1-1-2011

Video Stabilization Using SIFT Features, Fuzzy Clustering, and Kalman Filtering

Kevin Veon
University of Denver

Video Stabilization Using SIFT Features, Fuzzy Clustering, and Kalman Filtering

_____

A Thesis

Presented to

The Faculty of Engineering and Computer Science

University of Denver

_____

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

_____

by

Kevin L. Veon

August 2011

Advisor: Dr. Mohammad H. Mahoor, Ph.D.

Author: Kevin L. Veon
Title: Video Stabilization Using SIFT Features, Fuzzy Clustering, and Kalman Filtering
Advisor: Dr. Mohammad H. Mahoor, Ph.D.
Degree Date: August 2011

Abstract

Video stabilization removes unwanted motion from video sequences, often caused by vibrations or other instabilities. This improves video viewability and can aid in detection and tracking in computer vision algorithms. We have developed a digital video stabilization process using scale-invariant feature transform (SIFT) features for tracking motion between frames. These features provide information about location and orientation in each frame. The orientation information is generally not available with other features, so we employ this knowledge directly in motion estimation. We use a fuzzy clustering scheme to separate the SIFT features representing camera motion from those representing the motion of moving objects in the scene. Each frame's translation and rotation is accumulated over time, and a Kalman filter is applied to estimate the desired motion. We provide experimental results from several video sequences using peak signal-to-noise ratio (PSNR) and qualitative analysis to demonstrate the results of each design decision we made in the development of this video stabilization method.

Acknowledgements

I would like to thank Dr. Kimon Valavanis for bringing me to the University of Denver and giving me the opportunity to attend graduate school here. My thanks to the unmanned systems lab members who also came from Florida: Jonathan Girwar-Nath, Jason Monast, Allistair Moses, for helping me when needed and creating a fun environment in which to work.

I would also like to thank Dr. Mohammad Mahoor and Dr. Richard Voyles for providing me with an interesting project and for their advisement over the past year. They gave me the direction and focus necessary to complete my Masters with thesis.

Most importantly, I would like to thank my family, particularly my wife, for keeping me in line and making sure I finished my graduate school experience. Their support has been paramount to my successes in life.

This research was funded by the National Science Foundation grant IIP-1032047.

Table of Contents

List of Figures and Tables

CHAPTER 1. INTRODUCTION

Video stabilization is the removal of unwanted motion in video sequences. This unwanted motion, often called jitter, is induced by vibrations or an unsteady platform. This is true of any system using a camera, such as robotic systems or handheld cameras. Removal of this undesired motion results in a much steadier, smoother video. This smoother video is a more viewable, user-friendly depiction of the motion of the camera.

Video stabilization generally consists of three primary steps: correspondence calculation, motion estimation, and motion compensation. The correspondence calculation step finds all available matches between two frames which can potentially be used in the motion estimation step. Some techniques, such as phase correlation do not calculate correspondences. The motion estimation step then chooses which correspondences will be used and employs one of several motion models to estimate how the chosen corresponding points have moved between frames. This step has the most significant impact on the results of video stabilization, and is the primary step which is modified in the development of new stabilization techniques. Finally, the motion compensation step uses the reverse of the motion model to move the current frame in line with its desired location.

1.1. Motivation

The motivation behind the development of this video stabilization method is to create a general method that can be used for any camera system, with consideration for

both robotic systems and handheld cameras. The goal is to provide a smoother video with jitter removed, while maintaining the desired motion. The smooth video should be more viewable by a human operator without significantly affecting the understanding of the actual camera motion.

For robotic systems, stabilization has two major benefits. It provides a video sequence that is much more viewable to human operators because undesired high frequency motions can lead to discomfort and make it more difficult to notice important events. Stabilized video can also improve performance of other computer vision techniques which may not be as robust to motion as some feature-based techniques.

The benefit of video stabilization to handheld cameras is primarily improved viewability of the video. Videos from handheld cameras are prone to instability due to shaky hands or walking motions. Handheld cameras often have special hardware to help compensate for these instabilities, but undesired motion can fall beyond the range of the hardware's capabilities and further compensation is necessary.

1.2. Problem Statement

Stabilization of video is a difficult task in real-world environments. If nothing is known about the structure of the scene, or if the scene is dynamic, the problem becomes even more difficult. Real-time stabilization is also a major concern when considering situations involving robotic systems, though situations with handheld cameras might not be as stringent on timing requirements.

There are two primary types of motion within most real videos: global motion and local motion. Both types of motion contain a combination of desired and undesired

motions. There are certain situations where only one of these types of motion is present, but they rarely apply to robotic systems and handheld cameras.

Global motion describes the motion of the scene induced by the motion of the camera. This motion affects every object in the entire image, though its impact is dependent on the relative depth of the object from the camera. This is the ideal type of motion to be used in motion compensation, as stabilizing global motion effectively stabilizes the camera to a fixed point in space. The best candidates for estimating global motion in an image are background objects: objects which represent static structures such as buildings and streets.

Local motion describes the motion of non-static objects within the scene, such as cars or pedestrians. Non-static objects can be represented by a combination of global motion and their inherent motion. This motion is not useful in motion compensation unless an accurate model for the inherent motion of the object is available. It is unlikely that a good model will be available and easily assigned to specific objects automatically, leaving local motion as a detriment to motion estimation. Consider a static camera observing several objects moving in a single direction. Stabilizing the scene with respect to these non-static objects effectively forces the camera to be moved in the opposite direction of the objects instead of remaining stationary.

Separation of local and global motion is paramount to accurately estimating the actual motion of the camera. Without this separation, there is no way to accurately estimate the motion of the camera because local motion is parasitic and will introduce incorrect information into the motion estimation algorithm. If the local motion is

removed, then only global motion will be used, which is much more descriptive of the motion of the camera.

1.3. Proposed Solution

To achieve the goals stated in the Motivation section and address the concerns introduced in the Problem Description section, we developed a video stabilization algorithm involving scale-invariant feature transform (SIFT) features [22], fuzzy set theory, and Kalman filtering.

We use SIFT features to match interest points between frames. Interest points are locations in the image which are distinguishable and distinct enough to be accurately matched between frames with little or no incorrect matches. SIFT is widely regarded in the computer vision community as one of the most accurate and robust types of features. SIFT features provide information about scale, location, and orientation. The most important information SIFT provides is the orientation of the features, which is unavailable with most other feature types. By looking at how these features have moved, they can be used to measure the motion of the camera, though both local and global motion is provided with SIFT features.

We use Fuzzy set theory to separate local and global motion. The SIFT Features are clustered, and then assigned fuzzy membership values for each cluster. This information combined with a trust value we introduce for each feature is used to determine which features are most likely to represent the global motion of the camera. Fuzzy membership simplifies some design decisions with regards to clustering and provides a convenient way to separate information when uncertainty exists [18].

We use Kalman filtering to estimate the desired motion of the camera. This allows the camera to be mounted on a moving body, such as a mobile robot, while still successfully stabilizing the video. The estimated motion from the fuzzy clustering step is used as a sensor input for the Kalman filter and is combined with a state model which describes the horizontal, vertical, and rotational motion. We use a simple Kalman filter because a purely linear state model can sufficiently describe the motion of the camera in video stabilization scenarios.

1.4. Contribution

This thesis presents a new video stabilization algorithm using SIFT features, fuzzy clustering, and Kalman filtering. We provide the original code for our MATLAB implementation of our video stabilization algorithm.

Unlike other video stabilization methods that use SIFT features, our algorithm utilizes the orientation information provided by the SIFT features. The difference in orientation of matched features directly corresponds to the rotation between images. Unlike most other video stabilization algorithms, we estimate the rotation separately from the translation because the information is readily available.

Few video stabilization methods use fuzzy logic or fuzzy set theory. We use fuzzy set theory to assign membership values from each SIFT feature to clusters of features for the purpose of separation of local and global motion. The process of clustering features followed by membership assignment we will further refer to as fuzzy clustering. To our knowledge, no other video stabilization algorithm uses fuzzy clustering to perform this task.

We also discuss important problems that must be considered in all video stabilization methods which have been brought up during the development of our algorithm, but are not discussed in any of the papers in the Literature Review chapter. The most important of these problems is separating the estimation of rotation and translation in separate steps, and how to handle situations where the center of rotation of the image is not the center of the image plane.

1.5. Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 contains a review of video stabilization literature. Chapter 3 provides a background of SIFT, fuzzy set theory, and Kalman filtering. Chapter 4 introduces the video stabilization method we developed and discusses design decisions and issues that have been brought up during the development process. Chapter 5 presents experimental results on several video sequences, both qualitative and quantitative where applicable. Chapter 6 presents conclusions and future work.

CHAPTER 2. LITERATURE REVIEW

This chapter contains a review literature in the field of video stabilization. It covers a wide array of approaches including optical flow-based techniques and feature-based techniques and several methods of separating global and local motion. Some of the less relevant papers which had little impact beyond understanding the current state of the field of video stabilization will be mentioned in little detail. Some of the more relevant papers that had a larger impact on our research will be discussed in detail.

Several papers provided primarily literature review with only some additional details. Corsini et al. [12] provided a literature review of video stabilization up to 2006. Their discussion is primarily concerned with point non-feature-based correspondence techniques such as optical flow, block matching, and phase correlation. Luo et al. [24] provided a literature review on the use of Lucas-Kanade-Tomasi (LKT) features, which are based on features described by Shi and Tomasi [33], for video stabilization. They introduce the average pixel difference performance measure for use when the ground truth stabilized video is known. They also introduce a periodic method for dealing with accumulated error, though the period must be determined experimentally, and is not usable online with great accuracy. Niskanen et al. [27] discuss performance measurement for video stabilization. They discuss the drawbacks of mean squared error and peak signal-to-noise ratio, and provide examples showing that the visual quality did not directly correlate with each type of measurement in some cases.

2.1. Block Matching Methods

Several methods use block matching as a means to determine the translation between frames. They differ somewhat in where the blocks lie within the image and how each block is weighted. Liu et al. [21] use a relatively dense collection of blocks for matching. They separate global and local motion using random sampling consensus (RANSAC) to remove outliers, which they consider to be fast-moving objects. Hsiao et al. [16] choose to use one block in each corner of the frame and one in the center. The center block is given twice the weight because they believe the center can be trusted more than edges. They use Kalman filtering which does not necessarily remove jitter, but allows for desired motion. Ondrej et al. [28] use a very limited amount of blocks due to matching's computational complexity. They determine which blocks to use by choosing a few of the possible blocks in the frame with the highest contrast value defined by the difference between a pixel and its neighbors to the right and below. Shen et al.

[31] use circular blocks to create invariance to rotation. They use four blocks around the corners of the frame because they believe that too much local motion occurs in the center.

Shi et al. [34] use a block-matching method of video stabilization for the purpose of prosthetic eyes. They measure the offset between a block in the center of the previous frame to its location in the current frame. They use the sum of absolute differences to determine the error between the reference block and a window in the current frame. They make no consideration for local and global motion separation or accounting for desired motion.

2.2. SIFT-Based Methods

Shen et al. [32] use SIFT features reduced by principal component analysis (PCA) to estimate motion. The matched PCA-SIFT features are used in a RANSAC algorithm to remove outliers and provide an initial estimate for motion, which is further refined using particle filtering. The combination of RANSAC and particle filtering effectively remove local motion and undesired motion from the scene.

Yang et al. [40] perform video stabilization using SIFT features and particle filtering. They use SIFT features because they consider the accuracy of motion estimation to be the most important aspect of video stabilization, more important than speed. This is likely due to the fact that most video stabilization is not done in real-time on an embedded system. Instead, it is used offline to improve the quality of video for presentation purposes.

They utilize a motion model consisting of a scale, rotation, and translation. To estimate the current state of the system, they use particle filtering. Instead of using the posterior probability as is customary in particle filtering, they decided that it would not be accurate enough because it does not take into account the current state of the system. Instead, they calculate another measure for estimating the next state based on the mean of all matched features in the current state. They separate local and global motion under the assumption that global motion has overwhelmingly more representation than local motion and that local motion will have larger motion vectors than global motion.

They separate desired and undesired motion by using Kalman filtering where their state vector consists of the four elements of the two-dimensional rotation matrix, the x

and y translations, and the scaling factor. The estimated desired motion is removed from the accumulated motion so that only undesired motion is left for compensation.

Battiato et al. [3] perform video stabilization using SIFT features. They begin their paper by discussing SIFT features and explaining that some of the matches produced by SIFT are not correct. In order to improve their accuracy, they apply stricter parameters to the SIFT matching process. By imposing these restrictions, they reduce the number of incorrect matches that SIFT produces. They create a vector between the locations of matched features t represent the motion between frames.

A rigid motion model is then fitted using these vectors. Some of these vectors are not a good representation of the camera's motion. Some represent mismatched features, and others represent objects with local motion. To separate out these vectors, they implement an iterative least-squares refinement method. First, vectors with large Euclidean norms are discarded because they are less likely to represent camera motion. In each iteration, the least-squares solution is calculated, followed by a calculation of the error between the estimated motion vector and each individual motion vector. Any vector with an error value higher than an adaptive threshold is removed. Both rotation and translation errors are evaluated independently, and both have the ability to discard a bad motion vector. This effectively separates the mismatched features and local motion while preserving most of the good motion vectors.

To estimate intentional motion, they use a method called motion vector integration from [29]. The motion between each frame is accumulated and integrated using a damping factor δ:

$$IMV(n) = \delta IMV(n-1) + GMV(n) \quad (1)$$

where IMV is the integrated motion vector, and $GMV(n)$ represents the global motion vector calculated for frame n. This integrated motion vector is used to calculate the motion to be compensated for frame n, $C(n)$ using

$$C(n) = IMV(n) - IMV(n-1). \quad (2)$$

The value of δ is normally a predetermined factor which lies in the range of [0 1]. Battiato et al. instead chose to use an adaptive damping coefficient by utilizing the sum of the global motion vectors between the two previous frames. A small sum yields a higher damping coefficient under the assumption that the camera is intended to be stationary.

Battiato et al. [2] expanded upon this work by changing the way that bad motion vectors are discarded. Again, motion vectors with large Euclidean norms are discarded before a least-squares estimate for the motion is performed. Instead of using a fixed threshold to discard motion vectors based on error, they instead utilize a fuzzy logic system that maps the errors to a quality factor.

First, the error values of the motion vectors are fuzzified. The median error values are calculated. Subsequently, each error value is divided by this median error. Any vector with an error less than the median has a value less than one, and any vector with an error more than the median has a value more than one. A set of fuzzy rules is applied to the fuzzified values based on their quality. Lower values are rewarded with higher quality. The output of the fuzzy system is a quality factor between [0 1] for each motion vector. These vectors are then sorted and the highest 60% are utilized in another least-squares estimation to produce the final estimated motion.

The orientation of SIFT features is often ignored in motion estimation. This information is unavailable with features such as KLT or Harris corners, but is readily available with SIFT. Wu et al. [39] use the rotation of the tracked features to determine the motion of body parts for human action recognition. They introduce a feature called SIFT-ME. Derived from SIFT, SIFT-ME is a feature that describes both the translation and in-plane rotation of tracked SIFT features. The use of both rotational and translational components of SIFT-ME outperformed the use of only the translational component for activity recognition.

The SIFT-ME feature is a vector containing three values: the rotation β about the center of the feature, the magnitude ρ of the translation, and the direction α of the translation. The vector $<\beta, \rho, \alpha>$ is sufficient to describe both the translation of a feature and its in-plane rotation:

$$\begin{bmatrix} x_t \\ y_t \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\beta) & -\sin(\beta) & \rho \cdot \cos(\alpha) \\ \sin(\beta) & \cos(\beta) & \rho \cdot \sin(\alpha) \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{t-1} \\ y_{t-1} \\ 1 \end{bmatrix}. \quad (3)$$

The SIFT-ME equation is equivalent to the rigid motion model where β is equivalent to the rotation of the frame, $\rho \cdot \cos(\alpha)$ is equivalent to horizontal translation, and $\rho \cdot \sin(\alpha)$ is equivalent to vertical translation.

2.3. Optical Flow Methods

Matsushita et al. [26] focus on creating a full frame after motion compensation instead of trimming out information between frames. They use a hierarchical pyramid of optical flow for motion estimation and focus on deblurring and inpainting of pixels. They keep a record of the motion chain as well as local motion to approximate where a pixel

undergoing local motion would occur in the next frame based on the assumption of consistency. This outperforms other mosaicking techniques because it takes into account local motion. Their image blur algorithm looks at correlated pixels in nearby frames and chooses the sharpest pixel to replace the pixel in the current frame. Their methods are more appropriate for commercial video enhancement, though some aspects such as their deblurring method would be very useful in video stabilization.

Cai et al. [9] introduced a method called delta flow. They discuss SIFT features but reach the conclusion that it is not suitable for real-time operation. Instead, they adopt the pyramidal Lucas-Kanade optical flow method. This is a very common gradient-based optical flow method used in many computer vision applications, and is very fast.

Delta flow is a method for separation of local and global motion estimation. It looks at a combination of the consistency of the velocity as well as the consistency of the acceleration at any given pixel location where flow is calculated. The assumption is that local motion will induce inconsistent motion and/or acceleration over time. This inconsistency can be used to separate it from the consistent global motion vectors. This idea works because it is based on pixel location and not features. Features likely have consistent motion and acceleration regardless of whether or not they represent local or global motion, but gradient-based optical flow will be inconsistent when an object moves through a location. They build histograms of motion which are used to determine which optical flow vectors should be used in motion estimation and jitter estimation. Histograms are built for both optical flow and delta flow, which is useful when large objects move through the scene.

The delta flow process is used to estimate jitter in the frame. To obtain the desired motion, the jitter can simply be accumulated and removed from the estimated global motion. The estimated jitter can also be used as a guide for where in the histogram to look for the global motion. They use the latter method in their experiments as it outperforms the former.

2.4. Miscellaneous Methods

Two methods stabilize the video not necessarily with respect to the background as they do not have a means of separating the background and objects. Batur et al. [4] utilize a pyramid-based block matching algorithm to cut down on computational cost of finding the correct block within an entire frame. They choose which region has the lowest accumulated motion and treat it as the global motion estimation, which could represent the background or the foreground. Kurz et al. [20] describe the scene as a cloud of points in three dimensions and fixate on a point. The point is not necessarily representative of the background. Neither of these approaches considers intentional camera motion.

A few algorithms target video stabilization for automobiles specifically. Zhang et al. [41] use a central sub-window and optical flow for video stabilization. The central location is where intentional zooming least affects optical flow. Broggi et al. [7] use a Sobel edge filter and a horizontal histogram to determine vertical motion. They only stabilize vertical motion because it is the most prominent type of motion while driving a car. Neither of these methods has any consideration local motion or desired motion.

Two methods are meant to be run as offline algorithms, unable to run in real-time on embedded systems. Buehler et al. [8] created an algorithm to rebuild a stable trajectory

of a virtual camera moving through the scene. They use unstructured lumigraph rendering. After estimating motion, they project the unstable frames in an appropriate way to render them properly in the desired trajectory. Chang et al. [11] modify the Lucas-Kanade optical flow algorithm to work without the assumption of illumination consistency. They regularize the frames temporally, which causes their algorithm to require the entire video sequence before stabilization.

Ramachandran et al. [30] perform video stabilization for the purpose of unmanned aerial vehicles (UAVs) and micro aerial vehicles (MAVs) while creating a mosaic of the scene over time. They first calculate a rough stabilized video using phase correlation. Phase correlation was first introduced in 1975 by Kuglin and Hines [19]. The process for calculating phase correlation between images g_a and g_b is as follows:

$$G_a = \mathcal{F}\{g_a\}, G_b = \mathcal{F}\{g_b\} \tag{4}$$

where $\mathcal{F}\{g\}$ represents the two-dimensional Fourier transform of image g.

$$R = \frac{G_a G_b^*}{|G_a G_b|} \tag{5}$$

where multiplication is element-wise and $*$ represents the complex conjugate.

$$r = \mathcal{F}^{-1}\{R\} \tag{6}$$

where $\mathcal{F}^{-1}$ is the two-dimensional inverse Fourier transform.

$$(\Delta x, \Delta y) = \arg\max_{\text{x,y}}\{\text{r}\} \tag{7}$$

This step is followed by a calculation of optical flow between the previous frame and the frame shifted to compensate for the translation calculated using phase correlation. The optical flow method they use is introduced by Srinivasan et al. [35]. They use the

least-median of squares technique for discarding regions with incorrect flow estimates. This gives an initial estimate of there the frame should lie in the mosaic. The optical flow step is repeated between the current frame placed on the mosaic and the mosaic itself to refine the location. The motion parameters are further refined using an iterative least-squares approach.

They utilize information provided by the inertial measurement unit (IMU) on-board the UAVs and MAVs and prior knowledge of camera parameters to improve the accuracy of their method. Most video stabilization algorithms choose not to use this hardware due to costs, but most robots already have it on-board. This information is used as a starting point for the iterative least-squares step, which helps reduce the chance that convergence will not occur before the maximum number of iterations is exceeded.

2.5. Summary of Literature Review

The most popular methods for video stabilization use either SIFT feature tracking or Lucas-Kanade optical flow, including Lucas-Kanade-Tomasi (LKT) feature tracking. The most common reason to use Lucas-Kanade optical flow instead of SIFT feature tracking is speed. Lucas-Kanade optical flow can easily be performed in real-time on a CPU using their pyramidal algorithm while SIFT cannot be performed in real-time without using a GPU. The cost of using Lucas-Kanade optical flow is that it is not considered as accurate as SIFT features. It is important to note that SIFT features provide information about orientation, which can be seen as a benefit for using SIFT, though it is rarely used for video stabilization. The orientation provided by SIFT features was

successfully used in SIFT-ME for human action recognition, which led us to believe that it would be useful in video stabilization as well.

The primary focus of any of the video stabilization papers in this literature review is the process of determining which correspondences to use, and how to compensate for motion while maintaining intentional, desired motion. Several methods were introduced for the separation of local and global motion, such as particle filtering, majority vote, outlier removal, and even fuzzy logic. Each of these methods was successful in their task according to the experimental results provided, though fuzzy logic is the least explored and most interesting method to us. Methods such as low-pass filters, integrated motion vectors, and Kalman filtering have been used to estimate the desired motion. The Kalman filter is the most advanced of these methods, though it requires more *a priori* knowledge about the motion of the camera. We consider the price of *a priori* knowledge negligible and use Kalman filtering for this purpose in our method.

The most commonly used motion model in literature is the rigid motion model with or without scale, though a two-dimensional affine model is used frequently as well. Camera motion is relatively simple, and the rigid motion model performs well for the purpose of video stabilization. The scale factor is not important when the framerate of the camera is sufficiently high or the camera motion is sufficiently slow. For this reason, we use the rigid motion model for our video stabilization method.

CHAPTER 3. PREREQUISITE KNOWLEDGE

This chapter contains a short discussion of prerequisite knowledge necessary to fully understand the stabilization process presented in Chapter Four.

3.1. Scale-Invariant Feature Transform

The scale-invariant feature transform (SIFT) was introduced by David Lowe in 1999. The reader is referred to [22] for details including the mathematics behind SIFT feature calculation and matching. This level of detail is unimportant for the casual reader who is interested in understanding our video stabilization method.

The general idea behind SIFT is to find robust, stable features which are distinct and distinguishable. To make the features robust and invariant to scale, the image is rescaled several times while searching for them. To make the feature stable, areas of high contrast are discarded as they are very susceptible to changes in illumination. To make the features more distinct and distinguishable, the orientations of local gradients within the region surrounding it are stored in a histogram, called a feature descriptor. These descriptors are used in a nearest-neighbor search for feature matching.

The computer vision community regards SIFT as one of the most accurate and robust features for use in object recognition and image matching. In addition to being invariant to scale, SIFT features are also invariant to orientation, translation, and relatively robust to illumination variance.

SIFT features provide information about their location in the frame, their orientation, and their scale. Although SIFT can provide very accurate matching between similar images, a small number of these matches is incorrect. Applying stricter parameters to SIFT can aid in reducing the number of mismatches, but the total number of matches will be reduced as well, possibly removing a significant number of correct matches. Instead of doing this, it is common practice to perform an outlier removal procedure such as random sampling consensus (RANSAC), which allows for significantly more correct matches than strict SIFT matching parameters.

3.2. Fuzzy Set Theory

Classical set theory has binary membership qualities: either the element is a member of the set, or it is not [18]. This distinction is absolute. It follows classical logic, which similarly defines propositions to be definitely true or definitely false. Classical set theory is valid when the boundaries for membership can be clearly and precisely defined.

Fuzzy set theory is an extension of classical set theory that is not as restrictive about membership. It allows for partial membership, which can be any value in the range [0,1]. Unlike classical logic, fuzzy logic allows for propositions to have a degree of truth. This is particularly useful when data is imprecise, inaccurate, or not clearly defined and known. The classic example of fuzzy set theory's usefulness is in the case of height. The set "tall" is not clearly defined. Different people will have different opinions about what constitutes being tall. If a precise boundary is set at a height of 2.0m, then someone whose height is 1.99m will not be considered tall. This does not make sense, as we do not

distinguish such a small difference as significant enough to change our opinion of whether or not someone is tall.

Fuzzy set theory allows us to define a degree of membership to a set at any given value. A membership function is defined that describes the level of membership over the domain. In the example of the set "tall", we might define a ramp function that ramps from 0 to 1 starting at 1.6m and ending at 2.0m, maintaining a value of 1 for all values greater than 2.0m. This membership function follows the intuition that as a person's height increases, it is more likely that people will consider him to be tall.

3.3. Kalman Filtering

Kalman filtering is a well-known technique used in signal processing to iteratively and adaptively track a signal while removing noise and other inconsistencies. It is a means of estimating exact signal values given an *a priori* model, sensor input, and covariance matrices representing the relative uncertainty of both. It is assumed that both the state model and sensor input are subject to Gaussian noise. The Kalman filter is linear in nature, and does not handle nonlinearity well, though it can be extended to approximate nonlinearity by using the Jacobian matrix of the state model instead of the state model itself.

The system can be represented by:

$$x_k = A_k x_{k-1} + B_k u_k + w_k \tag{8}$$

where x_k is the current state of the system, x_{k-1} is the previous state of the system, A_k is the state model at time k, B_k is the control input model, u_k is the control input vector, and w_k is Gaussian noise.

Kalman filtering is performed in two major steps, prediction and update. The prediction step estimates the new state of the system x_k and the new uncertainty matrix for the system P_k:

$$x_k = A_k x_{k-1} + B_k u_k \tag{9}$$

where each variable is the same as in Equation (8), and

$$P_k = A_k P_{k-1} A_k^{\mathrm{T}} + Q_k \tag{10}$$

where Q_k is the covariance matrix representing the uncertainty of the state model.

The update step updates the new state and uncertainty matrix for the system by considering the sensor input.

$$y_k = z_k - H_k x_k \tag{11}$$

where y_k represents the residual between the sensor input z_k and the relevant state values, which are selected using the mask H_k.

$$S_k = H_k P_k H_k^{\mathrm{T}} + R_k \tag{12}$$

where S_k represents the residual covariance and R_k is the covariance matrix representing the uncertainty of the sensor input.

$$K_k = P_k H_k^{\mathrm{T}} S_k^{-1} \tag{13}$$

where K_k is the optimal Kalman gain.

Using these calculated values, the new state and new uncertainty of the system can be calculated using the following two equations:

$$x_k = x_k + K_k y_k \tag{14}$$

$$P_k = (I - K_k H_k) P_k. \tag{15}$$

CHAPTER 4. VIDEO STABILIZATION USING SIFT FEATURES, FUZZY CLUSTERING, AND KALMAN FILTERING

This chapter is split into two subsections: Our video stabilization algorithm using SIFT features, fuzzy clustering, and Kalman filtering and a discussion of design decisions and their impact on the quality of video stabilization. The algorithm subsection will go into detail about the steps performed for stabilization. The discussion subsection will contain information about design decisions and issues that have been brought up about our approach during our research.

The stabilization process uses SIFT features for motion estimation, fuzzy clustering for separation of local and global motion, and Kalman filtering for separation of desired and undesired motion. SIFT features are further augmented with a trust value that helps determine if they are good features to use in motion estimation. Fuzzy clustering is performed on the rotation and translation of features to fully separate global and local motion. Kalman filtering is used to separate desired and undesired motion, which is important when considering a moving camera, as is the case in almost all mobile robot applications.

Several design decisions have been made which helped construct our stabilization algorithm. The most important decisions were the use of SIFT features and the use of fuzzy clustering. Several issues were brought up during the creation of our video stabilization algorithm and are addressed in this subsection as well. Major issues include

the computational complexity of the algorithm, the separation of rotation and translation estimation, and the impact of a center of rotation that does not lie in the center of the image frame.

4.1. Video Stabilization Algorithm

This subsection contains a detailed description of each step in our video stabilization process. It will contain minimal discussion about the impact of each step, and primarily contain information about what each step actually does. Discussion of design decisions and potential issues are not included in this section for the purpose of clarity of the algorithm. They will instead be discussed in Section 4.2.

Figure 1 is a block diagram of the stabilization process with macroblocks describing the three main steps of video stabilization: correspondence calculation, motion estimation, and motion compensation. The motion estimation macroblock contains the bulk of the algorithm, which coincides with the importance of motion estimation in successful video stabilization.

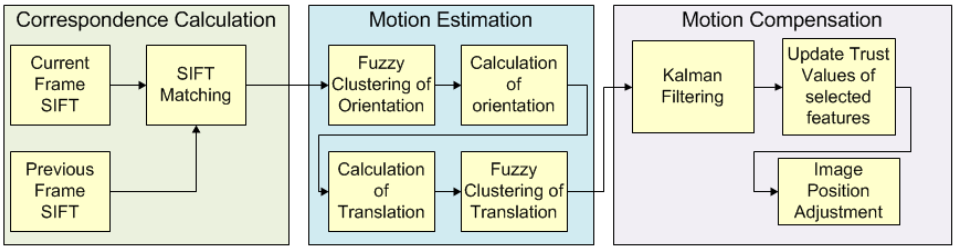


Figure 1: Block diagram of our video stabilization process using SIFT features, fuzzy clustering, and Kalman filtering.

Our stabilization algorithm consists of six steps. First, we calculate SIFT features and match them between the previous and current frames, augmenting them with appropriate trust values. Then we perform fuzzy clustering on the rotations between

matched SIFT features. This is followed by calculating the local translations for each matched SIFT feature. We perform a second fuzzy clustering on these translations. We then use this information as a sensor input into a Kalman filter along with a predetermined state model. Finally, we compensate for undesired motion and update the trust values of those matched features used.

4.1.1. Assumptions

First, we will discuss motion modeling in video stabilization and what assumptions are made when each of them is used, followed by our final decision of which motion model to use and the reasons why we use it. After this discussion, we will list our other assumptions.

We define the coordinate system as follows: the Z-axis is normal to the image plane, pointing forward, the X-axis points horizontally to the right, and the Y-axis points vertically upward. Roll (θ) is the counter-clockwise rotation around the Z-axis, pitch (σ) is the counter-clockwise rotation around the X-axis, and yaw (ψ) is the rotation around the Y-axis.

We assume that the scene is planar in nature. All objects in the scene are assumed to be at the same depth, which is a common assumption made in computer vision when there is no knowledge of the structure of the scene, or assumptions that allow the structure to be determined. This assumption is also made because the scene is projected down from the three-dimensional space to the two-dimensional image plane. The only information available in purely computer vision algorithms is this two-dimensional representation of the scene.

The most complicated motion model commonly used in video stabilization is the affine motion model:

$$\begin{bmatrix} x_t \\ y_t \\ 1 \end{bmatrix} = \begin{bmatrix} a_0 & a_1 & a_2 \\ a_3 & a_4 & a_5 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{t-1} \\ y_{t-1} \\ 1 \end{bmatrix}. \tag{16}$$

This model defines zooming, roll, skew due to pitch and yaw, and horizontal and vertical translations. This motion model accommodates many aspects of three-dimensional motion. Zooming represents motion along the Z axis and the skew represents rotations outside of the image plane. This motion model requires at least four non-collinear points to estimate all of the parameters.

The rigid motion model with scale is a special case of the affine model where some of the parameters are constrained. The two constraints $a_0 = a_4$ and $a_3 = -a_1$ remove the ability to account for skew due to pitch and yaw, though the assumption that $\theta \gg \sigma$ and $\theta \gg \psi$ allows us to assume that $\sigma = \psi = 0$, meaning that skew is not a factor. The rigid motion model with scale is:

$$\begin{bmatrix} x_t \\ y_t \\ 1 \end{bmatrix} = \begin{bmatrix} a_0 & a_1 & a_2 \\ -a_1 & a_0 & a_5 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathrm{x_{t-1}} \\ \mathrm{y_{t-1}} \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} x_t \\ y_t \end{bmatrix} = \begin{bmatrix} \gamma cos\theta & \gamma sin\theta \\ -\gamma sin\theta & \gamma cos\theta \end{bmatrix} + \begin{bmatrix} d_x \\ d_y \end{bmatrix} \tag{17}$$

which defines zooming through the scaling factor γ, roll, and translation.

The rigid motion model without scale is a special case of the rigid motion model with scale where the scaling factor $\gamma = 1$. This removes the ability to account for zooming between consecutive frames. The rigid motion model without scale defines roll and translation only:

$$\begin{bmatrix} x_t \\ y_t \end{bmatrix} = \begin{bmatrix} cos\theta & sin\theta \\ -sin\theta & cos\theta \end{bmatrix} + \begin{bmatrix} d_x \\ d_y \end{bmatrix}. \tag{18}$$

The motion model we chose to employ is the rigid motion model without scale from Equation (18). We assume that the framerate is sufficiently high, and that the motion is sufficiently modest in real scenarios that pitch, yaw, and zooming are negligible between consecutive frames. This same assumption guarantees that there will be significant overlap between consecutive frames as well, which ensures that there will be matched SIFT features between consecutive frames to use in motion estimation.

We assume that static background features are more stable than non-static foreground features. This assumption can be made because these non-static objects can move, rotate, deform, change in illumination from moving through a shadow, etc. which might cause them to disappear. Background features are not likely to succumb to these issues over short periods of time since they are affected primarily by camera motion, so they are likely to exist in every consecutive frame.

We assume that the first few frames of the video sequence do not contain local motion to aid in incrementing the trust value of good background features initially so that they are more likely to be picked for motion estimation through the duration of their existence in the video.

4.1.1. SIFT Feature Detection, Matching, and Augmentation

The first step in the stabilization process is to calculate the SIFT features of the current frame and the previous frame. Each iteration of the algorithm only requires the calculation of SIFT features of a single frame, since the previous frame's SIFT features are stored in memory for reuse.

After the SIFT features are calculated, matching must be performed. This is the most time-consuming step of the algorithm. Some of the matches will be incorrect, but outliers are not removed at this stage of the algorithm. The remaining steps of the algorithm will perform outlier removal indirectly in an efficient manner on a smaller set of data. The matching step creates a set F of n matched features.

The SIFT features used in this algorithm are augmented with a trust value. This trust value represents how often the feature has been used in the final estimation of motion for previous frames. Stable features are likely to have higher trust values because the history of a feature only considers the immediate previous frame. If a feature disappears for a single frame and returns, its trust value is reset to the minimum value. Trust value is never lowered when a feature is not used for the final motion estimation, only raised when it is used.

Each matched feature in the current frame inherits the trust value of the matched feature in the previous frame. Any feature without a match is given a nominal trust value which represents the minimum level of trust. A value of zero is not a good candidate for this nominal value due to the fuzzy clustering steps used in the remainder of the algorithm.

4.1.2. Fuzzy Clustering of Feature Rotations

The second step in the video stabilization process is the calculation of the global rotation between the previous and current frame. The goal of this step is to calculate the global rotation of the image and pass along only those features which have rotated consistently with this estimated global rotation.

It is important to note that often local motion will have rotations consistent with the global image rotation unless the feature has local rotation, which is the case if the feature lies on a wheel or other rotating body. This step alone does not completely separate local and global motion itself, which is why a second step of fuzzy clustering must be performed later.

The fuzzy clustering used in this algorithm is not the common c-means fuzzy clustering algorithm [5, [13]. Instead, it is a two-step method consisting first of calculating the *k*-means, and then assigning fuzzy membership values of each feature to each cluster. c-means fuzzy clustering could be used instead, but the intent is to eventually implement this manually in hardware, and k-means clustering of followed by membership assignment is simpler to perform.

The local feature rotations are clustered into k clusters in this step. The value of k does not have a significant impact on the performance of the algorithm due to the fuzzy memberships, so the common $k = \sqrt{n/2}$ is used, where *n* is the total number of matched features between the current and previous frame. The *k*-means algorithm ends with potentially less than *k* clusters because some become empty after several iterations, which is denoted by the variable k'.

The membership assignment is performed in two steps. The first step is to assign each matched feature f_i where $i = 1,2, \dots, n$ an initial membership value with each cluster by calculating the Euclidean distance to each cluster centroid c_j where $j = 1,2, \dots, k'$:

$$\mathrm{M}_0(f_i, c_j) = \frac{1}{\left\| f_{i_{ori}} - c_{j_{ori}} \right\|_2} \forall i, j. \tag{19}$$

The second step is to normalize the membership values for each matched feature so that each matched feature has a total membership value of one across all clusters:

$$\mathrm{M}(f_i, c_j) = \frac{\mathrm{M}_0(f_i, c_j)}{\sum_j \mathrm{M}_0(f_i, c_j)} \ \forall i, j. \tag{20}$$

After clustering is performed, we calculate the most trusted cluster given each matched feature's trust value and membership value for each cluster centroid. To calculate the most trusted cluster, a sum of the trust values or matched features weighted by their membership value to the cluster is determined. The cluster with the highest weighted sum is chosen as the most trusted cluster:

$$c_{best} = \underset{c}{\arg\max} \sum_{i=1}^{n} \mathrm{M}(f_i, c) \cdot \mathrm{T}_{f_i} \tag{21}$$

where T_f is the trust value of the feature f_i.

Once the most trusted cluster is chosen, we must determine which matched features should be considered as members. This is done by selecting those matched features whose membership value to the most trusted cluster is above a threshold value, which is determined based on the number of clusters k' which exist in the current iteration using the formula $t = \frac{1}{2k'}$. The resulting set C is represented as:

$$C = \{f_i | f_i \in F, \mathrm{M}(f_i, c_{best}) \geq t\}. \tag{22}$$

This indirectly removes many outliers by choosing only those features which belong to the best cluster. The size of C is $m \leq n$ since $C \subseteq F$.

To remove more outliers, which can survive the membership thresholding, we employ the interquartile range method [37]. This is a fast, nonlinear way to determine outliers which performs exceptionally well for non-Gaussian sets of data. The first quartile Q_1 is the first datum which is greater than or equal to 25% of the data. The second quartile Q_2 is the first datum which is greater than or equal to 50% of the data, also known as the median value. The third quartile Q_3 is the first datum which is greater than or equal to 75% of the data. The interquartile range is the difference between the first and third quartile:

$$IQR = Q_3 - Q_1. \tag{23}$$

Outliers can be identified by finding values which lie $1.5 * IQR$ above Q_3 or below Q_1. The set G of features which are considered inliers and usable for estimating motion can be found as follows:

$$G = \{f_i | f_i \epsilon C, Q_1 - 1.5 * IQR \leq f_i \leq Q_3 + 1.5 * IQR\} \tag{24}$$

which has size $p \leq m$ since $G \subseteq C$.

The final estimated rotation of the image is then calculated as the mean value of the set G. The membership thresholding is not very restrictive for rotations, so the mean value of the set G is preferred for rotation estimation because the outliers are removed and will not negatively impact the estimation.

4.1.2.1. With Inertial Data

The addition of inertial information can potentially provide a more accurate estimation of orientation because the Euler angle roll (θ) represents a redundant measure for the global rotation of the image. Ideally the angle from the inertial measurement unit

(IMU) or attitude heading reference system (AHRS) will be taken at the exact time that the frame is captured. This is unlikely to occur in perfect synchronization in real scenarios, but the value is often sufficiently close for use. It is very important that the IMU/AHRS is not providing filtered or stabilized results, as this will result in large inaccuracies between the IMU/AHRS and the angular offsets estimated by SIFT.

The best use of the AHRS/IMU roll value is to treat it as an additional data point for the fuzzy clustering step as previously described. The most important factor is the trust value that is placed with this information. If the information is considered very trustworthy, it is a good idea to assign it the maximum trust value of all matched features, and optionally add a flat value or multiply by a scalar value on top of this so that it is highly likely that the best cluster will contain the inertial information. This information can be given a lower value if the IMU/AHRS does not produce very accurate angle measurements.

4.1.3. Calculation of Feature Translations

The third step of the video stabilization process is to calculate the translation vector added to the previous feature's location after rotation. Once the global rotation is calculated, the translation of matched features can be calculated by determining the difference between where the feature would lie in the current frame if only rotation were applied to its location in the previous frame, and where the current feature actually lies in the current frame. The equation for calculating translation is derived from the motion model we employ, and is performed for each feature in the set G:

$$\begin{bmatrix} dx \\ dy \end{bmatrix} = \begin{bmatrix} x_t \\ y_t \end{bmatrix} - \begin{bmatrix} cos\theta & sin\theta \\ -sin\theta & cos\theta \end{bmatrix} \begin{bmatrix} x_{t-1} \\ y_{t-1} \end{bmatrix}. \quad (25)$$

Performing this on all p matched features in the set G yields a set of translation vectors that represent the motion of the objects in the scene. This set consists of both local and global motion as discussed in the Fuzzy Clustering of Feature Rotations section. As such, these vectors must be clustered again to determine the final set of matched features that will be used to calculate the global motion of the camera.

4.1.4. Fuzzy Clustering of Feature Translations

The fourth step of the video stabilization process is to estimate the translation component of the global motion given the set of translation vectors calculated previously. Similar to the fuzzy clustering of matched feature rotations, the set of matched feature translations is put through the same process to estimate the global translation.

The translations are clustered through the *k*-means algorithm, which is extendible to two dimensions easily. The same membership assignment is performed, albeit on a potentially smaller set of *m* matched features, with $k'' \leq \sqrt{m/2}$ cluster centroids. The same equations as before apply to this round of fuzzy clustering, where $i = 1,2, \dots, m$ and $j = 1,2, \dots, k''$:

$$\mathrm{M}_1(f_i, c_j) = \frac{1}{\left\| f_{i_{trans}} - c_j \right\|_2} \quad (26)$$

and the membership values are normalized:

$$\mathrm{M}(f_i, c_j) = \frac{\mathrm{M}_1(f_i, c_j)}{\sum_j \mathrm{M}_1(f_i, c_j)}. \quad (27)$$

The most trusted cluster of translations is now determined by finding the sum of matched feature trust values weighted by their membership value to each of the new clusters. The highest weighted sum is again chosen as the most trusted cluster:

$$c_{best} = \arg\max_{c} \sum_{i=1}^{m} \mathrm{M}(f_i, c)\mathrm{T}_{f_i}. \tag{28}$$

The most trusted matched feature set Z can then be determined by selecting those features with a membership value to c_{best} higher than $t = \frac{1}{k''}$. The set Z can be written as:

$$Z = \{f_i | f_i \in G, \mathrm{M}(f_i, c_{best}) \geq t\}. \tag{29}$$

because $Z \subseteq G \subseteq C \subseteq F$, the size of the set Z is $q \leq p \leq m \leq n$.

We do not perform the same interquartile range outlier removal step on translations as it does not have as large an impact as with the rotations. We calculate the mean of the most trusted cluster for our final estimation of translation. This value will not likely be significantly different from the most trusted cluster's centroid, but it can potentially be different depending on the distribution of cluster centroids and the distribution of data. We believe this mean represents the data more accurately than assuming that the most trusted cluster centroid is exactly correct.

4.1.5. Kalman Filtering

The fifth step in the video stabilization process is to estimate the camera motion which is desired. This motion should not be compensated in the next step of the algorithm, so it should be removed from the total estimated camera motion obtained thus far. One of the more common techniques for estimating desired motion is to perform

Kalman filtering on the data. Kalman filtering should roughly smooth the motion while keeping desired information.

It is important to know the type of motion that the camera will undergo throughout the entire video sequence. The primary concerns are whether or not the camera is supposed to move horizontally or vertically, and whether or not the camera is supposed to rotate. Forward motion should not be included, as it has no direct application to the estimation of camera motion, since the motion has no real impact on the image plane. Forward motion should primarily cancel itself out on the image plane, as it will produce an outward radiating bias with zero mean.

In the case of surveillance using a fixed camera, there is no desired motion. In the case of most robotic applications, horizontal and vertical motions are desired, but rotation is not. In some cases of ground vehicles where the terrain is known to have many incline changes, or with aerial vehicles undergoing complicated maneuvers where the vehicle's body is meant to be in varying orientations, rotation might be desired as the robot is meant to be at an angle at times.

The Kalman filter uses the motion model chosen to be appropriate for the assumed motion of the camera as its state model, and the estimated motion from this algorithm as sensor input. If a time step of one is assumed, the rotation and translation estimated should be treated as velocities, and the state model should include at a minimum both velocity and position. The desired position will be updated at each iteration, and it should be treated as an accumulation of desired motion, which is important for the motion compensation step.

The equations used in Kalman filtering are provided in the Prerequisite Knowledge chapter of this thesis.

4.1.6. Motion Compensation and Update of Trust Values

The sixth and final step of the video stabilization process is motion compensation and updating of values to be used in the next iteration. This step places the image with reference to the desired reference position, effectively smoothing the video to improve its viewability and stability, as well as updating the trust of the features used to estimate the motion from the previous step.

The set Z is considered to be the set of the most trusted values to be used for motion estimation. As such, each member of Z has its trust value incremented to reflect an increase in the trust of each feature. If these features are matched in the next frame, their trust value will be passed on. This method rewards stable features which are not occluded from view at any point in time.

The motion estimated through this algorithm is added to an accumulated total, representing the current position offset from the very first frame of the video:

$$\begin{bmatrix} d_{x_{acc_t}} \\ d_{y_{acc_t}} \\ \theta_{acc_t} \end{bmatrix} = \begin{bmatrix} \cos(\theta_t) & \sin(\theta_t) & 0 \\ -\sin(\theta_t) & \cos(\theta_t) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} d_{x_{acc_{t-1}}} \\ d_{y_{acc_{t-1}}} \\ \theta_{acc_{t-1}} \end{bmatrix} + \begin{bmatrix} d_{x_t} \\ d_{y_t} \\ \theta_t \end{bmatrix} \tag{30}$$

where the subscript acc represents the accumulated value and x_t, y_t, and θ_t are the estimated motion parameters for frame t. This accumulated contains both desired and undesired motions. The desired motion estimated in the previous step should be subtracted from the accumulated motion, which results in the total undesired motion that must be compensated in this step.

To move the current frame in line with the reference position, the rigid motion model must be reversed, and the variables must be replaced with the appropriate values for undesired motion. Each pixel in the image must be moved according to the following equation:

$$\begin{bmatrix} x_{aligned} \\ y_{aligned} \end{bmatrix} = \begin{bmatrix} \cos(\theta_{acc} - \theta_{kd}) & -\sin(\theta_{acc} - \theta_{kd}) \\ sin(\theta_{acc} - \theta_{kd}) & cos(\theta_{acc} - \theta_{kd}) \end{bmatrix} \begin{bmatrix} x_{acc} - x_{kd} \\ y_{acc} - y_{kd} \end{bmatrix} \quad (31)$$

where the subscript $aligned$ represents the location of a pixel after it's been placed in line with the reference position, the subscript acc represents the accumulated total, and the subscript kd represents the desired values obtained from Kalman filtering.

We perform this step by utilizing backward substitution using either nearest-neighbor or bilinear interpolation. Instead of taking each pixel location in the reference frame and moving it to a subpixel location in the resulting frame, each pixel location in the resulting frame is assigned a value based on reversing the angle of rotation and calculating the subpixel value in the reference frame.

4.2. Discussion

This subsection contains discussion of major design decisions which impacted the structure of the video stabilization process. It also contains discussion on issues brought up during the research for this thesis including computational complexity, the impact of separating rotation and translation estimation, and the impact of a center of rotation that is not in the center of the image frame.

Some issues discussed apply to more than just the video stabilization algorithm presented in this thesis. In robotics applications, computational complexity is extremely important due to the need for real-time operation. Also, it is likely that the center of

rotation will not lie in the center of the image frame because the camera is rarely mounted at the robot's center of mass.

We begin the discussion with the design decisions made with our method. We discuss why we use SIFT features, why we use fuzzy clustering, and why we augment SIFT features with trust values. This is followed by a discussion of issues brought up during the development of the algorithm. We discuss how inertial data is used in video stabilization, the computational complexity of our algorithm, the effects of the separation of rotation and translation estimations, and the impact of a biased center of rotation.

4.2.1. Use of SIFT Features

As discussed in the SIFT subsection of the Prerequisite Knowledge chapter of this thesis, SIFT features are widely considered one of the best image features that exist to date. They are very accurate and robust, leading to a much more reliable type of feature than others that exist. The SIFT matching algorithm produces only a small amount of incorrect matches, which is a highly desirable trait for any tracking algorithm.

The major drawback of the SIFT calculation and matching process is its high computational complexity and slow speed. Currently, microprocessors struggle to produce results in real-time. Real-time in this case is defined as 24 frames per second or more, since that is the frequency that most people will perceive as continuous video. This is a major obstacle in scenarios such as mobile robotics and real-time surveillance, since it is largely pointless to have either scenario functioning slower than real-time. Most computer vision algorithms define real-time as fifteen frames per second, which is a much more achievable goal, and is often considered to be sufficient for such tasks,

though commonly used implementations of SIFT cannot successfully produce results at this frequency either.

Even though a microprocessor cannot handle SIFT in real-time, it has been shown that a graphics processing unit (GPU) can, due to the massive parallelization built into graphics processors. It has also been shown [38] that a frequency of 45 frames per second is achievable for calculating SIFT features when using a GPU with the CUDA architecture. For non-CUDA graphics processing, 33 frames per second is achievable. With this in mind, any extra computation, including SIFT matching on the GPU could lower the framerate below 24 frames per second using non-CUDA architecture, though it would likely still lie above fifteen frames per second.

The algorithm we developed has relatively low computational complexity, and could easily run in real-time. The only bottleneck is the use of SIFT features, which are required due to the calculation of rotation in a separate step from calculation of translation. With the appropriate implementation using GPU programming, this algorithm can run in real-time. Appendix B contains sample C++ code that performs this video stabilization method in real-time on videos with VGA resolution or lower using SiftGPU [38]. It is possible for this algorithm to be used on any system that has a large enough profile to house a small motherboard and video card.

4.2.2. Use of Fuzzy Sets

As discussed in the fuzzy set theory section of the Prerequisite Knowledge chapter of this thesis, fuzzy set theory is very useful when there is a level of uncertainty with the data being processed. In this case, fuzzy sets were used primarily to lessen the

burden of using a heuristic method for determining the ideal number of clusters to be used during the two clustering steps of the algorithm.

Heuristic methods tend to require knowledge *a priori* about the type of data or the distribution of the data. If either of these characteristics changes during the course of running the algorithm, the heuristic needs to be changed on the fly as well. Even with this information known, a heuristic will not always provide the ideal solution, only the best available solution given the assumptions made based on the *a priori* knowledge.

Instead, we chose not to bother with any complicated methods of determining the ideal number of clusters. This is a level of uncertainty, and fuzzy set theory handles this situation well. With fuzzy set theory, it does not really make a difference if you have "too many" clusters, though an extremely large amount of clusters will erode performance because data points will become much more exclusive members to specific clusters instead of partial members to many nearby clusters. Like any other clustering method, if there are not enough clusters chosen, this method will succumb to poor results. With this in mind, it was a simple decision to utilize a generous number of clusters $k = \sqrt{n/2}$, which is generally considered sufficient.

The reason slightly overestimating the number of clusters needed is not problematic is due to the partial membership characteristic of fuzzy set theory. Since there is no discrete cutoff for membership in a cluster, all of the nearby features can be members of multiple clusters, meaning that any of those clusters could be chosen depending on the membership values of features and their trust values. Moreover, if any feature has sufficient membership in the chosen cluster, it is passed along in the

algorithm for further calculation. With discrete sets, the number of features passed along will be greatly reduced, leading to increasing trust in much fewer features.

We consider it beneficial to have a larger number of features passed along in the algorithm, eventually having their trust values incremented. If only a few features are highly trusted, there will be a major issue when any of those features disappear, which is likely to be the case in any mobile robot situation. It is not a significant problem that some local motion features will sneak in to the trusted set, because local motion features are less stable and more likely to disappear than good global motion features. More often than not, inconsistent local motion will not be chosen for motion estimation because it is unlikely that its membership to a good cluster is sufficiently high. The more background features that lie within the chosen set, the more likely it is that future background features will become trusted when they appear.

4.2.3. Use of Trust Value Augmentation

Augmenting the SIFT features with trust values is our attempt to create a gap between trustworthy background features and untrustworthy foreground features, which may be close to the camera or be non-static objects, so they can easily be separated from each other to improve the accuracy of motion estimation. The idea behind the trust value is that the more often a feature is used for the final motion estimation step, the more it should be trusted to estimate motion in the future.

Technically, this idea could potentially lead to issues where if untrustworthy features are chosen early, they will become more and more trusted when they should not, resulting in poor video stabilization quality. Two assumptions address this shortcoming:

there should be no local motion in the first few frames of video, and background features are more stable than foreground features.

This first assumption is made in many video stabilization algorithms, and is a convenient way to seed the correct features with higher trust values. It is not an unreasonable assumption to make. Depending on the application, there is often a large portion of frames where local motion does not occur. In some situations, such as monitoring of steady traffic, there is no guarantee that local motion will not occur. This situation has not been tested, nor has our algorithm been designed to handle it.

The second assumption comes from a combination of common sense, and the experience of many computer vision researchers. It makes sense that an object in the scene which does not move will be recognized more easily and more often. Being recognized consistently and consecutively is considered stable. On the other hand, objects which have local motion are less likely to be recognized as often. They might move through shadows, change orientation, or even move completely out of the scene. These possibilities all lead to a less stable class of features.

It is likely that, more often than not, there are more background features than foreground features. Moving objects generally cover a small portion of the screen, which usually yields fewer features. Although uncommon, we did not want to make the assumption that this would occur in every frame. Certain scenes will consist of a large portion of local motion, or an object will move very close to the camera, consuming a much larger portion of the scene than the background. As long as some background features are discovered in each frame, our stabilization algorithm should succeed.

4.2.4. Use of Inertial Information

It is not common practice to include inertial information from an attitude and heading reference system (AHRS) or inertial measuring unit (IMU) in video stabilization algorithms. Prohibitive cost and space requirements are the most common reasons for excluding inertial information from video stabilization algorithms. An accurate and precise IMU or AHRS is too expensive for many commercial applications, especially for handheld cameras. This is not the case for robotic systems. Most robotic systems have some form of inertial measurement capability, often used for navigation and three-dimensional orientation. It is possible to take this same information and inject it into a video stabilization algorithm.

Some cases of inertial information being used for video stabilization are presented in the Literature Review chapter. These algorithms use iterative least-squares methods for calculating motion and use this inertial information as a seed value. This effectively reduces the number of iterations necessary to reach the global minimum error because the starting point is relatively close to the optimal value. This only provides improved motion estimation if the global minimum error cannot be reached in the maximum number of iterations allowed by the algorithm designer when fed a poor seed value. This is not true sensor fusion and has no real impact on video stabilization quality in general.

We attempted to find a way to use this inertial information in a useful way. For rotation, we included IMU/AHRS roll in the algorithm as a trusted data point for fuzzy clustering, which has an actual impact on the final estimation of rotation. This is possible because roll represents the same information as camera rotation. The inertial information

provided by the AHRS we used was less accurate than SIFT features alone, which can be seen in the BOOKSHELF2 Video section of the Experimental Results chapter.

Inertial information must be used differently in the case of translation estimation. Unlike the roll value given by the IMU/AHRS, there is no direct equivalence of any other information that the IMU/AHRS can provide to motion of the image plane. The horizontal translation consists of a combination of horizontal linear motion and the Euler angle yaw (ψ). Similarly, vertical motion consists of a combination of vertical linear motion and the Euler angle pitch (φ).

To use this information directly, knowledge of the structure of the scene is necessary. Linear motion has a different impact on objects which are at different distances from the camera. It imposes a change of perspective which cannot be fully compensated, which can be seen in the LAB Video section of the Experimental Results chapter very clearly. The two relevant Euler angles can be used by assuming that the objects in the scene have translation due to angular motion of the camera but no translation due to linear motion [17]. We do not want to make any assumptions about the motion of objects in the scene outside of the Kalman filtering process, so we cannot guarantee that only Euler angles are needed for estimation of translation. For this reason, inertial information was not included in our stabilization process for translation estimation.

In general, calculating the offsets using digital video data without inertial information works well. We believe the best use of this information is to aid in separating global and local motion. Inertial information provides motion data which directly

correlates with the motion of the camera, which can help identify which matched features are more likely to represent global motion. We leave this belief as research to be performed in the future.

4.2.5. Computational Complexity

If our algorithm is ever to run in real-time, then the computational complexity of the entire video stabilization process must be considered. We do not analyze the SIFT algorithm or feature matching, but we do analyze the entire stabilization process after all point correspondences are matched. SIFT is very computationally expensive, and we show that it is the bottleneck that prevents this algorithm from running in real-time.

The first k-means clustering step calculates the distance from each feature to each cluster centroid in each iteration, up to the maximum number of iterations. This takes $I \cdot n \cdot c$ time, where I is the maximum number of iterations allowed and is constant, n is the total number of matched SIFT features, and $c \leq \sqrt{n/2}$ is the number of clusters calculated. The less than or equal sign is there because the k-means algorithm sometimes removes clusters when they become empty. The worst case time is then:

$$O(I \cdot n \cdot c) = O\left(I \cdot n \cdot \sqrt{n}\right) = O(n^{1.5}). \tag{32}$$

The first membership assignment step calculates the distance between each feature to each cluster centroid as an initial weight. It then normalizes the membership of each feature to a total membership of 1. This takes $n \cdot c + n \cdot c = 2 \cdot n \cdot c$ time, where n and c are the same as in Equation (32), leading to a worst-case time of:

$$O(2 \cdot n \cdot c) = O\left(2 \cdot n \cdot \sqrt{n}\right) = O(n^{1.5}). \tag{33}$$

The first best cluster calculation step determines the sum of all feature trust values weighted by their membership to each cluster, finds the maximum value, and then assigns features with a high enough membership values as a member of the best cluster. This takes $n \cdot c + c + n$ time, which is, in the worst case:

$$O(n \cdot c + c + n) = O\left(n \cdot \sqrt{n} + \sqrt{n} + n\right) = O(n^{1.5}). \quad (34)$$

This best cluster has a size of $m \leq n$, and outlier removal is performed on this set of data. The outlier removal step can be performed using a randomized algorithm to find the quartiles in expected linear time with respect to the size of the set, followed by checking each value to determine if it is an outlier:

$$O(2 \cdot m + m) = O(3 \cdot m) = O(m) \quad (35)$$

which results in a set of size p and to determine the mean location, the worst case time is:

$$O(p). \quad (36)$$

Calculation of the resultant vectors for translation calculation multiplies each of the p feature locations by the rotation matrix, which has a constant R number of calculations, and finds the difference between this location and the matched feature's location, which is $R \cdot p$ time. The worst case time of this step is:

$$O(R \cdot p) = O(p). \quad (37)$$

Similar to the first k-means clustering, membership assignment, and best cluster calculation steps, the second set of these steps is repeated with p matched features instead of n, noting that the number of clusters is now $k \leq \sqrt{m/2}$, leading to a worst case time of:

$$O(I \cdot p \cdot c) + O(p \cdot c + p \cdot c) + O(p \cdot c + c + p) = \quad (38)$$

$$O(p^{1.5}) + O(p^{1.5}) + O(p^{1.5}) = O(p^{1.5}). \quad (39)$$

The translation calculation then finds the mean of the best cluster of translations with size s in two dimensions:

$$O(2 \cdot s) = O(s). \quad (40)$$

The Kalman filtering step performs several fixed-size matrix multiplications, additions, and inversions. Because this is a small, fixed number of calculations per iteration, and is independent of the number of matched features, the worst-case time is:

$$O(1). \quad (41)$$

Motion compensation by backward projection multiplies $w \cdot h$ pixel locations by a rotation matrix, which is again a fixed number of calculations R. Bilinear interpolation is performed at each pixel location, which is a fixed number of calculations $O(1)$. The worst-case time of the compensation step is:

$$O(R \cdot w \cdot h) = O(w \cdot h). \quad (42)$$

The overall worst-case time of the entire algorithm, except SIFT-related steps is:

$$O(n^{1.5} + n + m + p^{1.5} + p + s + w \cdot h + 1) = O(n^{1.5} + w \cdot h) \quad (43)$$

which is less than quadratic in the number of matches, and linear in the total number of pixels in each frame. This low computational complexity leads us to the conclusion that the algorithm could easily be run in real-time if SIFT calculation and matching are also performed in real-time.

4.2.6. Separation of Rotation and Translation Estimation

One of the most important issues that have been brought up during the development of our video stabilization algorithm is what impact separating the estimation

of rotation and translation would have on the quality of stabilization. None of the video stabilization algorithms in the Literature Review chapter performed these estimation steps separately, leading us to the assumption that there might be a disadvantage to doing so.

The primary reason we separate rotation from translation is because we have the ability to do so given the orientation information that SIFT features provide. This is one of the reasons we chose to use SIFT features in the first place. We believed that by guaranteeing an accurate rotation, the accuracy of estimating translation afterwards would be improved. We learned that this is not necessarily true in terms of numerical error, however, we do believe that a separate rotation estimation using SIFT orientation is less susceptible to noise in pixel location and achieves a higher visual quality. The biggest benefit of this separation is that there is no preference of the location in the frame of matched features.

Most methods which perform these estimations together are least-squares approaches, which perform poorly in the presence of outliers. For this reason, large magnitude motion is discarded before any motion estimation is performed. When rotation occurs, the edges and corners of the frame are affected most, and are likely to have a large magnitude of motion. This puts an emphasis on the center of the frame for motion estimation.

There is no agreement in the video stabilization literature about which location in the frame is better for estimating motion. It is highly dependent on the scenario which parts of the frame are good or bad. In many situations, the center of the frame is more likely to have the infinite horizon, and the further away an object is, the more accurately

camera motion can be estimated. Objects around the edges of the screen are likely to be closer to the camera than those in the center. This would suggest that the center of the frame is indeed a better candidate for estimating motion. On the other hand, many situations have a significant portion of the local motion in the center of the frame, as that is the focus of the video. If more global motion occurs near the edges of the frame without moving out of the viewing window, that would suggest that the edges and corners are better candidates for estimating motion, because there is less local motion skewing the estimation.

Having an accurate estimation of rotation before considering translation removes the need to discard large magnitude motion and effectively removes any preference on the location of features in the frame. The large magnitudes are not important because each feature's rotated location is calculated, removing the magnification of motion around the edges and corners of the frame. All that is left after this step is purely translation, which should be consistent, among background features, regardless of the location in the frame.

Why do other methods, even those that use SIFT features, not separate these calculations? They almost all choose to use a least-squares approach, which calculates rotation and translation in the same step. In reality, this least-squares approach effectively calculates rotation before calculating translation. The most obvious example of this is Procrustes analysis [6], which is an often used least-squares method in computer vision algorithms. First, scale is optionally normalized. After this, the centroid of the cloud of points is moved to the origin, and the rotation necessary for the two clouds of points to

match is calculated. Finally, this estimated rotation is used to calculate the translation between the centroids of the original clouds of points.

Based on this knowledge of the least-squares approach, there are no detrimental effects introduced by separating these motion estimation steps in our algorithm. The only real difference between our approach and the least-squares approach is that the estimation of rotation is not affected by noise in the final location of the feature after rotation and translation. The least-squares approach estimates the best rotation and translation that provides the most accurate mapping of feature locations, which can be negatively affected by noise in the location before or after rotation and translation. The method for orientation calculation in SIFT features is separated from location, causing our rotation calculation to be less prone to errors due to mapping.

4.2.7. Impact of Biased Center of Rotation

In mobile robot scenarios, it is often the case that the camera is not mounted on the center of rotation of the robot itself. The original inspiration of this research was video stabilization for robotic systems, so the biased center of rotation will likely come into play, and any issues it poses must be addressed. How does a biased center of rotation affect the estimation of motion when assuming that there is none, and how does it affect the visual quality of the video stabilization results?

Imposing this bias in the center of rotation effectively changes the equation for motion to a slightly more complex one. The bias must be added to the previous frame location as well as the current frame location:

$$\begin{bmatrix} x_t \\ y_t \end{bmatrix} + \begin{bmatrix} b_x \\ b_y \end{bmatrix} = \begin{bmatrix} cos\theta_t & sin\theta_t \\ -sin\theta_t & cos\theta_t \end{bmatrix} \left(\begin{bmatrix} x_{t-1} \\ y_{t-1} \end{bmatrix} + \begin{bmatrix} b_x \\ b_y \end{bmatrix} \right) + \begin{bmatrix} dx \\ dy \end{bmatrix} \quad (44)$$

where b_x and b_y is the bias in the x and y directions respectively.

The first aspect of motion that must be considered is the rotation estimation. A biased center of rotation has no impact on the estimation of rotation. It does not matter how far away from the center of rotation a feature lies, the change in orientation will be exactly the same. Any cloud of points rotated, with or without bias, will keep its relative shape information while its orientation will change by exactly the angle of rotation. This is apparent in Equation (44), since the biasing factor has no impact on θ.

The second aspect of motion to be considered is the translation. Clearly, if rotation is unaffected by the added bias, then the translation estimation must be affected instead. The motion model can be rewritten as:

$$\begin{bmatrix} x_t \\ y_t \end{bmatrix} = \begin{bmatrix} cos\theta_t & sin\theta_t \\ -sin\theta_t & cos\theta_t \end{bmatrix} \begin{bmatrix} x_{t-1} \\ y_{t-1} \end{bmatrix} + \begin{bmatrix} B_x \\ B_y \end{bmatrix} \quad (45)$$

where B_x and B_y are defined as:

$$\begin{bmatrix} B_x \\ B_y \end{bmatrix} = \begin{bmatrix} d_x \\ d_y \end{bmatrix} + \begin{bmatrix} \cos\theta_t & sin\theta_t \\ -sin\theta_t & cos\theta_t \end{bmatrix} \begin{bmatrix} b_x \\ b_y \end{bmatrix} - \begin{bmatrix} b_x \\ b_y \end{bmatrix}. \quad (46)$$

Equation (45) is exactly equivalent to the rigid motion model as shown in Equation (18) with a modified translation value. Equation (46) shows that this modified translation value contains a combination of actual translation and error in translation due to bias. The fact that this translation contains error due to bias makes no difference in the ability for the motion model to describe the exact motion between consecutive frames. It is not important to know the bias in the center of rotation, and it does not matter if the bias changes during the duration of the video. The points are mapped properly between

frames, accounting for both translation and error due to bias at the same time, though not calculating either one directly.

CHAPTER 5. EXPERIMENTAL RESULTS

Presenting experimental results of video stabilization is not an easy task. There is no good quantitative measure for stabilization quality, and therefore, no good way of comparing video stabilization algorithms. The most often adopted quantitative measure of stabilization quality is the peak signal-to-noise ratio (PSNR), though it only partially applies to video stabilization. We will use PSNR to demonstrate that our stabilization process improves the quality of the videos quantitatively because more accurate means of quantitative measurement do not currently exist.

The formula for calculating PSNR is:

$$PSNR = 10\log\left(\frac{MAX^2}{MSE}\right) \tag{47}$$

where MAX is the maximum signal value, and MSE is the mean squared error over the entire frame. The maximum signal value in images is most often a value of 255, representing a white pixel in grayscale. The mean squared error between images I and J can be calculated using the formula:

$$\frac{1}{hw}\sum_{i=0}^{h-1}\sum_{j=0}^{w-1}|I(i,j)-J(i,j)|^2 \tag{48}$$

where h and w represent the height and width of the image respectively.

PSNR is primarily used in image or video compression quantitative measurement. It is used to compare the original image, and an image that has been compressed and

uncompressed. This measures how different a signal is after the compression and decompression process.

For video stabilization, PSNR is calculated by comparing the current frame to the first frame in the video sequence. The reason PSNR does not completely apply to video stabilization is because PSNR is meant to compare equivalent signals, which represent the exact same information. Video stabilization produces a similar, but not necessarily equivalent signal. Perspective changes, local motion, desired motion, and incomplete information lead to this lack of equivalence.

We will use PSNR for the videos which do not contain desired motion, as PSNR is more appropriate for these videos than others. The PSNR of the original, unstable, video will be compared with the PSNR of the stabilized video, and only on valid pixel locations in both cases. Frames being compared are incomplete because some of the information is missing and replaced with black pixels. Only those pixels which represent the actual frame are valid.

PSNR is generally not a means for comparing different video stabilization algorithms, or even the same algorithm using different parameters. The masking method is highly dependent on the calculated offsets of motion estimation, which will be different for different methods or parameters, leading the calculation of PSNR between them to be unrelated. If all of the offsets are known and all of the stabilized videos are available, it is possible to create a combined mask which contains the intersection of each masks valid pixels. It is a viable measure to show that the stabilized video has improved quality over the original in all cases.

Valid locations have been calculated by producing a mask and applying the appropriate rotation and translation for each frame. This mask has a value of 1 at valid pixel locations and a value of less than 1 at invalid pixel locations. It is not guaranteed invalid pixels will have a value of zero because bilinear interpolation is performed for this rotation.

Introduced in [3] is the idea of the interframe transformation fidelity (ITF). It is a singular measure for the quality of an entire stabilized video based on the PSNR. This value is effectively the mean of each frame's PSNR after the first, since the first frame is exactly the same as in the original video, leading to an infinite PSNR value. Because it is derived from PSNR, the ITF value has the same drawbacks that PSNR have with regards to measuring the quality of video stabilization. We will use this value to compare the original and stabilized videos where applicable.

Visual quality of stabilized videos can currently only be measured qualitatively in many cases. As such, we will discuss each video's results with regard to landmarks within the scene. Humans tend to look at how stationary the static objects appear to determine how stable a video is, so the discussion will focus on specific objects in the scene and how they have moved.

Most video sequence discussion sections are accompanied by a side-by-side comparison of several frames from the original and stabilized videos. These figures will be at the end of each video sequence section. Included with this thesis as supplemental material is the original and stabilized videos stacked vertically for each video sequence at a framerate of fifteen frames per second. To really understand what effect video

stabilization has on these video sequences, the videos supplied as supplemental material should be viewed.

5.1. Video Sequences

Six of the videos used in this research are not accompanied by inertial data. These videos are a proof of concept that this stabilization process can successfully produce results on its own without any external information necessary. We captured the LAB video in our lab room. Three of the videos were acquired from [40]. These videos are the ONDESK, STREET, and ONROAD sequences. The fifth video was taken from the Youtube Video Community [1], which we named BASEJUMP. The sixth and final video without inertial data, BOOKSHELF, we took in one of the University of Denver Unmanned Systems Lab's (DU^2SL) offices.

One video has been included with inertial data to test the use of the AHRS roll value as a trusted data point. The BOOKSHELF2 Video is a short video sequence which is accompanied by the roll provided by an AHRS.

5.1.1. LAB Video

The LAB video has a resolution of 640x480 and a framerate of 29.97 frames per second. The total duration of the video is about 5 seconds with a total of 144 frames. The video is set indoors and is taken from a handheld camera. The scene is a view of the back of the Computer Vision lab at the University of Denver. The primary objects of interest in this video are the black computer tower on the left side of the scene, the printer along the back wall, the door next to the printer, and the nearby chair in the bottom of the scene.

This video consists of no local motion and no desired motion. There is very little rotation in this video, and it is primarily used to demonstrate the translation portion of the video stabilization process. There is a secondary product of this video as well. The chair in the scene is very close to the camera, leading to a large perspective change as the camera is moved. This perspective change has an effect on both the quantitative and qualitative value of this video.

Figure 2 shows that the PSNR value of stable LAB video tends to be higher than the PSNR value of the original. There are a few peaks where the original video spikes very high. This is likely due to the camera approaching its original location and orientation causing the signal to approach its original state. The stable video does not peak at the same locations most likely due to an accumulated error causing the frame to not match as well. The ITF value of the original video is 11.24dB and the ITF value of the stabilized video is 19.42dB. This is an increase of 8.18dB.

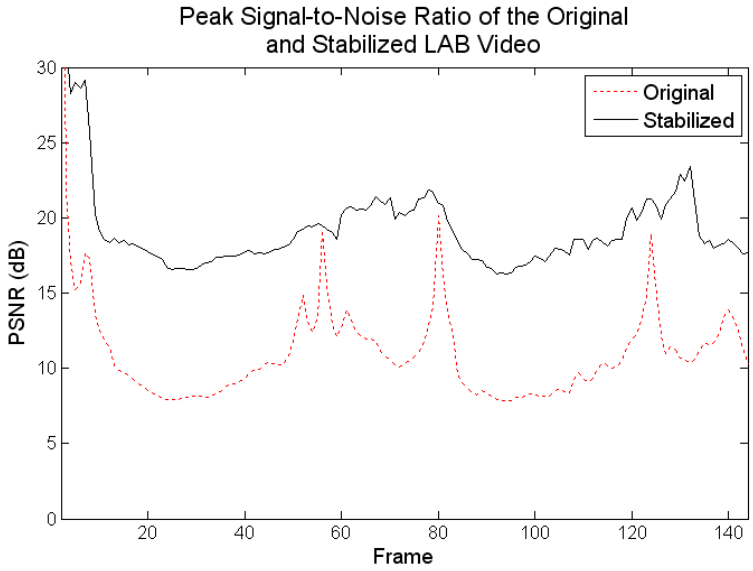


Figure 2: The peak signal-to-noise ratio, in decibels, of the original and stable LAB videos.

Comparing the original and stabilized lab video, as shown in Figure 3, it is apparent that there is a significant perspective change when looking at the chair in the bottom right of the frame. The chair is very close to the camera, so the horizontal translation of the camera changed the perspective significantly. This is an example of when the static objects in the scene are not at a consistent depth, which breaks the assumption made during the development of our algorithm. It is apparent that this has some impact on the quality of stabilization, though the resulting video is more stable overall than the original.

Both the computer tower and the printer appear to have relatively little change in location, though the printer is more stable than the computer tower. This is again due to the relative depth of the two objects from the camera. It is clear that these objects move slightly, which is likely due to the significant difference in depth adding biases to the motion estimation. The perspective change causes a change in the size of some of the static objects, leading to different levels of bias over time. Overall, this is the least visually appealing stabilization.

(a) (b)

Figure 3: Selected frames from the (a) original LAB video, and the (b) corresponding frames from the stable LAB video.

5.1.2. ONDESK Video

The ONDESK video is the first of three videos acquired from [40]. This video has a resolution of 160x120, and framerate of 15 frames per second. The total length of this video is about 10 seconds with a total of 146 frames. The video is set indoors and is taken from a handheld camera. The content in this video consists of several items resting on top of a white desk with no texture. The background of the video is a gray wall with a small amount of texture and a vertical separator. The most important of the items on the desk are the journal, which is standing upright and leaning on the wall, the tall stack of books and notebooks, and the small piece of paper which lies closest to the camera.

This video contains no local motion and no desired motion. This video provides a baseline for simple video stabilization by supplying only translations and rotations. The rotations in this video are of higher magnitude than those in the LAB video. The primary purpose of the ONDESK video is to demonstrate the accuracy of the rotation estimation in our video stabilization process. Another product of this video is that it demonstrates successful video stabilization in a very low resolution video.

The results of the ONDESK video are better than the results of the LAB video. This improvement is likely due to the relatively insignificant depth difference between static objects used to estimate motion. The PSNR values of the original and stabilized ONDESK videos are shown in Figure 4. The ITF value of the original ONDESK video is 16.73dB and the ITF value of the stable video is 23.53dB. This is an overall increase of 6.80dB.

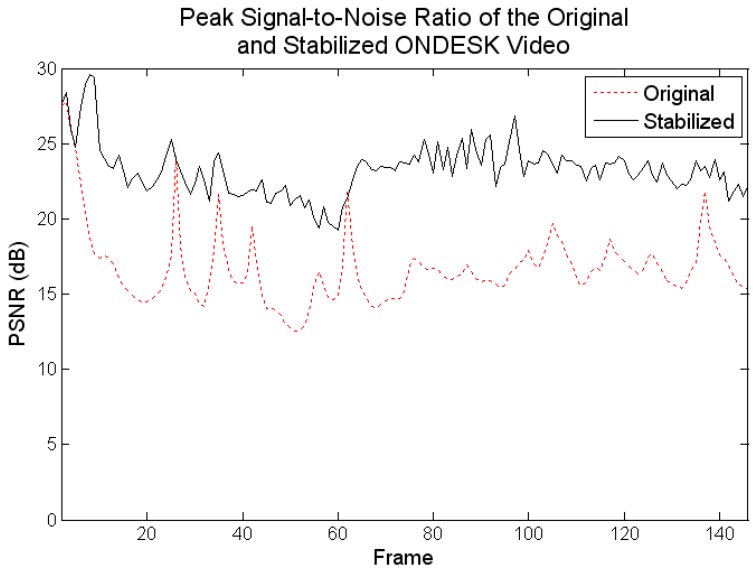
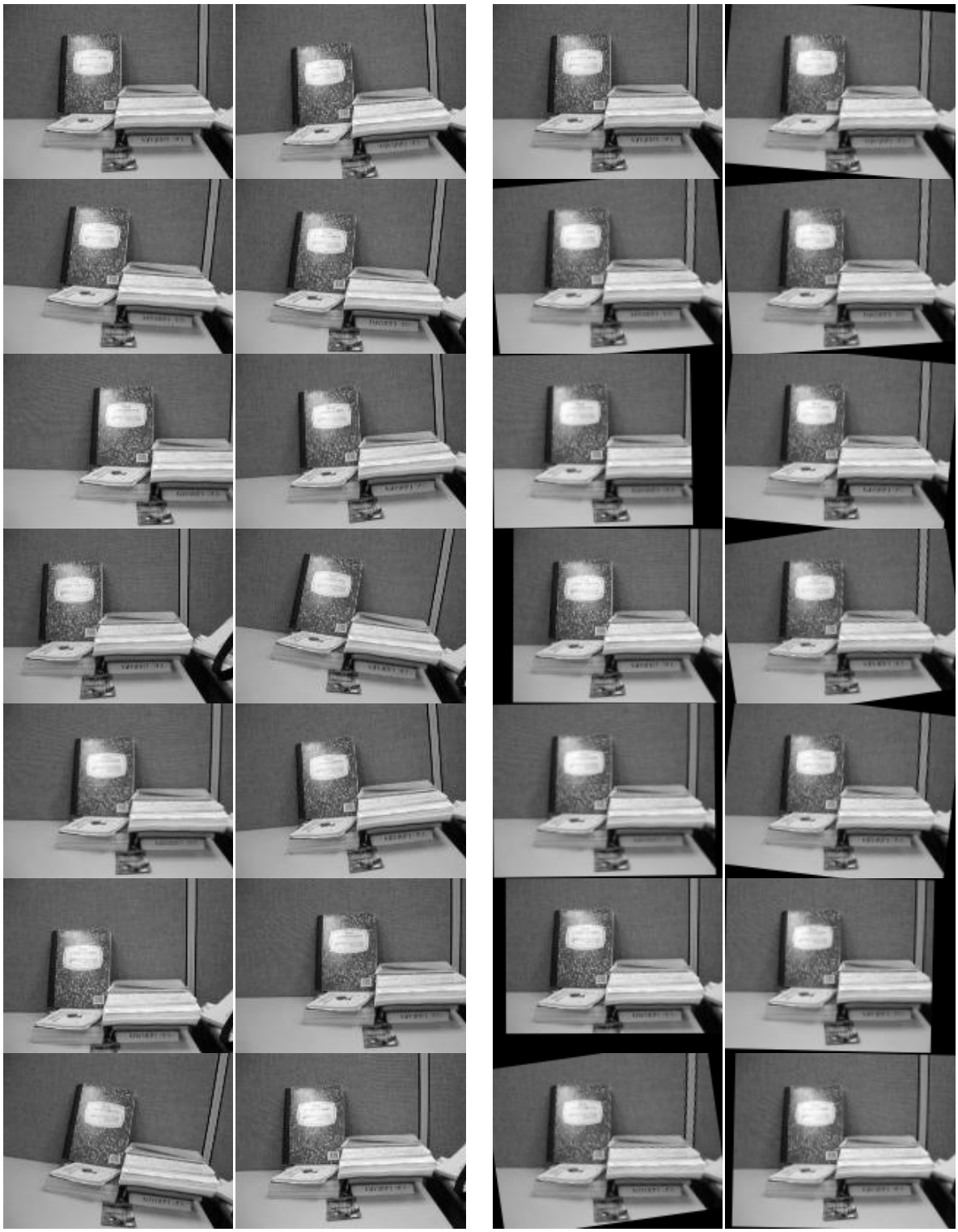


Figure 4: The peak signal-to-noise ratio, in decibels, of the original and stable ONDESK videos.

The notebook in the stabilized video appears to not move much at all as can be seen in Figure 5. The only noticeable difference throughout the video while looking at the notebook is the change in light reflection as the camera moves, but its location remains fixed. It is clear that there is slight movement of the large stack of books and even more movement of the small piece of paper closest to the camera. It appears that these objects did not have a significant impact on the stabilization, so most of the features chosen for stabilization must have belonged to the textured wall in the background and the notebook. Due to the significant rotation in the video, this result demonstrates that the orientation component of SIFT features is just as accurate as the translational components, validating its direct usage to calculate rotation.



(a) (b)

Figure 5: Selected frames from the (a) original ONDESK video, and the (b) corresponding frames from the stable ONDESK video.

5.1.3. STREET Video

The second video obtained from [40] is the STREET video. It also has a resolution of 160x120 and a framerate of 15 frames per second. The total length of this video is 4 seconds with a total of 60 frames. This video is set outdoors and is taken from a handheld camera. The video consists of a view across a street from a street corner. The primary objects of interest consist of the large building across the street, the tree and pole in from of the building, and a parked car on the street. The majority of the objects of interest lie in the left half of the scene. The right half of the scene is a view down the street next to the large building with several light poles at different depths.

This video consists of no desired motion. There is a car that drives through the scene after about 1 second, which introduces local motion which must be detected and discarded from the camera motion estimation. The video consists of primarily rotations and slight translations at the beginning of the video followed by primarily larger translations following the car and recentering the camera toward the end of the video. The purpose of including the STREET video is to demonstrate the successful separation of local and global motion using the augmented SIFT features and the fuzzy clustering method. The byproduct of this video is that it demonstrates stabilization outdoors.

Figure 6 shows that the stabilized video always has a higher PSNR value than the original video. There is a large dip in the center of the video due to a combination of the car driving through the scene and the camera panning to the left slightly following the car. The PSNR value rises again toward the end when the camera moves back toward the original location and the car is out of the scene. The ITF value of the original STREET

video is 13.89dB and the ITF value of the stabilized video is 21.21dB, which is an increase of 7.32dB.

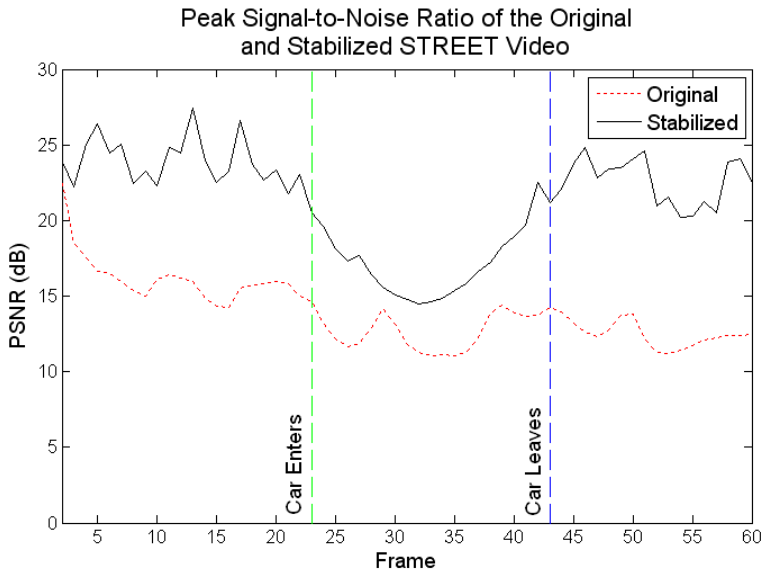


Figure 6: The peak signal-to-noise ratio, in decibels, of the original and stable STREET videos.

The STREET video shows similar results to the ONDESK video. Significant jitter in orientation occurs throughout the video, but our stabilization process handles it gracefully. The parked car and tree appear to remain stationary throughout the entire video as can be seen in Figure 7. Once again, the relatively small difference in the depth of static objects in the scene allows for the camera motion to be estimated accurately. This video demonstrates the successful separation of local and global motion due to the car driving through the scene. Throughout the time the car is driving through the scene, it is apparent that the building in the background remains fixed. The fuzzy clustering method we use successfully performs the desired separation.

(a) (b)

Figure 7: Selected frames from the (a) original STREET video, and the (b) corresponding frames from the stable STREET video.

5.1.4. ONROAD Video

The third and final video taken from [40] is the ONROAD video. This video has a resolution of 160x120 and a framerate of 15 frames per second. The total duration of the video is about 13 seconds with a total of 201 frames. This video is set outdoors and is taken with a handheld camera. The video is taken by a person walking forward along the sidewalk. The primary objects of interest are the prominent sidewalk down the center of the frame, the buildings on the left side of the sidewalk and the several trees and poles on either side of the sidewalk.

This video has no local motion and no desired motion. The translations and rotations are very high frequency, which causes the video to be difficult to view. Although the video has forward motion, this does not represent a desired motion with respect to the image plane of the camera. This video was included in this research because it represents a very common scenario where the camera might be mounted on a ground robot or person moving forward over rough terrain. It demonstrates that forward motion does not have a significant impact on the stabilization process when vibrations are prominent.

Figure 8 shows that the stabilized ONROAD video has a drastic decrease in jitter from the original video, even though the jitter is high frequency and relatively large magnitude. The sidewalk in the street remains in a fixed location, making it appear that the camera is moving smoothly forward. The forward motion did not have a negative impact on motion estimation even though it skews motion vectors outwards toward the edges.

(a) (b)

Figure 8: Selected frames from the (a) original ONROAD video, and the (b) corresponding frames from the stable ONROAD video.

5.1.5. BASEJUMP Video

The fifth video without inertial data is the BASEJUMP video, which is an excerpt of the base jump video [1] taken from the Youtube Video Community. It has a resolution of 320x240 and a framerate of 15 frames per second. The total duration is about 27 seconds with a total of 400 frames. This video is set outdoors and is taken with a camera mounted on the base jumper's helmet. The video is taken by a person performing a base jump off a cliff over a large natural landscape. The beginning and end of the video do not contain the desired video content, so a portion of the video during the actual jump was extracted for the purpose of this research. There are not many landmarks in this video, so the discussion will largely consist of comparing how smooth the motion is in the stabilized video compared to the original video.

This video consists of no local motion. The vertical translation is desired, and the horizontal translation is partially desired. This partial desire is reflected in the state update model used in the Kalman filter. No rotation is desired, and should all be compensated. This video was included in this research as a means of testing the Kalman filtering process for estimating desired motion, which allows for the stabilization of a moving camera. The reason we chose a base jump video for testing is because it has motion representative of a downward-facing camera attached to an aerial vehicle, such as a robotic unmanned aerial vehicle.

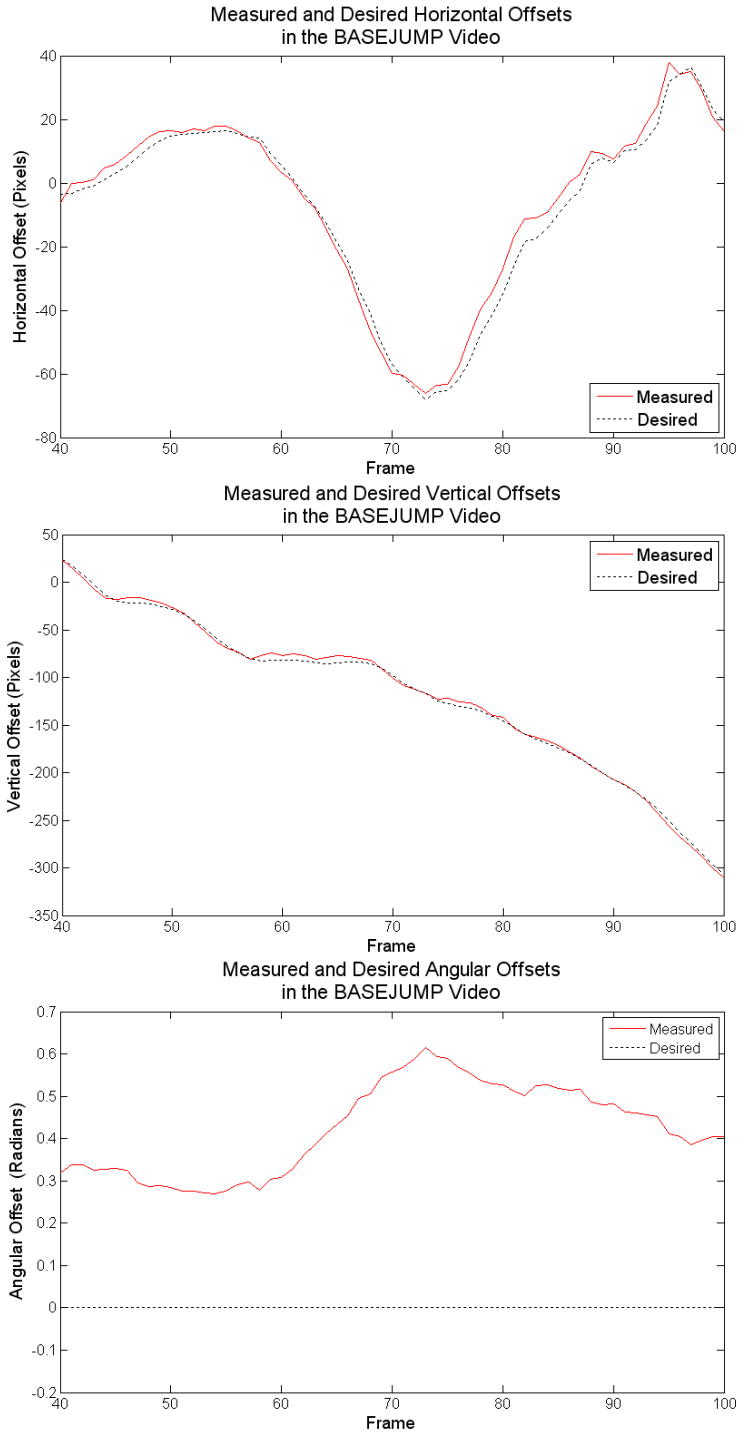


Figure 9: The measured and desired horizontal, vertical, and angular offsets of selected frames from the BASEJUMP video.

We believe that the stabilized BASEJUMP video has improved viewability over the original. Figure 9 shows the graphs of the measured and desired horizontal, vertical, and angular offsets of the BASEJUMP video. Figure 10 shows selected frames from the original and stable BASEJUMP videos. The desired horizontal and vertical offsets are smoother than the measured offsets. Peaks are less pronounced, but the motion tracks the sensed data closely. Because no angular motion is desired, the desired curve is a constant zero.

Without determining an appropriate state model for the Kalman filter, the video could move beyond the viewing frame, leaving the video completely unviewable after a short period of time. The state model that we use in the stabilization of the BASEJUMP video is:

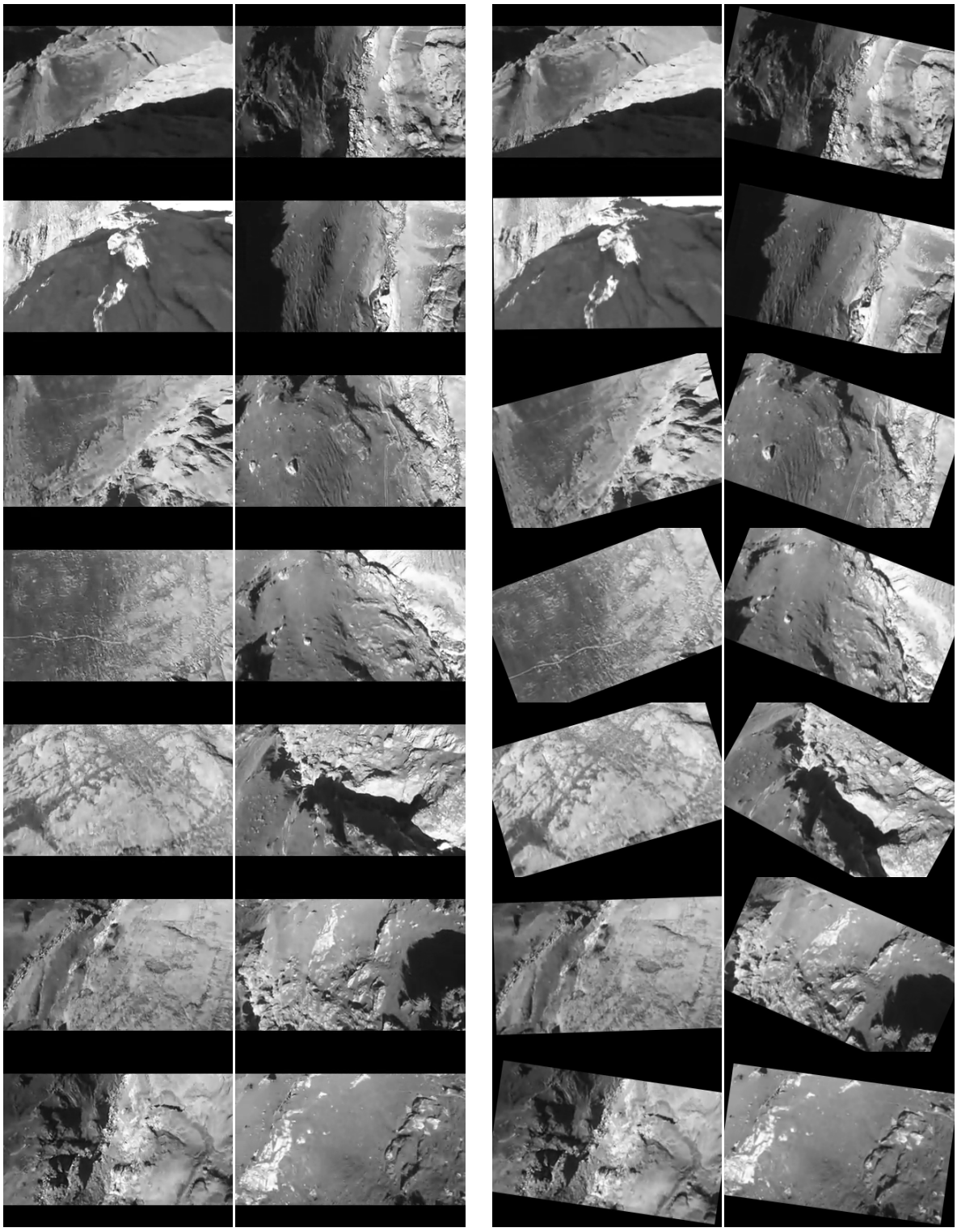
$$\begin{bmatrix} \cos(\hat{\theta}_t) & \sin(\hat{\theta}_t) & 0 & 1 & 0 & 0 \\ -\sin(\hat{\theta}_t) & \cos(\hat{\theta}_t) & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ \theta \\ \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} \tag{49}$$

where $\hat{\theta}_t$ represents the estimated rotation at time t, $x, y,$ and θ represent the horizontal, vertical, and angular offset respectively, and a single dot represents the velocity of the variable. We rotate the state vector of the Kalman filter with respect to the estimated motion because the translations are stored with respect to the new orientation at each frame. The sensing matrix we use is:

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ \theta \\ \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} \tag{50}$$

where the estimated translation represents the horizontal and vertical velocity and the rotation represents the angular velocity.

Due to the desired motion and the constantly changing background, PSNR and ITF are not applicable to the BASEJUMP video. As such, we have no quantitative measure of the quality of video stabilization in this case.



(a) (b)

Figure 10: Selected frames from the (a) original BASEJUMP video, and the (b) corresponding frames from the stable BASEJUMP video.

5.1.6. BOOKSHELF Video

The final video without inertial data is the BOOKSHELF video which we took of a bookshelf in the University of Denver's Unmanned Systems Lab (DU^2SL). It has a resolution of 640x480 and a framerate of 29.97fps. The total duration is about 11 seconds with a total of 332 frames. This video is set indoors and was taken using the cameras mounted to a USL ground robot. We imposed rotations and translations by holding the robot and moving it around in the air. There is a combination of linear translations and turning of the robot to obtain horizontal and vertical offsets. The camera is not mounted on the robot's center of mass, so the center of rotation is biased.

This video consists of no desired or local motion. This video was included in this research as a means of demonstrating that a biased center of rotation has no impact on the quality of video stabilization. A biased center of rotation is likely to occur in practice for robotics applications. It is less likely to occur while holding a handheld camera, but there are some scenarios that would cause this bias to occur even in this case. Frames from the original and stabilized videos are shown in Figure 12.

The stabilization of the BOOKSHELF video correctly removes the rotation and translation, including translation resulting from the biased center of rotation, for all frames. The bookshelf in the stabilized video does not appear to move significantly from its original location. Many frames were blurry due to quick motion, but this had little impact on the results due to the robustness of SIFT features.

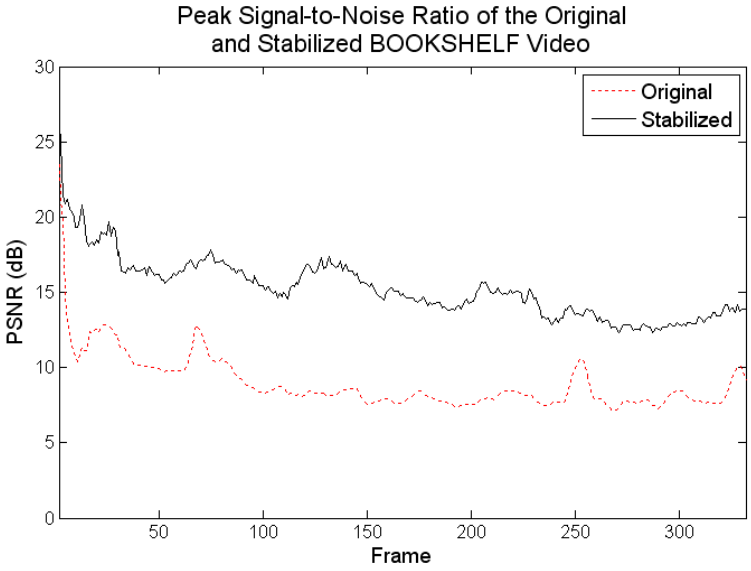


Figure 11: The peak signal-to-noise ratio, in decibels, of the original and stabilized BOOKSHELF videos.

Figure 11 shows the PSNR of the original and stabilized videos. The stabilized video's PSNR always lies significantly higher than the original video's PSNR. The original ITF is 8.97dB and the stabilized ITF is 15.48dB. This is an increase of 6.51dB.

(a) (b)

Figure 12: Selected frames from the (a) original BOOKSHELF video, and the (b) corresponding frames from the stable BOOKSHELF video.

5.1.7. BOOKSHELF2 Video

This video is similar to the first BOOKSHELF video, except that it contains AHRS data. It has a resolution of 640x480 and a framerate of 29.97 frames per second. Its total duration is about 3 seconds with a total of 100 frames. This video is taken indoors using a ground robot from DU^2SL. The robot was moved with linear motion and angular motion to impose translation and rotation, and the center of rotation is biased just like the previous BOOKSHELF video.

This video is included to study the effect of using the roll estimated by our AHRS as a data point in the fuzzy clustering of rotation step of our algorithm. No inertial data is used for translation in this example. Various trust values proportional to the maximum trust value of all matched SIFT features are assigned as the trust value of the AHRS roll data. We test using between 25% and 500% of the maximum trust value of all matched SIFT features.

Figure 13 shows the estimation of rotation by SIFT features and IMU/AHRS roll value. The motion of the robot is smooth, which the rotation estimated by SIFT demonstrates. The IMU/AHRS data roughly follows the SIFT estimation, but is affected by noise more than SIFT.

To compare the results of using different parameters, we created a mask of valid pixel locations for each test sample and found the intersection of all of these locations. This left us with a mask which represents valid pixel locations in every sample, so they can be compared quantitatively.

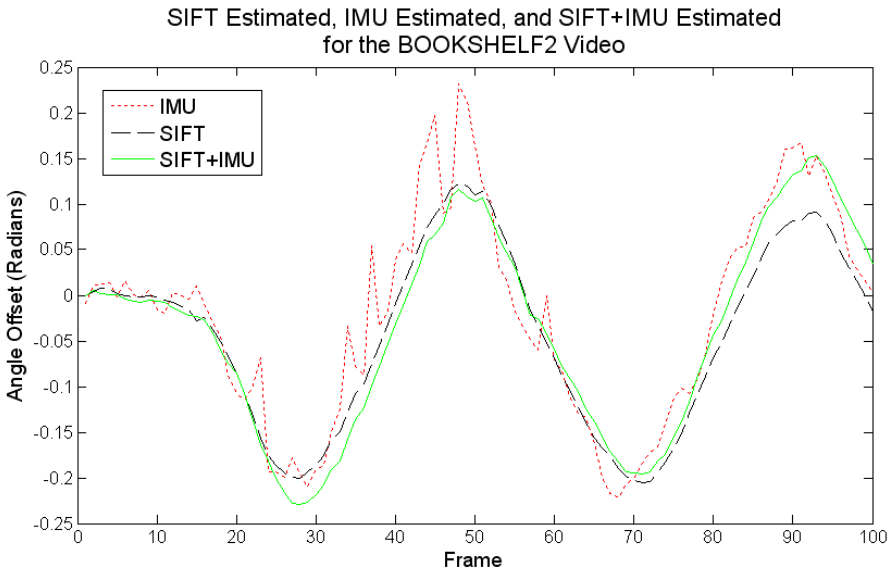


Figure 13: SIFT and IMU/AHRS estimated roll for the BOOKSHELF2 Video.

It appears that SIFT used alone produces better video stabilization results without the use of AHRS roll values. Figure 14 presents the ITF values of various scalar multipliers for the maximum trust value assigned to the AHRS data. The ITF value of the original video is 10.13dB. Using only SIFT, the ITF value is increased to 17.69dB. The ITF value when using just AHRS roll is 14.00dB, which is noticeably lower than using only SIFT. The ITF values when using IMU/AHRS roll along with SIFT features are an improvement over the original and using only the AHRS, but less than the SIFT-only stabilized video. The ITF values for the various multipliers lie between 15.79dB and 17.34dB with no discernable trend dependent on the multiplier used. This suggests that the roll value given by our AHRS introduces extra error into the stabilization process on average.

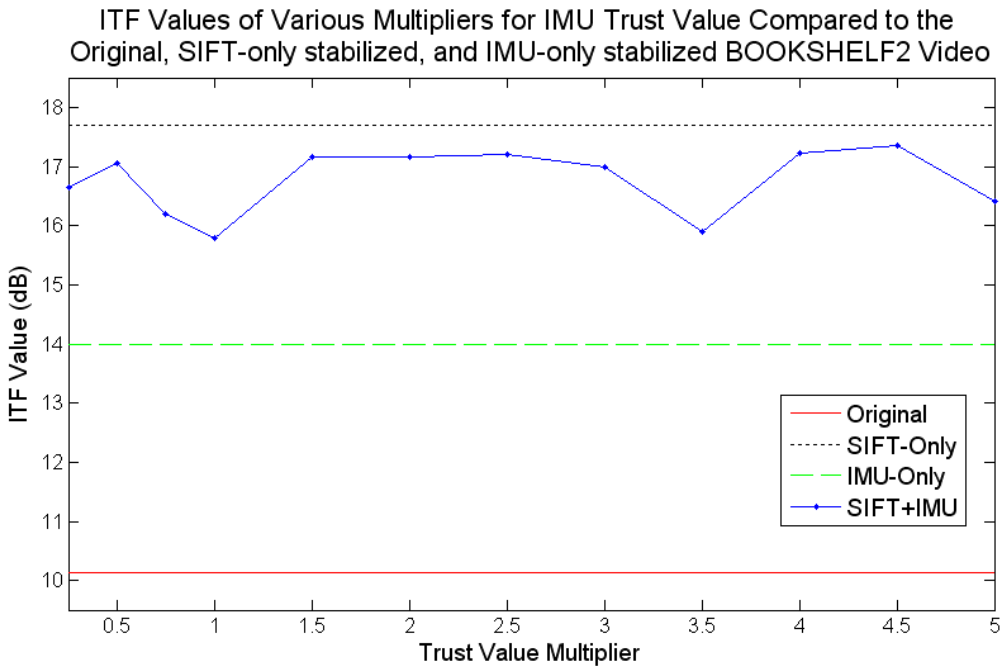


Figure 14: ITF values for various scalar multipliers for the maximum trust value applied to AHRS roll data for the BOOKSHELF2 Video compared to the original, SIFT-only stabilization, and IMU-only stabilization.

5.1.8. Experimental Comparison

In this section we compare the results using our video stabilization algorithm to the results obtained by Yang et al. in their particle filtering motion estimation (PFME) video stabilization approach. We did not implement their video stabilization algorithm, but instead used the stabilized video they supply. We compare the results from the three videos they supply: ONDESK, STREET, and ONROAD.

In all of these cases, it is assumed that there is no desired motion and that they stabilize their videos with respect to the first frame of the original video sequence, though the stabilized videos they supply are not of full length. The ONDESK and ONROAD videos have the first frame removed and the STREET video has the first twenty frames

removed. We have adjusted our resulting videos to match theirs for comparison. We do not have their motion estimation results for each frame, so we cannot use the same mask procedure to determine which pixels are valid to compare using PSNR. Instead, we elect to use the inner 36% of pixels in the center of the frame. We remove 20% of the width and height from each border, assuming that this central subwindow has only valid pixels.

The first video we compare is the ONDESK video. Both PFME and our SIFT-Fuzzy stabilization results are similar. Qualitatively, the results look similar, though there are some instances where SIFT-Fuzzy performs better than PFME and others where PFME performs better than SIFT-Fuzzy. Overall, both videos are significantly more stable than the original, and the results are comparable. The ITF values of the original video, PFME stabilized video, and SIFT-Fuzzy stabilized video are 15.44dB, 19.82dB and 22.29dB respectively. This follows the qualitative results that SIFT-Fuzzy slightly outperforms PFME. Figure 15 shows a graph of the PSNR values of the original, PFME stabilized and SIFT-Fuzzy stabilized video sequences over selected frames. There are a few occurrences where PFME lies above SIFT-Fuzzy, but for most of the video it lies below.

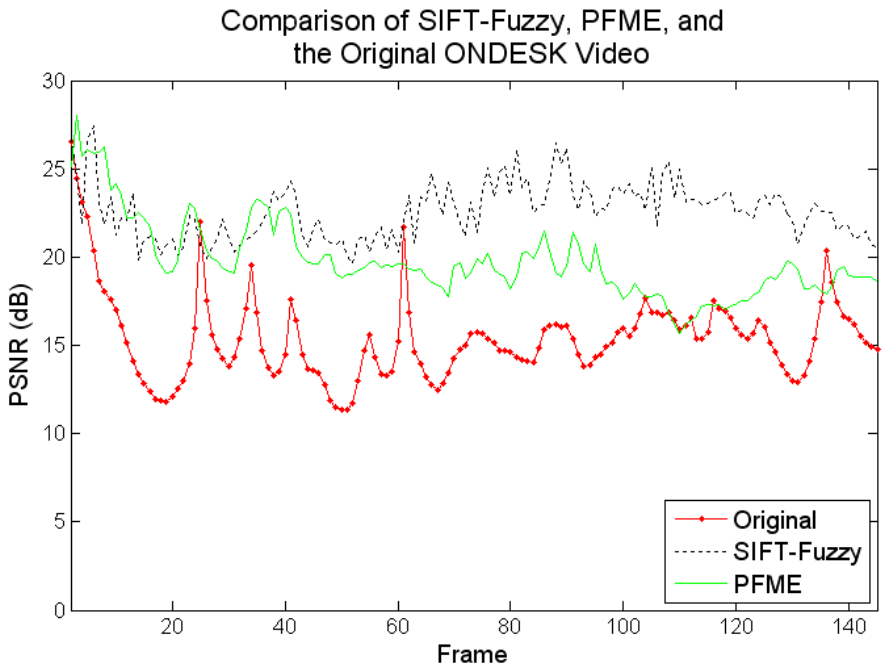


Figure 15: PSNR values of the original, PFME stabilized, and SIFT-Fuzzy stabilized videos over selected frames of the ONDESK video.

The STREET video results differ from the ONDESK video results. In this case, qualitatively, the PFME stabilized video is slightly superior to the SIFT-Fuzzy stabilized video. The SIFT-Fuzzy video suffers from slight jitter in rotation a few times throughout the video while the PFME video handles the rotation better. Both methods handle the local motion of the car driving through the frame well. Quantitatively, the results do not match the qualitative results. The ITF values of the original video, PFME stabilized video, and SIFT-Fuzzy stabilized video are 12.20dB, 14.55dB, and 19.62dB respectively. These results demonstrate that PSNR is not a good measure of video stabilization quality. Figure 16 shows the PSNR values of the original, PFME stabilized, and SIFT-Fuzzy stabilized STREET videos. Not once does the SIFT-Fuzzy PSNR drop below the PFME PSNR, even though PFME is better visually in this case.

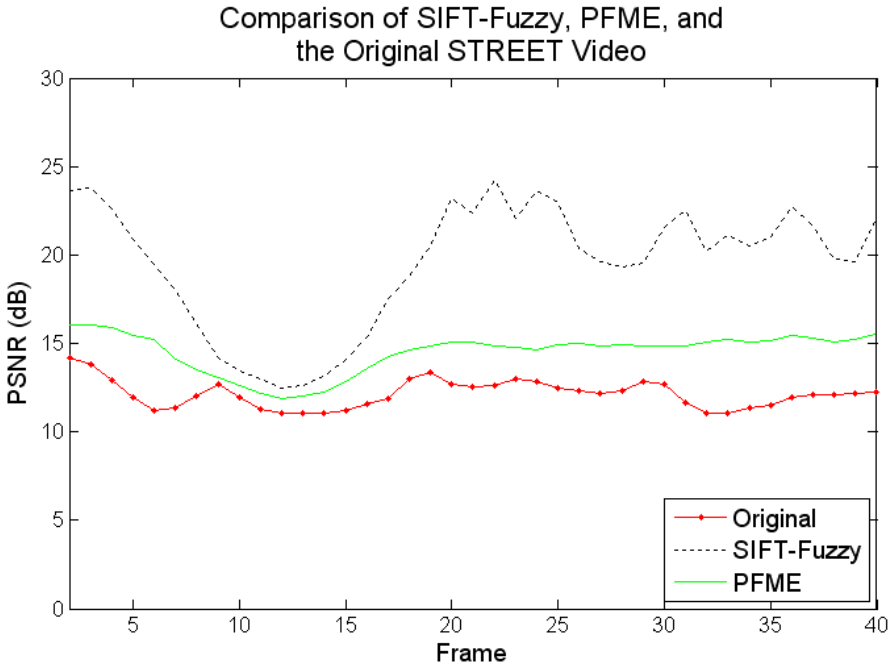


Figure 16: PSNR values of the original, PFME stabilized, and SIFT-Fuzzy stabilized videos over selected frames of the STREET video.

The results from the ONROAD video are more favorable toward SIFT-Fuzzy stabilization. In the SIFT-Fuzzy stabilization, the high frequency jitter in translation and rotation is almost completely removed, resulting in a very stable video. On the other hand, the PFME stabilization does not remove the jitter in translation nearly as well as SIFT-Fuzzy. Both videos handle rotation, but the PFME video has many instances of significant jitter in translation. This could result from one of two possibilities: the zooming aspect of the video due to forward motion caused problems for PFME, or the PFME video assumed that some of the translational motion was desired. The first case would show that PFME is sensitive to the outward radiating bias in motion that is caused by zooming. The second case would demonstrate a poor choice of assuming that

translation is desired. Forward motion is perpendicular to the image plane, so it should not be considered desired motion.

5.2. Summary of Results

We have presented several video sequences which tested the merits of our stabilization method and algorithm. We showed our video stabilization results on simple videos with no local motion and no desired motion as well as more complex videos which included local motion or desired motion. The resulting stabilized videos are improved both qualitatively and quantitatively (where applicable) over the original videos. Table 1 presents the ITF values from those video sequences where ITF is applicable.

Table 1: Summary of ITF values for all videos where ITF is applicable.

Video Sequence	Original Video ITF (dB)	Stabilized Video ITF (dB)	Increase (dB)
LAB	11.24	19.42	8.18
ONDESK	16.73	23.53	6.80
STREET	13.89	21.21	7.32
BOOKSHELF	8.97	15.48	6.51
BOOKSHELF2	10.13	IMU: 14.00 SIFT: 17.69 Mix: 15.79-17.34	3.87 7.56 5.63-7.21

We have discussed results with respect to design decisions, such as using fuzzy clustering for separation of local and global motion and Kalman filtering for estimating desired motion, as well as problems which occur in real videos, such as perspective changes due to objects in close proximity to the camera and a biased center of rotation. These results demonstrate that our video stabilization method performs well in many scenarios. SIFT feature orientation provides a very accurate method for calculating rotation between consecutive frames of video. The use of fuzzy clustering successfully

separates local and global motion. Kalman filtering for estimating desired motion allows for the use of a non-static camera. We also found that SIFT features provide a more accurate measure of rotation than the AHRS we used. Finally, we compared our results to those of Yang et. al in their PFME video stabilization method. In the case of the ONDESK video, the results were comparable. In the STREET video, their results were better than ours, although the PSNR and ITF values do not agree with this assessment. In the ONROAD video, our stabilization method outperformed their method.

CHAPTER 6. CONCLUSIONS AND FUTURE WORK

This chapter contains concluding remarks about the contents of this thesis as well as listing possible future work that would benefit the continuation of research on this project.

6.1. Conclusions

In this thesis, we have presented the video stabilization method and algorithm we have developed for general video stabilization scenarios. We discussed several important issues that affect all video stabilization methods in general with specific impact on our method. We showed through experimental videos that our method and algorithm performs successfully on a variety of videos.

Our video stabilization method utilized SIFT features for correspondence calculation, fuzzy clustering for the separation of global and local motion, and Kalman filtering to estimate desired motion, allowing for intentional motions. We include optional steps to perform when information about the Euler angles of the camera's orientation. We have designed our method to be a general video stabilization method, not tied to a specific application or scenario.

We discussed several issues with video stabilization methods including separating the estimation of translation and rotation into different steps and the impact of having a center of rotation which does not coincide with the center of the image plane as is normally assumed.

We presented results from several video sequences which tested different aspects of our video stabilization method and algorithm, and a few sequences which tested every aspect. Some of the video sequences were coupled with Euler angle data, and the inclusion of this information in the algorithm was tested and compared to the results without this information.

6.2. Future Work

The final goal of this research project is to stabilize the video from the Aquapod robot [10] from the University of Minnesota during its underwater navigation. As such, this algorithm should be implemented on embedded hardware such as a field-programmable gate array (FPGA) or other microcontroller or microprocessor with a small enough profile to be used on a miniature robot. Performing SIFT calculation and matching in real-time is the most challenging part, and could potentially be achieved by exploiting an FPGA's capabilities for massive parallelization.

Further research could be performed on utilization of SIFT-ME features instead of purely SIFT features. This could potentially lead to a less computationally complex algorithm since only a single fuzzy clustering iteration would need to be performed. It would be interesting to see if this would change the accuracy of stabilization negatively or positively.

Further research could be performed on using this video stabilization method as part of a control loop for a pan/tilt mechanism to physically stabilize the camera. This could be a slower outer loop controller working in concert with a faster inner loop controller using a fast IMU or AHRS sensor.

REFERENCES

[1] AMAZING base jump. August 15, 2007. [Online]. Available: Youtube, http://www.youtube.com/watch?v=5jWYIynY16k [Accessed: March 1, 2010].

[2] S. Battiato, G. Gallo, G. Puglisi, and S. Scellato. Fuzzy-based motion estimation for video stabilization using SIFT interest points. In Proceedings SPIE, volume 7250, 2009.

[3] S. Battiato, G. Gallo, G. Puglisi, and S. Scellato. SIFT features tracking for video stabilization. In 14th International Conference on Image Analysis and Processing, 2007. ICIAP 2007, pages 825-830, September 2007.

[4] A. Batur and B. Flinchbaugh. Video stabilization with optimized motion estimation resolution. In 2006 IEEE International Conference on Image Processing, pages 465-468, October 2006.

[5] J. C. Bezdek. Pattern Recognition with Fuzzy Objective Function Algorithms. Kluwer Academic Publishers, Norwell, MA, USA, 1981.

[6] F. L. Bookstein. Morphological tools for landmark data: geometry and biology. Cambridge University Press, 1997.

[7] A. Broggi, P. Grisleri, T. Graf, and M. Meinecke. A software video stabilization system for automotive oriented applications. In 2005 IEEE 61st Vehicular Technology Conference, 2005. VTC 2005-Spring, 5:2760- 2764, May-June 2005.

[8] C. Buehler, M. Bosse, and L. McMillan. Non-metric image-based rendering for video stabilization. In Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2001. CVPR 2001, 2:609-614, 2001.

[9] J. Cai and R. Walker. Robust video stabilisation algorithm using feature point selection and delta optical flow. Computer Vision, IET, 3(4):176-188, December 2009.

[10] A. Carlson and N. Papanikolopoulos. Aquapod: Prototype Design of an Amphibious Tumbling Robot. In 2011 IEEE International Conference on Robotics and Automation. ICRA 2011, May 2011.

[11] H. Chang, S. Lai, and K. Lu. A robust and efficient video stabilization algorithm. In 2004 IEEE International Conference on Multimedia and Expo, 2004. ICME '04, 1:29-32, June 2004.

[12] G. Corsini, M. Diani, and A. Masini. Video Sequence Stabilization for Real-Time Remote Sensing Applications. In IEEE International Conference on Geoscience

and Remote Sensing Symposium, 2006. IGARSS 2006, pages 988-991, July-August 2006.

[13] J. C. Dunn. A fuzzy relative of the isodata process and its use in detecting compact well-separated clusters. In Journal of Cybernetics, 3:32-57, 1973.

[14] M. Han and T. Kanade. Multiple motion scene reconstruction with uncalibrated cameras. In IEEE Transactions on Pattern Analysis and Machine Intelligence, 25(7):884-894, July 2003.

[15] B. K. Horn and B. G. Schunck. Determining optical flow. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1980.

[16] J. Hsiao, C. Hsu, T. Shih, P. Hsu, S. Yeh, and B. Wang. The real-time video stabilization for the rescue robot. In ICCAS-SICE, 2009, pages 4364-4369, August. 2009.

[17] M. Hwangbo, J. Kim, and T. Kanade. Inertial-aided KLT feature tracking for a moving camera. In IEEE/RSJ International Conference on Intelligent Robots and Systems, 2009. IROS 2009, pages 1909-1916, October 2009.

[18] G. J. Klir, U. St. Clair, and B. Yuan. Fuzzy set theory: foundations and applications. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[19] C. D. Kuglin and D. C. Hines. The phase correlation image alignment method. In Proceedings of the International Conference on Systems, 1975.

[20] C. Kurz, T. Thormahlen, and H. Seidel. Scene-Aware Video Stabilization by Visual Fixation. In Conference for Visual Media Production, 2009. CVMP '09, pages 1-6, November 2009.

[21] K. Liu, J. Qian, and R. Yang. Block matching algorithm based on RANSAC algorithm. In 2010 International Conference on Image Analysis and Signal Processing (IASP), pages 223-227, April 2010.

[22] D. G. Lowe. Distinctive image features from scale-invariant keypoints. In International Journal of Computer Vision, 60:91-110, November 2004.

[23] B. D. Lucas and T. Kanade. An iterative image registration technique with an application to stereo vision. In Proceedings of the 7th International Joint Conference on Artificial intelligence, 2:674-679, San Francisco, CA, USA, 1981. Morgan Kaufmann Publishers Inc.

[24] Q. Luo and T. Khoshgoftaar. An empirical study on estimating motions in video stabilization. In 2007 IEEE International Conference on Information Reuse and Integration. IRI 2007, pages 360-366, August 2007.

[25] L. Marcenaro, G. Vernazza, and C. Regazzoni. Image stabilization algorithms for video-surveillance applications. In Proceedings of the International Conference on Image Processing, 2001, volume 1, pages 349-352, 2001.

[26] Y. Matsushita, E. Ofek, W. Ge, X. Tang, and H.-Y. Shum. Full-frame video stabilization with motion inpainting. In IEEE Transactions on Pattern Analysis and Machine Intelligence, 28(7):1150-1163, July 2006.

[27] M. Niskanen, O. Silven, and M. Tico. Video stabilization performance assessment. In 2006 IEEE International Conference on Multimedia and Expo, pages 405-408, July 2006.

[28] M. Ondrej, Z. Frantisek, and D. Martin. Software video stabilization in a fixed point arithmetic. In First International Conference on the Applications of Digital Information and Web Technologies, 2008. ICADIWT 2008, pages 389-393, August 2008.

[29] J. Paik, Y. Park, and D. Kim. An adaptive motion decision system for digital image stabilizer based on edge pattern matching. In IEEE Transactions on Consumer Electronics, 38(3):607-616, August 1992.

[30] M. Ramachandran, R. Chellappa. Stabilization and Mosaicing of Airborne Videos. In 2006 IEEE International Conference on Image Processing, pages 345-348, October 2006.

[31] H. Shen, Q. Pan, Y. Cheng, and Y. Yu. Fast video stabilization algorithm for UAV. In 2009 IEEE International Conference on Intelligent Computing and Intelligent Systems, ICIS 2009, volume 4, pages 542-546, November 2009.

[32] Y. Shen, P. Guturu, T. Damarla, B. Buckles, and K. Namuduri. Video stabilization using principal component analysis and scale invariant feature transform in particle filter framework. In IEEE Transactions on Consumer Electronics, 55(3):1714-1721, August 2009.

[33] J. Shi and C. Tomasi. Good features to track. Technical report, Cornell University, Ithaca, NY, USA, 1993.

[34] P. Shi, Y. Zhu, and S. Tong. Video stabilization in visual prosthetics. In IEEE/ICME International Conference on Complex Medical Engineering, 2007. CME 2007, pages 782-785, May 2007.

[35] S. Srinivasan and R. Chellappa. Noise-resilient estimation of optical fbw by use of overlapped basis functions. In Journal of the Optical Society of America, 16(3), March 1999.

[36] A. Vedaldi and B. Fulkerson. VLFeat: An open and portable library of computer vision algorithms. http://www.vlfeat.org/, 2008.

[37] R. Wilcox. Fundamentals of Modern Statistical Methods. Berlin: Springer, 2010.

[38] C. Wu. Sift-GPU: A GPU Implementation of Scale Invariant Feature Transform (SIFT). December 10, 2010 [May 1, 2011]. University of North Carolina at Chapel Hill. http://cs.unc.edu/~ccwu/siftgpu/

[39] G. Wu, M. H. Mahoor, S. Althloothi, and R. M. Voyles. SIFT-motion estimation (SIFT-ME): A new feature for human activity recognition. In IPCV, pages 804-811, 2010.

[40] J. Yang, D. Schonfeld, C. Chen, and M. Mohamed. Online video stabilization based on particle filters. In 2006 IEEE International Conference on Image Processing, pages 1545-1548, October 2006

[41] Y. Zhang, M. Xie, and D. Tang. A Central Sub-image Based Global Motion Estimation Method for In-Car Video Stabilization. In Third International Conference on Knowledge Discovery and Data Mining, 2010, WKDD '10, pages 204-207, January 2010.

APPENDIX A. MATLAB SOURCE CODE

Code Usage

The input parameters must be prepared as follows. The mov variable must be a cell array with a height of 1 and a width of the duration of the video. If no mov variable is specified, the program will look for a file called "ONROAD_original.avi" and create a cell movie. Each cell must be a grayscale image of single precision. The IMU is optional. If it is included, it must be a matrix where the height is the duration of the video and the width is 3, representing pitch, roll, and yaw respectively in degrees.

The vlfeat package from http://www.vlfeat.org/ is necessary for SIFT feature calculation. The path to the vlfeat MATLAB toolbox must be included in MATLAB's path variable through the Set Path menu option or command.

The output variables represent the stabilized cell movie, the measured offsets, the filtered offsets, and the Kalman filter estimated motion. The stabilized cell movie (outmov) is a 1xduration cell matrix where each cell is a grayscale image. The estimated offsets (offsets) lie in a 6xduration matrix where the rows represent the horizontal position, vertical position, angle (in radians), horizontal velocity, vertical velocity, and angular velocity (in radians/sec). The filtered offsets (filtered) lie in a 3xduration matrix where the rows represent horizontal position, vertical position, and angle (in radians). The Kalman filter estimated motion (kalmans) lies in a 6xduration matrix where the rows represent the horizontal position, vertical position, angle (in radians), horizontal velocity, vertical velocity, and angular velocity (in radians/sec).

```
function [ outmov,offsets,filtered,kalmans ] = Stabilize( mov,IMU )
%STABILIZE Stabilizes the cell video (mov)
%   Stabilizes the cell video and outputs the the stabilized video
%   the measured offsets, the filtered offsets, and the Kalman filter
state
%   vectors

%% SIFT Initialization
% Initialize vlfeat for SIFT functionality
vl_setup;
clc;

%% Variable Initialization
% Create cell storage for the output video
outmov=cell(1,size(mov,2));
outmov{1,1}=mov{1,1}; % first frame is the same in both videos

% Start accumulated motions at 0 [x;y;ori]
acc_motion=zeros(3,1);

% Initialize output variables to 0
offsets=zeros(6,size(mov,2));
filtered=zeros(3,size(mov,2));
kalmans=zeros(6,size(mov,2));

%% Kalman Filtering Initialization
% Define all necessary Kalman Filter matrices
% State vector: [x,y,theta,vx,vy,vtheta]'
prev_x=zeros(6,1);
curr_x=zeros(6,1);

% Probability matrices
curr_P=zeros(6,6);
prev_P=zeros(6,6);

% Motion model. If no desired motion is desired, set desiredhv=0. This
will
% use zeros as the model. The covariance should also be changed below.
Set
% desiredhv=1 to allow for the desired horizontal and vertical motion
model
% as shown below. The Kalman filter state vector motion must be rotated
to
% fit the same reference as the data.
% F=[cos(best_ori) sin(best_ori) 0 1 0 0;
%     -sin(best_ori) cos(best_ori) 0 0 1 0;
%     0 0 0 0 0 0;
%     0 0 0 1 0 0;
%     0 0 0 0 1 0;
%     0 0 0 0 0 1];
```

```
F=zeros(6,6);
desiredhv=0;

% Q for w matrix, i.e. Trust in Motion Model. Leave this alone
Q=0.1;
w=diag(ones(6,1)*Q*Q);

% Sensing Matrix H, vx, vy, and vtheta are sensed
H=[0 0 0 1 0 0;
    0 0 0 0 1 0;
    0 0 0 0 0 1];

% R for v matrix, i.e. Trust in Sensors. Use 0.1 when there is desired
% motion and 2.1 when there is no desired motion.
% R=0.1;
R=0.1;
v=diag(ones(3,1)*R*R);

% residuals - sensor and probability
y=zeros(3,1);
S=zeros(3,3);

% Kalman Gain
K=zeros(6,3);

%% Get First Frame for use as Previous Frame
% Get first frame SIFT and give each feature a trust value of 1
fprintf('Calculating SIFT of frame 1...');
[prev_feats,prev_descrs]=vl_sift(mov{1,1});
prev_feats=[prev_feats;ones(size(prev_feats,2))];
fprintf('done\n');

%% Loop from Second to Last Frames and Stabilize
for f=2:size(mov,2) % for each frame

    % Get current frame's SIFT and give each feature a trust value of 1
    fprintf('Calculating SIFT of frame %d...',f);
    [curr_feats,curr_descrs]=vl_sift(mov{1,f});
    curr_feats=[curr_feats;ones(size(curr_feats,2))];
    fprintf('done\n');

    % Match features (and get the indices for matches)
    fprintf('Matching SIFT features between frames...');

[matches,mindex]=NNMatch(prev_feats(1:4,:),prev_descrs,curr_feats(1:4,:
),curr_descrs);
    fprintf('done\n');

    % Calculate fuzzy clusters of rotations, all rotations should be
    % put in the range [-pi pi]
    fprintf('Clustering rotations into fuzzy sets...');
    ori_diff=(matches(8,:)-matches(4,:))';
```

```
    ori_diff(ori_diff>pi)=ori_diff(ori_diff>pi)-2*pi;
    ori_diff(ori_diff<-pi)=ori_diff(ori_diff<-pi)+2*pi;
    if nargin<2 % IMU not provided
        [membership]=FuzzyCluster(ori_diff);
    else % IMU provided
        IMUroll=(IMU(f,2)-IMU(f-1,2))*-pi/180;
        [membership]=FuzzyCluster(ori_diff,IMUroll);
    end
    fprintf('done\n');

    % Find best fuzzy cluster of rotations for background
representation
    fprintf('Choosing the best cluster...');

[best_cluster,cindex]=FindBestCluster(ori_diff,membership,prev_feats,mi
ndex(:,1),1/(2*size(membership,2)));
    fprintf('done\n');

    % Remove outliers and calculate rotation from best fuzzy cluster
    fprintf('Calculating orientation...');
    [best_oris,idx]=IQROutliers(best_cluster);
    best_ori=mean(best_oris);
    fprintf('done\n');

    % Find translation given the calculated rotation
    fprintf('Calculating translation...');

[resvecs]=CalculateResultantVectors(prev_feats(:,mindex(cindex(idx),1))
,curr_feats(:,mindex(cindex(idx),2)),best_ori,size(mov{1,f}));
    [dmembership]=FuzzyCluster(resvecs');

[BestdCluster,didx]=FindBestCluster(resvecs',dmembership,prev_feats,min
dex(cindex(idx),1),1/(size(dmembership,2)));
    translation=mean(BestdCluster)';
    fprintf('done\n');

    % Increment trust value for appropriate features

curr_feats(5,mindex(cindex(idx(didx)),2))=prev_feats(5,mindex(cindex(id
x(didx)),1))+1;

    % Update total accumulated motion, rotate the accumulated
translation
    % before accumulating more.
    acc_motion(1:2,1)=[cos(best_ori) sin(best_ori);-sin(best_ori)
cos(best_ori)]*acc_motion(1:2,1)+translation;
    acc_motion(3,1)=acc_motion(3,1)+best_ori;

    % Filter Motion using Kalman to allow for desired motion
    if desiredhv==1
       F=[cos(best_ori) sin(best_ori) 0 1 0 0;
        -sin(best_ori) cos(best_ori) 0 0 1 0;
        0 0 0 0 0 0;
```

```
            0 0 0 1 0 0;
            0 0 0 0 1 0;
            0 0 0 0 0 1];
        end

        % Predict Step
        curr_x=F*prev_x;
        curr_P=F*prev_P*F'+w;

        % Update Step
        y=[translation;best_ori]-H*curr_x; %velocities
        S=H*curr_P*H'+v;
        K=curr_P*H'*inv(S);
        curr_x=curr_x+K*y;
        curr_P=(eye(6)-K*H)*curr_P;

        % Desired motion is the filtered data subtracted from the measured
    data
        desired_motion=acc_motion-curr_x(1:3);

        % Assign current frame's information to output variables
        kalmans(:,f)=curr_x;
        filtered(:,f)=desired_motion;
        offsets(1:3,f)=acc_motion;
        offsets(4:6,f)=[translation;best_ori];

        % Compensate for the total motion with respect to the reference
    frame
        fprintf('Compensating for motion...');
        outmov{1,f}=CompensateMotion(mov{1,f},desired_motion);
        fprintf('done\n');

        % Set prev variables with current data for next iteration
        prev_feats=curr_feats;
        prev_descrs=curr_descrs;
        prev_x=curr_x;
        prev_P=curr_P;

    end

    end
```

NNMatch.m

```
function [ matches,index ] = NNMatch( feats1,descrs1,feats2,descrs2 )
%NNMATCH Performs brute-force Nearest-Neighbor matching of SIFT
features
%   Performs brute-force Nearest-Neighbor matching of SIFT features
using
%   the SIFT descriptors.

matches=[];
index=[];

for i=1:size(descrs1,2) % for each feature in img 1
    reped1=repmat(descrs1(:,i),1,size(descrs2,2)); %replicate for
matrix subs
    reped1=(reped1-descrs2).*(reped1-descrs2); % squared differences
    reped1=sum(reped1); % sum of squared differences
    mini=find(reped1==min(reped1), 1 ); % find NN
    min2i=find(reped1==min(reped1([1:mini-1 mini+1:size(reped1,2)])), 1
); % Find 2NN

    % Threshold for removing bad matches, add to output if it matches
    if reped1(mini)/reped1(min2i)<0.5
        index=[index;i mini];
        m=[feats1(1:4,i);feats2(1:4,mini)];
        matches=[matches m];
    end
end

end
```

FuzzyCluster.m

```
function [ membership ] = FuzzyCluster( data , IMUdata )
%FUZZYCLUSTER Clusters the data and returns a membership matrix
%   Clusters the data and returns the membership matrix. IMU data can
%   optionally be included and will be appended to the end of the data
%   array

%% Simple k value for k-means
k=floor(sqrt(size(data,1)/2));
if nargin==2

[classlist,centroids]=kmeans([data;IMUdata],k,'emptyaction','drop');
else
    [classlist,centroids]=kmeans(data,k,'emptyaction','drop');
end

%% Get rid of any NaN centroids
removals=[];
for i=1:size(centroids,1)
    if sum(isnan(centroids(i,:)))>0
        removals=[removals i];
    end
end

centroids(removals,:)=[];
k=size(centroids,1);
threshold=1/(2*k);

%% Assign membership values
if nargin==2
    membership=zeros(size(data,1)+1,k);
else
    membership=zeros(size(data,1),k);
end
for i=1:size(data,1)
    for j=1:k
        d=data(i,:)-centroids(j,:);
        if d==0
            membership(i,j)=1;
        else
            membership(i,j)=1/sqrt(sum(d.*d)); % L2 norm
        end
    end
    % Normalize row
    d=sum(membership(i,:));
    membership(i,:)=membership(i,:)/d;
end

%% Add membership for IMU if included in the function call
if nargin==2
    for i=1:k
        d=IMUdata-centroids(j,:);
```

```
        if d==0
            membership(size(data,1)+1,i)=1;
        else
            membership(size(data,1)+1,i)=1/sqrt(sum(d.*d));
        end
    end

    d=sum(membership(size(data,1)+1,:));
    membership(size(data,1)+1,:)=membership(size(data,1)+1,:)/d;
end

end
```

FindBestCluster.m

```
function [ best_cluster,cindex ] = FindBestCluster(
diffs,membership,prev_feats,index,min_membership )
%FINDBESTCLUSTER Finds the cluster with the highest weighted trust
% Finds the cluster with the highest weighted trust value summation.
Trust
% values are weighted by their membership to the cluster and then
summed.
% The highest sum is chosen, and features with at least min_membership
to
% the best cluster are put into a matrix.

% Replicate trust values to avoid for loop
weights=repmat(prev_feats(5,index)',1,size(membership,2)); % replicate
trust values

% if IMU roll data is present, replicate highest trust for IMU info
if size(membership,1)>size(diffs,1)
    www=5.0;

weights=[weights;repmat(max(prev_feats(5,index))*www,1,size(membership,
2))];
end
weights=weights.*membership; % weight based on membership
weights=sum(weights); % add up each column

% if IMU roll data is present, remove it from the membership matrix for
indexing
% purposes
if size(membership,1)>size(diffs,1)
    membership(size(membership,1),:)=[];
end

cindex=find(membership(:,weights==max(weights))>=min_membership); %
min_membership or greater
best_cluster=diffs(cindex,:);

end
```

IQROutliers.m

```
function [ data,idx ] = IQROutliers( data )
%CWOUTLIERS Removes outliers from the data using interquartile range
%   Removes outliers from the data using the interquartile range
method.
%   The range between the 1st and 3rd quartiles is multiplied by 1.5
and
%   subtracted from (added to) the 1st (3rd) quartile. Anything beyond
this
%   extended range is an outlier.

%%Find IQR
% Find first and third quartile
quants=quantile(data,[0.25 0.75]);

% Find the innerquartile range
iqr=quants(2)-quants(1);

%% Find the locations of outliers
removals=[];
l=quants(1)-1.5*iqr;
r=quants(2)+1.5*iqr;
for i=1:size(data,1)
    if data(i)<l
        removals=[removals i];
    elseif data(i)>r
        removals=[removals i];
    end
end

%% Create an index and remove outliers
idx=1:size(data,1);
idx(removals)=[];
data(removals)=[];

end
```

CalculateResultantVectors.m

```
function [ resvecs ] = CalculateResultantVectors(
prev_feats,curr_feats,rotation,imgsize )
%CALCULATERESULTANTVECTORS Calculates the resultant vectors of the
local
%motion of matched features.
%   Calculates the resultant vectors of the local motion of matched
%   features using a global rotation estimate. The difference between
the
%   current location of a feature and the location of the previous
feature
%   after rotated by the global rotation estimate is the resultant
vector.

%% Initialization
% Center of rotation is the center of the frame
center=[(imgsize(2)+1)/2;(imgsize(1)+1)/2];

% Save the rotation matrix so it isn't calculated every iteration
rmatrix=[cos(rotation) sin(rotation);-sin(rotation) cos(rotation)];

resvecs=zeros(2,size(prev_feats,2));

%% Calculate the Vectors
for t=1:size(prev_feats,2)

    % Rotate previous features assuming center of frame is the center
of
    % rotation
    rotloc=rmatrix*[prev_feats(1,t)-center(1);center(2)-
prev_feats(2,t)];

    % Convert current feature pixel location to coordinates with
respect to
    % the center of the frame
    newloc=[curr_feats(1,t)-center(1);center(2)-curr_feats(2,t)];

    % Resultant vectors
    resvecs(:,t)=newloc-rotloc;
end

end
```

CompensateMotion.m

```
function [ out_frame ] = CompensateMotion( in_frame,motion )
%COMPENSATEMOTION Corrects the translation and rotation from motion
vector
%   Corrects the translation and rotation from the motion vector. First
%   translates the image and fills in black pixels, then rotates the
image
%   using bilinear interpolation.

%% Initialize
out_frame=in_frame;

%% Translate Image
% Round the horizontal and vertical translations to whole numbers
rmotion=round(motion(1:2));

% horizontal
if rmotion(1)<0 % has shifted left
    % Move it back to the right by padding left with zeros and removing
the
    % same amount of columns on the right side
    padding=zeros(size(in_frame,1),-rmotion(1));
    out_frame=[padding out_frame(:,1:(end+rmotion(1)))];
elseif rmotion(1)>0 % has shifted right
    % Move it back to the left by padding right with zeros and removing
the
    % same amount of columns on the left side
    padding=zeros(size(in_frame,1),rmotion(1));
    out_frame=[out_frame(:,(rmotion(1)+1):end) padding];
end

% vertical
if rmotion(2)<0 % has shifted down
    % Move it back up by padding bottom with zeros and removing the
same
    % amount of rows from the top of the image
    padding=zeros(-rmotion(2),size(in_frame,2));
    out_frame=[out_frame((1-rmotion(2)):end,:);padding];
elseif rmotion(2)>0 % has shifted up
    % Move it back down by padding top with zeros and removing the same
    % amount of rows from the bottom
    padding=zeros(rmotion(2),size(in_frame,2));
    out_frame=[padding;out_frame(1:(end-rmotion(2)),:)];
end

%% Rotate image, crops image to the appropriate size
out_frame=imrotate(out_frame,rad2deg(motion(3)),'bilinear','crop');

end
```

APPENDIX B. C++ SOURCE CODE

Code Usage

This code was developed on a PC using the Ubuntu 10.1 operating systems, and is meant to be run on a Linux machine. There is no guarantee that this will work out of the box on a Windows machine, though the underlying libraries and function calls are supported on Windows platforms. It uses SiftGPU [38] to calculate and match SIFT features in real-time. We compiled SiftGPU using CUDA from nVidia, though it would be possible to use the OpenGL version by removing "-cuda" from the argument list in the AugmentedSIFTList.cpp file. OpenCV2.1 from http://opencv.willowgarage.com/wiki/ is used for capturing video from a camera, k-means clustering, matrix multiplication, and for displaying the compensated video. The source files as well as the makefile should be put into a src folder and the header files should be put into an include folder. The makefile will create an executable file in the bin folder, and will create this folder if it does not exist. The executable takes no arguments. The executable will use OpenCV to automatically open the first available video capture device and perform video stabilization on the video stream. We used a Logitech C160 webcam to test our program. Functions are included that can open a video file with or without externally calculated SIFT feature information.

Notes and Findings

This implementation works in real-time as defined by most of the computer vision community on images of VGA resolution or lower. When displaying the live video, a 640x480 video can be stabilized at 17fps, and when the video is not displayed (i.e. for

video files where monitoring the video is not required), it can be stabilized at 18.4fps. A video with 160x120 resolution, which is the case of half of the videos contained in this thesis, can be stabilized at a framerate of 61fps if video is displayed and 70fps if the video is not displayed. The slowdown due to displaying videos is partially due to sending output to the screen, but there is a fixed 1-2ms delay that must be used for OpenCV to properly display an image on the screen.

The results using SiftGPU were not as good as the results using vlfeat. This could be caused by several differences between the implementations. SiftGPU uses floating point values and vlfeat uses double precision values. The two algorithms might perform steps differently, such as calculating the difference in scale differently when building the Gaussian pyramid for SIFT feature calculation. The largest problem that occurred using SiftGPU is that the orientation of SIFT features was not as accurately estimated, and in many cases a bias in rotation will build over time through a video sequence. This is less pronounced while using a capturing device as long as the motion is not too fast (the webcam used is not a high performance camera). We extracted SIFT features using vlfeat in MATLAB and used them in the C++ implementation to test the validity of the claim that vlfeat was more accurate, and produced similar results to the MATLAB implementation, leading us to the conclusion that the problem lies in the accuracy of the feature calculation of SiftGPU. The video stabilization method proposed in this thesis requires a very accurate representation of local feature motion.

Makefile

```
CC=g++
INCS= -I../include -I./SiftGPU/src/SiftGPU -Wall
LIBS= -lcv -lcvaux -lcxcore -lhighgui -L./SiftGPU/lib -lsiftgpu

#Features=FeatureList.o SIFTFeatureList.o AugmentedSIFTFeatureList.o

all : Runner

Runner : VideoKalman.o Image.o Clustering.o AugmentedSIFTList.o
SIFTFuzzyKalman.o SIFTFuzzyStructs.o Runner.cpp
	$(CC) Runner.cpp $(Features) VideoKalman.o Image.o Clustering.o
AugmentedSIFTList.o SIFTFuzzyKalman.o SIFTFuzzyStructs.o $(INCS)
$(LIBS) -o ../bin/SIFTFuzzyKalmanStabilize

VideoKalman.o : VideoKalman.cpp
	$(CC) VideoKalman.cpp $(INCS) $(LIBS) -c

Image.o : Image.cpp
	$(CC) Image.cpp $(INCS) $(LIBS) -c

Clustering.o : Clustering.cpp
	$(CC) Clustering.cpp $(INCS) $(LIBS) -c

AugmentedSIFTList.o : AugmentedSIFTList.cpp
	$(CC) AugmentedSIFTList.cpp $(INCS) $(LIBS) -c

SIFTFuzzyKalman.o : SIFTFuzzyKalman.cpp
	$(CC) SIFTFuzzyKalman.cpp $(INCS) $(LIBS) -c

SIFTFuzzyStructs.o : SIFTFuzzyStructs.cpp
	$(CC) SIFTFuzzyStructs.cpp $(INCS) $(LIBS) -c

clean :
	@rm -rf *.o
	@rm -rf ../bin/*
```

Runner.cpp

```
#include "SIFTFuzzyKalman.hpp"

int main(){
      SIFTFuzzy::StabilizeDevice(-1,640,480);
//    SIFTFuzzy::StabilizeFile("ONDESK_original.avi");
//    SIFTFuzzy::StabilizeFileWithMatches("LAB_test.avi","dummy");
      return 0;
}
```

SIFTFuzzyKalman.hpp

```
#ifndef SIFTFUZZYKALMAN_HPP
#define SIFTFUZZYKALMAN_HPP

#include <vector>
#include <string>

#include "Image.hpp"
#include "AugmentedSIFTList.hpp"
#include "VideoKalman.hpp"
//#include "SIFTFuzzyStructs.hpp"

namespace SIFTFuzzy{

// Stabilize a file
void StabilizeFile(const std::string filename);

// Stabilize a file using SIFT feature information (directory is a
dummy variable for now)
void StabilizeFileWithMatches(const std::string filename,const
std::string directory);

// Stabilize a camera connected to the computer
void StabilizeDevice(const int deviceNum=-1,const int width=640,const
int height=480);

// Dump information into files (used on failures mostly)
void DumpEverything(AugmentedSIFTList &prevS,AugmentedSIFTList &currS,
            Image &prevI,Image &currI,VideoKalman
&filter,std::vector<SIFTMatches> *matches,
            int numMatches,double tx,double ty,double rot);

}

#endif
```

SIFTFuzzyKalman.cpp

```
#include "SIFTFuzzyKalman.hpp"
#include "Clustering.hpp"
#include "SiftGPU.h"

#include <iostream>
#include <fstream>
#include <ctime>
#include <sys/time.h>

namespace SIFTFuzzy{

void StabilizeDevice(const int deviceNum,const int width,const int
height)
{
	// Declare all variables and Initialize

	// Kalman Filter with all motion desired
//	SIFTFuzzy::VideoKalman
kfilter(SIFTFuzzy::VideoKalman::DESIRED_TRANSLATION,0.1,0.1);
	SIFTFuzzy::VideoKalman
kfilter(SIFTFuzzy::VideoKalman::DESIRED_NONE,0.1,2.1);

	// Variables for storing motion
	float x_vel,y_vel,theta_vel; // estimated for each frame
	float x_accum,y_accum,theta_accum,temp_accum;
	float x_desired,y_desired,theta_desired; // desired i.e. Kalman

	// Image Variables
	SIFTFuzzy::Image prevImg,currImg,compensatedImg,capturedImg;

	// SIFT variables
	SiftGPU *sift=new SiftGPU;
	SiftMatchGPU *matcher=new SiftMatchGPU;
	SIFTFuzzy::AugmentedSIFTList prevFeats,currFeats;
	std::vector<SIFTFuzzy::Touple> mIndex;
	std::vector<SIFTFuzzy::SIFTMatches> matches;
	int numMatches;

	// Fuzzy Variables
	std::vector<std::vector<float> > membership(200);
	for(int i=0;i<(int) (membership.size());++i){
		membership[i]=std::vector<float>(20);
	}
	int numClusters;

	// Initialize capture device and get image size
	cv::VideoCapture cap(-1); // choose first available video device
	if(!cap.isOpened()){
		std::cout << "Error opening capture device"<<std::endl;
		return;
	}
```

```
	// I don't check to make sure these setters work, I know the
camera
	// that I'm using is 640x480 anyway.
	cap.set(CV_CAP_PROP_FRAME_WIDTH,width);
	cap.set(CV_CAP_PROP_FRAME_HEIGHT,height);

	// Initialize SIFT Calculator and Matcher
	SIFTFuzzy::AugmentedSIFTList::InitializeSiftGPU(sift,matcher);

	// Initialize Images
	compensatedImg=SIFTFuzzy::Image(width,height,SIFTFuzzy::Image::IM
G_FORMAT_RGB,"Compensated Video");
	capturedImg=SIFTFuzzy::Image(width,height,SIFTFuzzy::Image::IMG_F
ORMAT_RGB,"Captured Video");
	prevImg=SIFTFuzzy::Image(width,height,SIFTFuzzy::Image::IMG_FORMA
T_GRAY);
	currImg=SIFTFuzzy::Image(width,height,SIFTFuzzy::Image::IMG_FORMA
T_GRAY);

	// Eat several frames so camera can adjust
	for(int i=0;i<100;++i){
		cap>>(capturedImg.data);
		cv::waitKey(30);
	}

	// Get and Show initial frame
	cv::namedWindow(compensatedImg.imgName,CV_WINDOW_AUTOSIZE);
//	cv::namedWindow(capturedImg.imgName,CV_WINDOW_AUTOSIZE);
	cap>>(capturedImg.data);
	cv::cvtColor(capturedImg.data,prevImg.data,CV_RGB2GRAY);
	prevFeats.CalculateSIFTFeatures(sift,prevImg);
	compensatedImg.data=capturedImg.data.clone();
	cv::imshow(compensatedImg.imgName,compensatedImg.data);
//	cv::imshow(capturedImg.imgName,capturedImg.data);
	cv::waitKey(2);

	while(1){ // infinite loop
		// Perform SIFT+Matching+Augmentation
		// Capture image
		cap>>(capturedImg.data);
		cv::cvtColor(capturedImg.data,currImg.data,CV_RGB2GRAY);

		// Calculate and Match SIFT
		currFeats.CalculateSIFTFeatures(sift,currImg);

	numMatches=SIFTFuzzy::AugmentedSIFTList::MatchSIFTFeatures(matche
r,

prevFeats,currFeats,matches,mIndex,0);

		if(numMatches==0){
			std::cout<<"No matches, saving dump\n";
```

```
      DumpEverything(prevFeats,currFeats,prevImg,currImg,kfilter,NULL,0
,0.0,0.0,0.0);
                  break;
            }

            // Fuzzy Cluster / Best Cluster Calc Rotations

      numClusters=SIFTFuzzy::FuzzyCluster(matches,mIndex,numMatches,

membership,SIFTFuzzy::CLUSTER_FLAG_ROTATION);

      SIFTFuzzy::SelectBestCluster(membership,numClusters,numMatches,

matches,mIndex,1.0f/(2.0f*numClusters));

//          // Rotation Outlier Removal
            SIFTFuzzy::RemoveOutliersIQR(matches,mIndex,numMatches);


      theta_vel=SIFTFuzzy::MeanOfMatches(matches,mIndex,numMatches,

SIFTFuzzy::FLAG_ROTATION);
//          theta_vel=-theta_vel;
            std::cout << theta_vel<<std::endl;

            // Calculate translation vectors

      SIFTFuzzy::AugmentedSIFTList::CalculateTranslationOfMatches(match
es,

mIndex,numMatches,theta_vel);

            // Fuzzy Cluster / Best Cluster Calc Translations

      numClusters=SIFTFuzzy::FuzzyCluster(matches,mIndex,numMatches,

membership,SIFTFuzzy::CLUSTER_FLAG_TRANSLATION);

      SIFTFuzzy::SelectBestCluster(membership,numClusters,numMatches,

matches,mIndex,1.0f/(numClusters));

            x_vel=SIFTFuzzy::MeanOfMatches(matches,mIndex,numMatches,

SIFTFuzzy::FLAG_HORIZONTAL_TX);

      x_vel=(x_vel>=0)?(float)((int)(x_vel+0.5)):(float)((int)(x_vel-
0.5));

            y_vel=SIFTFuzzy::MeanOfMatches(matches,mIndex,numMatches,

SIFTFuzzy::FLAG_VERTICAL_TX);
```

```
      y_vel=(y_vel>=0)?(float)((int)(y_vel+0.5)):(float)((int)(y_vel-
0.5));

            // Accumulate Motion
            temp_accum=x_accum;

      x_accum=cos(theta_vel)*temp_accum+sin(theta_vel)*y_accum+x_vel;
            y_accum=cos(theta_vel)*y_accum-
sin(theta_vel)*temp_accum+y_vel;
            theta_accum+=theta_vel;

            // Kalman Filter using Calculated Motion
            kfilter.update(x_vel,y_vel,theta_vel);
            kfilter.GetPositions(x_desired,y_desired,theta_desired);
            std::cout<<"Theta accum = "<<theta_accum<<", theta
des="<<theta_desired<<"\n";

            // Update Trust Values

      AugmentedSIFTList::UpdateTrust(prevFeats,currFeats,mIndex,numMatc
hes);

            // Compensate

      SIFTFuzzy::Image::WarpImageRigid(capturedImg,compensatedImg,
                                       x_accum-x_desired,y_accum-
y_desired,
                                       theta_accum-theta_desired,

SIFTFuzzy::Image::IMAGE_INTERPOLATE_BILINEAR,1);
            cv::imshow(compensatedImg.imgName,compensatedImg.data);
//          cv::imshow(capturedImg.imgName,capturedImg.data);
            cv::waitKey(20); // Change to a lower number if desired

            // Set up for next iteration
            prevImg=currImg;
            prevFeats=currFeats;
      }

}

void StabilizeFile(const std::string filename)
{
      // Declare all variables and Initialize

      // Kalman Filter with all motion desired
//    SIFTFuzzy::VideoKalman
kfilter(SIFTFuzzy::VideoKalman::DESIRED_TRANSLATION,0.1,0.1);
      SIFTFuzzy::VideoKalman
kfilter(SIFTFuzzy::VideoKalman::DESIRED_NONE,0.1,2.1);

      // Variables for storing motion
      float x_vel,y_vel,theta_vel; // estimated for each frame
```

```
	float x_accum,y_accum,theta_accum,temp_accum;
	float x_desired,y_desired,theta_desired; // desired i.e. Kalman

	// Image Variables
	SIFTFuzzy::Image prevImg,currImg,compensatedImg,capturedImg;

	// SIFT variables
	SiftGPU *sift=new SiftGPU;
	SiftMatchGPU *matcher=new SiftMatchGPU;
	SIFTFuzzy::AugmentedSIFTList prevFeats,currFeats;
	std::vector<SIFTFuzzy::Touple> mIndex;
	std::vector<SIFTFuzzy::SIFTMatches> matches;
	int numMatches;

	// Fuzzy Variables
	std::vector<std::vector<float> > membership(200);
	for(int i=0;i<(int) (membership.size());++i){
		membership[i]=std::vector<float>(20);
	}
	int numClusters;

	// Time variables
	time_t t0,t1;
	timeval tt0,tt1;

	// Initialize capture device and get image size
	cv::VideoCapture cap(filename); // choose first available video
device
	if(!cap.isOpened()){
		std::cout << "Error opening capture device"<<std::endl;
		return;
	}

	// Get video resolution and number of frames
	int width=(int) (cap.get(CV_CAP_PROP_FRAME_WIDTH));
	int height=(int)(cap.get(CV_CAP_PROP_FRAME_HEIGHT));
	int numFrames=(int)(cap.get(CV_CAP_PROP_FRAME_COUNT));

	// Initialize SIFT Calculator and Matcher
	SIFTFuzzy::AugmentedSIFTList::InitializeSiftGPU(sift,matcher);

	// Initialize Images
	compensatedImg=SIFTFuzzy::Image(width,height,SIFTFuzzy::Image::IM
G_FORMAT_RGB,"Compensated Video");
	capturedImg=SIFTFuzzy::Image(width,height,SIFTFuzzy::Image::IMG_F
ORMAT_RGB,"Captured Video");
	prevImg=SIFTFuzzy::Image(width,height,SIFTFuzzy::Image::IMG_FORMA
T_GRAY);
	currImg=SIFTFuzzy::Image(width,height,SIFTFuzzy::Image::IMG_FORMA
T_GRAY);

	// Get start time
	t0=time(NULL);
	gettimeofday(&tt0,NULL);
```

```
	// Get and Show initial frame
	cv::namedWindow(compensatedImg.imgName,CV_WINDOW_AUTOSIZE);
//	cv::namedWindow(capturedImg.imgName,CV_WINDOW_AUTOSIZE);
	cap>>(capturedImg.data);
	cv::cvtColor(capturedImg.data,prevImg.data,CV_RGB2GRAY);
//	cv::cvtColor(capturedImg.data,prevImg.data,CV_BGR2GRAY);

	prevFeats.CalculateSIFTFeatures(sift,prevImg);
	compensatedImg.data=capturedImg.data.clone();
	cv::imshow(compensatedImg.imgName,compensatedImg.data);
//	cv::imshow(capturedImg.imgName,capturedImg.data);
	cv::waitKey(2);

	for(int q=1;q<numFrames;++q){ // loop 2->end
		// Perform SIFT+Matching+Augmentation
		// Capture image
		cap>>(capturedImg.data);
		cv::cvtColor(capturedImg.data,currImg.data,CV_RGB2GRAY);
//		currImg.data=capturedImg.data.clone();
//		cv::cvtColor(capturedImg.data,currImg.data,CV_BGR2GRAY);

		// Calculate and Match SIFT
		currFeats.CalculateSIFTFeatures(sift,currImg);

	numMatches=SIFTFuzzy::AugmentedSIFTList::MatchSIFTFeatures(matche
r,

prevFeats,currFeats,matches,mIndex,0);
		if(numMatches==0){
			std::cout<<"No matches, saving dump\n";

	DumpEverything(prevFeats,currFeats,prevImg,currImg,kfilter,NULL,0
,0.0,0.0,0.0);
			break;
		}

		// Fuzzy Cluster / Best Cluster Calc Rotations

	numClusters=SIFTFuzzy::FuzzyCluster(matches,mIndex,numMatches,

membership,SIFTFuzzy::CLUSTER_FLAG_ROTATION);

	SIFTFuzzy::SelectBestCluster(membership,numClusters,numMatches,

matches,mIndex,1.0/(2.0*numClusters));

		// Rotation Outlier Removal
		SIFTFuzzy::RemoveOutliersIQR(matches,mIndex,numMatches);


	theta_vel=SIFTFuzzy::MeanOfMatches(matches,mIndex,numMatches,

SIFTFuzzy::FLAG_ROTATION);
```

```
            theta_vel=-theta_vel;

            // Calculate translation vectors

      SIFTFuzzy::AugmentedSIFTList::CalculateTranslationOfMatches(match
es,

mIndex,numMatches,theta_vel);

            // Fuzzy Cluster / Best Cluster Calc Translations

      numClusters=SIFTFuzzy::FuzzyCluster(matches,mIndex,numMatches,

membership,SIFTFuzzy::CLUSTER_FLAG_TRANSLATION);

      SIFTFuzzy::SelectBestCluster(membership,numClusters,numMatches,

matches,mIndex,1.0/(2.0*numClusters));

            x_vel=SIFTFuzzy::MeanOfMatches(matches,mIndex,numMatches,

SIFTFuzzy::FLAG_HORIZONTAL_TX);

      x_vel=(x_vel>=0)?(float)((int)(x_vel+0.5)):(float)((int)(x_vel-
0.5));

            y_vel=SIFTFuzzy::MeanOfMatches(matches,mIndex,numMatches,

SIFTFuzzy::FLAG_VERTICAL_TX);

      y_vel=(y_vel>=0)?(float)((int)(y_vel+0.5)):(float)((int)(y_vel-
0.5));

            // Accumulate Motion
            temp_accum=x_accum;

      x_accum=cos(theta_vel)*temp_accum+sin(theta_vel)*y_accum+x_vel;
            y_accum=cos(theta_vel)*y_accum-
sin(theta_vel)*temp_accum+y_vel;
            theta_accum+=theta_vel;

            // Kalman Filter using Calculated Motion
            kfilter.update(x_vel,y_vel,theta_vel);
            kfilter.GetPositions(x_desired,y_desired,theta_desired);

            // Update Trust Values

      AugmentedSIFTList::UpdateTrust(prevFeats,currFeats,mIndex,numMatc
hes);

            // Compensate

      SIFTFuzzy::Image::WarpImageRigid(capturedImg,compensatedImg,
```

```
                                        x_accum-x_desired,y_accum-
y_desired,
                                        theta_accum-theta_desired,

SIFTFuzzy::Image::IMAGE_INTERPOLATE_BILINEAR,1);
            cv::imshow(compensatedImg.imgName,compensatedImg.data);
//          cv::imshow(capturedImg.imgName,capturedImg.data);
            cv::waitKey(2); // increase for smaller videos or it will
be fast

            // Set up for next iteration
            prevImg=currImg;
            prevFeats=currFeats;
      }

      t1=time(NULL);
      gettimeofday(&tt1,NULL);

      double diff=(tt1.tv_sec+tt1.tv_usec/1000000.f)-
(tt0.tv_sec+tt0.tv_usec/1000000.f);

      std::cout << "Total number of seconds: " << (t1-t0) << std::endl;
      std::cout << "Total number of seconds: " << (tt1.tv_sec-
tt0.tv_sec) << std::endl;
      std::cout << "USEC diff: "<<(tt1.tv_usec-tt0.tv_usec)<<std::endl;
      std::cout << "Total diff: "<<diff<<std::endl;
      std::cout << "Framerate: "<<numFrames/diff<<std::endl;


}

void StabilizeFileWithMatches(const std::string filename,const
std::string directory)
{
      // Declare all variables and Initialize

      // Kalman Filter with all motion desired
//    SIFTFuzzy::VideoKalman
kfilter(SIFTFuzzy::VideoKalman::DESIRED_TRANSLATION,0.1,0.1);
      SIFTFuzzy::VideoKalman
kfilter(SIFTFuzzy::VideoKalman::DESIRED_NONE,0.1,2.1);

      // Variables for storing motion
      float x_vel,y_vel,theta_vel; // estimated for each frame
      float x_accum,y_accum,theta_accum,temp_accum;
      float x_desired,y_desired,theta_desired; // desired i.e. Kalman

      // Image Variables
      SIFTFuzzy::Image prevImg,currImg,compensatedImg,capturedImg;

      // SIFT variables
      SIFTFuzzy::AugmentedSIFTList prevFeats,currFeats;
      std::vector<SIFTFuzzy::Touple> mIndex;
      std::vector<SIFTFuzzy::SIFTMatches> matches;
```

```
	int numMatches;

	// Fuzzy Variables
	std::vector<std::vector<float> > membership(200);
	for(int i=0;i<(int) (membership.size());++i){
		membership[i]=std::vector<float>(20);
	}
	int numClusters;


	// Initialize capture device and get image size
	cv::VideoCapture cap(filename); // choose first available video
device
	if(!cap.isOpened()){
		std::cout << "Error opening capture device"<<std::endl;
		return;
	}

	// Get video resolution and number of frames
	int width=(int) (cap.get(CV_CAP_PROP_FRAME_WIDTH));
	int height=(int)(cap.get(CV_CAP_PROP_FRAME_HEIGHT));
	int numFrames=(int)(cap.get(CV_CAP_PROP_FRAME_COUNT));

	// Initialize Images
	compensatedImg=SIFTFuzzy::Image(width,height,SIFTFuzzy::Image::IM
G_FORMAT_RGB,"Compensated Video");
	capturedImg=SIFTFuzzy::Image(width,height,SIFTFuzzy::Image::IMG_F
ORMAT_RGB,"Captured Video");
	prevImg=SIFTFuzzy::Image(width,height,SIFTFuzzy::Image::IMG_FORMA
T_GRAY);
	currImg=SIFTFuzzy::Image(width,height,SIFTFuzzy::Image::IMG_FORMA
T_GRAY);

	// Get and Show initial frame
	cv::namedWindow(compensatedImg.imgName,CV_WINDOW_AUTOSIZE);
//	cv::namedWindow(capturedImg.imgName,CV_WINDOW_AUTOSIZE);
	cap>>(capturedImg.data);
//	cv::cvtColor(capturedImg.data,prevImg.data,CV_RGB2GRAY);
//	cv::cvtColor(capturedImg.data,prevImg.data,CV_BGR2GRAY);

	prevFeats.GetSIFTFeaturesFromFile(1);
	compensatedImg.data=capturedImg.data.clone();
	cv::imshow(compensatedImg.imgName,compensatedImg.data);
	cv::waitKey(30);

	for(int q=2;q<=numFrames;++q){ // loop 2->end
		// Perform SIFT+Matching+Augmentation
		// Capture image
		cap>>(capturedImg.data);
//		cv::cvtColor(capturedImg.data,currImg.data,CV_RGB2GRAY);
//		cv::cvtColor(capturedImg.data,currImg.data,CV_BGR2GRAY);

		// Calculate and Match SIFT
		currFeats.GetSIFTFeaturesFromFile(q);
```

```
      numMatches=SIFTFuzzy::AugmentedSIFTList::GetMatchesFromFile(q,pre
vFeats,matches,mIndex);

            // Fuzzy Cluster / Best Cluster Calc Rotations

      numClusters=SIFTFuzzy::FuzzyCluster(matches,mIndex,numMatches,

membership,SIFTFuzzy::CLUSTER_FLAG_ROTATION);

      SIFTFuzzy::SelectBestCluster(membership,numClusters,numMatches,

matches,mIndex,1.0f/(2.0f*numClusters));

            // Rotation Outlier Removal
            SIFTFuzzy::RemoveOutliersIQR(matches,mIndex,numMatches);


      theta_vel=SIFTFuzzy::MeanOfMatches(matches,mIndex,numMatches,

SIFTFuzzy::FLAG_ROTATION);
//          theta_vel=-theta_vel;

            // Calculate translation vectors

      SIFTFuzzy::AugmentedSIFTList::CalculateTranslationOfMatches(match
es,

mIndex,numMatches,theta_vel);

            // Fuzzy Cluster / Best Cluster Calc Translations

      numClusters=SIFTFuzzy::FuzzyCluster(matches,mIndex,numMatches,

membership,SIFTFuzzy::CLUSTER_FLAG_TRANSLATION);

      SIFTFuzzy::SelectBestCluster(membership,numClusters,numMatches,

matches,mIndex,1.0f/(numClusters));
            x_vel=SIFTFuzzy::MeanOfMatches(matches,mIndex,numMatches,

SIFTFuzzy::FLAG_HORIZONTAL_TX);

      x_vel=(x_vel>=0)?(float)((int)(x_vel+0.5)):(float)((int)(x_vel-
0.5));

            y_vel=SIFTFuzzy::MeanOfMatches(matches,mIndex,numMatches,

SIFTFuzzy::FLAG_VERTICAL_TX);

      y_vel=(y_vel>=0)?(float)((int)(y_vel+0.5)):(float)((int)(y_vel-
0.5));

            // Accumulate Motion
```

```
            temp_accum=x_accum;

      x_accum=cos(theta_vel)*temp_accum+sin(theta_vel)*y_accum+x_vel;
            y_accum=cos(theta_vel)*y_accum-
sin(theta_vel)*temp_accum+y_vel;
            theta_accum+=theta_vel;

            // Kalman Filter using Calculated Motion
            kfilter.update(x_vel,y_vel,theta_vel);
            kfilter.GetPositions(x_desired,y_desired,theta_desired);

            // Update Trust Values

      AugmentedSIFTList::UpdateTrust(prevFeats,currFeats,mIndex,numMatc
hes);

            // Compensate

      SIFTFuzzy::Image::WarpImageRigid(capturedImg,compensatedImg,
                                       x_accum-x_desired,y_accum-
y_desired,
                                       theta_accum-theta_desired,

SIFTFuzzy::Image::IMAGE_INTERPOLATE_BILINEAR,1);
            cv::imshow(compensatedImg.imgName,compensatedImg.data);
            cv::waitKey(30);

            // Set up for next iteration
            prevImg=currImg;
            prevFeats=currFeats;
      }


}



void DumpEverything(AugmentedSIFTList &prevS,
                        AugmentedSIFTList &currS,
                        Image &prevI,Image &currI,
                        VideoKalman &filter,std::vector<SIFTMatches>
*matches,
                        int numMatches,
                        double tx,double ty,double rot)
{
      std::ofstream Img1,Img2,SIFTs,Estimates,KFilter;

      std::cout<<"Estimates\n";
      Estimates.open("Estimates.txt");
      Estimates <<tx<<","<<ty<<","<<rot<<"\n";
      Estimates.close();

      std::cout<<"PrevImg\n";
      Img1.open("previousImg.pgm",std::ios::binary);
```

```
	Img1 <<"P5\n"<<prevI.width<<" "<<prevI.height<<"\n255\n";
	int size=prevI.width*prevI.height;
	unsigned char *p=prevI.data.data;
	for(int i=0;i<size;++i){
		Img1 << *(p+i);
	}
	Img1.close();

	std::cout<<"CurrImg\n";
	Img2.open("currImg.pgm",std::ios::binary);
	Img2 <<"P5\n"<<currI.width<<" "<<currI.height<<"\n255\n";
	size=currI.width*currI.height;
	p=currI.data.data;
	for(int i=0;i<size;++i){
		Img2 << *(p+i);
	}
	Img2.close();

	std::cout<<"KFilter\n";
	float x,y,t,xd,yd,td;
	filter.GetStateValues(x,y,t,xd,yd,td);
	KFilter.open("KalmanStates.txt");
	KFilter<<x<<","<<y<<","<<t<<","<<xd<<","<<yd<<","<<td<<"\n";
	KFilter.close();

	std::cout<<"Keys1\n";
	SIFTs.open("SIFT1keys.txt");
	for(int i=0;i<prevS.numFeatures;++i){

	SIFTs<<prevS.keypoints[i].x<<","<<prevS.keypoints[i].y<<","<<prev
S.keypoints[i].o<<"\n";
	}
	SIFTs.close();
	std::cout<<"Descrs1\n";
	SIFTs.open("SIFT1descrs.txt");
	float *ptr=(float*) &(prevS.descriptors[0]);
	size=prevS.numFeatures*128;
	for(int i=0;i<size;++i){
		SIFTs<<*ptr<<",";
		++ptr;
		if(i%128==127) SIFTs<<"\n";
	}
	SIFTs.close();

	std::cout<<"Keys2\n";
	SIFTs.open("SIFT2keys.txt");
	for(int i=0;i<currS.numFeatures;++i){

	SIFTs<<currS.keypoints[i].x<<","<<currS.keypoints[i].y<<","<<curr
S.keypoints[i].o<<"\n";
	}
	SIFTs.close();
	std::cout<<"Descrs2\n";
	SIFTs.open("SIFT2descrs.txt");
```

```
	ptr=(float*) &(currS.descriptors[0]);
	size=currS.numFeatures*128;
	for(int i=0;i<size;++i){
		SIFTs<<*ptr<<",";
		++ptr;
		if(i%128==127) SIFTs<<"\n";
	}
	SIFTs.close();

	std::cout<<"Matches\n";
	if(matches!=NULL){
		SIFTs.open("matches.txt");
		for(int i=0;i<numMatches;++i){
			SIFTs<<matches->at(i).x1<<","<<matches-
>at(i).y1<<",";
			SIFTs<<matches->at(i).x2<<","<<matches-
>at(i).y2<<",";
			SIFTs<<matches->at(i).trust<<","<<matches-
>at(i).rot<<",";
			SIFTs<<matches->at(i).tx<<","<<matches-
>at(i).ty<<"\n";
		}
		SIFTs.close();
	}
}

}
```

SIFTFuzzyStructs.hpp

```
#ifndef SIFTFUZZYSTRUCTS_HPP
#define SIFTFUZZYSTRUCTS_HPP

namespace SIFTFuzzy{

typedef struct SIFTMatches_{
      float x1,y1,x2,y2,tx,ty,rot,trust;
      SIFTMatches_ &operator=(const SIFTMatches_ &other);
} SIFTMatches;

typedef struct Touple_{
      int v1,v2;
      Touple_ &operator=(const Touple_ &other);
} Touple;

}

#endif
```

SIFTFuzzyStructs.cpp

```
#include "SIFTFuzzyStructs.hpp"

namespace SIFTFuzzy{

SIFTMatches_ &SIFTMatches_::operator=(const SIFTMatches_ &other){
      this->x1=other.x1;
      this->y1=other.y1;
      this->x2=other.x2;
      this->y2=other.y2;
      this->tx=other.tx;
      this->ty=other.ty;
      this->rot=other.rot;
      this->trust=other.trust;
      return *this;
}

Touple_ &Touple_::operator=(const Touple_ &other){
      this->v1=other.v1;
      this->v2=other.v2;
      return *this;
}

}
```

Image.hpp

```
#ifndef IMAGE_HPP
#define IMAGE_HPP

#include <opencv/cv.h>
#include <opencv/highgui.h>
#include <string>

namespace SIFTFuzzy{

class Image{
public:
      // Possible values for image format
      static const int
IMG_FORMAT_GRAY=0,IMG_FORMAT_YUYV=1,IMG_FORMAT_RGB=2;
      // Possible methods for interpolation
      static const int
IMAGE_INTERPOLATE_BILINEAR=0,IMAGE_INTERPOLATE_NN=1;

      // Constructors
      Image();
      Image(int width,int height,int format,const std::string
imgName="Image");
      Image(int width,int height,int format,unsigned char *data,const
std::string imgName="Image");
      Image(int width,int height,int format,cv::Mat &data,const
std::string imgName="Image");

      // Public member functions
      // Warps the image with respect to the reverse rigid motion model
      static void WarpImageRigid(Image &in,Image &out,
                              double tx, double ty, double rotation,
                              int
interpMethod=Image::IMAGE_INTERPOLATE_BILINEAR,
                              int invert=0);

      // getters/setters
      void SetName(const std::string &name);
      unsigned char *GetDataPointer();

      // operators
      Image &operator=(const Image &I);

      // Image properties and data, public for ease of access
      int width,height;
      int format;
      std::string imgName;
      cv::Mat data;
};

}

#endif
```

Image.cpp

```
#include "Image.hpp"
#include <iostream>
#include <cmath>

namespace SIFTFuzzy{
	// Constructors
	Image::Image()
	{
		// empty
	}

	Image::Image(int width,int height,int format,unsigned char
*data,const std::string imgName)
	{
		int fmtflag,step;

		this->width=width;
		this->height=height;
		this->format=format;

		if(format == Image::IMG_FORMAT_GRAY){
			fmtflag=CV_8UC1;
			step=width;
		}
		else if(format == Image::IMG_FORMAT_RGB){
			fmtflag=CV_8UC3;
			step=width*3;
		}
		else if(format == Image::IMG_FORMAT_YUYV){
			fmtflag=CV_8UC2;
			step=width*2;
		}
		else{
			fmtflag=CV_8UC3; // default to 3 channels
			step=width*3;
		}

		this->data=cv::Mat(height,width,fmtflag,data);

		this->data.step=step;

		this->imgName=imgName;

	}

	Image::Image(int width,int height,int format,const std::string
imgName)
	{
		int fmtflag;

		this->width=width;
		this->height=height;
```

```
            this->format=format;

            if(format == Image::IMG_FORMAT_GRAY)
                  fmtflag=CV_8UC1;
            else if(format == Image::IMG_FORMAT_RGB)
                  fmtflag=CV_8UC3;
            else if(format == Image::IMG_FORMAT_YUYV)
                  fmtflag=CV_8UC2;
            else
                  fmtflag=CV_8UC3; // default to 3 channels

            this->data=cv::Mat(height,width,fmtflag);

            this->imgName=imgName;
      }

      Image::Image(int width,int height,int format,cv::Mat &data,const
std::string imgName){
            int fmtflag;

            this->width=width;
            this->height=height;
            this->format=format;

            if(format == Image::IMG_FORMAT_GRAY)
                  fmtflag=CV_8UC1;
            else if(format == Image::IMG_FORMAT_RGB)
                  fmtflag=CV_8UC3;
            else if(format == Image::IMG_FORMAT_YUYV)
                  fmtflag=CV_8UC2;
            else
                  fmtflag=CV_8UC3; // default to 3 channels

            this->data=data.clone();

            this->imgName=imgName;
      }

      // Methods
      void Image::WarpImageRigid(Image &in, Image &out,
                                          double tx, double ty,
double rotation,
                                          int interpMethod,int
invert)
      {
            int interpflag;

            cv::Mat tempholder(in.height,in.width,CV_8UC3);
            out.data=cv::Mat(in.height,in.width,CV_8UC3);
            cv::Mat warpMatrix;

            if(interpMethod==Image::IMAGE_INTERPOLATE_NN)
                  interpflag=cv::INTER_NEAREST;
            else
```

```
                  interpflag=cv::INTER_LINEAR; // default to bilinear
for invalids

            if(invert)
                  interpflag|=cv::WARP_INVERSE_MAP;


            warpMatrix=cv::getRotationMatrix2D(

cv::Point2f((in.width+1.0f)/2.0f,(in.height+1.0f)/2.0f),
                rotation*180/3.1415927,1);
            std::cout<<"tx "<<tx<<", ty "<<ty<<std::endl;
            warpMatrix.at<double>(0,2)+=tx;
            warpMatrix.at<double>(1,2)+=ty;


            cv::warpAffine(in.data,out.data,warpMatrix,
                              cv::Size(in.width,in.height),
                              interpflag,cv::BORDER_CONSTANT,0);
      }

      void Image::SetName(const std::string &name){
            this->imgName=name;
      }

      unsigned char *Image::GetDataPointer(){
            return (unsigned char*)this->data.data;
      }

      Image &Image::operator=(const Image &I){
            if(this!=&I){
                  if(! this->data.empty())
                        this->data.release();
                  this->width=I.width;
                  this->height=I.height;
                  this->format=I.format;
                  this->data=I.data.clone();
                  this->imgName=I.imgName;
            }
            return *this;
      }

}
```

Clustering.hpp

```
#ifndef CLUSTERING_HPP
#define CLUSTERING_HPP

#include <vector>
#include <opencv/cv.h>
#include "SIFTFuzzyStructs.hpp"

namespace SIFTFuzzy{

// Cluster
int FuzzyCluster(const std::vector<SIFTMatches> &matches,
                         std::vector<Touple> &mIndex,
                         int numMatches,
                         std::vector<std::vector<float> > &membership,
                         int txRotFlag);

// Choose cluster with highest weighted trust
void SelectBestCluster(const std::vector<std::vector<float> >
&membership,
                               int numClusters,int numMatches,
                               std::vector<SIFTMatches> &matches,
                          std::vector<Touple> &mIndex,float
threshold);

// Outlier removal for Orientation
void RemoveOutliersIQR(std::vector<SIFTMatches> &matches,
                         std::vector<Touple> &mIndex,int numMatches);

// Supplementary Methods
float EuclideanDistance(const cv::Mat &p1,const cv::Mat &p2);
float FindIthMember(std::vector<float> &data,int ith); // not currently
used any more

// Get the mean of rotation, horizontal translation, or vertical
translation
float MeanOfMatches(const std::vector<SIFTMatches> &matches,
                       std::vector<Touple> &matchIndex,
                       int numMatches,int txRotFlag);


// Flags and such
static const int FLAG_ROTATION=0;
static const int FLAG_HORIZONTAL_TX=1;
static const int FLAG_VERTICAL_TX=2;

static const int CLUSTER_FLAG_ROTATION=0;
static const int CLUSTER_FLAG_TRANSLATION=1;

}

#endif
```

Clustering.cpp

```
#include "Clustering.hpp"
#include <cmath>
#include <iostream>
#include <vector>

namespace SIFTFuzzy{

int FuzzyCluster(const std::vector<SIFTMatches> &matches,
                     std::vector<Touple> &mIndex,int numMatches,
                     std::vector<std::vector<float> > &membership,
                     int txrotflag)
{
      // Local Variables
      cv::Mat centers,lbls; // no need to initialize
      cv::Mat data;
      float dist,sum;
      int k;

      if(txrotflag){
            // translation
            int count=0;
            for(int i=0;i<numMatches;++i){
                  if(mIndex[i].v1!=-1){
                        ++count;
                  }
            }
            data=cv::Mat(count,2,CV_32FC1);
            count = 0;
            for(int i=0;i<numMatches;++i){
                  if(mIndex[i].v1!=-1){
                        data.at<float>(count,0)=matches[i].tx;
                        data.at<float>(count,1)=matches[i].ty;
                        ++count;
                  }
            }
            k=(int)ceil(sqrt(count/ 2.0f));
      }
      else{
            k=(int)ceil(sqrt(numMatches/ 2.0f));
            data=cv::Mat(numMatches,1,CV_32FC1);
            for(int i=0;i<numMatches;++i){
                  data.at<float>(i,0)=matches[i].rot;
            }
      }

      // Kmeans to identify cluster centroids

      cv::kmeans(data,k,lbls,cv::TermCriteria(),20,cv::KMEANS_PP_CENTER
S,&centers);
//
      cv::kmeans(data,k,lbls,cv::TermCriteria(),5,cv::KMEANS_RANDOM_CEN
TERS,&centers);
```

```
	// Resize membership if needed
	if((int) membership.size()<numMatches){
		membership.resize(numMatches);
	}
	for(int i=0;i<numMatches;++i){
		if((int) membership[i].size()<centers.rows)
			membership[i].resize(centers.rows);
	}

	k=0;
	for(int i=0;i<numMatches;++i){
		if(mIndex[i].v1!=-1){
			// Initial memberships: 1/Euclid or 1
			sum=0.0;
			for(int j=0;j<centers.rows;++j){

	dist=EuclideanDistance(data.row(k),centers.row(j));
//				dist=0.0;
//				for(int q=0;q<data.cols;++q){
//					dist+=(data.at<float>(k,q)-
centers.at<float>(j,q))*(data.at<float>(k,q)-centers.at<float>(j,q));
//				}
//				dist=sqrt(dist);
				membership[i][j]= (dist==0.0) ? 1 : 1.0f/dist;
				sum+=membership[i][j];
			}

			// Normalize row
			for(int j=0;j<centers.rows;++j){
				membership[i][j] /= sum;
			}

			++k;
		}
	}

	return centers.rows;

}

float EuclideanDistance(const cv::Mat &p1,const cv::Mat &p2)
{
	if(p1.cols!=p2.cols){
		return -1;
	}

	float dist=0.0;
	for(int i=0;i<p1.cols;++i){
		dist+=(p1.at<float>(0,i)-
p2.at<float>(0,i))*(p1.at<float>(0,i)-p2.at<float>(0,i));
	}

	return sqrt(dist);
```

```
}

void SelectBestCluster(const std::vector<std::vector<float> >
&membership,
                                int numClusters,int numMatches,
                           std::vector<SIFTMatches> &matches,
                        std::vector<Touple> &mIndex,float threshold)
{
      // local variables
      std::vector<float> sums(numClusters);
      std::fill(sums.begin(),sums.end(),0);

      int maxi=0;

      // Find the weighted sum of trust
      for(int i=0;i<numMatches;++i){
            if(mIndex[i].v1!=-1){
                  for(int j=0;j<numClusters;++j){
                        // Sum weighted trust
                        sums[j]+=matches[i].trust*membership[i][j];
                  }
            }
      }

      // find max weighted trust index
      for(int i=1;i<numClusters;++i){
            if(sums[i]>sums[maxi])
                  maxi=i;
      }

      // Count members above threshold and mark bad ones
      for(int i=0;i<numMatches;++i){
            if(mIndex[i].v1!=-1){
                  if(membership[i][maxi]<threshold)
                        mIndex[i].v1=-1;
            }
      }

}

void RemoveOutliersIQR(std::vector<SIFTMatches> &matches,
                           std::vector<Touple> &mIndex,int numMatches)
{
      int cnt=0,tempi=0;
      float lq,uq,iqr;
      SIFTMatches tempM;
      Touple tempT;

      // slow in-place sort to help with memory
      // move any -1 indices to the end, and sort only good matches
      for(int i=0;i<numMatches;++i){
            if(mIndex[i].v1==-1){
                  //move it
                  tempT=mIndex[i];
```

```
                mIndex[i]=mIndex[numMatches-tempi-1];
                mIndex[numMatches-tempi-1]=tempT;
                ++tempi;
            }
            else{
                ++cnt;
            }
            if(cnt+tempi==numMatches)
                break;
        }

        // slow sort
        for(int i=0;i<cnt;++i){
            tempi=i;
            for(int j=i+1;j<cnt;++j){
                if(matches[j].rot<matches[tempi].rot)
                    tempi=j;
            }
            tempT=mIndex[i];
            mIndex[i]=mIndex[tempi];
            mIndex[tempi]=tempT;

            tempM=matches[i];
            matches[i]=matches[tempi];
            matches[tempi]=tempM;
        }

        uq=matches[3*(cnt+1)/4-1].rot;
        lq=matches[(cnt+1)/4-1].rot;

        iqr=uq-lq;
        lq-=1.5f*iqr;
        uq+=1.5f*iqr;

        // flag outliers
        cnt=0;
        for(int i=0;i<numMatches;++i){
            if(mIndex[i].v1!=-1){
                if(matches[i].rot<lq || matches[i].rot>uq){
                    mIndex[i].v1=-1;
                }
                else ++cnt;
            }
        }
    }

    float FindIthMember(std::vector<float> &inSet,int ith){
        int pivot,left=0,right=inSet.size()-1,i;
        float temp,ret=0.0;

        //loop
        while(1){
            // exit if left=right
            if(left==right){
```

```
                ret=inSet[left];
                break;
            }

            // choose a random member
            pivot=rand()%(right-left+1)+left;

            // separate list into two parts
            temp=inSet[pivot];
            inSet[pivot]=inSet[right];
            inSet[right]=temp;
            pivot=left;
            for(i=left;i<right;++i){
                if(inSet[i]<=inSet[right]){
                    temp=inSet[pivot];
                    inSet[pivot]=inSet[i];
                    inSet[i]=temp;
                    ++pivot;
                }
            }
            temp=inSet[pivot];
            inSet[pivot]=inSet[right];
            inSet[right]=temp;

            if(pivot==ith){
                ret=inSet[ith];
                break;
            }
            else if(pivot>ith){
                right=pivot-1;
            }
            else{
                left=pivot+1;
            }
        }

        return ret;
}


float MeanOfMatches(const std::vector<SIFTMatches> &matches,
                    std::vector<Touple> &mIndex,
                    int numMatches,int txRotFlag){
        int count=0;
        float ret=0.0f;

        if(txRotFlag==SIFTFuzzy::FLAG_ROTATION){
            for(int i=0;i<numMatches;++i){
                if(mIndex[i].v1!=-1){
                    ++count;
                    ret+=matches[i].rot;
                }
            }
        }
```

```
	else if(txRotFlag==SIFTFuzzy::FLAG_HORIZONTAL_TX){
		for(int i=0;i<numMatches;++i){
			if(mIndex[i].v1!=-1){
				++count;
				ret+=matches[i].tx;
			}
		}
	}
	else{ // FLAG_VERTICAL_TX by default
		for(int i=0;i<numMatches;++i){
			if(mIndex[i].v1!=-1){
				++count;
				ret+=matches[i].ty;
			}
		}
	}

	return ret/count;
}

}
```

AugmentedSIFTList.hpp

```
#ifndef AUGMENTEDSIFTLIST_HPP
#define AUGMENTEDSIFTLIST_HPP

#include <opencv/cv.h>
#include <vector>
#include "SIFTFuzzyStructs.hpp"
#include "Image.hpp"
#include "SiftGPU.h"

namespace SIFTFuzzy{

class AugmentedSIFTList{
public:
	// constructors
	AugmentedSIFTList();

	// Calculate and Match
	static void InitializeSiftGPU(SiftGPU *sift, SiftMatchGPU
*matcher);
	void CalculateSIFTFeatures(SiftGPU *sift,Image &img);
	static int MatchSIFTFeatures(
			SiftMatchGPU *matcher,AugmentedSIFTList &list1,
			AugmentedSIFTList &list2,std::vector<SIFTMatches>
&matches,
			std::vector<Touple> &mIndex,int localRot);

	// Calculate translation (resultant vectors in the MATLAB code)
	static void CalculateTranslationOfMatches(
			std::vector<SIFTMatches> &matches,std::vector<Touple>
&mIndex,
			int numMatches,float globalRot);

	// Increment trust values of the appropriate features
	static void UpdateTrust(AugmentedSIFTList
&prevL,AugmentedSIFTList &currL,std::vector<Touple> &mIndex,int
numMatches);

	// operators
	AugmentedSIFTList &operator=(const AugmentedSIFTList &list);

	// functions added to test code using vlfeat features from MATLAB
	void GetSIFTFeaturesFromFile(int frame);
	static int GetMatchesFromFile(int frame,AugmentedSIFTList
&prevFeats,
			std::vector<SIFTMatches> &matches,std::vector<Touple>
&mIndex);


//private:
	// Members
	std::vector<SiftGPU::SiftKeypoint> keypoints;
	std::vector<float> trustValues;
```

```
        std::vector<float> descriptors;
        int numFeatures;
};

}

#endif
```

AugmentedSIFTList.cpp

```
#include "AugmentedSIFTList.hpp"

//#include "SiftGPU.h" // Needed for GPU Sift Implementation
#include <GL/glew.h> // Needed by SiftGPU for data types/flags
#include <string>

#include <iostream>

//#define _USE_MATH_DEFINES
#include <cmath>

#include <fstream>
#include <string>
#include <sstream>

namespace SIFTFuzzy{

	const float PI=(float) std::atan((float) 1.0f)*4.0f;

	AugmentedSIFTList::AugmentedSIFTList(){};

	void AugmentedSIFTList::InitializeSiftGPU(SiftGPU
*sift,SiftMatchGPU *matcher){

		// -fo 0 -> first octave is 0, can be -1 to upscale image
first
		// -v 0 -> do not report anything during SIFT runs
		// -m 1 -> max number of orientations per feature is 1
		// -loweo -> Origin of image is set in the center of the
0,0 pixel->0.5,0.5
		// -cuda -> Use CUDA instead of OpenGL, whichever device is
available
		char *argv[]={(char*)(std::string("-
fo").c_str()),(char*)(std::string("-0").c_str()),(char*)(std::string("-
v").c_str()),

	(char*)(std::string("0").c_str()),(char*)(std::string("-
m").c_str()),(char*)(std::string("1").c_str()),
				(char*)(std::string("-
loweo").c_str()),(char*)(std::string("-cuda").c_str())};
		int argc=sizeof(argv)/sizeof(char*);

		sift->ParseParam(argc,argv);
		matcher->SetLanguage(3);
		matcher->CreateContextGL();
		matcher->VerifyContextGL();
	}

	void AugmentedSIFTList::CalculateSIFTFeatures(SiftGPU *sift,Image
&img)
	{
		// Calculate SIFT
```

```
            sift-
>RunSIFT(img.width,img.height,img.GetDataPointer(),GL_LUMINANCE,GL_UNSI
GNED_BYTE);
            this->numFeatures=sift->GetFeatureNum();

            // Check to see if the vector needs to be expanded
            if(this->numFeatures>(int)(this->keypoints.size())){
                  this->descriptors.resize(128*this->numFeatures);
                  this->keypoints.resize(this->numFeatures);
                  this->trustValues.resize(this->numFeatures);
            }

            // Get keys and descriptors
            sift->GetFeatureVector(&(this->keypoints[0]),&(this-
>descriptors[0]));

            // Make sure orientations are between -pi/pi - assuming
only a single add/subtract
            for(int i=0;i<this->numFeatures;++i){
                  if(this->keypoints[i].o>PI)
                        this->keypoints[i].o-=PI;
                  else if(this->keypoints[i].o<-PI){
                        this->keypoints[i].o+=PI;
                  }

                  // adjust positions with respect to center of frame
                  this->keypoints[i].x-=(img.width+1.0f)/2.0f;
                  this->keypoints[i].y=(img.height+1.0f)/2.0f-this-
>keypoints[i].y;
            }
            // assign nominal trust
            for(int i=0;i<this->numFeatures;++i){
                  this->trustValues[i]=1;
            }
      }

      void AugmentedSIFTList::GetSIFTFeaturesFromFile(int frame){
            std::ifstream featureFile;
            std::stringstream strs;
            SiftGPU::SiftKeypoint k;
            std::string str;

            strs<<"features";

            if(frame<1000) strs<<0;
            if(frame<100) strs<<0;
            if(frame<10) strs<<0;
            strs<<frame;
            str=strs.str();
//          std::cout << str << std::endl;
            featureFile.open(str.c_str(),std::ios::in);

            this->keypoints.clear();
            this->trustValues.clear();
```

```
            while(1){
                  getline(featureFile,str);
                  if(featureFile.eof()) break;
                  strs.str(str);
                  strs >> k.x;
                  strs >> k.y;
                  strs >> k.s;
                  strs >> k.o;

                  k.x-=(641)/2;
                  k.y-=(481)/2;
                  this->keypoints.push_back(k);
                  this->trustValues.push_back(1);
                  strs.clear();
            }

            featureFile.close();

            this->numFeatures=(int)(this->keypoints.size());

            if((int)(this->descriptors.size())<(this->numFeatures*128))
                  this->descriptors.resize(this->numFeatures*128);
      }

      int AugmentedSIFTList::MatchSIFTFeatures(
                  SiftMatchGPU *matcher,AugmentedSIFTList &list1,
                  AugmentedSIFTList &list2,std::vector<SIFTMatches>
&matches,
                  std::vector<Touple> &mIndex,int localRot)
      {
            int numMatches;
            // List 1 is matched forward to List 2
            matcher-
>SetDescriptors(0,list1.numFeatures,&(list1.descriptors[0]));
            matcher-
>SetDescriptors(1,list2.numFeatures,&(list2.descriptors[0]));

            int (*matchBuffer)[2]=new int[list1.numFeatures][2];
            numMatches=matcher-
>GetSiftMatch(list1.numFeatures,matchBuffer);

            // Allocate matches
            if((int)(matches.size())<numMatches){
                  matches.resize(numMatches);
                  mIndex.resize(numMatches);
            }

            // Fill matches and index
            for(int i=0;i<numMatches;++i){
                  // Fill mIndex
                  mIndex[i].v1=matchBuffer[i][0];
                  mIndex[i].v2=matchBuffer[i][1];

                  // Fill matches
```

```
                  // x,y of list1
                  matches[i].x1=list1.keypoints[mIndex[i].v1].x;
                  matches[i].y1=list1.keypoints[mIndex[i].v1].y;

                  // x,y of list2
                  matches[i].x2=list2.keypoints[mIndex[i].v2].x;
                  matches[i].y2=list2.keypoints[mIndex[i].v2].y;

                  // trust values - always use list1 - assumed previous
                  matches[i].trust=list1.trustValues[mIndex[i].v1];

                  // fill zeros for tx and ty
                  matches[i].tx=0;
                  matches[i].ty=0;

                  // rotation
                  matches[i].rot=list2.keypoints[mIndex[i].v2].o-
list1.keypoints[mIndex[i].v1].o;
                  if(matches[i].rot>3.141597) matches[i].rot-
=3.1415927;
                  if(matches[i].rot<-3.141597)
matches[i].rot+=3.1415927;
            }

            delete matchBuffer; // clean up for the new keyword usage

            return numMatches;
      }

      int AugmentedSIFTList::GetMatchesFromFile(int
frame,AugmentedSIFTList &prevFeats,
                  std::vector<SIFTMatches> &matches,std::vector<Touple>
&mIndex){
            std::ifstream file1,file2;
            SIFTMatches m;
            Touple t;
            float scale,ori1,ori2;
            std::string str1,str2;
            std::stringstream strs1,strs2;

            strs1<<"matches";
            strs2<<"mindex";

            if(frame<1000){
                  strs1<<0;
                  strs2<<0;
            }
            if(frame<100){
                  strs1<<0;
                  strs2<<0;
            }
            if(frame<10){
                  strs1<<0;
                  strs2<<0;
```

```
		}
		strs1<<frame;
		strs2<<frame;

		matches.clear();
		mIndex.clear();

		str1=strs1.str();
		str2=strs2.str();
		file1.open(str1.c_str(),std::ios::in);
		file2.open(str2.c_str(),std::ios::in);

		while(1){
			getline(file1,str1);
			getline(file2,str2);
			if(file1.eof()) break;
			strs1.str(str1);
			strs2.str(str2);

			strs2>>t.v1>>t.v2;

			strs1>>m.x1>>m.y1>>scale>>ori1;
			m.x1-=641/2;
			m.y1=(481.0f/2)-m.y1;
			strs1>>m.x2>>m.y2>>scale>>ori2;
			m.x2-=641/2;
			m.y2=(481.0f/2)-m.y2;
			m.rot=ori2-ori1;
			if(m.rot<-3.1415927) m.rot+=3.1415927;
			if(m.rot>3.1415927) m.rot-=3.1415927;
			m.tx=0;
			m.ty=0;
			m.trust=prevFeats.trustValues[t.v1];

			matches.push_back(m);
			mIndex.push_back(t);

			strs1.clear();
			strs2.clear();

		}
		file1.close();
		file2.close();

		return (int)(matches.size());

	}


	void AugmentedSIFTList::CalculateTranslationOfMatches(
			std::vector<SIFTMatches> &matches,std::vector<Touple>
&mIndex,
			int numMatches,float globalRot)
	{
```

```
            float cosRot=cos(globalRot),sinRot=sin(globalRot);
            int count=0;
            for(int i=0;i<numMatches;++i){
                  if(mIndex[i].v1!=-1){
                        matches[i].tx=matches[i].x2-
                                          (cosRot*matches[i].x1+
                                          sinRot*matches[i].y1);
                        matches[i].ty=matches[i].y2-
                                          (cosRot*matches[i].y1-
                                          sinRot*matches[i].x1);
                        ++count;
                  }
            }
      }


      AugmentedSIFTList & AugmentedSIFTList::operator=(const
AugmentedSIFTList &other){
            if(this!=&other){
                  // resize as necessary
                  if((int) (this->keypoints.size())<other.numFeatures){
                        this->keypoints.resize(other.numFeatures);
                        this-
>descriptors.resize(other.numFeatures*128);
                        this->trustValues.resize(other.numFeatures);
                  }
                  this->numFeatures=other.numFeatures;
                  for(int i=0;i<this->numFeatures;++i){
                        this->keypoints[i]=other.keypoints[i];
                        this->trustValues[i]=other.trustValues[i];
                  }
                  int sizer=this->numFeatures*128;
                  for(int i=0;i<sizer;++i){
                        this->descriptors[i]=other.descriptors[i];
                  }
            }
            return *this;
      }

      void AugmentedSIFTList::UpdateTrust(AugmentedSIFTList
&prevL,AugmentedSIFTList &currL,std::vector<Touple> &mIndex,int
numMatches){
            for(int i=0;i<numMatches;++i){
                  if(mIndex[i].v1!=-1){

      currL.trustValues[mIndex[i].v2]=prevL.trustValues[mIndex[i].v1]+1
;
                  }
            }
      }

}
```

VideoKalman.hpp

```
#ifndef VIDEOKALMAN_HPP
#define VIDEOKALMAN_HPP

#include <opencv/cv.h> // For matrix class
namespace SIFTFuzzy {

class VideoKalman {
public:

	// Flags
	static const int DESIRED_NONE = 0, DESIRED_HORIZONTAL = 1,
			DESIRED_VERTICAL = 2,DESIRED_TRANSLATION=3;
	static const int DESIRED_ROTATION = 4, DESIRED_ALL = 7;

	// Constructors
	VideoKalman(int desiredMotion = VideoKalman::DESIRED_ALL,
			float stateCovariance = 1, float sensorCovariance =
1);

	// Methods
	void update(float tx, float ty, float angle);

	// Setters
	void SetStateCovariance(float stateCovariance);
	void SetSensorCovariance(float sensorCovariance);
	void SetDesiredMotion(int desiredMotion);
	void SetDesiredMotion(int desiredMotion,int theta);

	// Getters
	void GetStateValues(float &xpos, float &ypos, float &angle,
			float &xvel, float &yvel, float &rot);
	void GetPositions(float &xpos, float &ypos, float &angle);
	void GetVelocities(float &xvel, float &yvel, float &rot);

private:
	cv::Mat stateVector, stateModel, stateCovar, stateProb,
sensorMask,
			sensorCovar;
	cv::Mat residualVector, residualProb, kalmanGain;
};

}
#endif
```

VideoKalman.cpp

```
#include "VideoKalman.hpp"
#include <cmath>

using namespace SIFTFuzzy;

// Constructor
VideoKalman::VideoKalman(int desiredMotion,
                         float stateCovariance,
                         float sensorCovariance)
{
      int x,y,rot;
      desiredMotion&VideoKalman::DESIRED_HORIZONTAL ? x=1 : x=0;
      desiredMotion&VideoKalman::DESIRED_VERTICAL ? y=1 : y=0;
      desiredMotion&VideoKalman::DESIRED_ROTATION ? rot=1 : rot=0;

      this->stateModel=(cv::Mat_<float>(6,6) << x,0,0,x,0,0,
                                                0,y,0,0,y,0,
                                                0,0,rot,0,0,rot,
                                                0,0,0,1,0,0,
                                                0,0,0,0,1,0,
                                                0,0,0,0,0,1);
      this->stateVector=cv::Mat::zeros(6,1,CV_32FC1);
      this->stateProb=cv::Mat::zeros(6,6,CV_32FC1);
      this->stateCovar=cv::Mat::eye(6,6,CV_32FC1)*stateCovariance;

      this->sensorMask=(cv::Mat_<float>(3,6,CV_32FC1) << 0,0,0,1,0,0,

0,0,0,0,1,0,

0,0,0,0,0,1);
      this->sensorCovar=cv::Mat::eye(3,3,CV_32FC1)*sensorCovariance;
}

// Methods
void VideoKalman::update(float tx,
                         float ty,
                         float angle)
{
      // Predict step
      this->stateVector=this->stateModel*this->stateVector;
      this->stateProb=this->stateModel*this->stateProb*(this-
>stateModel.t())+this->stateCovar;

      // Update step
      this->residualVector=(cv::Mat_<float>(3,1) << tx,ty,angle)-this-
>sensorMask*this->stateVector;
      this->residualProb=this->sensorMask*this->stateProb*(this-
>sensorMask.t())+this->sensorCovar;
      this->kalmanGain=this->stateProb*(this->sensorMask.t())*(this-
>residualProb.inv());
      this->stateVector=this->stateVector+this->kalmanGain*this-
>residualVector;
```

```
	this->stateProb=(cv::Mat::eye(6,6,CV_32FC1)-this-
>kalmanGain*this->sensorMask)*this->stateProb;
}

// Setters
void VideoKalman::SetDesiredMotion(int desiredMotion)
{
	// not valid when accumulated motion is rotated, as this should
be rotated as well
	int x,y,rot;
	desiredMotion&VideoKalman::DESIRED_HORIZONTAL ? x=1 : x=0;
	desiredMotion&VideoKalman::DESIRED_VERTICAL ? y=1 : y=0;
	desiredMotion&VideoKalman::DESIRED_ROTATION ? rot=1 : rot=0;

	this->stateModel=(cv::Mat_<float>(6,6) << x,0,0,x,0,0,
                                              0,y,0,0,y,0,
                                              0,0,rot,0,0,rot,
                                              0,0,0,1,0,0,
                                              0,0,0,0,1,0,
                                              0,0,0,0,0,1);
}

void VideoKalman::SetDesiredMotion(int desiredMotion,int theta)
{
	// this should be run at every iteration, since the state model
	// changes with respect to current orientation when desired
motion
	// exists. If desired motion is set to none, then it doesn't
matter.
	int x,y,rot;
	desiredMotion&VideoKalman::DESIRED_HORIZONTAL ? x=1 : x=0;
	desiredMotion&VideoKalman::DESIRED_VERTICAL ? y=1 : y=0;
	desiredMotion&VideoKalman::DESIRED_ROTATION ? rot=1 : rot=0;

	this->stateModel=(cv::Mat_<float>(6,6) <<
cos(theta),sin(theta),0,x,0,0,
                                              -
sin(theta),cos(theta),0,0,y,0,
                                              0,0,rot,0,0,rot,
                                              0,0,0,1,0,0,
                                              0,0,0,0,1,0,
                                              0,0,0,0,0,1);
}


void VideoKalman::SetStateCovariance(float stateCovariance)
{
	// Diagonal covariance matrix, each variable has equal cov
	this->stateCovar=cv::Mat::eye(6,6,CV_32FC1)*stateCovariance;
}

void VideoKalman::SetSensorCovariance(float sensorCovariance)
{
	// Diagonal covariance matrix, each variable has equal cov
```

```
	this->sensorCovar=cv::Mat::eye(3,3,CV_32FC1)*sensorCovariance;
}

// Getters
void VideoKalman::GetStateValues(float &xpos,float &ypos,float &angle,
                                    float &xvel,float &yvel,float
&rot)
{
	xpos=this->stateVector.at<float>(0,0);
	ypos=this->stateVector.at<float>(1,0);
	angle=this->stateVector.at<float>(2,0);

	xvel=this->stateVector.at<float>(3,0);
	yvel=this->stateVector.at<float>(4,0);
	rot=this->stateVector.at<float>(5,0);
}

void VideoKalman::GetPositions(float &xpos,float &ypos,float &angle)
{
	xpos=this->stateVector.at<float>(0,0);
	ypos=this->stateVector.at<float>(1,0);
	angle=this->stateVector.at<float>(2,0);
}

void VideoKalman::GetVelocities(float &xvel,float &yvel,float &rot)
{
	xvel=this->stateVector.at<float>(3,0);
	yvel=this->stateVector.at<float>(4,0);
	rot=this->stateVector.at<float>(5,0);
}
```