = \{\ddot{x},\ \ddot{y},\ \ddot{z}\}^T \tag{3.5}$$

The vectors $\mathbf{p}$, $\mathbf{v}$, and $\mathbf{a}$ represent the vehicle's position, velocity, and acceleration, respectively. The quaternion[1] $\mathbf{q}$ represents the vehicle's orientation, and the vector $\boldsymbol{\Omega}$ represents the vehicle's angular rates. We now define the state vector $\mathbf{x}$ as

$$\mathbf{x} = \left\{\mathbf{p}^T,\ \mathbf{q}^T,\ \mathbf{v}^T,\ \boldsymbol{\Omega}^T,\ \mathbf{a}^T\right\}^T \tag{3.6}$$

and let $n = 16$ represent the number of state variables.

Let $\mathbf{P} \in \mathbb{R}^{n\times n}$ be the covariance matrix associated with the state $\mathbf{x}$. A set of $2n+1$ sigma points is then derived from $\mathbf{x}$ and $\mathbf{P}$ as

$$\begin{aligned}
\boldsymbol{\chi}^0_{k|k-1} &= \mathbf{x}_{k-1|k-1} \\
\boldsymbol{\chi}^i_{k|k-1} &= \mathbf{x}_{k-1|k-1} + \left(\sqrt{(n+\lambda)\,\mathbf{P}_{k-1|k-1}}\right)_i, && i = 1,\ \ldots,\ n \\
\boldsymbol{\chi}^i_{k|k-1} &= \mathbf{x}_{k-1|k-1} - \left(\sqrt{(n+\lambda)\,\mathbf{P}_{k-1|k-1}}\right)_{i-n}, && i = n+1,\ \ldots,\ 2n
\end{aligned} \tag{3.7}$$

where $\boldsymbol{\chi}^i$ is the i-th sigma point and $\left(\sqrt{(n+\lambda)\,\mathbf{P}_{k-1|k-1}}\right)_i$ is the i-th column of the square root matrix $\sqrt{(n+\lambda)\,\mathbf{P}_{k-1|k-1}}$. The square root matrix $\mathbf{A}$ of a matrix $\mathbf{B}$ is defined here as

$$\mathbf{A}\mathbf{A}^T = \mathbf{B} \tag{3.8}$$

and is evaluated using Cholesky decomposition[2] for computational efficiency.

To predict the next state, the sigma points are propagated through the nonlinear process function f (defined in Section 3.1.3):

$$\boldsymbol{\chi}^i_{k|k-1} = f\left(\boldsymbol{\chi}^i_{k-1|k-1}\right), \quad i = 0,\ \ldots,\ 2n \tag{3.9}$$

These transformed sigma points are then used to determine the predicted state $\hat{\mathbf{x}}_{k|k-1}$ and its

[1] All quaternions are represented here according to the convention used in equation 3.2, where q_w is the scalar component and is always placed in the last position. This convention was chosen to maintain consistency with the Eigen library's internal representation of quaternions. Unless otherwise noted, all quaternion quantities treated herein are assumed to be *unit* quaternions.

[2] https://en.wikipedia.org/wiki/Cholesky_decomposition#The_Cholesky_algorithm

associated covariance $\mathbf{P}_{k|k-1}$ as

$$\hat{\mathbf{x}}_{k|k-1} = \sum_{i=0}^{2n} W_s^i \boldsymbol{\chi}_{k|k-1}^i \tag{3.10}$$

$$\mathbf{P}_{k|k-1} = \left(\sum_{i=0}^{2n} W_c^i \left(\boldsymbol{\chi}_{k|k-1}^i - \hat{\mathbf{x}}_{k|k-1} \right) \left(\boldsymbol{\chi}_{k|k-1}^i - \hat{\mathbf{x}}_{k|k-1} \right)^T \right) + \mathbf{Q} \tag{3.11}$$

where $\mathbf{Q}$ is the process noise covariance matrix (defined in Section 3.1.5).

The state weights W_s and covariance weights W_c used in equations 3.10 and 3.11 (and later in equations 3.20, 3.21, and 3.22) are defined as

$$\begin{aligned} W_s^0 &= \frac{\lambda}{n+\lambda} \\ W_c^0 &= \frac{\lambda}{n+\lambda} + \left(1 - \alpha^2 + \beta\right) \\ W_s^i &= W_c^i = \frac{1}{2\left(n+\lambda\right)}, \quad i = 1, \ldots, 2n \end{aligned} \tag{3.12}$$

where the constants α, β, κ, and λ are defined as

$$\alpha = 0.75 \tag{3.13}$$

$$\beta = 2 \tag{3.14}$$

$$\kappa = 0 \tag{3.15}$$

$$\lambda = \alpha^2 \left(n + \kappa\right) - n = -7 \tag{3.16}$$

These constants control the spread of the sigma points chosen within the filter. According to Wan et al. in [17], setting $\beta = 2$ is optimal for a Gaussian state distribution, and κ is commonly set to zero. The value of α is usually small and positive. Here, we chose α by trial and error after observing the rate at which the filter converged in a number of preliminary experiments. We found that choosing $0 < \alpha \leq 0.5$ made the filter slow to converge. The filter's estimates would lag behind the pose sensor's readings by an interval that was sometimes several seconds in length and, in steady state, would undershoot the pose sensor. Conversely, choosing $\alpha \geq 1$ caused the filter to diverge. At $\alpha \approx 0.75$, the filter had no distinguishable lag and was highly stable.

3.1.2 Correction Step

Given the belief defined by $\mathbf{x}_{k|k-1}$ and $\mathbf{P}_{k|k-1}$, we compute $2n+1$ sigma points again as

$$\begin{aligned}
\boldsymbol{\chi}^0_{k|k-1} &= \mathbf{x}_{k|k-1} \\
\boldsymbol{\chi}^i_{k|k-1} &= \mathbf{x}_{k|k-1} + \left(\sqrt{(n+\lambda)\,\mathbf{P}_{k|k-1}}\right)_i, && i = 1,\ \ldots,\ n \\
\boldsymbol{\chi}^i_{k|k-1} &= \mathbf{x}_{k|k-1} - \left(\sqrt{(n+\lambda)\,\mathbf{P}_{k|k-1}}\right)_{i-n}, && i = n+1,\ \ldots,\ 2n
\end{aligned} \tag{3.17}$$

Next, the sigma points are projected into the sensor space through the observation function h (defined in Section 3.1.4):

$$\boldsymbol{\gamma}^i_k = h\left(\boldsymbol{\chi}^i_{k|k-1}\right), \quad i = 0,\ \ldots,\ 2n \tag{3.18}$$

Let $m = 7$ represent the number of measurements taken from each PTAM message. Each of these messages is mathematically interpreted as an m-dimensional measurement vector $\mathbf{z}_k$ of the form

$$\mathbf{z}_k = \left\{x,\ y,\ z,\ q_x,\ q_y,\ q_z,\ q_w\right\}^T_{\text{meas.}} \tag{3.19}$$

The predicted measurement vector $\hat{\mathbf{z}}_k$ and measurement noise covariance $\mathbf{P}_{zz}$ are then computed as

$$\hat{\mathbf{z}}_k = \sum_{i=0}^{2n} W^i_s \boldsymbol{\gamma}^i_k \tag{3.20}$$

$$\mathbf{P}_{zz} = \left(\sum_{i=0}^{2n} W^i_c \left(\boldsymbol{\gamma}^i_k - \hat{\mathbf{z}}_k\right)\left(\boldsymbol{\gamma}^i_k - \hat{\mathbf{z}}_k\right)^T\right) + \mathbf{R} \tag{3.21}$$

where $\mathbf{R}$ is the measurement noise covariance matrix (defined in Section 3.1.6). The state-measurement cross-covariance $\mathbf{P}_{xz}$ is then defined as

$$\mathbf{P}_{xz} = \sum_{i=0}^{2n} W^i_c \left(\boldsymbol{\chi}^i_{k|k-1} - \hat{\mathbf{x}}_{k|k-1}\right)\left(\boldsymbol{\gamma}^i_k - \hat{\mathbf{z}}_k\right)^T \tag{3.22}$$

Next, we compute the Kalman gain $\mathbf{K}_k$ per the definition

$$\mathbf{K}_k = \mathbf{P}_{xz}\mathbf{P}^{-1}_{zz} \tag{3.23}$$

The corrected state $\hat{\mathbf{x}}_{k|k}$ is the sum of the predicted state and the innovation, weighted by $\mathbf{K}_k$:

$$\hat{\mathbf{x}}_{k|k} = \hat{\mathbf{x}}_{k|k-1} + \mathbf{K}_k \left(\hat{\mathbf{z}}_k - \mathbf{z}_k\right) \tag{3.24}$$

The corrected covariance $\mathbf{P}_{k|k}$ is the difference between the predicted state covariance $\mathbf{P}_{k|k-1}$ and the predicted measurement covariance, weighted by the Kalman gain:

$$\mathbf{P}_{k|k} = \mathbf{P}_{k|k-1} - \mathbf{K}_k \mathbf{P}_{zz} \mathbf{K}_k^T \tag{3.25}$$

3.1.3 Process Model

Here we develop a process model for a small rotorcraft based on a similar model presented in [11]. This process model propagates the vehicle's state forward in time by performing a number of integration operations on the linear accelerations and angular rates measured by the IMU.

To determine the orientation of the vehicle at some time k, we integrate the measured angular rate vector $\mathbf{\Omega}_{\text{meas.}}$ per the relationship given in [18],

$$\mathbf{q}_k = \mathbf{q}_{k-1} + \frac{1}{2} \Theta\left(\mathbf{\Omega}_k\right) \mathbf{q}_{k-1} \Delta t \tag{3.26}$$

where $\Theta\left(\mathbf{\Omega}_k\right) \in \mathbb{R}^{4\times 4}$ is the quaternion multiplication matrix defined by

$$\Theta\left(\mathbf{\Omega}_k\right) = \begin{bmatrix} 0 & \omega_z & -\omega_y & \omega_x \\ -\omega_z & 0 & \omega_x & \omega_y \\ \omega_y & -\omega_x & 0 & \omega_z \\ -\omega_x & -\omega_y & -\omega_z & 0 \end{bmatrix} \tag{3.27}$$

and the vehicle's current angular velocity vector is measured directly by the gyroscope:

$$\mathbf{\Omega}_k = \mathbf{\Omega}_{\text{meas.}} \tag{3.28}$$

To calculate the vehicle's current acceleration, we first subtract gravity from the measured acceleration vector $\mathbf{a}_{\text{meas.}}$ and then rotate the difference into the inertial frame:

$$\mathbf{a}_k = R^g_{b,\ k-1} \left(\mathbf{a}_{\text{meas.}} - \mathbf{g}\right) \tag{3.29}$$

Here $R^{g}_{b,\,k-1}$ is the rotation matrix representation of the previous orientation quaternion $\mathbf{q}_{k-1}$ and $\mathbf{g}$ is the vector representation of gravity in the global frame. For a given unit quaternion $\mathbf{q}$, the rotation matrix R^{g}_{b}, given in [19], is

$$R^{g}_{b} = \begin{bmatrix} 1-2q_y^2-2q_z^2 & 2q_xq_y-2q_zq_w & 2q_xq_z+2q_yq_w \\ 2q_xq_y+2q_zq_w & 1-2q_x^2-2q_z^2 & 2q_yq_z-2q_xq_w \\ 2q_xq_z-2q_yq_w & 2q_yq_z+2q_xq_w & 1-2q_x^2-2q_y^2 \end{bmatrix} \tag{3.30}$$

With the vehicle's current acceleration known, we can compute the current velocity of the vehicle by integrating the vehicle's average acceleration between the current and previous time steps:

$$\mathbf{v}_k = \mathbf{v}_{k-1} + \frac{1}{2}\left(\mathbf{a}_k + \mathbf{a}_{k-1}\right)\Delta t \tag{3.31}$$

Similarly, we compute the vehicle's current position using the vehicle's average velocity:

$$\mathbf{p}_k = \mathbf{p}_{k-1} + \frac{1}{2}\left(\mathbf{v}_k + \mathbf{v}_{k-1}\right)\Delta t \tag{3.32}$$

Together, equations 3.26, 3.28, 3.29, 3.31, and 3.32 constitute the nonlinear process function f:

$$f\left(\mathbf{x}_{k-1|k-1}\right) = \begin{Bmatrix} \mathbf{p}_{k-1} + \frac{1}{2}\left(\mathbf{v}_k + \mathbf{v}_{k-1}\right)\Delta t \\ \mathbf{q}_{k-1} + \frac{1}{2}\Theta\left(\mathbf{\Omega}_k\right)\mathbf{q}_{k-1}\Delta t \\ \mathbf{v}_{k-1} + \frac{1}{2}\left(\mathbf{a}_k + \mathbf{a}_{k-1}\right)\Delta t \\ \mathbf{\Omega}_{\text{meas.}} \\ R^{g}_{b,\,k-1}\left(\mathbf{a}_{\text{meas.}} - \mathbf{g}\right) \end{Bmatrix} \tag{3.33}$$

For purposes of noise modeling, the process function f is modeled according to the relationship

$$\mathbf{x}_{k|k-1} = f\left(\mathbf{x}_{k-1|k-1}\right) + \boldsymbol{\delta}_k \tag{3.34}$$

where $\boldsymbol{\delta}_k \sim \mathcal{N}\left(0,\ \mathbf{Q}\right)$ is additive Gaussian-distributed process noise, assumed to have zero mean and covariance $\mathbf{Q}$ (explored further in Section 3.1.5).

3.1.4 Observation Model

Before presenting the observation function h, we will first explore several corrective measures employed to improve the numerical stability of the filter. The first of these is an algorithmic approach to quaternion continuity checking. The second is a derived measurement of the vehicle's velocity designed to prevent drift in the absence of direct velocity measurements.

Quaternion Continuity Correction

A defect was discovered in the ROS implementation of PTAM during preliminary testing. PTAM's quaternion estimates switch sign without warning after the camera rotates approximately 270 degrees, clockwise or counterclockwise. This aberrant behavior was provoked predictably in dozens of early tests in multiple testing areas, precluding the possibility of a one-off mapping error caused by a particular environment.

Quaternions are defined mathematically such that any quaternion $\mathbf{q}$ and its negative, $-\mathbf{q}$, encode precisely the same *orientation*, but opposite *rotations*. If $\mathbf{q}$ encodes a clockwise rotation of 90 degrees about the z-axis, $-\mathbf{q}$ will encode a counterclockwise 270-degree rotation about the same axis. The final orientation is the same in either case, but the two quaternions imply rotations in opposite directions. This has major implications for motion tracking. For example, when estimating the orientation of a vehicle moving in the xy-plane, such a sign flip would be visualized as an instantaneous 360-degree rotation about the z-axis, implying a massive angular acceleration that never took place.

A simple way to detect this type of errant behavior is to check, upon receipt of each new quaternion $\mathbf{q}_k$, whether $-\mathbf{q}_k$ implies a smaller rotation from the previous quaternion $\mathbf{q}_{k-1}$. Given that PTAM produces pose estimates at a rate of about 20 Hz, we assume only small-angle rotations between pose estimates. Thus, the smaller of the two rotations should always come from the correct quaternion. We can compare the relative magnitude of each possible rotation by taking the norm of the "delta-quaternions" $\mathbf{q}_{k-1} - \mathbf{q}_k$ and $\mathbf{q}_{k-1} - \left(-\mathbf{q}_k\right)$. The smaller delta-quaternion is associated with the correct quaternion estimate. We implement this logic using Algorithm 1.

Algorithm 1 Check for continuity between quaternion estimates.

1: **function** CHECKQUATERNIONCONTINUITY($\mathbf{q}_{k-1}$, $\mathbf{q}_k$)
2: sum $\leftarrow \left|\left|\mathbf{q}_{k-1} + \mathbf{q}_k\right|\right|$
3: diff $\leftarrow \left|\left|\mathbf{q}_{k-1} - \mathbf{q}_k\right|\right|$
4: **if** diff $\geq$ sum **then**
5: $\mathbf{q}_k \leftarrow -\mathbf{q}_k$
6: **return** $\mathbf{q}_k$

Pseudovelocity Correction

Of the various states being measured, only the acceleration vector **a** and angular rate vector $\boldsymbol{\Omega}$ are measured directly by the IMU:

$$\mathbf{x} = \left\{\mathbf{p}^T,\ \mathbf{q}^T,\ \mathbf{v}^T,\ \underbrace{\boldsymbol{\Omega}^T,\ \mathbf{a}^T}_{\text{IMU}}\right\}^T$$

The vehicle's position **p** and orientation **q** are measured directly by PTAM:

$$\mathbf{x} = \left\{\underbrace{\mathbf{p}^T,\ \mathbf{q}^T}_{\text{PTAM}},\ \mathbf{v}^T,\ \boldsymbol{\Omega}^T,\ \mathbf{a}^T\right\}^T$$

The vehicle's velocity **v**, however, is not measured by either the IMU or PTAM. There is no sensor in place by which **v** can be directly measured. Therefore, velocity must be determined through a derived measurement.

Because velocity is predicted by integrating the vehicle's acceleration (Section 3.1.3), Gaussian noise in the accelerometer can cause unbounded drift in the velocity. The resulting erroneous velocity then propagates through each successive prediction step, causing drift in the estimated position *regardless of corrective position measurements from PTAM*. In this situation, the filter produces position estimates which drift at a constantly increasing rate between PTAM messages. The Kalman Filter then corrects the position each time that a PTAM message arrives, only to have it drift again a fraction of a second later when the next IMU message arrives. This kind of intermittent drift produces sharp "bounces" in the vehicle's estimated trajectory and makes the filter's output numerically unstable.

To prevent this potentially catastrophic drift, velocity is estimated by numerical differentiation of the vehicle's position during each correction step. When a PTAM message is received, the position reading from the last PTAM message is subtracted from the current position reading.

This change in position is then divided by the time step between the two PTAM messages, Δt_P:

$$\mathbf{v}_k = \frac{\mathbf{p}_{k,\text{ meas.}} - \mathbf{p}_{k-1,\text{ meas.}}}{\Delta t_P} \tag{3.35}$$

This pseudovelocity correction prevents unbounded velocity drift, which in turn makes the filter's output numerically stable. Including this derived measurement of velocity in each correction step prevents the predicted velocities from drifting as quickly as they would otherwise. This, in turn, increases the reliability of the predicted velocities and makes the IMU-derived state estimates that are computed between PTAM corrections more trustworthy.

Observation Modeling

After implementing the quaternion and velocity corrections developed above, the m-dimensional measurement vector is augmented with the new pseudovelocity, giving it dimension $m_a = 10$:

$$\mathbf{z}_k = \left\{x,\ y,\ z,\ q_x,\ q_y,\ q_z,\ q_w,\ \dot{x},\ \dot{y},\ \dot{z}\right\}^T_{\text{meas.}} \tag{3.36}$$

The observation function h is then

$$h\left(\mathbf{x}_{k|k-1}\right) = \mathbf{H}\mathbf{x}_{k|k-1} \tag{3.37}$$

The matrix $\mathbf{H}$ is defined as

$$\mathbf{H} = \begin{bmatrix} \mathbf{I}_{m_a \times m_a} & \mathbf{0}_{m_a \times (n-m_a)} \end{bmatrix} \tag{3.38}$$

where $\mathbf{I}_{m_a \times m_a}$ is a 10×10 identity matrix and $\mathbf{0}_{m_a \times (n-m_a)}$ is an empty 10×6 matrix. We model the observation function h according to the relationship

$$\hat{\mathbf{z}}_k = h\left(\mathbf{x}_{k|k-1}\right) + \boldsymbol{\epsilon}_k \tag{3.39}$$

where $\boldsymbol{\epsilon}_k \sim \mathcal{N}\left(0,\ \mathbf{R}\right)$ is additive Gaussian measurement noise assumed to have zero mean and covariance $\mathbf{R}$ (explored in Section 3.1.6).

3.1.5 Process Noise Covariance Matrix

The process noise covariance matrix $\mathbf{Q} \in \mathbb{R}^{n \times n}$ is symmetric and positive definite. We assume that its nonzero coefficients are a function of multiple integrations between the linear accelerations

and angular velocities measured by the vehicle's 3-axis accelerometer and 3-axis gyroscope. We further assume that all axes of the accelerometer and gyroscope exhibit white noise with known variances[3] $\sigma_a^2 = 8.66125 \times 10^{-6}$ m^2/s^4 and $\sigma_\omega^2 = 1.21847 \times 10^{-7}$ rad^2/s^2, as reported by the IMU during operation. We further assume that no cross-axis covariance exists between any two accelerometer axes or any two gyroscope axes[4].

Because the IMU runs at a constant rate of 250 Hz, we assume $\mathbf{Q}$ is time-invariant. We further assume that the nonzero covariance terms associated with the unmeasured quantities $\mathbf{p}$, $\mathbf{q}$, and $\mathbf{v}$ are related to the variance terms by integration over a constant time step Δt. Let the variance terms for the accelerations and angular rates be set as

$$Q_{\ddot{x},\,\ddot{x}} = \sigma_a^2 \tag{3.40}$$

and

$$Q_{\omega_x,\,\omega_x} = \sigma_\omega^2 \tag{3.41}$$

respectively. We can now estimate the other nonzero terms of $\mathbf{Q}$ according to their integral relationships with the acceleration and angular velocity measurements. In the example case of the x-velocity, the variance would be computed as

$$Q_{\dot{x},\,\dot{x}} = \int_0^{\Delta t} \sigma_a^2 \, dt \tag{3.42}$$

This process of integrating known noise coefficients to determine the covariance terms associated with integrated quantities, detailed in [20], proved ineffective in preliminary experiments, as the resulting $\mathbf{Q}$ matrix made the filter numerically unstable. To compensate, this matrix was replaced by a diagonal estimated $\mathbf{Q}$ in which all of the coefficients were overestimated in an attempt to account for any unknown sources of error. The original terms were on the order of 10^{-6} or smaller, whereas the new matrix was defined as

$$\mathbf{Q} = k_\mathbf{Q} \, \mathbf{I}_{n \times n} \tag{3.43}$$

with $k_\mathbf{Q} = 0.01$. This replacement produced numerically stable output and was used for all of the experimental trials detailed in Chapters 4 and 5.

[3] This claim of equal variances between axes is supported by the IMU data sheet.

[4] Also reported by the IMU.

3.1.6 Measurement Noise Covariance Matrix

The matrix $\mathbf{R} \in \mathbb{R}^{m_a \times m_a}$ models the covariance between measurements made by the pose sensor, PTAM. Like the process noise covariance matrix $\mathbf{Q}$, the measurement noise covariance matrix $\mathbf{R}$ is symmetric and positive definite. For the purposes of this experiment, $\mathbf{R}$ is assumed to be time-invariant due to nearly constant feature density in the experimental scene. Because the vehicle camera moves through the environment at an almost constant height (that is, at a nearly constant distance to the features it observes on the floor), the measurement noise covariance of the pose sensor is assumed to be approximately constant in time. The coefficients of $\mathbf{R}$ were estimated by numerically computing the autocovariances between PTAM measurements over time in a steady-state configuration. These variances were computed as being on the order of 10^{-3} or less for all terms. However, during preliminary experiments, the dense estimated covariance matrix caused the filter's output to drift uncontrollably. Therefore, we instead assumed a diagonal measurement noise covariance matrix and attempted to compensate by overestimating the magnitude of its terms. Setting $\mathbf{R}$ as

$$\mathbf{R} = k_{\mathbf{R}} \, \mathbf{I}_{m_a \times m_a} \tag{3.44}$$

with $k_{\mathbf{R}} = 0.01$ made the filter numerically stable, and thus was used for all experimental trials detailed in Chapters 4 and 5.

3.2 Software Design Considerations

Much of the impetus for creating the `kalman_sense` ROS package came from a desire to create a generic UKF framework for estimating the state of an arbitrary system using any number of relative and absolute sensors. To achieve this, the `kalman_sense` package is organized in an object-oriented manner around an overarching abstract class called `UnscentedKf` (Figure 3.1). This abstract class contains a number of member functions ("methods") performing the different mathematical operations defined in Section 3.1. These methods have been written in a generic manner to enable easy extension of `UnscentedKf` by subclasses containing concrete implementations of various systems. Currently, the package contains only one subclass, known as `QuadUkf`. This subclass contains methods and data structures related directly to estimating the state of a quadcopter or other rotorcraft UAV.

The `UnscentedKf` class encapsulates the generic mathematics of the UKF without knowledge

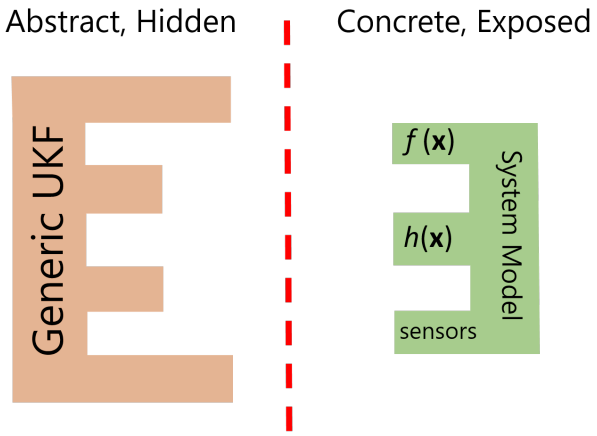

Figure 3.1: A conceptual drawing of the underlying software design. The UKF algorithm is broken into a generic abstract class and a concrete subclass encoding vehicle specifics.

of particular system constraints. This class does little other than matrix mathematics and is designed to take as input the number of a system's states n and its number of sensors m. With this information, `UnscentedKf` is able to populate a set of mean and covariance weights and then intelligently perform all of the requisite linear algebra for the UKF formulation. All other knowledge of particular states, sensors, vehicle geometry, and other metrics is hidden within subclasses such as `QuadUkf`.

`UnscentedKf` behaves in a manner similar to a Java interface in that it requires the extending class to supply functions codifying a process model and an observation model for the system under scrutiny. These two functions, along with n and m, form the entirety of what `UnscentedKf` "knows" about the vehicle. All other details, including the fact that the class is being used in a ROS environment, are hidden from `UnscentedKf`. Because of this encapsulation of system specifics, the `UnscentedKf` class could be considered *system agnostic*. This class implements only the bare-minimum functionality required for the Unscented Kalman Filter, and could thus be extended to any number of other systems, such as spacecraft, undersea vehicles, or automobiles. It is worth noting that `UnscentedKf`'s only software dependency is on the Eigen C++ linear algebra library[5], making it largely portable between operating systems.

The subclass (`QuadUkf` for the remainder of this thesis) handles all of the ROS communications for the given system. Specifically, this class has callback functions for receiving sensor data and is responsible for publishing state and covariance estimates. Figure 3.2 summarizes

[5] www.eigen.tuxfamily.org

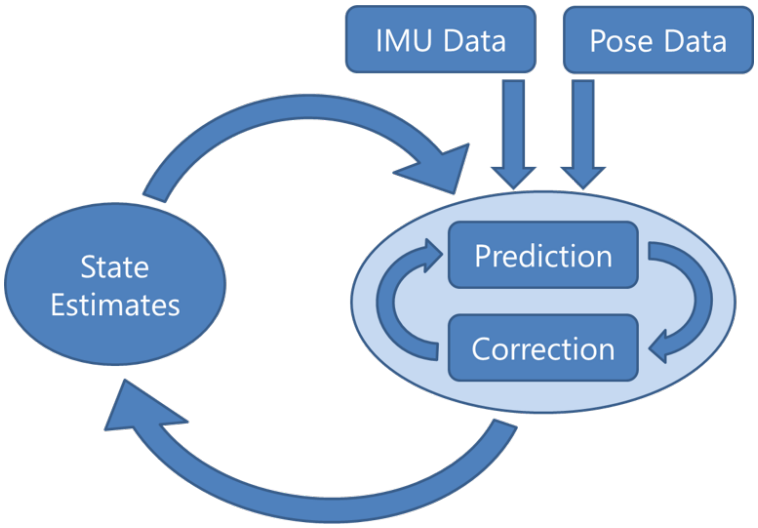

Figure 3.2: Data flow within the system. The arrival of IMU and pose messages triggers the callback functions which predict and correct the estimated state of the system. After each prediction or correction, a new state estimate is published and the system waits for new sensor data.

the data flow within the system. Upon arrival of an IMU message, the IMU callback function is triggered and the prediction step of the UKF is set in motion. When the system receives a pose message, the pose callback function is triggered and the correction step is set in motion. After every prediction and correction, a new belief is published and the system waits for further sensor data.

Chapter 4

Experimental Design

4.1 Testing Considerations

Before venturing further, we now summarize the goals of the UKF framework described previously, paying particular attention to the fundamental requirements and parameters underlying the operation of an unmanned aircraft system (UAS). This ROS package was designed with the intent of producing estimates of the state of a rotorcraft UAV in real time. Thus, the experiments testing the system's efficacy compare the filter's estimates to the ground truth as measured by Vicon. Vicon, like many commercially available motion capture systems, works by using a number of infrared cameras (usually 8–12) to track the positions of retroreflector balls in the cameras' field of view. Each camera is outfitted with infrared LED[1] bulbs which illuminate the area with infrared light. The retroreflector balls reflect this infrared light back to the cameras, allowing the balls to be tracked as they move. Sets of these retroreflectors can be grouped together in software to represent rigid objects, allowing the Vicon system to track not only the position of an object, but its orientation as well.

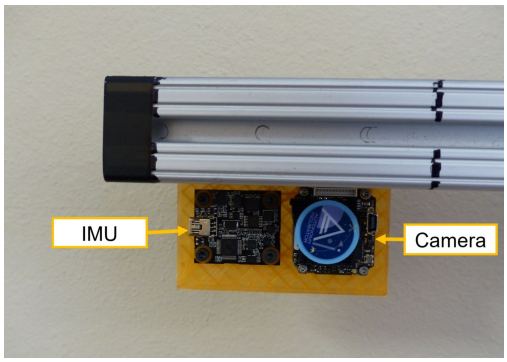

Figure 4.1: The 3D-printed sensor mount on the aluminum sliding rail.

The UKF framework depends upon two sensors: a global-shutter monocular camera and an IMU (Figure 4.1). The IMU used in this experiment contains a 3-axis accelerometer and

[1] "Light-emitting diode," https://en.wikipedia.org/wiki/Light-emitting_diode

Figure 4.2: The mobile test stand, fully assembled. The aluminum rail holds the sensor mount 1 meter above the floor and is marked with an adhesive metric ruler for measuring camera displacement during PTAM initialization.

3-axis gyroscope. To simulate both sensors moving through the scene in a manner reminiscent of hovering rotorcraft flight, a rolling test stand was constructed to carry the sensors safely throughout a large motion capture environment. Mounting the sensor suite on a large, steady, level platform (Figure 4.2) allows for a high degree of control over the accelerations and angular velocities felt by the IMU, as well as the motion captured by the ventral camera. In order to validate the UKF framework's effectiveness under theoretically ideal conditions, a modern laptop computer containing an Intel i7 processor with 16 GB of RAM[2] was used for all computations. The floor of the motion capture environment (Figure 4.3) was covered with rubber tiles and strewn with a mixture of April tags and Quick Response[3] (QR) codes in order to provide sufficient visual features for PTAM to track.

[2] "Random-access memory," https://en.wikipedia.org/wiki/Random-access_memory
[3] https://en.wikipedia.org/wiki/QR_code

Figure 4.3: The mobile test stand in the testing environment. The area shown is part of a larger motion capture facility. The floor is covered with rubber tiles, many of which are in turn covered with April tags and QR codes for visual geometry.

4.1.1 PTAM Anomalies

During preliminary experiments, the parameter α (Section 3.1.1) was tuned such that pose estimates from the UKF would almost exactly track those coming from PTAM. This is a desirable behavior given that, between PTAM and the IMU, PTAM will almost always be the more accurate sensor. However, because of this close coupling between PTAM's output and the UKF's output, idiosyncrasies in PTAM's behavior will have an outsized effect on the behavior of the filter. The experiments chosen to test the system's effectiveness were thus influenced largely by the known capabilities and limitations of the ROS PTAM implementation[4].

Previous experiments[5] have shown PTAM to lose tracking[6] frequently during large rotations. PTAM is particularly susceptible to losing tracking during *pure* rotations, in which the camera rotates in place without translational motion. Worse, whenever PTAM undergoes a rotation, it interprets this as a rotation coupled with an arc-like translation, regardless of whether any translation actually took place.

[4] From here on, "PTAM" refers to the particular ETHZ-ASL ROS implementation of PTAM used in the experiments (http://wiki.ros.org/ethzasl_ptam), not to the algorithm in general.

[5] Performed both by the author and by various other parties at NASA Langley, where the experiments described here took place.

[6] "Tracking" here refers to current pose knowledge.

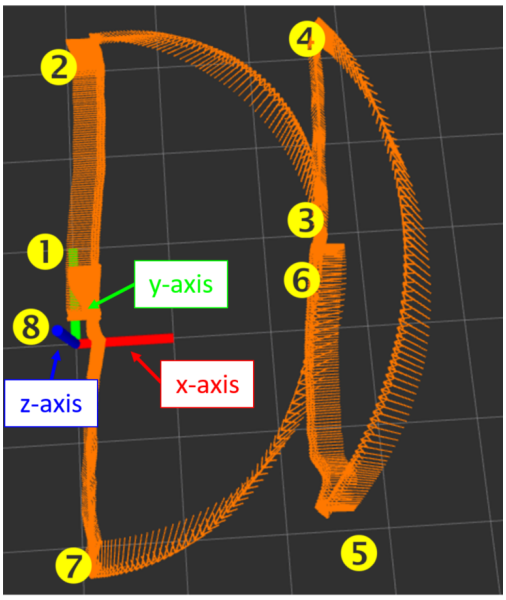

Figure 4.4: An Rviz visualization of PTAM's rotation defect. This image was created by moving the test stand in a 3 m square pattern, turning 90° CCW at each corner.

Figure 4.4 shows PTAM's susceptibility to losing tracking during rotation. This Rviz[7] visualization was produced by moving the test stand in a square trajectory and turning the test stand by 90 degrees at each corner. In the figure, the vehicle's trajectory is traced out by a series of orange arrows representing the vehicle's x-axis. The vehicle started at the origin (Point 1), then moved forward (in the $+y$ direction) in a straight line to Point 2. At Point 2, the test stand was rotated counterclockwise in place by 90 degrees, producing the arc between Points 2 and 3. At this point, the test stand was pointed in the $-x$ direction. As the figure shows, this arc extends *opposite* the camera's true direction of travel. After rotating 90 degrees, the vehicle moved in a straight line in the $-x$ direction (Point 3 to Point 4), then rotated again by 90 degrees such that the test stand was pointed in the $-y$ direction (Point 4 to 5). Again, the arc goes backward from the actual direction of travel, but this time its length is much greater. This was caused by rotating the vehicle at a higher speed than during the first turn, meaning that this rotation defect is velocity-dependent. From Point 5, the vehicle moves in the $-y$ direction to Point 6, rotates 90 degrees counterclockwise (Point 6 to 7), and then moves in a straight line back to a point near the origin (Point 8).

The trajectory plotted in Figure 4.4 is vastly distorted. A trusting observer might surmise

[7] A ROS visualization utility. http://wiki.ros.org/rviz

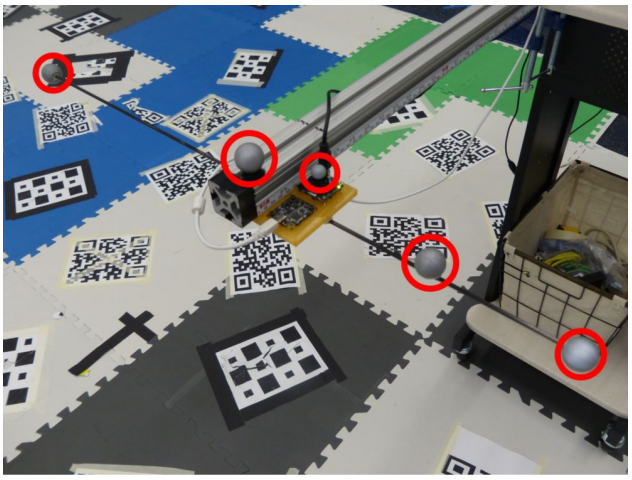

Figure 4.5: Close-up view of the sensor mount instrumented with retroreflector balls.

from the image that the vehicle moved in a pattern resembling two uppercase D's, as opposed to the square pattern traced out in reality. This rotation-translation defect in the ROS PTAM implementation is pervasive and prevents PTAM from being of any real use during rotation. For this reason, the experiments detailed in the next section were carefully performed without rotating the sensor suite (Figure 4.5).

In other preliminary experiments, anomalous behavior was discovered in PTAM's z-position output. Over long translations, the vehicle's estimated altitude could be seen rising erroneously the farther the vehicle moved from the origin. Figure 4.6 shows a trajectory traced out by moving the test stand several meters to the left and right of the origin in a straight line. In profile, the visualized trajectory is clearly curved. This "bowl-shaped" trajectory may arise from lens distortion not properly eliminated by PTAM. PTAM's calibration procedure was repeated several times to try to solve this problem, but the distortion persisted. The mvBlueFOX camera has a wide-angle lens with a 150° field of view. During the calibration procedure, camera parameters were computed that, according to the ROS PTAM documentation, were well within acceptable limits for a wide-angle lens. Nevertheless, the aberrant measurements continued.

4.2 Experimental Procedures

Two experiments were designed to characterize the UKF framework's effectiveness in various regimes of motion. The first experiment was a "long walk" in which the mobile test stand was

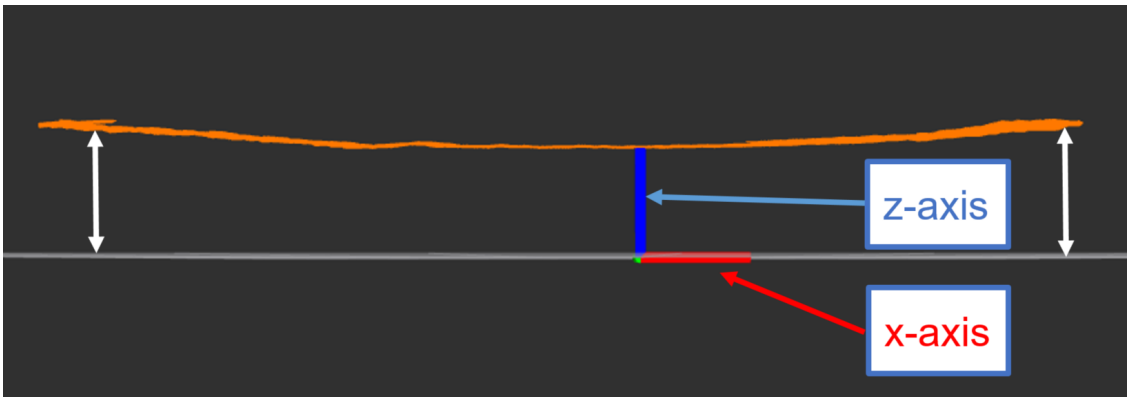

Figure 4.6: An Rviz visualization of the "bowl-shaped" world seen by PTAM during a long translational motion. This image was created by translating the mobile test stand 3–4 meters in the $+x$ direction and then translating back 4–5 meters past the origin in the $-x$ direction.

translated in a straight line for 4–6 meters along the y-axis. This test was designed to characterize the effect of lens distortion on PTAM's z-position output, as discussed previously.

In the second experiment, the mobile test stand was moved in a rectangular "box" pattern. This test was performed to determine the system's efficacy in planar motions indicative of hovering flight. Both experiments took place on a level floor to measure more accurately any z-directional distortion. Because of the rotation-translation coupling defect found in PTAM, the mobile test stand was moved carefully in each experiment such that it would undergo minimal rotation and thus accurately capture translational motion.

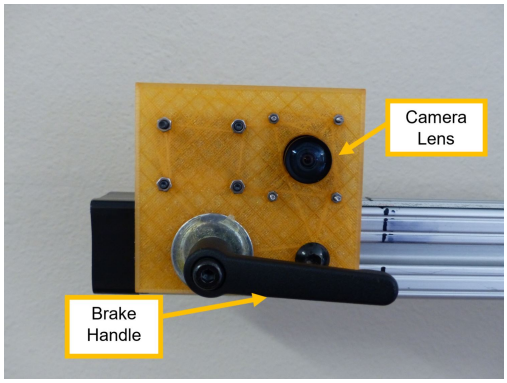

Figure 4.7: The sensor mount as seen from below. The linear brake and camera lens are visible.

Prior to each trial, PTAM was initialized according to the procedure outlined in the online documentation. To initialize the algorithm, the camera was placed parallel to the floor at a distance of one meter and then perturbed by 10% of the distance to the floor (that is, 10 cm). This perturbation tells PTAM its distance to the planar surface of the floor and thus initializes the mapping process. This initialization procedure was performed by releasing the linear bearing brake (Figure 4.7) on the sensor mount and sliding the mount along the aluminum rail, using the attached adhesive ruler to perturb the camera by exactly 10 cm. After perturbing

the camera, the brake was tightened, and the data recording was begun. Two data streams were recorded using the `rosbag`[8] recording utility: the pose messages published by `kalman_sense` and the pose messages published by Vicon.

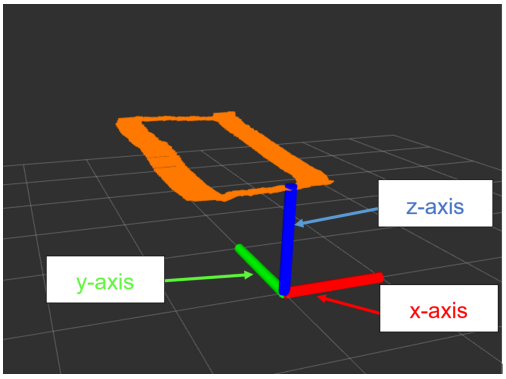

Figure 4.8: An Rviz visualization of a box pattern trajectory.

For each long walk trial, the mobile test stand was moved forward in the $+y$ direction by 4–6 meters, being careful to perturb the stand as little as possible along the x-axis. Some amount of lateral instability was unavoidable, however, due to the nature of the floor on which these trials were performed. The rubber tiles covering the floor of the testing area concealed a number of power cables for nearby computers, presenting small "bumps" over which the test stand had to pass. These bumps caused uneven left–right perturbations to the test stand, which, though minor in magnitude, proved to be noticeable in post processing (see Chapter 5). When the test stand had been pushed to the end of the long walk segment, it was then pulled backward in the $-y$ direction until it had approximately returned to the origin.

For the box pattern trials (Figure 4.8), the mobile test stand was first moved forward in the $+y$ direction by 2.5 to 3 meters. Then, without rotating, the test stand was pushed leftward in the $-x$ direction by approximately 1 meter. The test stand was then pulled backward in the $-y$ direction by 2.5 to 3 meters, returning approximately to $y = 0$. Finally, the test stand was pushed to the right ($+x$) to return to the origin.

In Section 4.3, we present information concerning the materials used to construct the mobile test stand.

[8] http://wiki.ros.org/rosbag

4.3 Materials

4.3.1 Computation and Sensing

1. One (1) MatrixVision mvBlueFOX-MLC Camera[9]
2. One (1) 1044_0 PhidgetSpatial Precision 3/3/3 High Resolution IMU[10]
3. One (1) Hewlett-Packard Spectre x360 Convertible Laptop 13-ac076nr[11]
4. Two (2) male Mini USB 2.0 to male USB Type A cables

4.3.2 Mobile Test Stand

1. One (1) Oklahoma Sound PRC200 Premium Presentation Cart[12]
2. One (1) 3D-printed Sensor Mount
3. Two (2) 4" C-Clamps
4. One (1) 1.2-meter 80/20® Inc. 1515 Rail[13]
5. One (1) 15 Series "L" Handle Linear Bearing Brake Kit[14]
6. One (1) $\frac{5}{16}$-18 × 0.687" Black FBHSCS (Screw)[15]
7. Two (2) Slide-In Economy T-Nuts[16]
8. Three (3) 1" Infrared Retroreflector Balls
9. Two (2) $\frac{1}{2}$" Infrared Retroreflector Balls
10. One (1) 0.7-meter Length of $\frac{1}{8}$"-thick Carbon Fiber Tube

[9] https://www.matrix-vision.com/USB2.0-single-board-camera-mvbluefox-mlc.html
[10] http://www.phidgets.com/products.php?product_id=1044
[11] http://store.hp.com/us/en/pdp/hp-spectre-x360---13-ac076nr
[12] http://www.oklahomasound.com/products/product-category/single/?prod=9
[13] https://8020.net/1515.html
[14] https://8020.net/6800.html
[15] https://8020.net/shop/3320.html
[16] Also available at https://8020.net/shop/3320.html

Chapter 5

Experimental Results

5.1 Long Walk Trials

Figures 5.1–5.5 plot the vehicle's x-, y-, and z-coordinates over time. These plots demonstrate that the UKF's pose estimates were able to track the vehicle's y-position accurately over the length of each long walk trial. The plots also show that the UKF pose estimates exhibit aberrant behavior causing the vehicle's z-position to rise more and more the farther the vehicle moves from the origin. The shaded regions in Figures 5.1–5.5 clearly highlight this behavior. In each plot, the vehicle's estimated altitude rises steadily to a maximum of about 1.4 meters, although in truth the vehicle's altitude remained constant in all trials. The maximum vertical error coincides with its maximal displacement from the origin at $(0,\ 0,\ 0)$.

In each trial, the UKF output showed increased "shakiness" in the form of large oscillations as the vehicle's y-position increased. As the mobile test stand moved farther and farther away from the origin, visualizations of the output showed erroneous vertical spikes as the altitude generally trended upward. The magnitude of these spikes increased with displacement from the origin, always achieving a maximal magnitude at the point of maximal displacement. Moreover, this behavior was coupled with aberrant displacements along the x-axis as well. This behavior seems to follow the supposition posited in Chapter 4 that the ROS PTAM implementation was not correctly eliminating lens distortion and, as a result, was perceiving a bowl-shaped world. If PTAM's view of the environment were curved, the large erroneous readings in x and z would reasonably be coupled to displacement along the y-axis, as the y-displacement would affect the vehicle's "height" along the inside of the "bowl." The farther the vehicle moved from the

bottom of the bowl (that is, the origin), the more z would increase and the more x would become susceptible to overestimating small perturbations. This trend can be seen, to varying degrees, in all five trials, with maximal error in the z coordinate being accompanied by maximal "shakiness" in the x coordinate—all coinciding with maximal displacement from the origin.

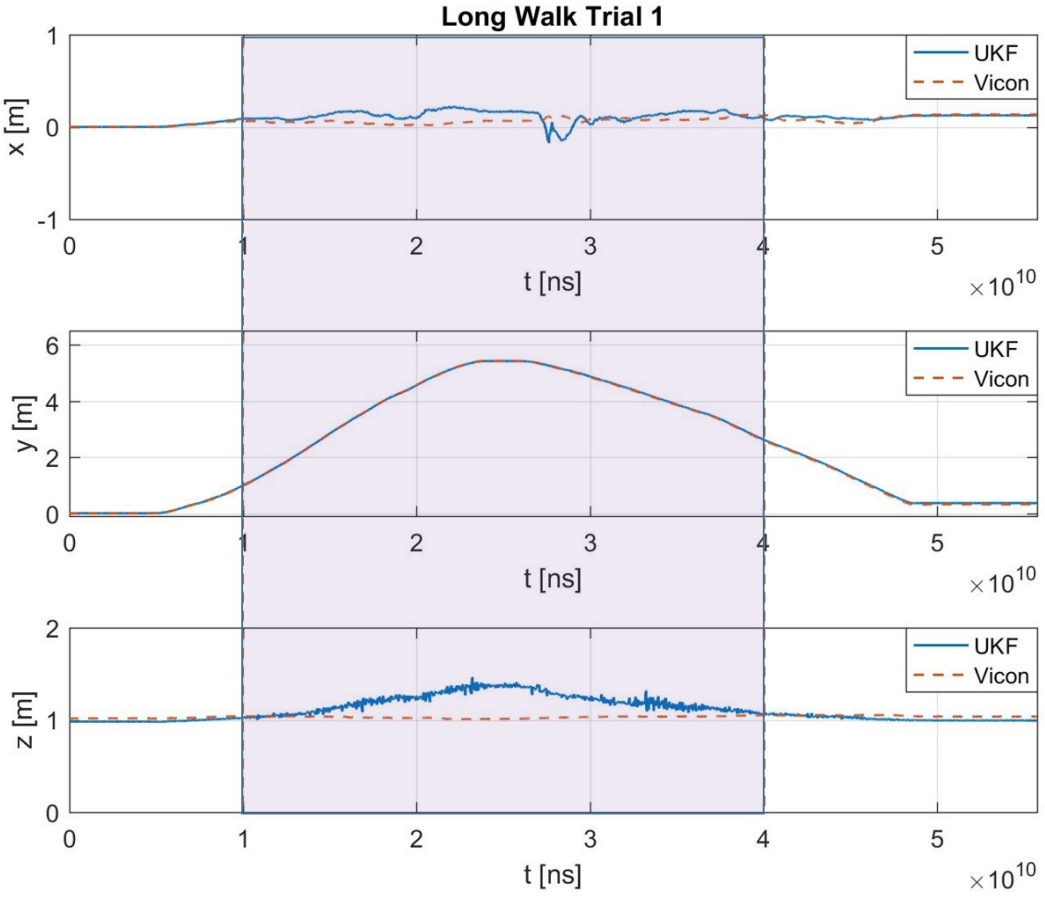

Figure 5.1: Long Walk Trial 1 coordinate plots. The shaded region highlights aberrant behavior coinciding with maximal displacement from the origin.

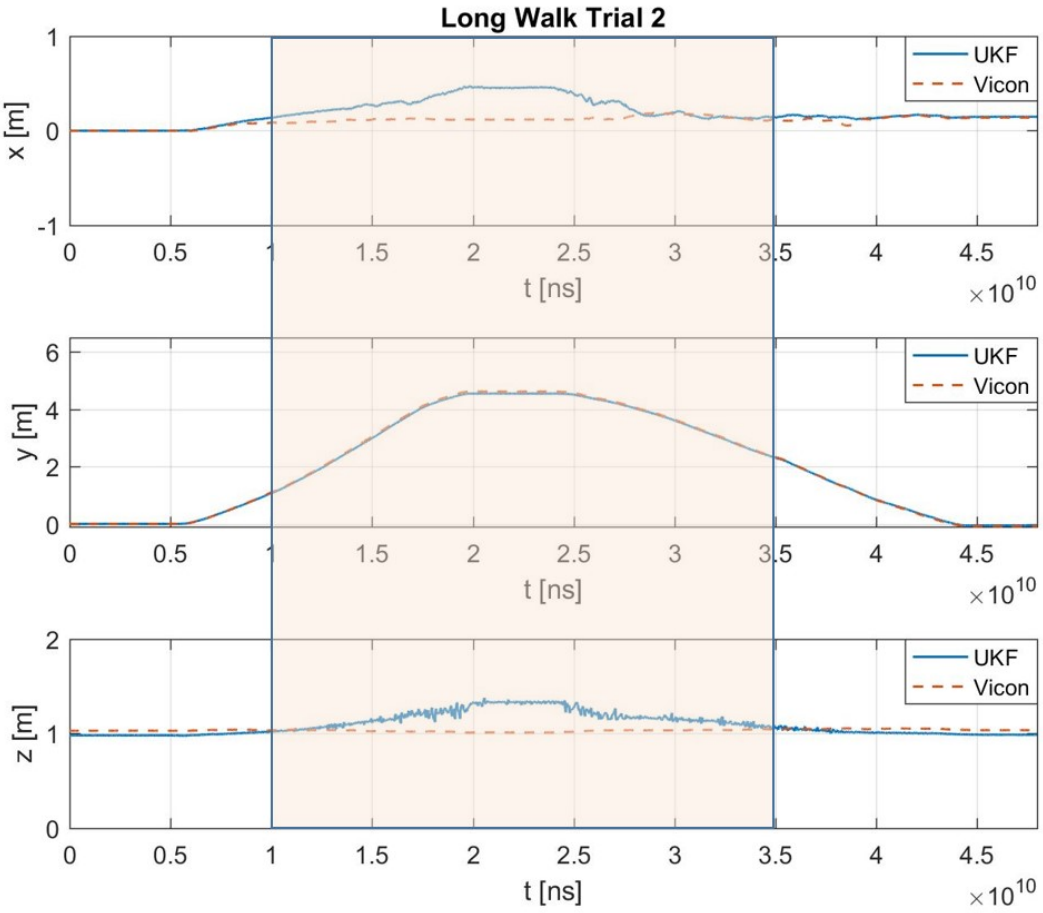

Figure 5.2: Long Walk Trial 2 coordinate plots.

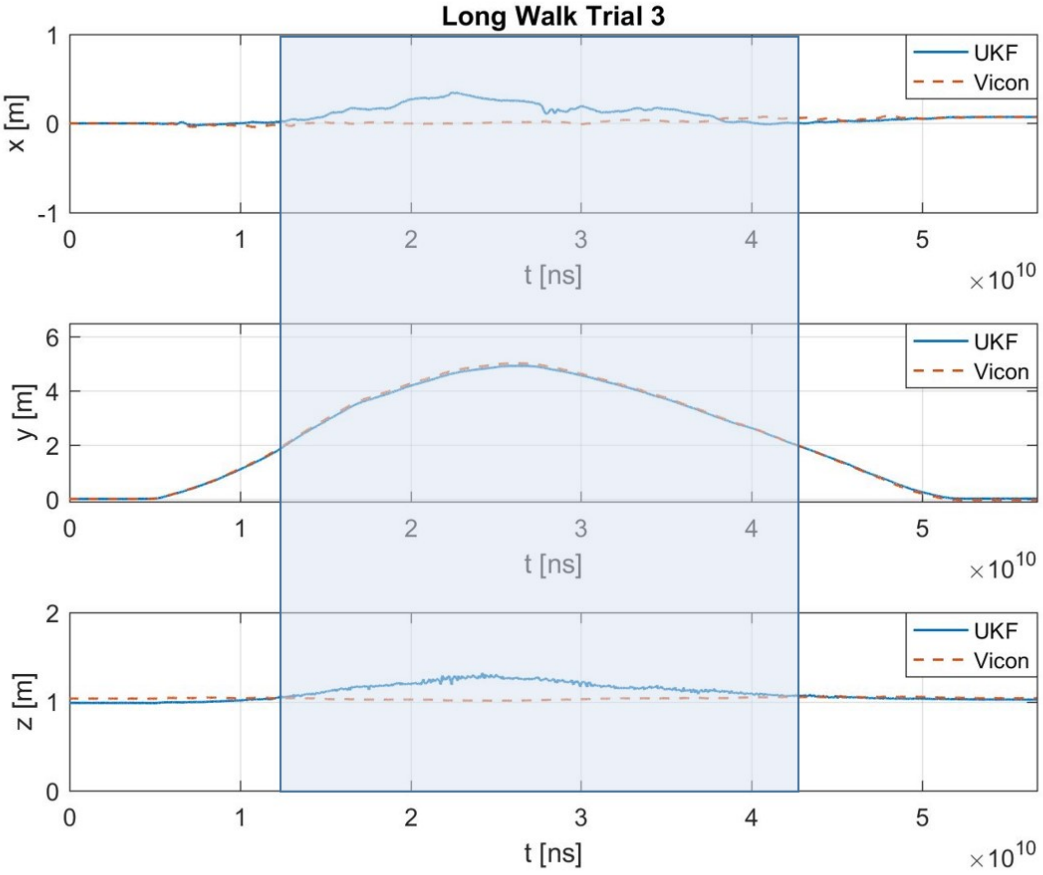

Figure 5.3: Long Walk Trial 3 coordinate plots.

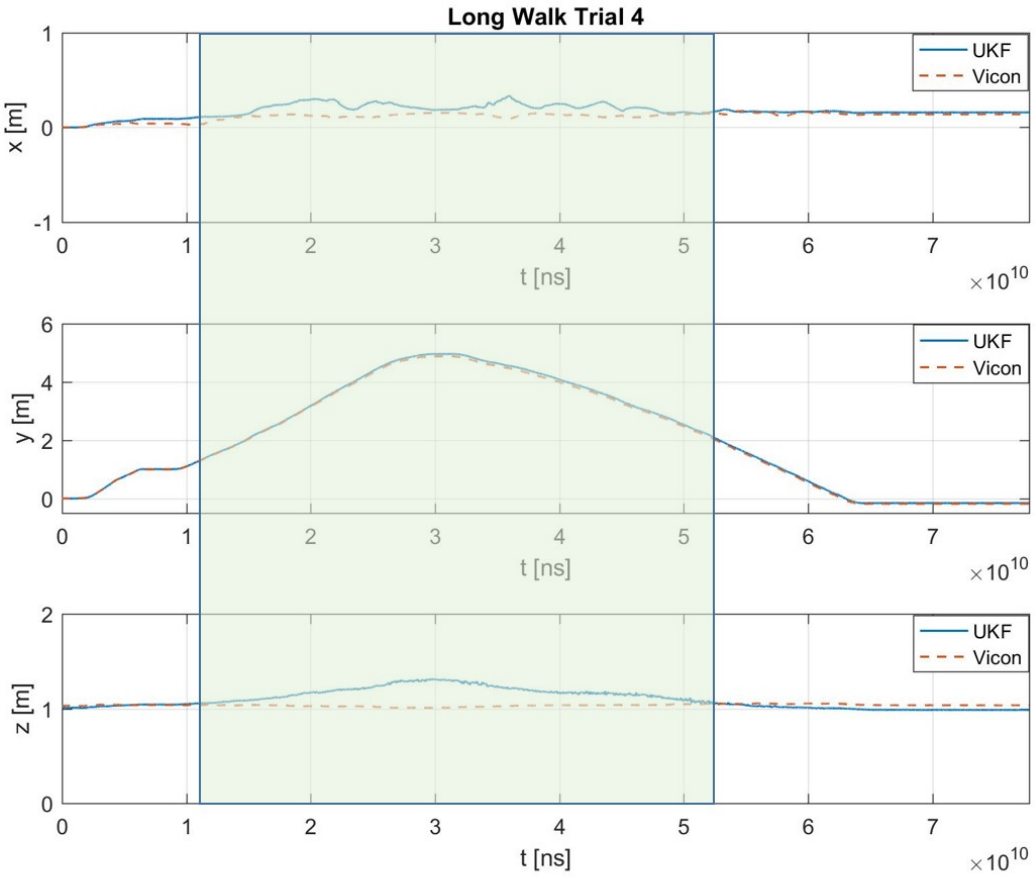

Figure 5.4: Long Walk Trial 4 coordinate plots.

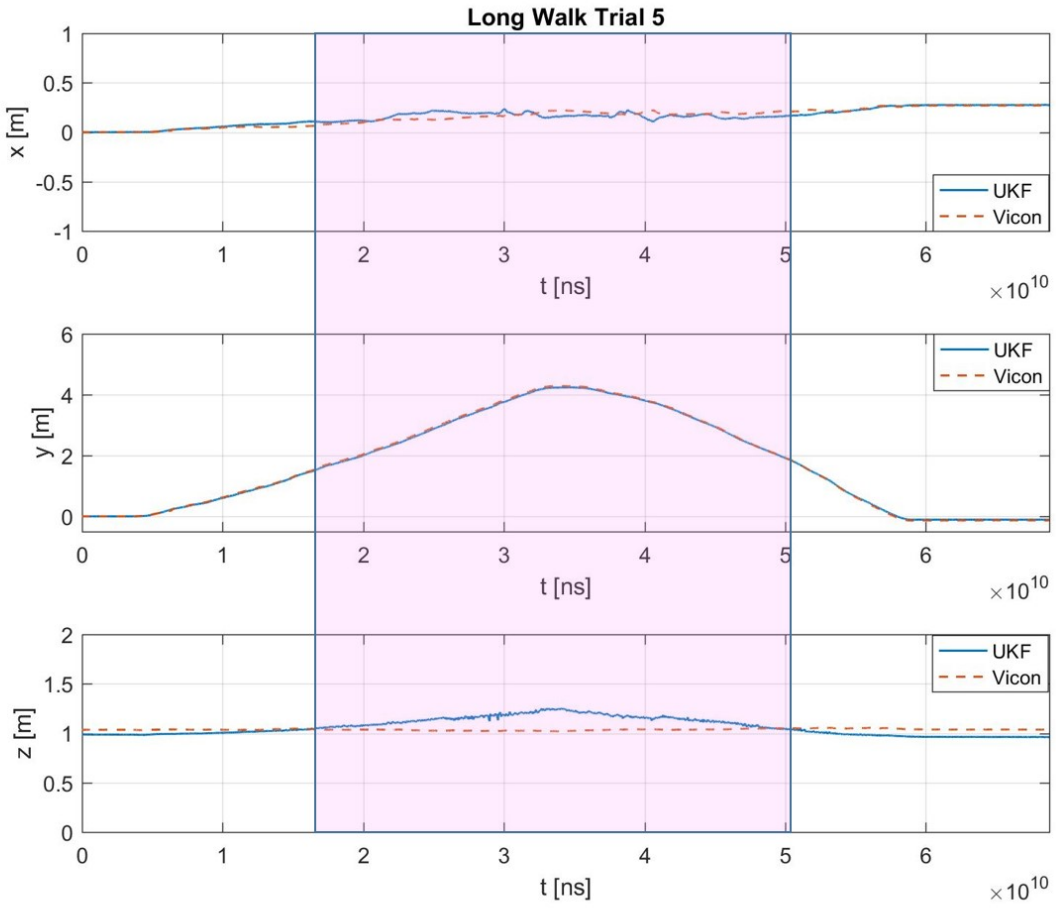

Figure 5.5: Long Walk Trial 5 coordinate plots.

5.2 Box Pattern Trials

Figures 5.7, 5.9, 5.11, 5.13, and 5.15 plot the UKF-estimated and Vicon-measured 3D trajectories of the mobile test stand during the five box pattern trials. Each of these plots exhibits the same bowl-shaped distortion seen previously in the trials of the long walk experiment. The z-position of the vehicle can be seen trending upward more and more the farther the vehicle moves from the origin. This causes the apparent slant in the estimated trajectories. This altitudinal error is most pronounced at the forward-left corner in each 3D plot, where the vehicle is farthest from the origin. The estimated trajectory also becomes noticeably shakier, in terms of vertical oscillations, in the neighborhood of the forward-left corner. These oscillations begin to appear on the first (right-side) leg of the trajectory and increase in magnitude up to the forward-left corner, at which point they begin to decrease in magnitude on the way back to the starting position.

The 2D trajectory plots (Figures 5.6, 5.8, 5.10, 5.12, and 5.14) are presented alongside the 3D plots to demonstrate the general accuracy of the UKF's estimates in planar motion. The UKF output follows the Vicon output faithfully, with some exceptions near the corners of the box pattern. This is caused partly by the bowl-shaped distortion mentioned previously, and partly by the fact that the mobile test stand moves on caster wheels. When the test stand changes direction, the caster wheels rotate underneath the test stand, causing a small lateral shift in the test stand's trajectory. This is particularly apparent in Figures 5.8 and 5.12, where a clear "wiggling" can be seen as the vehicle was pulled back from the top left corner of the pattern, and in Figure 5.14, where the estimated trajectory "sags" in the bottom left corner. These lateral disturbances, though small in magnitude (likely on the order of 3–5 cm), were picked up by PTAM and overestimated due to lens distortion.

Trial 3 (Figures 5.10 and 5.11) exhibits an anomalous x-directional offset along the first leg of the trajectory. This was most likely caused by inadvertent jostling of the test stand before initializing PTAM. The test stand was rotated at an angle to Vicon's x-axis at the outset of the trial, causing the estimated trajectory to be computed at an angle to the ground truth trajectory. Once the test stand reached the first corner, it must have been rotated slightly, causing the estimated trajectory to rejoin the ground truth measurements provided by Vicon.

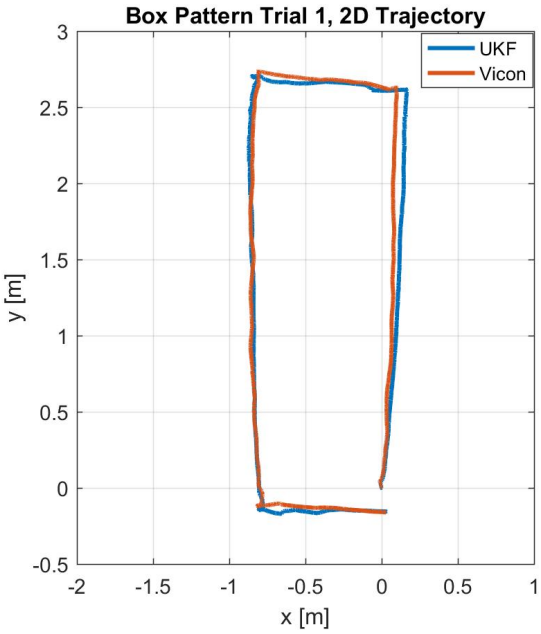

Figure 5.6: Box Pattern Trial 1 2D Trajectory.

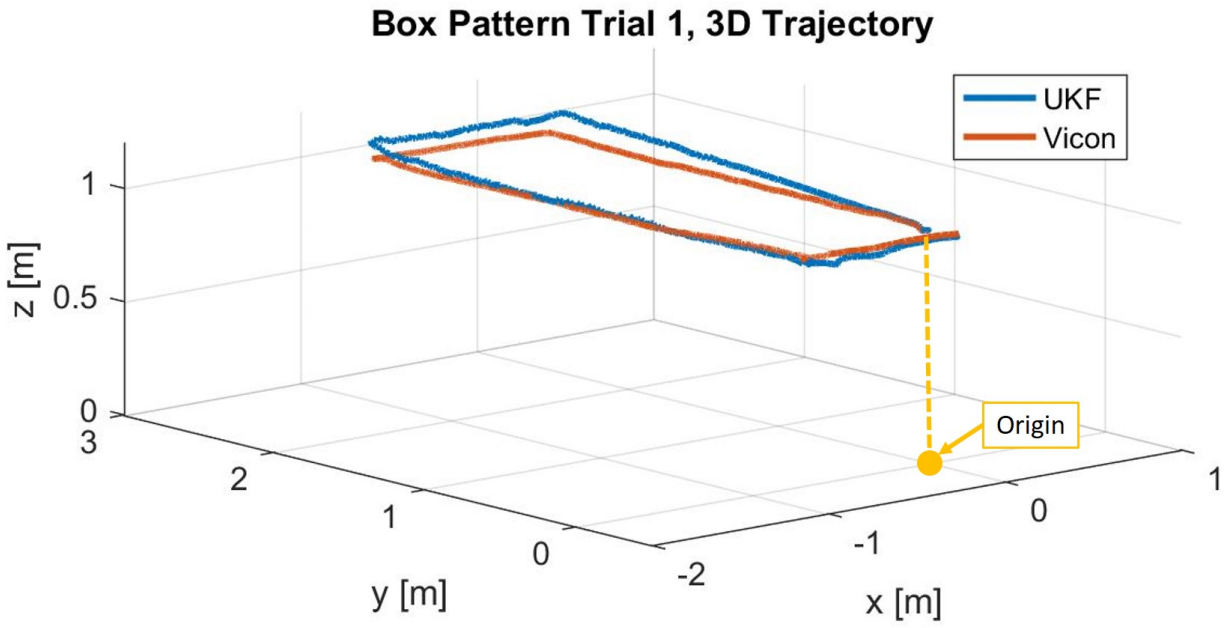

Figure 5.7: Box Pattern Trial 1 3D Trajectory.

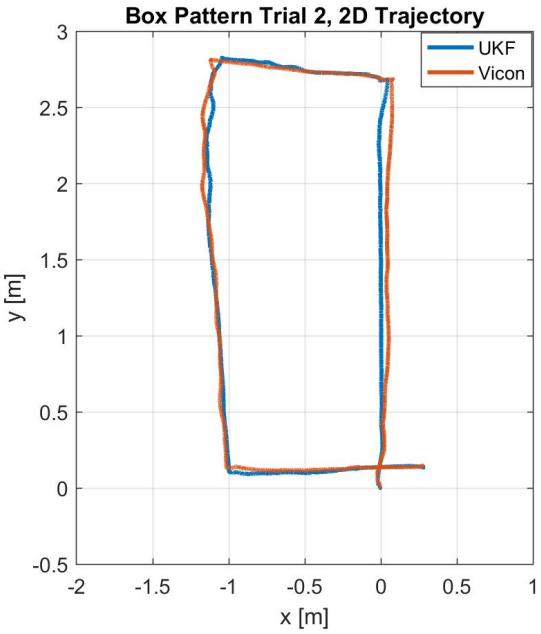

Figure 5.8: Box Pattern Trial 2 2D Trajectory.

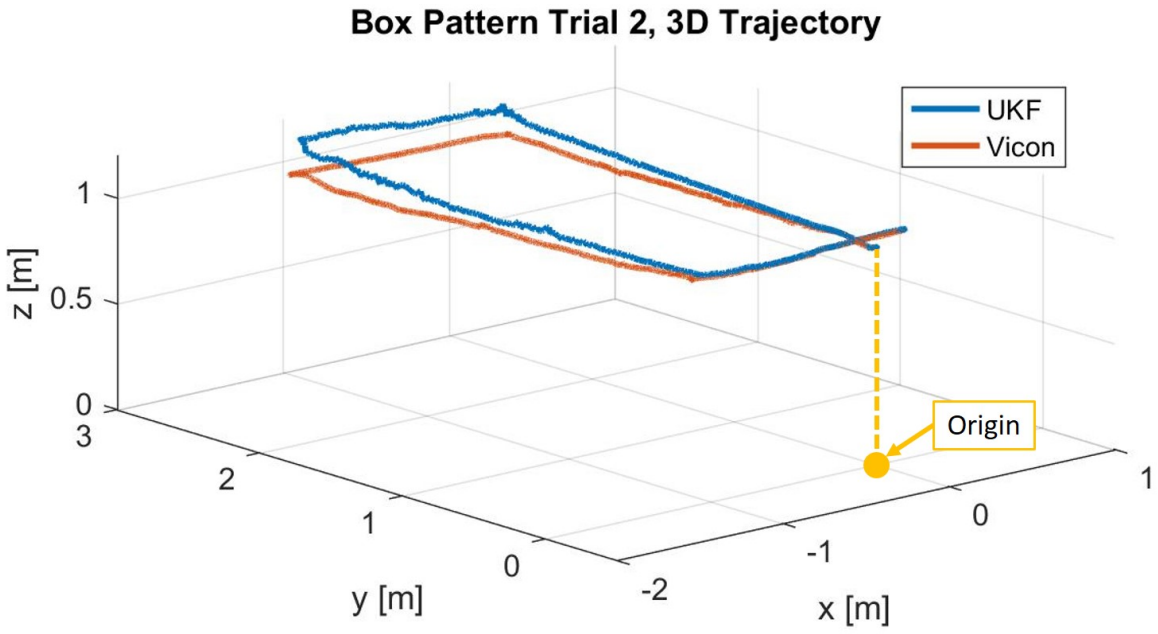

Figure 5.9: Box Pattern Trial 2 3D Trajectory.

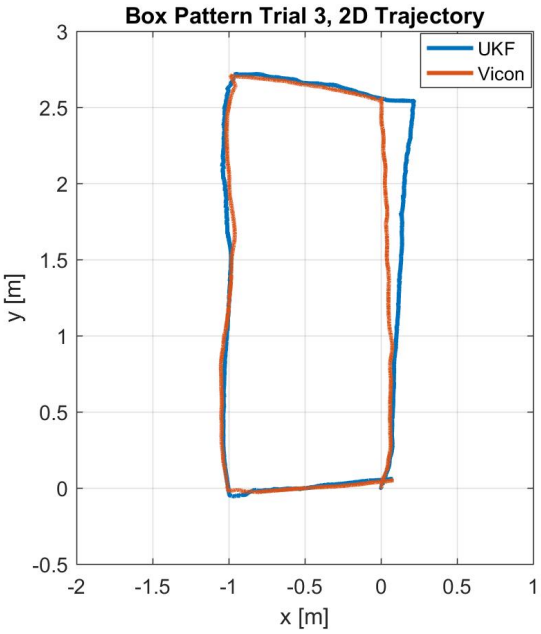

Figure 5.10: Box Pattern Trial 3 2D Trajectory.

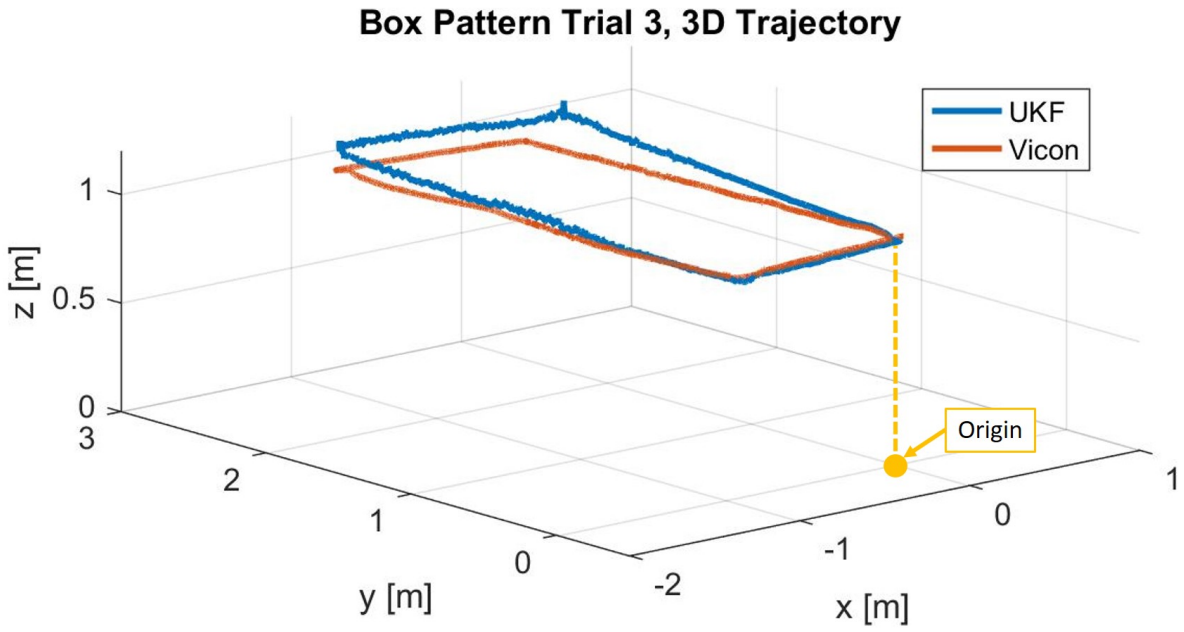

Figure 5.11: Box Pattern Trial 3 3D Trajectory.

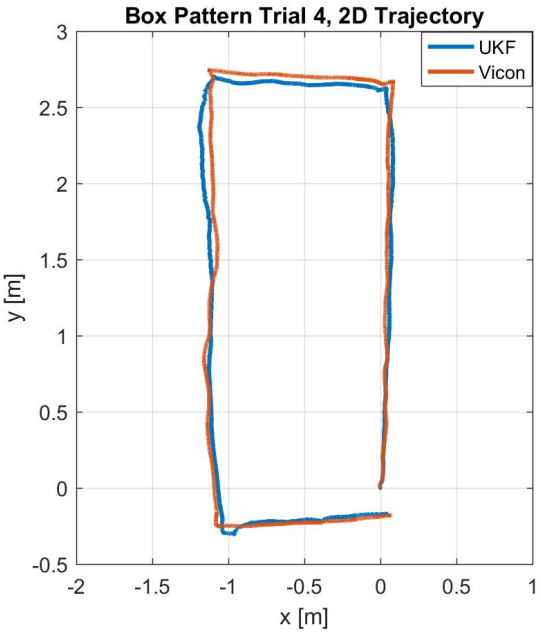

Figure 5.12: Box Pattern Trial 4 2D Trajectory.

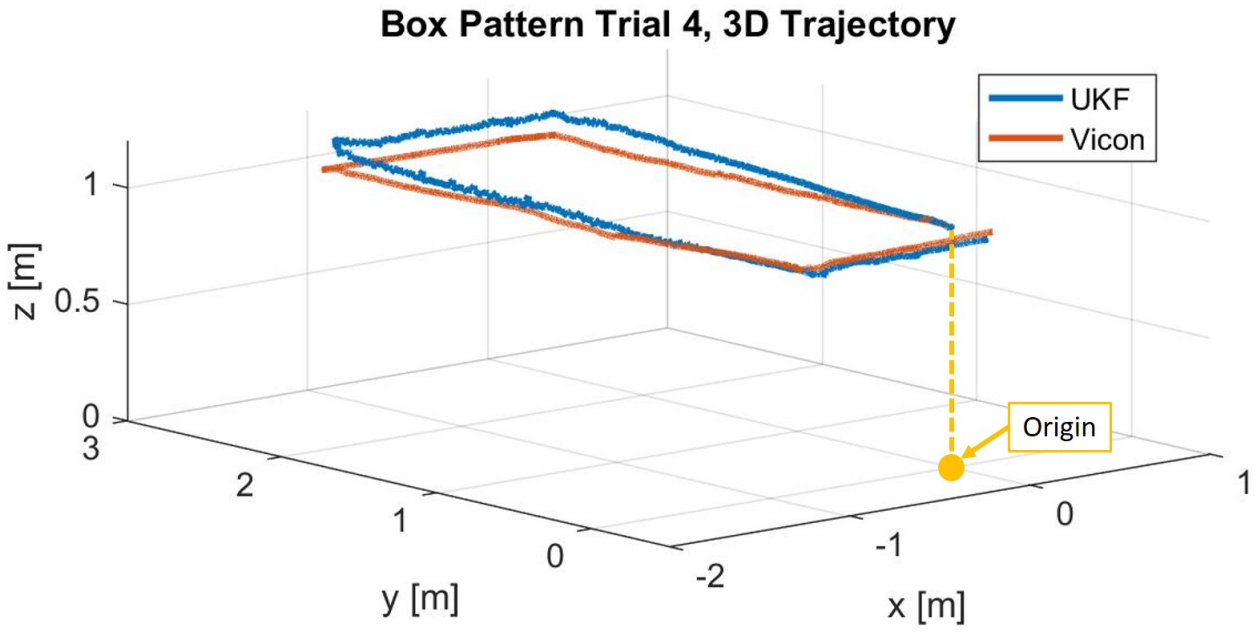

Figure 5.13: Box Pattern Trial 4 3D Trajectory.

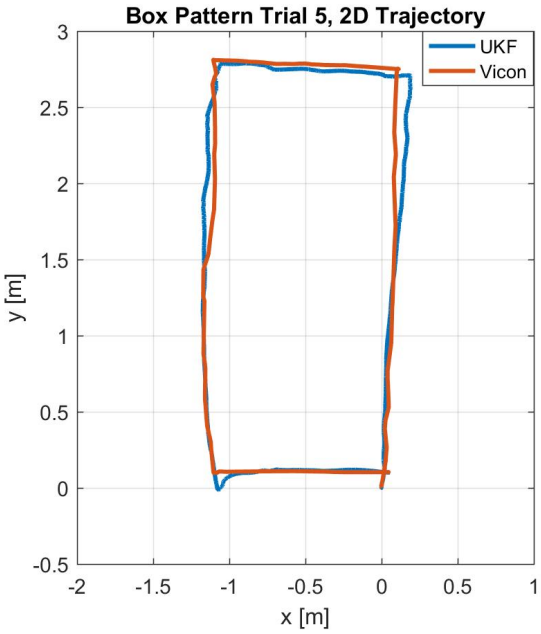

Figure 5.14: Box Pattern Trial 5 2D Trajectory.

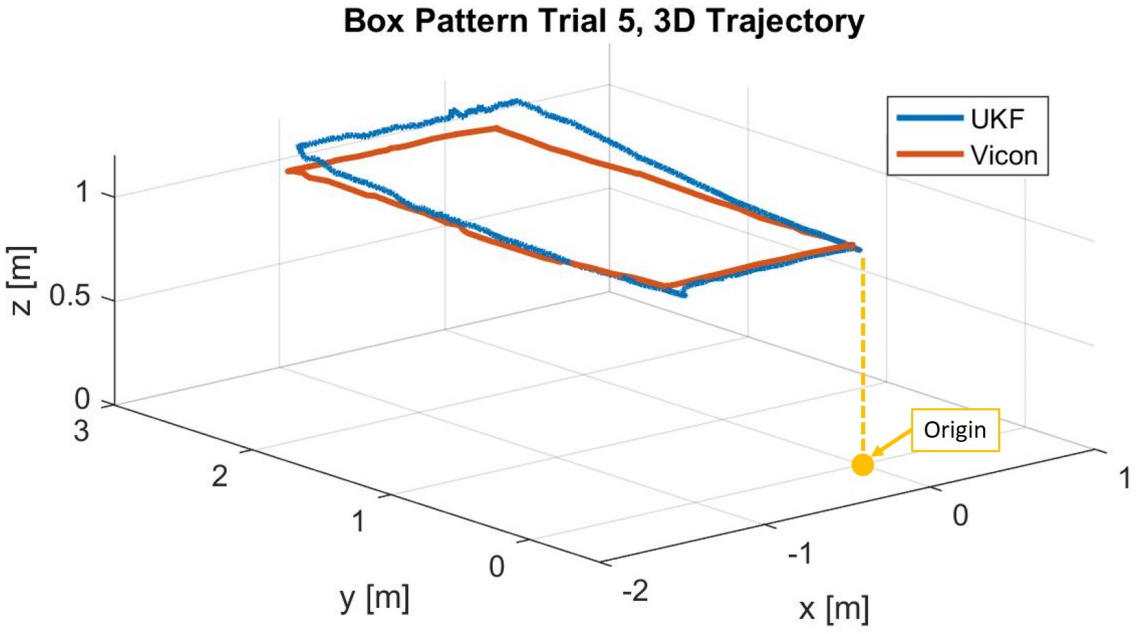

Figure 5.15: Box Pattern Trial 5 3D Trajectory.

5.3 Error Analysis

To assess the effectiveness of the UKF framework in estimating the position of the vehicle, we developed a number of statistical metrics to characterize the types and magnitudes of error in the filter's output. As shown previously (in Section 5.1), the positional error in the long walk trials is effectively unbounded. Because of the lens distortion defect, moving the vehicle in long translations produces ever-increasing altitudinal error which, in turn, seems to distort lateral sensitivity. As a result, in this section, we examine only the box pattern trials, which were comprised of shorter translations. The box pattern trials demonstrate the system's error in a scenario where PTAM's defects would have a limited influence on the filter.

Positional errors in the UKF output were computed after each box pattern trial per the following relationships:

$$\varepsilon_x = x_{\text{Vicon}} - x_{\text{UKF}} \tag{5.1}$$

$$\varepsilon_y = y_{\text{Vicon}} - y_{\text{UKF}} \tag{5.2}$$

$$\varepsilon_z = z_{\text{Vicon}} - z_{\text{UKF}} \tag{5.3}$$

These coordinate errors were computed over the length of each time history, creating an error history for each trial. The mean coordinate errors and their variances were then computed from these error histories to characterize the Gaussian noise distribution in each data set. The terms μ_x, μ_y, and μ_z encode the mean error in x, y, and z, respectively. The associated variances of these errors are represented by σ_x^2, σ_y^2, and σ_z^2. These statistics are presented in Table 5.1.

Table 5.1: Coordinate error mean and variance values by trial.

Trial	μ_x [m]	σ_x^2 [m^2]	μ_y [m]	σ_y^2 [m^2]	μ_z [m]	σ_z^2 [m^2]
1	−0.0048	9.4976e−04	0.0126	2.6861e−04	−0.0247	0.0015
2	−0.0084	0.0013	0.0079	1.9425e−04	−0.0583	0.0030
3	−0.0341	0.0043	0.0033	1.9481e−04	−0.0333	0.0028
4	−3.9788e−04	7.5902e−04	0.0199	8.0152e−04	−0.0219	0.0033
5	−0.0170	0.0019	0.0279	0.0036	−0.0372	0.0034
Mean Coord. Error	−0.0129		0.0143		−0.0351	

Table 5.1 shows that the mean coordinate errors are (1) small in magnitude, being less than 4 cm in any direction, and (2) close to zero, supporting our intuition of a zero-mean Gaussian

distribution. The variance values are also small in magnitude, being on the order of 10^{-3} m^2 or less in all cases. This indicates a tight distribution about the mean, implying that large erroneous spikes were rare during testing.

The positional errors were composed into a positional error vector $\boldsymbol{\varepsilon}$ describing the offset between ground truth and the estimated trajectory at a given point in time:

$$\boldsymbol{\varepsilon} = \{\varepsilon_x,\ \varepsilon_y,\ \varepsilon_z\}^T \tag{5.4}$$

The norm of this error vector is the total Euclidean offset $\|\boldsymbol{\varepsilon}\|$:

$$\|\boldsymbol{\varepsilon}\| = \sqrt{\varepsilon_x^2 + \varepsilon_y^2 + \varepsilon_z^2} \tag{5.5}$$

The total Euclidean offset encodes the straight-line distance between ground truth and the estimated position, providing a measure of total positional error at each point in time. The total Euclidean offset was computed over the entire error history in order to characterize the evolution of the error over time. This total error history was then analyzed for each experimental trial to determine the mean total error and maximum total error (Table 5.2).

Table 5.2: Mean total error and maximum total error by trial.

Trial	Mean $\|\boldsymbol{\varepsilon}\|$ [m]	Maximum $\|\boldsymbol{\varepsilon}\|$ [m]
1	0.0460	0.1145
2	0.0691	0.1999
3	0.0695	0.2596
4	0.0636	0.1508
5	0.0856	0.2497
Mean Total Error	0.0668	
Overall Maximum Total Error		0.2596

Table 5.2 shows that the system ran with a mean total error of 0.0668 m and a maximum total error of 0.2596 m, indicating that the system was able to estimate the vehicle's position consistently with sub-meter accuracy over the length of each trial. The maximum total error, though large, still falls within the 4.9-m radius that is realizable with today's smartphone GPS receivers [21], suggesting that the system could prove to be a reasonable choice for navigating environments where centimeter-level accuracy is not required.

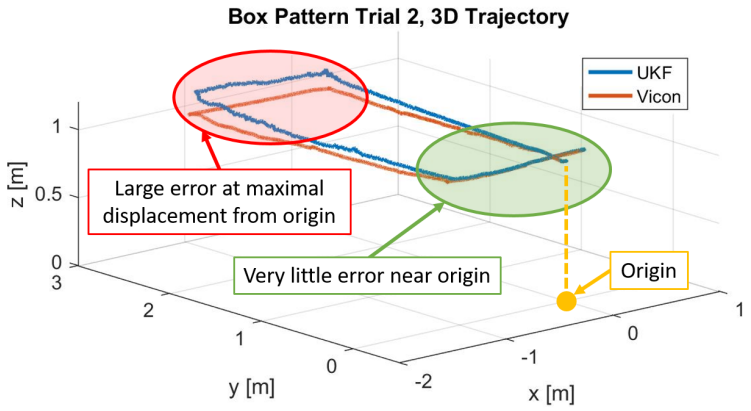

Figure 5.16: The system exhibits altitudinal error which increases as the vehicle moves farther from the origin and decreases as the vehicle returns to the origin.

In real-world applications, it is common for a SLAM algorithm to exhibit error as a function of both total displacement and mission time, with drift increasing over time as the vehicle moves. It is noteworthy, then, that the UKF's output exhibited error only as a function of total displacement, specifically *from the origin*. Figure 5.16 demonstrates how the UKF's error grew as the vehicle moved farther from the origin, but decreased as the vehicle returned to the origin, producing a consistently bowl-shaped distortion in the estimated trajectory. The UKF's output also showed no obvious drift as a function of time during periods when the vehicle was stationary. This error behavior is confirmed by the final coordinate error and final total error values presented in Table 5.3.

Table 5.3: Final coordinate error and final total error by trial.

Trial	Final ε_x [m]	Final ε_y [m]	Final ε_z [m]	Final $\|\boldsymbol{\varepsilon}\|$ [m]
1	0.0047	0.0027	−0.0129	0.0140
2	0.0105	−0.0083	0.0043	0.0141
3	−0.0054	0.0090	−0.0225	0.0248
4	−0.0236	0.0096	−0.0309	0.0400
5	0.0087	−0.0029	−0.0175	0.0198

These values were computed using only the last pose collected from the UKF and the last pose collected from Vicon. The magnitudes of these error terms demonstrate that, at the moment each trial ended, the UKF's total positional error $\|\boldsymbol{\varepsilon}\|$ had always fallen to 4 cm or less, substantially below the 20+ cm of total error achieved at maximal displacement from the origin. The cause

of this error behavior is currently unknown. As posited previously, this could be the result of lens distortion not eliminated by the ROS PTAM implementation. If so, this would indicate that the altitudinal error is primarily caused by the geometry of the camera lens, rather than by any problem intrinsic to the PTAM algorithm or the UKF algorithm. The drift—which normally would grow with time and displacement—may be small relative to this lens-based error due to the smaller translations and shorter durations of the box pattern trials. In other words, the "normal" drift may have been small enough to be masked by the larger lens-based drift, making it undetectable in these experiments. Further experiments of greater duration and translational magnitude may be able to confirm or refute this supposition.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In the course of this research, we have developed an algorithm for fusion of IMU and SLAM data for localization of a small aircraft capable of hovering flight. We have presented not only an Unscented Kalman Filter formulation for this purpose, but also two corrective measures for maintaining quaternion continuity and deriving velocity measurements from pose readings to maintain numerical stability in the filter's output. We have also presented software implementation details related to the creation of a ROS package performing this fusion algorithm and described a number of architectural traits of this package encouraging and allowing for the adaptation of this work to systems other than small rotorcraft.

This work is largely motivated by the confluence of increased computing power and increased payload capacity for small UAS. The system developed in this thesis is meant to empower the roboticist researcher with a general state estimation framework useful for modeling a wide range of systems with arbitrary levels of fidelity. The experiments detailed in Chapters 4 and 5 deployed the system in an indoor navigation scenario to characterize its behavior in a realistic environment where GPS would never be available as a backup sensing modality. In box pattern tests, the system consistently maintained a mean total positional error of less than 7 cm with overall maximum total error never exceeding 25.96 cm. By these performance metrics, the UKF framework is more accurate than today's smartphone-grade GPS receivers and on par with similar sensor fusion systems detailed in Chapter 2.

Nevertheless, the system faces several critical challenges. The PTAM implementation used

during the experiments suffers from a number of serious defects. This PTAM implementation exhibits aberrant z-position estimates over flat terrain, erroneously conflates rotations with translations, and appears not to eliminate camera lens distortion. These defects severely limit the possible use cases for the system, assuming the same PTAM implementation is used. For effective use in non-laboratory scenarios, the vehicle would have to fly in pure translation (that is, without substantial rotations). Though this may seem like an odd requirement, modern hobby-grade autopilots, such as the Pixhawk, already allow for precisely this type of flight.

Perhaps the more stringent requirement on this system is that it be used in environments with ample visual geometry. The SLAM algorithm used in experimentation was susceptible to losing tracking if the density of visible features was not sufficiently high. Thus, high-density environmental features would be a necessary condition for any SLAM/VO algorithm, meaning that the system could be deployed usefully only in areas having a vast array of high-contrast features (and sufficient lighting by which to see them).

With all of this being said, the UKF framework developed here still represents a useful jumping-off point for further work. Future developments in this area (explored in detail in the next section) will likely revolve around increasing accuracy and robustness, the two principle challenges of any navigation system. Even the much-glorified Global Positioning System is not without its systemic faults and environmental limitations. Because of its generality, this work could be applied to great effect in numerous state estimation scenarios. Beyond rotorcraft and fixed-wing airplanes, this UKF framework could be adapted for use with spacecraft, surface vessels, undersea vehicles, and automobiles. Due to increasing demand for intelligent systems and unmanned vehicles of many varieties, it is likely that research in this area will continue for decades to come. In any case, the emphasis will likely remain on higher accuracy and greater robustness. With the growing trend of miniaturization and the increasing density of human populations, the need for ever more precise localization will always be present.

6.2 Future Work

6.2.1 System Improvements

Experimentation with the `kalman_sense` ROS package has revealed a number of areas for improvement. Speaking generally, the system needs enhancements to improve accuracy and to become more robust with respect to potential sensor malfunction. Further development of this

work to augment accuracy and robustness would center on three main areas:

1. Integration of new sensors,
2. Improvements for current sensors, and
3. Refinements to the underlying algorithm.

Modifications to any or all of these areas would produce a marked improvement to the system in terms of both accuracy and robustness. For maximal effectiveness in adverse conditions, the system's sensors would need to be both individually robust and collectively redundant, so that complementary sensors could compensate for the loss of another sensor's information.

Integration of New Sensors

The accuracy and robustness of the system could be augmented by the integration of additional sensors, including but not limited to, the following:

1. One or more GPS receivers (particularly Real Time Kinematic[1] GPS),
2. A pressure altimeter,
3. A 3-axis magnetometer,
4. One or more laser rangefinders,
5. One or more forward-facing cameras (preferably depth cameras, such as RGB-D),
6. One-dimensional, multi-dimensional, or scanning LIDAR[2].

We previously mentioned work from the GRASP Lab ([8]) that demonstrated the efficacy of similar UKF frameworks in indoor-outdoor transitions and in navigating confined spaces using an expanded suite of sensors. In the case where more than one sensor is used to observe a given vehicle state variable, those multiple sensors check one another and can even perform real-time, in-air calibration. The system developed in this thesis has the advantage of being hardware-minimal in that it depends on only two sensors (the IMU and the ventral camera), but this convenience comes at the cost of robustness because either sensor presents a single point of

[1] https://en.wikipedia.org/wiki/Real_Time_Kinematic
[2] "Light Detection and Ranging," https://en.wikipedia.org/wiki/Lidar

possible failure. Moreover, the sensors have no overlap. Neither can observe any of the states observed by the other. This eliminates the possibility of checking one sensor against the other. Introducing a ventral laser rangefinder or pressure altimeter would allow for much more precise measurement of the vehicle's altitude and would allow for in-flight re-initialization of any metric SLAM/VO algorithm.

With the exception of certain sensors such as scanning LIDAR, which can cost tens of thousands of dollars, and RTK-GPS, which often falls in the same price range, these proposed new sensors would be inexpensive additions to the vehicle. Given the sophistication of current miniaturized sensors, it is not unreasonable to expect sub-meter or centimeter-level position accuracy from vehicles weighing fewer than fifteen pounds (including a substantial sensor payload) and costing less than 20,000 USD. It has been the case for a number of years now that meter-level accuracy could be taken for granted with "toy" quadcopters using only MEMS[3] IMUs and sub-$100 GPS receivers. The trend of miniaturization has made more and more sensors of better and better quality increasingly available in the weight- and cost-constrained world of aerial robotics. A variety of inexpensive off-the-shelf autopilots, such as the Pixhawk 2.1[4], already boast triple-redundant IMUs and support for multiple GPS modules. Many of the hardware integration challenges that once plagued hobbyists and researchers have been resolved by engineering developments stemming from popular demand for user-friendly drones.

The organization of the `kalman_sense` package's code base is such that the integration of new sensors would require only the addition of new callback functions and ROS subscribers, as opposed to a full redesign or major edits to the existing code. Further, a new SLAM or VO algorithm could also be used in lieu of PTAM, allowing an "apples-to-apples" comparison of different visual localization algorithms. For example, Donavanik et al. [5] have identified a new algorithm known as ORB-SLAM [22] as a promising candidate for robust SLAM, already implemented in ROS. Because of ROS's sophisticated ecosystem of interoperable packages and message formats, any pose sensor employing the same message type as PTAM could plug in to `kalman_sense` in a matter of seconds.

Much of the design thinking that went into the `kalman_sense` package was focused on this particular possibility. The aircraft of the future will be vehicles laden with a vast suite of heterogeneous sensors, the outputs of which would, at any given time, be used not only for

[3] "Microelectromechanical systems,"
https://en.wikipedia.org/wiki/Microelectromechanical_systems

[4] http://www.proficnc.com/

inner-loop and outer-loop flight control, but also for self-calibration, real-time diagnostics, and operator telemetry. Future UAVs will be akin to floating laboratories, mobile arrays of cutting-edge measurement systems allowing remote sensing over areas of several square miles.

Improvements for Current Sensors

For improved pose measurements, a more robust implementation of PTAM could be written and the same experiments could be repeated for comparison. PTAM's rotation errors and inability to eliminate camera lens distortion constitute severe limitations to its effectiveness in small UAS navigation. Moreover, if a rewritten formulation were able to interface with a downward-facing rangefinder, the initialization process could be eliminated altogether. The drawback is that making these changes would require a major software rewrite, a project that could be feasibly undertaken only by an experienced team of computer vision researchers and software developers. The mere cost in terms of both time and labor would likely make this an unappealing proposition, especially if other ready-made SLAM/VO algorithms were available to serve as PTAM replacements.

Processing power will also be central to the system's effectiveness during deployment on a real-world mission. This research is in many ways the product of recent advances in computing power. The advent of miniature desktop computers, such as the Intel NUC[5], as well as power-efficient Graphics Processing Units[6] (GPUs) has made a new frontier of estimation and localization algorithms easily accessible for the first time in the UAV world. A simple and generally inexpensive approach to augmenting the `kalman_sense` system would thus be the inclusion of top-of-the-line lightweight computing hardware. With modern onboard computers, even the need for cross-platform code compilation (so-called "transpiling") is eliminated because these computers employ the same x86 architecture used by most software developers.

Refinements to the Underlying Algorithm

The UKF formulation itself presents another set of opportunities for improvement. In particular, better process and measurement noise modeling would allow for increased accuracy. As described previously, the noise covariance matrices used during the experiments were only rough, static models meant to overestimate the Gaussian noise in the system. In practice, it would

[5] "Next Unit of Computing," https://en.wikipedia.org/wiki/Next_Unit_of_Computing
[6] https://en.wikipedia.org/wiki/Graphics_processing_unit

be better to replace these with time-varying matrices with terms more closely matching the intrinsic parameters of the system. Due to the high degree of coupling between state variables and the fundamental nonlinearity of the process model, this type of work was considered to be outside the scope of my thesis. In the future, however, a better **Q** matrix could be determined, perhaps by application of autocovariance and system identification methods. Moreover, any new sensor integrated into the system would require its own **R** matrix detailing its particular idiosyncrasies. With this done, the focus of attention could be on tuning the **Q**/**R** ratio of each sensor, thus making the system even more faithful to known sensor parameters. With more sensors, self-calibration techniques such as those employed in [10] could also be implemented to avoid the process of reinitializing one or more sensors in mid-air.

6.2.2 Future Experiments

Using a SLAM/VO algorithm that addresses pure rotation and camera lens distortion would be the easiest and most cost-effective way to improve the system. Testing different visual localization algorithms would thus be an appealing avenue of inquiry. It would behoove the researcher to characterize the system's behavior under the same conditions that caused the PTAM-based implementation to fail. In particular, experiments that include pure rotations and longer translations, as well as simulated flight movement over uneven but otherwise virtually unchanging terrain would mimic real-world flight missions more closely and provide a more holistic understanding of the system's effectiveness. Additionally, mounting the sensor suite on the underside of a UAV would allow for the collection of real-world flight data, which could shed light on the system's speed limits and behavior in the presence of in-flight vibrations.

After integrating new sensors, new experiments could be devised to characterize the system's effectiveness during sensor blackouts—for example, during an indoor-outdoor transition after integrating a GPS receiver. Experiments such as this could then inform the design of heuristic models for recognizing sensor degradation and perhaps even consensus-based estimation strategies.

Bibliography

[1] S. J. Julier and J. K. Uhlmann, "A New Extension of the Kalman Filter to Nonlinear Systems," in *Proc. SPIE*, vol. 3068, pp. 182–193, 1997.

[2] S. J. Julier, "A Skewed Approach to Filtering," in *Proc. SPIE*, vol. 3373, pp. 271–282, 1998.

[3] S. J. Julier, "The Scaled Unscented Transformation," in *Proceedings of the 2002 American Control Conference (IEEE Cat. No.CH37301)*, vol. 6, pp. 4555–4559, May 2002.

[4] S. J. Julier and J. K. Uhlmann, "Unscented Filtering and Nonlinear Estimation," in *Invited Paper Proceedings of the IEEE*, vol. 92, pp. 401–422, March 2004.

[5] D. Donavanik, A. Hardt-Stremayr, G. Gremillion, S. Weiss, and W. Nothwang, "Multi-Sensor Fusion Techniques for State Estimation of Micro Air Vehicles," 2016.

[6] G. Klein and D. Murray, "Parallel Tracking and Mapping for Small AR Workspaces," in *Proc. Sixth IEEE and ACM International Symposium on Mixed and Augmented Reality (ISMAR'07)*, (Nara, Japan), November 2007.

[7] S. Weiss, D. Scaramuzza, and R. Siegwart, "Monocular-SLAM-Based Navigation for Autonomous Micro Helicopters in GPS-Denied Environments," *Journal of Field Robotics*, vol. 28, pp. 854–874, Nov. 2011.

[8] S. Shen, N. Michael, and V. Kumar, "Autonomous Multi-Floor Navigation with a Computationally Constrained MAV," in *IEEE International Conference on Robotics and Automation*, pp. 20–25, 2011.

[9] S. Weiss and R. Siegwart, "Real-Time Metric State Estimation for Modular Vision-Inertial Systems," in *2011 IEEE International Conference on Robotics and Automation*, pp. 4531–4537, May 2011.

[10] S. Weiss, M. W. Achtelik, M. Chli, and R. Siegwart, "Versatile Distributed Pose Estimation and Sensor Self-Calibration for an Autonomous MAV," in *2012 IEEE International Conference on Robotics and Automation*, pp. 31–38, May 2012.

[11] S. Weiss, M. W. Achtelik, S. Lynen, M. Chli, and R. Siegwart, "Real-Time Onboard Visual-Inertial State Estimation and Self-Calibration of MAVs in Unknown Environments," in *2012 IEEE International Conference on Robotics and Automation*, pp. 957–964, May 2012.

[12] G. P. Huang, A. I. Mourikis, and S. I. Roumeliotis, "A Quadratic-Complexity Observability-Constrained Unscented Kalman Filter for SLAM," *IEEE Transactions on Robotics*, vol. 29, pp. 1226–1243, Oct 2013.

[13] S. Lynen, M. W. Achtelik, S. Weiss, M. Chli, and R. Siegwart, "A Robust and Modular Multi-Sensor Fusion Approach Applied to MAV Navigation," in *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 3923–3929, Nov 2013.

[14] J. G. Rogers, J. R. Fink, and E. A. Stump, "Mapping with a Ground Robot in GPS Denied and Degraded Environments," in *2014 American Control Conference*, pp. 1880–1885, June 2014.

[15] M. Faessler, F. Fontana, C. Forster, E. Mueggler, M. Pizzoli, and D. Scaramuzza, "Autonomous, Vision-Based Flight and Live Dense 3D Mapping with a Quadrotor Micro Aerial Vehicle," *Journal of Field Robotics*, vol. 33, no. 4, pp. 431–450, 2016.

[16] Cao, Yi, "Learning the Unscented Kalman Filter." https://www.mathworks.com/matlabcentral/fileexchange/18217-learning-the-unscented-kalman-filter, 2010. Accessed: 2016-10-10.

[17] E. A. Wan and R. V. D. Merwe, "The Unscented Kalman Filter for Nonlinear Estimation," in *Proceedings of the IEEE 2000 Adaptive Systems for Signal Processing, Communications, and Control Symposium (Cat. No.00EX373)*, pp. 153–158, 2000.

[18] B. L. Stevens, F. L. Lewis, and E. N. Johnson, *Aircraft Control and Simulation: Dynamics, Controls Design, and Autonomous Systems*, pp. 1–62. Wiley, 2015.

[19] K. Shoemake, "Animating Rotation with Quaternion Curves," *SIGGRAPH Comput. Graph.*, vol. 19, pp. 245–254, July 1985.

[20] R. G. Brown and P. Y. C. Hwang, *Introduction to Random Signals and Applied Kalman Filtering*. Wiley, 2012.

[21] "GPS Accuracy." http://www.gps.gov/systems/gps/performance/accuracy/. Accessed: 2017-04-23.

[22] R. Mur-Artal, J. M. M. Montiel, and J. D. Tardós, "ORB-SLAM: A Versatile and Accurate Monocular SLAM System," *IEEE Transactions on Robotics*, vol. 31, pp. 1147–1163, Oct 2015.

Appendix A

node.cpp Listing

```
#include "QuadUkf.h"

int main(int argc, char **argv)
{
  ros::init(argc, argv, "kalman_sense");
  ros::NodeHandle nh;

  ros::Publisher poseStampedPub = nh.advertise<geometry_msgs::PoseStamped>(
      "pose", 1000);
  ros::Publisher poseWithCovStampedPub = nh.advertise<
      geometry_msgs::PoseWithCovarianceStamped>("poseWithCov", 1000);
  ros::Publisher poseArrayPub = nh.advertise<geometry_msgs::PoseArray>(
      "poseHistory", 1);
  QuadUkf ukf = QuadUkf(poseStampedPub, poseWithCovStampedPub, poseArrayPub);

  ros::Subscriber imu_sub = nh.subscribe("/imu/data_raw", 1,
                                         &QuadUkf::imuCallback, &ukf);
  ros::Subscriber pose_sub = nh.subscribe("/vslam/pose", 1,
                                          &QuadUkf::poseCallback, &ukf);
  ros::spin();
  return 0;
}
```

Appendix B

UnscentedKf.h Listing

```
#ifndef UNSCENTEDKF_H_
#define UNSCENTEDKF_H_

#include <Eigen/Dense>

class UnscentedKf
{
public:
  UnscentedKf();
  virtual ~UnscentedKf() = 0;

  struct Belief
  {
    Eigen::VectorXd state;
    Eigen::MatrixXd covariance;
  };

  int numStates;
  int numSensors;
  void setWeightsAndCoeffs();

  UnscentedKf::Belief predictState(Eigen::VectorXd x, Eigen::MatrixXd P,
                                   Eigen::MatrixXd Q, double dt);
  UnscentedKf::Belief correctState(Eigen::VectorXd x, Eigen::MatrixXd P,
                                   Eigen::VectorXd z, Eigen::MatrixXd R);

private:
  Eigen::VectorXd meanWeights, covarianceWeights;
```

```
// Tunable parameters
const double ALPHA = 0.75;
const double BETA = 2;
const double KAPPA = 0;

// These values are updated by setWeights()
double lambda = 0;
double sigmaPointScalingCoeff = 0;

virtual Eigen::VectorXd processFunc(Eigen::VectorXd x, double dt) = 0;
virtual Eigen::VectorXd observationFunc(Eigen::VectorXd z) = 0;

struct Transform
{
  Eigen::VectorXd vector;
  Eigen::MatrixXd sigmaPoints;
  Eigen::MatrixXd covariance;
  Eigen::MatrixXd deviations;
};

struct SigmaPointSet
{
  Eigen::VectorXd vector;
  Eigen::MatrixXd sigmaPoints;
};

Transform unscentedStateTransform(const Eigen::MatrixXd sigmaPts,
                                  const Eigen::VectorXd meanWts,
                                  const Eigen::VectorXd covWts,
                                  const Eigen::MatrixXd measNoiseCov,
                                  const double dt);
SigmaPointSet sampleStateSpace(const Eigen::MatrixXd sigmaPts,
                               const Eigen::VectorXd meanWts,
                               const double dt);

Transform unscentedSensorTransform(const Eigen::MatrixXd sigmaPts,
                                   const Eigen::VectorXd meanWts,
                                   const Eigen::VectorXd covWts,
                                   const Eigen::MatrixXd noiseCov);
SigmaPointSet sampleSensorSpace(const Eigen::MatrixXd sigmaPts,
                                const Eigen::VectorXd meanWts);
```

```
  Eigen::MatrixXd computeCovariance(const Eigen::MatrixXd devs,
                                    const Eigen::VectorXd covWts,
                                    const Eigen::MatrixXd noiseCov) const;
  Eigen::MatrixXd computeDeviations(
      const UnscentedKf::SigmaPointSet sigmaPts) const;
  Eigen::MatrixXd computeSigmaPoints(const Eigen::VectorXd x,
                                     const Eigen::MatrixXd P,
                                     const double scalingCoeff) const;
  Eigen::MatrixXd fillMatrixWithVector(const Eigen::VectorXd vec,
                                       const int numCols) const;
};

#endif  // UNSCENTEDKF_H_
```

Appendix C

UnscentedKf.cpp Listing

```
#include "UnscentedKf.h"

UnscentedKf::UnscentedKf() :
    numStates(1), numSensors(1)
{
  this->setWeightsAndCoeffs();
}

UnscentedKf::~UnscentedKf()
{
}

UnscentedKf::Belief UnscentedKf::predictState(Eigen::VectorXd x,
                                              Eigen::MatrixXd P,
                                              Eigen::MatrixXd Q, double dt)
{
  // Compute sigma points around current estimated state
  Eigen::MatrixXd sigmaPts(numStates, 2 * numStates + 1);
  sigmaPts = computeSigmaPoints(x, P, sigmaPointScalingCoeff);

  // Perform unscented transform on current estimated state to predict next
  // state and covariance
  UnscentedKf::Transform tf = unscentedStateTransform(sigmaPts, meanWeights,
                                                      covarianceWeights, Q, dt);

  // Return a new belief
  UnscentedKf::Belief bel {tf.vector, tf.covariance};
  return bel;
}
```

```
UnscentedKf::Belief UnscentedKf::correctState(Eigen::VectorXd x,
                                              Eigen::MatrixXd P,
                                              Eigen::VectorXd z,
                                              Eigen::MatrixXd R)
{
  Eigen::MatrixXd sigmaPts(numStates, 2 * numStates + 1);
  sigmaPts = computeSigmaPoints(x, P, sigmaPointScalingCoeff);

  UnscentedKf::Transform sensorTf = unscentedSensorTransform(sigmaPts,
                                                             meanWeights,
                                                             covarianceWeights,
                                                             R);

  Eigen::VectorXd zPred = sensorTf.vector;     // Predicted measurement vector
  Eigen::MatrixXd P_zz = sensorTf.covariance;  // Sensor-to-sensor covariance

  // Compute state-to-sensor cross-covariance
  UnscentedKf::SigmaPointSet predPointSet {x, sigmaPts};
  Eigen::MatrixXd predDeviations = computeDeviations(predPointSet);

  Eigen::MatrixXd P_xz = Eigen::MatrixXd::Zero(numStates, numSensors);
  P_xz = predDeviations * covarianceWeights.asDiagonal()
      * sensorTf.deviations.transpose();

  // Compute Kalman gain
  Eigen::MatrixXd K = Eigen::MatrixXd::Zero(numStates, numSensors);
  K = P_xz * P_zz.inverse();

  // Update state vector
  Eigen::VectorXd xCorr = Eigen::VectorXd::Zero(numStates);
  xCorr = x + K * (z - zPred);

  // Update state covariance
  Eigen::MatrixXd PCorr = Eigen::MatrixXd::Zero(numStates, numStates);
  PCorr = P - K * P_xz.transpose();

  UnscentedKf::Belief bel {xCorr, PCorr};
  return bel;
}

UnscentedKf::Transform UnscentedKf::unscentedStateTransform(
```

```
    const Eigen::MatrixXd sigmaPts, const Eigen::VectorXd meanWts,
    const Eigen::VectorXd covWts, const Eigen::MatrixXd noiseCov,
    const double dt)
{
  int n = sigmaPts.rows();
  int L = sigmaPts.cols();
  Eigen::VectorXd vec = Eigen::VectorXd::Zero(n);
  Eigen::MatrixXd sigmas = Eigen::MatrixXd::Zero(n, L);
  Eigen::MatrixXd devs = Eigen::MatrixXd::Zero(n, L);
  Eigen::MatrixXd cov = Eigen::MatrixXd::Zero(n, n);

  UnscentedKf::SigmaPointSet sample = sampleStateSpace(sigmaPts, meanWts, dt);
  vec = sample.vector;
  sigmas = sample.sigmaPoints;
  devs = computeDeviations(sample);
  cov = computeCovariance(devs, covWts, noiseCov);

  UnscentedKf::Transform out {vec, sigmas, cov, devs};
  return out;
}

UnscentedKf::Transform UnscentedKf::unscentedSensorTransform(
    const Eigen::MatrixXd sigmaPts, const Eigen::VectorXd meanWts,
    const Eigen::VectorXd covWts, const Eigen::MatrixXd noiseCov)
{
  int L = sigmaPts.cols();
  Eigen::VectorXd vec = Eigen::VectorXd::Zero(numSensors);
  Eigen::MatrixXd sigmas = Eigen::MatrixXd::Zero(numSensors, L);
  Eigen::MatrixXd cov = Eigen::MatrixXd::Zero(numSensors, numSensors);
  Eigen::MatrixXd devs = Eigen::MatrixXd::Zero(numSensors, L);

  UnscentedKf::SigmaPointSet sample = sampleSensorSpace(sigmaPts, meanWts);
  vec = sample.vector;
  sigmas = sample.sigmaPoints;
  devs = computeDeviations(sample);
  cov = computeCovariance(devs, covWts, noiseCov);

  UnscentedKf::Transform out {vec, sigmas, cov, devs};
  return out;
}

Eigen::MatrixXd UnscentedKf::computeSigmaPoints(const Eigen::VectorXd x,
```

```
                                                  const Eigen::MatrixXd P,
                                                const double scalingCoeff) const
{
  // Compute lower Cholesky factor "A" of the given covariance matrix P
  Eigen::LLT<Eigen::MatrixXd> lltOfCovMat(P);
  Eigen::MatrixXd L = lltOfCovMat.matrixL();
  Eigen::MatrixXd A = scalingCoeff * L;

  // Create a matrix "Y", which is then filled columnwise with the given
  // column vector x
  Eigen::MatrixXd Y = Eigen::MatrixXd::Zero(numStates, numStates);
  Y = fillMatrixWithVector(x, numStates);

  // Create and populate sigma point matrix
  Eigen::MatrixXd sigmaPts(numStates, 2 * numStates + 1);
  sigmaPts << x, Y + A, Y - A;
  return sigmaPts;
}

Eigen::MatrixXd UnscentedKf::computeDeviations(
    const UnscentedKf::SigmaPointSet sample) const
{
  Eigen::VectorXd vec = sample.vector;
  Eigen::MatrixXd sigmaPts = sample.sigmaPoints;
  int numCols = sigmaPts.cols();
  Eigen::MatrixXd vecMat = fillMatrixWithVector(vec, numCols);

  return sigmaPts - vecMat;
}

Eigen::MatrixXd UnscentedKf::computeCovariance(
    const Eigen::MatrixXd deviations, const Eigen::VectorXd covWts,
    const Eigen::MatrixXd noiseCov) const
{
  return deviations * covWts.asDiagonal() * deviations.transpose() + noiseCov;
}

UnscentedKf::SigmaPointSet UnscentedKf::sampleStateSpace(
    const Eigen::MatrixXd sigmaPts, const Eigen::VectorXd meanWts,
    const double dt)
{
  int numRows = sigmaPts.rows();
```

```
  int numCols = sigmaPts.cols();
  Eigen::VectorXd vec = Eigen::VectorXd::Zero(numRows);
  Eigen::MatrixXd sigmas = Eigen::MatrixXd::Zero(numRows, numCols);
  for (int i = 0; i < numCols; ++i)
  {
    sigmas.col(i) = processFunc(sigmaPts.col(i), dt);
    vec += meanWts(i) * sigmas.col(i);
  }

  UnscentedKf::SigmaPointSet out {vec, sigmas};
  return out;
}

UnscentedKf::SigmaPointSet UnscentedKf::sampleSensorSpace(
    const Eigen::MatrixXd sigmaPts, const Eigen::VectorXd meanWts)
{
  int numCols = sigmaPts.cols();
  Eigen::MatrixXd sigmas = Eigen::MatrixXd::Zero(numSensors, numCols);
  Eigen::VectorXd vec = Eigen::VectorXd::Zero(numSensors);
  for (int i = 0; i < numCols; ++i)
  {
    sigmas.col(i) = observationFunc(sigmaPts.col(i));
    vec += meanWts(i) * sigmas.col(i);
  }

  UnscentedKf::SigmaPointSet out {vec, sigmas};
  return out;
}

Eigen::MatrixXd UnscentedKf::fillMatrixWithVector(const Eigen::VectorXd vec,
                                                  const int numCols) const
{
  int numRows = vec.rows();
  Eigen::MatrixXd mat(numRows, numCols);
  for (int i = 0; i < numCols; ++i)
  {
    mat.col(i) = vec;
  }
  return mat;
}

void UnscentedKf::setWeightsAndCoeffs()
```

```
{
  lambda = (pow(ALPHA, 2) * (numStates + KAPPA)) - numStates;
  sigmaPointScalingCoeff = sqrt(numStates + lambda);

  // Set up mean weights
  meanWeights = Eigen::VectorXd::Zero(2 * numStates + 1);
  meanWeights(0) = lambda / (numStates + lambda);
  for (int i = 1; i < meanWeights.rows(); ++i)
  {
    meanWeights(i) = 1 / (2 * (numStates + lambda));
  }

  // Set up covariance weights
  covarianceWeights = meanWeights;
  covarianceWeights(0) += (1 - pow(ALPHA, 2) + BETA);
}
```

Appendix D

QuadUkf.h Listing

```
#ifndef QUADUKF_H_
#define QUADUKF_H_

#include "UnscentedKf.h"

#include "ros/ros.h"
#include "geometry_msgs/PoseStamped.h"
#include "geometry_msgs/PoseWithCovarianceStamped.h"
#include "geometry_msgs/PoseArray.h"
#include "sensor_msgs/Imu.h"
#include "std_msgs/Empty.h"

#include <mutex>

class QuadUkf : public UnscentedKf
{
public:
  QuadUkf(ros::Publisher poseStampedPub, ros::Publisher poseWithCovStampedPub,
          ros::Publisher poseArrayPub);
  QuadUkf(QuadUkf&& other);
  ~QuadUkf();

  void imuCallback(const sensor_msgs::ImuConstPtr &msg_in);
  void poseCallback(
      const geometry_msgs::PoseWithCovarianceStampedConstPtr &msg_in);

  Eigen::VectorXd processFunc(const Eigen::VectorXd stateVec, const double dt);
  Eigen::VectorXd observationFunc(const Eigen::VectorXd stateVec);
```

```
private:
  struct QuadState
  {
    Eigen::Vector3d position;
    Eigen::Quaterniond quaternion;
    Eigen::Vector3d velocity;
    Eigen::Vector3d angular_velocity;
    Eigen::Vector3d acceleration;
  };

  struct QuadBelief
  {
    double timeStamp;
    double dt;
    QuadUkf::QuadState state;
    Eigen::MatrixXd covariance;
  } lastBelief;

  enum stateVars
  {
    POS_X = 0, POS_Y = 1, POS_Z = 2, QUAT_X = 3, QUAT_Y = 4, QUAT_Z = 5,
    QUAT_W = 6, VEL_X = 7, VEL_Y = 8, VEL_Z = 9, ANGVEL_X = 10, ANGVEL_Y = 11,
    ANGVEL_Z = 12, ACCEL_X = 13, ACCEL_Y = 14, ACCEL_Z = 15
  };

  Eigen::MatrixXd ProcessCovMatrixQ;
  Eigen::MatrixXd SensorCovMatrixR;
  const double Q_SCALING_COEFF = 0.01;
  const double R_SCALING_COEFF = 0.01;

  geometry_msgs::PoseWithCovarianceStamped lastPoseMsg;
  geometry_msgs::PoseArray quadPoseArray;
  const int POSE_ARRAY_SIZE = 10000;  // number of poses to keep for plotting

  const Eigen::Vector3d GRAVITY_ACCEL {0, 0, -9.81};

  std::timed_mutex mtx;

  //Eigen::MatrixXd ProcessCovMatrixQ(const double dt) const;

  ros::Publisher poseStampedPublisher;
  ros::Publisher poseWithCovStampedPublisher;
```

```
  ros::Publisher poseArrayPublisher;

  geometry_msgs::PoseStamped quadBeliefToPoseStamped(const QuadBelief qb) const;
  geometry_msgs::PoseWithCovarianceStamped quadBeliefToPoseWithCovStamped(
      const QuadBelief qb) const;
  void publishAllPoseMessages(const QuadBelief qb);
  void updatePoseArray(const geometry_msgs::PoseWithCovarianceStamped p);

  Eigen::Quaterniond checkQuatContinuity(
      const Eigen::Quaterniond lastQuat,
      const Eigen::Quaterniond nextQuat) const;
  Eigen::MatrixXd quatIntegrationMatrix(const Eigen::Vector3d angVel) const;

  Eigen::VectorXd quadStateToEigen(const QuadUkf::QuadState qs) const;
  QuadUkf::QuadState eigenToQuadState(const Eigen::VectorXd x) const;
};

#endif  // QUADUKF_H_
```

Appendix E

QuadUkf.cpp Listing

```
#include "QuadUkf.h"

QuadUkf::QuadUkf(ros::Publisher poseStampedPub,
                 ros::Publisher poseWithCovStampedPub,
                 ros::Publisher poseArrayPub)
{
  poseStampedPublisher = poseStampedPub;
  poseWithCovStampedPublisher = poseWithCovStampedPub;
  poseArrayPublisher = poseArrayPub;

  numStates = 16;
  numSensors = 10;
  this->UnscentedKf::setWeightsAndCoeffs();

  // Define initial position, quaternion, velocity, angular velocity, and
  // acceleration.
  Eigen::Quaterniond initQuat = Eigen::Quaterniond::Identity();
  Eigen::Vector3d initPosition, initVelocity, initAngVel, initAcceleration;
  initPosition << 0, 0, 1;  // "x = 0, y = 0, z = 1 meter above origin"
  initVelocity = Eigen::Vector3d::Zero();
  initAngVel = Eigen::Vector3d::Zero();
  initAcceleration = Eigen::Vector3d::Zero();

  // Define initial belief
  QuadUkf::QuadState initState {initPosition, initQuat, initVelocity,
                                initAngVel, initAcceleration};
  Eigen::MatrixXd initCov = Eigen::MatrixXd::Identity(numStates, numStates);
  initCov = initCov * 0.01;
  double initTimeStamp = ros::Time::now().toSec();
```

```
  double init_dt = 0.0001;
  QuadUkf::QuadBelief initBelief {initTimeStamp, init_dt, initState, initCov};
  lastBelief = initBelief;

  SensorCovMatrixR = R_SCALING_COEFF
      * Eigen::MatrixXd::Identity(numSensors, numSensors);
  ProcessCovMatrixQ = Q_SCALING_COEFF
      * Eigen::MatrixXd::Identity(numStates, numStates);

  // Initialize last PTAM message for pseudovelocity corrections
  lastPoseMsg.header.stamp.sec = initTimeStamp;
  lastPoseMsg.pose.pose.position.x = initPosition(0);
  lastPoseMsg.pose.pose.position.y = initPosition(1);
  lastPoseMsg.pose.pose.position.z = initPosition(2);

  // Initialize pose array for visualization in Rviz
  quadPoseArray.poses.clear();
  quadPoseArray.header.frame_id = "map";
  quadPoseArray.header.stamp = ros::Time();
}

QuadUkf::QuadUkf(QuadUkf&& other)
{
  std::lock_guard<std::timed_mutex> lock(other.mtx);

  SensorCovMatrixR = std::move(other.SensorCovMatrixR);
  other.SensorCovMatrixR = Eigen::MatrixXd::Zero(1, 1);

  ros::NodeHandle n;
  poseStampedPublisher = std::move(other.poseStampedPublisher);
  poseWithCovStampedPublisher = std::move(other.poseWithCovStampedPublisher);
  poseArrayPublisher = std::move(other.poseArrayPublisher);
  ros::Publisher p = n.advertise<std_msgs::Empty>("empty", 1);
  other.poseWithCovStampedPublisher = p;
  other.poseArrayPublisher = p;
}

QuadUkf::~QuadUkf()
{
}

void QuadUkf::imuCallback(const sensor_msgs::ImuConstPtr &msg_in)
```

```
{
  mtx.try_lock_for(std::chrono::milliseconds(100));

  QuadBelief xHat = lastBelief;
  xHat.state.angular_velocity(0) = msg_in->angular_velocity.x;
  xHat.state.angular_velocity(1) = -msg_in->angular_velocity.y;
  xHat.state.angular_velocity(2) = msg_in->angular_velocity.z;
  xHat.state.acceleration(0) = -msg_in->linear_acceleration.x;
  xHat.state.acceleration(1) = msg_in->linear_acceleration.y;
  xHat.state.acceleration(2) = msg_in->linear_acceleration.z;

  // Remove gravity
  xHat.state.acceleration = xHat.state.acceleration
      - xHat.state.quaternion.toRotationMatrix().inverse() * GRAVITY_ACCEL;

  // Predict next state and reset lastBelief
  Eigen::VectorXd x = quadStateToEigen(xHat.state);
  xHat.dt = msg_in->header.stamp.toSec() - lastBelief.timeStamp;
  UnscentedKf::Belief b = predictState(x, xHat.covariance, ProcessCovMatrixQ,
                                       xHat.dt);
  QuadUkf::QuadBelief qb {msg_in->header.stamp.toSec(), xHat.dt,
                          eigenToQuadState(b.state), b.covariance};
  qb.state.quaternion = checkQuatContinuity(lastBelief.state.quaternion,
                                            qb.state.quaternion);
  lastBelief = qb;

  publishAllPoseMessages(lastBelief);

  mtx.unlock();
}

void QuadUkf::poseCallback(
    const geometry_msgs::PoseWithCovarianceStampedConstPtr &msg_in)
{
  mtx.try_lock_for(std::chrono::milliseconds(100));

  Eigen::VectorXd z(numSensors);
  z(POS_X) = -msg_in->pose.pose.position.x;
  z(POS_Y) = msg_in->pose.pose.position.y;
  z(POS_Z) = msg_in->pose.pose.position.z;
  z(QUAT_X) = msg_in->pose.pose.orientation.w;
  z(QUAT_Y) = -msg_in->pose.pose.orientation.z;
```

```
z(QUAT_Z) = msg_in->pose.pose.orientation.y;
z(QUAT_W) = msg_in->pose.pose.orientation.x;

// Pseudovelocity correction
double dtPose = msg_in->header.stamp.toSec()
    - lastPoseMsg.header.stamp.toSec();
z(VEL_X) = (z(POS_X) - lastPoseMsg.pose.pose.position.x) / dtPose;
z(VEL_Y) = (z(POS_Y) - lastPoseMsg.pose.pose.position.y) / dtPose;
z(VEL_Z) = (z(POS_Z) - lastPoseMsg.pose.pose.position.z) / dtPose;

// Update lastPoseMsg
lastPoseMsg.header = msg_in->header;
lastPoseMsg.pose.pose.position.x = z(POS_X);
lastPoseMsg.pose.pose.position.y = z(POS_Y);
lastPoseMsg.pose.pose.position.z = z(POS_Z);

// Check incoming quaternion for rotational continuity and replace if not
// continuous.
Eigen::Quaterniond inQuat;
inQuat.x() = z(QUAT_X);
inQuat.y() = z(QUAT_Y);
inQuat.z() = z(QUAT_Z);
inQuat.w() = z(QUAT_W);
Eigen::Vector4d chosenQuat = checkQuatContinuity(lastBelief.state.quaternion,
                                                 inQuat).coeffs();
z.block<4, 1>(3, 0) = chosenQuat;

// Set time step "dt".
double dt = msg_in->header.stamp.toSec() - lastBelief.timeStamp;

QuadUkf::QuadBelief xHat = lastBelief;
xHat.state.velocity = lastBelief.state.velocity
    + lastBelief.state.acceleration * dt;
xHat.state.position = (xHat.state.velocity + lastBelief.state.velocity) / 2.0
    * dt + lastBelief.state.position;
Eigen::MatrixXd Theta = quatIntegrationMatrix(
    lastBelief.state.angular_velocity);
xHat.state.quaternion.coeffs() = lastBelief.state.quaternion.coeffs()
    + 0.5 * Theta * lastBelief.state.quaternion.coeffs() * dt;
xHat.state.quaternion.normalize();
Eigen::VectorXd xPred = quadStateToEigen(xHat.state);
Eigen::MatrixXd P = lastBelief.covariance;
```

```
  UnscentedKf::Belief currStateAndCov = correctState(xPred, P, z,
                                                     SensorCovMatrixR);

  // Update lastBelief.
  lastBelief.dt = dt;
  lastBelief.state = eigenToQuadState(currStateAndCov.state);
  lastBelief.covariance = currStateAndCov.covariance;
  lastBelief.timeStamp = msg_in->header.stamp.toSec();

  publishAllPoseMessages(lastBelief);

  mtx.unlock();
}

void QuadUkf::publishAllPoseMessages(const QuadUkf::QuadBelief b)
{
  const geometry_msgs::PoseWithCovarianceStamped pwcs =
      quadBeliefToPoseWithCovStamped(b);
  poseWithCovStampedPublisher.publish(pwcs);
  updatePoseArray(pwcs);
  const geometry_msgs::PoseStamped ps = quadBeliefToPoseStamped(b);
  poseStampedPublisher.publish(ps);
}

/*
 * Puts a given pose into the first position of quadPoseArray. Once
 * quadPoseArray reaches POSE_ARRAY_SIZE, the last pose is popped on each call.
 * After these operations are performed, this function publishes
 * quadPoseArray.
 */
void QuadUkf::updatePoseArray(const geometry_msgs::PoseWithCovarianceStamped p)
{
  quadPoseArray.poses.insert(quadPoseArray.poses.begin(), 1, p.pose.pose);
  if (quadPoseArray.poses.size() > POSE_ARRAY_SIZE)
  {
    quadPoseArray.poses.pop_back();
  }
  poseArrayPublisher.publish(quadPoseArray);
}

/*
```

```
 * Ensures rotational continuity by checking for sign-flipping in the
 * orientation quaternion.
 */
Eigen::Quaterniond QuadUkf::checkQuatContinuity(
    const Eigen::Quaterniond lastQuat, const Eigen::Quaterniond nextQuat) const
{
  Eigen::Vector4d lastVec, nextVec;
  lastVec = lastQuat.normalized().coeffs();
  nextVec = nextQuat.normalized().coeffs();

  double sum = (lastVec + nextVec).norm();
  double diff = (lastVec - nextVec).norm();

  Eigen::Quaterniond out;
  if (sum > diff)
  {
    out.coeffs() = nextVec;
  }
  else
  {
    out.coeffs() = -nextVec;
  }
  return out;
}

geometry_msgs::PoseStamped QuadUkf::quadBeliefToPoseStamped(
    const QuadUkf::QuadBelief b) const
{
  geometry_msgs::PoseStamped p;
  p.header.stamp.sec = b.timeStamp;
  p.header.stamp.nsec = (b.timeStamp - floor(b.timeStamp)) * pow(10, 9);
  p.pose.position.x = b.state.position(0);
  p.pose.position.y = b.state.position(1);
  p.pose.position.z = b.state.position(2);
  p.pose.orientation.w = b.state.quaternion.w();
  p.pose.orientation.x = b.state.quaternion.x();
  p.pose.orientation.y = b.state.quaternion.y();
  p.pose.orientation.z = b.state.quaternion.z();

  return p;
}
```

```
geometry_msgs::PoseWithCovarianceStamped QuadUkf::quadBeliefToPoseWithCovStamped(
    const QuadUkf::QuadBelief b) const
{
  geometry_msgs::PoseWithCovarianceStamped p;
  p.header.stamp.sec = b.timeStamp;
  p.header.stamp.nsec = (b.timeStamp - floor(b.timeStamp)) * pow(10, 9);
  p.pose.pose.position.x = b.state.position(0);
  p.pose.pose.position.y = b.state.position(1);
  p.pose.pose.position.z = b.state.position(2);
  p.pose.pose.orientation.w = b.state.quaternion.w();
  p.pose.pose.orientation.x = b.state.quaternion.x();
  p.pose.pose.orientation.y = b.state.quaternion.y();
  p.pose.pose.orientation.z = b.state.quaternion.z();

  // Copy covariance matrix from b into the covariance array in p
  Eigen::MatrixXd covMat = b.covariance.block<6, 6>(0, 0);
  for (int i = 0; i < covMat.rows() * covMat.cols(); ++i)
  {
    p.pose.covariance[i] = covMat(i);
  }

  return p;
}

Eigen::VectorXd QuadUkf::processFunc(const Eigen::VectorXd x, const double dt)
{
  QuadUkf::QuadState prevState = eigenToQuadState(x);
  prevState.quaternion.normalize();
  QuadUkf::QuadState currState;

  // Compute current orientation via quaternion integration.
  Eigen::MatrixXd Theta = quatIntegrationMatrix(prevState.angular_velocity);
  currState.quaternion.coeffs() = prevState.quaternion.coeffs()
      + 0.5 * Theta * prevState.quaternion.coeffs() * dt;
  currState.quaternion.normalize();

  // Rotate current and previous accelerations into inertial frame, then
  // average them.
  currState.acceleration = prevState.quaternion.toRotationMatrix()
      * prevState.acceleration;

  // Compute current velocity by integrating current acceleration.
```

```
  currState.velocity = prevState.velocity
      + 0.5 * (lastBelief.state.acceleration + currState.acceleration) * dt;

  // Compute current position by integrating current velocity.
  currState.position = prevState.position
      + 0.5 * (currState.velocity + prevState.velocity) * dt;

  // Angular velocity is assumed to be correct as measured.
  currState.angular_velocity = prevState.angular_velocity;

  return quadStateToEigen(currState);
}

Eigen::VectorXd QuadUkf::observationFunc(const Eigen::VectorXd stateVec)
{
  return stateVec.head(numSensors);
}

/*
 * Given a vector of angular velocities in radians per second, returns the
 * 4-by-4 angular rate integration matrix.
 */
Eigen::MatrixXd QuadUkf::quatIntegrationMatrix(
    const Eigen::Vector3d angVel) const
{
  Eigen::MatrixXd Theta(4, 4);

  // Upper left 3-by-3 block: negative skew-symmetric matrix of vector w
  Theta(0, 0) = 0;
  Theta(0, 1) = angVel(2);
  Theta(0, 2) = -angVel(1);

  Theta(1, 0) = -angVel(2);
  Theta(1, 1) = 0;
  Theta(1, 2) = angVel(0);

  Theta(2, 0) = angVel(1);
  Theta(2, 1) = -angVel(0);
  Theta(2, 2) = 0;

  // Bottom left 1-by-3 block: negative transpose of vector w
  Theta.block<1, 3>(3, 0) = -angVel.transpose();
```

```
  // Upper right 3-by-1 block: w
  Theta.block<3, 1>(0, 3) = angVel;

  // Bottom right 1-by-1 block: 0
  Theta(3, 3) = 0;

  return Theta;
}

Eigen::VectorXd QuadUkf::quadStateToEigen(const QuadUkf::QuadState qs) const
{
  Eigen::VectorXd x(numStates);

  x(POS_X) = qs.position(0);
  x(POS_Y) = qs.position(1);
  x(POS_Z) = qs.position(2);

  x(QUAT_X) = qs.quaternion.x();
  x(QUAT_Y) = qs.quaternion.y();
  x(QUAT_Z) = qs.quaternion.z();
  x(QUAT_W) = qs.quaternion.w();

  x(VEL_X) = qs.velocity(0);
  x(VEL_Y) = qs.velocity(1);
  x(VEL_Z) = qs.velocity(2);

  x(ANGVEL_X) = qs.angular_velocity(0);
  x(ANGVEL_Y) = qs.angular_velocity(1);
  x(ANGVEL_Z) = qs.angular_velocity(2);

  x(ACCEL_X) = qs.acceleration(0);
  x(ACCEL_Y) = qs.acceleration(1);
  x(ACCEL_Z) = qs.acceleration(2);

  return x;
}

QuadUkf::QuadState QuadUkf::eigenToQuadState(const Eigen::VectorXd x) const
{
  QuadUkf::QuadState qs;
```

```
  qs.position(0) = x(POS_X);
  qs.position(1) = x(POS_Y);
  qs.position(2) = x(POS_Z);

  qs.quaternion.x() = x(QUAT_X);
  qs.quaternion.y() = x(QUAT_Y);
  qs.quaternion.z() = x(QUAT_Z);
  qs.quaternion.w() = x(QUAT_W);

  qs.velocity(0) = x(VEL_X);
  qs.velocity(1) = x(VEL_Y);
  qs.velocity(2) = x(VEL_Z);

  qs.angular_velocity(0) = x(ANGVEL_X);
  qs.angular_velocity(1) = x(ANGVEL_Y);
  qs.angular_velocity(2) = x(ANGVEL_Z);

  qs.acceleration(0) = x(ACCEL_X);
  qs.acceleration(1) = x(ACCEL_Y);
  qs.acceleration(2) = x(ACCEL_Z);

  return qs;
}
```

Appendix F

CMakeLists.txt Listing

```
cmake_minimum_required(VERSION 2.8.3)
project(kalman_sense)

find_package(catkin REQUIRED COMPONENTS
  roscpp
  std_msgs
)
find_package(Eigen3 REQUIRED )
catkin_package()

set (CMAKE_CXX_FLAGS "--std=gnu++11 ${CMAKE_CXX_FLAGS}")

include_directories(
  ${catkin_INCLUDE_DIRS}
  ${EIGEN3_INCLUDE_DIR}
)

message( STATUS "*****************************************")
message( STATUS "Eigen include:  ${EIGEN3_INCLUDE_DIR}")
message( STATUS "*****************************************")

add_executable(node src/node.cpp
                    src/QuadUkf.cpp
                    src/UnscentedKf.cpp
)

target_link_libraries( node
   ${catkin_LIBRARIES}
)
```