FPGA-Aided MAV Vision-Based Estimation

by

Dember Alexander Giraldez

S.B., E.E.C.S., M.I.T., 2010

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology

June 2011

Author .....
Department of Electrical Engineering and Computer Science
May 20, 2011

Certified by .....
Nicholas Roy
Associate Professor
Thesis Supervisor

Certified by .....
Richard Madison
Senior Member of Technical Staff, C. S. Draper Laboratory
Thesis Supervisor

Accepted by .....
Dr. Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee

FPGA-Aided MAV Vision-Based Estimation

by

Dember Alexander Giraldez

Submitted to the Department of Electrical Engineering and Computer Science
June 30, 2011
In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The process of estimating motion trajectory through an unknown environment from a monocular image sequence is one of the main challenges in Micro Air Vehicle (MAV) navigation. Today MAVs are becoming more and more prevalent in both civilian and military operations. However, with their reduction in size compared to traditional Unmanned Aircraft Vehicles (UAVs), the computational power and payload that can be carried onboard is limited. While there is ample research in motion estimation for systems that are deployed on the ground, have various sensors, as well as multiple cameras, a current challenge consists of deploying minimalistic systems suited specifically for MAVs.

This thesis presents a novel approach for six-degrees of freedom motion estimation using a monocular camera containing a Field-Programmable-Gate-Array (FPGA). Most implementations using a monocular camera onboard MAVs stream images to a ground station for processing. However, an FPGA can be programmed for feature extraction, so instead of sending raw images, information is encoded by the FPGA and only frame information, feature locations, and descriptors are transmitted. This onboard precomputation greatly reduces bandwidth usage and ground station processing. The objectives of this research are (1) to show how the raw computing power of an FPGA can be exploited in this application and (2) to evaluate the performance of such a system against a traditional monocular camera implementation. The underlying motivation is to bring MAV systems closer to complete autonomy, meaning all the computation needed for estimation and navigation is carried out autonomously and onboard.

Thesis Supervisor: Nicholas Roy
Title: Associate Professor

Thesis Supervisor: Richard Madison
Title: Senior Member of Technical Staff, C. S. Draper Laboratory

Acknowledgments

This thesis was prepared at The Charles Stark Draper Laboratory, Inc., under an Internal Research and Development program.

Publication of this thesis does not constitute approval by Draper of the findings or conclusions contained herein. It is published for the exchange and stimulation of ideas.

First I would like to thank my advisor, Nick Roy, for giving me the opportunity to work with him. I think he saw my enthusiasm for working in this area from the time I first approached him. What I have learned working with him far exceeds what I could have imagined when I first stepped in his office.

From Draper Lab, I would like thank my supervisor Richard Madison, Paul DeBitetto, Peak Xu, the rest of the Cognitive Robotics Group and Linda Fuhrman for the opportunity to work with them.

I would like to thank Javier Velez for his dedication to my project from day one, as well as Abe Bachrach, Daniel Maturana, Piotr Mitros, and the rest of the group for always listening and offering ideas. I would also like to thank Gim Hom for his support, especially in the early stages of this project.

I would like to thank my parents, Nora and Dember, and my brother, Diego, for their constant support, for always believing in me, and for all their phone calls to see how I was doing. They always made sure I never felt like I was too far away from home. I would also like to thank my friends for making MIT fun. I would like to thank Aubrey for her constant encouragement and for truly making my time at MIT one I will never forget. Aubrey –Thanks for always being there whenever MIT dealt me a character building blow and for making the simplest things so neat: from grabbing lunch from the food trucks to working until dawn to finish a class assignment –and then snowboarding at 8 a.m.!

Finally, I would like to dedicate this work to my aunt Ana, who passed away while I was at MIT. My aunt and uncle have helped my family tremendously and without her I would not have had the privilege of attending MIT.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Imagine a disaster area such as the one struck by hurricane Katrina in 2005. In a matter of hours, the landscape changed completely: previous maps of the area no longer applied, thousands were missing, and an extensive search and rescue operation began. It is well known that the U.S. has an extensive fleet of UAVs, such as the predator, but with a wingspan of 48 ft, these vehicles can only be used at high altitudes for general reconnaissance. It would be useful to be able to deploy MAVs, measuring just $12in \times 12in$, so that swarms of these vehicles could search the area, going into buildings looking for survivors while also mapping the new landscape. Making this capability a practical reality is the underlying motivation behind this thesis.

Currently, an obstacle to making this scenario a reality is the amount of sensors required onboard MAVs and the computational power that has to be deployed alongside for successful operation. The focus of this research is on the implementation of a system that uses a single sensor, a monocular camera, for position and velocity estimation of the MAV, along with an onboard Field Programmable Gate Array (FPGA) that harvests the potential of hardware to perform expensive computations onboard, reducing the amount of data that needs to be transmitted to a ground station. The FPGA performs feature extraction from images, which is a crucial, but computationally expensive part of autonomous navigation.

A goal of this thesis is to show that it is possible to have a lightweight FPGA onboard a MAV that performs feature detection on an incoming video stream and

outputs just feature information for processing at a ground station. This alleviates the need to send the whole image stream (which is expensive in terms of bandwidth and latency), and also takes care of the feature detection computation, which otherwise would be performed at the ground station. FPGAs continue to grow in capacity and decrease in cost, so this thesis also hopes to show how raw hardware can be properly designed to aid in computationally expensive processes. This thesis also shows the whole end-to-end process for autonomous position and velocity estimation, showing how FPGAs can be integrated successfully and quantifying their performance against traditional implementations.

1.1 FPGAs and Hardware Trends

Since their introduction in 1985, FPGAs have steadily gained popularity in academia and the electronics industry. An FPGA is basically an integrated circuit with thousands or millions of logic blocks that can be programmed by the user. Since FPGAs are reprogrammable, they have a huge advantage over Application Specific Integrated Circuits (ASICs), which can only be designed and manufactured once. Recent research has shown that FPGAs can provide significant performance increases in computing, especially in applications best suited for hardware acceleration such as those that use integer or fixed-point data types, as well as those that benefit from massive parallelism [6]. Feature detection, which was implemented on the FPGA, processes images entirely, so it greatly benefits from being able to parallelize this process. Also, the fact that many feature detectors operate on grayscale images that consist of integer pixel values between 0 and 255 provides yet another reason for experimenting with FPGAs in this domain.

FPGAs consume significantly less power than even low-voltage consumer grade CPUs. To get an idea of power consumption, at full capacity the Spartan3e FPGA (used in this project) consumes 0.257 Watts [26]. In comparison, a low-voltage Intel Pentium M processor consumes 12 Watts [7]. On a MAV, power is very limited since it comes from batteries that are part of the payload, so this was yet another motivation

for exploring the use of FPGAs.

Finally, there is another trend that motivated the use of FPGAs. FPGAs consist of many logic gates that can be programmed to be "wired together" (as if on a breadboard to illustrate the simple case). In parallel to other technology trends, the number of gates per chip has greatly increased over the last few years while the cost per gate has decreased [18]. In the near future, FPGAs are expected to expand and offer even more computational resources at higher densities. That is another reason that motivated this project, since the best way to prepare is to exhaust the capabilities of today's FPGAs. This thesis shows that some algorithms can be successfully ported to hardware, but as FPGAs evolve, the complexity of algorithms that can be implemented in hardware will increase as well. Section 7.1 discusses the viability of implementing more complex feature detectors in hardware such as SIFT [20]. One of the goals of this thesis is to offer a glimpse into that future application.

1.2 Thesis Organization

This thesis is organized as follows: Chapter 2 provides a brief overview of the current state of this particular area of research and highlights some previous work that is directly related to the work done for this thesis. Chapter 3 is an overview of the design for the whole system: It describes design considerations in hardware and also covers the main concepts behind the implementation of feature detection and feature matching. Then, it provides an overview of motion estimation by describing the necessary components. Chapter 4 describes the implementation of the system, discussing design decisions and describing the major components. Chapter 5 contains the evaluation of the system. It includes a description of the setup used to establish baseline performance and also the process of obtaining truth measurements for comparison. Chapter 6 summarizes the main findings of this thesis and Chapter 7 provides ideas and considerations for further research.

Chapter 2

Related Work

2.1 Autonomous Flight in Unknown Environments

A system for autonomous flight through unknown indoor environments was demonstrated successfully by Bachrach, He and Roy [2]. The work consisted of outfitting a quadrotor helicopter with a Hokuyo laser rangefinder, laser deflecting mirrors for sensing altitude, a monocular camera, an inertial measurement unit (IMU), a Gumstix processor, and the helicopter's internal processor. The system used an EKF for data fusion and a high-level SLAM implementation. Successful results were obtained consisting of autonomous exploration of many different indoor environments. This work served as motivation for the work in this thesis, as it showcased all the sensors needed for successful autonomous navigation and the extensive amount of computation required at each time step. This led to the exploration of using different hardware to perform onboard computations and the idea of adding an onboard FPGA was born. This thesis takes this idea even further, in that it attempts to use a monocular camera as the only onboard sensor and a FPGA to aid in some of the heavy computation, more specifically, feature detection.

2.2 SLAM using MAVs

Simultaneous Localization And Mapping (SLAM) is a technique that allows a robot to build a map of an unknown environment, while simultaneously estimating the robot's location in the map using information gathered from its environment [8]. SLAM has been a challenge in MAVs due to the limitations in sensors that can be put onboard. On ground vehicles, SLAM is very successful due to the odometry information ground vehicles are able to obtain [2]. However, these odometry sensors, for example mid to long range laser rangefinders, cannot always be placed on a MAV due to payload restrictions. Another consideration is that extra instrumentation onboard can affect the vehicle's dynamics, further stressing the vehicle's control and stabilization system. For air vehicles, an alternative is to implement SLAM using stereo cameras or even a single monocular camera as the sole sensor.

One of the disadvantages of SLAM is that it is very computationally intensive, and therefore it is generally not an option to perform SLAM onboard the vehicle. Currently, the computation normally occurs at a ground station and the results, which consist of the map and vehicle position estimate, are transmitted back to the vehicle. However, transmission bandwidth tends to be an issue, especially if the vehicle is transmitting live video or a stream of high-resolution images.

Monocular SLAM

An important consideration is the case of Monocular SLAM. In traditional SLAM, different types of sensors such as laser range finders or sonar can be used to obtain 3D information about the environment. These types of sensors provide both range and bearing measures for each measured point, providing extensive information to reconstruct the environment's geometry. However, in the realm of MAVs, where payload and resources are limited, it is not always practical to have so many sensors onboard. The SLAM problem needs to be solved using a single camera.

2.3 Previous Work at Draper

The work for this thesis took place at The Robust Robotics Group in MIT's Computer Science and Artificial Intelligence Laboratory (CSAIL) and at The Cognitive Robotics Group in the Charles Stark Draper Laboratory. At Draper, the work builds on existing research described in [22]. The paper describes an implementation of a UAV navigation system for flying autonomously when GPS signal is lost. The system implements an EKF to fuse information from the IMU, as well as information from feature detection, in order to create a six degrees of freedom navigation solution. For feature detection the implementation uses Lukas-Kanade [21] features. For navigation, four features are used by the filter and between 16-20 are kept in reserve. This is because finding 1 or 20 features costs the same, so it does not cost more to calculate extra features to keep, as opposed to having to dynamically find a new one when a tracked feature is lost.

Chapter 3

Design

The main problem this thesis is trying to solve is position and velocity estimation of a MAV. This chapter describes all the steps necessary to go from a sequence of images to the position and velocity estimate. Since an important aspect of this thesis is the application of hardware acceleration, there are both hardware and software components. This chapter provides an overview of the system, with major components shown in Figure 3-1 (Page 22). Then this chapter discusses the hardware and how the Elphel Smart Camera [11] was chosen as the main hardware component. Section 3.3 explains the reason for feature detection and provides an overview of the ideas behind it. Section 3.4 explains the Harris corner detector and provides the mathematical basis behind it. Section 3.5 explains the process of position and velocity estimation using the EKF and the incorporation of observations obtained from one frame to the next. Although the EKF is described mathematically, the derivation is not provided in this thesis. [25] provides thorough derivations of both the Kalman filter and the EKF. Finally, Section 3.6 provides the results from implementing a simple 3 d.o.f. EKF, which was used implemented as a first step towards the 6 d.o.f. implementation.

3.1 System Overview

The system diagram in Figure 3-1 (Page 22) provides an abstraction of the system as a whole. The following presents a description of each component:

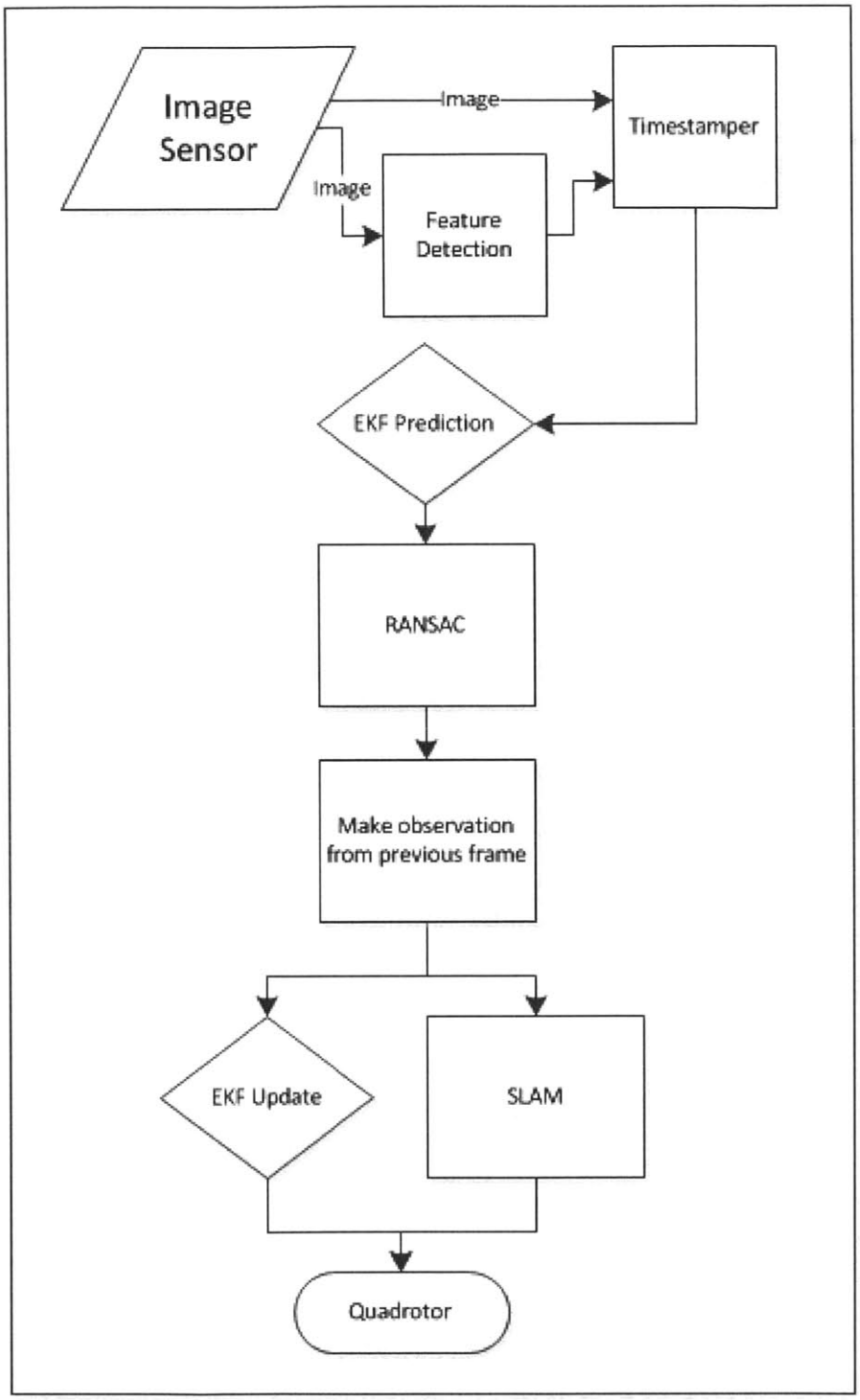

Figure 3-1: System Diagram.

- **Image Sensor:** The image sensor captures the images to be processed using a 5Mpix Aptina sensor.

- **Feature Detection:** The camera extracts features using the FPGA. It outputs feature locations and the corresponding descriptors. Features (a set of points in the image) are important because they will be used to figure out transformations from one frame to the next. The transformation is important because it is essentially made up of movements, whose estimation is a goal.

- **Timestamper:** The timestamper keeps track of which frame features correspond to and also applies a timestamp that is unique to a set of features. The timestamp is important for comparison against other measurements, for example those from a laser rangefinder. It can also facilitate the velocity calculation, since one can compute the change in position over time.

- **EKF Prediction:** At this stage the filter (EKF) makes a prediction of the state estimate and of the covariance. The state of the camera consists of position, orientation and respective velocities. Feature information is also added to the state vector. The filter gives a prediction, or belief, of the camera state. The objective of the next couple of steps is to refine this belief.

- **RANSAC:** To generate the transformation describing motion from one frame to the next, a model is tested against all the correspondences, or matches between features. However, normally there are too many inaccurate feature correspondences to simply compute the transformation using all the correspondences. RANSAC [13] allows testing of different transformations in order to find one that minimizes the number of outliers present in the transformation model.

- **Make Observation:** The observation consist of recording the new location of a previously tracked feature. This step relies on available features, and can initialize/delete features as necessary.

- **EKF Update:** The filter uses the observation to update its prediction, resulting in a more accurate state and covariance estimate. The state estimate

contains the updated position and velocity estimate, which is the goal of each iteration and the goal of the system.

- **SLAM:** This module keeps track of the map with observed features and also keeps track of the trajectory of the camera/MAV by using the measurements and its model for camera motion.

3.2 Hardware

One of the challenges of working with hardware is that it is very difficult to choose the ideal device. With FPGAs, there are state-of-the-art devices that offer millions of gates, plenty of onboard memory, and many extra components (USB connectivity, video I/O, Ethernet port(s), audio I/O, LCD screens etc.). However, with these offerings, size and weight of the board increase, as well as the cost and power consumption of the board. Since this research requires a piece of hardware that can be mounted onboard a MAV, size, weight, and total power consumption were key considerations. Another consideration was whether open-source tools could be used to develop, program and simulate the FPGA.

For the first prototype, Digilent's Nexys 2 FPGA board (Figure 3-2 was used. This board had a Xilinx Spartan3e FPGA along with a VDEC1 video decoder for NTSC analog video. This board could be programmed using Xilinx's free Webpack IDE, and had reasonable weight, size, and power requirements. However, two main difficulties surfaced using this board: One was integrating the analog video decoder with the rest of the system and the second was its memory I/O capability. In asynchronous mode, the fastest speed for the memory controller was $12.5MHz$. However, to run at 640×480 resolution refreshing at $60Hz$, the pixel clock (the rate at which each individual pixel is updated) had to be at least $25.175MHz$. Therefore it turned out the memory was not fast enough to keep up with all the pixels going in. After encountering these issues, the need for a different FPGA package was determined.

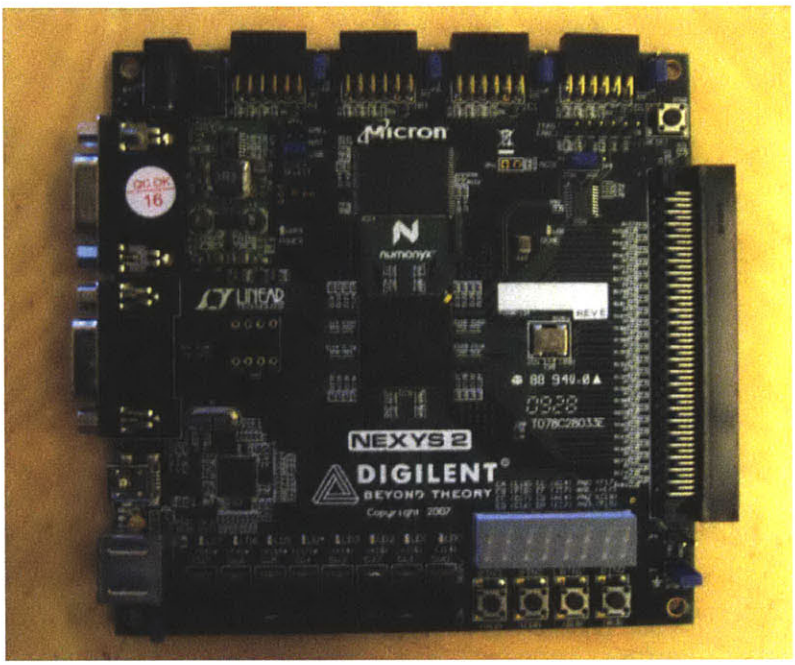

Figure 3-2: Nexys 2 board from Digilent, measuring approximately $5in \times 5in$.

3.2.1 Elphel Smart Camera

After considering many different options with FPGAs, the group that stood out consisted of "smart cameras". Smart camera is a general term for a camera that offers the functionality of a regular digital camera, as well as some other components inside it, such as flash memory, an FPGA or even a CPU. Many smart camera manufactures target industrial customers and work with clients to build custom-cameras. The challenge consisted of finding a camera that provided open access to its components. After speaking with several vendors, the camera that offered the best set of features was the Elphel-353 Smart Camera [11], which had a Xilinx Spartan3e FPGA with 1200K gates, an Axis ETRAX FS CPU, 64MB of system memory, 64MB of image memory, and an Aptina 5MPix CMOS sensor. Furthermore, Elphel provided open access to both the FPGA code and the CPU code, giving the user absolute control over the camera's functions. The open source code, plus the digital sensor, made this a very attractive choice.

The only drawback about this camera is that it uses a rolling-shutter, as opposed to a global-shutter, which is the preferred capture method used in robotics applications.

The problem with the rolling-shutter is that image acquisition happens sequentially: the image is scanned starting from the upper-left pixel, traversing the picture from left to right and top to bottom until the lower-right pixel is captured. The result is that not all parts of the image are captured at once. This poses a problem when there is motion involved, since scenes can change while the image is being captured. The resulting images can show bends where they do not exist, or objects wobbling for example. Figure 3-3 (Page 26) shows this effect:

Figure 3-3: Green lines highlight the rolling-shutter effect. The vertical lines on the cars are tilted in opposite directions, when they should be parallel to each other (Elphel).

A global shutter captures a whole image at the exact same instant by controlling all the photo sensors at once. The problem with global shutters is that they require wide data paths and high transfer rates in the underlying components, since all the pixel information for a frame is acquired instantly. However, at some point in the future, there should be a suitable global shutter for this camera and the rest of the system is expected to work with the updated shutter. After evaluating the specifications, the benefits of the camera outweighed the rolling shutter drawback, and implementation proceeded using the Elphel-353.

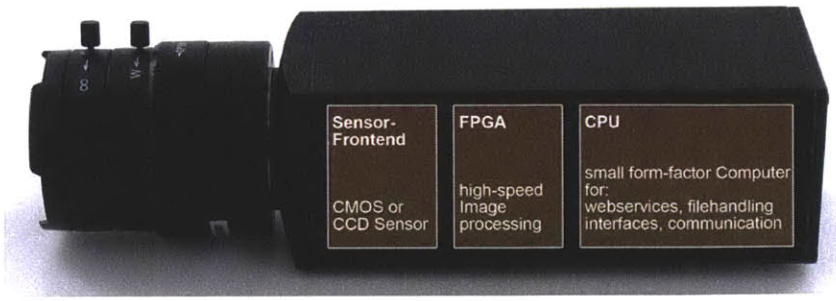

Figure 3-4: Elphel 353 Smart Camera (Elphel)

3.3 Feature Matching

The basis for autonomous navigation of any vehicle consists of understanding its environment, since aside from controlling the vehicle's dynamics, essentially a path has to be planned and obstacles need to be avoided. One way to "understand" the environment is to recognize certain points or *features* of an image from one frame or scene to a subsequent one. These known features provide an anchor around which motion estimation can be done. The following are the three main components that provide useful information from one frame to the next:

1. **Detector:** Attempts to independently find the same set of points (features) from one frame to the next.

2. **Descriptor:** Encoding for a given feature that is used as a basis for comparison. It can consist for example of an encoding of the local neighboring window of pixels. It can also include the expected location of the feature in the next frame in order to narrow down the search space for correspondences.

3. **Correspondence:** Based on the descriptor for one image, attempts to find the most similar descriptor in a subsequent image, possibly resulting in a match.

3.4 Harris Corner Detector

For feature detection in hardware, I implemented the Harris Corner Detector [16]. The Harris corner detector is an established feature detection algorithm that is highly

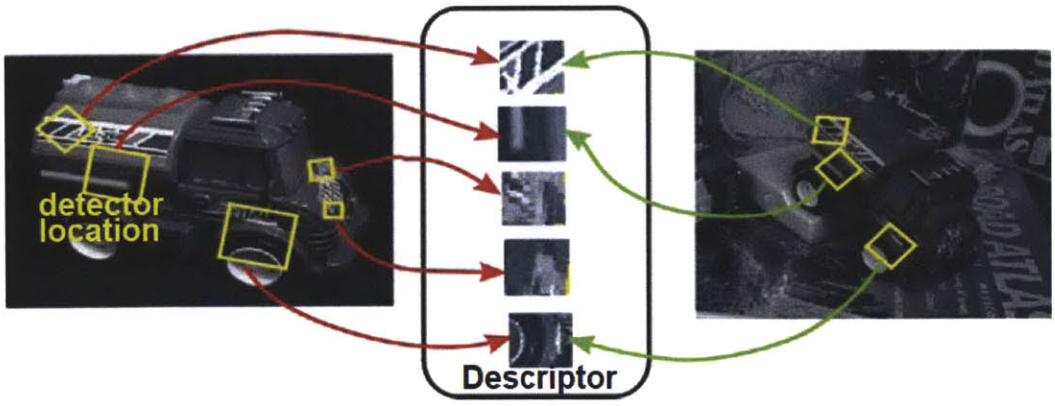

Figure 3-5: Detector, Descriptor and Correspondences (6.865 Course Slides)[15].

parallelizable and can be implemented using integer arithmetic (two requirements for successful FPGA implementations). The Harris corner detector essentially works by comparing adjacent gradients (in this domain gradients consist of the difference from one pixel to the surrounding eight) throughout the image. The basic idea behind the corner detector consists of the following three scenarios:

1. **Flat:** No change in pixel intensity along any one direction.

2. **Edge:** Change in pixel intensity along any one direction.

3. **Corner:** Change in pixel intensity along two opposing directions.

3.4.1 Mathematics

In a more rigorous way, the Harris corner detector can be interpreted as a window-averaged (over window w) change of intensity (I_x and I_y) obtained by shifting the image pixels by $[u, v]$, illustrated in Equation 3.1:

$$E(u,v) = \sum_{x,y} w(x,y) \cdot [I(x+u, y+v) - I(x,y)]^2 \tag{3.1}$$

A Taylor series approximation then gives a quadratic form for error as a function of image shifts:

$$E(u,v) \approx \sum_{x,y} w(x,y) \cdot [I(x,y) + uI_x + uI_y - I(x,y)]^2 \tag{3.2}$$

$$= \sum_{x,y} w(x,y) \cdot [uI_x + vI_y]^2$$

$$= (u\ v) \cdot \sum_{x,y} w(x,y) \cdot \begin{bmatrix} I_xI_x & I_xI_y \\ I_xI_y & I_yI_y \end{bmatrix} \cdot \begin{pmatrix} u \\ v \end{pmatrix}$$

$$\cong [u,v] \cdot M \cdot \begin{bmatrix} u \\ v \end{bmatrix} \tag{3.3}$$

where:

$$M = \sum_{x,y} w(x,y) \begin{bmatrix} I_xI_x & I_xI_y \\ I_xI_y & I_yI_y \end{bmatrix} \tag{3.4}$$

Matrix (3.4) is often called the structure tensor or in this particular context, the Harris Matrix.

Carrying out the eigenvalue analysis of this matrix results in the intensity change of the shifting window. This yields a measure of the corner response:

$$R = det(M) - k \cdot trace^2(M) \tag{3.5}$$

$$= \lambda_1\lambda_2 - k \cdot (\lambda_1 + \lambda_2)^2$$

where k is an empirical constant, usually in the range $0.04 - 0.06$[15].

The response, R, can tell us what we need to know about the particular pixel:

1. **If R is large:** Found a corner.

2. **If R is negative with large magnitude:** Found an edge.

3. **If $|R|$ is small:** Found flat region (no feature).

3.5 Estimation

One of the main goals of this project is to integrate feature detection in hardware with state (position and velocity) estimation using a single camera. So far the framework for feature detection has been described. Next is the framework for state estimation from a monocular image sequence.

3.5.1 Correspondences

The camera provides feature locations and descriptors for each frame. In essence, what happens next is illustrated in Figure 3-6.

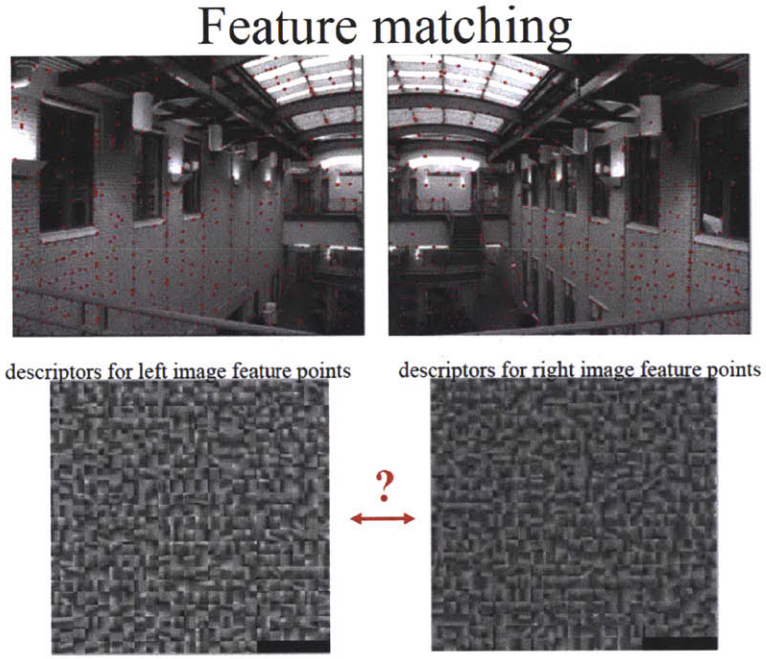

Figure 3-6: Features and descriptors for two different frames. Detected points are shown in the first pair of images. The second pair of images contains intensity patches corresponding to the detected features. Intensity patches between images can be compared to find correspondences between the two images [15].

For each feature point, the most similar point in the other image is found based on the the feature descriptor, which is the $n \times n$ intensity patch around the feature. Bad matches, such as those caused by ambiguous features where there are too many similar points, should be rejected. Rejection happens using a threshold. For similarity

to be considered a match, there cannot be too many points below the predetermined threshold. To find the best matches, there are some options [15]:

1. **Exhaustive search:** For each feature in image 1, compare to *all* the features in image 2.

2. **Hashing:** Compute a short descriptor from each feature vector (useful when larger feature descriptors are used).

3. **Nearest Neighbor Techniques:** K-tree and their variants.

This implementation uses a multihypothesis approach, since chances are a feature in frame $n+1$ will show up near its previous location in frame n. However, this is not necessary to match all the features, but only those for which there is high confidence of a match. Features are compared using a threshold on their sum of squared differences (SSD). The following is computed:

$$SSD(Patch1, Patch2) < th \tag{3.6}$$

for all features and the result consists of all the matches. The threshold in this case is set experimentally.

Transformation T

At the image level, given good matches, we need to look for possible transformations of the points from one frame to the other. The transformation is a key component in estimating motion from one frame to the next, since it models how the image changed. This transformation consists of translation, scaling and the affine changes the image could have undergone from one frame to the next. We want to compute T' such that $x_{frame1}T' = x_{frame2}$, where x consists of the set of points for the specified frame. A problem is the difficulty of matching a transformation model to all the correspondences between images. Since correspondences are found using a threshold, there is an embedded tolerance for error, so a perfect transformation is highly unlikely. Ideally, a transformation would be found using only perfect correspondences,

or at least those that produce the best transformation model. RANSAC, a sampling technique, can be used to provide the next best thing: a transformation using only the best correspondences.

RANSAC

Random Sampling Concensus[13] is a well-established technique that can be used to estimate the transformation in a model that contains points that fit the model (inliers) and points that do not fit the model (outliers). Ideally, the features found from one frame to the next would model the transformation exactly. In practice however, there will be many candidates for the *best* transformation. The *best* transformation is the one that minimizes the number of outliers from one frame to the next, providing a best fit for the model. The implementation of RANSAC in this project works as follows:

Algorithm 1 RANSAC implementation

while $T\ not\ found$ **do**
  $Select\ four\ feature\ points\ at\ random$
  $Compute\ transformation\ T$
  $Compute\ inliers\ where\ \parallel p_i', Tp_i \parallel < \varepsilon$
end while
$Keep\ largest\ set\ of\ inliers$
$Recompute\ least-squares\ T\ estimate\ on\ all\ of\ the\ inliers$

Figure 3-7 (Page 33) shows results obtained after implementing Algorithm 1 in Matlab.

3.5.2 Camera Dynamic Model

Since the goal of this thesis is position and velocity estimation, there needs to be a model of the object's motion, in this case the camera (which is the same as the MAVs dynamic model, since they are rigidly attached). Research has shown that for the case of pure visual estimation from a monocular image sequence, a constant velocity model provides enough information for smooth hand-held motion[10]. The model works as follows: Camera state is composed of a position r, an orientation q,

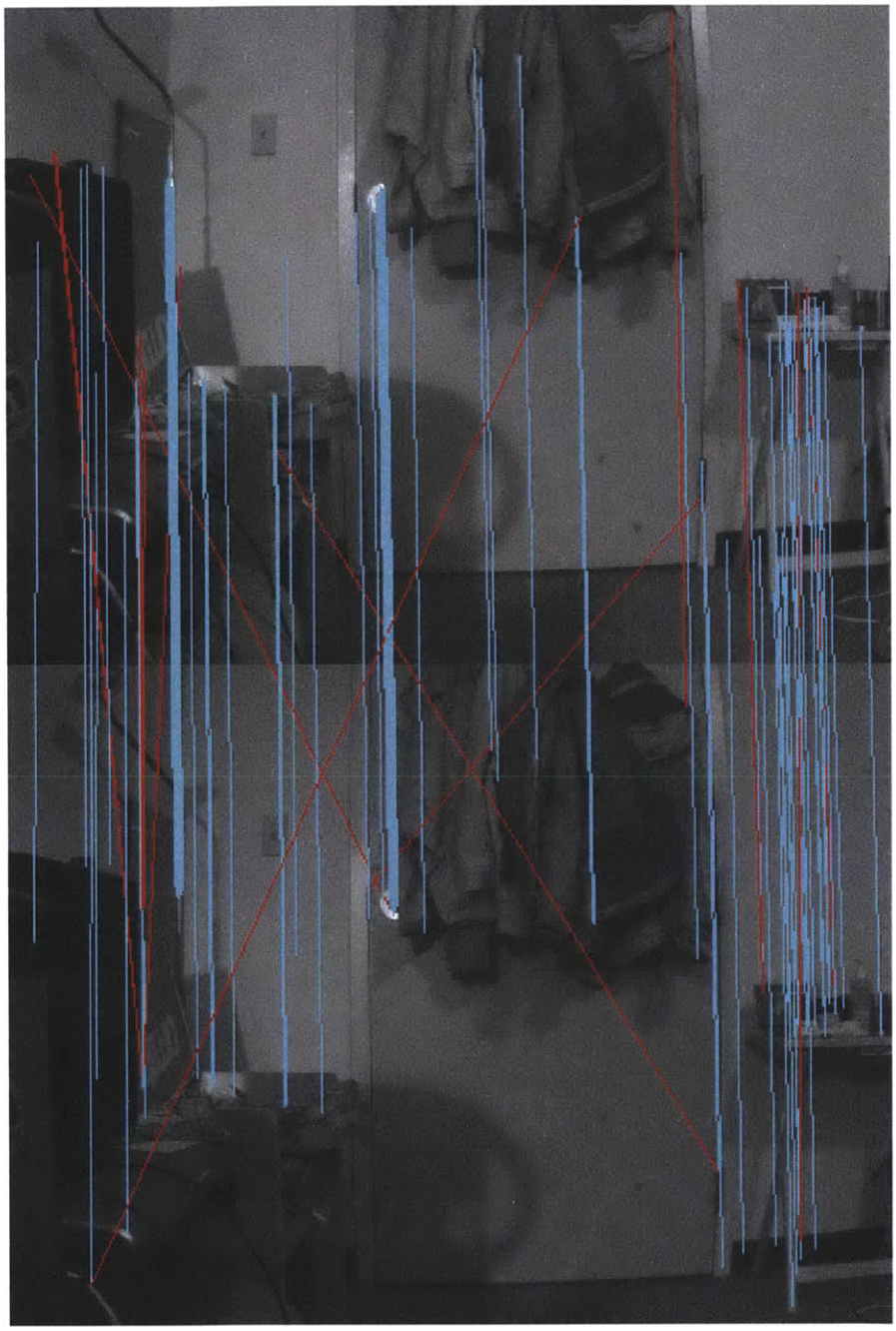

Figure 3-7: Ransac results. Inliers are shown in blue while outliers are shown in red.

linear velocity v and angular velocity ω. These can be referred to a static co-ordinate system W, but also to the co-ordinate system with the pose of camera as the origin C_k.

$$\mathbf{x}_{C_k}^{W} = \begin{pmatrix} \mathbf{r}_{C_k}^{W} \\ \mathbf{q}_{C_k}^{W} \\ \mathbf{v}^{W} \\ \omega^{C_k} \end{pmatrix} \tag{3.7}$$

Since constant velocity is assumed, the following equation can be used to model transitions:

$$f_v = \begin{pmatrix} \mathbf{r}_{C_{k+1}}^{W} \\ \mathbf{q}_{C_{k+1}}^{W} \\ \mathbf{v}_{k+1}^{W} \\ \omega_{C_{k+1}} \end{pmatrix} = \begin{pmatrix} \mathbf{r}_{C_k}^{W} + (\mathbf{v}_k^{W} + \mathbf{V}^{W})\Delta t \\ \mathbf{q}_{C_k}^{W} \times \mathbf{q}((\omega_{C_k} + \Omega^{C})\Delta t) \\ \mathbf{v}_k^{W} + \mathbf{V}^{W} \\ \omega_{C_k} + \Omega^{C} \end{pmatrix} \tag{3.8}$$

[1] V and Ω represent zero-mean Gaussian velocity noise while Δt represents the time difference between frames.

3.5.3 Point Representation

Points in the image can be represented using either (a) Euclidean Point Parameterization or (b) Inverse Depth Parameterization. Euclidean point parametrization uses only the x, y, z location of the feature to keep track of its location. Inverse depth parameterization adds more information about the feature, resulting in a more accurate description at higher computational cost. The following are the two representations:

- **Euclidean Point Parameterization:** In this case, a point is represented as:

$$\mathbf{x}_i = \begin{pmatrix} x_i & y_i & z_i \end{pmatrix}^{\top} \tag{3.9}$$

[1] As a sidenote, if the implementation had access to motion sensors, then the model would not need to estimate velocities and the camera state would simply be $\mathbf{x}_{C_k}^{W} = \begin{pmatrix} \mathbf{r}_{C_k}^{W} \\ \mathbf{q}_{C_k}^{W} \end{pmatrix}$

- **Inverse Depth Point Parameterization:** In this case, a point is defined by the 6 state vector:

$$\mathbf{y}_i = (\quad x_i \quad y_i \quad z_i \quad \theta_i \quad \phi_i \quad \rho_i \quad)^\top \tag{3.10}$$

where $\mathbf{y}_i$ encodes the ray from the first camera position from which the feature was observed. x_i, y_i, z_i represent the camera center. θ_i and ϕ_i represent the azimuth and elevation. The point's depth along the ray d_i is encoded by its inverse $\rho_i = 1/d_i$ [4].

3.5.4 Measurement Model

A measurement model allows the system to relate the parameters of the physical world to what is captured through the lens of the camera. As Forsyth states, the measurement model allows "to establish quantitative constraints between image measurements and the position and orientation of geometric figures measured in some arbitrary external coordinate system" [14]. The purpose here is to estimate the position (u, v) where the feature is expected to be found. The measurement model used in this implementation is a pinhole camera model (as described in [4]). Both Euclidean and inverse depth points can be transformed to the camera reference frame $\mathbf{h}^C = (h_x \quad h_y \quad h_z)^T$:

- **Euclidean:**

$$\mathbf{h}^C = h^C_{xyz} = \mathbf{R}(\mathbf{q}^W_{C_k}) \cdot (\mathbf{y_i^W} - \mathbf{r}^W_{C_k}) \tag{3.11}$$

- **Inverse Depth:**

$$\mathbf{h}^C = h^C_\rho = \mathbf{R}(\mathbf{q}^W_{C_k}) \cdot (\rho_i((x_i \quad y_i \quad z_i)^T - \mathbf{r}^W_{C_k}) + \mathbf{m}(\theta_i, \phi_i)) \tag{3.12}$$

where $\mathbf{R}(q^W_{C_k})$ represents a rotation matrix that is computed from $\mathbf{q}^W_{C_k}$ and $\mathbf{m}$ is the conversion function from azimuth-elevation angles to a unit vector.

At this stage the points can be projected using the standard pinhole model [14], since the camera does not directly observe $\mathbf{h}^C$, but instead uses its projection, which is modeled next:

$$\mathbf{h} = \begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} u_0 - \frac{f}{d_x}\frac{h_x}{h_z} \\ v_0 - \frac{f}{d_y}\frac{h_y}{h_z} \end{pmatrix} \tag{3.13}$$

where (u_0, v_0) is the camera's principal point (center coordinates), f is the focal length and d_x, d_y the pixel size. $\mathbf{h}$ then can be used to narrow down the possible locations of the feature in the next frame, since it predicts a feature location (u, v). The actual measurement, the observation $\mathbf{z}_i$ = (u,v), is obtained by the feature detector.

3.5.5 Putting it all together: Extended Kalman Filter (EKF)

The Extended Kalman Filter (EKF) is what brings all the previous parts together to produce the motion estimates. The main idea consists of having a probabilistic feature-based map. At any given timestep, this map contains an instant snapshot of current estimates of the state of the camera (Illustrated in Equation (3.7)), as well as the features being successfully tracked from one frame to the next and their respective uncertainties. Features that are occluded or simply not available in one frame are kept around as candidates for a set number of frames. If the feature is not seen after the preset number of frames, it is deleted. The map is initialized and is updated continuously by the EKF in subsequent steps –constantly evolving. The camera motion model (Equation (3.8)) is used to update camera motion. Features are also updated during camera motion and feature observation. When a new feature is found, the state of the map is enlarged; if necessary, features can also be deleted.

The state $\hat{\mathbf{x}}$ is then composed of the camera and all the map features. The covariance matrix $\mathbf{P}$ estimates a first order uncertainty distribution showing the size of possible deviations from the values in $\hat{\mathbf{x}}$ (estimating mean and covariance assuming a Gaussian distribution).

$$\hat{\mathbf{x}} = \begin{pmatrix} \mathbf{x}_{C_k}^W \\ \mathbf{y}_1 \\ \mathbf{y}_2 \\ \vdots \\ \mathbf{y}_n \end{pmatrix}, \mathbf{P} = \begin{bmatrix} P_{xx} & P_{xy_1} & P_{xy_2} & \cdots \\ P_{y_1x} & P_{y_1y_1} & P_{y_1y_2} & \cdots \\ P_{y_2x} & P_{y_2y_1} & P_{y_2y_2} & \cdots \\ \vdots & \vdots & \vdots & \end{bmatrix} \tag{3.14}$$

Prediction

The first step in the EKF loop is to produce predictions $\tilde{\mathbf{x}}$, and $\tilde{\mathbf{P}}$. Since features are assumed to be static, this operation only concerns states and covariances related to the camera after time period Δt_k. For the covariance calculation, Jacobians $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial x}^\top$ are used.

$$\begin{aligned} \tilde{\mathbf{x}}_{C_k(k+1|k)}^W &= \tilde{f}_v(\tilde{\mathbf{x}}_{C_k(k|k)}^W, \Delta t_k) \\ \tilde{\mathbf{y}}_{i(k+1|k)} &= \tilde{\mathbf{y}}_{i(k|k)}, \forall i \\ \tilde{\mathbf{P}}_{(k+1|k)} &= \frac{\partial f}{\partial x} P_{(k|k)} \frac{\partial f}{\partial x}^\top + Q_k \end{aligned} \tag{3.15}$$

Measurement/Observation

During operation, the FPGA is constantly providing feature locations and descriptors for the current frame. At each new EKF time step, correspondences between features are found and features are tracked, and the transformation from frame to frame is found using RANSAC. Using the measurement model described in Section 3.5.4, positions of features relative to the camera can be estimated. Specifically, equations (3.12) along with (3.13) are used to estimate the position of a point feature relative to the camera, given $\mathbf{x}_{C_k}^W$.

Feature Initialization

From the first observation of a feature, no depth information can be obtained since there is no other information besides the observed 2D position of the feature in the image. Therefore, feature parameters need to be initialized somehow. For feature

initialization, this implementation uses the approach suggested by Civera [4]. The main idea is to assign a Gaussian prior in inverse depth which assumes the feature will be in front of the camera, and account for all possible locations with different probabilities (even infinity, which is given a negligible probability). However, the end-point of the ray where the feature might be is most likely at the current camera location estimate:

$$\begin{pmatrix} \hat{x}_i & \hat{y}_i & \hat{z}_i \end{pmatrix}^\top = \hat{\mathbf{r}}^{WC} \tag{3.16}$$

The direction of the ray can be computed from the observed point, expressed in the world coordinate frame:

$$\mathbf{h}^W = \mathbf{R}(\mathbf{q}_{C_k}^{\hat{W}})(u \quad v \quad 1)^\top \tag{3.17}$$

Parameters θ and ϕ (azimuth and elevation) can be computed as follows:

$$\begin{pmatrix} \theta_i \\ \phi_i \end{pmatrix} = \begin{pmatrix} \arctan\left(\mathbf{h}_x^W, \mathbf{h}_z^W\right) \\ \arctan\left(-\mathbf{h}_y^W, \sqrt{\mathbf{h}_x^{W^2} + \mathbf{h}_z^{W^2}}\right) \end{pmatrix} \tag{3.18}$$

Parameter ρ_0 (representing depth) is set empirically such that the 95% confidence region is on the ray tracing from the camera to infinity. Civera [4] set $\rho_0 = 0.1$, $\sigma_\rho = 0.5$ and that is the value used in the implementation. Finally, the state covariance matrix, $\hat{\mathbf{P}}_{k|k}^{new}$ is initialized as follows:

$$\hat{\mathbf{P}}_{k|k}^{new} = \mathbf{J} \begin{pmatrix} \hat{\mathbf{P}}_{k|k} & 0 & 0 \\ 0 & \mathbf{R}_i & 0 \\ 0 & 0 & \sigma_\rho^2 \end{pmatrix} \mathbf{J}^\top \tag{3.19}$$

where $\mathbf{J}$ is the Jacobian and $\mathbf{R}$ is the image measurement error covariance, defined as (as described by Davison in [9]):

$$\mathbf{R} = \begin{pmatrix} \Delta_{\alpha_p^2} & 0 & 0 \\ 0 & \Delta_{\alpha_p^2 e} & 0 \\ 0 & 0 & \Delta_{\alpha_p^2 v} \end{pmatrix} \tag{3.20}$$

where $\alpha_{p,e,v}$ are the pan, elevation and (symmetric) vergence angles respectively.

Updating EKF and Maintaining the Map

The last step in the EKF loop is the update. Once a measurement $\mathbf{z}_i$ (the observation of (u,v)) of a feature has been returned, the Kalman gain $\mathbf{W}$ and the filter update can be performed.

$$\mathbf{W} = P\frac{\partial h_i^\top}{\partial x}S^{-1} \tag{3.21}$$

$$\hat{\mathbf{x}}_{new} = \hat{\mathbf{x}}_{old} + W(\mathbf{z}_i - \mathbf{h}_i) \tag{3.22}$$

$$P_{new} = P_{old} - WSW^\top \tag{3.23}$$

A feature is deleted if after a preset number of attempts to detect and match a feature that should be be visible, the process fails more than 50% of the time. The corresponding entries from the state and covariance matrices are removed, and the map is pruned.

The next section describes a basic EKF simulation, used as a starting point for building the EKF as described in this section.

3.6 Basic Simulation

In the first EKF simulation, the state vector consisted of the 3 d.o.f. camera pose: (x, y, θ). A predetermined range for landmark observation was set, and observation information from landmarks was added to the state vector as landmarks were encountered. Since the main objective was to validate the performance of the EKF, factors such as data association were simplified by assigning recognizable ID's to landmarks. Figures 3-8 and 3-9 (Page 40) show the actual path traversed by the camera and the EKF path estimate obtained from the simulation. Figure 3-8 shows the estimated path without using measurements. Since the EKF cannot use any measurements to update its initial prediction, the estimate diverges from the true path traversed by the camera. Figure 3-9 shows the estimated path integrating measurements. In this

case the EKF uses landmark observations to update its predictions and output accurate estimates. The actual and estimated paths only differ slightly, which is expected because some Gaussian noise was added in the simulation. However, the error is very small (almost unnoticeable at times), showing the EKF is estimating the camera pose correctly.

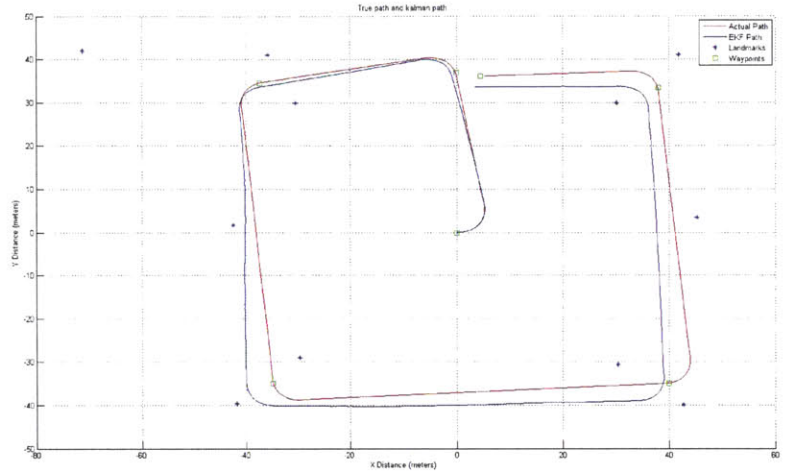

Figure 3-8: Basic EKF simulation results without integrating measurements

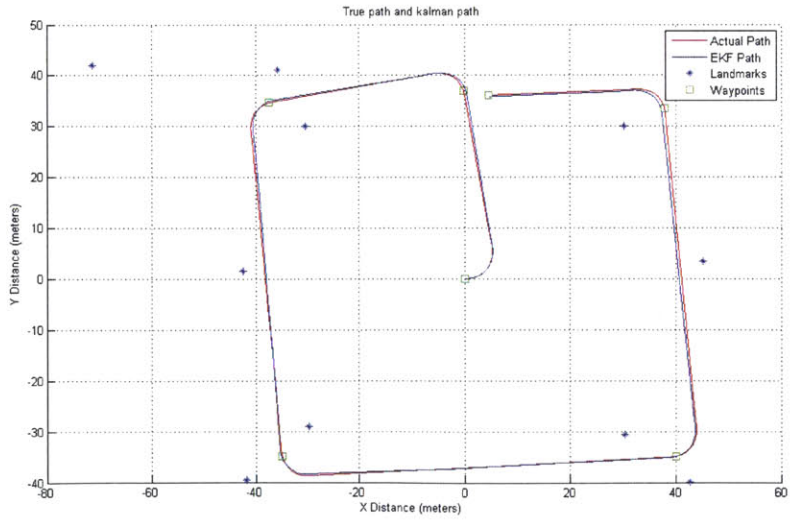

Figure 3-9: Basic EKF simulation results integrating measurements

Figures 3-10 and 3-11 (Page 3-10) illustrate the covariance of x and y respectively.

A covariance of zero means two variables are uncorrelated. In this case however, covariance increases as the simulation advances, meaning the variables are highly correlated. This is the expected behavior for covariance, since estimates should become fully correlated as the simulation advances.

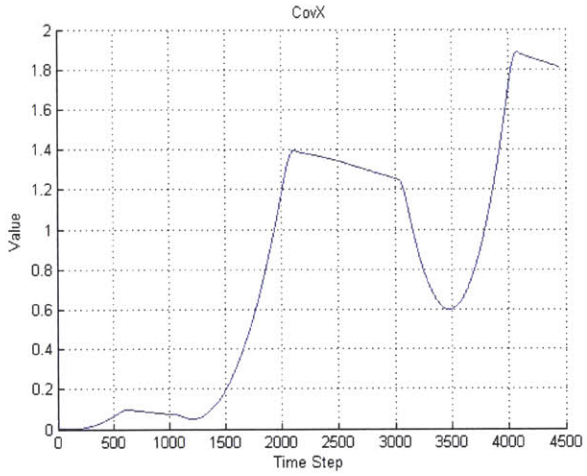

Figure 3-10: Covariance of x

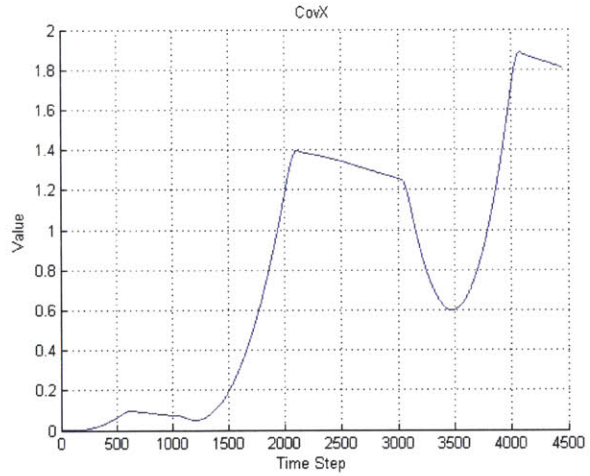

Figure 3-11: Covariance of y

Chapter 4

Implementation

The implementation of the system is distributed across two main components: The smart camera (FPGA and CPU) and a separate computer (ground station). The FPGA interfaces with the image sensor, computes features and outputs feature locations, and descriptors. The FPGA can also output the original image, either raw or JPEG compressed. The CPU inside the camera handles all communications and provides an interface between the memory and the Ethernet port (camera output). A ground station is used to find correspondences between features seen in consecutive frames and to perform estimation of the camera pose and velocity.

4.1 FPGA

One of the benefits of the Elphel 353 camera is that it has available had many open-source modules implemented in the FPGA, as well as many drivers and programs stored in memory to be run by the CPU. Figure 4-1 provides an overview of the camera components, including optional add-ons[1]. The following sections will describe the hardware and software implementations in detail, starting with the relevant parts already in the camera and then describing modifications and components added to the design.

[1]Only the base camera was used for this implementation, but possible add-ons include a hard-drive, a compact-flash memory adapter and an extra board with expansion ports.

4.1.1 Elphel-353 Stock FPGA Design

Some definitions:

- **Fixed Pattern Noise (FPN):** This corresponds to pixel noise, which is a weakness of CMOS sensors (Used in the Elphel Camera) [2]. This is caused by "different DC and gain offsets in the signal amplification" [28]. A hardware module in the FPGA mitigates this effect.

- **Vignetting:** This refers to the uneven amount of light that permeates the optical system of the camera. The result is a lighter image in the center that gets darker in the periphery. If the illumination distribution is known, then this undesirable effect can be compensated [29]. A module in the camera implements vignetting compensation.

- **Gamma Correction:** Gamma correction refers to compensating for nonlinearities in the camera's sensor, resulting in slightly different pixel values given the same light stimulus [12]. This effect is compensated by a module in the FPGA.

In the stock camera design, the main purpose of the FPGA is to carry out JPEG compression. The image sensor outputs directly to the FPGA, and as the image is stored in the 64 MB SDRAM for the first time, it undergoes vignetting compensation, gamma correction and histogram calculation. Then the memory controller acts as the main administrator of pixel data. The controller has four channels [3] that multiplex data access according to a preset priority (Incoming pixels using channel zero have the highest priority). Each channel has its own timing behavior and requirements that adapt to function properly with the memory controller.

Once a portion of the image is stored, the controller uses channel two to send 20×20 pixel tiles to the JPEG compressor. The controller also sends the raw image

[2] CMOS (Complementary metal-oxide semiconductor) sensors, as opposed to CCD (charge-coupled device) sensors are easier to integrate with other hardware components and are therefore more prevalent in custom hardware designs [28].

[3] In this context, a channel corresponds to dedicated data input and/or output paths accessing the SDRAM chip.

to the CPU SDRAM using channel three, so the uncompressed image is available as well. The compressed tile is then sent to the CPU where it is stored in a 19 MB circular buffer, along with the other tiles corresponding to the same image. When the circular buffer has a full frame, output for that frame is ready to be read by the CPU so the frame reaches the Ethernet port.

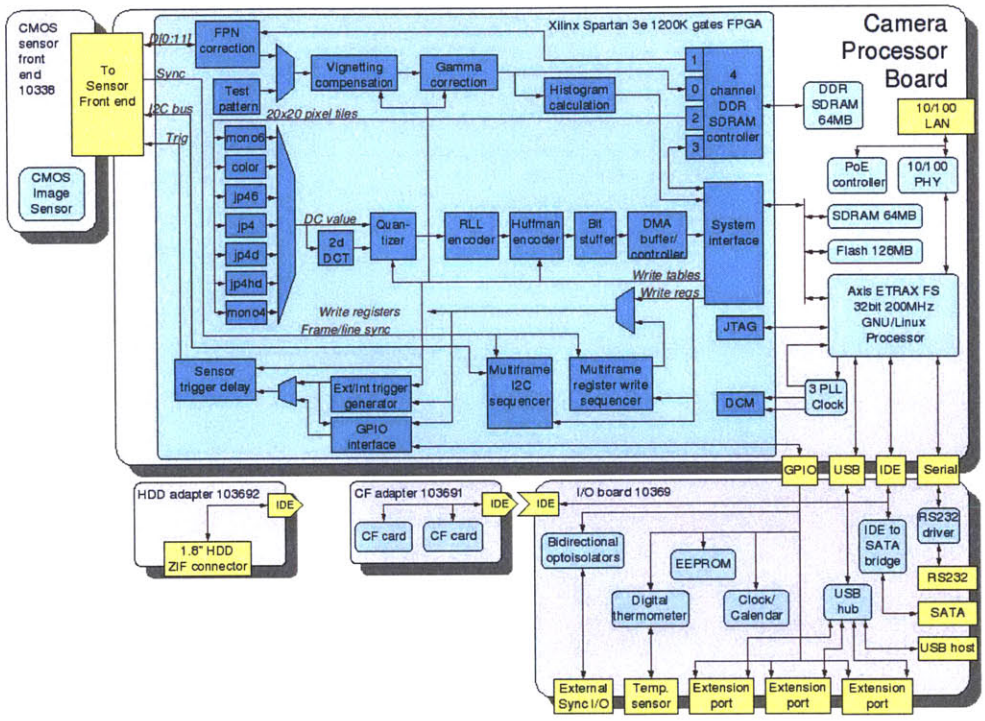

Figure 4-1: Elphel-353 System Diagram (Elphel)

Frame Rates

The following tables (Table 4.1 and Table 4.2) illustrate resolutions and the corresponding frame rates supported by the image sensor and the JPEG compressor.

4.1.2 In-FPGA Feature Detection

This section describes my custom design for the camera's FPGA. Although part of the focus of this work has been the implementation of corner detection in hardware,

Resolution	Frame Rate (FPS)
Full Resolution 5mp	15
1920 x 1080 (Full HD)	30
1280 x 720 (HD)	60

Table 4.1: Aptina image sensor supported resolutions and frame rates.

Resolution	Frame Rate (FPS)
Full Resolution 5mp	2
1920 x 1080 (Full HD)	27
1280 x 720	29
640 x 480	30
320 x 240	33

Table 4.2: JPEG compressor supported resolutions and average frame rates.

the underlying research is the feasibility of carrying out feature detection on FPGAs. Therefore the design had to be modular from the beginning, such that different feature detectors could be swapped in and out as necessary. This modularity will also pave the way for more complex implementations in the future. A feature detector such as SIFT is significantly more complex than the Harris corner detector, but the implementation in hardware has the potential to offer tremendous speedup vs. the implementation in CPU architectures.

The following is a description of FPGA modules in the design. Figure 4-2 (Page 47) shows the feature detection design implemented on the FPGA. The image sensor module, vignetting gamma correction and the histogram were not changed from the stock design. Since the memory controller was very tightly coupled to support all four channels, the design leaves the four channels in place. Next is an explanation in more detail of two new key modules: the Double Buffer and the Sliding Window.

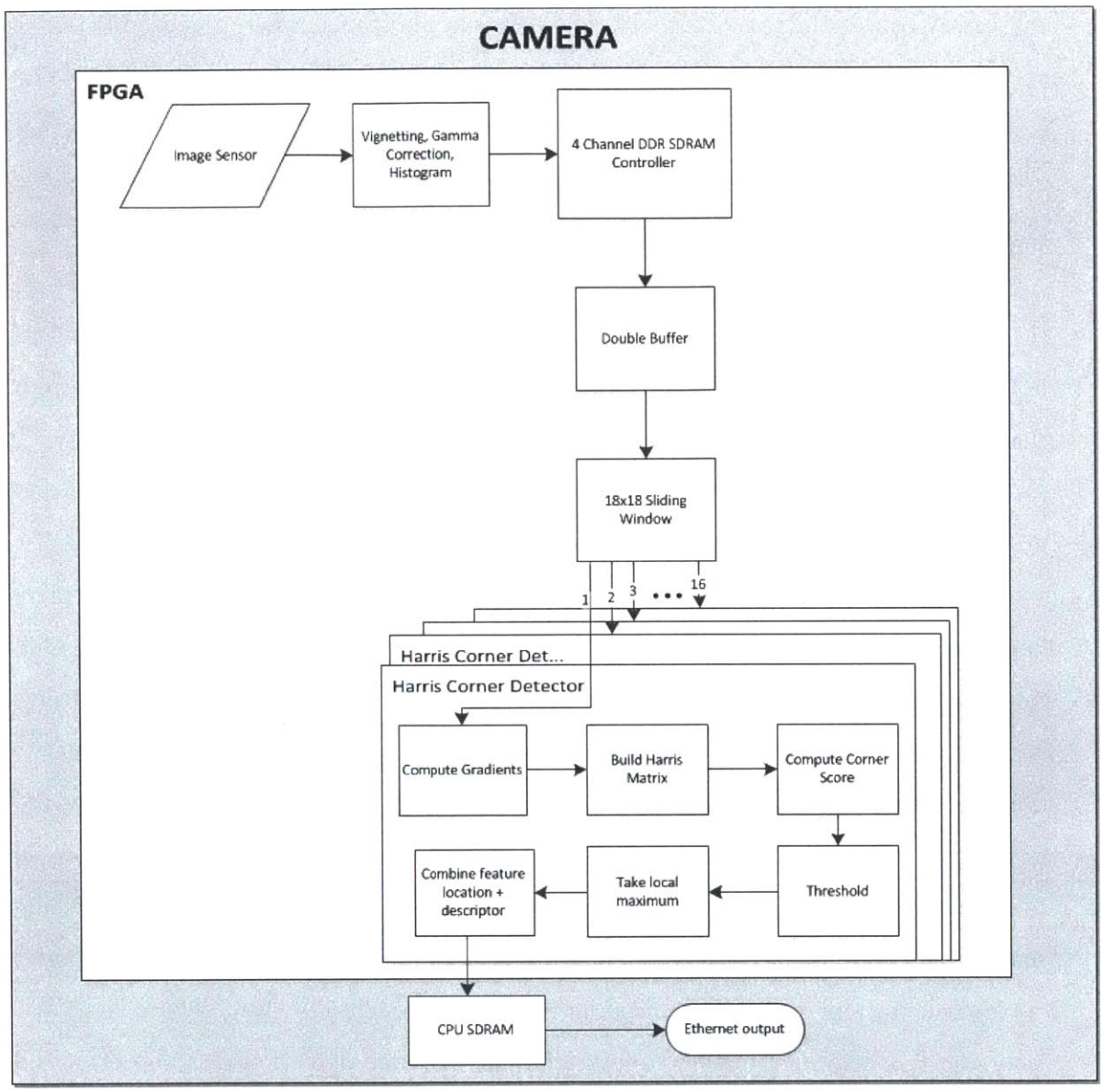

Figure 4-2: FPGA Design Overview

Double Buffer

In order to maximize the frame rate of the system (or when the output is a feature, the rate of features per second), it is important not to stop the stream of pixels coming from the image sensor due to resources being tied up by previous pixel processing. A naive approach to image processing would be to store a whole frame in memory, process it (in this case extract features), and then request the next frame. However,

constantly processing pixels is much more efficient and this is where the double buffer proves useful. In this case, a set of logic gates can be used to store a portion of the incoming image, while another set is used to process the previously stored portion of the image. This parallelism means that the total frame rate is constrained by the longer of these two operations, not the sum of the two operations.

Given a window of $n \times n$ pixels, a buffer of size n^2 bytes is necessary to store all the pixels in the window. A double buffer scheme implements two of these buffers; *buffer 1* can be used to store incoming pixels, while *buffer 2* holds the pixels from the previous $n \times n$ set of pixels. Since *buffer 1* is storing incoming pixels, *buffer 2* can be used for processing. Once processing is done in *buffer 2*, the roles switch and now *buffer 2* can store incoming pixels while *buffer 1* feeds pixels for processing.

A variable size window of pixels is particularly important because different pixel operations require different sets of surrounding pixels. Two dimensional edge detection requires a 3×3 window at a minimum, while Harris corner detection requires at least a 5×5 window. The main consideration however to deciding whether the double buffer is needed is to evaluate the computational need of the operation of the time. The system currently waits approximately 200 cycles between sets of pixels, so if 200 cycles provide enough processing time, a single buffer can be used (processing pixels as they are read from memory). However, if a window is needed, using a buffer is required since pixels are produced by the sensor sequentially, from left to right and from top to bottom. However, operations like 1D edge detection can be performed on the incoming set of pixels, and an example of this is shown in Section 4.3.1.

Sliding Window

Many feature detection algorithms require analysis of a given pixel and the pixels around it. In order to gather the required pixels for processing, a sliding window of pixels can traverse the image, processing pixels as it slides (either sideways or from top to bottom). In this implementation the sliding window is organized as an $n \times m$ array of pixels that are read from the buffer. After processing the current window, the window shifts by adding more values and copying old values to their update locations.

Figure 4-3 illustrates this process with numbers assigned to unique pixels.

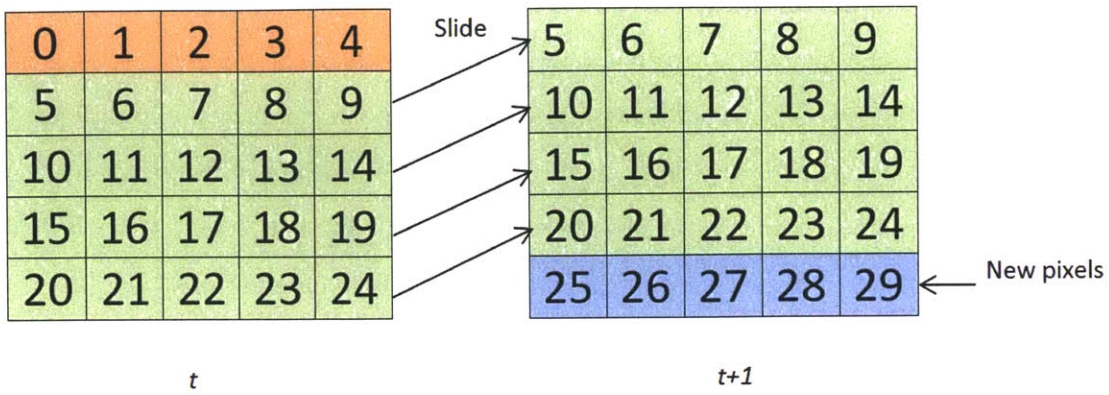

Figure 4-3: Sliding Window showing transition from t to $t+1$ for a 5×5 window. Values correspond to unique pixels.

4.1.3 Harris Corner Detector

For the FPGA implementation, the Harris Corner detector was split into six parts: gradient computation, Harris matrix, corner response, threshold computation, local maximum response, and finally output feature location and descriptor.

Sliding Window for Harris Corner Detector

The sliding window for the Harris corner detector has dimensions $5\,rows \times 16\,columns$. The minimum requirement for the detector to work is a 5×5 window. The buffer in the implementation holds 16×16 pixels, so it is more efficient to have a window that just slides down as opposed to down and horizontally. Another benefit of having 16 columns is that it is a multiple of 4, which is important because each memory read can retrieve at most 4 bytes[4] (4 pixels). In this case a row can be updated in 4 memory reads (16 pixels).

[4]The width of the memory is 32 bits. Therefore the width of intermediate buffers is also 32 bits, creating a 32-bit pipeline and allowing 4 pixels to be read at once.

Gradient Computation

The first step of the algorithm is to compute image gradients I_x and I_y. In this case, all the gradients around the pixels for which the corner response is required are computed. The gradient computation is equivalent to convolving the 3×3 pixel matrix[5] with horizontal and vertical gradient operators. Since finding a gradient is equivalent to finding a derivative, a derivative operator such as the Sobel operator can be used. This step results in the following two calculations:

$$\begin{bmatrix} p_0 & p_1 & p_2 \\ p_3 & p_4 & p_5 \\ p_6 & p_7 & p_8 \end{bmatrix} * \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \; and \; \begin{bmatrix} p_0 & p_1 & p_2 \\ p_3 & p_4 & p_5 \\ p_6 & p_7 & p_8 \end{bmatrix} * \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \tag{4.1}$$

This computation is ideal for an FPGA because it can be done completely using combinational logic where the latency is just the propagation delay of the signal. For example, the computation of I_x is simply $(p2 - p0) + ((p5 - p3) << 1) + (p8 - p6))$, where the shift by 1 bit is equivalent to multiplying by two[6].

Harris Matrix

The next step is to build the Harris Matrix (Equation (3.4)). For this step, the following are needed:

$$\sum_{x,y \in W} I_x^2, \sum_{x,y \in W} I_y^2, \sum_{x,y \in W} I_x I_y. \tag{4.2}$$

Once the sliding window is ready (meaning pixels have been loaded), summations shown in (4.2) can be computed since the gradient calculation is practically instant (taking only the propagation delay). Since the FPGA can implement the adder circuit needed for this computation, the summation outputs are available in the next cycle.

[5]This is the 3×3 pixel matrix surrounding the pixel for which we want the response.

[6]Using bit shifts whenever possible as opposed to multiplying is preferred because FPGAs have a limited number of available multipliers (The Spartan3e has only 28 multipliers).

Corner Response

Once the Harris Matrix is built, the next step is to calculate the corner response, shown in Equation (3.5). In terms of the matrix components, this is equivalent to the following equation:

$$R(x,y) = \underbrace{\sum_{x,y\in W} I_x^2 \cdot \sum_{x,y\in W} I_y^2 - (\sum_{x,y\in W} I_x I_y)^2}_{\alpha} - \underbrace{k \cdot (\sum_{x,y\in W} I_x^2 \cdot \sum_{x,y\in W} I_y^2)^2}_{\beta} \tag{4.3}$$

In order to meet timing constraints, equation (4.3) is solved in three clock cycles. During the first and second cycles, α and β are computed and on the third cycle, β is subtracted from α to produce R. Since division and multiplication should be a power of two whenever possible, $k = 1/8$. This division is computed by performing a 3-bit right shift, since this is equivalent to dividing by $8 = 2^3$.

Threshold Computation

From the corner response, the module determines whether the pixel is a corner by comparing the response to a threshold. This threshold varies depending on factors such as lighting conditions, distance of features, focus of camera, etc. and is therefore set experimentally. If the response is above the threshold, then the pixel is considered a corner; if it is not above the threshold, it is discarded.

Local Maximum Response

To prevent too many features conglomerating in small areas of the image, the system keeps the point of locally maximum R as the detected feature point. This means keeping only pixels where $R(x, y)$ is bigger than R for all the local neighbors. In this implementation, the neighborhood is defined as the local 16×16 window.

Output Feature Location And Descriptor

After computing local maximum responses, features are known to the system. Therefore, once it is determined that a given pixel is a corner, the location of the pixel

and the 5×5 intensity patch around it is saved to a sector in memory organized as a circular buffer. The memory driver constantly scans this circular buffer and outputs features as they are received. Although features and corresponding intensity patches could be outputted directly, having an intermediate buffer (Assuming the buffer is big enough to hold the necessary amount of data) helps prevent the problem of features being found at a faster rate than the CPU can transmit the required bytes consisting of previous feature information. It also provides more flexibility for implementing intensity patches of different sizes. In some similar implementations 7×7 all the way to 15×15 intensity patches are preferred[9].

4.1.4 Parallelism

One of the strongest advantages of FPGAs is their ability to process data in parallel. Unlike a CPU, which runs processes sequentially, the FPGA can replicate many identical modules that can perform the same computation in parallel. This implementation takes advantage of this parallelism when it computes the Harris response for a given pixel. Once the 16×16 pixel window is loaded, 14 5×5 sub-windows of the 16×16 window are loaded into 14 different instantiations of the Harris module. This parallelism is also shown in Figure 4-2. Since the modules are identical, their latency is the same, so 14 corner responses are obtained at once. To find a local maximum, a current maximum of the response is stored and that value is compared against the newly calculated responses. If a new maximum is found, then the current local maximum is updated, along with the corresponding descriptor.

4.2 Estimation

I implemented the estimation system described in Section 3.5 in Matlab using my own modules and the open-source libraries from OpenSLAM [24]. Inputs can be a sequence of images, live images from the Elphel camera in this case, and feature locations and descriptors. Matlab functions take care of the parsing of raw byte data coming from the Elphel camera, for the case of feature location and descriptor

input. Details on the testing setup, marshalling, and obtaining truth measurements are described in Section 5.5.

4.3 Modularity

A key consideration when designing this system was modularity. The system had to be easily extensible, since a future goal consists of having an FPGA that can deploy different feature detectors depending on the specific need. A good way to test a system's modularity is to see how easy it is to exchange similar components. In this case an edge detector was swapped for the corner detector.

4.3.1 Edge Detector

In order to test the modularity of the design, the Harris corner detector was swapped for an edge detector. This proved to be very straightforward, since the pipeline needed was almost the same. Some modules, such as those that derive gradients, could be reused. The result, illustrated in Figure 4-4 and Figure 4-5 (Page 4-4), could be accomplished essentially in real-time, since pixels were processed as the sensor produced them, only needing the three previous pixels to produce a pixel output.

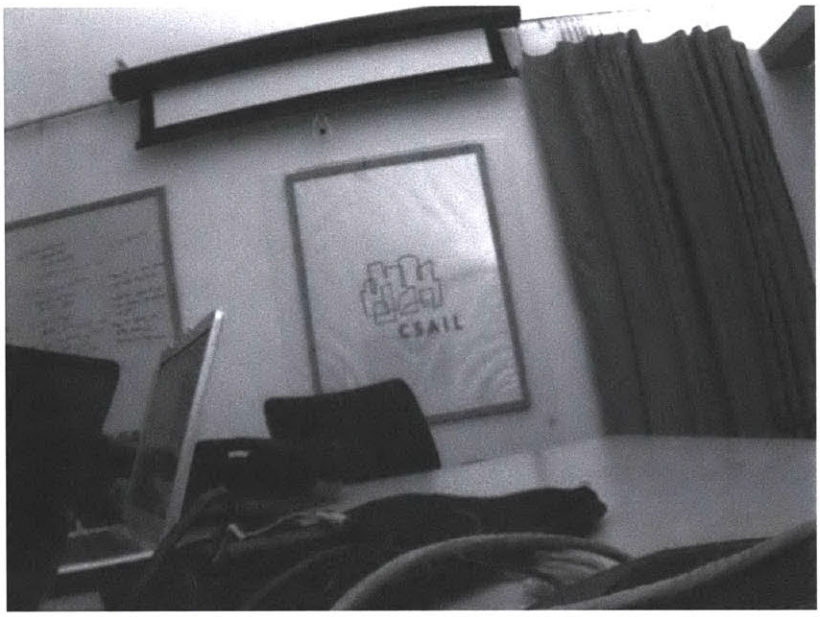

Figure 4-4: Source image for edge detection.

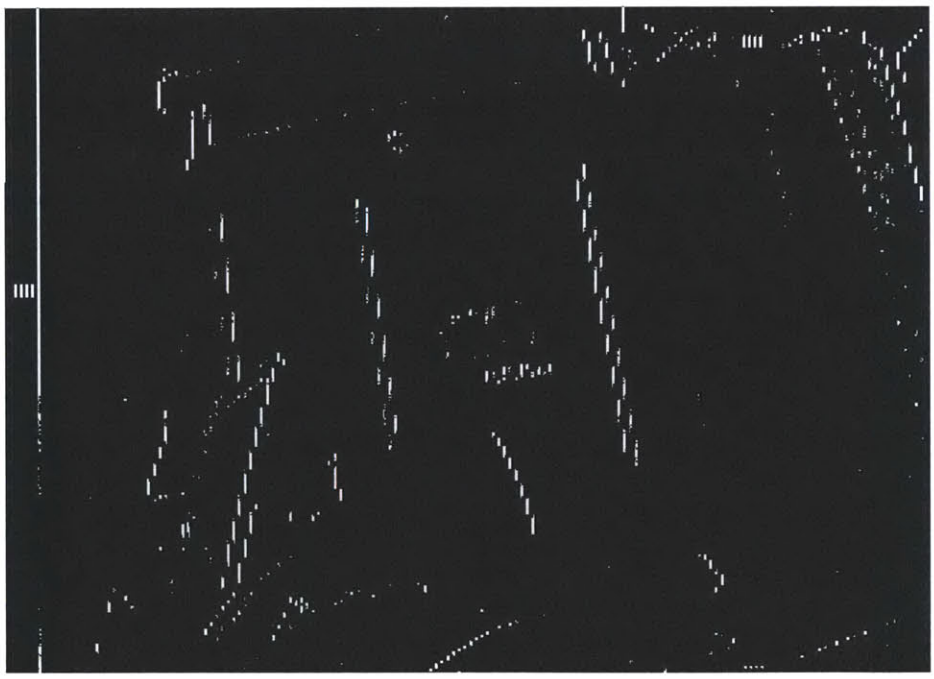

Figure 4-5: Image showing edges.

Chapter 5

Evaluation

Evaluation took place in stages. The first step was to validate some of the modules and general functionality (such as the double buffer, sliding window, etc.) in simulation. The implementation of the Harris corner detector was also validated by replicating it in Matlab. Having a copy in software was useful for getting an idea of how values within the module scale and to fine-tune the required constants (threshold and gain).

The estimation components of the system were also simulated. First, an EKF Matlab simulation was implemented using a simplified 3 d.o.f. model [25]. Then the 6 d.o.f. implementation of EKF SLAM in Matlab was used to validate the whole system pipeline and establish some measures of performance. The system was first tested using pre-stored images along with the software implementation of the Harris corner detector. Then the whole system using both the FPGA and the camera was tested. Position and/or velocity error were measured for each scenario and constitute the main measure of performance.

5.1 Validation

Simulation tools were used initially to validate the overall system flow and the actual implementation of the Harris corner detector. A second validation method consisted of a software implementation that mimics (to the extent possible) all the steps taken by the FPGA.

5.2 Simulation

Since compiling FPGA designs can take significant amounts of time (approximately 30 min. for this design), synthesizing and testing after every change is not a viable option. Simulation is a much better alternative. Simulation was key in verifying that the double buffer, sliding window, and corner responses among other modules were functioning as expected. Although most Verilog simulators are part of commercial offerings, only open-source tools were used. Icarus Verilog was used for Verilog simulation. GTKWave, another free tool, was used to interpret and display simulation results. Figure 5-1 shows a screenshot of the simulation environment for the Harris corner detector.

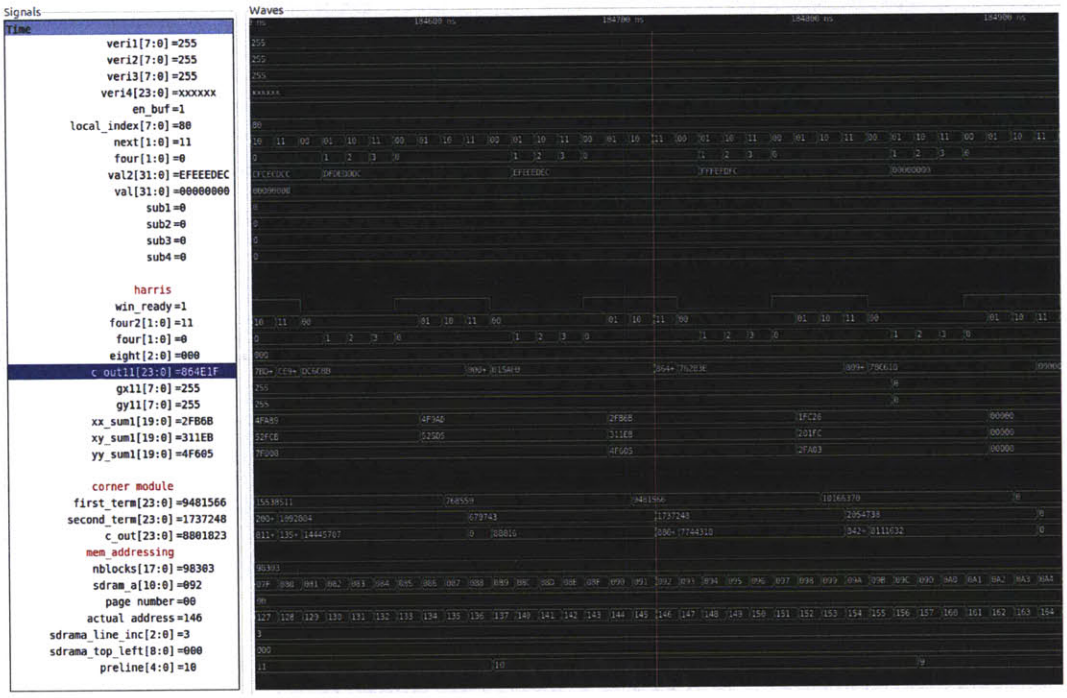

Figure 5-1: Simulation using Icarus Verilog and GTKWave.

5.3 Matlab Harris Corner Detector

A Matlab implementation was used to test the Harris corner detector implementation and get an idea of the results. The corner detector was implemented the same

way as the hardware implementation to the extent possible (without using Matlab optimizations or built-in functionality that would be different from the hardware implementation, for example image filtering or convolution). This was also very useful for getting an idea of how numbers scale (meaning which typically whats orders of magnitude are involved) during the necessary calculations. For testing using the Matlab implementation, numbers were first capped at 32-bits, and then adjusted to use fewer bits for certain steps. The FPGA has 18×18 bit multipliers with 36-bit output, so the corner response calculation had to be adjusted accordingly. Also the FPGA can implement at most 16-bit comparators, which matters due to the comparison of corner responses against a threshold. Some precision had to be sacrificed for the corner response calculation, since the maximum possible value exceeds 16-bits. However, the software implementation showed that as long as the threshold was adjusted properly, results were similar in the scaled down version. Figures 5-2, 5-3, 5-4 and 5-5 [5] show some results using the Matlab implementation with different thresholds for the corner response R.

Figure 5-2: Original Test Image

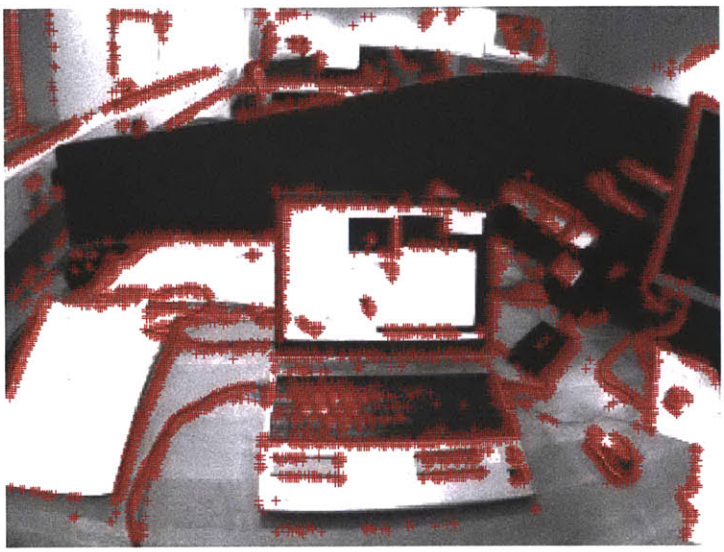

Figure 5-3: High threshold: 8940 corners

Figure 5-4: Medium Threshold: 4828 corners

5.4 Baseline Performance

To establish baseline performance, the implementation described in Chapter 4 was deployed in Matlab. A set of images, camera calibration information, and Vicon[1]

[1] A motion capture system.

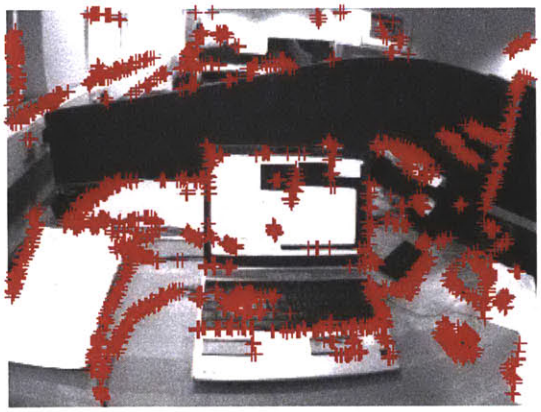

Figure 5-5: Low Threshold: 1965 corners

truth data were obtained from the publicly available dataset[19] published by the Computer Vision and Geometry Laboratory at ETH Zurich. Figure 5-6 (Page 60) shows four consecutive frames captured from the sequence of images. The implementation tracked features using the software implementation of the Harris corner detector. The minimum number of features tracked was set such that at least fifteen features per frame were tracked. Features being tracked are shown in red. Most of the features tracked are shown in all four frames, demonstrating that the implementation successfully predicts a feature's next location and finds it in the next frame. Figure 5-7 (Page 60) shows the source images for Figure 5-6 (Page 60).

Figure 5-8 (Page 61) and Figure 5-9 (Page 61) show the error for x, y, z and roll, pitch and yaw respectively. These correspond to the position, meaning elements 1-7 in the state vector (corresponding to x, y, z and the four elements of the orientation quaternion). For ease of interpretation, the quaternion was converted to Euler angles representing roll, pitch, and yaw.

The results are satisfactory. Over 100 frames, error does not diverge. When the error attempts to diverge, for example around frame 18 in the x-error (Figure 5-8), the EKF uses new measurements to reduce the error since it eventually returns to almost 0 percent error (around frame 30). Another interesting observation is that

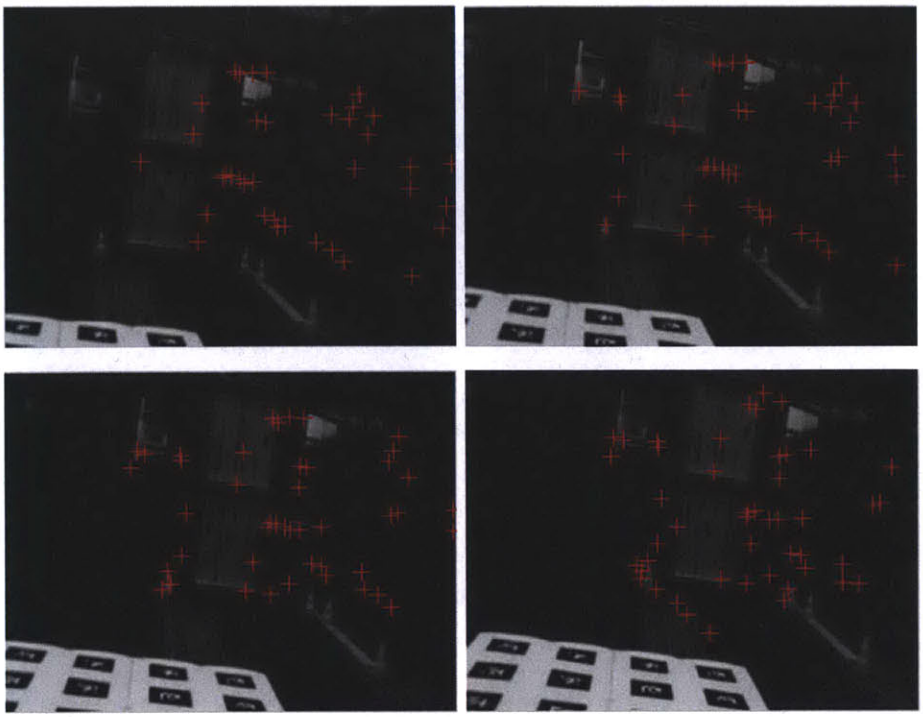

Figure 5-6: Starting from top left going clockwise are frames 22, 23, 24 and 25. Features being tracked are shown in red. The elapsed time from frame 22 to frame 25 was 0.1332 seconds.

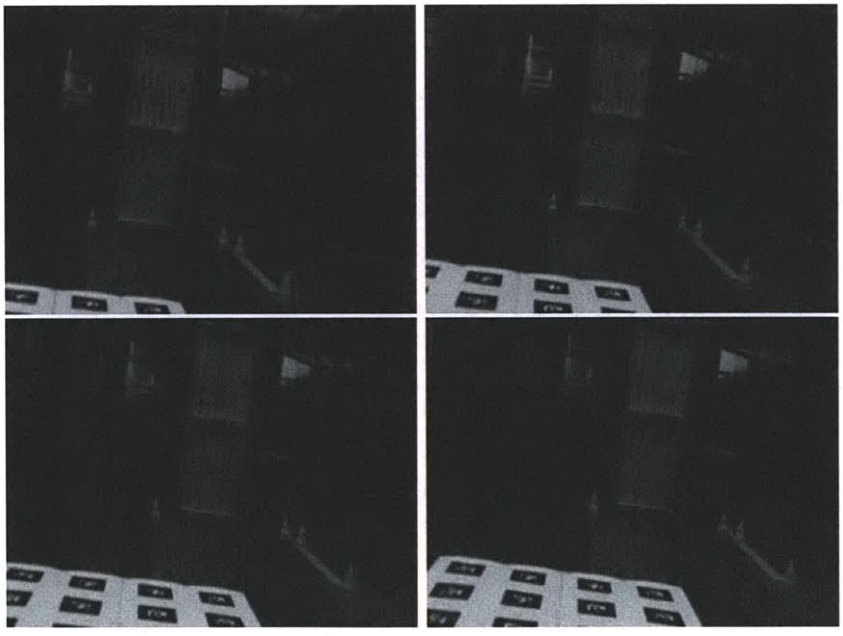

Figure 5-7: Image sequence used in Figure 5-6, but without features.

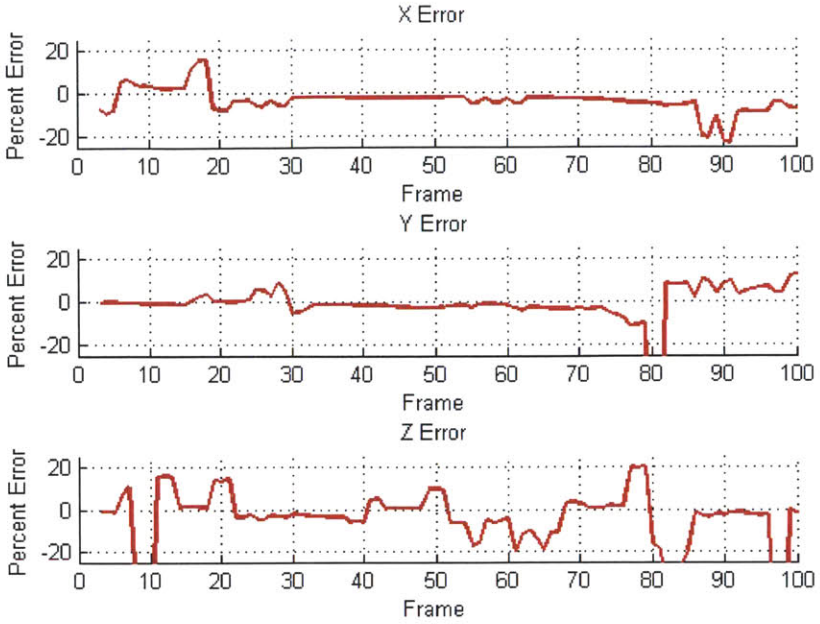

Figure 5-8: Error for X, Y and Z (first three terms of state vector).

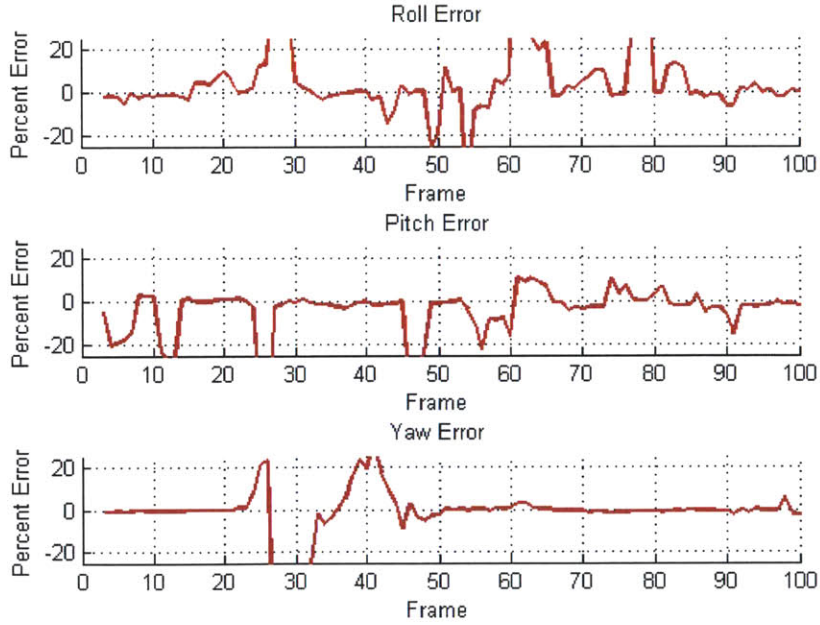

Figure 5-9: Error for roll, pitch and yaw (elements $4-7$ in the state vector).

error sometimes increases for a given set of frames. For example, again looking at x-error, error is close to zero between frames 63 and 75. However, starting with frame 85, it goes briefly to 20 percent error and then starts going back to close to zero over fifteen frames. This behavior over roughly 15 frames could be attributed to a feature that is being tracked inaccurately. It is important to remember that comparison between features is done using the descriptor and a threshold, so if the threshold is low enough for a feature that is not the intended match, a false match could result, introducing error.

Looking at the image sequence between frames 85 and 95, shown in Figure 5-10 (Page 62), an observation is that illumination in one area of the image changes noticeably. There are also some lights that are on in one frame and then are either turned off or occluded. The Harris corner detector is not able to handle illumination changes, so this could account for some of the error encountered in the corresponding section of the sequence. The area around each light (red circle) is also very similar, which could have resulted in a false match from one frame to the next, especially considering that one of the lights is occluded in the transition.

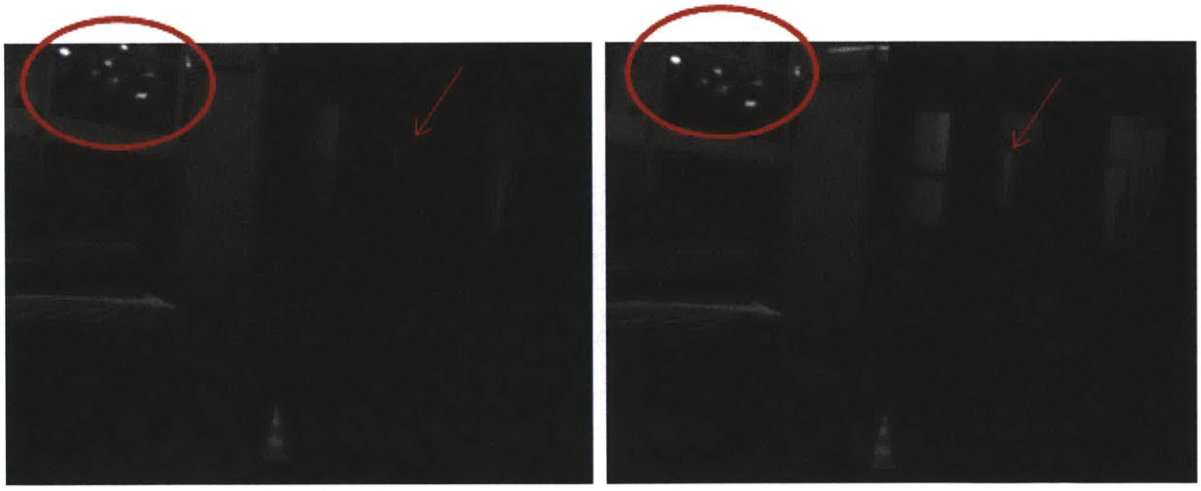

Figure 5-10: Frames 90 and 91 show changes in illumination in a specific area (red arrow) and possible occlusion (red circle), which the Harris corner detector is not able to handle.

Figure 5-11 shows the velocity error, which is close to zero for the most part and

does not diverge. A favorable factor was that while taking the images, the camera was kept meticulously at constant velocity. Figure 5-12 (Page 64) shows the angular velocity error, which is the error in roll, pitch and yaw. The error is close to zero for the most part and does not diverge. The fact that the error is close to zero for all six measures of velocity and none of them diverge verifies that velocity estimation is working correctly.

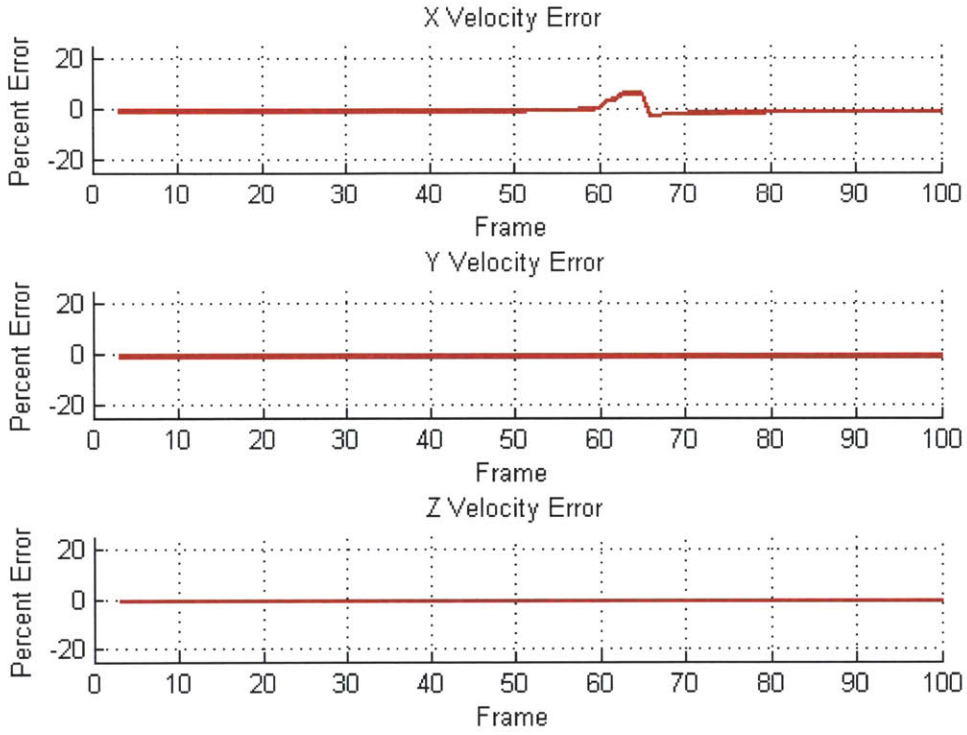

Figure 5-11: Velocity error.

5.5 Performance Using FPGA

To test the performance of the FPGA, a Hokuyo laser rangefinder was used to establish *truth* measurements. The Hokuyo works by emitting a laser around a 240 degree radius that goes out to a maximum range of four meters and calculates displacement based on the return time of the laser. For synchronization and communication,

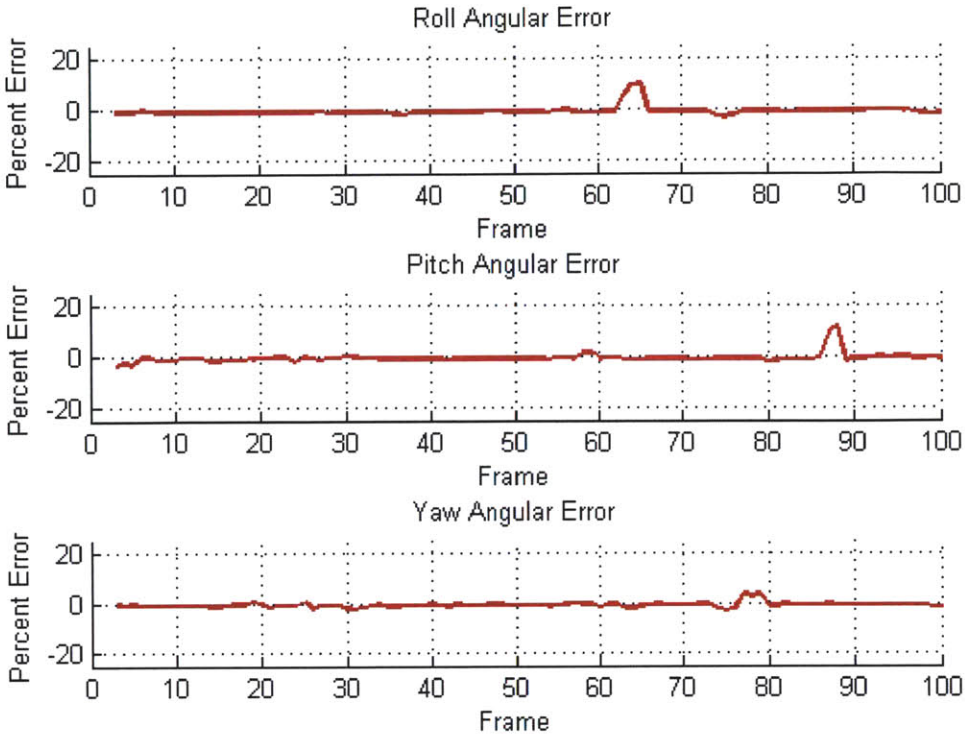

Figure 5-12: Angular velocity error.

the LCM (Lightweight Communications and Marshalling [17]) library was used. The FPGA was programmed to output feature locations and descriptors corresponding to a specific frame. This information was requested by the Matlab module and then processed. Timestamps were used to compare estimated values (from the EKF) against laser obtained measurements. For scan matching, which is "the problem of registering two laser scans in order to determine the relative positions from which the scans were obtained"[23], the open-source Fast and Robust Scan Matching Library[1] was used. The output of the scan matcher is a pose that can be compared to the motion estimate.

5.5.1 Raw speedup

An important measure when evaluating the FPGA implementation is the raw speedup obtained when compared against a CPU implementation. For this test, the latency of

the software implementation of the harris corner detector was compared against the latency of the Harris corner detector module in the FPGA for a 320×240 pixel image. The software implementation was tested on a computer with an Intel i5 M520 CPU running at 2.40GHz and 4.0GB of RAM. Initial loading of the image into memory was not taken into account in either case. The Matlab implementation took, on average, 2.4703 seconds from start to finish of the Harris corner detector. On the FPGA, computation of all the corners takes approximately: 100 cycles per 16×16 window (20 cycles for initial load + 9 cycles each time the window slides down) + 200 cycles between pixel windows = 300×300 total tiles in a 320×240 image, results in 90,000 cycles. At 100 MHz (the clock speed of the FPGA), 90,000 cycles are equivalent to $90,000\, cycles \times \frac{1 second}{100,000,000 cycles} = 0.9$ milliseconds. The FPGA implementation for the Harris corner detector is over 2,500 times faster, attesting to the potential of the FPGA. At higher resolutions, the speedup is of the same order of magnitude since the FPGA can handle many more tiles without much extra overhead.

The next section discusses findings and results obtained during testing with the FPGA.

5.5.2 Lateral Motion

The first test consisted of lateral motion along the x-axis. First, the software implementation established a measure for baseline performance using the Harris corner detector. During this test, camera motion was restricted to just the x-axis. The Hokuyo laser, attached to the camera, recorded truth measurements. Timestamps were assigned as close as possible to the time of the camera sensor output, and also on arrival of the Hokuyo measurement. Figure 5-13 (Page 66) shows some frames recorded during the experiment. Features that were being tracked are market in red. Some features were successfully tracked during the whole frame sequence or for most of it, for example, the one on the white cable in the middle of the image, or the one at the lower-right corner of the monitor. Figure 5-14 (Page 66) shows the error in the x-direction. The error does not diverge and it is relatively small. The largest deviations are in the middle of the sequence, but in subsequent frames the error is reduced.

This means measurements are being useful and the EKF is successfully updating its belief.

Figure 5-13: This sequence contains frames 1, 10, 16 and 28 of the simulation.

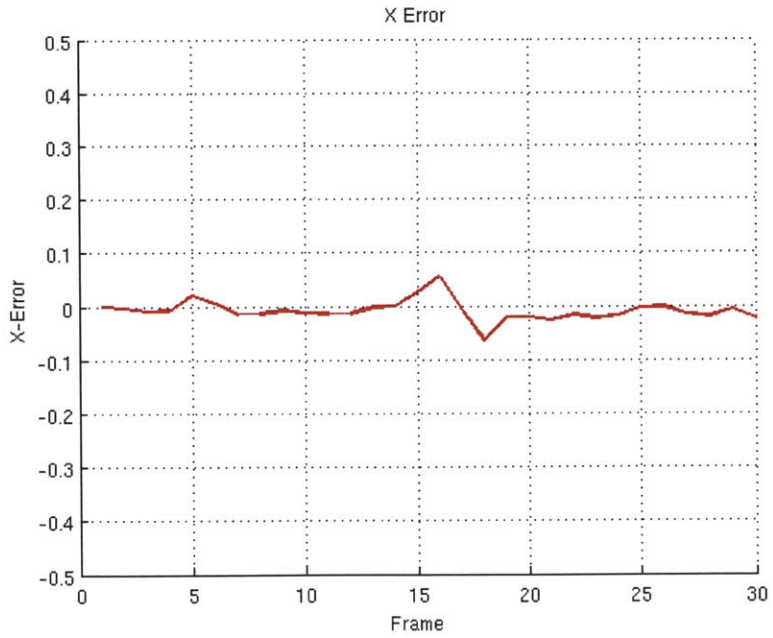

Figure 5-14: This shows the lateral error reading.

FPGA Harris Corner Detector

After establishing baseline performance, the camera ran using the FPGA Harris corner detector during lateral motion. The most important part is that the system implementing the FPGA Harris corner detector worked. However, results were not as expected. Figure 5-15 (Page 68) shows the error in the x-direction. The error does not diverge and it is relatively small. The largest deviations are in the middle of the sequence, but in subsequent frames the error is reduced. This means measurements are being useful and the EKF is successfully updating its belief. shows some images taken during the test with feature points being tracked marked in red.

With each image shown, in this case frames 2, 8, 13 and 20, the number of features being tracked increased. The most likely cause is that features are not being matched effectively in certain frames, so the system resorts to initializing new features when matches to old features are not found. However, some features do show from frame to frame, for example many of the features on the monitor shown in frame 2 continue to be tracked. Since we know from earlier testing that matching given features locations and descriptors works, the most likely cause is in the FPGA. One possibility is that some accuracy in detecting features is being lost due to the approximations that are part of the FPGA compatible Harris corner detector.

Figure 5-16 (Page 68) shows the error measured during the test. Although it diverges initially, it is able to use new measurements to reduce the amount of error between the actual position and the predicted one. The error also supports the claim that some features are being tracked correctly (otherwise the error would just diverge). The spots where error grows correspond to when the system cannot match old features and needs to initialize new ones. Frame numbers with larger error coincide with frames that introduce many new features. This observation comes from comparing the error plot to the sequence of images containing the overlay of tracked features.

Figure 5-15: This sequence contains frames 2, 8, 13 and 20 taken during the test.

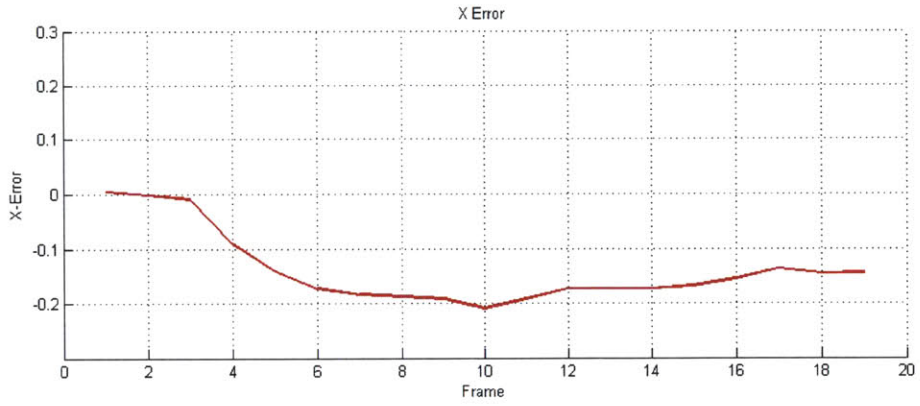

Figure 5-16: Measured error along the x-coordinate.

5.5.3 Different Velocities

Another set of tests consisted of running the system at a different camera translational velocities. However, it turns out moving the camera at different velocities is not as

important as the camera being able to capture small transitions from frame to frame. If in one frame the camera captures a set of features, and then it moves so fast that in the next frame the camera cannot match any of the old features, the system will not yield good results. Also, at higher velocity error can grow faster and faults are evident faster. Figure 5-17 shows how the number of features increases much faster (takes about half the number of frames) compared to the lateral translation case illustrated in Figure 5-18.

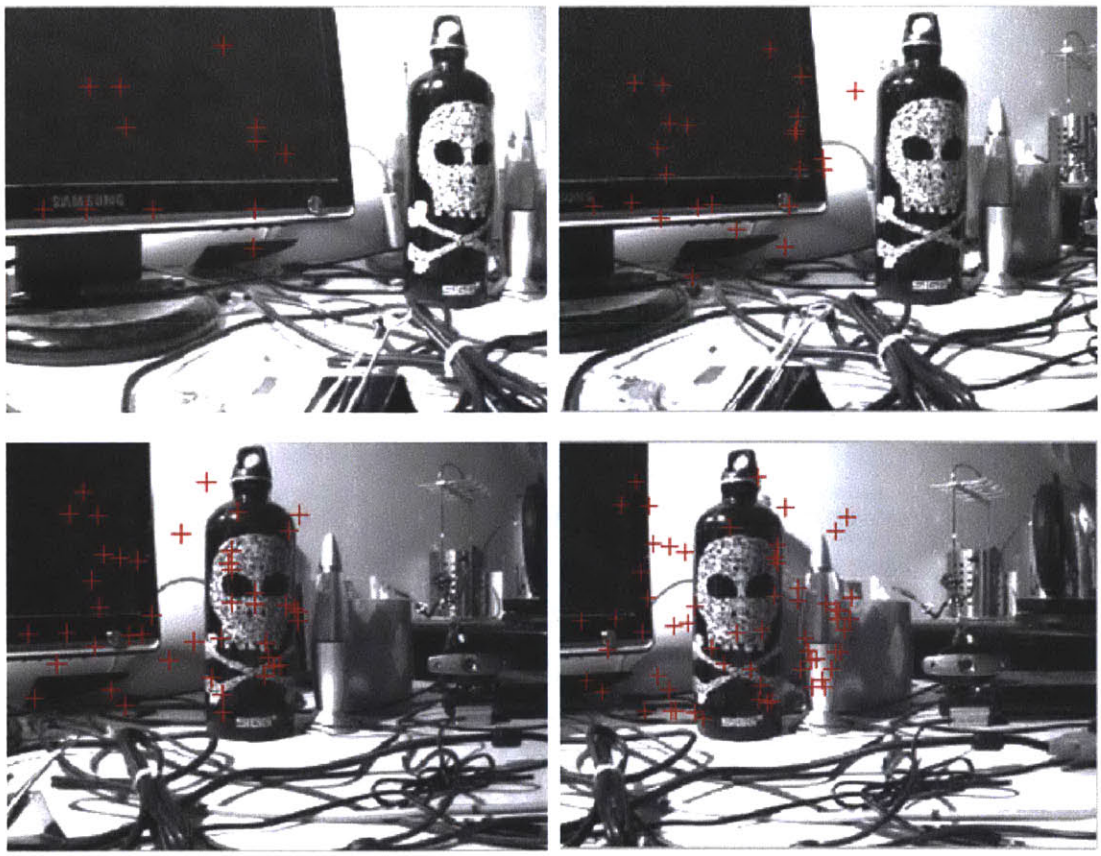

Figure 5-17: This sequence contains frames 2, 4, 5 and 7 taken during the test.

Figure 5-18 shows the error measured during the test. The number of tracked features is not enough to produce accurate estimates, so error begins to diverge.

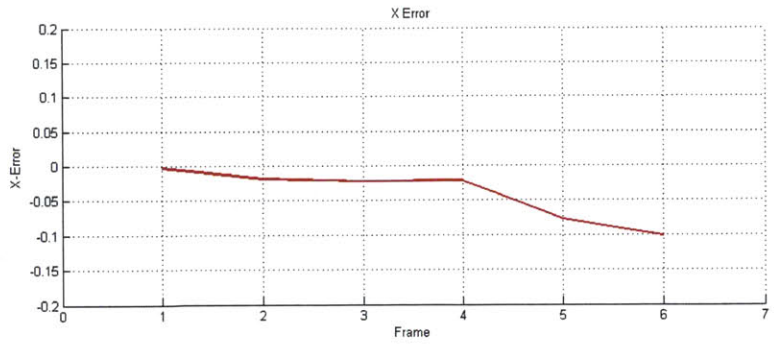

Figure 5-18: Measured error along the x-coordinate.

5.5.4 Forward Motion

Another test using the FPGA was forward motion. Figure 5-19 shows a frame captured during the process. Figure 5-20 (Page 71) shows the associated error. In this

Figure 5-19: Sample frame taken during forward motion showing tracked features.

case error seemed to diverge. This makes sense however, considering the Harris corner detector is not scale invariant, so features are lost as their relative size changes. Red arrows in Figure 5-21 indicate features with infinite depth uncertainty, meaning those features could not be tracked from frame to frame and were left with infinite depth. A higher frame rate however would mitigate this weakness of the Harris corner detector, as changes in scale would be very small and features would be tracked during longer

frame sequences.

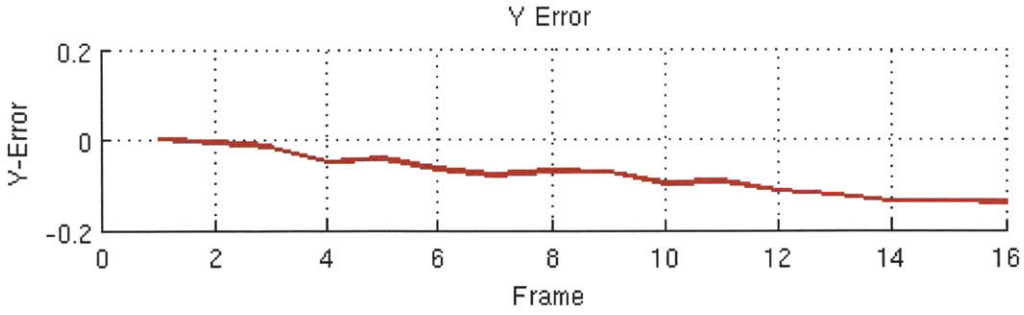

Figure 5-20: Measured error along the y-coordinate (forward direction).

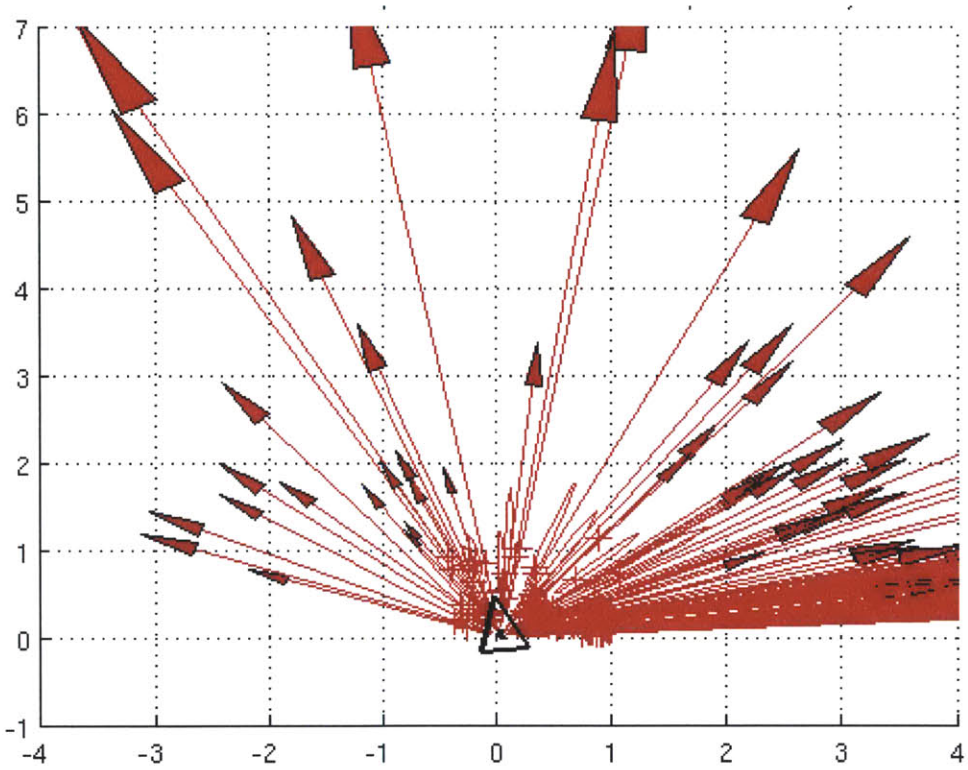

Figure 5-21: Black triangle indicates camera location at frame 20. Red arrows indicate features with infinite depth uncertainty.

5.5.5 Non-Planar Motion

Although non-planar motion performed well on the simulation with the sequence of images, it did not perform the same way during the experiments. One possible reason

is the fact that the image sequence reflects very small transitions between frames, while in the experiments, transitions were more abrupt. With very small differences in orientation, the hardware was able to keep track of features to a limited degree. Another consideration is that the Harris corner detector is not rotation invariant, so possibly a rotation invariant detector such as SIFT would have performed better. As an experiment, I ran a test using the images from the camera with a FAST corner detector, but the results were not satisfactory either. This also supports the theory that the frame rate was not fast enough so that there are only small changes between frames.

5.5.6 Lighting

Running tests under different lighting levels yielded similar results *g*iven that illumination levels did not change. The lens of the camera can be adjusted to let in different amounts of light, allowing for equalization of different lighting conditions. Since the threshold to compare against the Harris corner response is set before running a test, the threshold could be adjusted to yield good results under most reasonable lighting conditions. However, different lighting levels in a sequence of frames posed a problem because the resulted in different intensity patches around features that should have been matched, but consequently could not be determined equal. Camera exposure, which controls the total amount of light that is let into the camera, was another factor related to lighting. Most cameras implement an auto-exposure mechanism, so that the level is adjusted depending on the specific setting. However, changing exposure results in different pixel values being recorded for the same image, which again poses a problem in the matching stage. To prevent this problem, exposure was set once in the beginning and then left unchanged for the rest of the session under similar lighting conditions. One important point is that a bright environment allows for faster exposure, which minimizes the chance of having blur or another distortion in the particular frame.

Bright Lighting vs. Dim Lighting

With exposure locked however, different results were obtained under bright lighting when compared to dim lighting. Figure 5-22 shows the results for both scenarios. Under bright lighting, matches can be more easily found as shown by the convergence of the error when it seems to diverge.

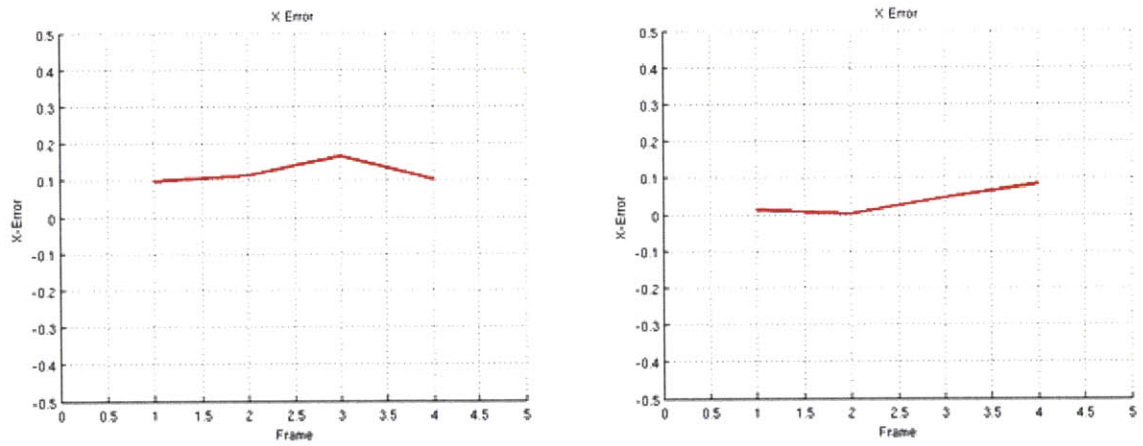

Figure 5-22: Error shown under two different lighting conditions. The figure on the left shows the error under bright lighting, while the figure on the right shows the error under dim lighting.

5.5.7 Limitations

Experiments exposed some of the limitations of the implementation. One bottleneck is in communication. Since the FPGA and CPU are synchronized and the FPGA outputs through the CPU, output is delayed by processing time on the CPU. Currently output is acquired through http requests to the camera, which is relatively slow. An ideal way of communication would be to have direct memory access from the CPU, coupled with LCM running onboard. The main task behind this would be writing a new memory driver for the CPU. Currently, the driver that accesses raw data in memory is slow. Also, initial trials with the embedded image patches did not work as expected, probably because of the small patch size. For testing, patches had to be taken from the images. The main limitation against larger patches was the availabil-

ity of memory (buffers) onboard the FPGA, since buffers are restricted in size. The FPGA also posed limitations, the main one being its available resources. While in theory it seemed like the size would be enough, replicating modules for parallelism and synchronization significantly increases the amount of logic resources required. A big obstacle were the limits on buffers and multipliers (only 28 available multipliers). Another barrier is how the design is synthesized. In a couple iterations of synthesis, the design failed even though requirements were under the limits. This was because the placement process of the logic gates attempts to meet the placement and timing requirement , but in practice, this is not always possible. In one instance the error pointed that buffers had been placed too close to each other, and other placements could not be found.

5.5.8 Bandwidth Reduction

Bandwidth usage was effectively reduced by limiting the camera output. For a resolution of 640×480 308 KB are needed to transfer a frame. If the camera is only outputting features and the number of features is capped at, for example, 500, then the required output is 4 Bytes * 500 = 2 KB (4 Bytes are used to encode a feature's location). If image patches are included, the number increases based on the size of the patch but is still significantly less than the original amount of data.

Chapter 6

Conclusion

The main accomplishment of this thesis is successful integration of a monocular camera, small-factor FPGA, and separate computer system to do motion estimation. The system for feature detection, finding correspondences, filtering and map management also performed successfully. There were cases that did not perform well using the FPGA, such as non-planar motion and relatively high speed transitions, but as more robust feature detectors are ported to the FPGA, these problems should be less evident. At the beginning of this project, the Robust Robotics Group had very limited experience with FPGAs. Now there exists an extensible, modular platform for further research in the area. Furthermore, all the tools used for this research were either free or open-source, which is not that easy in hardware development. The group is also well positioned to take advantage of increases in FPGA capacity. Another drawback was the camera rolling shutter and its sequential access to memory. However, it is just a matter of time until a sensor that enables each photosensor with direct memory access becomes available. Finally, this work is a great example of what can be accomplished when the fields of electrical engineering and computer science are fused into one. Both CSAIL and Draper believe in this interdisciplinary approach, and both labs will without a doubt keep benefiting from pushing these boundaries.

Chapter 7

Future Work

Since this project started as an exploration of the feasibility of combining a camera with an FPGA for image processing, the doors have been opened to a much wider array of possible implementations. The following sections discuss some interesting and particularly useful uses for the FPGA, as well as some considerations that should be taken into account in the future.

7.1 SIFT

Scale-Invariant Feature Transform (SIFT)[20] provides a more robust method (compared to the Harris corner detector) to extract distinctive features. Some of the strengths of SIFT are its ability to deal with different scales, different illumination levels, addition of noise and affine distortion. However, the steps needed to go from an input image to the 128 element descriptor are computationally expensive. Although there are many proposed architectures, analysis by Cabani in [3] estimates needing 230 buffers (2972 Kbits of on-chip RAM) and 344 multipliers (ranging from 9 to 24 bits) just for a multiscale Harris corner detector (which is embedded in SIFT). As a reference, the Spartan3e FPGA used in this project has 28 18×18 bit multipliers and at most 36 dedicated on-chip RAM modules, each with capacity of 18 Kbits [27].

A more viable alternative to harness the power of hardware would be to implement a hybrid design that performs only certain operations on the FPGA. This

project has show gradient computation is highly parallelizable and very efficient in hardware, making implementation of Difference of Gaussians, used in SIFT, a possible implementation. SIFT requires building an image pyramid that is filtered at different resolutions. Filtering is also another highly parallelizable operation on the FPGA, making the pyramid building step another suitable candidate for FPGA implementation.

7.2 Considerations Going Forward

Going forward, there are some key points that should be kept in mind when developing a system like the one in this project. The goals of this project were to develop an FPGA solution that was small (in terms of size of the board, since the main purpose was to be mounted onboard a MAV), fast, and accurate. Going forward, a better approach would be to develop some of the ideas described in this chapter using a full-size development board, that provides the necessary I/O (output pins, LCD screen, push-buttons) for debugging and testing. On the software side, some tools to consider are System Generator, a tool that allows development in Matlab using Simulink and translates the design to hardware; and Bluespec, a package that provides a set of tools to create synthesizable FPGA models.

Bibliography

[1] Abraham Bachrach. Fast and robust scan matcher. `http://code.google.com/p/frsm/`, 2011.

[2] Abraham Bachrach, Ruijie He, and Nicholas Roy. Autonomous flight in unknown indoor environments. *International Journal of Micro Air Vehicles*, 1(4):217–228, December 2009.

[3] Cristina Cabani and W. James MacLean. A proposed pipelined-architecture for fpga-based affine-invariant feature detectors. In *Proceedings of the 2006 Conference on Computer Vision and Pattern Recognition Workshop*, CVPRW '06, pages 121–, Washington, DC, USA, 2006. IEEE Computer Society.

[4] J. Civera, A.J. Davison, and J. Montiel. Inverse depth parametrization for monocular slam. *Robotics, IEEE Transactions on*, 24(5):932 –945, oct. 2008.

[5] Javier Civera, Oscar G. Grasa, Andrew J. Davison, and J. M. M. Montiel. 1-point ransac for extended kalman filtering: Application to real-time structure from motion and visual odometry. *J. Field Robot.*, 27:609–631, September 2010.

[6] Katherine Compton and Scott Hauck. Reconfigurable computing: a survey of systems and software, 2002.

[7] Intel Corporation. Microprocessor quick reference guide. `http://www.intel.com/pressroom/kits/quickreffam.htm`, 2003.

[8] A.J. Davison. Real-time simultaneous localisation and mapping with a single camera. In *Computer Vision, 2003. Proceedings. Ninth IEEE International Conference on*, pages 1403 –1410 vol.2, oct. 2003.

[9] Andrew J. Davison and David W. Murray. Simultaneous localization and map-building using active vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24:865–880, 2002.

[10] Andrew J. Davison, Ian D. Reid, Nicholas D. Molton, and Olivier Stasse. Monoslam: Real-time single camera slam. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29:1052–1067, 2007.

[11] Inc Elphel. Model 353 overview. `http://www.elphel.com/353_overview`, 2011.

[12] H. Farid. Blind inverse gamma correction. *Image Processing, IEEE Transactions on*, 10(10):1428 –1433, oct 2001.

[13] Martin A. Fischler and Robert C. Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM*, 24(6):381–395, June 1981.

[14] David A. Forsyth and Jean Ponce. *Computer Vision: A Modern Approach.* Prentice Hall, us edition, August 2002.

[15] William T. Freeman. 6.865 advanced computational photography lecture 14: Image features. `http://stellar.mit.edu/S/course/6/sp10/6.815/index.html`, April 2010.

[16] C. Harris and M. Stephens. A Combined Corner and Edge Detection. In *Proceedings of The Fourth Alvey Vision Conference*, pages 147–151, 1988.

[17] A.S. Huang, E. Olson, and D.C. Moore. Lcm: Lightweight communications and marshalling. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pages 4057 –4062, oct. 2010.

[18] H. Krupnova and G. Saucier. Fpga technology snapshot: current devices and design tools. In *Rapid System Prototyping, 2000. RSP 2000. Proceedings. 11th International Workshop on*, pages 200 –205, 2000.

[19] Gim Hee Lee, M. Achtelik, F. Fraundorfer, M. Pollefeys, and R. Siegwart. A benchmarking tool for mav visual pose estimation. In *Control Automation Robotics Vision (ICARCV), 2010 11th International Conference on*, pages 1541 –1546, dec. 2010.

[20] David G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60:91–110, 2004.

[21] Bruce D. Lucas and Takeo Kanade. An iterative image registration technique with an application to stereo vision (darpa). In *Proceedings of the 1981 DARPA Image Understanding Workshop*, pages 121–130, April 1981.

[22] R.W. Madison, G.L. Andrews, P.A. DeBitetto, S.A. Rasmussen, and M.S. Bottkol. Vision-aided navigation for small uavs in gps-challenged environments. *AIAA Infotech at Aerospace Conference and Exhibit*, December 2007.

[23] Edwin B. Olson. Real-time correlative scan matching. In *ICRA'09*, pages 4387–4393, 2009.

[24] Cyrill Stachniss, Udo Frese, and Giorgio Grisett. Openslam. `http://openslam.org/`, 2011.

[25] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents).* Intelligent robotics and autonomous agents. The MIT Press, September 2005.

[26] Inc Xilix. Xilinx spartan-3 web power tool version 8.1.01. http://www.xilinx.com/cgi-bin/power_tool/power_Spartan3, 2003.

[27] Inc Xilix. Xilinx spartan-3e fpga family: Data sheet. http://www.xilinx.com/support/documentation/data_sheets/ds312.pdf, 2009.

[28] K. Yonemoto and H. Sumi. A cmos image sensor with a simple fixed-pattern-noise-reduction technology and a hole accumulation diode. *Solid-State Circuits, IEEE Journal of*, 35(12):2038 –2043, dec 2000.

[29] Wonpil Yu. Practical anti-vignetting methods for digital cameras. *Consumer Electronics, IEEE Transactions on*, 50(4):975 – 983, nov. 2004.