SYNCHRONIZATION OF AN INERTIAL MEASUREMENT UNIT AND STANDALONE CAMERA SYSTEM BASED ON ATTITUDE AND OPTICAL FLOW DATA

A THESIS SUBMITTED TO THE GRADUATE DIVISION OF THE UNIVERSITY OF HAWAIʻI AT MĀNOA IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN

MECHANICAL ENGINEERING

AUGUST 2012

By
Lee A. Arnold

Thesis Committee:

Brian Bingham, Chairperson
Scott Miller
Curtis Ikehara

We certify that we have read this thesis and that, in our opinion, it is satisfactory in scope and quality as a thesis for the degree of Master of Science in Mechanical Engineering.

THESIS COMMITTEE

_____
Chairperson

_____

_____

To my great grandmother,

Betty P. Winstedt,

a great woman who never lost the love for learning.

Acknowledgments

I owe everything I am, and everything I have accomplished, to all the people that have helped me along the way. I would like to thank in particular my grandma Gloria, my girlfriend Keisha, and her father Les for all their support throughout the years. I would also like to thank Professor Scott Miller and Professor Curtis Ikehara for their advice and guidance on this thesis. Finally, I would like to thank Professor Brian Bingham for his patience, support, and mentorship over the last few years. Your guidance has shaped my life and it has been a real privilege working in the Field Robotics Lab.

Abstract

The fusion of camera and inertial data is well established and useful for a wide range of tasks. With advancements in camera technology and renewed interest in computer vision, standalone cameras have emerged as a possible research tool. These cameras offer high quality video recording, in a small, lightweight form factor, but are limited by connectivity, thus limiting their usefulness. This thesis presents a software correlation method which correlates inertial data and camera data based on attitude estimates, eliminating the need for additional hardware or connections.

This thesis presents a brief background and description of the correlation process, along with a review of similar work. The underlying methods for feature detection, optical flow, and signal comparison are explained in detail in the context of the software correlation method. The software correlation method is then demonstrated in an experiment that looks at a single axis of rotation. Finally, the error in the software correlation process is quantified and compared to that in an uncorrelated system. Software correlation proves to be a viable option for the correlation of inertial measurement units and standalone cameras.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Chapter 1 introduces this thesis by first looking at the technology which has prompted this work. The synchronization issue is then explained with software correlation a proposed solution. The software correlation method is described, along with the motivation behind this work.

1.1 Background

Camera technology is constantly changing and improving due to consumer demand. Recent interest is focused on small, inexpensive, easy-to-use systems. Standalone units like Pure Digital Technologies' Flip Video and Creative Labs' Vado camcorders introduced cheap, high quality video recording to the public in a small, easy-to-use package. These systems have more recently been replaced by smart phones, such as Apple's iPhone, and specialized systems, like GoPro's HD Hero. While primarily intended for the commercial market, these small standalone systems have found a place in research as well with several groups using the HD Heros in their work [10] [17]. The same features that make these cameras popular with consumers also make them useful for many research projects where similar characteristics are desired. These camera systems may also interest researchers who never before considered a camera system due to cost, size, or complexity.

Along with the advance in camera technology, there has also been a surge of interest in image processing and computer vision. Images contain a vast amount of information but require complex processing to interpret and make use of this information. Consumer electronics like Microsoft's Kinect and projects like the Open Source Computer Vision Library (OpenCV) have moved computer vision capability into the public domain. OpenCV, in particular, allows for user-friendly implementation of computer vision and image processing algorithms with a consolidated software

library of functions. Computer vision tools are now readily available to both researchers and the general public.

1.2 Problem Statement

With the camera technology available, and the means to process such data also available, many projects are now incorporating standalone cameras into their work. As with many sensors, cameras may benefit from sensor fusion, the incorporation of data from multiple sensors to provide better information. Inertial measurement units (IMU) are of particular interest for tasks such as navigation, tracking, and 3D reconstruction. In order to properly fuse data, the data sets must be correlated; the simplest way to accomplish this correlation is to associate precise timing with each image. For IMUs and cameras, each data point from the IMU and each frame from the camera must be defined relative to each other, or relative to another common signal. Hardware can accomplish this task by either timestamping the two data sets, or triggering each sensor at specific instances. This, however, requires additional hardware, increases complexity, and eliminates standalone systems which lack the hardware interface.

This paper examines a software correlation method to correlate IMU and camera data based purely on sensor data from each instrument. No additional hardware is required and neither instrument needs to be hardwired to the other. This method simplifies hardware design and allows the use of a much broader range of sensors, such as standalone camera systems like the HD Hero.

1.3 Approach

The software correlation method relies on predicted sample rates and distinct features within the data to correlate the two data streams. The camera data is processed to determine the optical flow of the background, providing an estimate for the attitude of the camera. To start, the average frame rate of the camera is used to estimate a timestamp for each image frame. This estimate is then periodically updated by matching distinct features in the camera data with similar features in the IMU data. With the IMU and camera fixed relative to each other, both experience similar rotations and thus similar features are present in each data set. Updates are performed periodically based on the desired accuracy of the correlation.

Chapter 5 presents an experiment to demonstrate the software correlation process. Data is captured using a single computer to timestamp the data from the IMU and camera and provide a

ground truth for comparison. Computer vision algorithms are used to process the image data from the camera and determine optical flow, while a matched filter is used for periodic synchronization. The system is limited to only rotation, no translation, to simplify the optical flow calculations and estimation of the camera attitude. The background must also be stationary for proper motion estimation of the camera. Furthermore, rotation is limited to a single axis for simplicity, but the work presented could be easily expanded to include other axes of rotation. While the software correlation method could be applied to "near" real-time applications, the work presented in this thesis focuses on the post processing of data, which is more applicable to standalone systems. While this may limit some applications, it allows for simpler systems and more dedicated computing resources.

1.4 Motivation

Software correlation will enable the use of standalone cameras and IMUs in many situations that were not previously possible. This will allow standalone cameras and IMUs to be fixed to any structure and correlated in post-processing. This has enormous advantages for autonomous systems in which space and weight are at a premium and additional hardware for synchronization is not an option. Small, unmanned, aerial vehicles could fuse IMU and camera data for tracking, mapping, and 3D reconstruction of terrain and structures. Underwater and land based systems could benefit similarly. Any system that requires a camera and benefits from IMU data could apply software correlation. This could include tasks such as surveying, inspection, disaster relief, and numerous other tasks.

Chapter 2

Closely Related Work

Chapter 2 discusses work related to the software correlation method, starting with the fusion of IMU and camera data. The hardware synchronization methods are then explained in more detail, along with some applications. Finally, similar software synchronization techniques are discussed with an emphasis on aspects related to the software correlation method presented in this thesis.

2.1 IMU and Camera Fusion

The fusion of IMU and camera data is a well established technique used in many situations. Visual tracking and stabilization often involves camera motion and knowledge of the camera's motion can aid in such tasks [14] [21] [36] . Navigation and mapping also benefit from camera and IMU integration [26] [16] [27], along with structure from motion [11]. To accomplish any of these tasks, the camera data must be synchronized with all other sensors in use, or timestamped with a time standard.

2.2 Hardware Synchronization

The most common method of sensor correlation involves hardware to timestamp data from each sensor. This can be specific hardware designed for the task, or something more general like a computer. For many autonomous systems, or partially autonomous systems, a processor is usually already being employed and can be used to timestamp all the incoming observations, including imagery and various sensor measurements. This hardware correlation method is common enough that it is not a point of interest in many papers. A few projects do note the time correlation process due to a novel use or because it plays a large role in their work. The ROBOCAST robotic neurosurgery

group has looked at the issue of sensor synchronization in surgery robots and has developed a client-host method using a computer system to synchronize the data [37] . The unmanned surface vehicle ALANIS uses a dedicated system for sensor integration, navigation, and control on an unmanned surface vehicle [6] . While hardware timestamping is the most prevalent method of sensor correlation, it is not the most accurate. Delays in the software can be difficult to quantify and maintain, making hardware timestamps inaccurate.

Another method of synchronization relies on hardware triggers. The trigger can be a number of things, such as a clock or other sensor. Yoon et al. [39] use an external synchronization device to trigger cameras and synchronize other sensors based on a pulse per second (PPS) signal from a GPS unit. In Brown and Sullivan's work [5], the camera is triggered at specified instances and the camera's strobe pulse triggers the GPS to output the time, which is then applied to the image. Furthermore, this approach uses the PPS signal from the GPS to synchronize the IMU data. Chou et al. [8] have developed hardware to trigger various sensors based on a clock circuit pulse. Hardware triggers are in general more accurate at timestamping than the previously discussed hardware timestamping method. Hardware triggers do not necessarily have to interface with any software, eliminating any software related delays. Transmission delays are still present, but may be quantified and are fairly consistent, making them easy to account for.

2.3 Software Synchronization

While both of the hardware methods previously discussed are quite effective, they both require additional centralized hardware specifically for correlation. They also require data to be streamed from each sensor and accessible to the hardware. Another method of synchronization, the method that this paper examines, correlates data in post processing and does not require additional hardware. This thesis looks at synchronizing cameras and IMUs in particular, and relies on common motion detected by the two sensors to correlate the data.

Hwangbo et al. [14] have looked at a similar software correlation method which synchronizes an IMU and camera based on attitude data. With a camera and IMU, they produce a sinusoidal motion in the system which presents itself in the IMU data and image data. Optical flow is used to detect the camera's motion and estimate attitude. They are concerned with the delay in the two signals due to the signals passing through the hardware. They look at only initialization, and do not mention long term synchronization. Both data sets have identical phases with a phase lag equivalent

to the difference in delay between the two signals. They use a Fast Fourier Transform (FFT) to determine the phase lag and compensate for it.

Lee et al. [18] look at the synchronization of an IMU and Vicon pose estimation system. Both systems present similar data but in different reference frames. To synchronize the data, they perform distinct rotations before and after the data of interest. They then do a cross-correlation between the two data sets to determine the time shift between the two and compensate for it. Similar to Hwangbo et al. [14], all data is post processed for correlation.

The software correlation process discussed in this thesis is similar to the software correlation methods previously discussed. Similar to the work by Hwangbo et al. [14], optical flow is used to estimate camera attitude. However, rather than using an FFT to determine phase lag, the IMU and camera attitude data are cross-correlated to determine phase lag, similar to the work by Lee et al. [18].

Chapter 3

Background

Chapter 3 reviews the hardware, software, and techniques relevant to the software correlation process. Inertial measurement units and digital cameras, the two sensors this work is built around, are discussed in detail. Computer vision techniques for feature detection and optical flow are discussed along with signal analysis techniques essential for signal comparison.

3.1 Inertial Measurement Unit

Inertial measurement unit (IMU) is a term used to describe a set of sensors that measure inertial data relative to an inertial frame of reference. These systems typically employ accelerometers for acceleration information, gyroscopes for angular velocity information, and often magnetometers, which provide orientation information in relation to a magnetic field. While technically not an inertia based sensor, magnetometers are useful for heading and attitude determination and are often incorporated into IMUs. IMUs are commonly employed on a wide range of systems that require position or attitude information, including land, sea, air, and space based systems. New microelectromechanical systems (MEMS) technology has made IMUs small, cheap, and rugged, expanding IMU use.

Modern MEMS sensors are far different from their mechanical counterparts and are described in detail in [38]. IMUs typically contain three of each sensor, set orthogonal to each other to provide information in a cartesian coordinate system. MEMS accelerometers typically employ either a test mass and determine displacement, or a vibrating element and determine change in frequency, to measure acceleration. MEMS gyroscopes typically employ some style of vibrating element and determine the Coriolis effect induced by rotation to measure angular velocity.

While describing the operation [38] also discusses the sources and characteristics of error in MEMS inertial sensors. MEMS accelerometers and gyroscopes suffer from several sources of

error, particularly constant bias, thermo-mechanical white noise, flicker noise, temperature effects, and calibration errors. The constant bias is the average output of the sensor while not undergoing any motion and can be determined with a long term average. Thermo-mechanical white noise is present as a high frequency, zero-mean randomness in the data. This produces a zero-mean random walk error in angular measurements integrated from the gyroscope and velocity measurements integrated from the accelerometer. Integrating again produces a second order random walk in position from the accelerometer. Flicker noise is present due to noise within the electronics and causes the bias to wander, affecting bias stability. Over short periods, bias stability can be modeled as a random walk, which produces a second order random walk in angle and a third order random walk in position. MEMS sensors are typically sensitive to temperature due to the nature of their design, and in accelerometers and gyroscopes, temperature fluctuations affect the bias. Calibration error is an all encompassing term referring to error in scale factors, alignments, and linearities in the sensors and can usually be corrected for.

Information from an IMU is typically integrated to provide position and rotation information. This type of position and rotation estimation is known as dead reckoning, where each estimate is based on a previous estimate. Without another measurement for correction, this method is prone to error accumulation. Any error present in an estimate is carried forward and added to future estimates. While this is an important aspect of inertial navigation, it does not affect the work discussed in this paper. The software correlation process is minimally affected by drift because absolute attitude values are not needed as long as the relative "shape" of the data remains the same.

3.2 Digital Camera

Digital cameras are a new variant of their analog (film) counterparts. While both types of cameras work on similar principles of optics, they accomplish the task of saving this information very differently. While film cameras save optical data on a film medium, digital cameras process this data using special electrical light sensors and store the information digitally. Most consumer digital cameras are built around one of two sensor types, either a charge-coupled device (CCD) image sensor or a complementary metal-oxide-semiconductor (CMOS) image sensor. While each device operates differently, with unique pros and cons, both accomplish the same task of saving light information. These image sensors are composed of an array of discrete elements known as pixels (picture elements), each of which convert light energy into an electric signal, which is then converted to a digital representation of the optical information. Pixels determine the resolution of an image and

are also the smallest unit in a photo. Each pixel in an image is an approximation of the optical information gathered by its physical counterpart on the image sensor.

Still images are a snapshot of the state of the image sensor at one particular instant. Video is essentially a sequence of still images captured at a particular frequency. This frequency is known as the frame rate and is expressed as frames per second (fps). At around 24 fps the human eye cannot distinguish between separate frames and a sequence appears smooth. While 24 fps is usually the minimum frame rate of video, it can vary widely depending on the application and capability of the hardware. The hardware required to process and record digital still images and video is very similar, with differences primarily in optimization of either process. Because of this, many modern digital cameras also record video and vice versa.

Whether a camera captures a still image or video, the data is often compressed. Compression is carried out by encoding the raw data into a new data set that occupies a smaller space. The encoding algorithm, or codec, determines the method of encoding. Some cameras are capable of saving the raw data as well, while others only offer compressed images or video. When a still image or video is recorded, the raw data for each frame is captured by the image sensor, stored in a buffer, and then either compressed and saved, or simply saved. This process takes time and limits how many images can be captured in a set time period. Higher resolution images require more time to process than lower resolution images. Video resolution is limited by the time it takes to process each image at a desired frame rate. Cameras capturing still images have much more time to acquire and process higher resolution images than video cameras running at 24 fps or higher.

3.3 Computer Vision

Computer vision is a broad term used to describe the acquisition, processing, and interpretation of optical information by computers. Computer vision has seen renewed interest in recent years with large advancements in computing and imaging. While computer vision is an attempt to encapsulate the function of the human eye and brain, many functions which are arbitrary to humans prove exceedingly difficult for computers. Abstract tasks such as recognition and classification prove challenging, while more tangible tasks such as motion analysis, reconstruction, and interpolation are better handled.

Feature Detection

Feature detection is the computer vision process of detecting points of interest in an image. These

points of interest are often edges, corners, or blobs and help distinguish one object from another. Feature detection is a low level operation and often precedes other computer vision tasks such as object recognition and classification. A wide range of techniques have been proposed to extract various types of features [7] [12] [32] [30] [24] [20] [28] [19].

Corners are a popular feature to extract because they are discrete yet well defined. Similar to a single pixel, a corner can be described by a single point, yet is well defined by the edges that make up the corner. These characteristics also make corners a good feature for motion tracking. A wide range of corner detection techniques have been proposed [25] [12] [32] [30] [35]. Early work by Moravec defined corners by looking at intensity changes in a small region as the region was shifted in different directions [25]. Harris and Stephens built upon and improved Moravec's work by addressing several of the shortcomings in the original work [12]. Shi and Tomasi further refined Harris and Stephens' work by using a similar process but altering the selection criteria [30]. The feature detection process used in this paper is based on the work by Shi and Tomasi. While many of the other proposed methods would surely work, Shi and Tomasi developed their feature detection method in the context of tracking, specifically with affine motion. Having good features for tracking is important since they are used for optical flow calculations in the software correlation process.

Optical Flow

Optical flow is a computer vision technique that looks at the motion in sequences of images. By comparing two or more images, motion vectors can be calculated for various features, determining the movement between frames. Optical flow can not only track features in an image, but also provide information on the movement of the camera. If the apparent motion is induced by camera movement, as opposed to scene movement, the egomotion, or 3D motion, of the camera can be extracted.

Similar to the feature detection methods previously discussed, a wide range of optical flow techniques have been proposed. Barron, Fleet, Beauchemin, and Burkitt divided optical flow techniques into four categories (differential techniques, region-based matching, energy-based matching, phase-based matching) and quantitatively compared various methods [2]. This thesis uses a modified version of the differential technique developed by Lucas and Kanade [23]. Bouguet proposes a pyramidal implementation of Lucas and Kanade's work to expand the tracking window, which allows for larger motions to be tracked [3]. This builds upon the iterative approach to Lucas and Kanade's work which improved accuracy [3]. Bouguet's method utilizes Shi and Tomasi's work for picking good features to track.

3.4 Signal Analysis

Signals are an important aspect in many fields of research and can be generally defined as any time-varying physical quantity [31]. Sinha [31] further notes that signals often vary spatially, as well as temporally, such as with some sensor measurements. Signal analysis is a broad field that looks at the interpretation and processing of these signals, which can be either continuous (analog) or discrete (digital). Signal comparison is one of the many fields that make up signal analysis and plays an important part in the software correlation method discussed in this paper.

Signal Comparison

Signal comparison looks at the similarity between two signals. Several methods can be employed for signal comparison, and they are all related to each other. Convolution and crosscorrelation are two techniques for signal comparison. Both are very similar, but convolution is typically between a signal and filter, while crosscorrelation is usually between two signals [33]. The only difference between the two is that in correlation both indices run in the same direction, while in convolution they run in opposite directions [33]. A signal comparator can be turned into a filter by setting the filter's coefficients to the time-reversed reference signal. This is called a "matched filter". A special type of crosscorrelation, where both signals are the same signal, is called an autocorrelation.

The experiment in Chapter 5 involves periodic crosscorrelation of the IMU and camera data to determine the time lag between the two signals. Templates from the camera data are matched to the IMU data. While crosscorrelation is used to align the two data sets, the term "matched filter" is used to describe the process. The details are essentially the same and will be covered in Section 4.3. The term "matched filter" expresses the process better, in the sense that we are matching a template to a signal. It also distinguishes the matching of the template and signal from the overall goal of correlating the two signals.

Chapter 4

Analytical Methods

Chapter 4 takes a closer look at the techniques described in Chapter 3, particularly those employed in the software correlation process. Shi and Tomasi's feature detection algorithm [30] and Bouguet's optical flow method [3] are described in detail, along with crosscorrelation for signal comparison. Feature detection is further explored in a slightly wider context, looking particularly at variance as a tool to pick distinct features from the camera data for matching.

4.1 Feature Detection

The feature detection method discussed in this section incorporates the work by Shi and Tomasi [30]. Shi and Tomasi look at not only feature detection, but feature detection in the context of tracking, specifically with affine motion. They put forth, "a good feature is one that can be tracked well" [30]. Similar to the work by Harris and Stephens [12], Shi and Tomasi look at a region around each pixel and judge it based on the gradient of the image intensity. By taking the derivative of the image intensity in orthogonal directions and composing these derivatives into a spatial gradient matrix, they propose that good regions for tracking can be picked based on the eigenvalues of the gradient matrix.

For each pixel, a region around the pixel is examined. This region is sometimes referred to as the neighborhood, and can be any shape. The gradient of the image intensity I, for the rectangular neighborhood defined by x and y, is given in Equation 4.1.1.

$$\nabla I = \frac{\partial I}{\partial x}\hat{\imath} + \frac{\partial I}{\partial y}\hat{\jmath} \tag{4.1.1}$$

To determine the gradient, the partial derivative of the image intensity is taken along the x and y axis. Typically, the image region is also smoothed to reduce noise. A Sobel filter accomplishes both

of these tasks. Sobel filters convolve a mask, or kernel, with the image to simultaneously smooth and calculate the gradient [29]. The filter is broken into two separate masks G_x and G_y, with

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$$

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} = \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 1 \end{bmatrix}$$

for the x-axis and y-axis respectively.

The Sobel masks can be thought of as composed of two separate masks, one for smoothing and one for differentiating. The smoothing mask weights the center pixel higher than the surrounding pixels, while the differentiator emphasizes change. The Sobel filter results in a scaled value for the gradient, which may or may not be corrected. Each mask is convolved with the image to determine the image gradient

$$I_x = \frac{\partial I}{\partial x} = G_x \otimes I$$
$$I_y = \frac{\partial I}{\partial y} = G_y \otimes I$$

for the respective axis. The $\otimes$ symbol denotes the convolution operation. The gradients for each axis are then organized into a spatial gradient matrix M, where

$$M = \begin{bmatrix} {I_x}^2 & I_x I_y \\ I_x I_y & {I_y}^2 \end{bmatrix}.$$

From this gradient matrix, the eigenvalues are calculated and used to determine the "goodness" of the region. For a 2×2 matrix A, with

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix},$$

the eigenvalues λ_1 and λ_2 are calculated such that

$$\lambda_1, \lambda_2 = \frac{1}{2}\left((a+d) \pm \sqrt{4bc + (a-d)^2}\right).$$

Shi and Tomasi propose that these eigenvalues are indicative of the "goodness" for a given region. Two small eigenvalues mean a roughly constant intensity profile within the window, while a large and small eigenvalue indicates a unidirectional texture, and two large eigenvalues could indicate corners, salt-and-pepper textures, or any other "good" pattern. They also note that a minimum eigenvalue is the only parameter that must be specified. If both eigenvalues are large enough to meet the noise requirement, then the system is usually well-conditioned as well since the eigenvalues have an upper bound limited by the maximum allowable pixel value. For a specified window, if the eigenvalues are above a specified threshold, it contains good features to track.

The experiment discussed later in Chapter 5 utilizes OpenCV function *goodFeaturesToTrack* to detect features for tracking. The function *goodFeaturesToTrack* utilitzes Shi and Tomasi's method of feature detection previously discussed.

goodFeaturesToTrack [1]

The *goodFeaturesToTrack* function is part of the OpenCV library and is used in the software correlation process. Appendix A.1 provides documentation of the function from the OpenCV documentation [1]. The function utilizes the Shi and Tomasi method previously discussed to determine good features for tracking.

The function *goodFeaturesToTrack* looks for strong corners in an image and writes their coordinates to an array. The user specifies the number of points to track, the minimum distance between each point, and the size of the search window. Features are judged based on an input "qualityLevel" parameter which is a function of the minimum eigenvalue. The gradient for each window is calculated using a Sobel filter and composed into a spatial gradient matrix using another OpenCV function, *cornerEigenValsAndVecs*, described in Appendix A.2.

4.2 Optical Flow

The optical flow method discussed in this section is based on the work by Bouguet [3]. Bouguet proposes a pyramidal implementation of an iterative Lucas Kanade tracking algorithm. The iterative approach increases accuracy, while the pyramidal implementation allows for larger tracking windows, increasing the robustness of the algorithm with respect to tracking larger displacements.

The goal of the optical flow process is to determine a vector, which, when added to the original image, minimizes the error between the first and second image. Without prior knowledge, this vector must lie within the search region of the algorithm. The pyramidal approach first examines

a small portion of the image to estimate the displacement, which is then used as an initial guess in progressively larger regions until the entire image has been examined. At each level of the pyramidal approach, the optical flow is calculated using an iterative Lucas-Kanade algorithm. The Lucas-Kanade method looks for a vector to minimize the error between the two sequential images (or portions of the image). The iterative approach involves calculating an estimate, taking the residual, and then recalculating an estimate, repeating until the residual is smaller than a set threshold. The final optical flow vector is passed to the next level in the pyramid as an initial guess. Features to track are chosen based on Shi and Tomasi's method discussed in Section 4.1. The iterative Lucas-Kanade approach increases accuracy at each level, while the pyramidal approach allows for larger search regions, increasing the overall robustness of the optical flow calculation.

For an image I of size $n_x \times n_y$ a set of levels are defined recursively, with each subsequent level having dimensions

$$n_x^L \leq \frac{n_x^{L-1}+1}{2}$$

$$n_y^L \leq \frac{n_y^{L-1}+1}{2}$$

where L is the highest level, typically no higher than 4, and n^L are the largest integers that satisfy each equation. For example, for the 640×480 pixel image shown in Figure 4.1, the images I^1, I^2, and I^3 are 320×240, 160×120, and 80×60 pixels respectively.

For each level, starting at the highest (smallest), an integration window, typically 4, 6, 8, 10, 12, 14 pixels square, is moved through the image. This integration window is compared with the equivalent window in the next image of the sequence J. The Shi and Tomasi method previously discussed in Section 4.1 is used to pick features for tracking. These features are used in an iterative Lucas-Kanade algorithm to calculate optical flow.

The goal is to find the residual pixel displacement vector $\mathbf{d}^L$ that minimizes the matching error function ϵ^L at each level

$$\epsilon^L(\mathbf{d}^L) = \epsilon^L(d_x^L, d_y^L) = \sum_{x=u_x^L-w_x}^{u_x^L+w_x} \sum_{y=u_y^L-w_y}^{u_y^L+w_y} \left(I^L(x,y) - J^L(x+g_x^L+d_x^L, y+g_y^L+d_y^L)\right)^2 \tag{4.2.1}$$

where x and y are the coordinates for the center of the integration window, u is the equivalent point in the image for level L, w is half the size of the integration window, I is the primary image, J is the

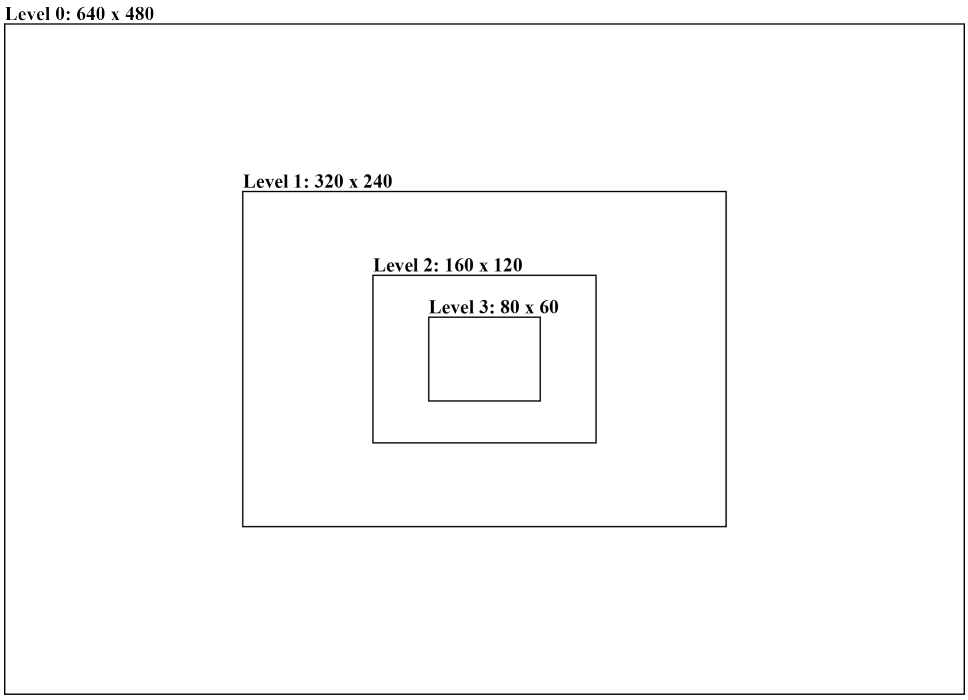

Figure 4.1. Example depicting 3 pyramid levels for a 640×480 pixel image. Calculations start at the highest level, level 3 in this case, and work towards level 0, the original image.

next image in the sequence, and g is an initial guess for the optical flow. For a single level, Equation 4.2.1 can be rewritten as

$$\epsilon(\bar{\nu}) = \epsilon(\nu_x, \nu_y) = \sum_{x=p_x-w_x}^{p_x+w_x} \sum_{y=p_y-w_y}^{p_y+w_y} (A(x,y) - B(x+\nu_x, y+\nu_y))^2 \tag{4.2.2}$$

where

$$\begin{aligned}
\bar{\nu} &= [\nu_x \quad \nu_y]^T = \mathbf{d}^L \\
\mathbf{p} &= [p_x \quad p_y]^T = \mathbf{u}^L \\
A(x,y) &= I^L(x,y) \\
B(x,y) &= J^L(x+g_x^L, y+g_y^L).
\end{aligned}$$

With an optimum value for $\bar{\nu}$, the first derivative of ϵ with respect to $\bar{\nu}$ is zero, and after expansion of the derivative

$$\frac{\partial \epsilon(\bar{\nu})}{\partial \bar{\nu}} = -2 \sum_{x=p_x-w_x}^{p_x+w_x} \sum_{y=p_y-w_y}^{p_y+w_y} (A(x,y) - B(x+\nu_x, y+\nu_y)) \cdot \begin{bmatrix} \frac{\partial B}{\partial x} & \frac{\partial B}{\partial y} \end{bmatrix} \tag{4.2.3}$$

For a small displacement vector, $B(x+\nu_x, y+\nu_y)$ can be replaced by its first order Taylor expansion about $\bar{\nu} = [0 \quad 0]^T$, giving

$$\frac{\partial \epsilon(\bar{\nu})}{\partial \bar{\nu}} \approx -2 \sum_{x=p_x-w_x}^{p_x+w_x} \sum_{y=p_y-w_y}^{p_y+w_y} \left(A(x,y) - B(x,y) - \begin{bmatrix} \frac{\partial B}{\partial x} & \frac{\partial B}{\partial y} \end{bmatrix} \bar{\nu} \right) \cdot \begin{bmatrix} \frac{\partial B}{\partial x} & \frac{\partial B}{\partial y} \end{bmatrix} \tag{4.2.4}$$

The difference between images A and B can be interpreted as the temporal image derivative and calculated with a Sobel or Sharr operator as previously discussed. Equation 4.2.4 becomes

$$\frac{1}{2} \frac{\partial \epsilon(\bar{\nu})}{\partial \bar{\nu}} \approx \sum_{x=p_x-w_x}^{p_x+w_x} \sum_{y=p_y-w_y}^{p_y+w_y} \left(I^T \bar{\nu} - \delta I \right) I^T \tag{4.2.5}$$

with $\delta I(x,y) = A(x,y) - B(x,y)$. Multiplying out the terms gives

$$\frac{1}{2} \left[\frac{\partial \epsilon(\bar{\nu})}{\partial \bar{\nu}} \right]^T \approx \sum_{x=p_x-w_x}^{p_x+w_x} \sum_{y=p_y-w_y}^{p_y+w_y} \left(\begin{bmatrix} {I_x}^2 & I_x I_y \\ I_x I_y & {I_y}^2 \end{bmatrix} \bar{\nu} - \begin{bmatrix} \partial I I_x \\ \partial I I_y \end{bmatrix} \right) \tag{4.2.6}$$

with

$$G = \sum_{x=p_x-w_x}^{p_x+w_x} \sum_{y=p_y-w_y}^{p_y+w_y} \begin{bmatrix} {I_x}^2 & I_x I_y \\ I_x I_y & {I_y}^2 \end{bmatrix} \quad and \quad \bar{b} = \sum_{x=p_x-w_x}^{p_x+w_x} \sum_{y=p_y-w_y}^{p_y+w_y} \begin{bmatrix} \partial I I_x \\ \partial I I_y \end{bmatrix} \tag{4.2.7}$$

Equation 4.2.6 becomes

$$\frac{1}{2} \left[\frac{\partial \epsilon(\bar{\nu})}{\partial \bar{\nu}} \right]^T \approx G\bar{\nu} - \bar{b} \tag{4.2.8}$$

with the derivative going to zero at the optimum optical flow vector

$$\bar{\nu}_{opt} = G^{-1}\bar{b} \tag{4.2.9}$$

As for the iterative aspect, $\bar{\nu}$ is calculated once, then used as a guess for the next iteration, and so on. The residual tends towards zero, and once a certain threshold or number of iterations is met, the cycle is stopped. The final value of $\bar{\nu}$ is the optical flow vector for this particular feature, at this particular level. This value is then used in the next level as an initial guess.

The experiment discussed later in Chapter 5 uses another OpenCV function for calculating the optical flow. The *calcOpticalFlowPyrLK* function implements the work by Bouguet to track features using an iterative Lucas-Kanade method with pyramids.

calcOpticalFlowPyrLK [1]

calcOpticalFlowPyrLK tracks features between a sequence of images. Features to track are specified for the first image, and may also be specified for the second as an initial guess. The image is set up in a pyramid fashion and each feature is searched for in a specified search window. The user defines the max number of iterations and threshold value for the residual. The *calcOpticalFlowPyrLK* function outputs the location of the points in the second image which can then be used to determine the optical flow based on displacement. Appendix A.3 provides the OpenCV documentation for *calcOpticalFlowPyrLK*.

4.3 Signal Comparison

Signal comparison is an important aspect of the software correlation method. To synchronize the two data streams, features in the camera data must be periodically matched to the IMU data. This section looks at the similarities between crosscorrelation, convolution, and the matched filter and explains how these techniques are used for signal comparison.

Crosscorrelation is defined, for continuous variables, as

$$(f \star g)(t) = \int_{-\infty}^{\infty} f^*(\tau) g(t+\tau) d\tau \tag{4.3.1}$$

or, for discrete variables, as

$$(f \star g)[n] = \sum_{m=-\infty}^{\infty} f^*[m] g[n+m] \tag{4.3.2}$$

where $f(t)$ and $g(t)$ are the signals of interest, $\star$ denotes the crosscorrelation operation, and * denotes the complex conjugate.

Convolution is defined, for continuous variables, as

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau) g(t-\tau) d\tau \tag{4.3.3}$$

or, for discrete variables, as

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m] g[n-m] \tag{4.3.4}$$

where $f(t)$ and $g(t)$ are the signals of interest, and $*$ denotes the convolution operation. Crosscorrelation and convolution are very similar, as previously mentioned, and can be related

$$(f \star g)(t) = f^*(-t) * g(t) \tag{4.3.5}$$

Autocorrelation is a special case of crosscorrelation, where a signal is compared with itself. There will always be a peak at a time lag of zero. This fact is used in the software correlation process described in Chapter 5. While technically the two signals are distinct, the similarity of the two are used to correlate the signals.

Matched filters accomplish a similar task as crosscorrelation. Matched filters are the optimal signal detector, designed to detect signals in the presence of noise [33]. Similar to crosscorrelation, matched filters can be used to match a template signal to a reference signal. The coefficients for a matched filter are the template, or reference signal, reversed in time [33]. While the calculations in Chapter 5 are based on cross-correlation, the term "matched filter" is used throughout this paper to distinguish the template matching from the overall goal of correlating the IMU and camera data.

4.4 Variance

Feature detection has been discussed in the context of computer vision, but it is a field which spans many areas of interest. The challenge of picking distinct or unique features, and distinguishing these features from others, can be challenging. The software correlation process discussed in Chapter 5 uses variance to detect distinct features in the camera data for matching with the IMU data. Knowing that the system is relatively steady with periodic rotations, the variance can be used to locate features. A moving variance was used to determine the template region.

The population variance is defined as

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2 \tag{4.4.1}$$

where the mean μ is defined as

$$\mu = \frac{1}{N}\sum_{i=1}^{N} x_i \tag{4.4.2}$$

The sample variance is found by looking at a smaller subset of the overall population. The moving variance is essentially the sample variance calculated over a window which moves through the data. Equation 4.4.3 shows how the moving variance may be calculated as

$$s_m^2 = \frac{1}{n} \sum_{i=m-\frac{n}{2}}^{m+\frac{n}{2}} (y_i - \bar{y})^2 \tag{4.4.3}$$

where n is the window size, m is the index, and the total population size can be defined as N. The moving variance is calculated for each point m from 0 to N. For indexes where the window extends beyond the range of the data, the variance is held constant. It is equivalent to the next closest point in which the window is within range of the data. Indexes 0 to $\frac{n}{2}$ will have the same variance as calculated for point $\frac{n}{2}$ and indexes $N - \frac{n}{2}$ to N will have the same variance as that calculated at $N - \frac{n}{2}$.

Chapter 5

Experimental Results

Chapter 5 presents an experiment to demonstrate the software correlation of IMU and camera data based on attitude estimation. The experimental setup is discussed in detail along with specifics for the hardware and software used. Raw data from the IMU and camera is presented and explained, along with processed optical flow data from the camera. The correlation process is explained in detail and demonstrated using the IMU data and processed image data.

5.1 Experimental Setup

The experiment required several pieces of hardware and software. A Logitech C910 webcam was used to capture optical data, while a Microstrain 3DM-GX2 IMU was used to capture inertial data. Both sensors were connected to a 13" Apple MacBook Air (3,2), which recorded data from the camera and IMU. Separate programs were used to log data from each sensor to reduce latency. One program, written in C++, logged IMU data at approximately 100 Hz and timestamped the data after each sample was received by the computer. Another program, written in C++, utilizing the OpenCV library, was used to capture, save, and timestamp each frame retrieved from the camera at approximately 17 Hz. Camera data was then processed for optical flow using another C++ program and the OpenCV library. This data, along with the IMU data, was then imported into MATLAB to examine the correlation of the two data streams. While a wired camera is needed to conduct the experiment, a standalone camera system like the GoPro HD Hero is a better representation of the systems this work targets; however, using a wired camera allowed for a common timestamp between the camera and the IMU which served as a ground truth for evaluating the experimental results. The characteristics of the HD Hero are examined and integrated into the results in Chapter 6.

The IMU and camera were attached to a rigid piece of rectangular aluminum tubing. The tubing ensured that both sensors experienced similar rotation but remained fixed relative to each other. The tubing, along with the IMU and camera, was then mounted to a tripod to provide smooth rotation and steadiness between rotations. Two pieces of wooden dowel were attached to the tripod head to provide stops, allowing for consistent minimum and maximum rotation values of about $-26°$ and $18.5°$. Figure 5.1 shows the entire setup used in the experiment with the tripod, camera, IMU, and stops.

To simplify the processing of the camera data, only a single axis of rotation, corresponding to pitch in the IMU and camera frame, was examined. IMU and camera data was captured for approximately 45 minutes for a total of 45879 image frames and 268538 IMU data points. Lighting was kept fairly constant throughout the experiment to minimize changes in camera shutter speed which could affect frame rate. The scene was well lit to increase brightness and contrast in the image, which reduces acquisition time and compression time respectively. The system was rotated periodically to provide distinct features in the data to match. There were a total of 10 features, each spaced approximately 5 minutes apart.

5.2 Camera and IMU Specifics

Microstrain 3DM-GX2

The 3DM-GX2 is a 6 degree of freedom IMU produced by Microstrain and was used to capture inertial data during testing. The 3DM-GX2 contains three orthogonal accelerometers, three orthogonal gyroscopes, and three orthogonal magnetometers, with all sensors built using MEMS technology. This particular IMU also contains four thermometers for temperature compensation. The IMU is capable of output speeds up to 300 Hz over a USB wire. The 3DM-GX2 can output acceleration, angular rate, change in angle, change in velocity, Euler angles, a rotation matrix, or raw magnetometer readings. Orientation range and further specifics are shown in Table 5.1.

Logitech C910

The C910 is a high definition USB webcam produced by Logitech. The camera utilizes a CMOS image sensor coupled with a $78°$ wide angle lens. Frame rates for the camera are handled at the driver level, with the driver translating codec, frame rate, and resolution requests to the hardware. Table 5.2 lists the capable frame rates of the camera for specific encodings and resolutions. Ini-

Figure 5.1. Experimental setup consisting of tripod, camera, and IMU. Stops were placed to limit rotation and provide consistent minimum and maximum values.

Roll	$\pm 180°$
Pitch	$\pm 70°$
Yaw	$\pm 180°$
Resolution	$< 0.1°$ minimum
Repeatability	$0.20°$
Accuracy	$\pm 0.5°$ static $\pm 2.0°$ dynamic

Table 5.1. Attitude and performance specifics for the Microstrain 3DM-GX2 IMU

tial testing uncovered driver problems and delay issues which resulted in an average frame rate of approximately 17 fps at 640×480 resolution.

Driver issues prevented optimal performance of the C910. While Logitech provides drivers for the C910, they do not allow access to frame rate and codec controls. Software provided by Logitech allows basic control of the camera, such as capturing video, setting brightness, etc. but does not address the issue of frame rate and codec control. The C910 is a USB video class (UVC) device that is supported by any machine that supports the UVC class. While Apple

Codec	Frame rate	Resolution
MJPEG	60 fps	640x480
MJPEG	30 fps	1280x720
MJPEG	30 fps	1920x1080
RGB3	60 fps	640x480
BGR3	60 fps	640x480
YU12	60 fps	640x480
YV12	60 fps	640x480

Table 5.2. Possible frame rates for the specified resolution and codec for the Logitech C910 webcam.

Macintosh computers support the UVC class, the C910 provides minimal access to device specifics through the UVC class. The UVC class allowed image frames to be captured, but the default settings of the camera could not be changed.

While driver issues prevented access to the C910 settings, other factors may also have limited performance. Each function in the camera logging software has delays associated with it which limit the overall performance of the software. The applicable functions in the software were tested for delays using a technique similar to the timestamping process discussed in Section 5.3. Tests show that capturing an image from the camera takes approximately 1.2 milliseconds, while saving the image with OpenCV's *imwrite* function takes about 26 milliseconds. These delays, along with other minor delays in the software, limit the maximum frame rate this logging software can handle to about 30 fps. While frame rates were closer to 17 fps, the optimal 60 fps seems unattainable with this software.

Without access to the C910 settings and without knowing all the sources of delays, it is impossible to estimate the frame rate of the camera. Since an average frame rate is needed in the synchronization process discussed later in Section 5.4, a test was conducted to measure the average frame rate of the C910. The camera was fixed in a well lit area and recorded data for approximately one hour. Each frame was timestamped and then processed to determine the instantaneous frame rate, mean frame rate, and standard deviation in frame rate. Figure 5.2 shows the instantaneous frame rate of the camera, along with the average for the duration of the test. For the software and conditions, the C910 has an average frame rate of 17.1023 fps with a standard deviation of 2.7465 fps.

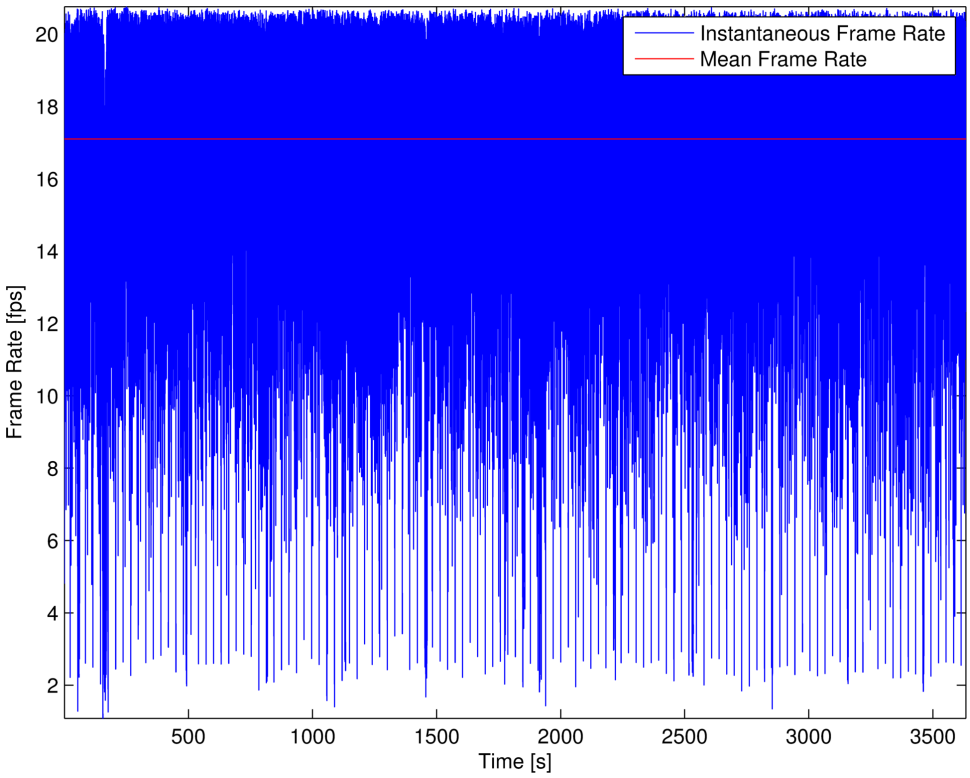

Figure 5.2. Measured instantaneous frame rate for the C910, along with the mean frame rate. The frame rate varies considerably which makes the mean unreliable for timestamp estimation.

GoPro HD Hero

The GoPro HD Hero is a simple, small, standalone camera system. The HD Hero is designed for continuous recording or picture taking with little interaction required and is capable of several hours of standalone recording to an SD card slot. The camera has a 170° field of view and is capable of recording 1280x720 resolution video at 59.940076 fps, and 1920x1080 resolution video at 29.970030 fps. The HD Hero has a limited interface with no method of external hardware synchronization.

Similar to the frame rate test of the C910, the HD Hero was tested over a period of time to determine the variability in frame rate. Unlike the C910, the HD Hero was set to record a moving background with a clock in view. The moving background forces each frame to be encoded independently, with no information saved from previous frames. This presents a difficult setting for the camera and really tests the accuracy of the frame rate estimate. The clock recorded in the video provides a reference time to evaluate the frame rate. After 4278 seconds, the HD Hero recorded 256501 frames at 1280x720 resolution using the h.264 codec. This gives an average frame rate of 59.958158 fps for the HD Hero, 0.018082 fps more than the published value of 59.940076 frames per second.

5.3 IMU and Camera Measurements

IMU Data

Data was captured from the Microstrain 3DM-GX2 IMU using custom software written in C++. The IMU was allowed to warmup for 15 minutes before capturing the data at 100 Hz for a total count of 268538 samples, roughly 45 minutes of runtime. Roll, pitch, and yaw were logged along with a computer based timestamp. Although all three Euler angles were recorded, only pitch will be used, since this work only looks at rotation about a single axis.

Timestamps were applied to each data point based on the Apple Macintosh function *mach_absolute_time*, which returns the number of processor ticks since startup. By timestamping each data point using *mach_absolute_time* and comparing this value to an arbitrary count taken before the experiment, an accurate timestamp based on the processor can be applied. This timestamp can then be converted to seconds based on processor information and still retains the accuracy of the processor clock. Any software running on the same computer, using the same count taken before the experiment, can accurately apply a relative timestamp.

The 3DM-GX2 has an internal clock to regulate operations, but the software logging IMU data is running on a computer alongside many other processes. Each of these processes is fighting for CPU time which can cause bottlenecks and affect performance. This is an issue unique to the experiment and is not a factor in standalone systems. To reduce the affects of other processes, any unnecessary functions were killed before the experiment. The IMU logging software was then started using the Unix "nice" command with a -20 priority. This gives the software highest priority access to CPU resources, preventing slowdowns caused by lower priority processes.

Giving the IMU logging software higher priority helps but does not completely eliminate delays. Other software with similar priority, and essential processes with higher priority, are still competing for system resources. Figure 5.3 shows the delay between each sample from the IMU. Figure 5.4 shows a magnified view of Figure 5.3. While there are a few large spikes, the majority of the delays are 0.01 seconds in duration, consistent with a 100 Hz sample rate, along with a significant number of 0.02 second delays. The large spikes may be caused by other essential processes kicking in, while the periodic 0.02 second delays are most likely caused by other processes with similar priority, possibly the camera logging software which is discussed later. These delays are important because they affect the estimated sample rate of the IMU and can affect the interpolation required to match the IMU and camera data.

Raw data captured during the experiment was imported into MATLAB for further processing. Figure 5.5 shows the pitch data recorded during the experiment. There are 10 spikes, each one corresponding to intentional rotations of the IMU and camera system on the tripod. These features will be matched with the camera data to synchronize the two data streams as part of the correlation process. Figure 5.13 shows a magnified view of each feature in the IMU and camera data. These features do not need to be unique, but they must be distinct, in order to be matched.

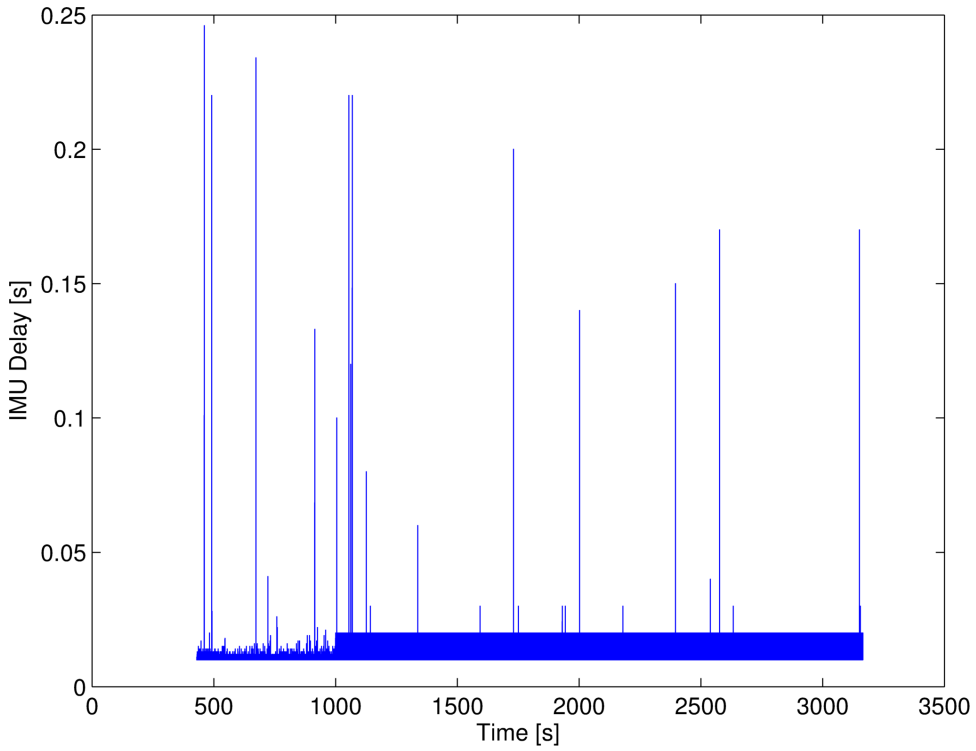

Figure 5.3. Time delays between IMU data points calculated using the applied timestamps.

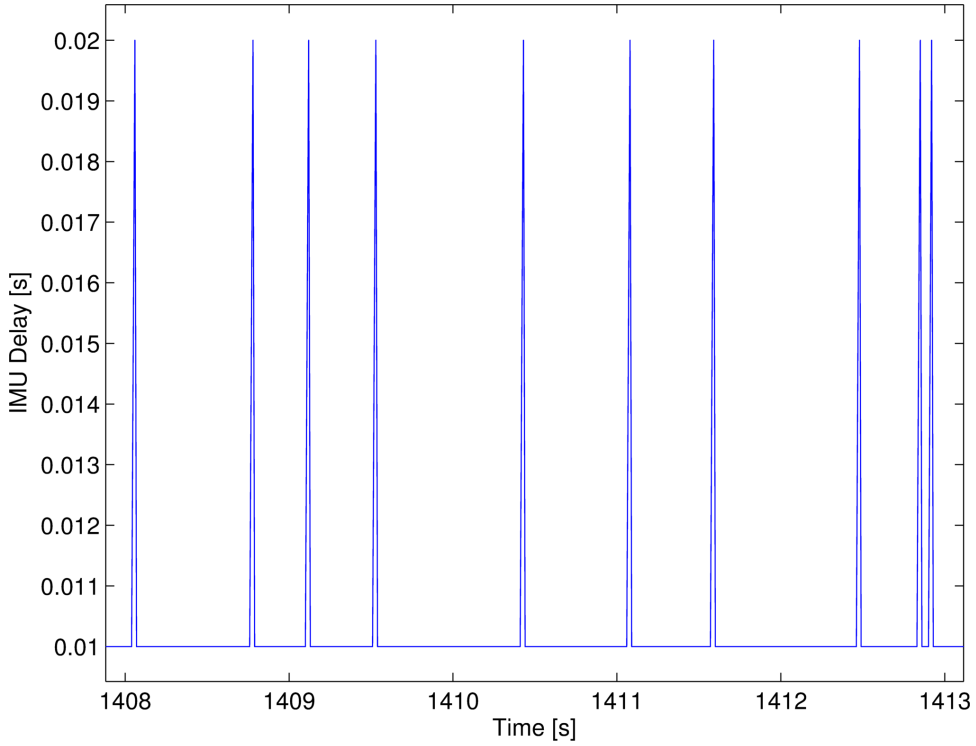

Figure 5.4. An enlarged view of the IMU time delays showing consistent spikes of 0.02 seconds.

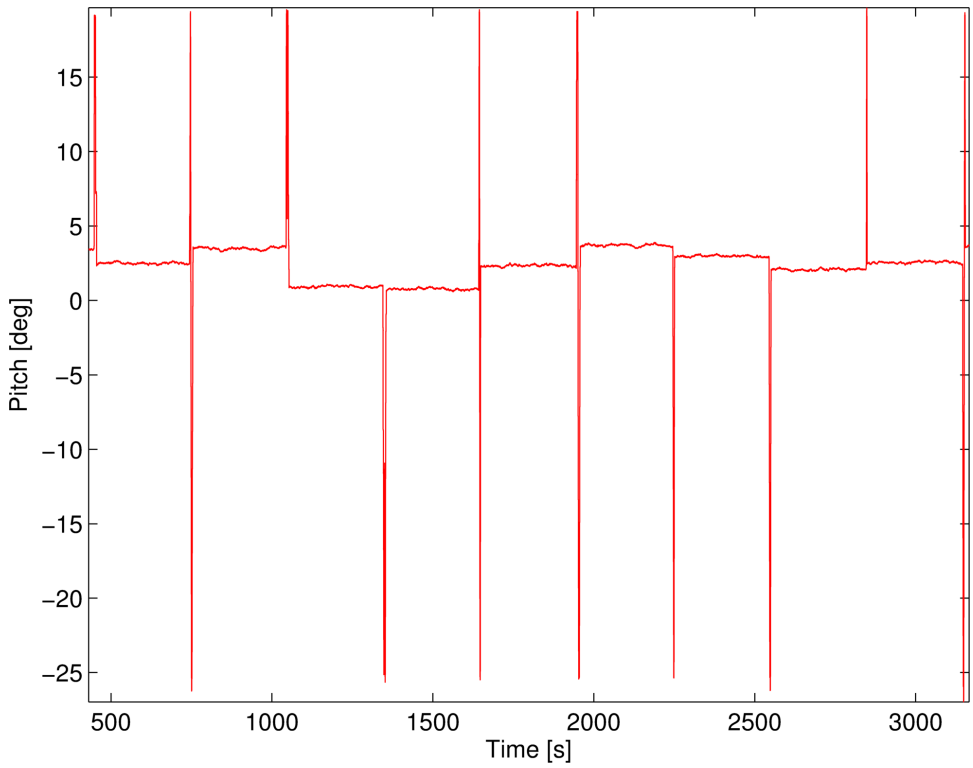

Figure 5.5. IMU pitch data from the experiment. Spikes indicate features created by rotating the system. Minimum and maximum values are consistent due to the stops placed to limit rotation.

Camera Data

Image data from the Logitech C910 webcam was captured using custom software utilizing the OpenCV library and written in C++. Each frame was saved using OpenCV's *imwrite* function and not displayed to reduce processor load. Image frames were captured at 640×480 resolution at approximately 17 fps for about 45 minutes, resulting in a total of 45879 images.

As with the IMU data, a timestamp was applied to each image frame using the Apple Macintosh *mach_absolute_time* function. By using the same reference count taken before the experiment, an accurate timestamp relative to the IMU data can be applied to each frame. *mach_absolute_time* was called immediately before each request for an image in the hope of reducing transmission delays. Each count was then compared to the reference value, converted to seconds, and applied as a timestamp to the image. Similar to the IMU logging software, the camera logging software was started using the Unix "nice" command with a -20 priority in an attempt to limit slowdowns.

Similar to the IMU data, the camera data also experienced inconsistent delays between image frames. While the camera logging software is given the highest priority, other system tasks may still interfere. Figure 5.6 shows the delays between each frame for the duration of the experiment. Figure 5.7 and Figure 5.8 show enlarged portions of Figure 5.6. The majority of the delays are 0.06 seconds long with fairly consistent 0.01 second fluctuations. There are also fairly consistent spikes of about 0.24 seconds spaced roughly 30 seconds apart. The smaller fluctuations may be camera specific, related to hardware limitations, or possibly influenced by the IMU logging software. The larger spikes may be related to the computer hardware used to log the IMU and camera data. With so much data coming in through the camera logging software, the computer may pause periodically to process the data if it has been stored in a buffer. This would account for periodic, longer than usual delays.

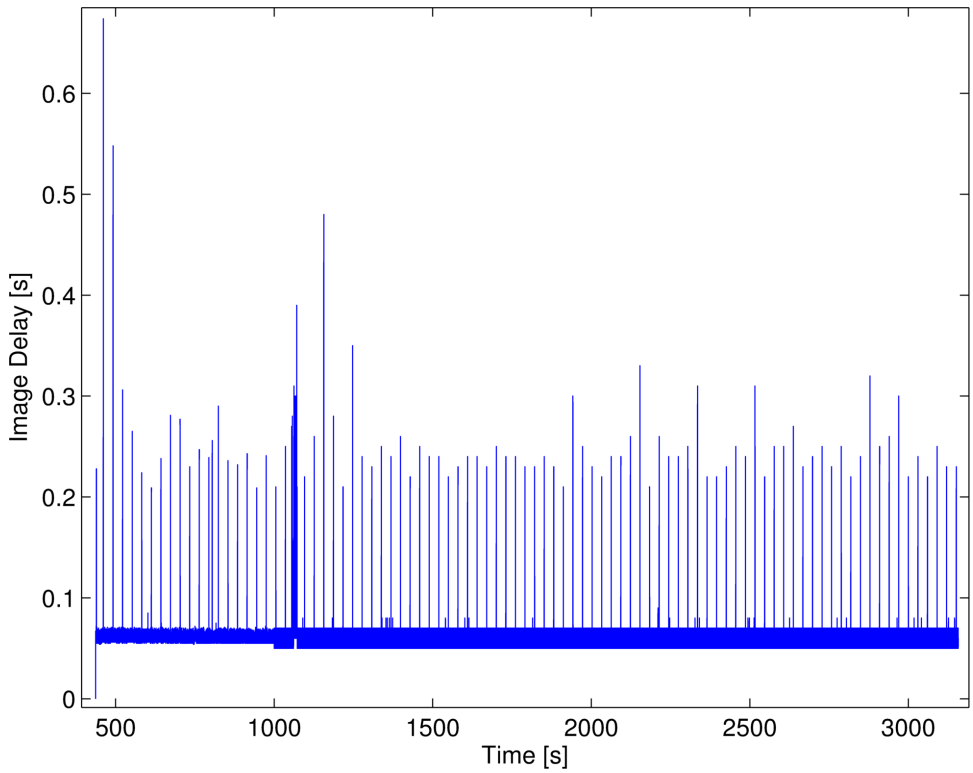

Figure 5.6. Time delays between image frames calculated using the applied timestamps.

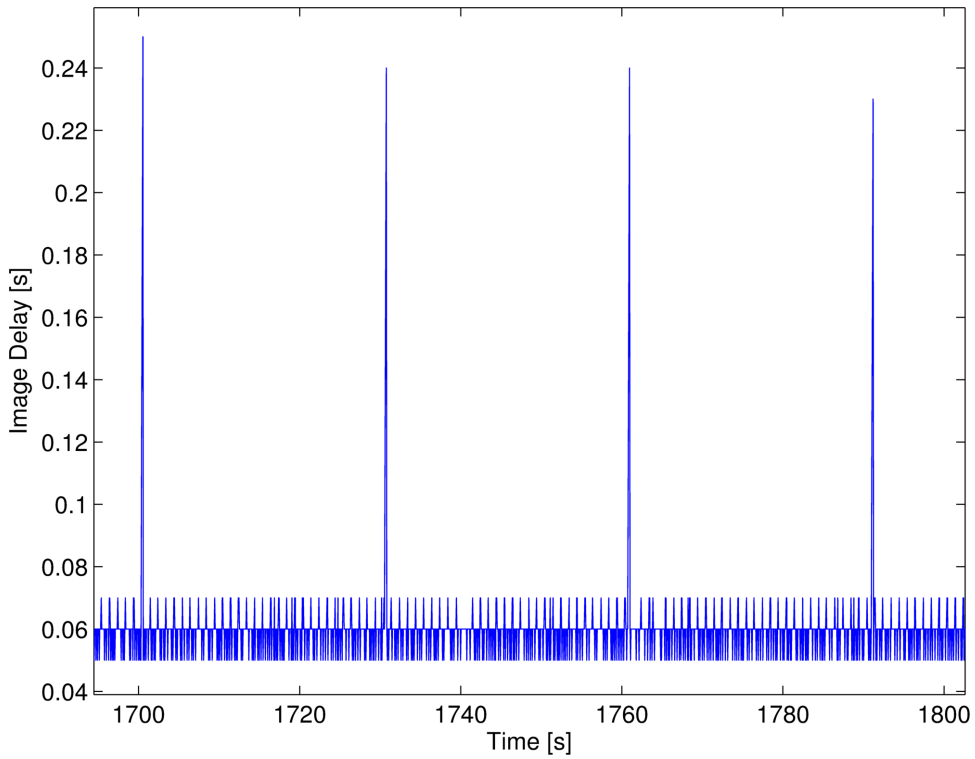

Figure 5.7. An enlarged view of the image delays showing periodic spikes of about 0.24 seconds.

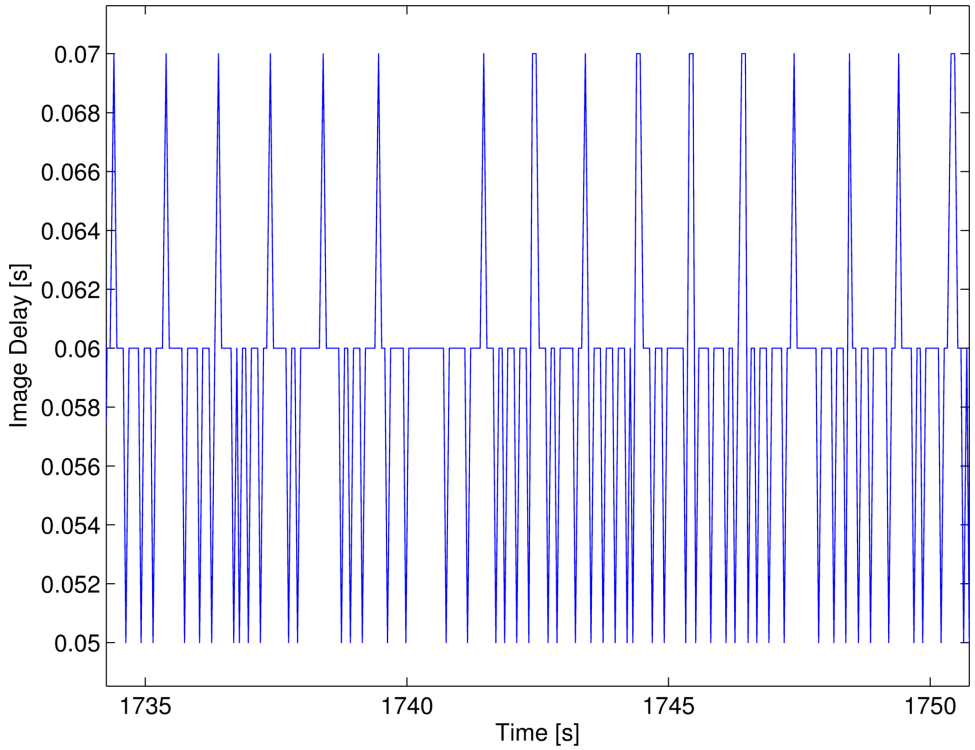

Figure 5.8. A further enlarged view of image delays showing consistent spikes of 0.01 and -0.01 seconds.

While the IMU directly outputs attitude, the camera only outputs visual data. Figure 5.9 shows an example of the image data captured during the experiment. To correlate the two data streams, both sets of data must express similar information. Attitude information can be extracted from the camera data based on optical flow. With no translation, only rotation, in the system, any observed movement in the background is directly related to the attitude of the camera. The camera's attitude can be estimated by tracking features in the background and determining their displacement. This process is carried out after the experiment to reduce processor demand and avoid further delays in capturing data. Another piece of software, written in C++ and utilizing the OpenCV library, is used to calculate the optical flow using the OpenCV function *calcOpticalFlowPyrLK* explained previously in Section 4.2.

To compute the optical flow, each image frame is compared with the previous to determine the offset of particular points. The OpenCV function *goodFeaturesToTrack*, discussed in Section 4.1, is used to pick points for tracking. 20 points are chosen for tracking with a minimum distance of 5 pixels between each point. A quality level of 0.05 is used to reject corners that don't meet the minimum eigenvalue requirement. Square windows, 3×3 pixels in size, are evaluated for corners.

Both images in a sequence are evaluated for corners. The first image provides a start value for each feature being tracked, while the second image provides an estimate of the feature's new location. With start values and estimates for the next location, *calcOpticalFlowPyrLK* can be used to track features between the two images. Pyramid levels were limited to no more than 5 levels with a square 15×15 pixel search window for each. Using 5 pyramid levels aids in searching with larger search windows and the 15×15 pixel search window ensures larger motions are captured. The optical flow calculation was limited to 20 iterations with a threshold of 0.3 pixels. With the original points and the new points calculated by *calcOpticalFlowPyrLK*, the optical flow vectors can be determined based on displacement.

The magnitudes of the optical flow vectors for the 20 points were averaged to estimate the attitude of the camera. Errant points were eliminated by setting a threshold of 0.5 pixels from the median flow magnitude value. Figure 5.10 shows an example of the vectors calculated between each image. In this case, the camera was rotating upwards so the background appears to move in the opposite direction, downwards. With a starting rotation of zero, each vector is added to the previous location to determine the total rotation of the camera. Figure 5.11 shows the accumulated position of the pixels tracked throughout the experiment, which is representative of the attitude of the camera.

Figure 5.9. Sample image frame from experiment depicting the background used for tracking.

Figure 5.10. Example of optical flow vectors calculated for an image sequence. Blue indicates previous pixel location, green indicates current pixel location, and red indicates the flow vector.

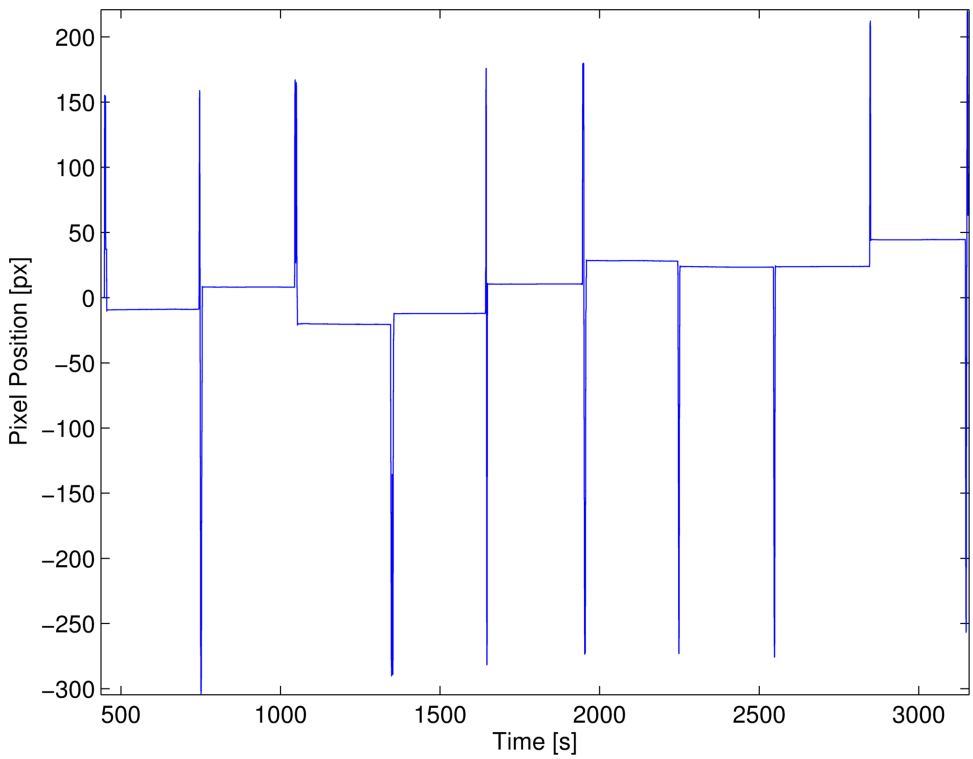

Figure 5.11. Accumulated pixel location calculated from optical flow calculations. This data is representative of the camera's attitude.

5.4 Correlating IMU and Camera Data

Overview

With the IMU attitude data and processed attitude data from the camera, we can compare the two data sets, which is key to correlation. The similar features in the two signals will be matched to synchronize the separate signals. Figure 5.12 shows the IMU and processed camera data overlaid based on the computer timestamp. The scale distorts the features in the signal, but Figure 5.13 gives a zoomed in view of each feature to show the details. Figure 5.14 and Figure 5.15 provide a larger view of the first two features emphasizing the similar motion detected by both the camera and IMU.

While these graphs are based on the computer timestamp, in reality, these timestamps would not be available. The average camera frame rate calculated in Section 5.2 can be used to estimate a timestamp, but the error quickly grows. Figure 5.16 shows the camera data with the computer timestamp and the same data with a timestamp based on average frame rate. The two plots quickly begin to diverge and will continue to diverge unbound if the estimate is not periodically corrected.

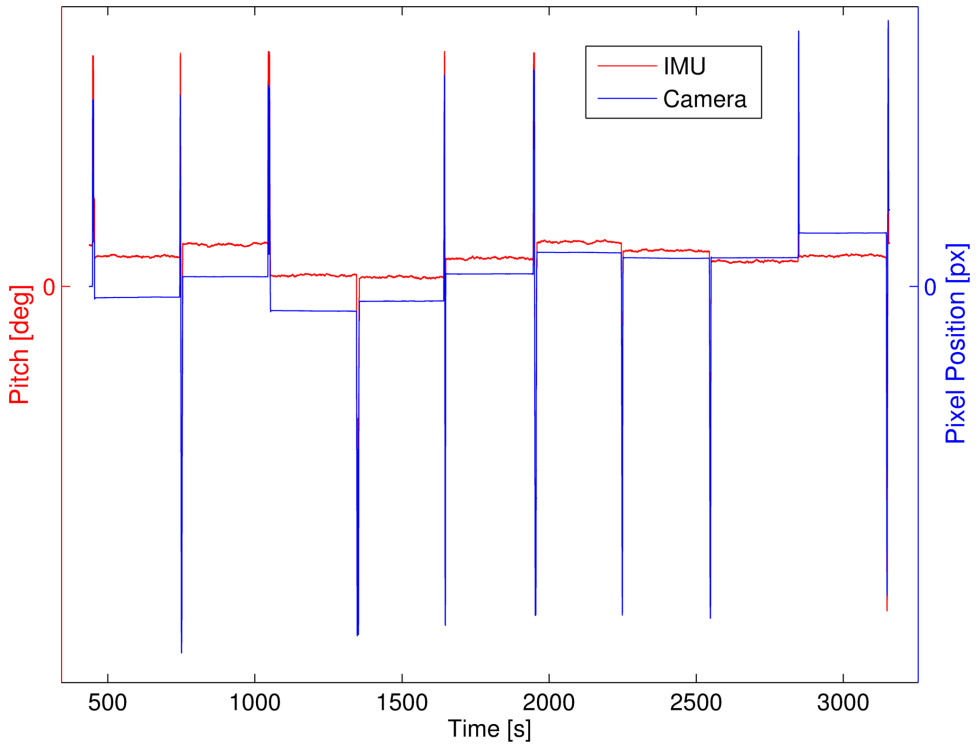

Figure 5.12. IMU and camera attitude data overlaid based on computer timestamp. While the two data sets are scaled differently, both depict similar motion.

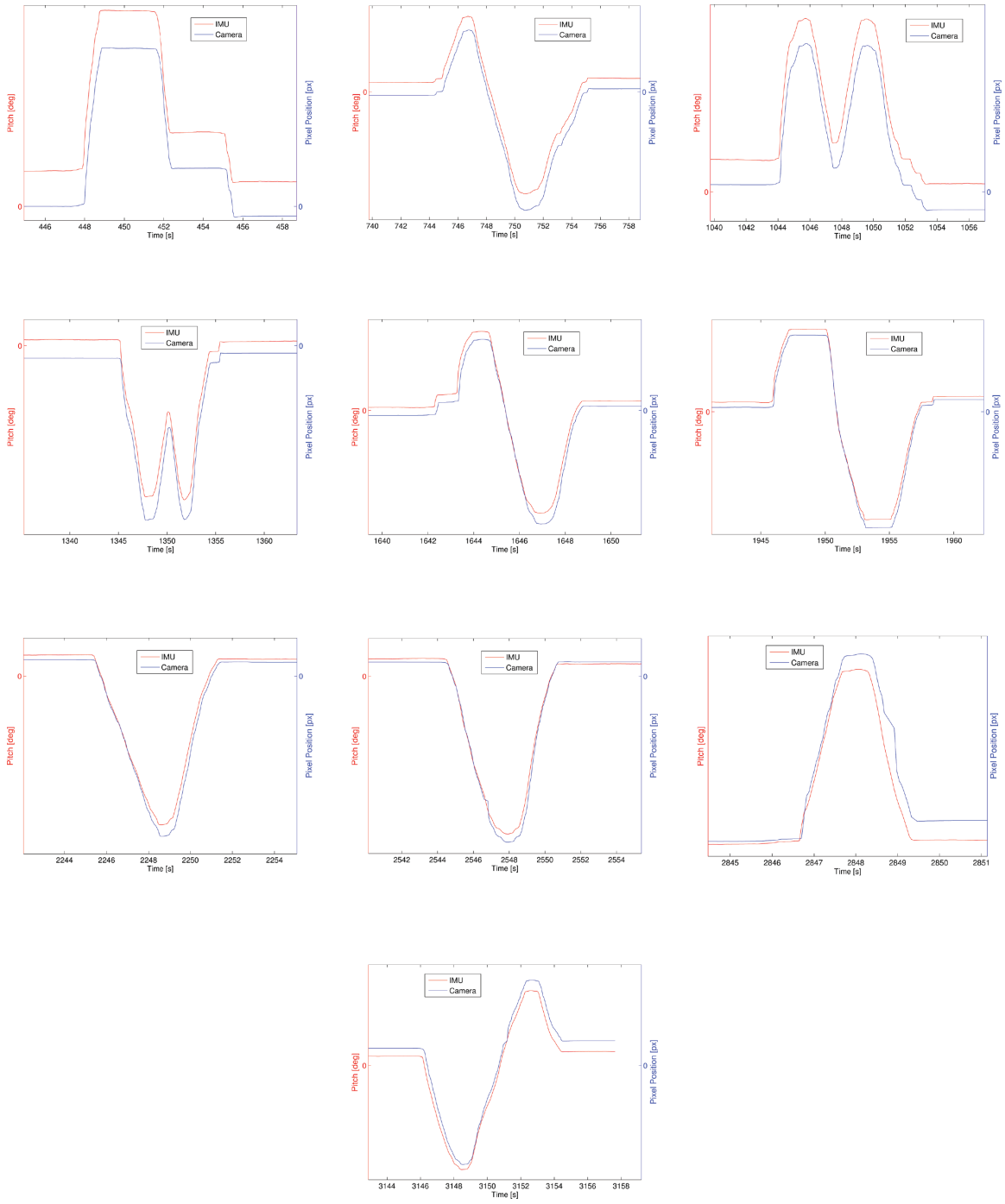

Figure 5.13. An enlarged view of the features present in both data sets.

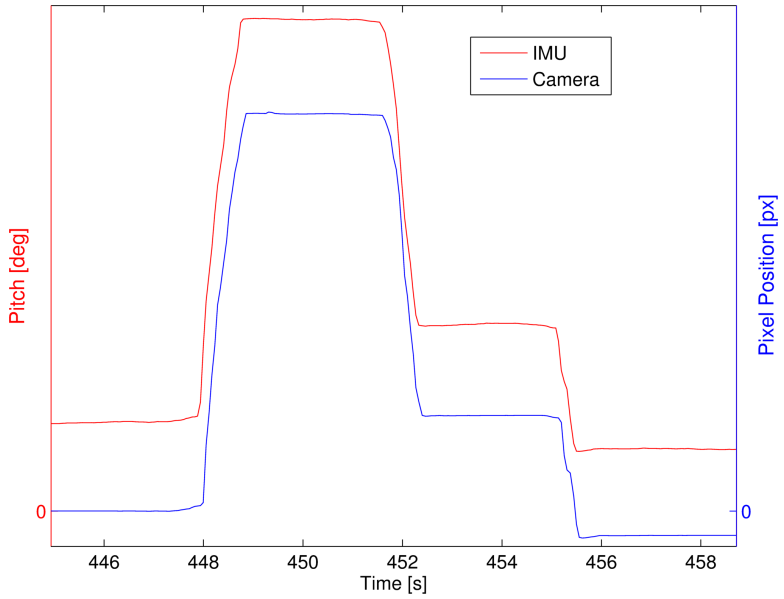

Figure 5.14. Enlarged view of the first feature depicting similar motion detected by both the IMU and camera.

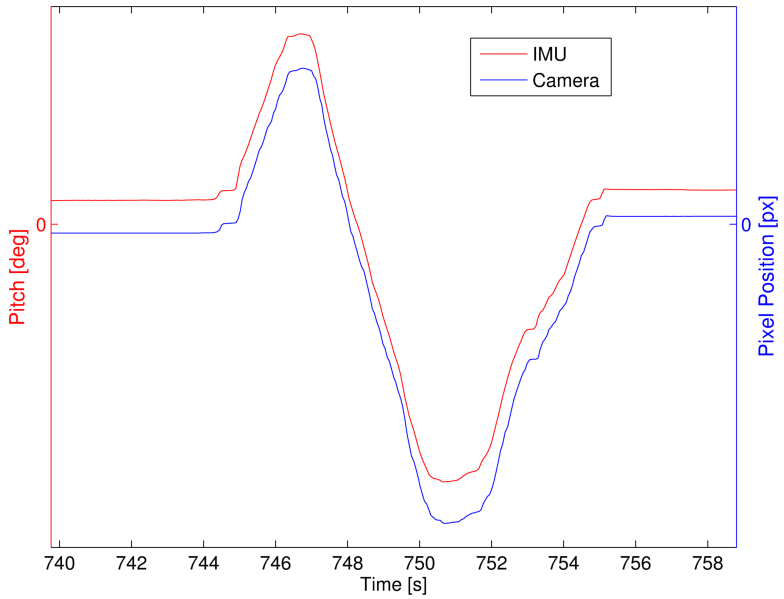

Figure 5.15. Enlarged view of the second feature depicting similar motion detected by both the IMU and camera.

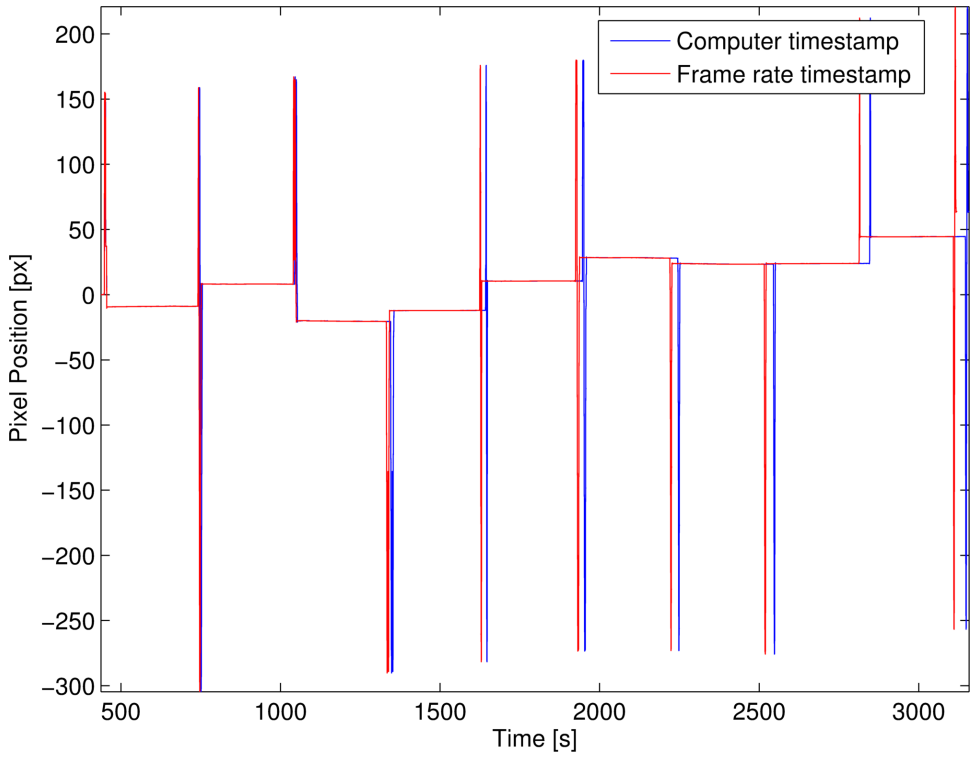

Figure 5.16. Camera data with computer timestamp and timestamp based on average frame rate. Variations in the frame rate cause the two sets to diverge.

To correct the error in the timestamp estimate, a matched filter is used to match features in the camera data with features in the IMU data. These features can be added in specifically for synchronization or can be pulled from the available data. Unique features are not needed, but it is important to have distinct features to match, otherwise the error can increase significantly. For this experiment, features were added to the data by rotating the system periodically. To find good features, we look at the variance in the data, in this case the camera data since we are pulling the templates from the camera data. The moving variance was calculated as described in Section 4.4 with a 100 sample window and threshold of 100 $pixels^2$. Figure 5.17 shows the moving variance of the camera data. From this, we set a threshold and pull the templates from areas of variance higher than the threshold. Figure 5.18 shows the selected regions and Figure 5.19 shows detailed views of these regions. Figure 5.20 and Figure 5.21 show a detailed view of the first two templates depicting the template region chosen based on the moving variance. Table 5.3 provides detailed information for each template. These templates will be matched to the IMU data to correct the estimate made using the average frame rate of the camera.

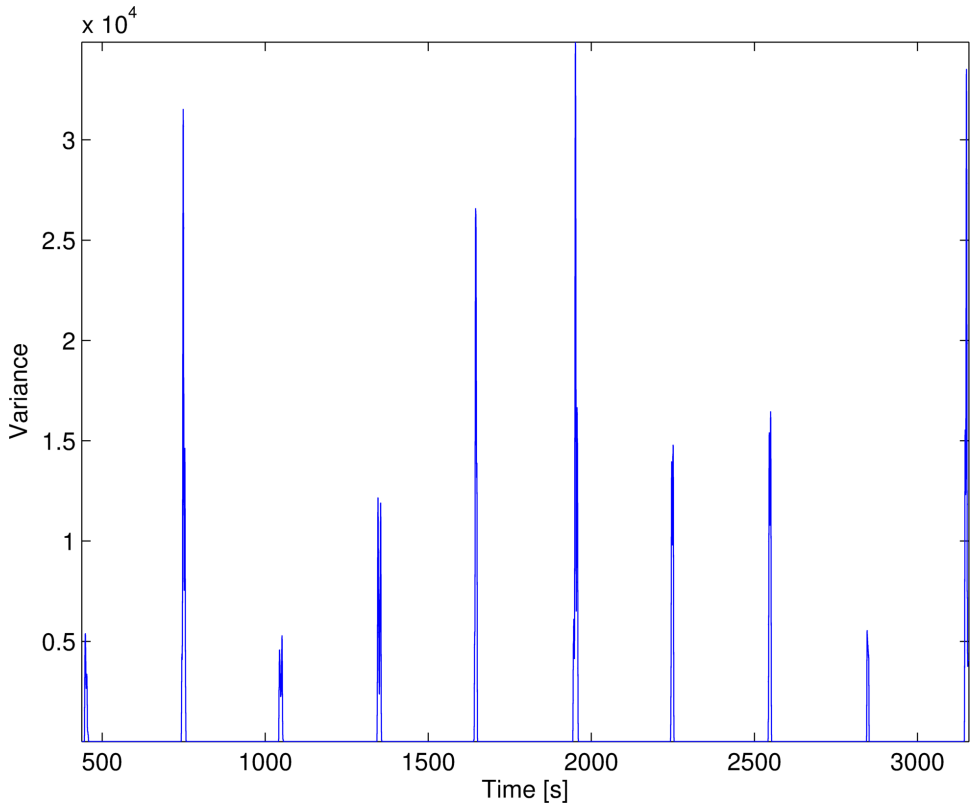

Figure 5.17. Moving variance of optical flow data. Areas of high variance are chosen as templates for matching.

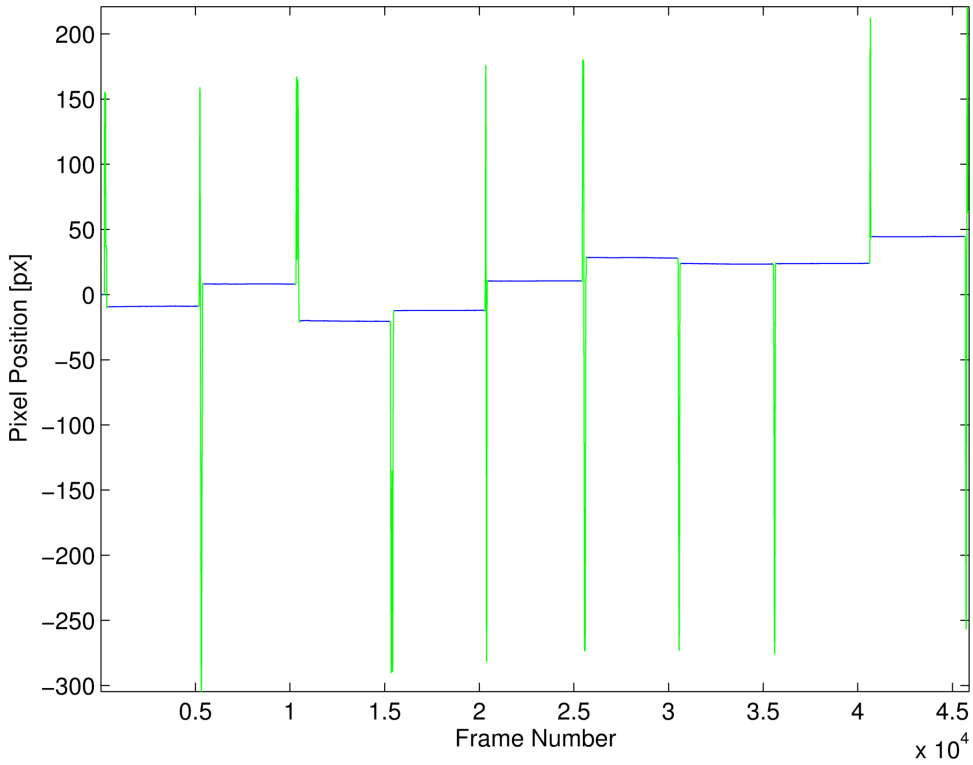

Figure 5.18. Camera data with overlaid templates in green. Templates were chosen from areas of high variance.

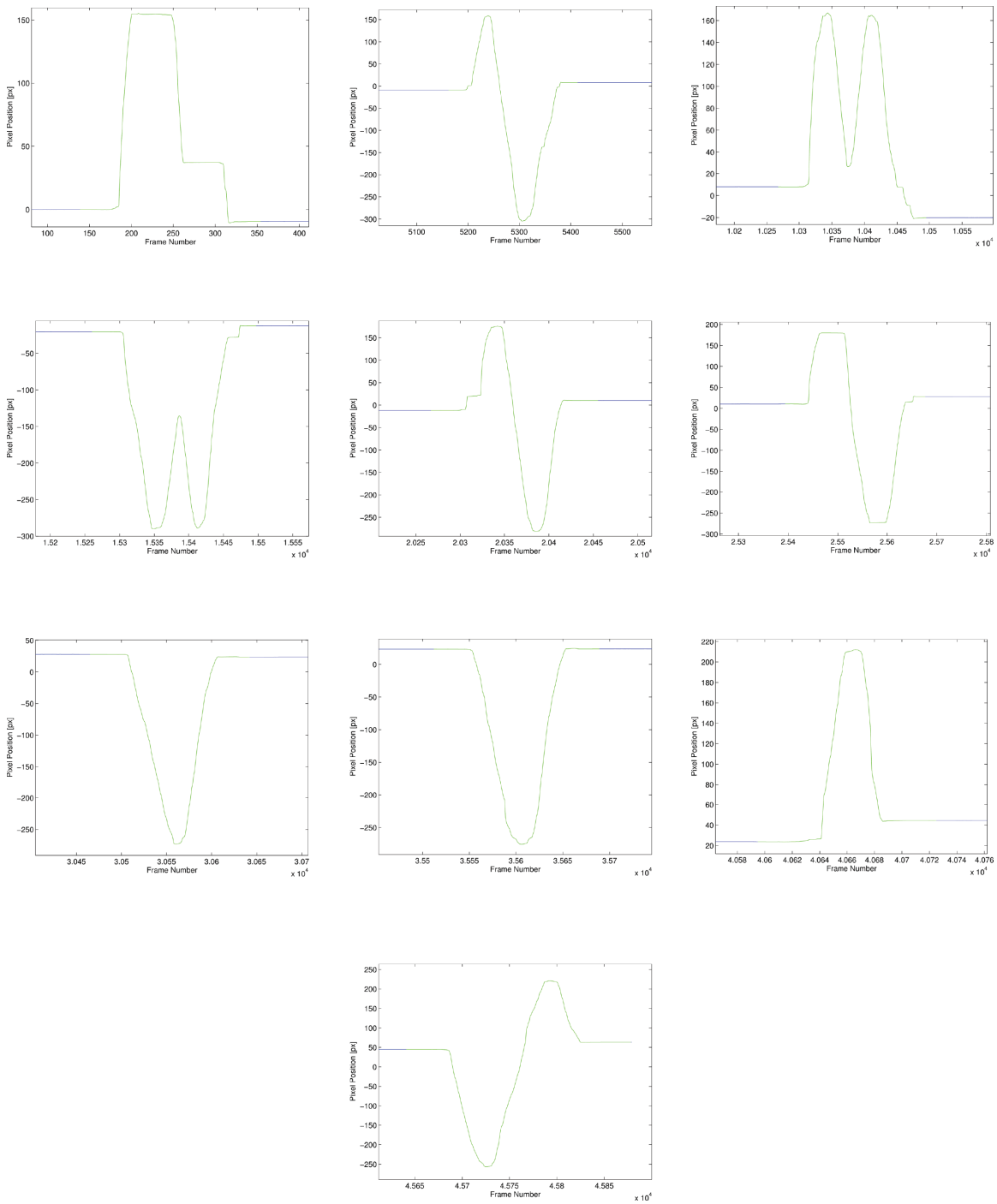

Figure 5.19. An enlarged view of the templates chosen from the camera data. Regions with distinct features were chosen based on variance.

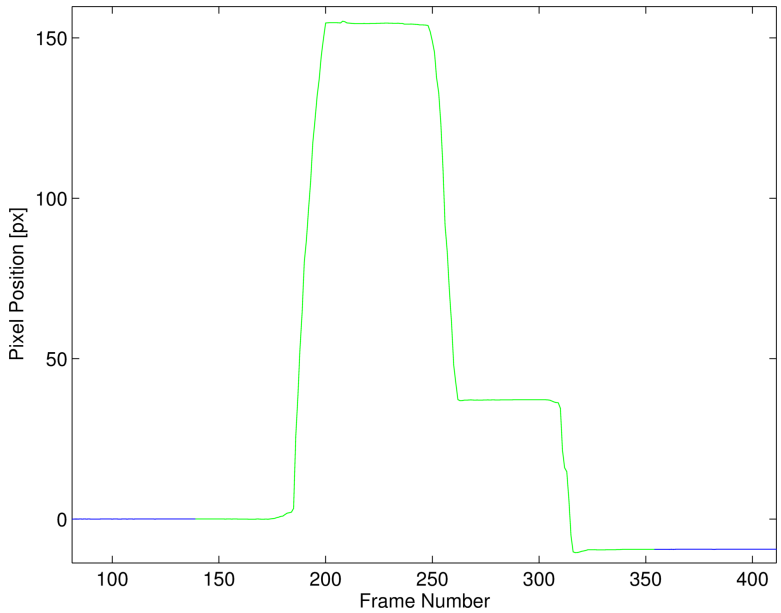

Figure 5.20. An enlarged view of the first template demonstrating the template region picked using the moving variance.

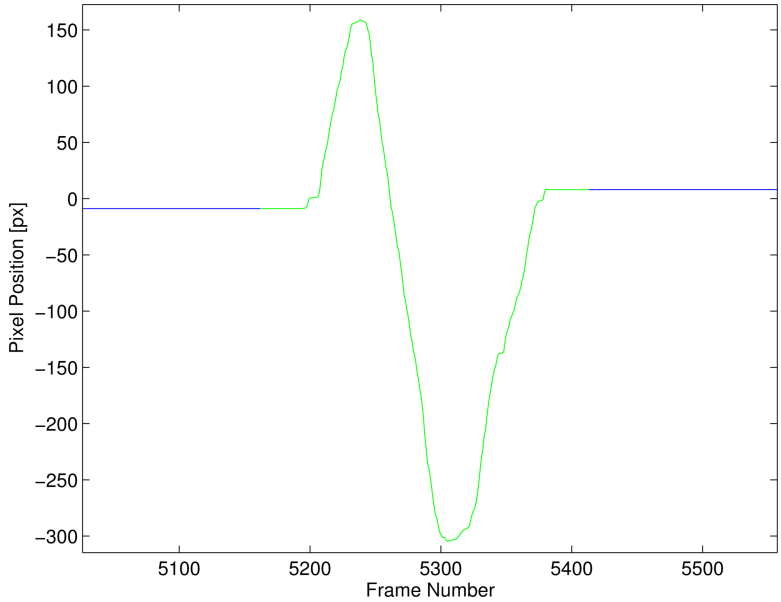

Figure 5.21. An enlarged view of the second template demonstrating the template region picked using the moving variance.

Template	Start Frame	Template Length
1	139	216
2	5162	252
3	10268	228
4	15260	238
5	20267	189
6	25394	284
7	30465	178
8	35512	178
9	40595	131
10	45641	238

Table 5.3. Details for the templates chosen from the camera data. This information is used to match the templates to the IMU data.

To match the processed camera data to the IMU data, both data sets must be scaled appropriately. While the IMU outputs attitude in degrees, the attitude estimate from the camera data is given in pixel position. The units differ but are not important as long as the shape of the two data sets is preserved for matching. To scale the two data sets vertically, they are each zeroed and normalized by subtracting the mean and dividing by the standard deviation of the data. While this satisfies vertical scaling, horizontal scaling proves slightly more challenging. In an actual scenario with no timestamps available, the only information available to guide scaling is the IMU sample rate and camera frame rate. These estimates are used to linearly interpolate the correct number of points in the processed camera data so as to match with the equivalent number in the corresponding IMU data. This vertical and horizontal scaling allows the templates taken from the processed camera data to correctly align with the corresponding features in the IMU data. Any error in the scaling results in distortion of the template and jeopardizes a match. An accurate estimate for the average frame rate and sample rate helps to ensure an accurate match between the two data sets.

Correlation is broken down into two steps, initialization and synchronization. Initialization involves aligning the start of the two data sets relative to each other. Initialization is a fundamentally more difficult task since no prior knowledge relating the data is known. The synchronization process is very similar to the initialization process, but easier in the sense that some prior knowledge relating the systems is known from initialization. For this experiment, both processes are essentially the same, with initialization searching a larger region towards the start of the data to determine a match. Synchronization then takes over and periodically searches a much smaller region in an attempt to match the templates and correct the timestamp error. Between synchronization, timestamps are estimated based on the average frame rate of the camera.

Initialization

To correlate the IMU and camera data, we need a start point, and initialization provides that start point. With no prior knowledge, both data streams would have to be compared in their entirety to ensure proper initialization. To be more realistic, knowledge from the experiment is used to set limits. We know that the camera started soon after the IMU and that there is a feature close to the beginning of the camera data. With that information, we can limit the search region to approximately the first minute of data from the IMU, shown in Figure 5.23.

The first template from Figure 5.19, shown in Figure 5.22, is matched to the beginning region of the IMU data to initialize the two data sets. Figure 5.24 shows the absolute value of the results from the initialization. The peak at IMU sample number 1453 indicates a high match between the template and signal. This point corresponds to the relative time of 445.1770 seconds. This is the timestamp estimate for the first point in the template, frame number 139 in this case, which has a computer timestamp of 445.327 seconds. The other peaks, or sidelobes, in Figure 5.24 indicate false matches caused by overlapping regions in the template and signal. Sidelobes are always present in a match but can be minimized through optimization which is not explored in this thesis.

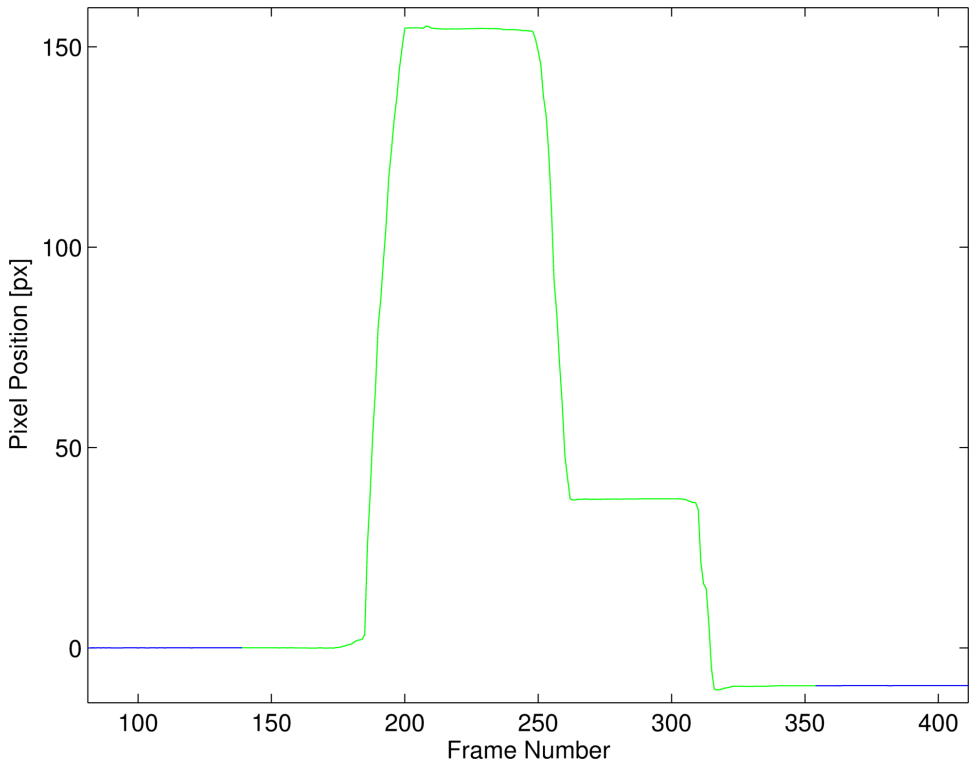

Figure 5.22. Template from camera data used in initialization.

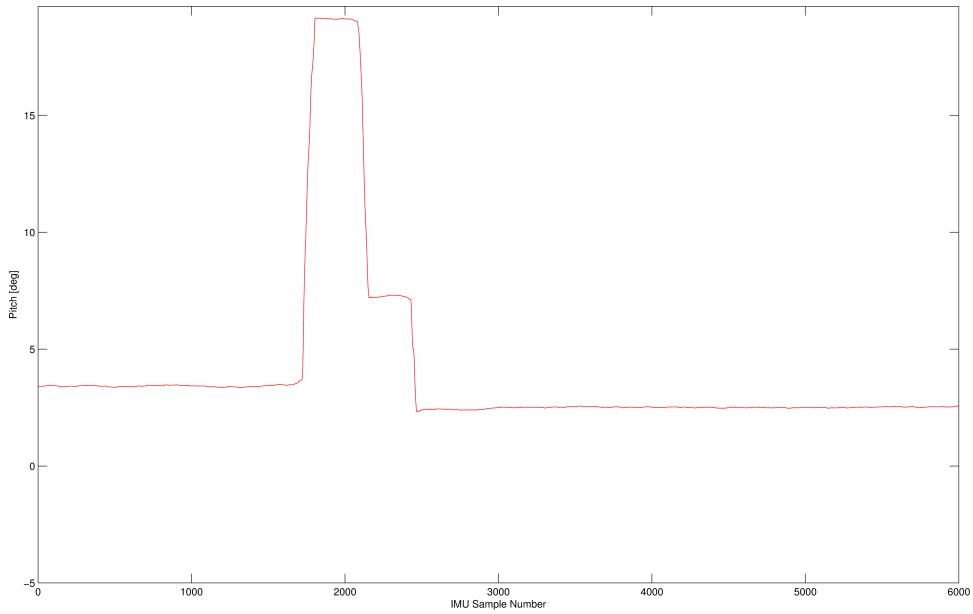

Figure 5.23. Corresponding search region in IMU data for initialization.

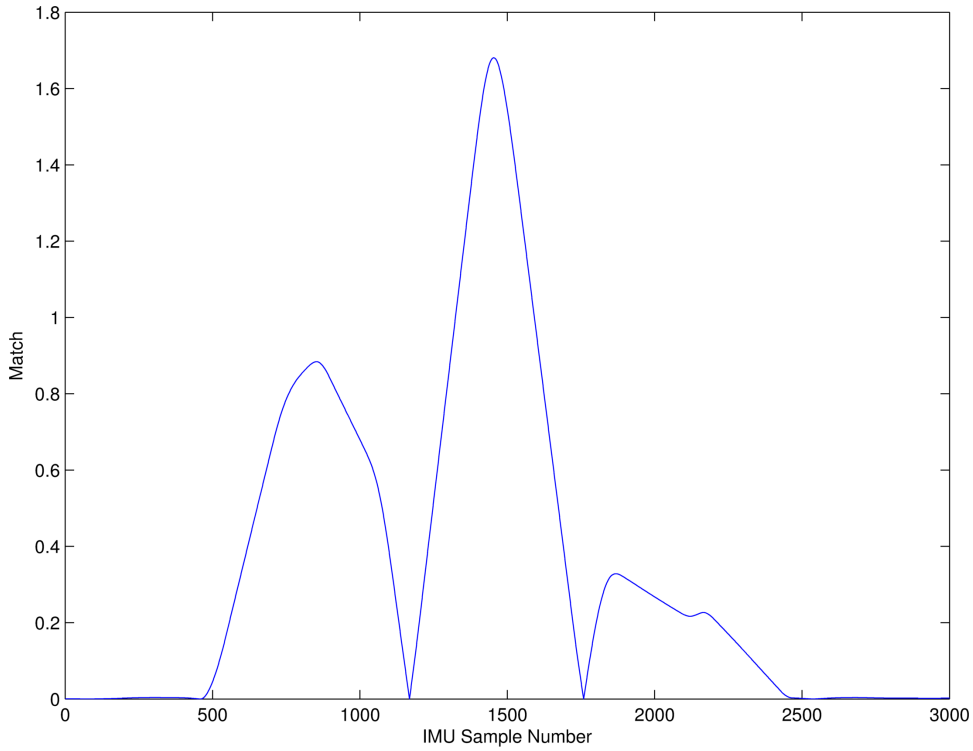

Figure 5.24. Initialization results from matching template to IMU data. The peak indicates the best match, with the sidelobes forming as the template passes over various regions of the signal.

Synchronization

Synchronization follows a similar process as initialization, but is performed periodically over a shorter range. Once initialized, we can estimate a search region based on the start point of the next template, length of the template, and the camera and IMU sample rates. Narrowing down the search region for the template greatly reduces processing time and reduces the chances and effects of false positives.

To eliminate any error from the initialization, the initialization time for the first image frame and corresponding IMU sample is taken directly from the data. Based on the templates in Figure 5.19, a search region is defined for each template which is then updated as matches are made. The templates are matched to each search region and a corresponding timestamp estimate is saved for each template. Table 5.4 gives the timestamp estimates for templates 2 through 9. The first template is ignored because the data was initialized manually. The last template is ignored because the template overlaps the end of the signal.

Template	Estimated Timestamp [s]
2	742.410
3	1045.61
4	1342.69
5	1639.95
6	1943.32
7	2243.24
8	2542.29
9	2843.95

Table 5.4. Synchronization results showing the estimated timestamp for each template based on a match with the IMU data.

Chapter 6

Results and Conclusion

6.1 Error Quantification

The timestamp error demonstrates the effectiveness of the software correlation process. It shows that the IMU and camera data can be correlated and that the maximum error in the timestamp can be limited within set bounds, dependent on the period of synchronization. The computer timestamp applied during the experiment acts as a ground truth to which the correlation estimates can be compared. Without some method of correlation, the average frame rate of the camera is the only information available for timestamping and Figure 5.16 showed how quickly this estimate can diverge. Figure 6.1 shows the timestamp error with and without synchronization. Without synchronization, the error grows unbound, however, with the software correlation method, the error is bound within controllable limits. The error bounds show that the software correlation method is an adequate approach to synchronize IMU and camera data. To bound this error, the two signals are periodically synchronized as discussed in Section 5.4. With this synchronization, the timestamp estimate based on average frame rate is corrected periodically, drastically reducing the error in the timestamp estimate. The desired precision is controllable based on the frequency of synchronizations. Table 6.1 gives the times for each template and the corresponding error.

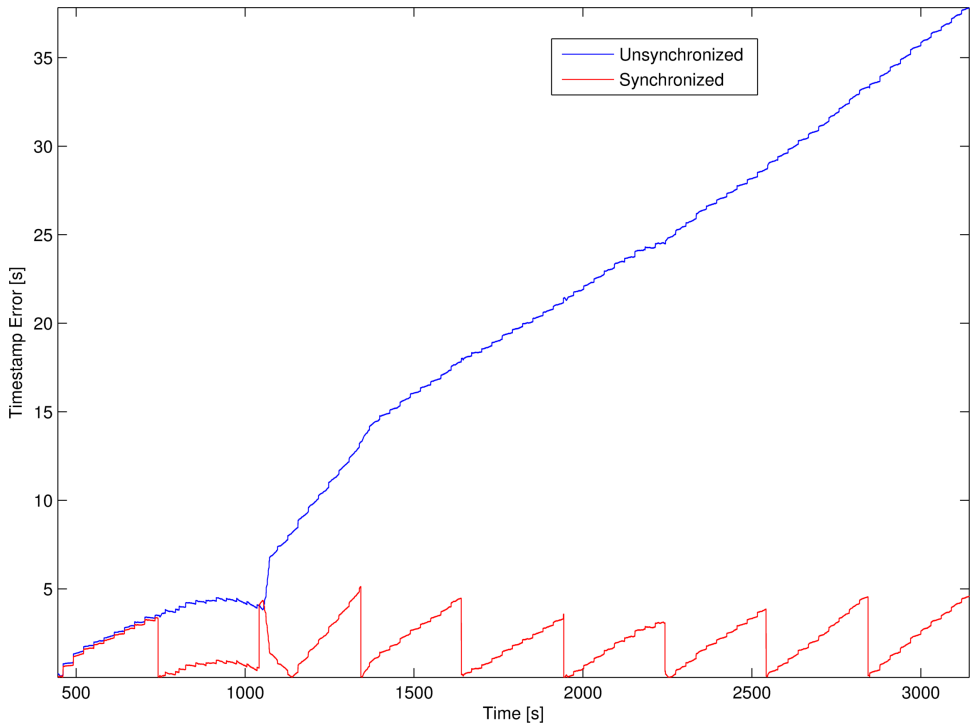

Figure 6.1. Error between the computer timestamp and the estimated timestamp. The unsynchronized timestamp is based on the average frame rate of the camera. The synchronized timestamp is based on the average frame rate of the camera with periodic synchronizations using a matched filter to realign the camera and IMU data.

Template	Computer Timestamp [s]	Estimated Timestamp [s]	Abs Error [s]
1	445.327	445.177	0.150
2	742.311	742.410	0.099
3	1041.45	1045.61	4.16
4	1342.57	1342.69	0.12
5	1639.87	1639.95	0.08
6	1943.24	1943.32	0.08
7	2242.84	2243.24	0.40
8	2542.14	2542.29	0.15
9	2843.99	2843.95	0.04

Table 6.1. Results of synchronization showing the estimated timestamp for each template, the true timestamp from the computer, and the absolute error between the two.

By periodically synchronizing the two data streams and correcting the timestamp estimate, the error in the estimate is bound. The error in the estimate is a function of the uncertainty in the average frame rate of the camera. For the Logitech C910 used in the experiment, with a frame rate standard deviation of 2.7465 fps, the error grows quickly. With approximately 5 minutes between templates, the timestamp error reaches a max of 5.127 seconds. The frequency of synchronization can be determined based on the desired limit of the error bounds.

For a standalone system like the GoPro, the two data streams would not have to be synchronized as often. With internal clocks, standalone cameras generally have a much more accurate frame rate than webcams, decreasing the error growth rate between synchronizations. As discussed in Section 5.2, the GoPro HD Hero has an average frame rate 0.018 fps faster than the published value. After 216000 frames, approximately an hour worth of footage, the timestamp error would be 1.086762 seconds, much less than the 5.127 seconds of error after only 5 minutes for the C910.

Table 6.1 shows that template 3 had 4.16 seconds of error in the synchronization. This demonstrates the importance of an accurate, consistent frame rate and sample rate. Figures 5.3 and 5.6 showed a larger than usual delay around 1050 seconds. Figure 6.1 shows a deviation in the error of the timestamp estimate based on average frame rate around the same time. This unusually large delay may be due to another, higher level, process kicking in and taking system resources. The sudden variance in frame rate and sample rate distorts the template and can cause false positives in the match. In this case, the error corrected itself since the data drifted back towards the mismatch. If the data had not drifted in that direction, the following search regions may not have encompassed the templates and the estimates may have diverged.

The error at each synchronization point is independent of each other. While the frame rate and sample rate affect scaling, which affects the error, the error does not accumulate between points. While the error does not accumulate, erroneous estimates may throw off future estimates by shifting the signal search regions as previously discussed. Tight search ranges limit the maximum effect of false positives, but also make it difficult to recover from false positives. The false positive cannot be greater or less than the search region, but a small search region may possibly exclude the template.

6.2 Conclusion

This thesis has presented a method to correlate the data between a standalone camera and IMU. A software correlation method was developed to post process and correlate the two data streams based on similar attitude measurements. An experiment was designed to test and quantify the correlation method using a wired USB webcam and IMU. With each sensor connected to a common computer, timestamps were applied to each data sample collected during the experiment. With a ground truth based on the computer timestamp, and a timestamp estimate from the software correlation, I showed that the two data streams could be correlated within set error bounds.

The software correlation method demonstrated in this work can be used to synchronize two independent data streams, one from a camera, and one from an IMU, for rotation only without translation. A higher uncertainty in camera frame rate demands a higher synchronization rate for higher precision estimates. The C910 webcam, with a 2.7465 fps frame rate standard deviation, reached 5.127 seconds of error in approximately 5 minutes. The GoPro HD Hero, with frame rate modulated by an internal clock, develops approximately 1 second of error over one hour of recording. These error values are unique to each camera and can be controlled through synchronization. A higher synchronization frequency will decrease the error bounds for each system.

This work shows that it is possible to correlate a standalone camera and IMU based on motion alone. The two sensors record measurements independently and are correlated through post processing, greatly reducing the hardware requirements during recording. Error in the timestamp estimate for the system is controlled through the frequency of synchronizations. While other methods have looked at software correlation of sensor data, this is the first method designed for standalone cameras and IMUs which uses a matched filter for periodic synchronization.

6.3 Future Work

This thesis looked at a single axis of rotation. While this is enough to correlate the two sensors, incorporating the other two axes could make the method more robust. With two additional sets of data to compare, false positives in one could be eliminated with the others. It would also guard against objects moving within the frame, possibly distorting the observed movement of the background and therefore altering the attitude estimate of the camera. Incorporating the two additional axes would simply require applying the same methods already discussed to the two extra sets of data.

While this work limits the system to only rotation, it could be expanded to allow for translation. Feedback from the IMU could correct for translation or visual cues, such as parallax, could be used to differentiate between rotation and translation. This would greatly expand the practicality of this software correlation.

6.4 Applications

The software correlation method targets systems which benefit from the characteristics of standalone sensors, specifically the reduced complexity, size, and weight of such setups. However, with standalone sensors, the work must not be time dependent to allow for the acquisition and processing of the data. While this limits certain applications, there are a wide range of tasks which are still applicable.

Subsurface and surface vessels are often used for bathymetric mapping in oceans, inlets, rivers, and other bodies of water. Autonomous, or semiautonomous, systems are typically designed to follow set paths for mapping with minimal user input. A standalone camera and IMU could be used in such a scenario to aid in optical mapping. The minimal size of standalone systems allows for better integration in subsurface systems where space is often limited. Surface vessels also benefit where weight might be a concern. Mapping is typically not time dependent and allows for the data to be retrieved and post processed.

On land, where size and weight are typically not high priority, systems may benefit from the low complexity of a standalone setup. A standalone IMU and camera can be attached to any portion of a system in which the two sensors experience similar motion. Athletes and hobbyists can attach standalone IMUs and cameras to themselves or equipment and record both optical and inertial

information. The inertial information could then be used to aid in image stabilization. Athletes would also benefit from the small size and low weight of standalone systems.

Aerial platforms benefit from all three aspects of standalone systems. Size and weight play a significant role in aerial systems, and the low complexity of standalone units allows for easy application. For example, researchers tracking animals could attach a standalone IMU and camera to a small remotely controlled aerial vehicle and survey an area. This data could then be retrieved, post processed, and analyzed to determine if the animal of interest is present and its location.

Space platforms also benefit from reduced size and weight; both are large factors in determining the cost to get systems into space. Standalone sensors could be used on short duration projects for image stabilization or tracking. Long term projects would require a method of transmitting the acquired data.

Appendix A

Computer Vision Functions

Appendix A provides the OpenCV documentation for several of the computer vision functions used in this thesis.

A.1 Feature Detection

goodFeaturesToTrack

Determines strong corners on an image.

C++: void **goodFeaturesToTrack**(InputArray **image**, OutputArray **corners**, int **maxCorners**, double **qualityLevel**, double **minDistance**, InputArray **mask**=noArray(), int **blockSize**=3, bool **useHarrisDetector**=false, double **k**=0.04)

Python: `cv2.goodFeaturesToTrack`(image, maxCorners, qualityLevel, minDistance[, corners[, mask[, blockSize[, useHarrisDetector[, k]]]]]) → corners ¶

C: void **cvGoodFeaturesToTrack**(const CvArr* **image**, CvArr* **eigImage**, CvArr* **tempImage**, CvPoint2D32f* **corners**, int* **cornerCount**, double **qualityLevel**, double **minDistance**, const CvArr* **mask**=NULL, int **blockSize**=3, int **useHarris**=0, double **k**=0.04)

Python: `cv.GoodFeaturesToTrack`(image, eigImage, tempImage, cornerCount, qualityLevel, minDistance, mask=None, blockSize=3, useHarris=0, k=0.04) → corners

Parameters:
image – Input 8-bit or floating-point 32-bit, single-channel image.
eigImage – The parameter is ignored.
tempImage – The parameter is ignored.
corners – Output vector of detected corners.
maxCorners – Maximum number of corners to return. If there are more corners than are found, the strongest of them is returned.
qualityLevel – Parameter characterizing the minimal accepted quality of image corners. The parameter value is multiplied by the best corner quality measure, which is the minimal eigenvalue (see **cornerMinEigenVal()**) or the Harris function response (see **cornerHarris()**). The corners with the quality measure less than the product are rejected. For example, if the best corner has the quality measure = 1500, and the `qualityLevel=0.01` , then all the corners with the quality measure less than 15 are rejected.
minDistance – Minimum possible Euclidean distance between the returned corners.
mask – Optional region of interest. If the image is not empty (it needs to have the type `CV_8UC1` and the same size as `image`), it specifies the region in which the corners are detected.
blockSize – Size of an average block for computing a derivative covariation matrix over each pixel neighborhood. See **cornerEigenValsAndVecs()** .
useHarrisDetector – Parameter indicating whether to use a Harris detector (see **cornerHarris()**) or **cornerMinEigenVal()**.
k – Free parameter of the Harris detector.

The function finds the most prominent corners in the image or in the specified image region, as described in [Shi94]:

1. Function calculates the corner quality measure at every source image pixel using the **cornerMinEigenVal()** or **cornerHarris()** .
2. Function performs a non-maximum suppression (the local maximums in *3 x 3* neighborhood are retained).
3. The corners with the minimal eigenvalue less than $\mathtt{qualityLevel} \cdot \max_{x,y} qualityMeasureMap(x,y)$ are rejected.
4. The remaining corners are sorted by the quality measure in the descending order.
5. Function throws away each corner for which there is a stronger corner at a distance less than `maxDistance`.

The function can be used to initialize a point-based tracker of an object.

Note: If the function is called with different values `A` and `B` of the parameter `qualityLevel` , and `A` > {B}, the vector of returned corners with `qualityLevel=A` will be the prefix of the output vector with `qualityLevel=B` .

See also: **cornerMinEigenVal()**, **cornerHarris()**, **calcOpticalFlowPyrLK()**, **estimateRigidTransform()**, **PlanarObjectDetector**, **OneWayDescriptor**

Figure A.1. OpenCV documentation for *goodFeaturesToTrack*. This function is used to detect features for the optical flow calculation.

A.2 Eigenvalue Calculation

cornerEigenValsAndVecs

Calculates eigenvalues and eigenvectors of image blocks for corner detection.

C++: void **cornerEigenValsAndVecs**(InputArray **src**, OutputArray **dst**, int **blockSize**, int **ksize**, int **borderType**=BORDER_DEFAULT)

Python: cv2.**cornerEigenValsAndVecs**(src, blockSize, ksize[, dst[, borderType]]) → dst

C: void **cvCornerEigenValsAndVecs**(const CvArr* **image**, CvArr* **eigenvv**, int **block_size**, int **aperture_size**=3)

Python: cv.**CornerEigenValsAndVecs**(image, eigenvv, blockSize, aperture_size=3) → None

Parameters:
- **src** – Input single-channel 8-bit or floating-point image.
- **dst** – Image to store the results. It has the same size as `src` and the type `CV_32FC(6)`.
- **blockSize** – Neighborhood size (see details below).
- **ksize** – Aperture parameter for the **Sobel()** operator.
- **borderType** – Pixel extrapolation method. See **borderInterpolate()**.

For every pixel p, the function `cornerEigenValsAndVecs` considers a `blockSize` $\times$ `blockSize` neighborhood $S(p)$. It calculates the covariation matrix of derivatives over the neighborhood as:

$$M = \begin{bmatrix} \sum_{S(p)}(dI/dx)^2 & \sum_{S(p)}(dI/dx dI/dy)^2 \\ \sum_{S(p)}(dI/dx dI/dy)^2 & \sum_{S(p)}(dI/dy)^2 \end{bmatrix}$$

where the derivatives are computed using the **Sobel()** operator.

After that, it finds eigenvectors and eigenvalues of M and stores them in the destination image as $(\lambda_1, \lambda_2, x_1, y_1, x_2, y_2)$ where

λ_1, λ_2 are the non-sorted eigenvalues of M
x_1, y_1 are the eigenvectors corresponding to λ_1
x_2, y_2 are the eigenvectors corresponding to λ_2

The output of the function can be used for robust edge or corner detection.

Figure A.2. OpenCV documentation for *cornerEigenValsAndVecs*. This function is used to calculate the eigenvalues for a matrix and is part of the feature detection process.

A.3 Optical Flow Calculation

calcOpticalFlowPyrLK

Calculates an optical flow for a sparse feature set using the iterative Lucas-Kanade method with pyramids.

C++: void **calcOpticalFlowPyrLK**(InputArray **prevImg**, InputArray **nextImg**, InputArray **prevPts**, InputOutputArray **nextPts**, OutputArray **status**, OutputArray **err**, Size **winSize**=Size(15,15), int **maxLevel**=3, TermCriteria **criteria**=TermCriteria(TermCriteria::COUNT+TermCriteria::EPS, 30, 0.01), double **derivLambda**=0.5, int **flags**=0)

Python: cv2.**calcOpticalFlowPyrLK**(prevImg, nextImg, prevPts[, nextPts[, status[, err[, winSize[, maxLevel[, criteria[, derivLambda[, flags]]]]]]]]) → nextPts, status, err

C: void **cvCalcOpticalFlowPyrLK**(const CvArr* **prev**, const CvArr* **curr**, CvArr* **prevPyr**, CvArr* **currPyr**, const CvPoint2D32f* **prevFeatures**, CvPoint2D32f* **currFeatures**, int **count**, CvSize **winSize**, int **level**, char* **status**, float* **trackError**, CvTermCriteria **criteria**, int **flags**)

Python: cv.**CalcOpticalFlowPyrLK**(prev, curr, prevPyr, currPyr, prevFeatures, winSize, level, criteria, flags, guesses=None) -> (currFeatures, status, trackError)

Parameters:

prevImg – First 8-bit single-channel or 3-channel input image.
nextImg – Second input image of the same size and the same type as `prevImg` .
prevPts – Vector of 2D points for which the flow needs to be found. The point coordinates must be single-precision floating-point numbers.
nextPts – Output vector of 2D points (with single-precision floating-point coordinates) containing the calculated new positions of input features in the second image. When `OPTFLOW_USE_INITIAL_FLOW` flag is passed, the vector must have the same size as in the input.
status – Output status vector. Each element of the vector is set to 1 if the flow for the corresponding features has been found. Otherwise, it is set to 0.
err – Output vector that contains the difference between patches around the original and moved points.
winSize – Size of the search window at each pyramid level.
maxLevel – 0-based maximal pyramid level number. If set to 0, pyramids are not used (single level). If set to 1, two levels are used, and so on.
criteria – Parameter specifying the termination criteria of the iterative search algorithm (after the specified maximum number of iterations `criteria.maxCount` or when the search window moves by less than `criteria.epsilon` .
derivLambda – Not used.
flags –
Operation flags:
- **OPTFLOW_USE_INITIAL_FLOW** Use initial estimations stored in `nextPts` . If the flag is not set, then `prevPts` is copied to `nextPts` and is considered as the initial estimate.

The function implements a sparse iterative version of the Lucas-Kanade optical flow in pyramids. See [Bouguet00].

Figure A.3. OpenCV documentation for *calcOpticalFlowPyrLK*. This function finds the optical flow for specified points between two input images.

Bibliography

[1] Opencv 2.4.1 documentation. http://docs.opencv.org/.

[2] J. Barron, D. Fleet, S. Beauchemin, and T. Burkitt. Performance of optical flow techniques. In *Computer Vision and Pattern Recognition, 1992. Proceedings CVPR '92., 1992 IEEE Computer Society Conference on*, pages 236–242, 1992.

[3] Jean-Yves Bouguet. Pyramidal implementation of the lucas kanade feature tracker description of the algorithm. Technical report, Intel Corporation Microprocessor Research Labs, 2000.

[4] Gary Bradski and Adrian Kaehler. *Learning OpenCV*. O'Reilly Media, Inc, 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2008.

[5] Alison Brown and Dan Sullivan. Precision kinematic alignment using a low- cost GPS/INS system. In *Proceedings of ION GPS*, September 2002.

[6] M. Caccia, M. Bibuli, R. Bono, G. Bruzzone, and E. Spirandelli. Aluminum hull USV for coastal water and seafloor monitoring. In *OCEANS 2009 - EUROPE*, pages 1 – 5, may 2009.

[7] John Canny. A computational approach to edge detection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, PAMI-8(6):679–698, nov. 1986.

[8] Shih-Jie Chou, Rui-Cian Weng, and Tai-Shan Liao. Synchronize sensors and cameras. Technical report, National Applied Research Laboratories, Instrument Technology Research Center, HsinChu, Taiwan, September 2010.

[9] James L. Crowley and Alice C. Parker. A representation for shape based on peaks and ridges in the difference of low-pass transform. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, PAMI-6(2):156–170, march 1984.

[10] M.R. Fetterman, T. Hughes, N. Armstrong-Crews, C. Barbu, K. Cole, R. Freking, K. Hood, J. Lacirignola, M. McLarney, A. Myne, S. Relyea, T. Vian, S. Vogl, and Z. Weber. Distributed

multi-modal sensor system for searching a foliage-covered region. In *Technologies for Practical Robot Applications (TePRA), 2011 IEEE Conference on*, pages 7 – 14, april 2011.

[11] Rama Chellappa Gang Qian and Qinfen Zheng. Robust structure from motion estimation using inertial data. *J. Opt. Soc. Am. A*, 18(12):2982 – 2997, Dec 2001.

[12] Chris Harris and Mike Stephens. A combined corner and edge detector. *In Proceedings of The Fourth Alvey Vision Conference*, pages 147–151, 1988.

[13] Berthold K. P. Horn and Brian G. Schunck. Determining optical flow. *Artificial Intelligence*, 17:185–203, 1981.

[14] Myung Hwangbo, Jun-Sik Kim, and T. Kanade. Inertial-aided KLT feature tracking for a moving camera. In *Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on*, pages 1909–1916, oct. 2009.

[15] A. Jepson and M.J. Black. Mixture models for optical flow computation. *Computer Vision and Pattern Recognition, 1993. Proceedings CVPR '93., 1993 IEEE Computer Society Conference on*, pages 760–761, jun 1993.

[16] Jong-Hyuk Kim and S. Sukkarieh. Airborne simultaneous localisation and map building. In *Robotics and Automation, 2003. Proceedings. ICRA '03. IEEE International Conference on*, volume 1, pages 406 – 411, sept 2003.

[17] K.M. Kitani, T. Okabe, Y. Sato, and A. Sugimoto. Fast unsupervised ego-action learning for first-person sports videos. In *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*, pages 3241 – 3248, june 2011.

[18] 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, December 2010.

[19] T. Lindeberg. Edge detection and ridge detection with automatic scale selection. *Computer Vision and Pattern Recognition, 1996. Proceedings CVPR '96, 1996 IEEE Computer Society Conference on*, pages 465–470, jun 1996.

[20] Tony Lindeberg. Detecting salient blob-like image structures and their scales with a scale-space primal sketch: A method for focus-of-attention. *International Journal of Computer Vision*, 11:283–318, 1993.

[21] J. Lobo and J. Dias. Vision and inertial sensor cooperation using gravity as a vertical reference. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 25(12):1597 – 1608, dec 2003.

[22] David G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, nov 2004.

[23] Bruce D. Lucas and Takeo Kanade. An iterative image registration technique with an application to stereo vision. In *Proceedings of the 7th international joint conference on Artificial intelligence*, volume 2 of *IJCAI'81*, pages 674–679. Morgan Kaufmann Publishers Inc., 1981.

[24] J Matas, O Chum, M Urban, and T Pajdla. Robust wide-baseline stereo from maximally stable extremal regions. *Image and Vision Computing*, 22(10):761–767, 2004.

[25] Hans Moravec. Obstacle avoidance and navigation in the real world by a seeing robot rover. In *tech. report CMU-RI-TR-80-03, Robotics Institute, Carnegie Mellon University doctoral dissertation, Stanford University*, number CMU-RI-TR-80-03. September 1980.

[26] T. Oskiper, Zhiwei Zhu, S. Samarasekera, and R. Kumar. Visual odometry system using multiple stereo cameras and inertial measurement unit. In *Computer Vision and Pattern Recognition, 2007. CVPR '07. IEEE Conference on*, pages 1 – 8, june 2007.

[27] S.I. Roumeliotis, A.E. Johnson, and J.F. Montgomery. Augmenting inertial navigation with image-based motion estimation. In *Robotics and Automation, 2002. Proceedings. ICRA '02. IEEE International Conference on*, volume 4, pages 4326 – 4333, 2002.

[28] Juan Serrat, A. López, and D. Lloret. On ridges and valleys. *Pattern Recognition Internatiional Conference (15th)*, 2000.

[29] Linda G. Shapiro and George C. Stockman. *Computer Vision*. Prentice Hall, 2001.

[30] Jianbo Shi and C. Tomasi. Good features to track. *Computer Vision and Pattern Recognition, 1994. Proceedings CVPR '94., 1994 IEEE Computer Society Conference on*, pages 593–600, jun 1994.

[31] Priyabrata Sinha. *Speech Processing in Embedded Systems*. Springer, 1st edition, 2009.

[32] Stephen M. Smith and J. Michael Brady. SUSAN—a new approach to low level image processing. *International Journal of Computer Vision*, 23:45–78, 1997.

[33] Jonathan Y. Stein. *Digital Signal Processing: A Computer Science Perspective*. John Wiley Sons, Inc., 2000.

[34] C.N. Taylor, M.J. Veth, J.F. Raquet, and M.M. Miller. Comparison of two image and inertial sensor fusion techniques for navigation in unmapped environments. *Aerospace and Electronic Systems, IEEE Transactions on*, 47(2):946 – 958, april 2011.

[35] Miroslav Trajković and Mark Hedley. Fast corner detection. *Image and Vision Computing*, 16(2):75–87, 1998.

[36] A. Unterholzner, M. Rohland, M. Schweitzer, and H.-J. Wuensche. Vision-based online-calibration of inertial gaze stabilization. In *Intelligent Vehicles Symposium (IV), 2010 IEEE*, pages 646 – 651, june 2012.

[37] A. Vaccarella, M.D. Comparetti, A. Enquobahrie, G. Ferrigno, and E. De Momi. Sensors management in robotic neurosurgery: The ROBOCAST project. In *Engineering in Medicine and Biology Society,EMBC, 2011 Annual International Conference of the IEEE*, pages 2119 –2122, 30 2011-sept. 3 2011.

[38] Oliver J. Woodman. An introduction to inertial navigation. Technical Report UCAM-CL-TR-696, University of Cambridge, Computer Laboratory, August 2007.

[39] Chunjoo Yoon, Donghoon Jeong, and Junggon Sung. Developmnet strategy of the road safety survey and analysis vehicle : ROSSAV. In *Proceedings of the Eastern Asia Society for Transportation Studies*, volume 5, pages 1891 – 1899, 2005.