THESIS TITLE: Design and Evaluation of an Indoor Navigation and Mapping System for Autonomous Vehicles

AUTHOR: Benjamin Arneberg

M.S. Thesis Approved as an Elective towards the Master of Science Degree in Computer Science.

Marcette Warzala
Thesis Advisor

Date

Date

Marcette Warzala
Thesis Reader

Date

Date

Marcette Warzala
Thesis Reader

Date

Date

GRADUATE SCHOOL APPROVAL:

Date

Date

Date

Date

ORIGINAL COPY DEPOSITED IN LIBRARY:

Date

Date

COPY RECEIVED IN GRADUATE SCHOOL OFFICE:

Date

Date

Distribution:
One Copy to Thesis Advisor
One Copy to Each Member of Thesis Committee
One Copy to Director of Graduate School
One Copy to Graduate School Office (with signature approval sheet, including all signatures)
Design and Evaluation of an Indoor Navigation and Mapping System for Autonomous Vehicles

by

Benjamin John Arneberg

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Science

Supervised by

Assistant Professor Dr. Marsette Vona
College of Computer and Information Science
Northeastern University
Boston, MA
April 2014

Approved by:

[Signature]
Dr. Marsette Vona, Assistant Professor
Thesis Advisor, College of Computer and Information Science

[Signature]
Andrew Reiter, Member of Technical Staff
Committee Member, Draper Laboratory

[Signature]
Dr. Paul DeBitteto, Principal Member of Technical Staff
Committee Member, Draper Laboratory
Thesis Release Permission Form

Northeastern University
College of Computer and Information Science

Title:

Design and Evaluation of an Indoor Navigation and Mapping System for Autonomous Vehicles

©2014 Benjamin John Arneberg
The author hereby grants to Northeastern University and The Charles Stark Draper Laboratory, Inc. permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole or in any part medium now known or hereafter created. The views expressed in this thesis are those of the author and do not necessarily reflect the official policy or position of the Air Force, the Department of Defense or the U.S. Government.

Benjamin John Arneberg

Date
Acknowledgments

It is with immense gratitude that I acknowledge the support of Draper Laboratory for funding, and the assistance of Andrew Reiter, Marsette Vona, Matthew Graham, Peter Lommel, Timothy Dyer, and Theodore Steiner in various areas of my research.
Abstract
Design and Evaluation of an Indoor Navigation and Mapping System for Autonomous Vehicles

Benjamin John Arneberg

Supervising Professor: Dr. Marsette Vona

This thesis presents a quadrotor vehicle that performs onboard Simultaneous Localization and Mapping, allowing it to navigate and create 3D maps in unknown environments. Various methods used to arrive at this end result are discussed, as well as a comparison of existing navigation algorithms and visual odometry methods. This thesis makes several improvements on existing navigation algorithms, and runs the most promising configuration in real-time onboard a custom quadrotor system for a full flight demo.
# Contents

Acknowledgments .................................................. iii

Abstract .............................................................. iv

1 Introduction ...................................................... 1  
  1.1 Research Objective ........................................... 2  
  1.2 Contributions ................................................ 3  

2 Background ....................................................... 4  
  2.1 Navigation Algorithms ....................................... 4  
    2.1.1 Visual Odometry ........................................... 4  
    2.1.2 Simultaneous Localization and Mapping .................. 6  
    2.1.3 Robot Operating System Framework ....................... 8  

3 Related Work ................................................... 10  
  3.1 Early Work with Kinect, GPU-Based Algorithms ............... 10  
  3.2 Using RGB-D Cameras for Navigation and Mapping ............ 11  
  3.3 Navigation and Mapping with Cameras and other Sensors ..... 12  
  3.4 Navigation and Mapping with Aircraft ....................... 13  

4 System Hardware Design ....................................... 14  
  4.1 Approach ...................................................... 14  
    4.1.1 Camera Selection ......................................... 15  
    4.1.2 Quadrotor Selection ....................................... 16  
    4.1.3 Computing Platform ...................................... 17  
    4.1.4 Sensor Fusion .............................................. 20  
  4.2 Results ....................................................... 21  
    4.2.1 Combining Hardware for Offboard SLAM .................. 21  
    4.2.2 Onboard SLAM ............................................. 24  
    4.2.3 Sensor Fusion ............................................. 24
## Navigation Algorithm Development

- 5.1 Improving the CCNY Algorithm with IMU
- 5.2 Integrating CCNY with DVO
- 5.3 Switching between CCNY and DVO
- 5.4 Weighing and Combining CCNY and DVO
- 5.5 Combining CCNY and DVO Cost Functions

## Experimental Results

- 6.1 Testing and Evaluation Methods
  - 6.1.1 Evaluation Metrics
- 6.2 Evaluation of Algorithms on Freiburg Datasets
  - 6.2.1 “Freiburg1 Room”
  - 6.2.2 “Freiburg2 Desk”
  - 6.2.3 “Freiburg3 Long Office”
- 6.3 Evaluation of Algorithms on Building Datasets
  - 6.3.1 “Building QVGA”
  - 6.3.2 “Building Slow”
  - 6.3.3 “Building Fast”
- 6.4 Quadrotor with groundtruth

## Conclusion

- 7.1 Analysis of Results
- 7.2 Future Work
  - 7.2.1 Conclusion

## Appendix

- A.1 ANEES
- A.2 DVO Jacobian
- B.1 AprilTag

## Bibliography
Chapter 1

Introduction

Unmanned Aerial vehicles (UAVs) have been increasing in popularity due to their usefulness in areas such as surveillance, aerial photography, disaster monitoring, and more. Quadrotors (four-propeller rotorcrafts) are popular UAVs due to their small size, maneuverability, and decreasing costs.

Quadrotors are often used to assist in disaster-related scenarios. A quadrotor was used to inspect a damaged building in Japan following the 2011 Tohoku earthquake, seen in Figure 1.1 [1]. Another quadrotor was used to inspect a damaged cathedral from a 2011 New Zealand earthquake, which was too dangerous for a human to enter [2].

Figure 1.1: Quadrotor inspecting the damaged building [1]

Both of these scenarios, however, required an operator to manage the vehicle via remote control. The operator must have a prior knowledge of the building’s layout to fly the
quadrotor from point to point. In a severely damaged building or in a building that is on fire this would be much more difficult. Additionally, manual control requires an uninterrupted high-bandwidth link to stream video, which is not always available in disaster scenarios. Moreover, unanticipated obstacles in the environment might be difficult for the operator to distinguish from the video stream. An alternative to operating from video streams is operating from line-of-sight, but this is too dangerous for the operator in many situations.

For these reasons, it is desirable to have a UAV that can autonomously explore a disaster area. Such a vehicle could create maps of the new environment, giving first responders a visualization of what to expect and inform them where to focus their efforts (removing the need to be unnecessarily put in harm’s way). This would require neither a skilled pilot nor constant attention on the UAV.

Search and rescue scenarios often occur indoors where no GPS signal is present. For a vehicle to operate autonomously in such an unknown environment, it needs to perform simultaneous localization and mapping (SLAM). That is, the vehicle must build up a map of the current environment while also locating itself within that unknown environment (without relying on GPS). The SLAM problem is an open research question that has been explored for more than two decades, and will be touched on more in the background section.

1.1 Research Objective

With the end goal of allowing UAVs to autonomously assist in search and rescue scenarios, this Master’s thesis will focus on the SLAM portion of autonomous search and rescue. This thesis looks at integrating a SLAM system with a quadrotor in order to navigate in unknown areas while building 3D maps of the environment. The platform should allow autonomous operation of the quadrotor to be easily added in the future. The system hardware must be lightweight enough to fly on a small quadrotor. Additionally, the software must run in real-time to provide live control to the quadrotor.
1.2 Contributions

This thesis integrates navigation and mapping solutions with a quadrotor. It improves upon the visual odometry and navigation accuracy of existing solutions. Specifically, it evaluates existing sensors and algorithms with which to create a SLAM-capable quadrotor, combining and modifying them to create a more robust solution. A discussion of which approaches are more accurate for different situations is given, as well as the various methods used to combine the different approaches.
Chapter 2

Background

This chapter provides an overview of the different navigation algorithms used in this thesis, and explains the two primary navigation techniques utilized—visual odometry (VO) and simultaneous localization and mapping (SLAM). It also discusses the Robot Operating system (ROS), the software framework used in this thesis.

2.1 Navigation Algorithms

This thesis uses two different classes of VO methods that both occur within a SLAM framework. The background on VO and SLAM and the two types of VO used will now be examined.

2.1.1 Visual Odometry

Visual odometry, a term coined by Nister et al, is the method of estimating the motion of a vehicle from visual input alone [3]. The VO process incrementally calculates the vehicle’s pose by examining the changes its motion causes on the camera images. VO is only concerned with the local consistency of the trajectory, which can be more accurately estimated with techniques such as bundle adjustment [4].

Two camera positions at adjacent times \( k \) and \( k - 1 \) can be related by the rigid body transformation \( T \in \mathbb{R}^{4\times4} \):
\[ T = \begin{bmatrix} R_{k,k-1} & t_{k,k-1} \\ 0 & 1 \end{bmatrix} \]  \hspace{1cm} (2.1)

where \( R_{k,k-1} \in SO(3) \) is the rotation matrix, and \( t_{k,k-1} \in \mathbb{R}^{3 \times 1} \) is the translation vector. Thus the set \( T_{1:n} = T_1, ..., T_n \) contains all ensuing motions. The set of camera poses \( C_{0:n} = C_0, ..., C_n \) contains the camera transformations with respect to the initial coordinate frame at \( k = 0 \). The camera’s current pose \( C_n \) can be found by concatenating the transformations \( C_n = T_n...T_1, C_0 \). The main job of VO is to compute the relative transformations \( T_k \) from the camera’s images, and incrementally concatenate the resulting transformations to yield the entire trajectory \( C_{0:n} \) [4].

There are two main classes of methods used in VO to calculate the transformation between images. Sparse methods extract features from the images and match features between images to generate the transformation. Dense methods utilize all the pixels in each image to calculate the transformation [4]. Both are summarized below.

**Sparse VO: Iterative Closest Point**

Numerous methods can be used to calculate the camera motion for sparse visual odometry. A 2-D method for motion estimation is 2-D feature matching between frames [4]. For 3D motion estimation, a method commonly used is Iterative Closest Point (ICP), first proposed by Zhang [5] and used in this thesis. ICP computes the transformation required to align two sets of points. A least-squares technique is used to iteratively align the points in both sets as closely as possible. It should be noted that ICP can also be used for dense VO, but since this thesis focuses on a method that employs sparse ICP, the explanation of ICP will be in regards to its use as a sparse VO technique.

For a given image set, the algorithm first pairs the 3D points of the features with the closest 3D points in a second image or model, respectively \( \mathbf{x}_i \in \mathbf{X} \) and \( \mathbf{y}_i \in \mathbf{Y} \). With \( N \) as
the number of associated pairs, ICP minimizes the cost function:

$$F(R, t) = \frac{1}{N} \sum_{i=1}^{N} ||(Rx_i + t) - y_i||^2 \quad (2.2)$$

to find the rotation $R$ and translation $t$ that best align all pairs of points. For each iteration, the algorithm updates the point associations between the first and second image or model. The algorithm terminates when $\delta R$ and $\delta t$, the changes in rotation and translation using chosen metrics, respectively, fall below a threshold, or the number of iterations exceeds a predefined limit.

**Dense VO: Minimizing Photometric Error**

Several methods can be used to calculate the camera motion with dense visual odometry. One is dense ICP, which works in the same way as ICP above, except it matches all pixels in a camera frame to a prior frame or model, not just extracted features. Another technique and one used in this thesis is minimizing photometric error between image sets. This algorithm estimates the camera motion by aligning two consecutive intensity images $I_1$ and $I_2$. A point $p$ observed in both images should yield the same brightness, hence

$$I_1(x) = I_2(\tau(\xi, x)) \quad (2.3)$$

where $\tau(\xi, x)$ is the warping function that maps a pixel coordinate from the first image to the second image, given camera motion $\xi \in \mathbb{R}^6$. This algorithm finds the camera motion $\xi$ that causes the most photo-consistency for all pixels [6].

### 2.1.2 Simultaneous Localization and Mapping

Simultaneous localization and mapping is closely related to visual odometry. The classic SLAM problem is for a mobile robot to be placed in an unknown environment, building
up a map of the environment and localizing itself within that map [7]. While VO maintains a locally consistent trajectory estimate, SLAM produces a globally consistent trajectory and maintains a global estimate of observed landmarks. Maintaining a global map allows SLAM to recognize when a robot has returned to an already-seen location, called loop closure. VO and SLAM are subject to drift over time since each incremental trajectory update has errors, but SLAM’s loop closure seeks to eliminate drift while VO’s drift continues unbounded.

The navigation algorithms in this work occur within a SLAM framework advanced by Dryanovski et al. at the City College of New York (CCNY) [8]. This variant of EKF-SLAM maintains a 3D model of all seen features stored in a ring buffer. For each camera frame, a nearest-neighbor search is used to generate correspondences between the model and the current frame’s features, running ICP to align the model’s points to the frame’s corresponding features, computing the delta motion. The pose refinement from ICP updates the 3D model of features seen; all associated features undergo a Kalman Filter update. We use the filter introduced in [8], shown below for completeness.

The set of 3D features in the current frame can be represented as the dataset $D$, where each feature $d = \{\mu^D, \Sigma^D\}$ has a mean and covariance matrix. The state vector $\mu$ contains the $(x, y, z)$ coordinates of the given point in the frame, and $\Sigma$ represents the point’s 3D uncertainty. The model is similarly represented by the set $M$, with each member $m = \{\mu^M, \Sigma^M\}$.

With the distribution $d$ as the observation and distribution $m$ as the prior, the predicted distribution is the same as the prior state of the model at time $t-1$:

$$\tilde{\mu}_t = \mu_{t-1}^M$$

$$\tilde{\Sigma}_t = \Sigma_{t-1}^M$$
For associated features in the dataset, the Kalman gain at timestep $t$ is

$$K_t = \Sigma_t (\Sigma_t + \Sigma_t^D)^{-1}$$

(2.6)

The measurements are updated with

$$\mu_t^M = \bar{\mu}_t + K_t (\mu_t^D - \bar{\mu}_t)$$

(2.7)

Finally, the covariance is propagated:

$$\Sigma_t^M = (I - K_t) \Sigma_t$$

(2.8)

Any features not yet seen are added as new members into the model. Loop closure is performed implicitly, as already-seen features in the current frame will be matched to those same features in the model, as long as they are still in the ring buffer. A back-end pose-graph optimization system is also available to perform offline long-term loop closure, based upon the g2o framework [9].

Chapter five of this work analyzes combining the dense VO technique of minimizing photometric error with the sparse VO method of ICP all within CCNY’s SLAM framework. That is, the dense visual odometry algorithm by Kerl et al. [6] and ICP’s trajectory estimates by Dryanovski et al. [8] are combined to compute the camera motion and update landmarks and the global map accordingly.

2.1.3 Robot Operating System Framework

The robot operating system is used as the framework for flying the quadrotors with the navigation algorithms. ROS is a flexible open-source framework for developing robotics software. It acts as a middleware, streamlining the process of combining different sensors and algorithms through the use of nodes and topics. Nodes are executables that perform
specific tasks and communicate with other nodes via ROS topics. For example, a program might have a camera node that pulls data from the camera, which communicates with a navigation node that generates a navigation pose from the camera data. For more information on ROS, the interested reader is invited to view www.ros.org.
Chapter 3

Related Work

A variety of sensors and hardware platforms can be used to perform navigation and mapping, with both dense and sparse methods. Since the advent of the Microsoft Kinect, researchers have published a myriad of papers that focus on using this device as the primary sensor for running such algorithms. Stereo, mono, and time-of-flight cameras are also used, but can rarely surpass the Kinect and similar structured light RGB-D sensors in the overall combination of lightness, accuracy, and simplicity. Laser scanners and Inertial Measurement Units (IMUs) can also be used to compute the overall solution. Of particular interest are works that apply these sensors and algorithms on aircraft.

Section 3.1 will focus on papers that utilize the Kinect and similar RGB-D sensors in conjunction with Graphics Processing Unit (GPU) algorithms to perform navigation and mapping. Section 3.2 looks at research with non-GPU algorithms that use RGB-D cameras to conduct navigation and mapping. Section 3.3 explores research on fusing other sensors with the camera data to perform SLAM. Finally, section 3.4 looks at works that apply the sensors and algorithms for SLAM on aircraft.

3.1 Early Work with Kinect, GPU-Based Algorithms

KinectFusion by Newcombe, Davison, et al. was one of the first algorithms to take advantage of the capabilities offered by the Kinect, producing a volumetric representation of a scene with unprecedented accuracy [10]. It uses ICP to track the current camera frame
against a global model, which is maintained with a volumetric, truncated signed distance function (TSDF). This approach is computationally intensive, requiring a GPU to run in real-time.

Similar works followed, such as Roth and Vona’s Moving Volume KinectFusion [11]. The original KinectFusion is locked to a fixed volume in space, and Moving Volume KinectFusion allows that volume to translate and rotate through space, providing visual odometry. Kintinuous by Whelan, Kaess et al. expand on this by allowing the volume to move and extracting a dense point cloud from regions that leave the volume [12]. This results in the ability to create dense 3D maps of the environment. In their most recent work, Kintinuous is improved to perform SLAM, where a pose graph system accomplishes deformation-based loop closure [13].

Moreno, Necombe et al. also extend KinectFusion by implementing object-based SLAM with their SLAM++ algorithm [14]. Using prior knowledge of objects in a scene, they are able to perform ICP-based camera tracking for each live frame, matching objects to those in the model and performing loop closure for a complete SLAM system. All these algorithms and others employing a TSDF require the use of GPUs due to the intense but parallelizable computations.

3.2 Using RGB-D Cameras for Navigation and Mapping

One of the first approaches to performing RGB-D SLAM was that by Henry, Krainin et al [15]. Unlike the dense TSDF methods, they extract sparse visual features from the RGB frame. They combine this with the depth data to perform ICP, aligning the current frame to the previous frame, and add the new frame to the 3D model. Loop closure is performed, and the end result is a 3D map represented as surfels, which are concise representations of point clouds. Processing is done on a CPU but not in real-time.

Endres, Hess, et al. did similar work with the RGB-D SLAM system, which is available as open-source [16]. Here, sparse features and descriptors combine with depth data and
are matched to prior extracted descriptors, computing the relative transformation between poses. A backend pose graph performs loop closure. The OctoMap framework is used to produce a volumetric 3D representation of the environment [17].

Dryanovski, Valenti, et al. at the City College of New York (CCNY) released an open-source package that performs sparse visual odometry and mapping with an RGB-D camera [8]. They extract sparse features from the RGB image, and combine those with the depth image to obtain 3D points. They then use ICP to associate the points with a model, and update points in the model using a Kalman Filter. This variation of Extended Kalman Filter (EKF) SLAM can be run on a CPU at 30Hz and generates dense 3D point clouds as well as occupancy grid maps of the environment. Implicit loop closure is performed by associating the current frame’s features with features in the model, and loop closure from an offline pose graph optimization is also used.

Kerl, Strum et al.’s dense visual odometry (DVO) system minimizes photometric error between the current and prior camera frame to obtain a navigation solution in real-time [6]. They have also incorporated the DVO system within a SLAM framework, using pose-graph optimization to detect loop closure [18]. They have released both the VO-only and SLAM solutions as open-source.

3.3 Navigation and Mapping with Cameras and other Sensors

Other groups have combined cameras with additional sensors to achieve more accurate navigation and mapping solutions. Voigt, Nikolic et al. combine an IMU with a stereo camera in an EKF to obtain motion estimation [19]. The tight coupling between the two allows the pose to still be tracked when vision fails. Fallon, Johannsson et al. created a SLAM system that combines RGB-D cameras, LIDAR, inertial and barometric sensors to perform scan-matching to infer motion. It performs in real-time and uses pose graph optimization with loop closure detection [20]. Similar work with navigation and mapping by combining cameras with other sensors has been done by [21–24].
3.4 Navigation and Mapping with Aircraft

Current work is being done to put navigation and mapping algorithms on unmanned aircraft, often using cameras and other sensors. Engel, Sturm et al at the Technical University of Munich (TUM) combine monocular SLAM with sensors on the AR Drone quadrotor in an offboard EKF to obtain a navigation solution, and released the code as open source [25]. Shen et al. use a stereo camera with IMU to obtain state estimation and trajectory control on a quadrotor [26]. Blosch et al. solely use a monocular camera mounted on a quadrotor to run SLAM [27].

In his Master’s thesis, Christian Kerl uses a dense visual odometry system to stabilize and control a quadrotor’s position [28]. Lange et al similarly use a Kinect to autonomously navigate a corridor [29]. Huang, Bachrach et al put an RGB-D camera onboard a quadrotor to autonomously plan trajectories and build a dense 3D model of the environment [30]. Control, sparse visual odometry and sensor fusion are done onboard, while SLAM and the occupancy grid map are calculated offboard in real-time; textured surfaces are generated offline. Sturm, Kerl et al. quadrotor-mounted a structured light sensor, tethered with an offboard GPU to generate textured 3D models of indoor spaces. The model is internally represented as a signed distance function [31].

Bry, Bachrach et al demonstrate that a fixed-wing aircraft carrying an IMU and laser range finder can be accurately localized by extending the Gaussian Particle Filter. They partition the state according to measurement independence relationships, which after a pseudo-linear update allows them to use 20x fewer particles while achieving the same level of state estimation. This method allows the aircraft to perform aggressive maneuvers [32]. Shen, Michael et al designed a quadrotor that can autonomously explore multi-floor building using a stochastic differential equation-based exploration algorithm [33]. It utilizes a Kinect, IMU, and laser scanner. All processing is done onboard, and it builds up a sparse 3D map of the environment. Other works that perform navigation and mapping with quadrotors using stereo cameras and laser scanners are [34] and [35].
Chapter 4

System Hardware Design

We underwent several design iterations to select the components required for a quadrotor system that can perform SLAM, where the target environment is indoor office buildings. This chapter will examine the selection process of the camera, quadrotor, computing platform, additional sensors and the fusion of those sensors. We first constructed a proof of concept quadrotor that built upon the AR Drone 2.0. Due to difficulty in syncing the AR Drone’s preexisting onboard sensors with an added 3D imager, we built a second custom quadrotor with onboard processing that could fully integrate an IMU into its solution and perform onboard SLAM.

4.1 Approach

Section 4.1.1 explores the process of choosing a camera that is light enough to fly onboard a quadrotor, but accurate enough to perform SLAM in indoor office environments. The ASUS Xtion was found to have the best combination of lightness and accuracy. Section 4.1.2 looks at selecting a quadrotor to perform SLAM. The AR Drone 2.0 was used as a proof of concept, and a custom-built quadrotor was later built. Section 4.1.3 examines the computers used for both onboard and offboard scenarios. Finally, temporally syncing multiple sensors is discussed in 4.1.4.
4.1.1 Camera Selection

A number of factors went into selecting the right camera, as it needs to be light enough to fly on a quadrotor, yet also accurate enough to yield high-quality 3D maps. Monocular cameras can be light and easily flown on quadrotors performing SLAM, as demonstrated by Engel, Sturm et al [25]. However, depth accuracy is diminished as triangulation between frames is the only way to obtain depth information, and the quality of images is subject to environment lighting. Stereo cameras are another option, and the presence of two cameras fortifies the triangulation necessary for determining depth. However, stereo cameras are also passive and thus subject to environment lighting. Furthermore, both mono and stereo cameras can’t estimate depth in areas with scarce features, such as a featureless wall.

Time-of-Flight (ToF) cameras are a type of camera that employ an active method to measure depth, using an illumination unit and image sensor. The illumination unit emits a light pulse, and the image sensor measures the phase shift to determine round-trip per-pixel time-of-flight, yielding a depth map. While ToF cameras can provide more accurate depth information than passive techniques, the hardware is often times too heavy for flight on a light vehicle. The Swiss RangeFinder [36] is an oft-used ToF camera for performing SLAM, but its weight of almost a half kilogram makes flight on a quadrotor difficult.

A newer lightweight ToF camera is PMD Technologies’ Camboard Nano [37], and is attractive for use on a quadrotor as it is one of the smallest ToF depth sensors available. However, upon testing the device, its range was found to be insufficient to map and navigate in unknown environments where the required range is greater than one meter (such as that required by typical rooms and hallways). Additionally, its lack of color means the 3D model produced won’t be in color. Furthermore, the lack of color prevents use of SLAM algorithms that rely on pixel color.

Another method of imaging that uses active sensing techniques to detect depth is structured light. These cameras project an IR pattern onto the environment, and use the stereo principle to triangulate and determine depth [39]. RGB-D cameras are a type of camera
that provide both color and depth information by combining a color sensor with a ToF or structured light camera. The ASUS Xtion Pro Live [40] is an RGB-D camera that uses structured light to determine depth. We found it had the best combination of weight and range of all considered cameras. Its resolution is also superior to the ToF cameras, and framerate and field of view is adequate. In light of this, the Xtion was used to conduct all tests. Table 4.1 outlines a comparison of the cameras that were evaluated.

<table>
<thead>
<tr>
<th>Camera</th>
<th>Dim. (m)</th>
<th>Weight (kg)</th>
<th>FOV (deg)</th>
<th>Res. (px)</th>
<th>FPS</th>
<th>Range (m)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SwissRanger</td>
<td>0.65 x 0.65 x 0.68</td>
<td>0.47</td>
<td>43.6 x 34.6</td>
<td>176 x 144</td>
<td>50</td>
<td>0.1-10</td>
</tr>
<tr>
<td>Nano</td>
<td>0.037 x 0.03 x 0.025</td>
<td>0.034</td>
<td>90x68</td>
<td>160x120</td>
<td>90</td>
<td>0.05-0.5</td>
</tr>
<tr>
<td>Xtion</td>
<td>0.18 x 0.035 x 0.05</td>
<td>0.217</td>
<td>58x45</td>
<td>640x480</td>
<td>30</td>
<td>0.8-3.5</td>
</tr>
</tbody>
</table>

Table 4.1: ToF and Structured Light Camera Comparison [37] [36] [40] Note: The Xtion can run at 60 FPS, but only with 320x240 resolution.

### 4.1.2 Quadrotor Selection

For the first iteration of research, we wanted to use a third-party quadrotor with existing flight software. When flying with a mounted Xtion, the quadrotor needs to support the Xtion’s weight of 217 grams. However, this number can be decreased to about 88 grams
when the Xtion’s external casing is removed and USB cable shortened. We chose the AR Drone 2.0 for the first iteration of research to make a proof-of-concept quadrotor that could perform SLAM. Its combination of price, ease-of-use, and ability to carry additional payload made it a clear choice. Additionally, an open-source autopilot system was already released for the system [25].

The AR Drone 2.0 comes equipped with many sensors detailed in Table 4.2. It also has an embedded computer onboard that allows the sensors to communicate with a ground station over Wi-Fi. The onboard computer also runs a filter with the sensor data, generating an attitude estimate for the quadrotor.

<table>
<thead>
<tr>
<th>Sensor</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gyroscope</td>
<td>0.1°/sec -rms noise</td>
</tr>
<tr>
<td>Accelerometer</td>
<td>+/-50mg precision</td>
</tr>
<tr>
<td>3 axis magnetometer</td>
<td>6° precision</td>
</tr>
<tr>
<td>Pressure Sensor</td>
<td>+/-10Pa precision</td>
</tr>
<tr>
<td>Ultrasound sensors</td>
<td>for ground altitude measurement</td>
</tr>
<tr>
<td>Vertical camera</td>
<td>60FPS at QVGA, for ground speed measurement</td>
</tr>
<tr>
<td>Front camera</td>
<td>720p at 30fps</td>
</tr>
</tbody>
</table>

Table 4.2: AR Drone 2.0 Onboard Sensors [41, 42]

The desire to perform SLAM onboard and simplify sensor fusion motivated building a custom quadrotor. We built a custom quadrotor that contains a laptop-grade computer, IMU, optical flow sensor, ultrasonic altimeter, and Xtion. Its MPU-6000 IMU consists of a 3D accelerometer and gyroscope, which it uses to measure linear acceleration and angular rates. Table 4.3 below outlines the sensors onboard the quadrotor.

4.1.3 Computing Platform

When choosing a computer to run the SLAM algorithms, we wanted to use a laptop-grade computer without a high-powered GPU. This grade of computer is more flexible than powerful GPU computers, since it can fly onboard a quadrotor. For an initial iteration we
<table>
<thead>
<tr>
<th>Sensor</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gyroscope</td>
<td>0.05°/sec -rms noise</td>
</tr>
<tr>
<td>Accelerometer</td>
<td>400µg/Hz noise</td>
</tr>
<tr>
<td>3 axis magnetometer</td>
<td>1° heading accuracy</td>
</tr>
<tr>
<td>Pressure Sensor</td>
<td>10cm resolution</td>
</tr>
<tr>
<td>Optic Flow Camera</td>
<td>VGA resolution at 250 Hz</td>
</tr>
<tr>
<td>Xtion RGB-D Camera</td>
<td>VGA depth and color at 30 Hz</td>
</tr>
</tbody>
</table>

Table 4.3: Custom Quadrotor Onboard Sensors [43–46]

decided to use a laptop-grade computer on the ground to process streamed images from the Xtion-mounted AR Drone, since the AR Drone doesn’t have the payload capacity for a medium-sized computer. With this approach we used an offboard laptop to run the SLAM algorithm and a small embedded-grade computer on the AR Drone to stream images from the onboard Xtion. As a second approach we used the custom quadrotor’s onboard computer to run the SLAM algorithms.

**Offboard Processing**

To run the SLAM algorithms offboard we chose the Macbook Pro with a 2.8GHz Intel Core 2 Duo CPU and 8 GB of RAM, running Ubuntu 12.04. Choosing the method to wirelessly stream data from the onboard Xtion to the laptop proved to be tricky. The method needed to stream both depth and color images, but be light enough to fly onboard the AR Drone. We first tested wireless USB adapters from Cables Unlimited, as the ability to stream Xtion data with a simple USB adapter would eliminate the need for an onboard computer. However, we quickly determined this method was unfeasible when we learned such devices do not provide isochronous support, which is required for the Xtion and other cameras.

Thus we needed an onboard computer that could stream images from the Xtion wirelessly to a ground station. Additionally, it has to be lightweight enough to be flown onboard the AR Drone, which is already near its payload limit when carrying the Xtion. We first
evaluated the Raspberry Pi computer. It has a small form factor and weighs 45 grams [47]. However, the CPU only runs at 700 MHz, and there is just 512MB of RAM. We determined that the Raspberry Pi is not powerful enough to run the OpenNI drivers necessary to read from the Xtion and wirelessly stream its data.

We next evaluated the ODROID-U2 computer. It has a similarly small form factor like the Raspberry Pi, but without its heat sink is even lighter at 33 grams, which is light enough to fly onboard the AR Drone. It also sports a 1.7 GHz quad-core processor and 2GB of RAM, and only requires 10 Watts of power. [48] This board was sufficient to run the required OpenNI drivers to read from the Xtion and stream its data over Wi-Fi, using an EDUP EP-MS8512 wireless adapter. The adapter is 802.11n and capable of streaming at 150 Mbps, which is sufficient to handle the Xtion’s streaming rate when the depth and color data is compressed [49].

![ODROID U2 Board](image)

Figure 4.2: ODROID U2 Board [48]

**Onboard Processing**

We evaluated the GIGABYTE BRIX computer to fly onboard the custom quadrotor and run SLAM. Model GB-XMI-3537 has a hyperthreaded dual core 2 GHz Intel i7 processor and
8GB of RAM [50]. Weighing only 171 grams with its casing removed, the GIGABYTE
BRIX is sufficiently light to fly onboard a custom-made quadrotor, and the i7 processor is
able to run all SLAM algorithms that were evaluated and developed. From testing, the com-
puter and camera combined consume 30 watts, of which the camera alone is responsible
for 2.5 watts.

4.1.4 Sensor Fusion

Using the AR Drone meant having access to its IMU and other sensors, in addition to the
Xtion flown onboard. We evaluated fusing these different sensors with the onboard Xtion
for use in a SLAM solution. First, we combined the AR Drone’s IMU with its mounted
Xtion. This approach involves two separate Wi-Fi links, as the IMU is streamed over the
AR Drone’s wireless link to the ground station, and the Xtion data is streamed with the
onboard ODROID to the ground station. It should be noted that the IMU data from the AR
Drone is a combination of its gyroscope and accelerometer sensors, which is fused in the
AR Drone’s onboard filter to produce yaw data.

Second, we fused the AR Drone’s IMU, optical flow sensor, and controller feedback
with the mounted Xtion all on a ground station. This approach also has two separate Wi-Fi
links. We use an open source autopilot package released by TUM that implements an EKF
with the aforementioned sensors [25]. TUM’s software wirelessly pings the AR Drone to
provide a constant estimate of timing offsets for the different sensors.

Finally, we fused an IMU on-board over hard-line with the Xtion, in contrast to the
previous wireless fusions. We tested this approach with the MPU-6000 IMU available on
the custom-build quadrotor and the onboard Xtion.
4.2 Results

Choosing the Xtion, AR Drone 2.0, and ODROID U2 computer culminated with a working system that could wirelessly stream the Xtion’s depth and color images to a groundstation running SLAM with the images. This system will be explained in detail in section 4.2.1. The custom-built quadrotor was additionally able to perform SLAM onboard, which will be explored in section 4.2.2. Finally, section 4.2.3 will evaluate fusing the Xtion with the IMU and other sensors over both wireless and hardwired connections.

4.2.1 Combining Hardware for Offboard SLAM

Streaming the Xtion depth and color data over Wi-Fi in real-time was a success. An OpenNI driver pulls data from the Xtion and streams it over the Wi-Fi network, all within the ROS framework. The Xtion depth and RGB data is stepped down to QVGA resolution and then compressed in order to be sent over Wi-Fi. With these modifications, the Xtion is able to stream depth and color images at 28Hz to the ground station while the quadrotor is flying. In comparison, an Xtion plugged in directly is able to run at 30Hz with VGA resolution. Figure 4.3 shows the block diagram of this system, and Figure 4.4 is a picture of the working system.

To demonstrate this system working, we modified an open-source SLAM algorithm to work with the wireless RGB-D data. The CCNY visual odometry and mapping algorithm is able to run in real-time (28 Hz) on the ground station with data wirelessly streamed from the AR Drone’s Xtion [8]. A 3D point cloud generated from this configuration is demonstrated in Figure 4.5. More work done to modify this algorithm will be discussed in Chapter 5.
Figure 4.3: System Design of AR Drone with mounted Xtion and ODROID running SLAM from RGB-D data on Ground Station.
Figure 4.4: AR Drone quadrotor with mounted Xtion and onboard ODROID board to wirelessly stream camera data to ground station.

Figure 4.5: 3D point cloud of office environment obtained from running CCNY algorithm by wirelessly streaming Xtion data from the AR Drone quadrotor.
4.2.2 Onboard SLAM

The custom-built quadrotor was able to fly with the Xtion, IMU, optic flow camera, and GIGABYTE BRIX computer. The onboard computer is able to run the CCNY algorithm at 30Hz with VGA resolution to perform SLAM in real-time, additionally using an IMU (to be discussed more in the next section). Figure 4.6 shows an image of the working system.

![Custom Quadrotor with Xtion, IMU, optic flow sensor, and GIGABYTE BRIX computer.](image)

Figure 4.6: Custom Quadrotor with Xtion, IMU, optic flow sensor, and GIGABYTE BRIX computer.

4.2.3 Sensor Fusion

Syncing a camera with other sensors is a challenging problem, as the sensors often run at different rates and even a slight timing offset can cause large errors. This is even more difficult when the camera and sensors are on separate Wi-Fi links, as is the case with the AR Drone’s sensors and the Xtion that both communicate with the ground station. That is, data from the AR Drone’s sensors are streamed on a different Wi-Fi connection than
the Xtion camera data. Through experimentation we discovered that timing issues abound with this setup, though we attempted several software workarounds that will be explained shortly.

Syncing the Xtion with the AR Drone’s IMU will first be evaluated, followed by the Xtion combined with TUM’s EKF software that uses multiple sensors on the AR Drone. Finally, fusing the Xtion with a hardwired IMU on the custom quadrotor will be assessed.

**Wireless Camera and IMU Fusion**

The AR Drone’s IMU reports data wirelessly at 200 Hz, while the camera runs at about 28 Hz over Wi-Fi. As can be seen in Figure 4.3, both messages transmit on separate Wi-Fi links. We wrote software to store camera and IMU messages in a queue. Each timestamped camera message is associated with the IMU message that has the closest timestamp.

To test temporally aligning the Xtion and AR Drone’s IMU messages, the CCNY algorithm was run on the ground station to process the streamed Xtion images and output a navigation pose. CCNY’s pose is timestamped with the time at which the ground station receives each Xtion message. The AR Drone’s IMU data is also timestamped when the ground station receives the message.

To evaluate temporal alignment, the IMU attitude data is compared to CCNY’s pose attitude estimate.

Figure 4.7 shows that the yaw generated from the camera lags behind the IMU’s yaw measurement by 100 ms or more. This delay is likely caused by that fact that camera messages are much larger than IMU messages, and thus take a longer time to arrive over Wi-Fi.

However, we were not able to characterize the camera and IMU data with a consistent offset. In Figure 4.7 it can be seen that the timing difference is not consistent, as the second peak has a camera-IMU delay of about 100 ms, while the third peak has a delay of almost 200 ms. The inconsistent timing delays with the IMU motivated evaluating fusing
a camera with multiple sensors in an Extended Kalman Filter that could smooth out timing differences.

**Wireless Fusion of Camera with TUM EKF**

We next evaluated syncing the data from TUM’s AR Drone EKF with the Xtion data. We used the same method of using a queue to match closest timestamps to associate EKF and camera messages. While the EKF runs at about the same rate as the camera (30Hz), messages sometimes come in at varying rates, so the queue smooths out variations and temporally aligns messages. Moreover, a time of validity (TOV) estimate is used to timestamp each sensor message from the AR Drone. The EKF generates an estimate for the Wi-Fi...
delay by pinging the AR Drone and subtracts the delay from the timestamp, which was generated when the ground station received the message. The result is that each timestamp for EKF messages accounts for Wi-Fi latency to better approximate the TOV.

We implemented generating a TOV for the Xtion. Rather than having the ground station timestamp the camera message when it was received, the ODROID onboard the AR Drone timestamps each Xtion message before sending it over the Wi-Fi link.

Figure 4.8 shows that the EKF and Xtion attitude data become much more closely-aligned when using this approach. However, their temporal difference is not consistent, as the EKF initially lags behind, but then the camera begins to lag.

Figure 4.8: Xtion-driven CCNY solution(red) and EKF (blue) yaw data, where the varying yaw demonstrates that Xtion and IMU measurements arrive inconsistently. Time is in seconds, and yaw is in radians.

A likely cause for these changing delays is inconsistent Wi-Fi latency that was not properly estimated by pinging the AR Drone. Another probable reason is clock skew. That is, the Xtion data is timestamped with a different computer than the AR Drone EKF data. A Network Time Protocol server was implemented on the ground station computer
to sync both computers’ clocks, as they were on the same local network. However, it was discovered that clock skew can still occur even when using an NTP server, since the NTP server is subject to varying Wi-Fi delays when used on a local wireless network. With this approach we could sync clocks within 10 ms with errors up to 25 ms, which is unsuitable for use with the algorithms since camera images are streamed every 33 ms.

Other possible reasons for the delays is that the Xtion connection to the onboard computer could experience USB latency. Another is that since a real-time system is not being used, timestamps for the Xtion and AR drone data can be delayed by the operating system scheduler.

We used the aforementioned TOV with an NTP server approach when fusing the IMU and Xtion, but it still did not resolve timing issues. We could not find a method that allowed fusing the IMU or EKF with the Xtion data over Wi-Fi closely enough to run the algorithms. The issues encountered with dual Wi-Fi links strongly motivated fusion with the CCNY algorithm and IMU using a hardwired camera and IMU.

**Wired Camera and IMU Fusion**

Fusing the MPU-6000 IMU with the Xtion over hardline was a success. Both sensors are connected to the custom quadrotor’s computer, which timestamps them upon arrival. The absence of wireless links significantly decreases timing discrepancies, and the IMU messages are able to align with the correct camera messages. A further detailed explanation of using the IMU with the Xtion to improve the CCNY algorithm is discussed in chapter five. Figure 4.9 shows the closely aligned camera and IMU yaw data, both timestamped with their TOV.
Figure 4.9: Xtion-driven CCNY solution (red) and IMU (blue) yaw data, where the varying yaw demonstrates that Xtion and IMU measurements are closely aligned. Time is in seconds, and yaw is in radians.
Chapter 5

Navigation Algorithm Development

Many different algorithms were evaluated and modified when developing the SLAM navigation solution. GPU-based SLAM algorithms are a hot research area as of late, with Kintinuous and others making impressive demonstrations [13]. We initially looked at the open-source GPU-based SLAM algorithm KinFu Large Scale [51]. While providing a 3D rendering of maps whose quality is unmatched by CPU-based algorithms, its dependence on a GPU made it unfavorable when compared to lighter algorithms. We could not find a system that both satisfied the custom quadrotor’s size and weight performance, and supported the CUDA parallel computing platform for GPUs [52]. Hence an entire class of SLAM algorithms requiring a GPU were eliminated.

We turned our attention to CPU-based SLAM algorithms that could run on the custom-built quadrotor’s onboard GIGABYTE BRIX computer. The open-source CCNY “Fast Visual Odometry and Mapping” algorithm met the criteria for being CPU-based while performing SLAM. Its lighter processing requirements, ability to run SLAM, and open-source availability made it the basis for developing a navigation algorithm. This algorithm uses an RGB-D sensor to perform SLAM, creating 3D maps [8]. Another open-source CPU-based algorithm we evaluated and used is TUM’s Dense Visual Odometry (DVO) [6]. It uses an RGB-D sensor to perform visual odometry, and likewise has light-enough processing requirements to be flown on the quadrotor’s computer.

Section 5.1 describes the approach of enhancing CCNY with an IMU. Section 5.2 gives
an overview of combining CCNY with DVO. Switching from CCNY to DVO when CCNY fails is explored in section 5.3. Then, the approach to weighing and combining both algorithms is discussed in section 5.4. Finally the approach of combining both of their cost functions to generate a combined navigation solution is analyzed in section 5.5.

5.1 Improving the CCNY Algorithm with IMU

As a first approach to improving upon the navigation solution, we enhanced CCNY by adding in IMU attitude estimates. As discussed in the previous chapter, temporally aligning the Xtion over Wi-Fi with the IMU and/or other sensors on a separate Wi-Fi link proved to be unfeasible. Timing issues with the wireless links motivated the approach of hardwiring the IMU and Xtion to the same computer. This approach successfully temporally aligned the Xtion and IMU messages, which the CCNY algorithm can use.

The CCNY algorithm performs ICP to match the current camera frame to a model. This computes a delta motion and generates a constantly updating navigation solution. The transform $T_{bf}$ keeps track of the accumulated transformations, representing the overall transformation from the base (camera) frame to the fixed (global) frame. It is updated by each new motion estimate from ICP that is represented as a delta transformation. The rotation update for $T_{bf}$ is represented as a quaternion:

$$q_{bf} \leftarrow q_{ICP} \times q_{bf}$$ (5.1)

where $q_{ICP}$ is the quaternion of the motion found from ICP. The translation update for $T_{bf}$ is

$$t_{bf} \leftarrow q_{ICP} \times t_{ICP} \times q_{ICP}^{-1} + t_{bf}$$ (5.2)

where $t_{bf}$ is $T_{bf}$’s translation and $t_{ICP}$ is the translation of the motion found from ICP.

We modified CCNY so the IMU’s attitude estimate updates the attitude estimate of the
navigation solution. Camera messages arrive at 30 Hz, while IMU message come in at 200 Hz. IMU messages arriving between camera frames are used to propagate \( q_{bf} \). This provides a rotation prediction for ICP.

To accomplish this, each IMU message’s x, y, and z angular velocities \( \omega \) are used to construct the quaternion \( q_\omega \) as

\[
\begin{bmatrix}
\omega \\
0
\end{bmatrix}
\]

As outlined in [53], the quaternion time derivative \( \dot{q} \) is calculated as

\[
\dot{q} = \frac{1}{2} q_\omega \times q_{bf}
\]

(5.3)

Where \( \dot{q} \) is the finite change from one quaternion to the next due to angular velocity. One step of the integration is then performed to update \( q_{bf} \):

\[
q_{bf} \leftarrow q_{bf} + \dot{q} \Delta t
\]

(5.4)

where \( \Delta t \) is set to 1/(rate) of the IMU, or .005 seconds. We only do the first order numerical integration since we are sampling so fast (200 Hz). The updated quaternion \( q_{bf} \) is then normalized:

\[
\hat{q}_{bf} = \frac{q_{bf}}{|q_{bf}|}
\]

(5.5)

Thus each new IMU message updates \( T_{bf} \)’s rotation, represented by \( q_{bf} \). This approach’s effectiveness will be evaluated in Chapter 6.

5.2 Integrating CCNY with DVO

In addition to exploring internal improvements to CCNY, the main work done to develop the navigation algorithm was combining different methods. CCNY uses ICP to match sparse features found in the current frame to a model of already-seen features, computing the motion required to align the current frame to the model [8]. The open-source DVO algorithm of Kerl, Strum et al. minimizes photometric and depth error between the current
and previous frame to compute the motion between frames [6]. Since two different algorithms are being utilized, combining their solutions could lead to more information and accuracy for the navigation solution than either one alone.

Furthermore, the algorithms tend to work better in different situations. CCNY performs better with larger datasets, while DVO has better accuracy for smaller datasets. CCNY works poorly on datasets with few depth features, where DVO tends to do better since it is not reliant on depth features (yet still does rely on some visual texture being present). Examples of this can be seen in chapter six.

For switching to work properly, both algorithms have to be converted to the same coordinate frame. Figure 5.1 shows the coordinate frame of each algorithm.

![Coordinate frames of the CCNY and DVO algorithms](image)

Figure 5.1: Coordinate frames of the CCNY and DVO algorithms

Since CCNY is used as the foundation for combining the algorithms, DVO must be converted into CCNY's coordinate frame. The conversion is calculated with

\[
R \times R_{DVO} = R_{CCNY}
\]

(5.6)

where \( R \in SO(3) \) is the rotation matrix necessary to convert DVO into the CCNY coordinate frame. Its value is found to be
\[ R = \begin{bmatrix} 0 & 0 & 1 \\ -1 & 0 & 0 \\ 0 & -1 & 0 \end{bmatrix} \] (5.7)

In order to combine the algorithms, we wrote software allowing the CCNY and DVO algorithms to run separately but at the same time, using the same camera color and depth image from the RGB-D camera. Both DVO and CCNY are run in the ROS framework to make data transfer seamless.

We decided to use CCNY’s SLAM framework as the foundation for all algorithm combinations. Combining the algorithms determines the delta motion computed, since the motion becomes a combination of that found by ICP and DVO. The features CCNY finds in the frame are updated by the computed combined motion, and added to the global model. For each incoming frame, ICP utilizes the model that is created by both algorithm’s solutions to generate a motion estimate, while DVO only looks at the previous frame to generate a motion estimate.

There are different ways to utilize both methods to compute the overall navigation algorithm, and three are considered in this work. The first is switching between the algorithms’ solutions when one fails. Second, weighing each algorithm’s solution and then combining them is evaluated. Finally, combining the cost function of each algorithm and minimizing the overall cost function to determine the final navigation solution is analyzed.

### 5.3 Switching between CCNY and DVO

The CCNY algorithm relies on ICP. However, we noted that ICP fails in scenes where there are very few depth features to extract, since there is no way to align the current (almost featureless) frame to the model. Sometimes when a few features are extracted, ICP
erroneously matches the current frame to the wrong place in the model, since the lack of features forces ICP to guess. An example of a scene where ICP would fail is a featureless hallway. These shortcomings of ICP result in a navigation solution that either can’t find enough correspondences to pass a preset threshold for ICP, or that experiences large jumps due to matching the current frame to the model incorrectly. The circled area in Figure 5.2 is where a dark hallway occurs in a dataset which should depict straight-line movement with several turns. ICP jumps around and fails to move forward in a straight fashion, highlighted by the close-up shown in figure 5.3.

Figure 5.2: ICP failing in a dark hallway (area with a circle drawn around it)
The DVO algorithm does not always fail in the same locations as the CCNY algorithm, since minimizing photometric error does not depend on extracting depth features. For instance, a wall could have no features, but if it has sufficient visual texture DVO would do a better job of obtaining a solution.

We established a metric so that when the number of correspondences generated by CCNY’s ICP falls below a certain threshold, the navigation algorithm switches to relying on DVO for that frame. This allows the algorithm to continue generating a solution even when ICP is not able to generate enough correspondences.

To use DVO’s solution within CCNY’s framework, DVO’s delta transform must first be transformed into CCNY’s coordinate frame. DVO’s delta transform consists of a translation and rotation. Using the rotation matrix $R$ found in 5.2 to convert DVO to CCNY’s coordinate frame, the translation conversion is

$$
t_{DVO'} = R \times t_{DVO}
$$

(5.8)
where \( t_{DVO'} \) is DVO’s newly converted translation.

The transform’s rotation is handled with quaternions, which can be converted from axis angle pairs. First, the new axis must be calculated:

\[
A_{DVO'} = R \times A_{DVO} \tag{5.9}
\]

where \( A_{DVO} \) is the axis from DVO. Using the same angle \( \theta_{DVO} \) from DVO, \( A_{DVO'} \) and \( \theta_{DVO} \) can be converted to form the new quaternion \( q_{DVO'} \):

\[
q_{DVO'} = (\cos(\frac{\theta_{DVO}}{2}), A_{DVO'} \sin(\frac{\theta_{DVO}}{2})) \tag{5.10}
\]

For the CCNY algorithm to use DVO’s converted delta transform, a few more adjustments must be made. As explained in 5.1, the transform \( T_{bf} \) keeps track of CCNY’s accumulated transformations. It is updated by each new motion estimate from ICP that is represented as a delta transformation. The rotation and translation updates in equations 5.1 and 5.2 are dependent on the fact that ICP runs in the global frame. That is, the current incoming camera features are first transformed into the global frame, whereupon ICP is run. The resultant calculated ICP motion is thus found in the global frame. Since the DVO delta transformation is found in the body frame, it must first be converted to the global frame before it can be used as a solution.

To update \( T_{bf} \) by DVO’s solution, the quaternion is

\[
q_{bf} \leftarrow q_{DVO'} \times q_{bf} \tag{5.11}
\]

This modifies the rotation of DVO by the rotation found in \( T_{bf} \), and then sets \( T_{bf} \)’s rotation to this updated rotation. Translation is updated as

\[
t_{bf} \leftarrow q_{bf} \times t_{DVO'} \times q_{bf}^{-1} + t_{bf} \tag{5.12}
\]
$T_{bf}$’s new translation is equal to rotating the DVO translation according to $T_{bf}$, and then adding $T_{bf}$’s translation to that. The results of using this approach will be examined in Chapter 6.

### 5.4 Weighing and Combining CCNY and DVO

In the next evaluated approach for using both the CCNY and DVO algorithm we weighed and combined them, in order to compute the averaged delta motion for each frame. We decided to use a calculation of uncertainty for each algorithm’s delta motion to weigh their motions. DVO already computes a covariance for each delta motion, but CCNY did not, so we wrote a custom method to estimate the uncertainty for ICP, calculated as

$$
\frac{1}{N^2} \sum_{i=1}^{N} (p_{m,i} - p_{F,i})^2 \tag{5.13}
$$

where $N$ is the number of points of interest, $p_{m,i}$ and $p_{F,i}$ are each 3D point in the model and current frame respectively.

Every point (or feature) in the current frame attempts to match with a model point during the Nearest Neighbor search in ICP. Equation 5.13 calculates the error between the points in the model and the points in the frame, after the frame’s points have been transformed according to the found motion. The points’ error is divided by the number of points to find the average error for each point, and divided again by the number of points so there is a negative correlation between number of points and the uncertainty. As the number of points increases, the uncertainty should decrease.

Several assumptions were made in this ad-hoc approach. First, the method only yields uncertainties for the computed transformation’s x, y, and z coordinates. The uncertainties for roll, pitch, and yaw should be roughly correlated to the uncertainties of the axes perpendicular to them. Thus in the CCNY coordinate frame, roll uncertainty is set to the average of y and z uncertainty, for example.
Another assumption is that the per-axis errors are solely respective to themselves, thus the uncertainties do not depend on each other. DVO’s uncertainty is a full 6x6 covariance matrix, so a third assumption is in using the ICP uncertainty to compare to DVO’s full covariance. Finally, DVO’s covariance is generated on a frame-to-frame basis, while CCNY’s uncertainty is generated from a frame-to-model method.

Since the uncertainties for CCNY and DVO were not generated in the same way or on the same scale, they can not be used directly to weigh their solutions when combining. A method advanced by Li et al. called Average Normalized Estimation Error Squared (ANEES) is used to average and normalize the uncertainties so they can be used to weigh the solutions [54].

The ANEES method is used to normalize the uncertainty values for x, y, z, roll, pitch, and yaw. With $x$ representing the pose, $M$ the number of measurements, and $P$ the uncertainty, ANEES is defined as:

$$\bar{\epsilon} = \frac{1}{M} \sum_{i=1}^{M} (x_i - \hat{x}_i)^T P_i^{-1} (x_i - \hat{x}_i)$$

(5.14)

where $\bar{\epsilon}$ is desired to equal 1. To obtain the ANEES, we needed a truth dataset to compare truth poses with estimated poses for both CCNY and DVO. The truth dataset needed to be similar to the environment the combined algorithm would run in, since it would characterize the ANEES values for all future tests. Sturm, Engelhard et al created a benchmarking dataset to aid in the evaluation of RGB-D SLAM algorithms [55]. They released a large set of image sequences from a Kinect along with time-synchronized ground truth data obtained from a motion capture system. Moreover, this dataset has many sequences that occur in office and hallway environments, which is this work’s initial target. Thus we chose the sequence ”Freiburg1 Room” where the camera moves around a room in an office environment.

Figures A.1 and A.2 show DVO’s delta x, y, z and roll, pitch, yaw pose versus ground truth, respectively. We calculated the ANEES $\bar{\epsilon}$ as shown in Equation 5.14 and multiplied
by $\hat{P}_{DVO}$, the covariances for x, y, z, roll, pitch, and yaw, obtaining the corrected covariance $P_{DVO}$:

$$P_{DVO} = \hat{P}_{DVO} \times \bar{\epsilon} \quad (5.15)$$

Figures 5.4 and 5.5 show that the scaled covariances are equal to the error.

Similar work was done with CCNY. Figures A.1 and A.2 show CCNY’s delta x, y, z and roll, pitch, yaw pose versus truth’s, respectively. The corrected uncertainty $P_{CCNY}$ is calculated from the ANEES $\bar{\epsilon}$ found in Equation 5.14 and the original uncertainty $\hat{P}_{CCNY}$:

$$P_{CCNY} = \hat{P}_{CCNY} \times \bar{\epsilon} \quad (5.16)$$

Figures 5.6 and 5.7 show the scaled uncertainties characterize the error.

The scaled uncertainty for each solution is used to weigh them when adding together. After DVO’s solution has been transformed into the CCNY coordinate frame as shown in the previous section, the solutions are combined according to:

$$\hat{x} = P_{CCNY}(P_{DVO} + P_{CCNY})^{-1}x_1 + P_{DVO}(P_{DVO} + P_{CCNY})^{-1}x_2 \quad (5.17)$$

where $\hat{x}$ is the new combined 6-Degrees-of-Freedom (DoF) solution (x, y, z, roll, pitch, yaw), $x_1$ and $x_2$ are the 6-DoF measurements for DVO and CCNY. As explained previously, $P_{DVO}$ is obtained from the DVO program, $P_{CCNY}$ is generated from equation 5.13 (which only populates the matrix’s diagonal; all other elements are set to 0). The solution can be simplified to run more efficiently:

$$\hat{x} = (I - K)x_1 + kx_2 \quad (5.18)$$

where $K = \frac{P_{DVO}}{P_{DVO} + P_{CCNY}}$. Thus $\hat{x}$ is computed for each new camera frame to provide a pose estimation. The effectiveness of this approach will be explored in Chapter 6.
Figure 5.4: DVO delta xyz error and scaled covariance

Figure 5.5: DVO delta rpy error and scaled covariance
Figure 5.6: CCNY delta xyz error and scaled uncertainty

Figure 5.7: CCNY delta rpy error and scaled uncertainty
5.5 Combining CCNY and DVO Cost Functions

In a final approach to utilizing both CCNY and DVO we combined their cost functions. We were motivated by similar work done in [56], where the cost function of a dense ICP algorithm was combined with the original DVO implementation (referred to as RGB-D). In this thesis we combine the cost function of sparse ICP (from CCNY) with the cost function of minimizing photometric error (from DVO).

DVO’s cost function seeks to minimize photometric error between the current and previous frame. As explained in [6], the algorithm solves for the camera motion since the previous frame, $\xi \in \mathbb{R}^{6 \times 1}$, with a weighted least squares solver. As shown previously in equation 2.3, each pixel coordinate $x$ is mapped from the current frame to the previous frame by the warping function $\tau(\xi, x)$ given the camera motion $\xi$. The residuals $r_i$ for the camera motion can be computed as:

$$r_i(\xi) = I_2(\tau(\xi, x)) - I_1(x_i)$$  \hspace{1cm} (5.19)

where $I_1$ and $I_2$ are two consecutive intensity images. The first order Taylor approximation of the residuals $r_i(\xi)$ is

$$r_i(\xi, x_i) = r(0, x_i) + \frac{\partial r(\tau(\xi, x_i))}{\partial \xi}_{\xi=0} \Delta \xi$$  \hspace{1cm} (5.20)

$$= r(0, x_i) + J_i \Delta \xi$$  \hspace{1cm} (5.21)

where $J_i \in \mathbb{R}^{1 \times 6}$ is the Jacobian of the i-th pixel with respect to the camera’s 6-DoF motion. After simplification explained in [6], they obtain the normal equations

$$J^TWJ\Delta \xi = -J^WR(0)$$  \hspace{1cm} (5.22)

where $J \in \mathbb{R}^{n \times 6}$ is the matrix of all $J_i$ pixel-wise Jacobians, and $W$ is the diagonal matrix of weights explained in [6]. The interested reader is invited to read appendix A.2 where the
calculation of the Jacobian is explained in more detail.

The original implementation of CCNY’s ICP relies on a two step-alignment procedure to minimize the cost, employing a singular value decomposition. This method does not build up a Jacobian or residual matrix. In order to easily combine the cost functions of both algorithms, we implemented a custom least squares solver in software to minimize cost in CCNY’s ICP so a Jacobian and residual matrix could be extracted. Given $F$ containing all 3D points of the current frame’s features, and $M$ containing all of the model’s associated 3D points, each point in $F$ can be mapped to $M$ with:

$$M_i = (I + S(\psi))\hat{R}F_i + \hat{t} + \Delta t$$

(5.23)

with $\hat{R} \in SO(3)$ rotation matrix, $\hat{t} \in \mathbb{R}^{3 \times 1}$ translation, and $\Delta t \in \mathbb{R}^{3 \times 1}$ incremental translation. $S(\psi) \in \mathbb{R}^{3 \times 1}$ is the skew symmetric matrix that is a first-order approximation of the $SO(3)$ rotation matrix equivalent to the axis-angle rotation $\psi$, and can be constructed as:

$$
\begin{bmatrix}
0 & -\psi_z & \psi_y \\
\psi_z & 0 & -\psi_x \\
-\psi_y & \psi_x & 0
\end{bmatrix}
$$

(5.24)

The residual for each point $r_i$ is

$$r_i = M_i - (I + S(\psi))\hat{R}F_i - (\hat{t} + \Delta t)$$

(5.25)

The i-th row of the Jacobian $J_i \in \mathbb{R}^{N \times 6}$ is found by taking the partial derivatives of $r_i$ with respect to $\Delta t$ and $S(\Psi)$:

$$J_i = \begin{bmatrix}
\frac{\partial r_i}{\partial \Delta t} & \frac{\partial S(\psi)}{\partial (\psi)}
\end{bmatrix}
$$

(5.26)

The camera motion $\xi \in \mathbb{R}^{6 \times 1}$ is the optimal solution to the linear system:

$$J^T J \xi = J^T r$$

(5.27)
Both CCNY and DVO’s cost functions can now be solved jointly by utilizing their Jacobians $J$ and residuals $r$ in one combined cost function. This is done by minimizing the least-squares problem by solving the normal equations

$$
\begin{bmatrix}
J_{ICP} \\
v J_{DVO}
\end{bmatrix}
\begin{bmatrix}
J_{ICP} \\
v J_{DVO}
\end{bmatrix} \xi =
\begin{bmatrix}
J_{ICP} \\
v J_{DVO}
\end{bmatrix}
\begin{bmatrix}
r_{ICP} \\
r_{DVO}
\end{bmatrix}
$$

(5.28)

where $v$ is a weight set to reflect the difference in metrics used for DVO and ICP costs. This can be simplified to

$$(J_{ICP}^T J_{ICP} + wJ_{DVO}^T J_{DVO})\xi = J_{ICP}^T r_{ICP} + v J_{DVO}^T r_{DVO}$$

(5.29)

where $w = v^2$.

In order to solve the cost for both algorithms jointly, all terms must be in the same coordinate frame. Since CCNY and DVO operate in different coordinate frames, $J_{DVO}$ and $r_{DVO}$ were transformed into CCNY’s coordinate frame per the method discussed in section 5.2. Another requirement is that both algorithms’ methods must run relative to the same frame of reference. CCNY’s ICP runs in the global frame since the current frame’s features are matched to a global model. On the other hand, each iteration of DVO runs in the body frame since it is a visual odometry method. Thus we modified CCNY as subsequently explained so ICP could run in the body frame, allowing $\xi$ in equation 5.29 to be solved properly.

To run ICP in the body frame, each of the model $M$’s 3D points must be transformed into the body frame by the global transformation $T_{bf}$:

$$
M'_i = R_{bf}^T M_i - R_{bf}^T t_{bf}
$$

(5.30)

where $R_{bf}$ is the transpose of $T_{bf}$’s rotation, $t_{bf}$ is $T_{bf}$’s translation, and $M'_i$ is each model’s points transformed into the body frame.
The features in the current frame can then be matched to $M'$ using ICP since the features are already in the body frame. ICP yields the transformation $T_{bb'}$ that transforms the new body frame $b'$ to the old body frame $b$. However, the update for global transformation $T_{bf}$ must be changed since ICP is no longer run in the global frame. Hence to find the updated global transformation $T_{b'f}$:

$$T_{b'f} = (T^{-1}_{bb'} \times T_{fb})^{-1}$$

(5.31)

where $T_{fb} = T_{bf}^{-1}$, and the inverse $T^{-1}_{bb'}$ is required since body ICP calculates motion from the current body frame to the old body frame, but we need motion to be from the old body frame to the new body frame.

Modifying both algorithms to run in the same coordinate frame and have the same frame of reference allows equation 5.29 to be solved. To aggregate the terms specified in equation 5.29, we meshed the software flows of CCNY and DVO together. DVO employs a coarse-to-fine pyramid scheme for its images, so the combined software flow allows the cost function to iterate within each of those levels [6].

Furthermore, each pyramid level only loops until one of three conditions is triggered—the error increases, the error only marginally improves, or the number of iterations surpasses a preset threshold. For each iteration, a nearest-neighbor search finds correspondences for the current image’s point cloud, and ICP is called to generate $J_{ICP}$ and $r_{ICP}$. Likewise, DVO is run with the current image to generate $J_{DVO}$ and $r_{DVO}$. After solving for the combined cost yields the delta transformation, the current point cloud and image are transformed by the delta motion so the solutions can be closer to converging on the next iteration. Finally, the overall transformation is updated by the found delta motion. Algorithm 2 belows shows the pseudocode:

The transformation found from the combined cost function is used to update CCNY’s model $M$ as explained in chapter 2; hence both algorithms are able to operate within CCNY’s SLAM framework.
Algorithm 1 Combined Cost Functions

Transform $finalT=\text{Identity}$
Transform $deltaT=\text{Identity}$
Error $e = 0$
Last Error $le = 0$
Minimum Error Threshold $minE$
$count = 0$
Max Count $mc$
Current Image Point Cloud $pc$
Current Image $im$

for each image pyramid level do
    while $e < le$ and $e - le > minE$ and $count < mc$ do
        Find ICP Nearest Neighbors
        ICP($pc$)
        DVO($im$)
        $deltaT =$Combined Cost(ICP, DVO)
        Transform $pc$ by $deltaT$
        Warp $im$ by $deltaT$
        $finalT = deltaT \times finalT$
        $count = count + 1$
        $le = e$
    end while
end for
Chapter 6

Experimental Results

This chapter examines the four different software approaches for improving the navigation solution, and culminates with an evaluation of the quadrotor running with the optimal navigation solution. We used six different datasets to evaluate the approaches. Since this work has an initial target of performing SLAM in office environments across multiple rooms, all datasets used were taken in office environments. We used three datasets from the RGB-D benchmark dataset discussed in Chapter 5 and released by [55], which shall hereon be referred to as the “Freiburg” datasets.

We discovered that the RGB-D benchmark dataset turns off the auto-exposure typically used with cameras, which improves the DVO algorithm. Since DVO computes motion by reducing photometric error, lighting changes between images (sometimes brought on by auto-exposure adjustments) can severely impact its accuracy. Thus DVO performs better on the RGB-D dataset than on datasets where the auto-exposure is enabled.

In light of this and in order to test our CCNY with IMU approach, we developed datasets involving a large loop in an office environment, capturing full RGB-D camera and MPU-6000 IMU data. These datasets, hereon referred to as the “Building” datasets, have auto-exposure enabled to see how the algorithms handle exposure changes, since a robust algorithm should be flexible to handle a camera automatically adjusting to lighting differences.

We will now overview the datasets used, beginning with Freiburg’s. “Freiburg1 Room”
involves rapid movement around a small room. “Freiburg2 Desk” involves slower movement around two desks in a typical office room, while “Freiburg3 Long Office” has movement around a space the size of several office rooms. We did not choose Freiburg datasets involving only minor movement or large movement (such as throughout a warehouse) since they are outside the scope of this work. The truth trajectories can be seen in Figures 6.1, 6.2 and 6.3. Table 6.1 outlines the distances and velocities of all datasets.

For all Building datasets we performed the same large rectangular loop in an office building. “Building QVGA” is a loop recorded at QVGA resolution to test the algorithms’ performance with lower resolution. For “Building Slow” we moved the camera slowly around the loop and its corners. For “Building Fast” we traversed the loop and its corners quickly. All tests involve varying lighting levels and a hallway with dim lighting, making them rather challenging. Figure 6.4 outlines the truth trajectory of all Building datasets, and Table 6.1 displays the distances and velocities of all datasets.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Distance (m)</th>
<th>Velocity (m/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Freiburg1 Room”</td>
<td>15.99</td>
<td>0.334</td>
</tr>
<tr>
<td>“Freiburg2 Desk”</td>
<td>18.88</td>
<td>0.193</td>
</tr>
<tr>
<td>“Freiburg3 Long Office”</td>
<td>21.45</td>
<td>0.249</td>
</tr>
<tr>
<td>“Building QVGA”</td>
<td>91.60</td>
<td>0.752</td>
</tr>
<tr>
<td>“Building Slow”</td>
<td>93.70</td>
<td>0.592</td>
</tr>
<tr>
<td>“Building Fast”</td>
<td>92.30</td>
<td>0.774</td>
</tr>
</tbody>
</table>

Table 6.1: Distances and velocities of all datasets used
Figure 6.1: “Freiburg1 Room” ground truth

Figure 6.2: “Freiburg2 Desk” ground truth
Figure 6.3: “Freiburg3 Long Office” ground truth

Figure 6.4: “Building” Datasets ground truth

Our switching approach involves changing from CCNY to DVO’s estimate when ICP can’t generate enough correspondences (from a lack of features). In testing, we could only
get ICP to fail on QVGA datasets, thus we could only evaluate the switching approach on dataset “Building QVGA”. “Building QVGA” involves segments where a lack of features occurs, causing ICP to fail, but has enough intensity changes to allow DVO to still work. Additionally, the IMU approach could only be evaluated on Building datasets since Freiburg datasets don’t provide full IMU data.

6.1 Testing and Evaluation Methods

For testing the accuracy of the different algorithm approaches on the Building datasets we needed ground truth. We used the AprilTag system to create an ad-hoc ground truth to which we could compare our algorithms’ performances [57]. AprilTag uses visual fiducials, which are landmarks a computer vision algorithm can easily recognize and distinguish from one another. For each AprilTag detected in the camera’s current frame, the AprilTag system provides distance and orientation from the detected tag to the current camera pose, and a timestamp of when it occurred. Thus concatenating this information over the duration of a test results in an accurate estimation for the camera trajectory, which we use as truth. The interested reader can view the necessary transformations to obtain global truth data from each AprilTag detection in Appendix B.1.

We placed 51 AprilTags around our office building testing environment, lining both sides of the hallways. After placing the AprilTags, we carefully surveyed their global orientations and positions using a laser range finder and measuring tape. Figure 6.5 below shows a sample AprilTag, and Figure 6.6 displays the tag locations in our test environment, with the estimate of camera truth shown for a complete loop test. While some slight noise can be seen in the trajectory, the AprilTag truth still captures the rectangular movement and perfectly closes the loop.
Figure 6.5: Sample AprilTag

Figure 6.6: AprilTag positions and estimated camera truth trajectory
6.1.1 Evaluation Metrics

We evaluated all the algorithms’ performances against the datasets with Absolute Trajectory Error (ATE). We first initially align the trajectories of the estimate and truth for the best overall fit. We then compute ATE as the Root Mean Square Error for all matching time indices between the estimate and the ground truth global translation:

\[ \sqrt{\frac{1}{n} \sum_{i=1}^{n} ||t_{Ti} - t_{Ei}||^2} \]  (6.1)

where \( t_{Ti} \) and \( t_{Ei} \) are the ground truth and estimate positions respectively at each matching time index, and \( n \) is the number of matching time indices for which the ground truth and estimate position can be compared. For evaluating the Freiburg dataset ATE, we used a tool provided by [55]; for our Building datasets, we implemented a custom ATE calculator. For our Building datasets we also provide an ATE metric where the truth and estimate trajectories are only initially aligned. This metric penalizes early failures which cause the resulting global trajectories to be far apart, but is of interest for use with a quadrotor since early failures are undesired.

As an additional metric to evaluate our algorithms on the Freiburg datasets we used Relative Pose Error (RPE), evaluating the relative 3D delta pose error of truth versus the estimate. [55] defines this as:

\[ \frac{1}{n-1} \sum_{i=1}^{n-1} ||(t_{Ti+1} - t_{Ti}) - (t_{Ei+1} - t_{Ei})|| \]  (6.2)

where \( i \) incrementally advances by one second, evaluated at every time step \( n \), since we evaluate the RPE of our algorithms in one second time intervals.
6.2 Evaluation of Algorithms on Freiburg Datasets

We will first examine the performance of CCNY, DVO, weighing and combining, and combined cost function approaches on the Freiburg datasets. As mentioned, switching and CCNY with an IMU could not be evaluated on these datasets.

6.2.1 “Freiburg1 Room”

The weighed and combined approach performed the best of all algorithms for dataset “Freiburg1 Room” for the primary metric of ATE. From this fast-paced small dataset, it appears DVO is better able to handle rapid movements on a smaller scale, since it performs much better than CCNY. The combined cost function approach outperformed CCNY, but was eclipsed by DVO and the weighed and combined approach. It should be noted that for all Freiburg datasets both algorithms’ costs received equal weight, as the $v$ term in equation 5.29 was set to 1. Adjusting the weights in either direction only made the solution worse, thus the optimal solution was to weigh both equally.

Table 6.2 shows the different approaches’ performances, and Figures 6.7 and 6.8 and show a plot of all approaches versus ground truth.

<table>
<thead>
<tr>
<th>Mode</th>
<th>ATE RMSE (m)</th>
<th>RPE RMSE (m)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCNY</td>
<td>1.046</td>
<td>0.522</td>
</tr>
<tr>
<td>DVO</td>
<td>0.398</td>
<td>0.519</td>
</tr>
<tr>
<td>Weighed and Combined</td>
<td>0.294</td>
<td>0.527</td>
</tr>
<tr>
<td>Combined Cost</td>
<td>0.404</td>
<td>0.562</td>
</tr>
</tbody>
</table>

Table 6.2: Evaluation of approaches on dataset “Freiburg1 Room”
Figure 6.7: CCNY and DVO algorithms for “Freiburg1 Room”

Figure 6.8: Weighed and Combined and Combined Cost algorithms for “Freiburg1 Room”
6.2.2 “Freiburg2 Desk”

The CCNY stand-alone algorithm performed best on dataset “Freiburg2 Desk”, and appears to be better than DVO at handling slower-movement datasets such as this one. The weighed and combined and combined cost function approaches both had less ATE error than DVO. Table 6.3 displays the different approaches’ performances, and all trajectories compared to truth can be seen in Figure 6.9.

<table>
<thead>
<tr>
<th>Mode</th>
<th>ATE RMSE (m)</th>
<th>RPE RMSE (m)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCNY</td>
<td>0.162</td>
<td>0.281</td>
</tr>
<tr>
<td>DVO</td>
<td>0.412</td>
<td>0.272</td>
</tr>
<tr>
<td>Weighed and Combined</td>
<td>0.169</td>
<td>0.281</td>
</tr>
<tr>
<td>Combined Cost</td>
<td>0.275</td>
<td>0.282</td>
</tr>
</tbody>
</table>

Table 6.3: Evaluation of approaches on dataset “Freiburg2 Desk”
6.2.3 “Freiburg3 Long Office”

CCNY once again performed the best on this slower-moving dataset for ATE. Combined cost outperformed DVO, while weighed and combined did the worst. Table 6.4 highlights the different approaches’ error metrics, and Figure 6.10 shows all trajectories versus truth.
<table>
<thead>
<tr>
<th>Mode</th>
<th>ATE RMSE (m)</th>
<th>RPE RMSE (m)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCNY</td>
<td>0.170</td>
<td>0.328</td>
</tr>
<tr>
<td>DVO</td>
<td>0.437</td>
<td>0.318</td>
</tr>
<tr>
<td>Weighed and Combined</td>
<td>0.652</td>
<td>0.381</td>
</tr>
<tr>
<td>Combined Cost</td>
<td>0.308</td>
<td>0.335</td>
</tr>
</tbody>
</table>

Table 6.4: Evaluation of approaches on dataset “Freiburg3 Long Office Room”

Figure 6.10: CCNY, DVO, Weighed and Combined, and Combined Cost algorithms for “Freiburg3 Long Office”
6.3 Evaluation of Algorithms on Building Datasets

All approaches were evaluated on Building datasets “Building QVGA”, “Building Slow”, and “Building Fast”.

6.3.1 “Building QVGA”

Evaluating switching between CCNY and DVO could only be performed on dataset “Building QVGA” since the lower resolution causes ICP to fail for lack of features in areas where DVO does not. Since these tests were conducted at QVGA, all algorithms performed worse than tests at VGA. It can be seen in Figure 6.11 that CCNY doesn’t finish its trajectory, and that DVO’s trajectory is horribly misshapen since it jumped out of plane. That is, DVO’s trajectory shot off in the +Z direction, while the ground truth had a stable height for the entire test. However, using the switching method has a much better trajectory, as seen in that same figure. From analyzing the best fit ATE, DVO actually performs very well, since its out-of-plane jump has been corrected for, as seen in Figure 6.13.

CCNY with an IMU performed best on this dataset for both initial and best fit ATE, followed closely by switching. Both the IMU and switching solutions for this dataset built a complete trajectory in plane. Due to DVO’s solution jumping out of plane, the weighed and combined and combined cost approaches performed poorly on this dataset. Table 6.5 shows the error metrics for the different approaches, and all estimate trajectories versus truth can be seen in Figures 6.11 and 6.12.

This dataset shows that while DVO has a good trajectory when fitted to truth, its plane jumps make it unacceptable for sole use on a quadrotor. As seen in the two remaining tests, combining cost functions with CCNY helps to smooth out both trajectories and leads to the best overall solution.
<table>
<thead>
<tr>
<th>Mode</th>
<th>Initial Fit ATE (m)</th>
<th>Best Fit ATE (m)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCNY</td>
<td>8.83</td>
<td>5.10</td>
</tr>
<tr>
<td>DVO</td>
<td>14.61</td>
<td>1.82</td>
</tr>
<tr>
<td>CCNY with IMU</td>
<td>7.62</td>
<td>1.64</td>
</tr>
<tr>
<td>Switching</td>
<td>8.77</td>
<td>2.34</td>
</tr>
<tr>
<td>Weighed and Combined</td>
<td>11.97</td>
<td>3.12</td>
</tr>
<tr>
<td>Combined Cost</td>
<td>49.83</td>
<td>7.45</td>
</tr>
</tbody>
</table>

Table 6.5: Evaluation of all approaches on "Building QVGA"

Figure 6.11: CCNY, DVO, and Switching algorithms for “Building QVGA”
Figure 6.12: CCNY with IMU, Weighed and Combined, and Combined Cost algorithms for “Building QVGA”
6.3.2 “Building Slow”

The combined cost method was best of all approaches for dataset “Building Slow”, yielding trajectories closely matching that of truth, as seen in Figure 6.15 below. A weight of $v = .9$ was optimal for the combined cost approach. On this and the other full resolution test “Building Fast”, CCNY experiences some scaling issues which DVO’s cost helps correct for, while CCNY’s cost helps smooth out the jumps in DVO’s solution. While performing the best of all algorithms, the combined cost approach couldn’t be evaluated in real-time due to computational constraints (experienced on all datasets).

CCNY with an IMU did not improve over the CCNY stand-alone solution. One likely reason is that since corners in this test were taken slowly, vision-only was sufficient to capture the turns. The weighed and combined approach performs poorly, largely due to jumps in DVO’s solution. Figure 6.14 shows truth, CCNY, DVO, and CCNY with IMU, while Figure 6.15 shows truth, weighed and combined, and combined cost approaches. Table 6.6 displays each approach and its error metrics.
<table>
<thead>
<tr>
<th>Mode</th>
<th>Initial Fit ATE (m)</th>
<th>Best Fit ATE (m)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCNY</td>
<td>8.78</td>
<td>1.61</td>
</tr>
<tr>
<td>DVO</td>
<td>6.70</td>
<td>2.81</td>
</tr>
<tr>
<td>CCNY with IMU</td>
<td>8.97</td>
<td>1.84</td>
</tr>
<tr>
<td>Weighed and Combined</td>
<td>17.11</td>
<td>3.82</td>
</tr>
<tr>
<td>Combined Cost</td>
<td>4.18</td>
<td>1.07</td>
</tr>
</tbody>
</table>

Table 6.6: Evaluation of all approaches on “Building Slow”

Figure 6.14: CCNY, DVO, and CCNY with IMU algorithms for “Building Slow”
6.3.3 “Building Fast”

The combined cost approach once again performed best on dataset “Building Fast”, where equal weights \( v = 1 \) for each algorithm’s cost was optimal. CCNY with an IMU outperformed CCNY, since on this faster dataset vision-alone couldn’t model the turns as well. The weighed and combined solution performed poorly due to the jumps in DVO’s solution. Figure 6.16 shows truth, CCNY, DVO, and CCNY with IMU, while Figure 6.17 shows truth, weighed and combined, and combined cost approaches. Table 6.7 displays each approach and its error metrics.
<table>
<thead>
<tr>
<th>Mode</th>
<th>Initial Fit ATE (m)</th>
<th>Best Fit ATE (m)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCNY</td>
<td>8.93</td>
<td>4.54</td>
</tr>
<tr>
<td>DVO</td>
<td>12.20</td>
<td>4.59</td>
</tr>
<tr>
<td>CCNY with IMU</td>
<td>7.61</td>
<td>3.53</td>
</tr>
<tr>
<td>Weighed and Combined</td>
<td>22.36</td>
<td>5.63</td>
</tr>
<tr>
<td>Combined Cost</td>
<td>6.35</td>
<td>1.58</td>
</tr>
</tbody>
</table>

Table 6.7: Evaluation of all approaches on “Building Fast”
Figure 6.17: Weighed and Combined and Combined Cost algorithms for “Building Fast”

6.4 Quadrotor with groundtruth

From the Building datasets, it appears that for real-time processing in areas with variable lighting, using CCNY with an IMU is the best approach. For use in variable lighting scenarios where real-time processing is not a requirement, the combined cost approach is optimal. We wanted to fly our quadrotor and process the navigation solution in real-time, so we chose the CCNY with IMU approach to do so.

We successfully flew the quadrotor in our building environment, flying down one hall about 9 meters, and returning back to the start. The navigation solution seen in Figure 6.18 was generated in real-time on the quadrotor’s GIGABYTE BRIX computer. Vibrations from flying the quadrotor caused the navigation solution to not be as smooth as other
datasets, but overall performance was still decent as it had an initial fit ATE of 1.78m. Figure 6.19 shows the 3D point cloud of a hallway generated from the flying quadrotor test.

Moreover, with this flight test we accomplished the end goal of allowing autonomy to be easily added to the quadrotor in the future. Figure 6.20 shows the occupancy grid map of the hallway generated from the quadrotor flight test. An occupancy grid map bridges the gap between SLAM and autonomous operations, as a path-planning algorithm can utilize the information to direct the quadrotor where to fly.

Figure 6.18: Flying quadrotor with trajectory estimated by CCNY with IMU in real-time
Figure 6.19: 3D point cloud of hallway generated from flying quadrotor test
Figure 6.20: 3D Occupancy Grid Map of office building hallway generated from the flying quadrotor test, created with software released by [8] and [17]
Chapter 7

Conclusion

7.1 Analysis of Results

This thesis presents a system that successfully integrates a navigation and mapping algorithm with a quadrotor, running in real-time onboard the quadrotor. Our working quadrotor system integrates an RGB-D camera and IMU, and generates a navigation trajectory onboard in real-time. It additionally is able to generate a 3D point cloud reconstruction of the environment, and as seen in Figure 6.20 can generate an occupancy grid map to be used for autonomous operations.

This thesis analyzed and improved upon different visual odometry and mapping algorithms. We discovered from the variable-lighting “Building” datasets that our CCNY with an IMU algorithm performs the best of all evaluated approaches for real-time processing when faster turns are used, which is what would be experienced on a quadrotor. Testing with slower turns shows that using the IMU doesn’t help the solution, but barely degrades CCNY’s stand-alone solution.

Our CCNY-DVO combined cost function algorithm works the best of all evaluated approaches when real-time processing is not required. As seen in Chapter 6, it helps mitigate the out-of-plane jumps that DVO’s solution sometimes experiences, while correcting for the scaling issues CCNY’s solution is susceptible to. While the weighed and combined approach worked well on a few Freiburg datasets, in variable lighting scenarios it appears
to be suboptimal due to the jumpy DVO solution. Switching from CCNY to DVO works fairly well when ICP fails, though its use here is limited to QVGA camera resolution since ICP is more likely to fail with QVGA resolution, and a failure at VGA resolution results in DVO failing as well.

7.2 Future Work

There are several areas of future work. Once the combined cost function algorithm is implemented on a GPU for real-time performance, one could design a quadrotor that can fly with the GPU onboard. Another area is to use dense ICP instead of sparse ICP; sparse ICP is used in this thesis to allow for real-time computing on a CPU. Dense ICP is generally more accurate, yet would require implementation on a GPU to run in real-time.

Yet another area of future work is utilizing more sensors. Our quadrotor currently flies with a working optical flow sensor whose information is not yet being used. The IMU, optical flow, and RGB-D camera data could be combined in an EKF framework for a more robust overall solution. Utilizing the IMU and optical flow in an EKF should alleviate any out-of-plane or scaling issues encountered from the vision solution.

Finally, allowing the quadrotor to operate autonomously is another area of future work. Our system can currently generate occupancy grid maps as seen above in Figure 6.19. Those occupancy grids can be used to plan an obstacle-free path to a goal, or provide an exploration algorithm information about unexplored frontiers.
7.2.1 Conclusion

This thesis evaluated several different hardware and software approaches to designing a working quadrotor system capable of performing navigation and mapping in real-time. We combined and enhanced several different navigation and mapping algorithms, and chose the optimal augmented algorithm to run in real-time onboard our custom quadrotor. Our quadrotor system can generate a navigation solution in real-time, and the generated point clouds and occupancy grid maps can be used to add autonomy in the future.
Appendix A

Approach

A.1 ANEES

Figure A.1: CCNY and DVO delta position versus truth for dataset "Freiburg1 Room"
A.2 DVO Jacobian

Building off residual equation 5.19, [28] states that the Jacobian $J_i(\xi)$ is calculated from $I_2(\tau(\xi, x_i))$ after applying the chain rule. $J_i(\xi)$ can be decomposed into a product of Jacobians as:

$$J_i(\xi) = J_J J_\tau$$  \hspace{1cm} (A.1)

$$= J_J J_\pi J_g J_G$$  \hspace{1cm} (A.2)
\[
\frac{\partial I_2(x)}{\partial \pi} \bigg|_{x=\pi(g(G(\xi), p_i))} \times \frac{\partial \pi(p)}{\partial g} \bigg|_{p=g(G(\xi), p_i)} \times \frac{\partial g(G, p)}{\partial G} \bigg|_{G=G(\xi), p=p_i} \times \frac{\partial G(\xi)}{\partial \xi} \quad (A.3)
\]

with \( p_i = (x, y, z)^T = \pi^{-1}(x_i, Z_1, (x_i)) \) as the 3D point corresponding to the \( i \)th pixel. \( J_I \in \mathbb{R}^{1 \times 2} \) is the image derivatives in the \( x \) and \( y \) direction. \( J_\pi \in \mathbb{R}^{2 \times 3} \) contains the derivatives of the projection function, respective to the point coordinates. \( J_g \in \mathbb{R}^{3 \times 12} \) is the Jacobian of the motion transformation with respect to its twelve parameters (ie rotation \( R \in \mathbb{R}^{3 \times 3} \) and translation \( t \in \mathbb{R}^{3 \times 1} \)). \( J_G \in \mathbb{R}^{12 \times 6} \) is the Jacobian of the translation \( t \in \mathbb{R}^{3 \times 1} \) and skew-symmetrical matrix \( S(\psi) \in \mathbb{R}^{3 \times 1} \). Therefore, the four Jacobians are defined as:

\[
J_I = \frac{\partial I_2(x)}{\partial \pi} \bigg|_{x=\pi(g(G(\xi), p_i))} = (\nabla I_{2,x} \ \nabla I_{2,y}) \quad (A.4)
\]

\[
J_\pi = \frac{\partial \pi(p)}{\partial g} \bigg|_{p=g(G(\xi), p_i)} = \begin{pmatrix} f_x \frac{1}{x'} & 0 & -f_x \frac{x'}{x'^2} \\ 0 & f_y \frac{1}{y'} & -f_y \frac{y'}{y'^2} \end{pmatrix} \quad (A.5)
\]

\[
J_g = \frac{\partial g(G, p)}{\partial G} \bigg|_{G=G(\xi), p=p_i} = \begin{pmatrix} x & 0 & 0 & y & 0 & 0 & 1 & 0 & 0 \\ 0 & x & 0 & 0 & y & 0 & 0 & 1 & 0 \\ 0 & 0 & x & 0 & 0 & y & 0 & 0 & 1 \end{pmatrix} \quad (A.6)
\]
\[ J_G = \frac{\partial G(\xi)}{\partial \xi} = \begin{pmatrix} 0 & 0 & 0 & 0 & r_{31} & -r_{21} \\ 0 & 0 & 0 & -r_{31} & 0 & r_{11} \\ 0 & 0 & 0 & r_{21} & -r_{11} & 0 \\ 0 & 0 & 0 & 0 & r_{32} & -r_{22} \\ 0 & 0 & 0 & -r_{32} & 0 & r_{12} \\ 0 & 0 & 0 & r_{22} & -r_{12} & 0 \\ 0 & 0 & 0 & 0 & r_{33} & -r_{23} \\ 0 & 0 & 0 & -r_{33} & 0 & r_{13} \\ 0 & 0 & 0 & r_{23} & -r_{13} & 0 \\ 1 & 0 & 0 & 0 & t_z & -t_y \\ 0 & 1 & 0 & -t_z & 0 & t_x \\ 0 & 0 & 1 & t_y & -t_x & 0 \end{pmatrix} \]  

(A.7)

with \( x', y', z' \) as the transformed 3D coordinates of \( p_i \). Finally the product of the image Jacobian \( J_I \) and warp Jacobian \( J_\tau \) is:

\[ J_I J_\tau = (\nabla I_{2,x} \nabla I_{2,y}) = \begin{pmatrix} \frac{f_x}{x'z^2} & 0 & -f_x \frac{x'y'}{z^2} & -f_x \frac{x'x'}{z^2} & f_x \left(1 + \frac{x'^2}{z^2}\right) & -f_y \frac{y'}{z^2} \\ 0 & \frac{f_y}{y'z^2} & -f_y \frac{y'y'}{z^2} & -f_y \left(1 + \frac{y'^2}{z^2}\right) & f_y \frac{x'x'}{z^2} & f_y \frac{x'}{z^2} \end{pmatrix} \]  

(A.8)
Appendix B

Results

B.1 AprilTag

Every AprilTag detection yields a rotation matrix $R_{C}^{Tag}$ of the tag’s orientation from the camera, and $t_{C}^{Tag}$, the tag’s position in the camera frame. The orientation of the camera to the truth coordinate frame can be found by:

$$R_{C}^{Truth} = R_{Tag}^{Truth} \times R_{C}^{Tag}$$

(B.1)

where $R_{Tag}^{Truth}$ is the rotation matrix of the tag to the truth coordinate frame. The current camera position $t_{C}^{Truth}$ in the global coordinate frame is thus:

$$t_{C}^{Truth} = t_{Tag}^{Truth} - R_{C}^{Truth} \times t_{Tag}^{C}$$

(B.2)

where $t_{Tag}^{Truth}$ is the surveyed global position of the tag.
Bibliography


