Low Power Parallel Rolling Shutter Artifact Removal

by

Nick Stupich, B.Eng.

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs in partial fulfillment of the requirements for the degree of

Master of Applied Science in Electrical and Computer Engineering

Ottawa-Carleton Institute for Electrical and Computer Engineering
Department of Electronics
Carleton University
Ottawa, Ontario

Abstract

This work presents an algorithm capable of modeling and correcting video artifacts caused by movements of a rolling shutter video camera. A distortion model is fit to feature points extracted from pairs of frames to quantity camera movements across image scanlines. An affine transformation is used to model full frame camera movements, and sinusoids model high frequency camera movements and vibrations in the x and y directions, as well as rotations. The model parameters that fit to the extracted feature points are robust to outliers using an m–estimator solution that is efficiently optimized by iteratively decreasing the m–estimator kernel width. An exponential moving average filter is used to produce smooth output camera motion before the distortion in individual frames is removed. Automated code optimization is applied to inner model fitting loops to improve performance. An implementation suitable for a low power parallel processing platform is presented.

The distortion model was found to be capable of accurately modeling rolling shutter distortions, especially those caused by high frequency camera vibrations. The m-estimator solution was found to accurately discount outlier features, and combined with the iteratively decreasing kernel width the global optimum solution is reliably and efficiently found. Automated code optimization decreased model parameter calculation time by 49 times by factoring out common terms from matrix element computations.

Contents

List of Tables

List of Figures

List of Acronyms and Symbols

G_x Image derivative taken in the x-direction.

G_y Image derivative taken in the y-direction.

I An Image.

$I(x, y)$ A pixel at location (x, y) in Image I.

I_{ds} drain to source current.

M_{cov} Covariance matrix of a sub image used for feature extraction.

$M_{min(\lambda)}$ Matrix of the minimum eigenvalue of the covariance matrix of each pixel in an image.

V_{gs} gate to source voltage.

λ Parameter used in LM optimization to interpolate between gradient descent and the Newton Raphson method.

$\psi(x)$ Influence function of an m-estimator.

$\rho(x)$ Kernel function of an m-estimator.

σ_{in} Standard deviation of inlier data points.

σ_{out} Standard deviation of outlier data points.

$f_{distort}$ frequency of distortion.

f_{frame} frame rate.

f_{row} row scanning frequency.

n_{cols} number of columns.

$n_{features}$ Maximum number of features to be extracted from a single image.

n_{in} Number of inlier data points.

n_{layers} Number of pyramid layers used in the KLT tracking algorithm.

n_{out} Number of outlier data points.

n_{rows} number of rows.

$n_{subImages}$ Number of sub-images used for feature extraction.

$qualityLevel$ Minimum eigenvalue to be considered for a feature point, relative to the $\max(M_{min(\lambda)})$.

$w(x)$ Weight function of an m-estimator.

$winSize$ Window size used to extract sub-pixel accurate features.

`centerX` The center x value, about which the image is allowed to rotate.

ADC analog to digital converter.

APS active pixel sensor.

ASIC application specific integrated circuit.

CMOS complementary metal oxide semiconductor.

CPU Central Processing Unit.

DFT discrete Fourier transform.

EKF	extended Kalman filter.
EMA	exponential moving average.
FFT	Fast Fourier transform.
FPGA	Field Programmable Gate Array.
FPU	Floating Point Unit.
GPU	Graphics Processing Unit.
IIR	Infinite Impulse Response.
KLT	Kanade-Lucas-Tomasi.
LM	Levenberg-Marquardt.
LO-RANSAC	Locally optimized RANSAC.
minDistance	Minimum distance between any two feature points.
MLE	maximum likelihood estimator.
MOSFET	Metal Oxide Semiconductor Field Effect Transistor.
NaN	Not a Number.
PPS	passive pixel sensor.
PROSAC	Progressive Sample Concensus.
RAM	Random Access Memory.
RANSAC	random sample consensus.
RMSE	root mean squared error.

SIFT Scale Invariant Feature Tracking.

SLR single lens reflex.

SURF Speeded-Up Robust Features.

SVD singular value decomposition.

Chapter 1

Introduction

1.1 Motivation

In recent years cameras are being found in more and more everyday devices, and increases in technology are bringing high quality image sensors to the mass market. Many of these cameras use a CMOS image sensor which features an electronic rolling shutter that can cause artifacts in the captured video. These cameras are often found on small devices or mounted on moving platforms, neither of which are stationary, making them susceptible to rolling shutter distortions. Many high end cameras also feature rolling shutters, but these cameras are typically much larger and heavier and therefore more difficult to move quickly enough to cause severe rolling shutter distortions. A system to correct rolling shutter distortions is required to produce visually pleasing videos without the need for professional grade equipment.

The goal of this work was to develop an algorithm to model and correct rolling shutter distortions caused by camera movements during recording. The algorithm will work with no prior knowledge of the camera settings or movements, and will output a corrected video with the artifacts corrected. A parallel implementation capable of running on a low power embedded system is also presented.

The distortion model used here is vastly different from those found in state of the art algorithms. Sinusoids model the distortion caused by camera translation and rotation instead of piecewise models that will scale with the image dimensions. An efficient fitting method was used to calculate model parameters in the presence of outliers using m-estimators with a decreasing kernel width is presented. Automated code development and optimization was used to efficiently and automatically calculate Hessian matrix entries given only a cost function.

The original goal of this work was to implement this algorithm on a portable processor designed for image processing. While this project did not end up happening, this goal drove many decisions relating to the algorithm design. This particular processor had a large number of individual processor cores, each with their own memory. Hardware floating point arithmetic was not available, leading to the work on a fixed point implementation in Section 4.3. The targeted system was a 32 bit processor contained in a very small enclosure, requiring low power consumption to avoid overheating.

Several contributions to research can be found within this work. The sub-image based feature extraction work of Grundmann *et al.* was extended to an optimal number of sub-images, allowing for more uniformly extracted features than standard feature extraction. A sinusoid based distortion model was developed to model rolling shutter distortions. An efficient fitting method was used to provide accurate distortion model parameter estimation in the presence of outlier features. An iterative, m-estimator based solution was used with a decreasing kernel parameter to reduce the contribution of outliers at each step. Automated code optimization capable of factoring common terms out of matrix element computation is presented to speed up iterations of LM optimization.

1.2 Introduction to rolling shutter sensors

Most modern digital cameras feature CMOS sensors, which use a rolling shutter to capture frames [1]. This includes everything in the range of digital single lens reflex (SLR) cameras, video cameras, and even the small cameras found in cellphones. When a camera equipped with a rolling shutter moves quickly relative to the subject of the video, several types of artifacts (described below) can be found in the captured video. The goal of this work is to detect and remove many of these artifacts.

1.2.1 CMOS pixel circuits

There are two main types of CMOS pixel circuits: active pixel sensor (APS) circuits, and passive pixel sensor (PPS) circuits [2]. PPS circuits were the first type of pixel circuits created, but are rarely used in modern cameras. APS circuits are usually used in modern CMOS image sensors, as they provide a lower noise level than the passive detectors previously used [3]. The main difference between PPS and APS circuits is that the latter use a source follower buffer between the photodiode and analog to digital converter (ADC), while the former do not.

A basic single pixel circuit is composed of the photodiode itself, a reset transistor, a source-follower amplifier and a row addressing transistor [3]. The APS detector circuit is shown in Figure 1.1. Other types of APS circuits are listed below, along with a brief description of their notable attributes:

- Photogate APS: A photogate replaces the photodiode of a basic APS circuit, resulting in lower noise [4]. These sensors are suited well to low light imaging applications.

- Logarithmic APS: A logarithmic APS replaces the reset circuitry of each pixel with a Metal Oxide Semiconductor Field Effect Transistor (MOSFET) biased to operate in the sub-threshold region. The slow leakage of this transistor acts as a low pass averaging filter for the number of photons detected, as opposed to counting the total number of photons in a typical APS circuit. Since the relationship between drain to source current (I_{ds}) and gate to source voltage (V_{gs}) is approximately exponential in this region, and the photodiode current is proportional to the amount of incident light, the voltage across the photodiode is approximately proportional to the logarithm of the light intensity [4]. Due to this logarithmic relationship, logarithmic APS circuits are well suited to high dynamic range applications [2].

- Photodiode shutter APS: A capacitor is added to the gate of the source follower circuit, and a MOSFET or transmission gate is inserted between the capacitor and the photodiode [5]. The new gate acts as a global shutter, where the voltage of each pixel can be transferred to a capacitor where it is stored until it can be read. A shutter APS will not create rolling shutter artifacts, at the expense of die area and cost required to include the extra transistors and capacitors.

A single pixel exposure can be described in three main steps:

1. The reset transistor is driven high to reset the photo diode, setting the voltage across the diode to Vdd

2. Incident photons cause the photo diode to conduct a small amount of charge to ground, each one lowering the voltage drop across the diode.

3. The row select transistor is driven high so that the output of the source follower can be read externally. The source follower buffers the voltage across

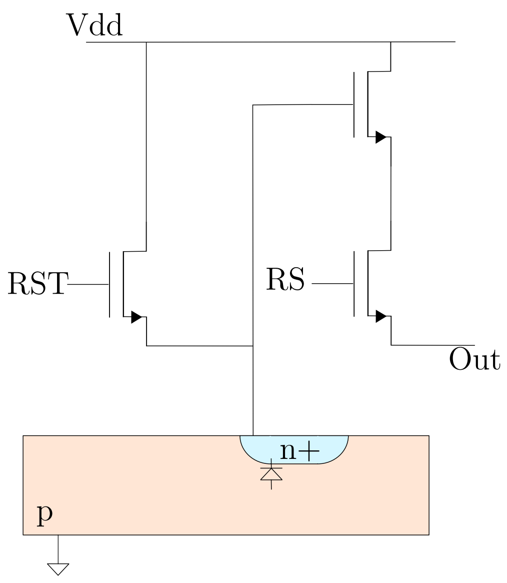


Figure 1.1: Active CMOS pixel with basic circuitry [3]. RST is the signal used to reset the photodiode to a full reverse bias and RS is the row select signal used to read the voltage on the photodiode. The diode formed by the n+ implant in the p substrate is shown.

the photodiode, so reading the output voltage we can estimate the total number of incident photons during the entire exposure time.

1.2.2 Pixel scanning circuit

CMOS image sensors use a two dimensional grid of pixel circuits, each of which is composed of one or more photodiodes and a transistor based addressing circuit [3]. In the vast majority of CMOS cameras, a rolling shutter circuit is used to address individual pixels of the grid, while reducing the number of components in order to reduce cost by reducing die area. First the row of pixels to be gathered is selected with a vertical scan circuit, next the individual column is selected with a horizontal scan circuit, as seen in Figure 1.2.

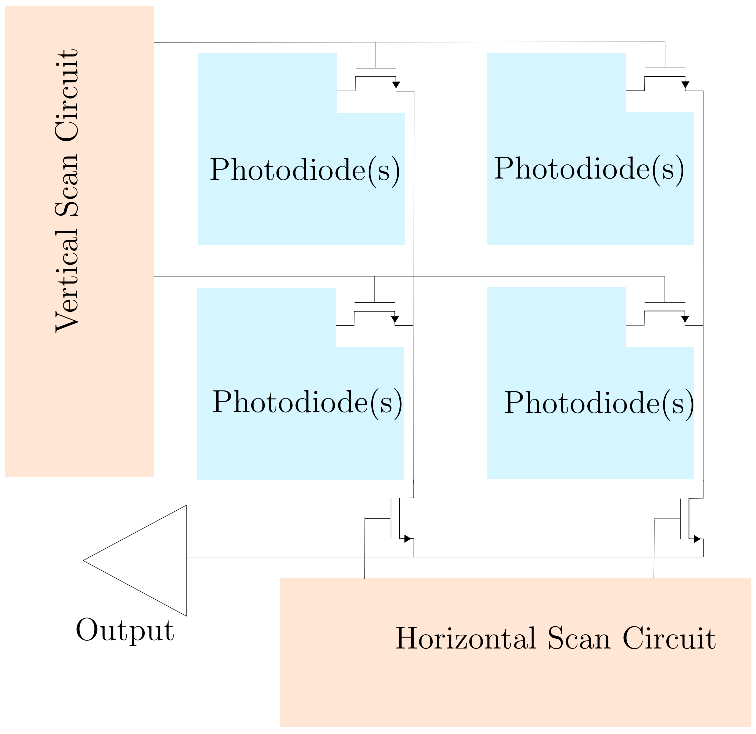


Figure 1.2: The architecture of the scanning circuit used to read individual pixels in a rolling shutter CMOS image sensors. The circuit shown has only four pixels, in reality there are hundreds of thousands or millions of pixels.

1.3 Thesis Organization

A brief background section first outlines the operation of CMOS image sensors and explains the root cause of rolling shutter distortions. Apart from the operation of CMOS sensors, Section 2 also summarizes some existing solutions to rolling shutter distortion, including conventional full frame video stabilization, electronic global shutters, gyroscope based solutions and purely software rolling shutter distortion correction algorithms. Section 3 outlines the algorithm design, including assumptions made, feature pair extraction, the distortion transfer function, the outlier robust fitting method and the method to filter camera motion. In Section 4, the implementation and parallelization method are presented. Several concepts presented here drastically improve performance, but are not directly related to the algorithms or models used. The results of the algorithm are presented in Section

5, including performance results, intermediate value results, and the output video quality improvement for several test videos. This work concludes with Section 6, where several important contributions are detailed, as well as possible future work to improve results.

Chapter 2

Background

The delay in capturing rows of a single frame in a video can cause problems if the scene has changed during the delay. Rotations or translations of the camera itself or a change in the scene during the delay will produce various types of video artifacts that are undesirable. Four major types of rolling shutter video artifacts are described below in Section 2.1. Each of these can be found in both still images and video, though some are more apparent in one or the other.

2.1 Types of distortions

There are four major types of rolling shutter distortions: skew, wobble, smear and partial exposure. In order to describe these artifacts, some notation is required. The row scanning frequency (f_{row}) is the rate at which rows are scanned by the vertical scan circuit, while the frame rate (f_{frame}) is the rate at which frames are recorded. n_{rows} and n_{cols} are the number of rows and columns in a frame respectively. It is important to note that the exposure time is often adaptive to lighting conditions, so in general $f_{row} \neq f_{frame}\ n_{rows}$, but we assume that the number of rows is fairly large, so $f_{row} \gg f_{frame}$. Finally, the frequency of distortion

$(f_{distort})$ is the highest frequency component of a frequency domain representation of the motion of the camera relative to the scene. For each distortion description, the assumption is made that pixel rows are scanned horitonally. If a camera is held such that rows are scanned vertically, the effects will appear rotated from how they are described below.

2.1.1 Skew

Skew occurs when the subject of a video moves relative to the viewpoint of the camera. Skew is considered to be a linear motion, so we can say that the $f_{distort}$ $\ll$ f_{frame}. In the simple case of a camera panning horizontally while capturing a frame, the last rows to be captured will be offset from the first rows in the direction of motion. This effect is illustrated in Figure 2.1. If the camera pans in the vertical direction during the capture of a frame, the image may be vertically compressed or stretched. It is worth noting that skew will produce a shear transformation – any parallel lines will be parallel in the distorted frame.

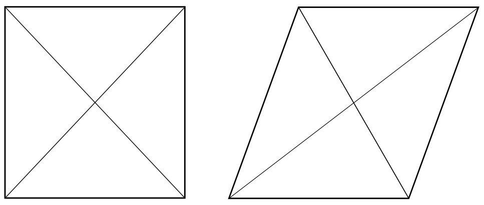


FIGURE 2.1: Skew effect of a camera panning horizontally to the left while capturing a frame.

2.1.2 Wobble

Wobble occurs when there is a high acceleration of either the camera or the scene, and the resulting $f_{distort} > f_{frame}$. This is a particularly large problem in situations where a small camera is non–stationary, such as one held by a person or mounted on a vehicle [6]. Straight lines in a scene will occur curved, and different parts of a single frame can be either compressed or stretched in the vertical direction. In cases where the camera is rotating, different sides of the same row and be compressed and stretched at the same time. Wobble distortions tend to be fairly small and as such are often difficult to notice in a still image, but the oscillations produced can be easily seen in video. This effect is illustrated in Figure 2.2, where an image is distorted by a camera rapidly accelerating horizontally.

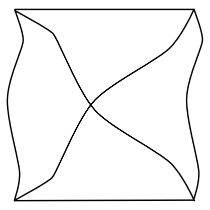


FIGURE 2.2: Wobble effect of a camera rapidly accelerating back and forth horizontally during exposure.

2.1.3 Smear

Smear is a higher frequency artifact than wobble, and is essentially an aliasing effect occuring between rows. This means that $f_{distort} \geq \frac{f_{row}}{2}$, and in the most obvious examples $f_{distort} \geq f_{row}$. When this occurs, portions of objects can appear to be floating and disconnected from the rest of the object itself. Smear is usually seen when an object in the scene is moving at high speed, rather than the camera itself. Since it can be difficult to illustrate a 'typical' example of smear, and example photograph is shown in Figure 2.3.

2.1.4 Partial Exposure

Partial exposure occurs when the scene being recorded changes drastically during the frame scan [7]. This will result in a two or more distinct portions of the image

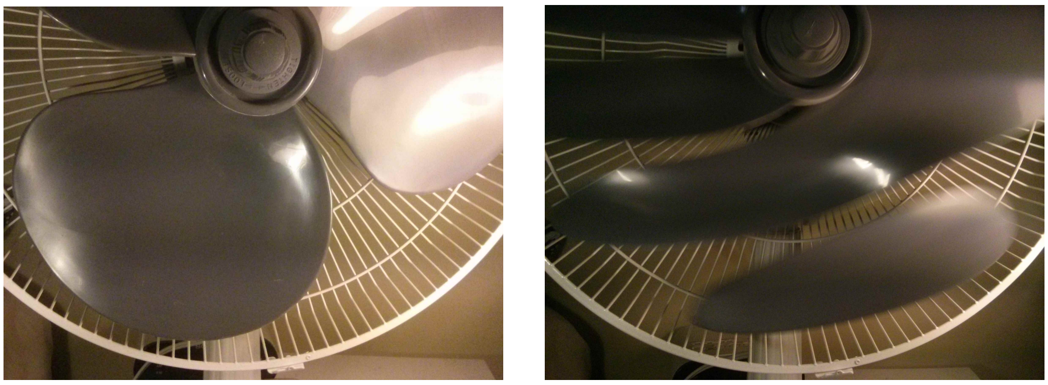

(A) Stationary Fan (B) Moving Fan

FIGURE 2.3: An example of frame smearing. Figure 2.3a shows the fan blades stationary, while Figure 2.3b shows a floating portion of a fan blade caused by the aliasing smear effect of a rolling shutter camera. This was captured as a photograph (not a video) using the camera in the LG Nexus 4 cellphone, with 8 megapixel resolution and an exposure time of 1/763 s.

that look strange when combined. An example of this would be having a flash go off mid frame scan, resulting in a bright portion and dark portion of the image.

2.1.5 Scope of Work

The focus of this work is to correct rolling shutter wobble in videos, using an approach that should help to correct skew as well. Smear violates the Nyquist–Shannon sampling theorem, which would essentially make it impossible to accurately construct the motion of the distortion without prior knowledge or large assumptions, and therefore difficult to reliably fix artifacts. Partial exposure will not be considered in this work.

A basic full frame camera stabilization will also be implemented, as it is necessary to stabilize the full frame in order to accurately calculate the motion of the camera as it moves between row captures in order to accurately capture the smaller rolling shutter distortions.

2.1.6 Importance of Work

Cameras are being found in more and more devices, many of which are very susceptible to large camera movements that can degrade video quality. Examples include cellphones, aerial drones, wearable devices such as Google Glass, and cars. Each of these will often employ a rolling shutter CMOS camera to reduce cost or power consumption compared to alternatives.

These devices are limited in size, requiring a low power solution that can correct rolling shutter artifacts without consuming too much energy and producing too much heat for a small, contained processor. Commercial image processing systems exist to provide real time image processing in small, contained units that can be one cubic inch or less in size[8]. An algorithm that can be run in parallel can be helpful in reducing power requirements, as several processor cores can be run simultaneously at a lower frequency than a single core while performing the same calculations.

2.2 Previous work

Several types of video stabilization have been developed, for both stabilization of the entire video frame and removal of rolling shutter artifacts. Systems involving both hardware and software systems are discussed below.

2.2.1 Full frame video stabilization

Full frame stabilization is intended to remove large shaking from the entire video frame, and is applicable to cameras with and without a rolling shutter image sensor. In many cases, hardware systems designed to provide full frame video stabilization will mitigate rolling shutter effects as well, since large camera accelerations are filtered from the camera path. They also help with issues such as parallax that cannot be corrected using a global image transform for each frame.

Fully mechanical systems for video stabilization include devices such as tripods, camera dollies and steadycams. These devices are typically large, expensive, and difficult to use, making them impractical for small consumer cameras, such as those found in most cell phones.

Hydrid mechanical– electrical stabilization systems, such as the STABiLGO have been developed more recently [9]. A gyroscope is mounted on a small camera, connected to motors that are able to control the orientation of the camera. By monitoring the angular velocity, high speed camera shakes can be detected and counteracted using the motors.

Many software–only solutions to full frame video stabilization have been developed, and most employ the same three major steps: camera motion estimation, motion smoothing and image warping [10].

Camera motion estimation is typically done in one of two ways: a feature based approach where “interesting” points in the image are tracked from frame to frame, or global pixel alignment methods where entire images are matched together [10]. Most modern stabilization algorithms use a feature based approach to calculate video motion, as they tend to run much faster than global alignment methods [11]. All work described in this section uses a feature based approach to calculate the overall video motion.

Once camera motion estimation has been performed, a smoothed output motion must be calculated. This smoothed motion should be free from large accelerations; any camera motions should be as smooth as possible.

Early methods computed integrals of camera motion over the past 10-15 frames in the horizontal and vertical directions, and used this cumulative motion as the camera path [12].

Litvin *et al.* used a Kalman filter to fully model the camera motion and zoom, under the assumption of a static planar scene [13]. This motion model allows superior shake rejection compared to previous translation– only models.

Bezier curves have been used to smoothly estimate true camera motion while removing large accelerations [14]. In this work a single Bezier curve is fit to an entire video sequence, though the order of the curve can be set. Long videos are split into sub-videos, with a section of overlap to provide continuity. For each video, an optimization process iteratively updates a Bezier curve to minimize an objective function that is the weighted sum of three terms: the difference between feature positions in successive frames, the absolute value of the acceleration of the output motion, and the difference between the four corners of each pair of successive frames (this last term is intended as a stability constraint when few or no features are available in a frame).

An L1 optimized path algorithm was developed by Grundmann *et al.* in 2011 to attempt to further smooth camera motions [11]. One distinction between their work and others' is that the crop region is predefined, and the motion path is set according to the size of the crop. Output camera motion is to be divided into three distinct types of path segments: a static non-moving camera, segments of constant velocity, and segments of constant acceleration. L1 optimization is used to force the camera path to follow one of those three path types exactly, rather than following the path on average if L2 optimization were used. L1 optimization attempts to minimize the absolute value of the difference between camera movements between subsequent frames, while L2 will minimize the sum of the square of camera movements. The objective function is the weighted sum of first, second, and third derivatives of output camera motion through all frames. Linear programming is used to perform this optimization, while constrained by the size of the crop window. An affine transform is used to model camera motion, allowing the algorithm to correct image skew. By adjusting the weight of first, second or third derivatives different types of camera motion can be achieved (if greater weight is given to the first derivative for example, camera motion will be perfectly still for parts of the video, then move quickly). Saliency constraints were also implemented, allowing certain parts of the video to be deemed more important, making those object(s) more likely to appear within the crop region.

Kim *et al.* proposed an algorithm designed to provide video stabilization for CMOS image sensors in 2011 [15]. This approach uses a seven degree of freedom camera model to estimate motion from frame to frame. The exposure time of the camera is calculated, and linear interpolation is used to calculate a transformation specific to each row of pixels. LM optimization was used to estimate model parameters. This approach was shown to be effective in reducing skew in the output video, but is not able to correct wobble.

2.2.2 Electronic global shutters

As mentioned in Section 1.2.1, a global shutter can be created in hardware on a CMOS sensor. This is essentially a sample and hold circuit for each pixel, allowing all pixels to be exposed to light at the same time. A possible implementation of a hardware global shutter pixel circuit is shown in Figure 2.4.

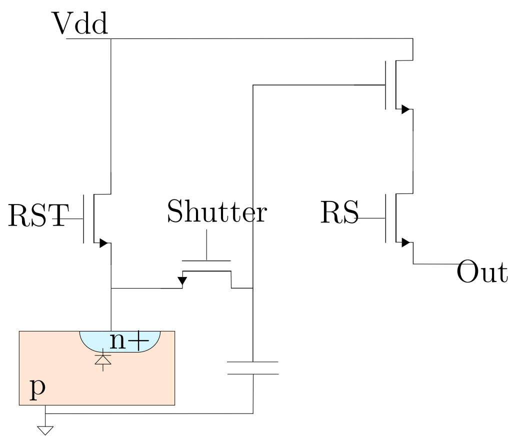


FIGURE 2.4: A pixel circuit for an electronic global shutter [2]. The shutter line for all pixels is a connected signal that will cause all pixels to hold their current voltage levels. At this point each sample and hold circuit can be sampled using regular rolling shutter readout circuitry. The shutter MOSFET can be replaced with a CMOS transmission gate for improved performance at the expense of added area.

2.2.3 Gyroscope based software solutions

Gyroscopes have been used to measure camera rotations while filming instead of using video itself to estimate camera motion [16]. Karpenko *et al.* used the gyroscope inside of an iPhone 4 to measure the rotation of the camera while recording videos, and used these rotations to correct rolling shutter artifacts in real time. Translational camera shakes were not considered in their work, as accelerometer data needs to be integrated twice to calculate the translation. This double integration yields a large amount of error, and correction of translational shakes is more difficult due to parallax effects. A one time calibration video analysis using extracted feature points is used to determine the delay between each line.

Hanning *et al.* developed an algorithm similar to that of Karpenko et al [17]. Their work uses the accelerometer on the iPhone as well as the gyroscope, using an extended Kalman filter (EKF) to perform sensor fusion. The authors have published an iPhone App called DollyCam using this algorithm.

2.2.4 Software rolling shutter solutions

Several purely software algorithms have been developed to correct rolling shutter artifacts, without the need for a gyroscope or a global shutter. These attributes make these methods far more convenient, as they can be applied to any previously recorded video without special equipment.

Liang *et al.* estimate global translations from frame to frame, and interpolate motion between frames to compensate for the rolling shutter effect [18]. Velocity interpolation between frames is done using Bezier curves. This algorithm has two main limitations: distortions occurring at frequencies higher than the frame rate cannot be corrected, and the time delay between pixel rows must be known *a priori*.

Baker *et al.* calculate the overall translation from frame to frame, but use additional information to calculate transformations for individual rows [6]. Images are broken into a grid of sub-images and an affine transformation is calculated for each sub-image. Transformations are optimized to simultaneously minimize the optical flow error for each sub-image and the difference in transformations between adjacent blocks, while requiring that the integral of all transforms are equal to the overall motion from frame to frame. This approach requires the time delay between pixels rows to be known, and is computationally intensive (∼100 seconds per frame on a 2.0Ghz Dual-Core laptop).

Liu *et al.* take a different approach towards rolling shutter stabilization, where feature are tracked over 50 frames, and placed into a sparse matrix of feature trajectories. This matrix is factorized into two low rank matrices: a coefficient matrix and an eigen-trajectories matrix. The coefficient matrix allows a combination of eigen-trajectories to be combined to calculate the transform at a particular pixel location, and the eigen-trajectories matrix is a low rank matrix representing the trajectory of objects within the video. The eigen-trajectories are smoothed in one of three ways: simple low-pass filtering, polynomial path fitting (up to a quadratic fit), or spline fitting. By multiplying the smoothed eigen-trajectories matrix by the coefficients matrix, an ideal path for any location in the image can be calculated. The input video is then smoothed using content preserving warps developed by Liu *et al.* in 2009 [19]. This method does not require the time delay between pixel rows to be known.

Grundmann *et al.* developed a stabilization technique using a mixture model of homographies to model rolling shutter distortions [1]. An adaptive use of the Kanade-Lucas-Tomasi (KLT) feature tracker is used, where the image is divided into 4x4 sub-images. This helps to distribute feature points more evenly, providing distortion information in regions of low contrast. Outlier rejections is performed in each sub-image individually by first estimating a translation for that sub-image using random sample consensus (RANSAC), and rejecting points that deviate from that translation by more than two pixels. Inlier features from all sub-images are then combined.

Ten affine transformations are calculated centered around regularly spaced pixel rows by weighting the feature points according to a gaussian weight of the scan-line to feature point distance. This smoothly interpolates the transformations across scanlines. Singular value decomposition (SVD) is used to calculate the transformation parameters. SVD is a method of factoring a matrix from which rotation and scaling values can be extracted. To ensure robustness, a regularizer of $\lambda||h_k - h_{k-1}||$ is used, where h is a vector of model parameters, k denotes the affine transform number, and λ is a constant value chosen to be 1.5. After the initial fit of model parameters, the regularizer is introduced and model parameters are optimized using iterative least squares. Smoothed full frame camera motion is calculated using an L1 optimized path algorithm developed by Grundmann *et al.*, described here in Section 2.2.1. An implementation of this algorithm is used by YouTube.

2.3 Summary

Several methods have been developed to stabilize video, either through hardware, software or a combination of the two. Hardware and hybrid methods have been successful, but cannot be applied post-recording and require hardware that can be bulky, inconvenient and expensive compared to a simple CMOS camera. Software methods have been proposed for full frame video stabilization, and rolling shutter stabilization. Many rolling shutter stabilization systems require calibration of the camera, which is especially difficult to do in changing light conditions where exposure times are changing. These algorithms have a large number of degrees of freedom, which can make optimal parameter fitting very difficult and prone to errors, especially when objects are moving within the scene.

An algorithm is proposed which will estimate and correct rolling shutter distortions from video. The proposed algorithm is intended to be efficient and easily run in parallel. A sinusoid based distortion model is able to remove distortions, while being robust to outliers caused by movement within the scene or feature mismatch. This model is especially powerful on vehicle mounted cameras, where camera movement tends to follow from engine vibrations. An iterative m-estimate based approach to outlier rejection is used, with an increasingly aggressive outlier criteria that allows the global minimum error to be found reliably and quickly.

Chapter 3

Algorithm Design

In this chapter, the algorithm design of the rolling shutter distortion detection and correction system is explored. Assumptions made in the design process will be explained and justified in Section 3.1.

The algorithm design can be broken down into several high level components. Features are extracted in each frame, and their location in the next frame is calculated. The extraction of feature pairs is discussed in Section 3.2. A distortion model that attempts to capture the distortions between each pair of frames is explained in Section 3.3. Outlier rejection is a large consideration in attempting to obtain reliable model parameters, and is discussed in Section 3.4. Motion smoothing and the integration of rolling shutter distortion is presented in Section 3.5. Finally, image warping to remove distortions is described in Section 3.6.

Algorithm development was programmed using of C++ using G++ 4.2 to compile, and OpenCV 4.2. Python 2.7.3 was also used for development, especially testing and visualization.

3.1 Assumptions

Several assumptions were made in order to simplify algorithm design. These include the orientation and type of movements of the camera, as well as the content of the scene being recorded.

The camera is assumed to be oriented in such as way that the image is scanned row by row (each row being horizontal). Many cameras will automatically rotate a video that was filmed in the portrait direction. The distortion transfer function assumes that each video will be scanned row by row, so any video scanned column by column should be rotated by 90 degrees before processing, then rotated back.

Camera motion is assumed to be purely rotational. This is frequently violated. The algorithm is robust to some translational camera motion, but parallax effects are not taken into account, and can cause errors in the calculation of the transfer function. If the camera is translating, all objects in the scene are considered to be in one plane, or equivalently, infinitely far away.

The majority of the scene being filmed is assumed to be static. Features detected on moving objects will be ignored as long as the majority of the scene is static.

The motion of the camera is assumed to be slow enough that that majority of successive frames will overlap. This is required so that features can be matched between one frame and the next. The lighting conditions are assumed to be slowly changing or constant with respect to the frame rate.

Motion of both the camera and any objects within the frame should be sufficiently slow that it can be properly captured by scanning at f_{row}. This means that no smear should be present in any videos.

Many assumptions made here are general, and frequently violated. While the distortion model is not able to incorporate these phenomenon, the robust optimization technique used will often consider these effects to be outliers, and ignore any extracted features where these effects are present.

3.2 Extracting feature pairs

The first step in calculating a distortion transfer function is to collect data that will be used to map locations in one frame to the next. This can be done using many different algorithms, but most ultimately return a list of features found in the first frame, and a second list of the location of those same features in the second frame.

Several methods are commonly used to find these ideal features. In this work, the KLT feature tracker is used. The operation of the KLT feature tracker is discussed in Section 3.2.1, while some alternatives are discussed in Section 3.2.2. Section 3.2.3 explains how the KLT feature tracker can be modified to return features that more evenly cover frames, simplifying the model fitting process.

3.2.1 KLT feature extractor

The KLT tracker can be broken down in three components: initial extraction of features to track in a single frame, an optional step to improve feature locations to the sub-pixel level, and finally the optical flow algorithm to find the location of the features in the second image.

Each of these steps is typically done using a grayscale version of the two frames. Grayscale conversion was performed using the OpenCV 2.4.8 function `cvtColor` with `code` set to `BGR2GRAY`. This function turns each pixel value from color to a grayscale value according to Equation 3.1

$$Y = 0.299R + 0.587G + 0.114B, \tag{3.1}$$

where Y is the grayscale pixel value, and R G and B are the red, green and blue pixel channels respectively.

3.2.1.1 Extracting features

Useful features to track from frame to frame can be extracted from images using a variety of criteria, such as brightness and color, edges, or corners [20]. Kanade and Thomasi proposed the idea that the quality of features should not be determined by any predefined criteria, but rather by the ability of an algorithm to track these features from frame to frame [21]. Their work led them to an algorithm that uses the minimum eigenvalue of the covariance matrix of the image at small blocks as a measure of the pixel's tracking potential. The pixels are then sorted, and those with a sufficiently large minimum eigenvalue are used for tracking. This algorithm is described in detail below. An OpenCV 2.4.8 implementation of their work was used by calling the function `goodFeaturesToTrack`.

To explain the operation of the corner extraction process, we must first define some variables that will be used. Let I represent the image from which features will be extracted, so that $I(x, y)$ represents a single pixel location in the image.

The first step in extracting features is to calculate derivatives of the image at each pixel location in both the x and y directions. This is calculated using the Sobel operator, which will both calculate a derivative of discrete data and smooth out noise by averaging in the direction orthogonal to the differentiation. The size of the Sobel kernel was specified to be three pixels, producing the image derivatives

G_x and G_y shown in Equations 3.2 and 3.3

$$G_x = \frac{\partial I}{\partial x} = I * \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \tag{3.2}$$

$$G_y = \frac{\partial I}{\partial y} = I * \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}. \tag{3.3}$$

For each pixel in the image, a sub-block of 3×3 adjacent pixels was extracted with the desired pixel in the center. The covariance matrix of each sub-block was calculated, as shown in Equation 3.4 [22]

$$M_{cov}(x,y) = \begin{bmatrix} \sum_{i_x=x-1}^{x+1} \sum_{i_y=y-1}^{y+1} G_x^2(i_x,i_y) & \sum_{i_x=x-1}^{x+1} \sum_{i_y=y-1}^{y+1} G_x(i_x,i_y)G_y(i_x,i_y) \\ \sum_{i_x=x-1}^{x+1} \sum_{i_y=y-1}^{y+1} G_x(i_x,i_y)G_y(i_x,i_y) & \sum_{i_x=x-1}^{x+1} \sum_{i_y=y-1}^{y+1} G_y^2(i_x,i_y) \end{bmatrix}. \tag{3.4}$$

The eigenvalues λ_1 and λ_2 of the covariance matrix are calculated. In order to be easily tracked, the eigenvalues of M_{cov} should be much greater than the noise level (the average eigenvalues of difficult to track pixels), and the matrix should be well conditioned [22]. If both eigenvalues are large, the corner contained within the sub image is distinct, and if both eigenvalues are of similar magnitude then the matrix is well conditioned. By storing the minimum of the two eigenvalues we guarantee that the corner is distinct, and in practical cases the matrix M_{cov} tends to be well conditioned when the smaller eigenvalue is sufficiently large. A matrix $M_{min(\lambda)}$ is constructed of the minimum eigenvalue of the covariance matrix of each sub image.

To prevent many features from being too close together, only the local maximum eigenvalue is taken from each 3×3 neighborhood of $M_{min(\lambda)}$, others are discarded.

To discard feature points with weak corners, any element in $M_{min(\lambda)}$ with a value less than $max(M_{min(\lambda)})$ $qualityLevel$ are discarded. The value of $qualityLevel$ was chosen to be default value used by OpenCV of 0.02. Neither increasing nor decreasing this value had a significant effect on the number of extracted features in any analyzed video.

All remaining potential feature points are sorted by eigenvalue in descending order. The top $n_{features}$ points are kept, and others discarded. As a final check, if any points fall within minDistance of each other, the one with the smaller eigenvalue will be discarded.

It is worth noting that the majority of this process can easily be implemented into hardware that can be run in parallel. The derivative calculation runs in $\mathrm{O}\big(n_{pixels}\big)$, where n_{pixels} is the number of pixels in the image. Each calculation can be run simultaneously in either an application specific integrated circuit (ASIC) or Field Programmable Gate Array (FPGA), as seen in [23]. Next is the calculation of minimum eigenvalues of each window, which can once again easily be implemented in hardware due to relatively low complexity. The sorting algorithm used to order pixel locations by minimum algorithm runs in $\mathrm{O}\big(n_{pixels}\log\left(n_{pixels}\right)\big)$ time, and is more difficult to implement in hardware. A modification intended to provide more uniform feature coverage introduced in Section 3.2.3 would decrease this complexity and allow for some parallelization.

3.2.1.2 KLT feature tracking

What is now known as the KLT feature tracking algorithm was originally proposed by Lucas and Kanade as a method to match feature for depth estimation of stereoscopic images in 1981 [24]. It was later refined and a more efficient pyramidal algorithm was developed. The KLT algorithm works by finding an optimal translation for each feature point that will minimize the image difference within a small window surrounding the point.

Within this section, w_x and w_y will be used to denote x and y values that fall within a small window surrounding the point (x and y), the location of the feature to be tracked. This window is also commonly called the integration window, as we will be summing pixel values through this entire region [20]. I will continue to denote the first image (from which the feature point was extracted), and J will denote the second image. Our goal is to find an optimal d_x and d_y that will represent the x and y components of a translation mapping the feature point from image I to image J.

The error at a translation value is defined as the sum of the square of the difference of all pixels in the integration window surrounding the points, as seen in Equation 3.5

$$e(d_x, d_y) = \sum_{w_x} \sum_{w_y} (I(x, y) - J(x + d_x, y + d_y))^2. \tag{3.5}$$

A naive implementation of feature tracking could be to simply run an optimization routine to minimize $e(d_x, d_y)$ for each feature point individually. However, it is assumed that many points will be moving in a similar trajectory due to camera motion. Therefore, a pyramidal approach is taken to more efficiently calculate the motion of multiple feature points. Each level of the pyramid, denoted I^L will be an image sub-sampled from the layer beneath it by a factor of 2. The number of layers, n_{layers} was chosen to be five (though in low resolution videos it will automatically decrease). The sub-sampling low pass filter used to calculate each level of the pyramid is recursive, and shown in Equation 3.6 [20]

$$
\begin{aligned}
I^L(x,y) = {} & \frac{1}{4} I^{L-1}(2x,2y) \\
& + \frac{1}{8}\Big(I^{L-1}(2x-1,2y) + I^{L-1}(2x+1,2y) \\
& + I^{L-1}(2x,2y-1) + I^{L-1}(2x-1,2y+1)\Big) \\
& + \frac{1}{16}\Big(I^{L-1}(2x-1,2y+1) + I^{L-1}(2x+1,2y+1) \\
& + I^{L-1}(2x-1,2y-1) + I^{L-1}(2x+1,2y-1)\Big).
\end{aligned}
\tag{3.6}
$$

Using this approach, we can start at the highest layer of the pyramid and find the optimal translation at that resolution for each feature point. Then the previous translation can be multiplied by two, and used as a starting point for the optimization routine at the next level down. This algorithm allows for efficient computation of many points that could each shift by a large amount. The optimization routine used at each level is discussed next.

The core optimization routine used is essentially using the Newton-Raphson method to iteratively minimize Equation 3.5. Using the pyramid based tracking system, we can assume that the displacement at any given level is small, and therefore assume that the image gradient is linear about the center of the integration window. The use of the Newton-Raphson method is also a motivation behind the need for a non-singular M_{cov} matrix that was described in Section 3.2.1.1. Full details of the optimization derivation can be found in [20].

Many of the assumptions made by the KLT tracking algorithm can be violated, which can result in a variety of problems. The assumption that the image gradient is quadratic is violated nearly always to varying degrees. This can cause the Newton Raphson method to become unstable. Features can be lost to the boundaries of an image, and can be either declared "lost" or matched to an incorrect translation. Any problems in the optimization routine will result in incorrect, outlier features that need to be ignored when calculating camera motion.

3.2.1.3 Sub-pixel resolution of features

Sub-pixel accuracy is necessary to track features, both during feature extraction and feature tracking [20].

During feature tracking, bilinear interpolation is used to determine $I^L(x, y)$ when x and y are non-integers. With this simple modification, the iterative optimization routine can determine the location of feature beyond integer accuracy at each pyramid level.

Sub-pixel accuracy is gained during feature extraction by iteratively refining the corner locations of each feature using bilinear interpolation as approximate values between pixels. This allows us to approximate the image intensity as a continuous function. Equation 3.7 is minimized using the Newton Raphson method over a window with a half side length of $winSize$. The OpenCV 2.4.8 function `cornerSubPix()` was used with $winSize$ set to 5.

$$Cost = \sum_{i=-winSize}^{winSize} \sum_{j=-winSize}^{winSize} G_x\ i + G_y\ j \tag{3.7}$$

3.2.2 Alternatives to the KLT algorithm

Several alternatives to the KLT algorithm have been developed.

Scale Invariant Feature Tracking (SIFT) was developed by David Lowe to attempt to provide feature extraction that is invariant to image translation, scaling, rotation and partially invariant to more complex image transformations and illumination changes [25]. SIFT keys are extracted from each image where the difference of Gaussian functions is at a maximum or minimum. Keys are matched together using a nearest neighbor search algorithm implemented with a k-d tree algorithm, and basic outlier rejection is performed using a Hough transform hash table that looks for a minimum of three keys that agree.

Speeded-Up Robust Features (SURF) is similar to the SIFT algorithm, but designed to be much faster and more robust [26]. The feature extraction process is similar to the method used in SIFT, except that Gaussian functions are replaced with a box filter. A Hessian matrix of the convolution of the second derivative of the image and a box filter is constructed, and features are extracted where the determinant of the Hessian matrix exceeds a threshold. Images are iteratively smoothed and reduced in size, with new features extracted at each iteration to provide scale-invariant features. The response of Haar wavelet transforms on features is used to provide orientation matching information.

3.2.3 Division of images into sub-images

One significant problem with the features extracted using the basic KLT feature tracker is that feature density is often non-uniform. Many features are found within smaller areas with more texture, leaving large portions of the images without any extracted features, which complicates motion model fitting. This same problem was noted by Grundmann *et al.* in 2011 [1], whose solution was to use an adaptive, local threshold approach to choosing features to track. This was done by dividing the entire frame into a 4x4 grid of sub-images, and extracting features from each sub-image individually. Here we determine an improved number of sub-images to use, and present a recursive algorithm to determine an ideal number of sub-images at runtime. Pseudocode to perform sub-image extraction is shown in Figure 3.1.

To improve feature distribution, we must first define a measure of the quality of feature distribution. The image is divided into a fine grid, and the number of features that fall into each grid is counted. The χ^2 goodness of fit of the number of feature points contained each grid segment is calculated, compared to a perfectly uniform feature distribution. Our goal is to lower the χ^2 value, closer towards the value of 0, which is the result of a perfectly uniform feature distribution.

```
Data: N = grid dimensions
Data: Img = Input Image
Result: L = list <feature point structure>
begin
    for $i \in 0..N$ do
        for $j \in 0..N$ do
            Get the sub-image corners
            left = Img.width * i / N
            right = Img.width * (i+1)/N
            top = Img.height * j / N
            bottom = Img.height * (j+1) / N

            Extract the sub-image from the input image
            SubImg = ExtractSubImage(Img, left, right, top, bottom)

            Extract features from the sub-image
            SubFeatures = OpenCv::goodFeaturesToTrack(SubImg, ...)

            Add SubFeatures to total list
            L += SubFeatures
        end
    end
end
```

FIGURE 3.1: Sub-image feature extraction algorithm pseudocode

The work of Grundmann *et al.* was extended by determining an ideal number of sub-images, denoted $n_{subImages}$, to use. This was done by evaluating the χ^2 value on an average of 10 frames, on several different videos, using a varying number of sub-images. The result of this test is presented in Table 3.1. All tests were run with $n_{features}$ equal to 1000 (though due to the need for an integer number of features per sub-image, this may be rounded down), and averaged over the first 50 frames of each video. The number of bins in the x and y directions for the χ^2 fit were 40 and 25 respectively. The average minimum eigenvalue was also recorded for each test, and is also shown. It can be seen that the average eigenvalue decreases as the number of sub-images increases, meaning the average feature quality has decreased. This means that although the feature distribution will be more uniform, there is a higher probability of mismatched or lost features.

This information is shown graphically in Figure 3.2.

Grid dimensions	Driving Video			Parliament Video			Forest Video		
	χ^2	$\bar{n}_{features}$	$min(\lambda)$	χ^2	$\bar{n}_{features}$	$min(\lambda)$	χ^2	$\bar{n}_{features}$	$min(\lambda)$
1	7837.4	932.9	0.0105	6451.2	1000.0	0.0488	5177.4	1000.0	0.0791
2	4938.3	1000.0	0.0085	6112.4	1000.0	0.0481	4348.9	1000.0	0.0778
4	4635.5	992.0	0.0083	5754.2	985.9	0.0439	3601.6	992.0	0.0727
6	4019.7	971.9	0.0078	4155.0	938.2	0.0396	3147.2	961.4	0.0710
10	3345.6	999.4	0.0067	3172.0	990.7	0.0354	2579.4	993.6	0.0671
15	2528.2	899.3	0.0058	2420.6	899.5	0.0331	2012.4	897.7	0.0651
20	1918.1	799.2	0.0053	1711.9	798.6	0.0311	1599.3	798.6	0.0636

TABLE 3.1: Summary of χ^2 values, counts of extracted features and average minimum eigenvalues for different sub-image grid dimensions. A statistical test is performed to determine how uniformly features are spread throughout the full frame for different numbers of sub-images. Theχ^2 values were calculated by placing the features into a grid of bins of dimension 40x25, compared to a completely uniform feature distribution. The number of degrees of freedom is 1000. The χ^2 values and number of features counts were averaged over the first 50 frames of each video. See Section 5.4 for a description and screenshots of the videos used here.

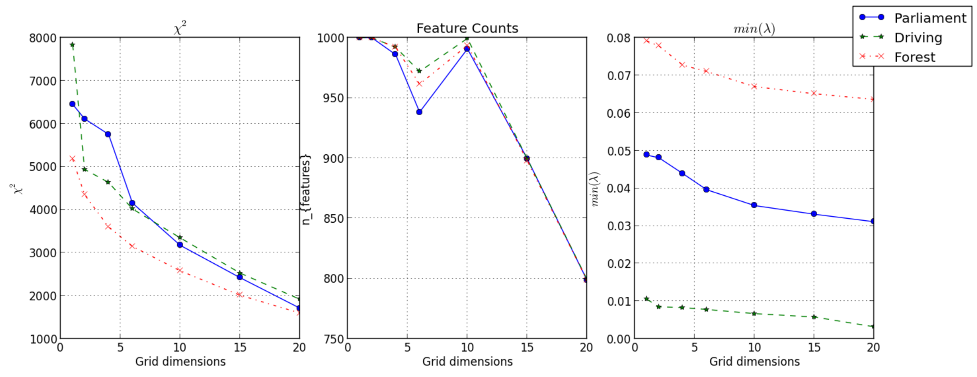
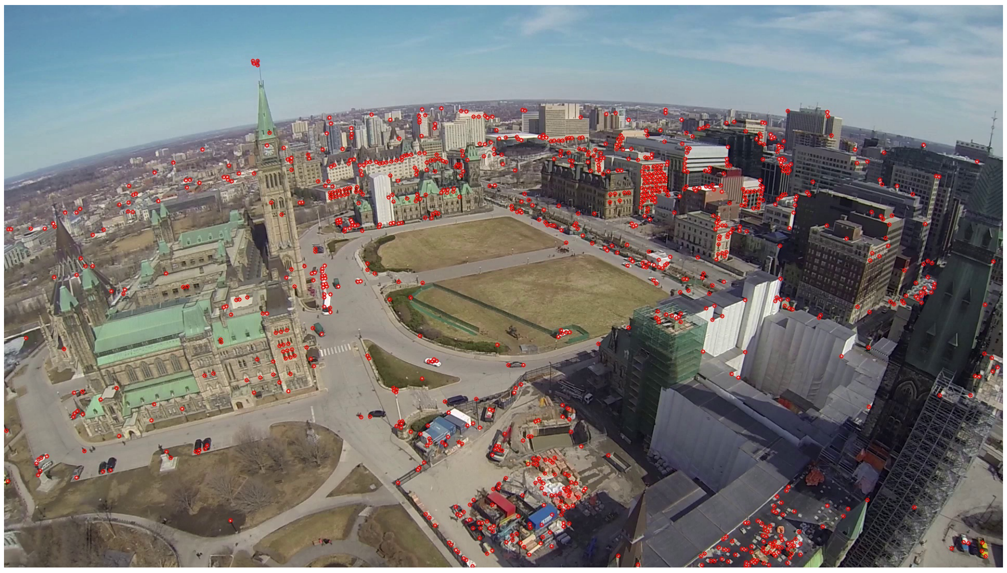


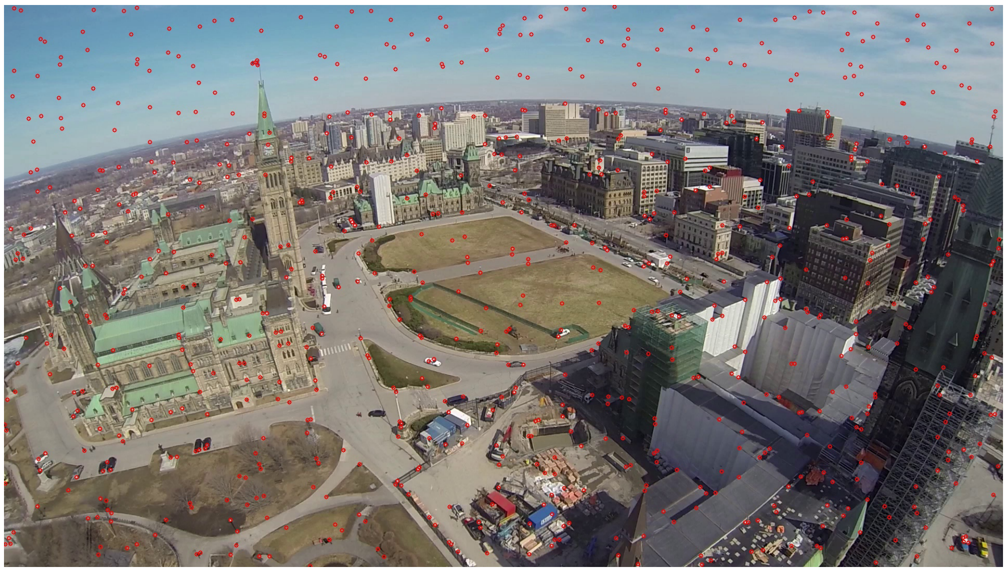
FIGURE 3.2: χ^2 values, counts of extracted features and average minimum eigenvalues for different sub-image grid dimensions. See Table 3.1 for raw data and details on how this data was found.

Based on the information found in Table 3.1 and Figure 3.2, as well as qualitative analysis of the quality of rolling shutter stabilization, an ideal grid dimension of 15x15 was chosen for the tested videos. A large drop in minimum eigenvalue was noticed in the driving scene when moving above 15x15, while the change in χ^2 value was found to decrease as the grid dimensions increase. The difference in extracted features between a 15x15 grid and the standard feature extraction process can be seen in Figure 3.3. In general, videos will strong corners more evenly spread throughout the frames will benefit not benefit as much from the use of sub-images, though the computational complexity is slightly reduced compared to conventional feature extraction. Scenes with difficult to track features (such as rapidly varying lighting conditions) would require fewer sub-images to be used in order to increase the minimum eigenvalues and provide more easily tracked features.

The use of sub-images in feature extraction can be used to speed up the initial feature extraction, as well as parallelize it more easily. The feature extraction is broken down from a single large problem to 225 identical small problem that can easily be solved in parallel. Additionally, each feature extraction requires the sorting of the minimum eigenvalue at all pixel locations, which runs with $O\big(n_{pixels} \log\left(n_{pixels}\right)\big)$ complexity. By dividing into sub-images, we must perform



(A) Standard Procedure



(B) 15x15 Grid

FIGURE 3.3: Comparison of standard feature extraction and grid based extraction. The standard procedure is shown in Figure 3.3a, with a 15x15 grid based feature extraction shown in Figure 3.3b. Note that the standard procedure finds no features in the sky and very dense coverage over some buildings, while the grid version is much more uniform

$n_{subImages}$ sorts of a smaller amount of data. This results in total complexity of $O\big(n_{subImages}\frac{n_{pixels}}{n_{subImages}}\log\big(\frac{n_{pixels}}{n_{subImages}}\big)\big)$, which reduces to $O\big(n_{pixels}\log\big(\frac{n_{pixels}}{n_{subImages}}\big)\big)$.

A recursive algorithm was also developed to determine a suitable grid depth to use at runtime. This algorithm worked by first extracting features for an entire image, then splitting the image into four squares and counting the number of features that appeared in each square. If any quarter had less than 10% of the total features, the entire image would be split into four images, and features extracted from each individually, and running the same check on each of those images. This recursive algorithm continues until one of two conditions is met: each quarter of the image contains more than 10% of all features, or the number of features to be extracted is four or less. Using 1000 features, this results in a maximum recursive depth of five. The results of this algorithm are shown in Table 3.2.

Metric	Driving Video	Parliament Video	Forest Video
χ^2	2770.7	5194.8	4140.2
$n_{features}$	962.8	1000	1000
$min(\lambda)$	0.0071	0.0488	0.0791
$depth_0$	0	50	50
$depth_1$	148	0	0
$depth_2$	43	0	0
$depth_3$	202	0	0
$depth_4$	1832	0	0

Table 3.2: Results of the recursive sub-image feature extraction algorithm. $depth_z$ denotes the number of times (over the entire 50 frame test window) that the recursive algorithm reached a particular depth z. Note that for the parliament and forest videos, the algorithm never descended beyond the top level (full image feature extraction).

The recursive algorithm works well in the case of the driving video, but fails completely on the parliament and forest videos. All metrics aside, in the parliament video no features are extracted from the sky portion of the frame, which ultimately leads to a poorly fit model. This is due to the fact that these regions of low contrast make up less than 1/4 of the screen, so there are still many trackable features in each quarter. In the driving video, the average $min(\lambda)$ and χ^2 values would fall above the curves of the simple grid feature extraction procedure shown in Figure 3.2. However, the feature improvement is small in this case, and worse in others. Additionally, the performance is worse than the grid based extraction since we are performing the eigenvalue sort multiple times for some features. For these two reasons, the recursive feature extraction algorithm was not used.

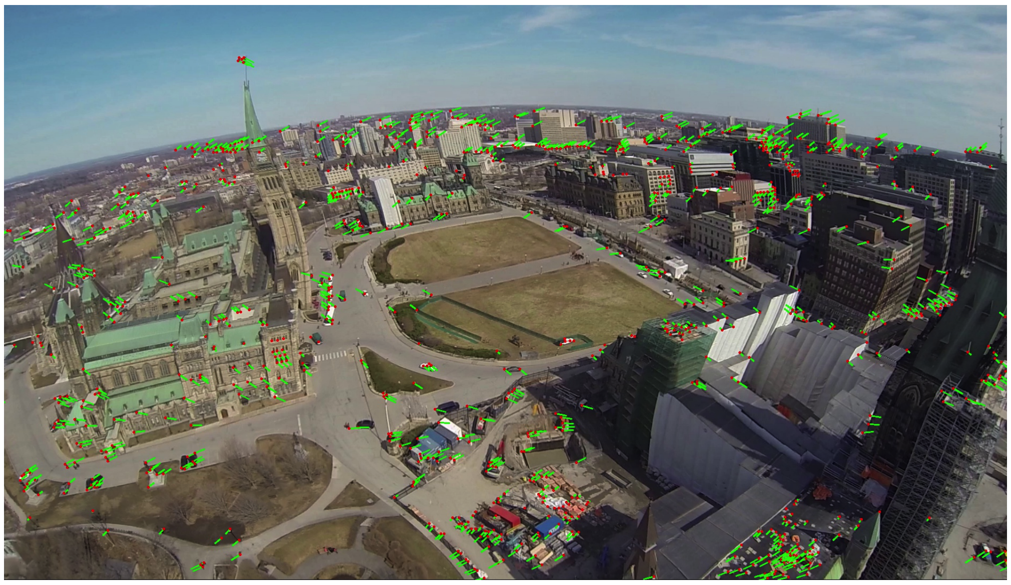


FIGURE 3.4: Visualization of distortion and desired correction. An exaggerated example of a distorted video frame from the parliament video, and lines showing how the image should be corrected.

3.3 Distortion transfer function

Once feature points have been extracted from pairs of frames, we need to define a transfer function that can model camera model and be used to distort frames to remove rolling shutter artifacts. This function should be able to model the motion of the camera as rows are scanned, such that the inverse transformation can be applied to remove distortions, as demonstrated in Figure 3.4.

Previous work in the area used simple translations or affine transformations spread across the images. These are calculated by a weighted sum of feature points, typically weighted with a gaussian of the distance between a feature's y-value and the center of the image transformation. These allow for complex motion to be modeled, but have a large number of degrees of freedom, and require feature points to be spread evenly across rows of each frame for an accurate fit. A new sinusoidal model is presented based on a Fourier series expansion of camera motion during the exposure of rows of a frame. This model is able to capture translations in the x and y directions, as well as rotations that happen within the exposure time of a single frame.

There are two main motivations behind the use of sinusoids to model camera movement: cameras mounted on vehicles have a sinusoidal motion due to spinning engines, and any arbitrary digital signal can be decomposed into a sum of sinusoids [27]. While the decomposition of a signal into sinusoids can theoretically require an infinite number of terms, in practice it was determined that only two terms were required to model the distortions found in the test rolling shutter videos. Qualitatively, we can expect that the number of terms required would be fairly small since time delay is small between frames, and if camera motion was not fairly smooth the video would be blurry, and unusable even if the distortion was perfectly corrected. Additionally, the remaining uncorrected distortion was calculated with a variety of terms, and is shown in Figure 3.5. The majority of the uncorrected errors are due to a noise floor (typically caused by parallax, objects moving within the scene, or interpolation errors introduced in the feature extraction step) and fully removing this would result in severe overfitting when the image distortion is corrected. Adding additional sinusoidal terms scales quadratically with the number of terms, and should be kept to a minimum. Based on Figure 3.5 and a desire to decrease the number of terms to improve performance, two sinusoid terms were used in each direction of camera wobble.

There are four main components of the camera motion model used here. A full frame movement model that encompasses translation and rotation of the full image, as well as a translation and rotation that is a linear function of the row is first described in Section 3.3.1. Two sinusoids are fit to the x-translation of the image, as a function of the row, as described in Section 3.3.2. Section 3.3.3 describes how the model fits the y-translations as a function of the row. Finally, a row based rotation component of the model is described in Section 3.3.4.

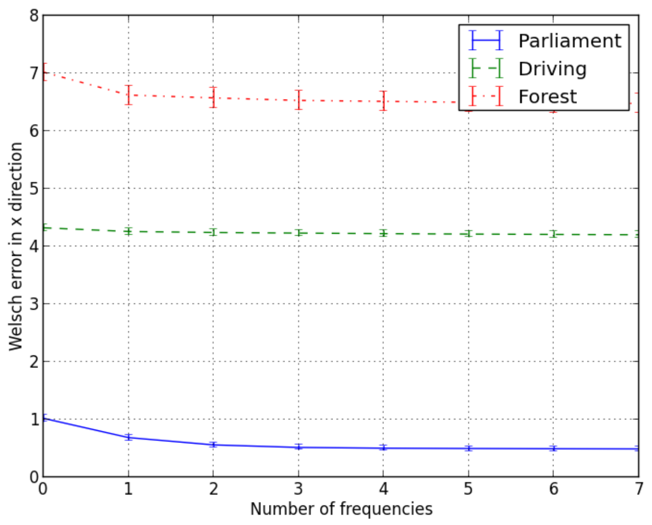


Figure 3.5: Fitting errors with different numbers of sin terms. The errors shown are the squared errors in the x-direction only.

The fitting procedure used to calculate the parameters within this model will not be discussed here. See Section 3.7 for this information.

3.3.1 Full frame movement

There are two main needs for full frame movement modeling: by subtracting overall frame motion we simplify fitting of the more complex rolling shutter artifacts, and this motion can be used to stabilize the overall motion in the output video.

The full frame motion is modeled by a rotational transformation and three additional parameters that represent the translation the in x and y components, as well as the rotation as a linear function of the row. This differs from the affine transformation that is commonly used in several ways:

- The frame skew is only a function of y, and not both x and y like a regular affine transformation. The justification for this is that frames are scanned in the y-direction, so any skew should only be a function of y.

- The size of the frame is kept constant. Camera motion being modeled is assumed to be purely rotational, so we there should not be any change in size of the scene being analyzed.
- We allow for a rotation to be a linear function of the row, allowing for a slow changing rotation to be accurately taken into account.

The equations used to predict the x and y movements from frame to frame are shown in Equations 3.8 and 3.9

$$x_2 = x_1 \cos\left(p_{r0} + y_1 p_{r1}\right) - y_1 \sin\left(p_{r0} + y_1 p_{r1}\right) + p_{dx} + y p_{dx2} \tag{3.8}$$

$$y_2 = x_1 \sin\left(p_{r0} + y_1 p_{r1}\right) + y_1 \cos\left(p_{r0} + y_1 p_{r1}\right) + p_{dy} + y p_{dy2} \tag{3.9}$$

respectively. Any symbol p represents a variable to be optimized. p_{dx} and p_{dy} are the image translations in the x and y directions respectively. p_{dx2} is used to fit horizontal skew where the image is shifted linearly as a frame is captured. p_{dy2} fits the vertical linear expansion or compress of the image caused by accelerating the camera vertically during frame exposure. p_{r0} models the rotation of the frame during the entire exposure, and p_{r1} linearly models the changing rotation of the camera through the exposure of the entire frame.

3.3.2 Row based x-translation

A row based x-translation model is needed for camera rotation about the z-axis at a rate greater that f_{frame}. Figure 3.6 shows an example of camera motion requiring a row based x-translation model to correct the rolling shutter distortion.

Two sinusoids are fit to this shift in x values between frames. The amplitude, frequency and phase of the first curve is fit first, then all parameters of the second

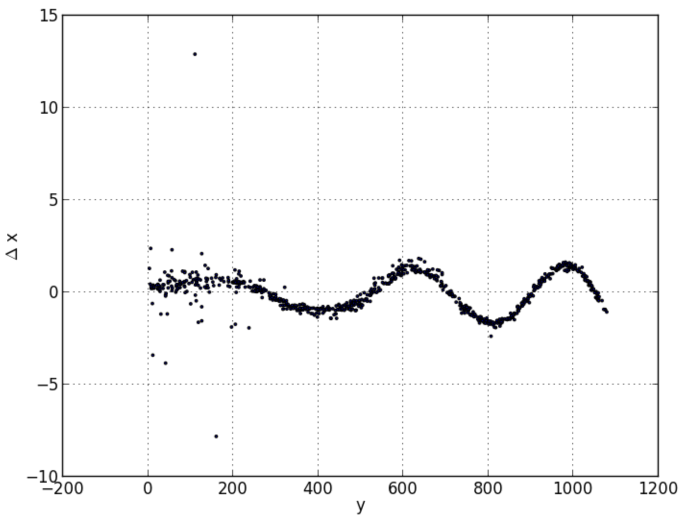


FIGURE 3.6: x-Translation between video frames, measured across all y values. Each data point represents a feature matched between both frames, with y values taken from the first frame. Note that this particular frame was chosen as it contains few outliers, so the curve is easily visible. Note that the frequencies of oscillation shown here are higher than is typical, and was also chosen so that two distinct frequencies could easily be seen.

are calculated. By subtracting the sinusoid once fit, the leftover frame distortion can be visualized, as seen in Figure 3.7.

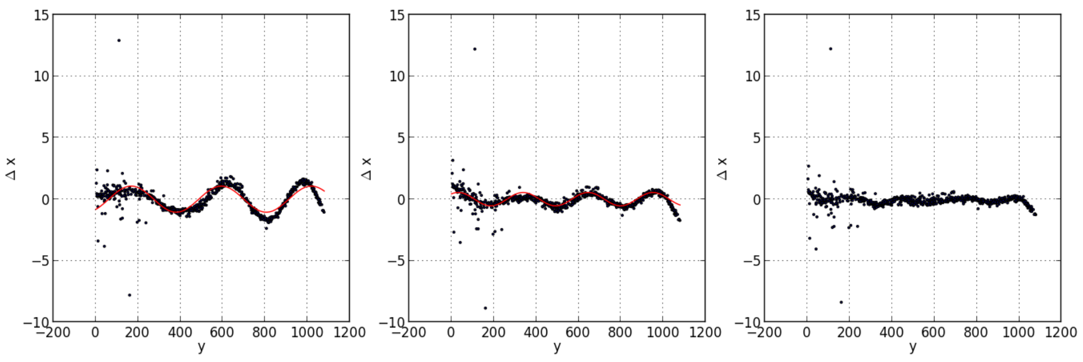


FIGURE 3.7: Fitting of x-translation by sinusoids. On the left, the raw data is seen, with an overlaid line showing the sinusoid calculated to fit the distortions. In the middle, the first sinusoid is subtracted from the data, and another fit to the residual wobble. On the right, the second sinusoid is subtracted. Note that the optimization process will further reduce any leftover distortions by performing a global optimization of both curves, as well as other parameters.

The total row based translation transfer function is shown in Equation 3.10

$$\Delta x = p_{dx2} \sin\left(y_1 p_{dx3} + p_{dx4}\right) + p_{dx5} \sin\left(y_1 p_{dx6} + p_{dx7}\right) \tag{3.10}$$

.

3.3.3 Row based y-translation

The row based y translation is very similar to the row based x-translation. Since we are fitting a shift in y with respect to the y coordinate, the distortion will appear as a compression or decompression of pixel areas, instead of a localized skew. The math used to model this is the same as the x translation, and the transfer function is shown in Equation 3.11

$$\Delta y = p_{dy2} \sin\left(y_1 p_{dy3} + p_{dy4}\right) + p_{dy5} \sin\left(y_1 p_{dy6} + p_{dy7}\right). \tag{3.11}$$

3.3.4 Row based rotation

High frequency rotational wobble of the camera results in y-translational effects that are most apparent near the sides of the frame. This is due to the frame rotating about the center of the screen since we have already removed full frame rotation and rotation as a linear function of y. By plotting the shifts on a 3D plot, the rotation about the center of the screen can be seen in Figure 3.8.

Row based camera rotation could be fit by attempting to reduce the x and y errors through a rotation matrix that is a sinusoidal function of the y value. The assumption is made that the camera rotation will be small from row to row, so the effect on x-values is minimal. This is due to the fact that $\Delta x = a\cos\left(\theta y\right)$, and $\lim_{y\to 0} \frac{\partial}{\partial\theta}\cos\left(\theta y\right) = 0$. As a second simplification, we can consider the x value of

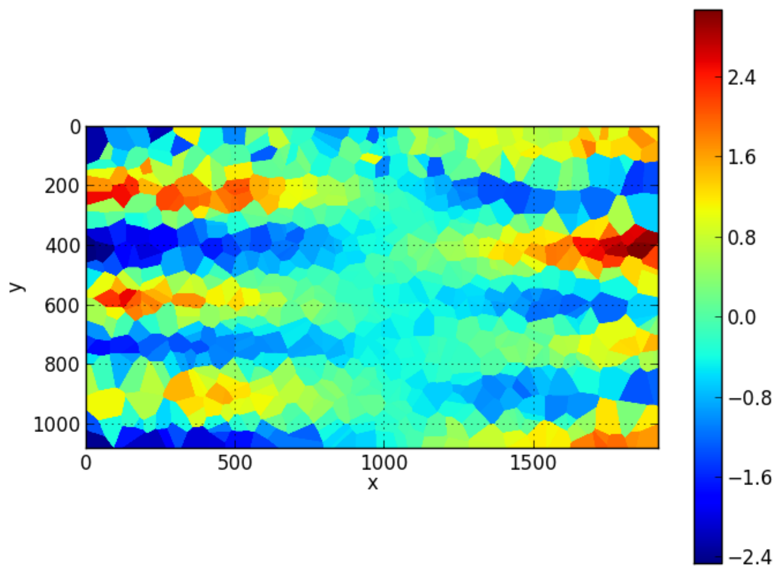


FIGURE 3.8: Y shifts due to high speed oscillating camera rotation. The color represents the shift in y value. Locations are interpolated using the nearest feature match. To improve the visual, the 10% of pixels with the largest shift are removed prior to plotting as a simple outlier rejection method.

each feature to be a feature weight for the sample fitting process used in Sections 3.3.2 and 3.3.3. These weights are normalized by the width of the image, so that parameters found here on of a similar magnitude to parameters for the row based translation parameters to avoid numerical errors. The equation used to model row based rotation is shown in Equation 3.12. The variable `centerX` is used to denote the x value of the middle of the screen, the point about which we allow pixels to rotate

$$\Delta y = p_{r2} \sin\left(y p_{r3} + p_{r4}\right)\left(\frac{x - centerX}{2\ centerX}\right) + p_{r5} \sin\left(y p_{r6} + p_{r7}\right)\left(\frac{x - centerX}{2\ centerX}\right). \tag{3.12}$$

3.3.5 Summary

A model is designed to incorporate four types of camera motion: a full frame transfer function to calculate gross movements, and three transfer functions to remove rolling shutter wobble from translation and rotation using two frequencies each. The total transfer function equations for x and y shifts are shown in Equations 3.13 and 3.14

$$\begin{aligned} x_2 = {} & x_1 \cos(p_{r0} + y_1 p_{r1}) - y_1 \sin(p_{r0} + y_1 p_{r1}) + p_{dx} + y p_{dx2} \\ & + p_{dx2} \sin(y_1 p_{dx3} + p_{dx4}) + p_{dx5} \sin(y_1 p_{dx6} + p_{dx7}) \end{aligned} \tag{3.13}$$

$$\begin{aligned} y_2 = {} & x_1 \sin(p_{r0} + y_1 p_{r1}) + y_1 \cos(p_{r0} + y_1 p_{r1}) + p_{dy} + y p_{dy2} \\ & + p_{dy2} \sin(y_1 p_{dy3} + p_{dy4}) + p_{dy5} \sin(y_1 p_{dy6} + p_{dy7}) \\ & + p_{r2} \sin(y p_{r3} + p_{r4}) \left(\frac{x - centerX}{2 * centerX} \right) \\ & + p_{r5} \sin(y p_{r6} + p_{r7}) \left(\frac{x - centerX}{2 * centerX} \right) \end{aligned} . \tag{3.14}$$

A full list of parameters and their functions can be found in Table 3.3.

3.4 Outlier rejection

Outliers can result from a failure of the feature extraction process, objects moving within the scene or parallax effects, and are a major problem when attempting to accurately fit a model. Additionally, we are dealing with an unsupervised problem since we cannot say for sure whether any individual data point should be considered an inlier or outlier. A method for detecting and ignoring outliers is necessary to accurately compute accurate model parameters.

Several different methods to deal with outlier data points were considered. The basic least squares method to fit data is discussed in Section 3.4.1. RANSAC (and its variants) are considered the gold standard in computer vision [28], and will be

Symbol	Description
p_{dx0}	Full frame x shift
p_{dy0}	Full frame y shift
p_{r0}	Full frame rotation
p_{dx1}	Full frame x skew
p_{dy1}	Full frame y skew
p_{r1}	Full frame linear rotation
p_{dx2}	Amplitude of first x-translation sinusoid
p_{dx3}	Frequency of first x-translation sinusoid
p_{dx4}	Phase of first x-translation sinusoid
p_{dx5}	Amplitude of second x-translation sinusoid
p_{dx6}	Frequency of second x-translation sinusoid
p_{dx7}	Phase of second x-translation sinusoid
p_{dy2}	Amplitude of first y-translation sinusoid
p_{dy3}	Frequency of first y-translation sinusoid
p_{dy4}	Phase of first y-translation sinusoid
p_{dy5}	Amplitude of second y-translation sinusoid
p_{dy6}	Frequency of second y-translation sinusoid
p_{dy7}	Phase of second y-translation sinusoid
p_{r2}	Amplitude of first row based rotation sinusoid
p_{r3}	Frequency of first row based rotation sinusoid
p_{r4}	Phase of row first based rotation sinusoid
p_{r5}	Amplitude of second row based rotation sinusoid
p_{r6}	Frequency of second row based rotation sinusoid
p_{r7}	Phase of second row based rotation sinusoid

TABLE 3.3: Model parameter symbols and descriptions.

discussed in Section 3.4.2. The least median of squares is a statistical technique using medians, and is discussed in Section 3.4.3. M-estimates were ultimately chosen, and are explained in Section 3.4.4. The kernel choice and parameter is also presented, along with a fitting technique to reliably fit a model to the data.

In order to simplify visualization and algebra, data shown and discussed in this section will come from a straight line in two dimensions, following the equation $f(x) = mx + b$, where b is the y-intercept parameter, m is the slope parameter, and x is an independent variable. Everything discussed here can be extended to more complex problems with more dimensions.

3.4.1 Least Squares

Least squares is a simple approach to solving a fitting problem. The algebraic distance squared between the estimated y value and the actual y value is minimized, summed over all data points, as seen in Equation 3.15

$$R^2 = \sum_i (y_i - f(x_i))^2. \tag{3.15}$$

By taking the partial derivative of the residual value R^2 with respect to each parameter and setting to zero, we can often solve a least squares system algebraically. Those that can be solved are called ordinary least squares problems, others that require an iterative solution are called non-linear least squares problems.

Outliers can have a huge effect on a least squares fit since the residuals are squared for each point, so a data point with a large error will have an even larger contribution towards the fit. This makes a least squares solution impractical for use in machine vision problems, where extreme outliers are frequently present. This is seen in Figure 3.9, where a single outlier results in a poor fit.

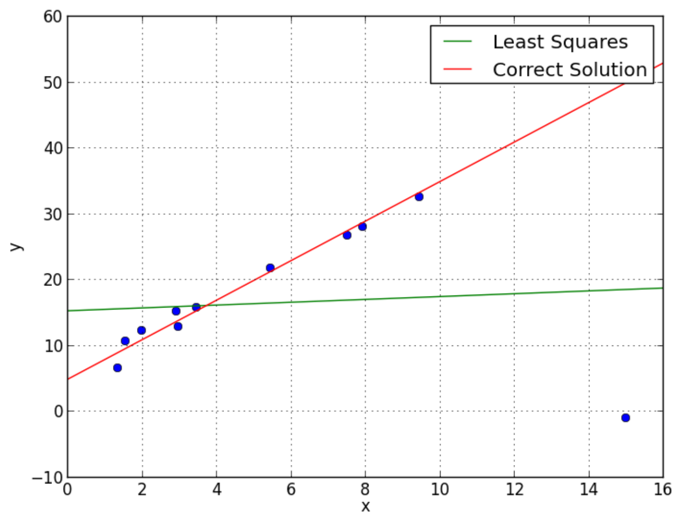


FIGURE 3.9: Failure of least squares fit due to a single (fairly extreme) outlier.

3.4.2 RANSAC & variants

RANSAC was first developed by Fischler and Bolles [29] in 1981 to deal with outliers in computer vision problems. Since then, several variants have been proposed to increase performance or accuracy such as locally optimized RANSAC and Progressive Sample Concensus (PROSAC) [28].

RANSAC uses a minimum number of data points to estimate the true model parameters iteratively, each time with a random sample of data points. The number of points selected each time is equal to the number of degrees of freedom of the model. A score is calculated for each model, typically the proportion of total data points that fall within a maximum distance of the calculated model. This is repeated until we find a model with a high enough score, or we reach a maximum number of iterations. The maximum number of iterations is calculated from the expected fraction of inliers in the data set, as seen in Equation 3.16 [30]. Here u is used to denote the fraction of data points that are expected to be inliers, m is the number of degrees of freedom of the chosen model, and p is the required probability that at least one model is calculated using only inliers

$$N = \frac{\log(1-p)}{\log(1-(1-u)^m)}. \tag{3.16}$$

At the end of all iterations, a local optimization step takes place. The model is recalculated using all data points that are considered inliers to the best model found thus far.

In 2003, Chum, Matas and Kitler introduced Locally optimized RANSAC (LO-RANSAC) to attempt to speed up computation. They observed that RANSAC took longer than expected to find an acceptable model, since it makes the assumption that any combination of inliers will lead to a good model, in essence assuming that inliers do not have any noise [31]. Their solution to this problem is to perform the local optimization step done at the end of typical RANSAC during iterations anytime a new best model is found. They found this modification decreased running time by greater than 50%.

The PROSAC algorithm attempts to sample data points in such a way that there is a greater than random chance of picking inliers. A small sample set is initially constructed by looking at the similarity between data points. Samples are picked from this sample set and models are constructed similar to regular RANSAC. This is repeated while constantly adding more data into the sample set until an acceptable model is found, following ordinary RANSAC criteria. This heuristic was found to speed up computation by more than one hundred times in some cases [32]. In some cases it will fall back to a normal RANSAC sampling method.

The basic RANSAC algorithm can be rewritten as an optimization problem where we attempt to find the minimum of the problem shown in Equations 3.17 and 3.18

$$Cost = \sum_i \rho(y_i - f(x_i)) \tag{3.17}$$

$$\rho = \begin{cases} 0 & \text{if } \epsilon_i < \delta \\ 1 & \text{otherwise} \end{cases}, \tag{3.18}$$

where ϵ_i is the error associated with point i, and δ is a threshold on the maximum distance between an inlier and the calculated model [33]. This formulation leads to the use of m-estimators, described in Section 3.4.4.

3.4.3 Least Median of Squares

The least median of squares approach attempts to minimize the median residual value [34]. Alternatively, the mean of several values surrounding the median can also be used. This approach is very robust, allowing up to 50% of data to be outliers.

Unfortunately, there is no closed form solution to solve a least median of squares minimization problem, so a monte carlo type approach must be used, similar to RANSAC. For this reason, a least median of squares approach was not used here.

3.4.4 M-estimators

M-estimators are similar to least squares, except instead of squaring each residual value we apply a kernel function $\rho(x)$ to it [35]. $\rho(x)$ can be chosen carefully such that we minimize the contribution of outliers towards the final solution. Regardless of the kernel function, we wish to minimize the sum of the kernel function over all data points, as seen in Equation 3.19

$$Cost = \sum_i \rho(r_i). \tag{3.19}$$

$\rho(x)$ should be a symmetric, positive definite function with a minimum at 0 [34].

Several kernel functions have been used with m-estimators since the Huber function was first published by Huber in 1964 [36]. Several common kernels are shown in Table 3.4.

The derivative of the kernel function with respect to x is referred to as the influence function $\psi(x)$, and measures the influence of a particular data point on the final solution. Dividing the influence function by x we can come up with a weighting function $w(x)$. This weighting function defines how heavily weighted each point should be, and can be used as a weighting on each data point so that a problem can be solved with a normal least squares approach.

Name	Kernel Function $\rho(x)$
L_2 (Least Squares)	$\frac{x^2}{2}$
L_1 (Least Absolute)	$\|x\|$
Huber	$\begin{cases} \frac{x^2}{2} & \text{if} \|x\| \leq c \\ c(\|x\| - \frac{k}{2}) & \text{if} \|x\| > c \end{cases}$
Cauchy	$\frac{c^2}{2} \log{(1 + (\frac{x}{c})^2)}$
Welsch	$\frac{c^2}{2} \left(1 - \exp{(-(\frac{x}{c})^2)}\right)$
Tukey	$\begin{cases} \frac{c^2}{6} \left(1 - (1 - (\frac{x}{c})^2)\right) & \text{if} \|x\| \leq c \\ \frac{c^2}{6} & \text{if} \|x\| > c \end{cases}$

Table 3.4: Several popular m-estimator kernels [34]. Note that x is defined as the Euclidean distance between a data point and the model's prediction, and c is a model parameter used to weight the influence of inliers and outliers.

M-estimators problems are typically solved in one of three ways [34]. Gradient descent can be applied to the optimization function. We are guaranteed to find a local minimum, but this process is often slow. The Newton-Raphson method is commonly used to quickly find a local minimum, but this can be unstable if the starting point is far from the minimum. Iterative re-weighted least squares is often used, but this procedure requires both an ordinary least squares problem and a kernel that is strictly convex.

An m-estimator solution was chosen instead of other outlier robust regression methods. Basic least squares are not robust to outliers, especially in machine vision where outliers can be severe due to mismatched features. Compared to least median of squares problems they are much more efficient to solve. RANSAC is susceptible problems caused by inlier noise, and can take a long time to run when fitting a model with a large number of degrees of freedom.

3.4.5 Use of m-estimators

In this section, specific choices made within the m-estimator framework will be discussed and justified. The kernel choice is discussed in Section 3.4.5.1. Our optimization method is shown in Section 3.4.5.2, and the calculation of an ideal kernel parameter is presented in Section 3.4.5.4.

3.4.5.1 Kernel choice

Different kernel functions were considered for use and had to be evaluated on their accuracy on inliers, robustness towards outliers, stability, and performance considerations. The Welsch kernel was ultimately chosen as it had a good combination of all desired criteria.

Looking exclusively at inliers, the least squares approach to parameter fitting has optimal efficiency, since it is equivalent to the maximum likelihood estimator (MLE). An ideal kernel would therefore be one that treats inliers as close as possible to the L_2 kernel. To determine a kernel's treatment of inliers we can take the limit of the kernel as its parameter goes to infinity, effectively making all points considered to be inliers. These limits are shown in Table 3.5. We can see that all the considered kernel functions except the least absolute kernel will asymptotically treat inliers in the same way as the least squares approach. This property also allows us to use the technique shown in Section 3.4.5.2 to efficiently optimize the model parameters.

Name	Kernel Limit as $c \to \infty$
L_2 (Least Squares)	$\frac{x^2}{2}$
L_1 (Least Absolute)	$\lvert x \rvert$
Huber	$\frac{x^2}{2}$
Cauchy	$\frac{x^2}{2}$
Welsch	$\frac{x^2}{2}$
Tukey	$\frac{x^2}{2}$

Table 3.5: Kernel treatment of inliers. The limit of the kernel function is taken as the kernel parameter c goes to infinity. This allows us to see how the kernel will treat data composed entirely of inliers, and is meant to approximate how the kernel will treat inliers of real data.

KLT feature extraction can occasionally produce matches that deviate from inliers by a large margin. For this reason, a kernel function to be used should be bounded to avoid large effects from these outliers. Table 3.6 shows the value of each kernel function as $x \to \infty$, allowing us to see the result of each kernel to a significant outlier. Only the Welsch and Tukey kernel functions are bounded, though all kernels result in asymptotically smaller results than the least squares approach for large x.

Name	Kernel Limit as $x \to \infty$
L_2 (Least Squares)	∞
L_1 (Least Absolute)	∞
Huber	∞
Cauchy	∞
Welsch	$\frac{c^2}{2}$
Tukey	$\frac{c^2}{6}$

Table 3.6: Kernel treatment of extreme outliers. The limit of the kernel function is taken x goes to infinity, allowing the effect of the kernel parameter on outliers that deviate significantly from the proper model.

The stability and convergence is an important consideration in choosing a suitable kernel function. To improve stability, the kernel function and its first and second derivatives (depending on the convergence algorithm used) should be continuous. The second derivative of the L_1 has a discontinuity at $x = 0$, but all others kernels should exhibit good stability.

For this project, performance considerations on a low power processor are of importance. Branching can be computationally expensive for a processor to implement, especially simple processors that lack branch prediction. Any piecewise kernel will require branching in the optimization routine, which would likely increase the time required to run an iteration of any optimization routine. Additionally, a piecewise kernel complicates the derivative optimization outlined in Section 4.1. For these reasons, the Tukey and Huber kernels were not considered.

Following these criteria, the most suitable kernel function was deemed to be the Welsch function. With a proper kernel parameter it will provide a solution equivalent to least squares on a dataset of exclusively inliers, and also has a bounded response as x $\rightarrow \infty$, so large outliers will have a limited effect on the model. It is a non-piecewise function with defined first and second derivatives, making it both stable and efficient to implement. The response of the Welsch kernel is shown in Figure 3.10. The influence function $\psi(x)$ and weight function $w(x)$ of the welsch kernel are shown in Equations 3.20 and 3.21 respectively.

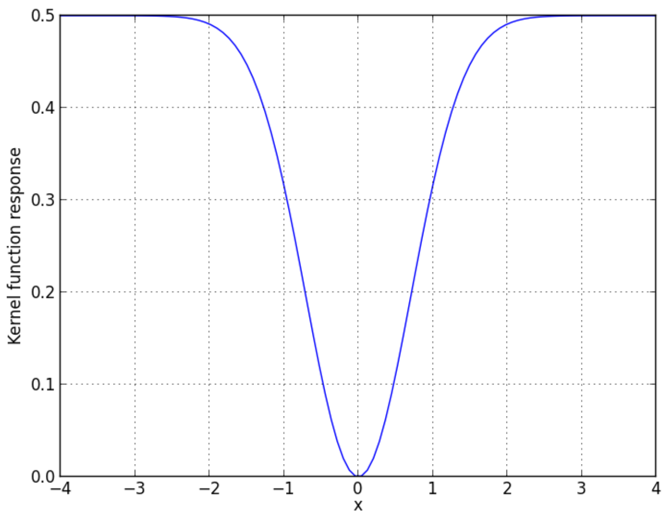
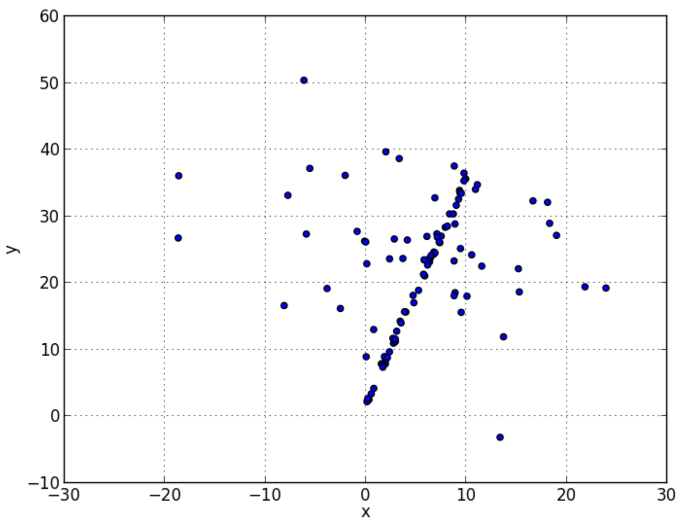


FIGURE 3.10: Welsch kernel response. The kernel parameter $w = 1$.

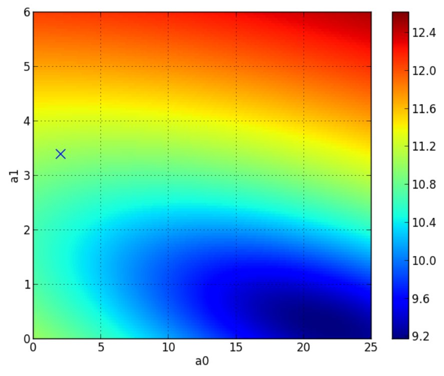
$$\psi_{welsch} = x \exp\left(-(\frac{x}{c})^2\right) \tag{3.20}$$

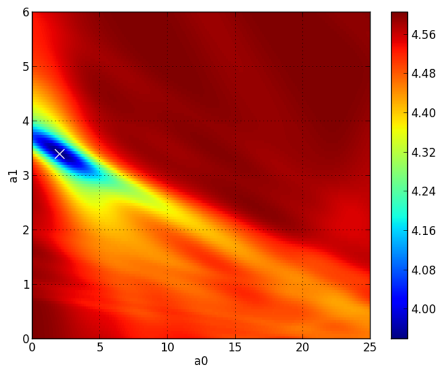
$$w_{welsch} = \exp\left(-(\frac{x}{c})^2\right) \tag{3.21}$$

To illustrate the response of the Welsch kernel, a 2-D line fitting example with outliers was created. The raw data, data parameters, and both Welsch and least squares optimization surfaces are shown in Figure 3.11. The least squares solution is far from the optimal parameters, while the Welsch fit appears to be close to the true parameters. However, there are local minima on the Welsch plot which complicate fitting compared to the quadratic least squares plot.



(A)





(B) (C)

FIGURE 3.11: Comparison of Welsch kernel and least squares. 50 inliers following a straight line with slope 3.4 and a y-intercept of 2.0, with gaussian noise with a standard deviation of 0.5 added. 50 outliers are added following the same line but with a standard deviation of 10.0 are added. The raw data is shown in Figure 3.11a. Figure 3.11b shows the optimization surface of this problem using a least squares kernel. Figure 3.11c shows the optimization surface of this problem using a Welsch kernel with a width of 2.0. An 'x' symbol marks the true solution parameters on both optimization surfaces.

3.4.5.2 Optimization method

Iterative weighted least squares is commonly used to compute parameters of an m-estimator model. However, this requires that the underlying model could be solved using an ordinary least squares solution – the output must be a linear function of all input parameters. In all sub-models used here and the overall model tuning this is violated – each contains at least one sinusoidal term. The Newton-Raphson method is an alternative solution to least squares to solve an m-estimate problem [34], and was used here.

The multivariable Newton Raphson method assumes that the optimization surface is quadratic, and will iteratively compute a solution that will lie at a point where the gradient in each direction is 0. The point can be a local minimum, maximum or saddle point of the function.

The Welsch m-estimator kernel belongs to a family of redescending m-estimator kernels [37]. While these are known to outperform bounded m-estimators, they are typically non-convex and have multiple local minima. One heuristic to attempt to find the local maximum is to use the solution to a convex m-estimate approach as a starting point, and iterate from there. Instead of this approach, a solution is described below where the problem is slowly modified from a convex least squares solution to the final m-estimate problem by slowly varying the kernel parameter.

The Welsch kernel will asymptotically approach a least squares solution as the parameter $w(x)$ tends towards infinity. The least squares solution is quadratic, so given a problem that can be solved using ordinary least squares, Newton's method will provide the exact solution after a single iteration (barring any numerical errors). With a sufficiently large $w(x)$, we are able to come close to the least squares solution using a Welsch kernel and one iteration of the Newton Raphson method. Additionally, it was observed that the area immediately surrounding the global minimum was locally convex, and an optimization algorithm with a starting point sufficiently close to the global maximum will move towards a better solution. The Welsch kernel function is convex surrounding x = 0, so the sum of all features shifts surrounding the solution (for a particular value of w) will produce a convex solution space under the assumption that the majority of feature points are inliers.

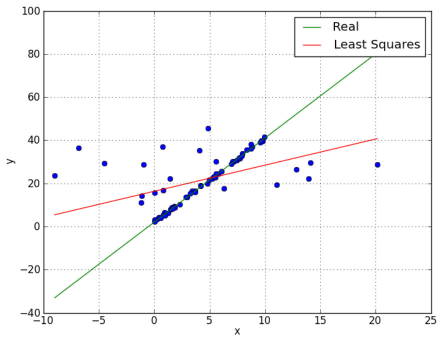
By decreasing the kernel parameter, an optimization algorithm can take advantage of the global convexity of the least squares solution, and stay contained within the local convexity of the global minimum as local minima begin to appear. The changing surface as w decreases in shown in Figure 3.12. The contour line surrounding the minimum can be seen to move towards the 'X' marking the true parameters as w decreases. Additionally, the area about the minimum where the surface is convex decreases as we decrease w.

The proposed optimization algorithm is to start at a large value of w, and find the global maximum that is expected to be close to the least squares solution. From there, we iteratively decrease w by a small amount, and find the optimal parameters using the previous optimum as a starting point. This is continued until we reach our desired final value of w, which is determined in Section 3.4.5.4.

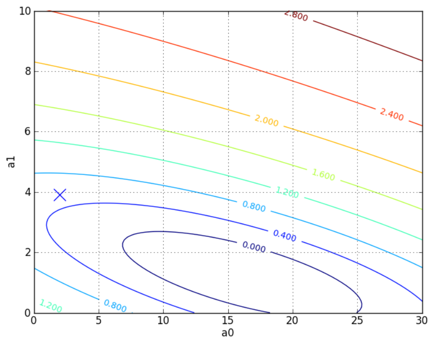
A least squares solution is desired for the first iteration. However, a Welsch kernel is used with a large w. While this is more expensive computationally, it simplifies the code and allows reuse through all iterations. This can be important on low power processors where the code memory is limited.

As seen in Figure 3.12, changes in w do not appear to affect the solution surface in a linear manner. Instead it was found that as w decreases, small changes will have a greater effect. For this reason, discrete values of w were logarithmically spaced. The choice to use logarithmic spacing is likely not ideal, and further work required in this area is discussed in Section 6.2.3. Figure 3.13 shows the location of the parameters as w is iteratively decreased and the problem is fit using a single iteration of the Newton-Raphson method. The parameters begin near to the least squares solution and slowly move toward the global minimum of the Welsch m-estimator solution.

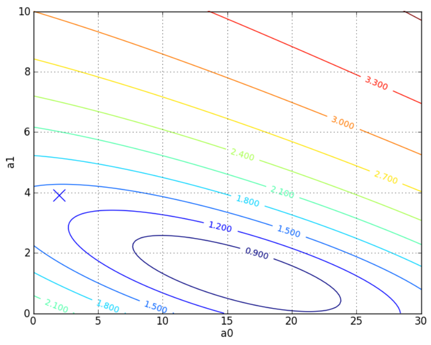
The starting value for the initial full frame movement model was chosen to be 1000. This value was determined experimentally by performing full frame video



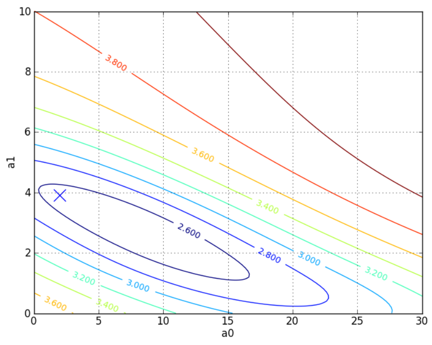
(A) Raw data and the least squares solution.



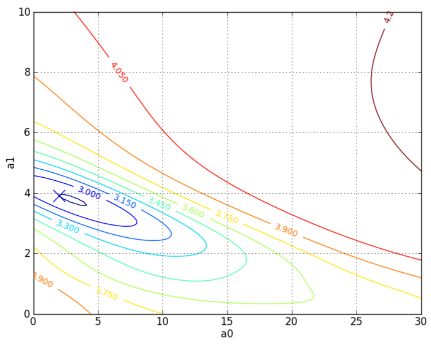
(B) $w = 100$



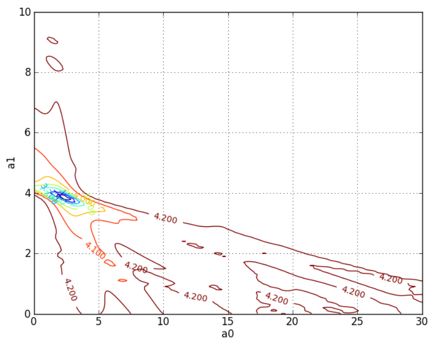
(C) $w = 60$



(D) $w = 20$



(E) $w = 10$



(F) $w = 1$

FIGURE 3.12: Optimization surfaces as w decreases. The parameters a_0 and a_1 were fit in the model $y = a_0x + a1$ according to data shown in Figure 3.12a. The remaining figures show the optimization surface as w decreases from 100 to 1. Note that the surface is close to quadratic in Figure 3.12b. As the width decreases, an increasingly smaller region is convex surrounding the global minimum. In each plot, and 'X' marks the true solution used to generate data.

stabilization with varying values of w. Smaller values of w were tested until frames were found to contain incorrectly optimized solutions at $w = 100$. Due to the logarithmic spacing of w values to be used, increasing the starting value by a large amount does not actually result in many more iterations, but may result in greater stability in particularly difficult to track video sequences.

The number of discrete values for w was determined in a similar value to w. It was determined that 10 values of w provided sufficient stability for the full frame model, all sinusoidal models and the final full model optimization.

Figure 3.13 shows how the parameters progress are the value of w changes. While the parameters used here do not exactly match those used in video motion models, this serves to illustrate the way in which the parameters will trend toward the true solution as w decreases.

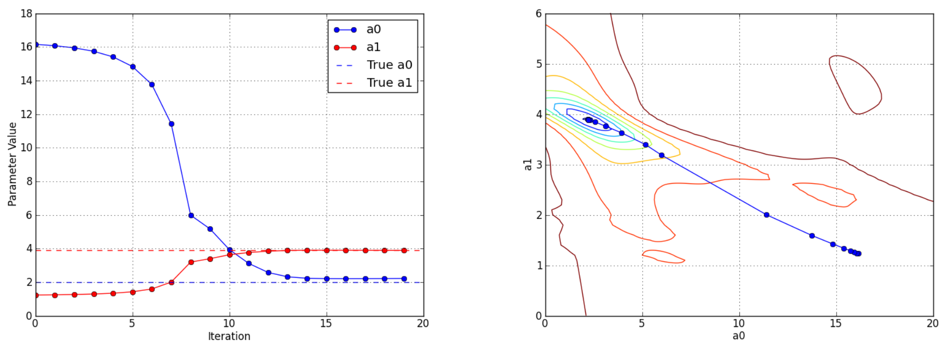


FIGURE 3.13: Change in model parameters as w decreases. Here w goes from 100 to 2 in 20 logarithmically spaced steps. The true parameter values are shown with dotted lines. The raw data for this problem can be seen in Figure 3.12a. The parameters can be seen to slightly overstep their ideal path at iteration 8. This effect can cause problems when optimizing using the Newton-Raphson method, but LM optimization will prevent the solution from overstepping to an area of higher cost than the previous iteration. The parameters remain nearly constant for the final five iterations, suggesting that the final kernel width is smaller than ideal (this data was artificial, and an optimal ending value was not calculated).

3.4.5.3 Levenberg-Marquardt Optimization

LM is commonly used to minimize non-linear least squares solutions that cannot be solved using an algebraic solution [38]. LM optimization interpolates between gradient descent and the Newton-Raphson method according to a parameter λ, as shown in Equation 3.22

$$\vec{p}_{n+1} = \vec{p}_n - (\mathbf{H} + \lambda diag(\mathbf{H}))^{-1}\vec{J}, \tag{3.22}$$

where $\vec{p}_n$ is the vector of parameters at iteration n, $\mathbf{H}$ is the Hessian matrix of second derivatives of the cost function with respect to all parameters and $\vec{J}$ is the Jacobian vector of first derivatives of the cost function with respect to all parameters. As λ tends towards zero this algorithm approaches the Newton-Raphson method, and as λ tends toward ∞ we move towards basic gradient descent. λ can be adjusted according to a number of heuristics designed to provide a small value of λ where the solution space is convex and we can take larger steps and a large value of λ on flatter portions of the space where the robustness of gradient descent is needed [38].

Both LM optimization and the Newton-Raphson method require the construction of Hessian and Jacobian matrices. In this case, matrix elements are set to the sum of partial derivatives of the cost function over all feature points. The Hessian matrix is a matrix of second partial derivatives of the cost function with respect to parameters to be optimized, as seen in Equation 3.23

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 \rho}{\partial p_1^2} & \frac{\partial^2 \rho}{\partial p_1\, \partial p_2} & \cdots & \frac{\partial^2 \rho}{\partial p_1\, \partial p_n} \\ \frac{\partial^2 \rho}{\partial p_2\, \partial p_1} & \frac{\partial^2 \rho}{\partial p_2^2} & \cdots & \frac{\partial^2 \rho}{\partial p_2\, \partial p_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 \rho}{\partial p_n\, \partial p_1} & \frac{\partial^2 \rho}{\partial p_n\, \partial p_2} & \cdots & \frac{\partial^2 \rho}{\partial p_n^2} \end{bmatrix}, \tag{3.23}$$

where ρ is the cost functions, and p_i is a parameter to be optimized. Similarly, the Jacobian vector contains first derivatives of the cost function, as shown in Equation 3.24

$$\vec{J} = \begin{bmatrix} \frac{\partial \rho}{\partial p_1} & \frac{\partial \rho}{\partial p_2} & \cdots & \frac{\partial \rho}{\partial p_n} \end{bmatrix}. \tag{3.24}$$

Partial derivatives were computed using a symbolic math library, and were large expressions due to the size of the cost function (Equations 3.13 and 3.14).

LM optimization was used instead of the Newton-Raphson method in the final solution to provide further stability at a small performance cost. If the gap between subsequent values of w would be too large for the Newton-Raphson method to remain stable, the gradient descent would begin to take over, and we are guaranteed to take a (potentially very small) step in the direction of an improved solution. The provides stability in two main ways:

- If the current solution falls outside the trust region of the Newton-Raphson method, we can take a step in the correct direction and fall back within the trust region for the next step of the algorithm.
- If the solution moves a large amount and we are far from any local minimum, we are likely to take a small step. This contrasts with the Newton-Raphson algorithm, which will often take large steps and can become unstable. While such a solution is unlikely to ever find the global minimum, the final solution will not be any worse than the solution at the step before we lost the local minimum.

In this work, the following heuristic was used to adjust λ according to our position in the solution space, according to [38]:

$$\lambda_{n+1} = \begin{cases} 2\lambda_n & \text{If} \quad Cost(\vec{p}_{n+1}) > Cost(\vec{p}) \\ \frac{\lambda_n}{2} & \text{Otherwise} \end{cases} . \tag{3.25}$$

The solution is not allowed to step to a point where the cost is higher than it was previously. Instead, we continue to increase λ by a factor of two until we find a step such that the cost is decreased. This does not require recalculation of the Hessian and Jacobian matrices, which are the most time consuming parts of each iteration. The value of λ begins at 0.01 for each optimization step. To choose this value, the starting value of λ was swept while counting the number of times this value had to be increased. The chosen value was the point at which λ had to be increased the fewest number of times over the test videos. The results in each optimization step being nearly equal to the Newton-Raphson method, which tends to find the optimum using few iterations. In rare cases where the solution cannot be found, λ will increase until we move in an improving direction according to gradient descent.

In some cases, the Hessian matrix is ill conditioned, even with the added diagonal values. Inverting such a matrix will not work, and typically produces Not a Number (NaN) in the resulting matrix. To avoid such conditions, diagonal values are checked to ensure that they are greater than a minimum threshold value. If diagonal values are less than 10^{-6}, the diagonal value is replaced with 1.0, and all values on the same row and column are set to zeros. The corresponding entry of the Jacobian matrix is also set to 0. This allows the Hessian to be inverted as if the problem parameter was not being updated. Since the Jacobian matrix is zero for that element, that parameter will not be updated. In many cases, the next iteration's Hessian matrix will no longer be ill conditioned, either due to a change in the Welsch kernel parameter or updated model parameters from the previous iteration.

3.4.5.4 Ideal final kernel width parameter computation

The ideal kernel width for the Welsch function needs to be chosen properly – too large and the computation will approach a least squares solution, too small and no global optimum solution will exist. A method to determine optimal widths will be presented, and two optimal widths will be calculated: one for the initial full frame motion estimation (where pixels distorted by rolling shutter effects should still be considered inliers), and another for rolling shutter modeling and final model tuning.

We first define the number of inliers and number of outliers as n_{in} and n_{out} respectively. Both inliers are outliers are assumed to follow normal distributions with standard deviations σ_{in} and σ_{out} respectively. The assumption that outliers follow a normal distribution is likely incorrect in many cases, but outlier distribution is video-dependent, and it would be difficult to construct a single distribution accurate in all cases.

One approach to parameter calculation is to maximize the sum of the weighting function $w(x)$ over all inliers, while minimizing the sum of $w(x)$ over all outliers. To do this, we first multiply the inlier distribution by the weight function, obtaining another gaussian k_{in}, as seen in Equation 3.26

$$k_{in} = \frac{n_{in}}{\sigma_{in}\sqrt{2\pi}} e^{-\frac{x^2}{2\sigma_{in}^2}} \quad \exp\left(-(\frac{x}{c})^2\right) = \frac{n_{in}}{\sigma_{in}\sqrt{2\pi}} e^{-\frac{x^2(c^2+2\sigma_{in}^2)}{2\sigma_{in}^2 c^2}}. \tag{3.26}$$

Next we integrate k_{in} over all of x to find the sum of all inlier contributions:

$$j_{in} = \int_{-\infty}^{\infty} k_{in}\ \partial x = \frac{nc}{\sqrt{c^2 + 2s^2}}, \tag{3.27}$$

where j_{in} has a maximum when $c = \infty$, and a minimum when $c = 0$. This means the inlier contributions will be maximized when $c = \infty$. Using the same approach with outliers, we find that they will also be maximized at $c = \infty$. These opposing goals would require us to set a weight on the importance of maximizing inlier contributions vs. minimizing outlier contributions.

Instead of calculating an algebraic limit, a monte carlo type simulation was run to simulate the accuracy of various kernel parameters on a translation and rotation fitting model. The convergence method described in Section 3.4.5.2 was used to find the solution optimum. By finding an appropriate kernel width on a simple problem with similar inlier and outlier distribution, we can determine a suitable $w(x)$ for use more complex models.

The inner simulation will be to calculate the R^2 value over only inliers, using a Welsch kernel fit on both inliers and outliers. For this test, we assumed that 80% of data would be inliers and the remaining 20% would be outliers, based on visual examination of extracted feature points over several videos. Translation and rotation values were generated according to random normal distributions with a standard deviation of three pixels for translation and two degrees for rotation. x and y values were generated to follow a uniform random distribution over the entire video frame size. Inlier noise was added with a standard deviation of 0.1. Outlier noise had a standard deviation of 10.0, based on visual examination of feature points from video. This was done by plotting the point density of feature shifts against x and y parameters, and comparing that density distribution to a normal distribution. While this is a fairly arbitrary method, the problem is unsupervised and additional information such as gyroscopic data is necessary to calculate parameters. The distribution varied slightly between frames and videos so an approximate mean was over all sources. Figure 3.14 shows the root mean squared error (RMSE) over several different widths for the translation in x and y, and the rotation of the frame.

The optimal values of w were 2.1, 2.1 and 2.4 for x, y, and R parameters respectively. The final kernel parameter w was chosen to be 2.3 - an average of the three minima. This value was used for the row based x-translation, row based y-translation and row based rotation, and the final full model optimization.

To calculate an ideal initial frame fit (while rolling shutter distortions remain in the frame), the noise on the inliers was increased to 2.0. This value was iteratively refined by picking a w value, fitting models to the data, and inspecting the amplitudes of the sinusoids used to model the rolling shutter distortion. The inlier

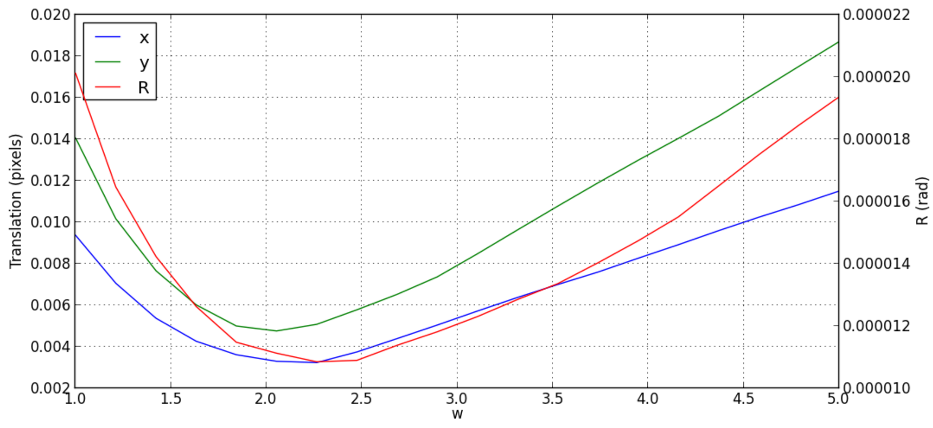


FIGURE 3.14: Monte carlo simulation to find an ideal $w(x)$. A rotation and x and y translations were calculated using simulated data containing 80 inliers and 20 outliers. The RMSE is calculated for 50 sets of data points for each kernel width. The least squares solution errors were 0.091 pixels, 0.161 pixels and 0.00021 Rad for x, y and R respectively, each more than an order of magnitude higher than the Welsch kernel fit.

noise value is equal to the standard deviation of data randomly sampled from the sinusoids in the distortion model, which is equal to the amplitude divided by $\sqrt{2}$. The same monte carlo simulation was run with increased inlier noise, resulting in the data shown in Figure 3.15. The ideal full frame Welsch kernel parameter was chosen to be 7.0, the average of the minima of 6.0, 7.0 and 8.0 for x, y and R respectively. This value was used for the full frame movement, and also for initial weights for the row based x-translation, row based y-translation and row based rotation.

3.5 Motion Filtering

Once camera motion from frame to frame has been calculated, the motion must be filtered before the frames are distorted to make motion appear smooth. This filter should result in camera motion that has been low pass filtered to remove high frequency movements.

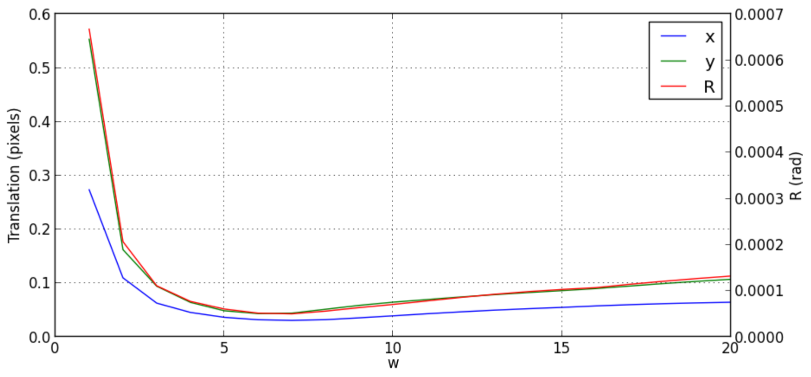


FIGURE 3.15: Monte carlo simulation to find an ideal $w(x)$ for full frame parameters. Inlier noise was increased to 2.0 using the same simulation described in Figure 3.14. Due to the added inlier noise, the advantage of a smaller kernel parameter is less than with the decreased kernel noise shown in Figure 3.14.

Motion filtering was not a main focus of this work, but a solution had to be developed in order to output videos. The output camera motion was chosen to be an exponential moving average (EMA) filtered integral of the calculated motion. This acts as a low pass filter, and can be computed very efficiently. The EMA filter was chosen for its simplicity to implement and its computational efficiency. More complex motion filtering algorithms are discussed in Section 6.2.4. The motion of each pixel was calculated and decayed for each frame, according to Equation 3.28

$$I^{n+1}(x,y) = \alpha I^{n}(x,y) + T^{n}(x,y), \tag{3.28}$$

where $I^n(x,y)$ is the shift applied to a pixel located at (x,y) at frame n, α is a decay factor, and T^n is the calculated transformation at time n.

α was set to 0.98, but could be adjusted by a user if a different balance between stabilization quality and minimization of crop area is desired. Note that setting α to a larger value will often result in a smaller output video after corners are clipped (see Section 3.6 for details), while a smaller value will result in less smooth video. The relationship between α and the crop size depends on the motion calculated from the video.

The result of the EMA filter can be seen in Figure 3.16. The calculated total camera motion is shown, as well as the smoothed camera motion that will be seen in the output video.

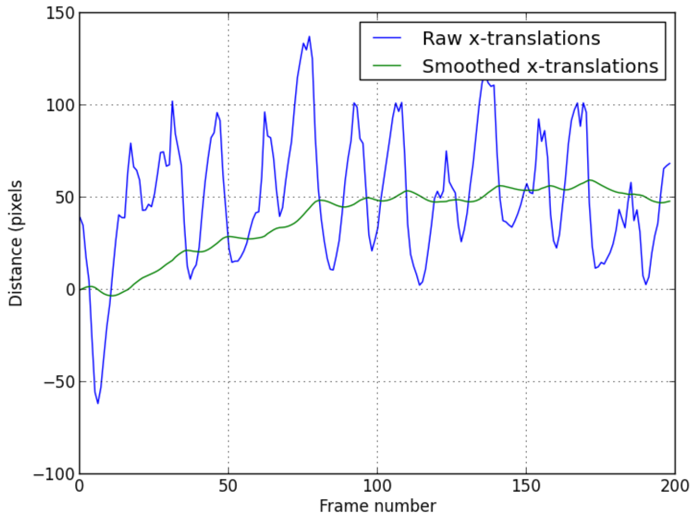


FIGURE 3.16: Motion smoothing using an EMA filter. Here the calculated total camera motion is shown, along with the smoothed motion that will be present in the output video. This data comes from the first 200 frames of the walking video – each large spike is the result of taking a step while walking.

The integrating EMA filter is a special case of an Infinite Impulse Response (IIR) filter, where a single recursive tap is used. The filter response is shown in Equation 3.29

$$S(\omega) = \frac{1}{1 - 2\alpha\cos(\omega) + \alpha^2}, \tag{3.29}$$

where ω is the angular frequency of the integral of camera motion. The frequency response is also shown graphically using $\alpha = 0.98$ in Figure 3.17

3.6 Image Warping

Image warping was performed by calculating the original pixel location of every pixel in an output image, and using linear interpolation to calculate the pixel value in the new image. For each pixel on a particular frame, the x and y coordinates

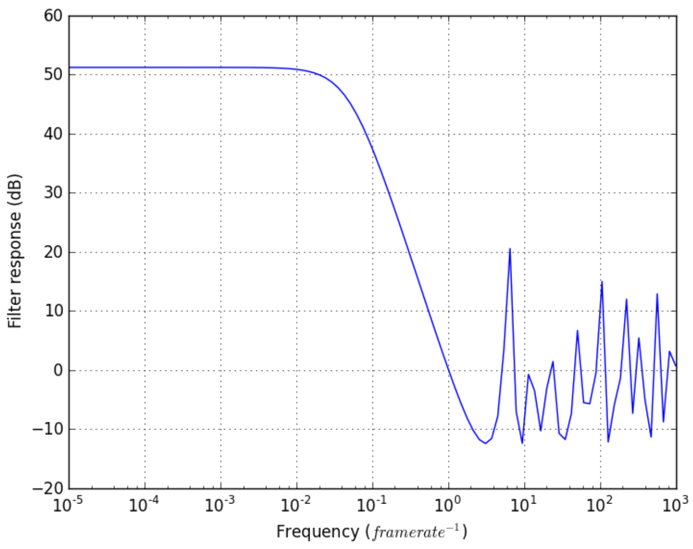


FIGURE 3.17: Frequency response of the low pass motion EMA filter, using $\alpha = 0.98$. Note that the low frequency response is a significant amplification since this filter is designed to provide a low pass filtered integral of the frame to frame motion. The unstable region at higher frequencies is cause by aliasing.

of the original distorted video frame are calculated using the camera motion determined in Section 3.5. If this original location is outside the video frame, the clip region is adjusted since we have no information of what should appear in that location.

Transformed pixel locations are no longer integer values, so a method is needed to determine what pixel value to use. In this case, bilinear interpolation was used, as shown in Equation 3.30

$$\begin{aligned} I(x,y) = &I(x_{floor},y_{floor})(x_{ceil}-x)(y_{ceil}-y) \\ &+ I(x_{floor},y_{ceil})(x_{ceil}-x)(y-y_{floor}) \\ &+ I(x_{ceil},y_{floor})(x-x_{floor})(y_{ceil}-y) \\ &+ I(x_{ceil},y_{ceil})(x-x_{floor})(y-y_{floor}) \end{aligned}, \tag{3.30}$$

where x_{floor} and y_{floor} are the values of x and y rounded down to the nearest integer, while x_{ceil} and y_{ceil} are the values of x and y rounded up.

Bicubic interpolation is often used in image processing applications as it will give a smoother result with fewer artifacts since the derivatives of the resulting interpolation are smooth. However, it takes longer to run and was not used for this reason. In situations where accuracy is preferred to speed bicubic interpolation could easily replace bilinear interpolation in this work.

3.7 Overall Model Fitting Algorithm

In this section, the overall fitting method will be presented. This will combine details presented earlier in Section 3 to explain how the all of the subsections work together. Figure 3.18 shows the overall fitting process that is explained below.

A series of image frames is first extracted from the video sequence, and grayscale versions (since the KLT tracking algorithm operates only on grayscale images) of each image are created for processing. Up to 1000 feature points are extracted from each image and matched to feature points in the frame that follows is, using the grayscale images. This number of feature points was suggested by Grundmann as being an ideal number of features to use. Stabilization performed with fewer features did not perform as well, and adding more features decreases performance, but no improvement in video quality could be seen. A rolling shutter model is fit to each set of feature pairs, using the method described below in Section 3.7.1.

A shift matrix for each direction (x and y) is created to be the size of the image in pixels, initialized to all zeros. Starting from the first transform model, the shift for each pixel is added to that index of the matrix for both the x and y directions. This produces two matrices showing where each output pixel should be taken from. These matrices are used to implement the motion filtering algorithm. Due to the non-linearity of the distortion model, a simple additive approach cannot be taken. These shifts are used with a bilinear interpolation algorithm to distort a

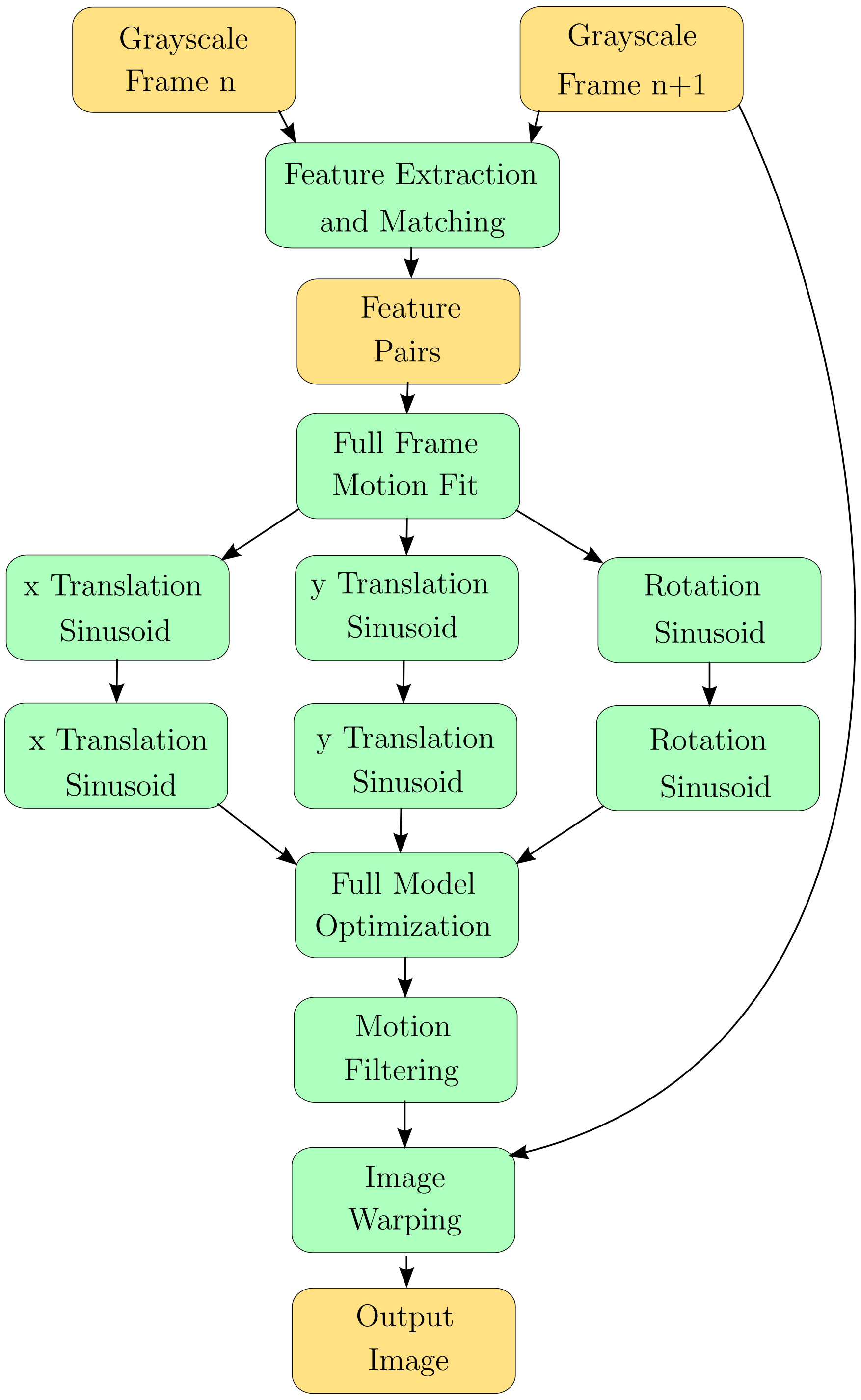


FIGURE 3.18: Overall fitting process flowchart

color input frame to produce an output image, which is stored in memory. The required crop window is recalculated as each output image is created. Once all output images have been created the crop window is applied to each frame, and all frames are saved as a video.

3.7.1 Model Fitting to Feature Pairs

The fitting of the distortion model is complex and requires the optimization of parameters to a non-convex cost function. A fast yet robust optimization approach is required to fit the model. Several sub-models (individual sinusoids) are created and applied onto the result of the previous sub-model, resulting in several simple models that can be fit quickly and largely in parallel.

The full frame camera motion model is first applied to remove full frame camera motion. Subtracting this motion makes the parameter fitting of the other models easier, since the full frame motion is considered noise by those models, and we are removing noise through the full frame fit. This fit must be completed before any of the next steps. This is performed using 10 iterations of LM optimization using a Welsch kernel. The kernel width is decreased from 1000 to 7 using 10 logarithmically spaced points (see Section 3.4.5.2 for details).

The three row based translation models are then fit, and could be done in parallel. The first of the models is fit to x shifts of the data, and is completely separate from the y shifts of the next models. The first of the y models fits compression or expansion of entire rows, while the second models the rotation of pixels about the center of the frame. By assuming the mean of all feature's y values is centered in the frame (the feature extraction algorithm presented in Section 3.2 does this well) the two y shift models will produce orthogonal corrections and can therefore be fit in parallel. Each individual fit has two frequency components that must be performed in sequence, otherwise the single largest frequency would be fit twice.

The method used to fit each of these row based translation models is described in detail in Section 3.7.2.

Once each of these three sub-models have been fit, the entire model is optimized using 10 iterations of LM optimization and a Welsch kernel of decreasing width. The kernel width is decreased from 7 to 2 using 10 logarithmically spaced points.

3.7.2 Row-based Translation Model Fitting

In each of the row based translation models, the amplitude, frequency and phase of two sinusoids are being fit to weighted data. Data is extracted from feature points in such a way that the same code can be reused for all three sinusoidal models.

Starting weights for each feature point are calculated using the optimized parameters of the the full frame model fit, and denoted $\vec{d}$. The m-estimator weight function is calculated for each point using the final parameter values. These weights allow us to have some prior knowledge of outliers as we move to a model that is more difficult to fit than the full frame model. These weights are normalized so that the mean weight is one to avoid requiring normalization in each model calculation, since several model fits will be performed using the same weights.

For the row based x-translation, the x-shift between features pairs is taken with all previously calculated camera motion subtracted and labeled as the vector $\vec{a}$. Note that for the second sinusoidal fit, this previously calculated camera motion includes the first sinusoidal fit. The y position of each feature point in the first frame is used as the vector $\vec{b}$. An additional vector $\vec{c}_{weights}$ is provided with all values set to 1.

The row based y-translation is similar to the x-translation, with the only difference being that the vector $\vec{a}$ is equal to the y-shift between feature pairs with previous camera motion subtracted.

The row based rotation is slightly more complex, but can be fit using the same function by using the $\vec{c}_{weights}$ vector, which is set to $(x - centerX)/(2 * centerX)$ using the x values from the first frame. The denominator is not required, but normalizes data to provide greater stability for different video resolutions. The numerator expression projects the y shift of a rotation into a two dimensional space by normalizing out the x dimension. $\vec{a}$ is set to the y-shift between feature pairs, and $\vec{b}$ is set to the y value of each feature in the first frame.

The actual fitting of the data has two main steps: a brute force discrete Fourier transform (DFT) – like step intended to find the approximate parameter values, and several iterations of LM optimization to refine those parameters and decrease the Welsch kernel width to reduce the significance of outliers.

In the first step, we calculate the sum of the product of sin and cos terms at different frequencies with $\vec{a}$ at discretely spaced frequencies, as seen in Equation 3.31

$$\begin{aligned} S^f_{sin} &= \textstyle\sum_i \sin(\vec{b}[i]\ f)\ \vec{a}[i]\ \vec{d}[i]\ \vec{c}_{weights} \\ S^f_{cos} &= \textstyle\sum_i \cos(\vec{b}[i]\ f)\ \vec{a}[i]\ \vec{d}[i]\ \vec{c}_{weights}, \end{aligned} \tag{3.31}$$

where i is the feature point index, and S^f is the total power of a particular sinusoid at frequency f. Specific frequencies used are described below.

Each frequency's signal power is calculated as the sum of the squares of sin and cos terms, as seen in Equation 3.32

$$S^f = (S^f_{sin})^2 + (S^f_{cos})^2. \tag{3.32}$$

The frequency with the maximum power is taken as the starting point for the iterative optimization routine.

This procedure is very much like the DFT. The more efficient Fast Fourier transform (FFT) could not be used here, since the data points are not equally spaced or equally weighted. For this reason the number of frequencies are kept low. Frequencies between $0.002\frac{Rad}{row}$ and $0.1\frac{Rad}{row}$ at 20 linear spaced steps were used. Lower frequencies than $0.001\frac{Rad}{row}$ were not used since a linear fit and constant offset were previously subtracted in the full frame model, and frequencies below $0.002\frac{Rad}{row}$ are within approximately 3% of a linear fit. The upper limit was chosen by examining the test videos described in Section 5.4, and finding the upper limit before significant motion blurring starts to happen in the video. At this point, even perfectly compensated videos will not appear smooth since significant amounts of motion blur will still be present.

The number of frequency steps was chosen by examining the output of several frequency analyses. An example from the Parliament video is seen in Figure 3.19. Each frequency peak is blurred from two main causes: noise present on data points, and spectral spillage caused by non– integer cycle frequencies. Noise on each data point causes a small amount of spreading on each peak. The sinusoidal terms used in Equation 3.31 do not complete an integral number of cycles during the frame window, resulting in sinusoidal spreading of each peak. This effect could be mitigated by using a window function in Equation 3.31, at the expense of accuracy of each frequency's power, and a small decrease in performance. The spread caused by these effects limits our frequency resolution, so we only check 20 points to save time.

Once a frequency starting point has been chosen, it is used as a starting point for the iterative Welsch kernel-based optimization routine. The amplitude, and phase are calculated from the sine and cosine and total powers as seen in Equations 3.33

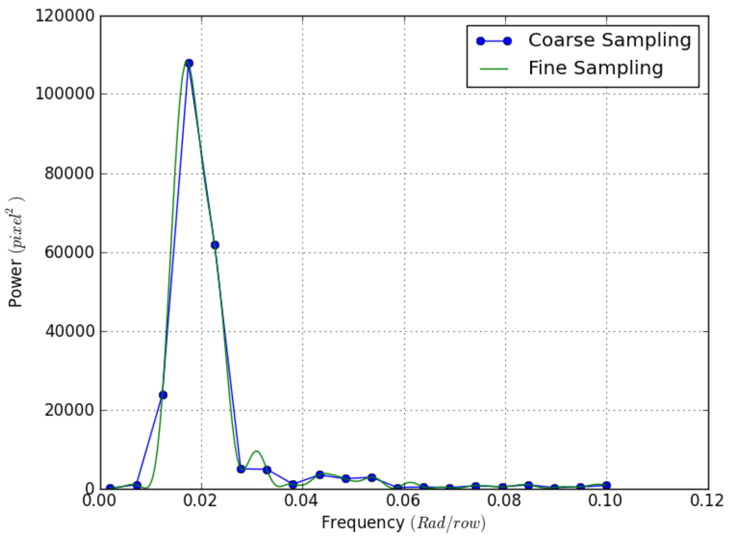


FIGURE 3.19: Frequency analysis of row based x-translation. A finely sampled frequency analysis is shown along with the coarse sampling used in the final system.

and 3.34

$$p_{amplitude} = \frac{2\sqrt{S^f}}{\|\vec{d}\|^2} \tag{3.33}$$

$$p_{phase} = \tan^{-1}\left(\frac{S^f_{cos}}{S^f_{sin}}\right). \tag{3.34}$$

Once a starting point has been found, LM iterations are used to improve the fit using a Welsch kernel of decreasing width. The objective function is shown in Equation 3.35

$$Cost = \sum_i \vec{a}_i - \vec{c}_{weights}\ p_{amplitude} \sin\left(p_{frequency}\ \vec{b}_i + p_{phase}\right). \tag{3.35}$$

The welsch kernel width parameter begins at 7.0 – the width that resulted in the initial starting weights used for the first part of the fit. The final parameter value is 2.0 which was previously calculated in Section 3.4.5.4 as the final parameter value for all fits. 10 steps were used with values spaced logarithmically. This value was chosen by starting with a large value, and decreasing it until the LM optimization frequently had to increase the λ parameter to remain stable – in other words when the parameters routinely fell outside the convex trust region surrounding the global minimum when the step size was decreased.

Chapter 4

Implementation

In this chapter several concepts intended to improve code performance of the predetermined algorithm are discussed. An algorithm for automated code optimization designed to speed up iterations of LM optimization is discussed in Section 4.1. Next, an approach to parallelization of the algorithm is discussed in Section 4.2, aimed at a system with a large number of processing cores. Finally, an approach to fixed point conversion is discussed in Section 4.3 which would allow the algorithm to efficiently run on an embedded platform lacking floating point hardware.

4.1 Automated code optimization

Iterations of LM optimization can be slow, especially for the full model optimization with 24 parameters to be optimized simultaneously. Here an algorithm to optimize code to calculate the Hessian and Jacobian matrices will be presented. While this method does not asymptotically improve speed with respect to the number of model parameters or feature points, for the optimization problems in this work the iteration time decreased dramatically.

The calculation of a single element in either the Jacobian or Hessian matrix involves a sum over all feature pairs in this problem (1000 data points). Additionally, the size of the Jacobian and Hessian matrices are $1 \times n_p$ and $n_p \times n_p$, where n_p is the number of parameters. For the final full model optimization with 24 parameters this results in 600 individual equations, each of which is a sum over all data points. One approach to simplify this computation is to approximate the derivatives using forward difference approximations. Using only the basic cost function call, $n_p + n_p^2/2$ calls are required, which remains slow. In this approach, derivatives were calculated algebraically for each matrix element.

To derive these equations, automatic differentiation was performed on the cost function and code was generated to compute the Hessian and Jacobian. It was observed that matrix elements are similar, and common terms could be factored out to decrease run time.

An iterative process was created to find suitable sub-expressions to precompute. A sub-expression here is considered any part of an equation that could be removed from the equation and replaced with a single term. That expression is then replaced with the temporary value wherever it appears in the Hessian or Jacobian matrices.

Sub-expressions were calculated by considering each equation as a tree. The expression implemented by each node was counted in a hash table as the tree was traversed. This was repeated for each matrix element equation. In expressions taking an arbitrary number of parameters (such as multiplication) an expression was added for every combination of parameters, with each combination sorted alphabetically. For example, the expression abc would produce sub-expressions abc, ab, bc and ac, which allows a common temporary variable to be shared with something like $\sin(ac)$. Pseudocode to create a list of sub-expressions is shown in Figure 4.1.

```
Data: E = Pointer to root node of expression tree
Result: L = list <Sub-expressions>
Function GetSubExpressions(E)
begin
    for argument A in E.arguments do
        L += GetSubExpressions(A)
    end
    return L
end

Adds extra nodes for combinations of arguments in the case of multiplication or
addition
Data: L = list <Sub-expressions>
Result: R = list<Sub-expressions>
Function AddCombinations(L)
begin
    for sub-expression S in L do
        if S is multiplication or addition then
            Add all combinations of arguments
            for sub-expression C in Combinations(S) do
                R += C
            end
        else
            R += S
        end
    end
    return R
end
```

FIGURE 4.1: Sub-expression generation for automated code optimization.

Once a list of sub-expressions and their respective counts has been gathered, the list is sorted (by a strategy described below). Beginning at the 'best' expression to replace, each matrix equation has all instances of that expression replaced with a temporary variable. This continues until we reach the stopping criteria described in Section 4.1.3. A flowchart depicting the process is shown in Figure 4.2.

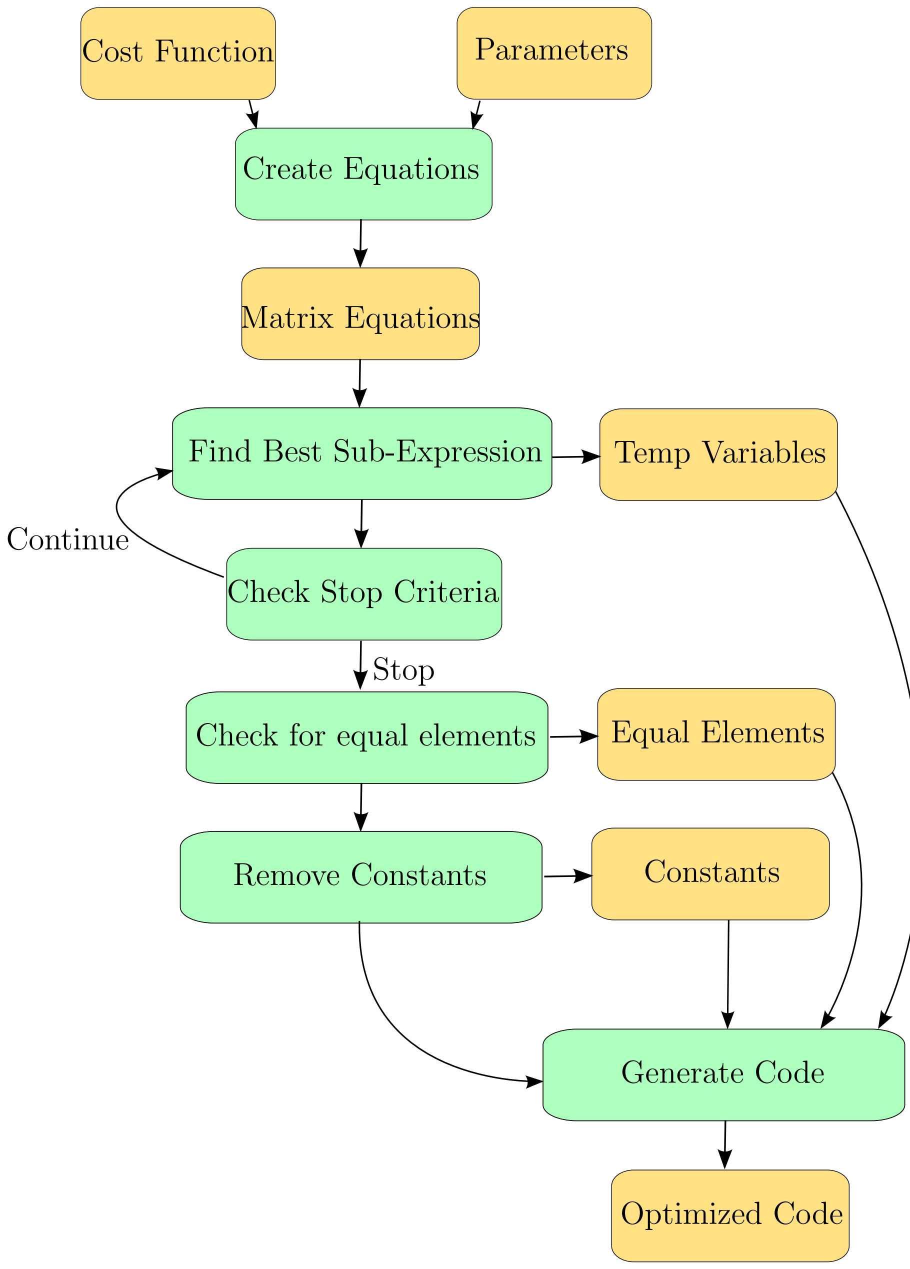


FIGURE 4.2: Automated optimization flowchart. The cost function is sum of the Welsch error of the distortion function (sum of Equations 3.13 and 3.13) summed over all feature points. The parameters are the list of model parameters found in Table 3.3.

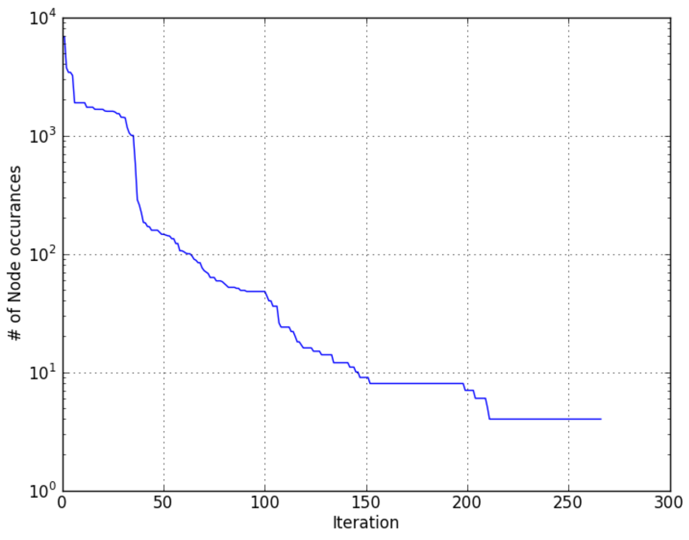


FIGURE 4.3: Sub-expression frequencies in Hessian and Jacobian computations. At each iteration, the most common sub-expression is replaced with a single temporary variable. The number of occurences of the most common sub-expression is plotted against the iteration number.

4.1.1 Sub-expression selection strategy

The sub-expression selection strategy was to look for the node that appears the most times over all equations. This is continued until no expression occurs more often than a predetermined threshold. The number of occurrences of the most common expression as they are replaced is shown in Figure 4.3.

Many alternative expression selection strategies exist, and may prove to be more effective than looking only a sub-expression counts. On a simple processor where a definite number of clock cycles are required for a given computation, this knowledge could be taken advantage of to select slower expressions first. One such strategy could be to sort sub-expressions by the total clock cycles required to evaluate them – the number of times that expression exists multiplied by the number of cycles required for a single evaluation. The node selection strategy would be especially important on platforms where few terms could be used due to memory constraints. In this work, memory usage was not a priority so a simple strategy could be used without a significant performance penalty. A simple strategy also decreases the optimization time before code compilation.

4.1.2 Additional Optimizations

In additional to node pre-computation, several other techniques were used to improve performance:

- Any constant equations were removed from the loop and multiplied by the number of loops
- Any multiplicative factors at the root node of equations (Ex: $2\sin(x)$ would qualify but $\sin(2x)$ would not) were removed from the inner loop and the sum was multiplied post-loops.
- Many Hessian matrix elements across the matrix are equal. These are copied over after the inner loop instead of adding to both matrix elements for each data point.

4.1.3 Stopping Criteria

A constant threshold was chosen for all model optimizations to be a minimum of nine occurrences of a particular sub-expression. This was chosen by sweeping over different minimum sub-expression occurrences, generating optimized code, and timing multiple runs of the code (100 runs were used for less-optimized solutions, and this was increased as speed increased such that the full run took approximately 10 seconds). Note that these results were performed on a laptop with an Intel i7 processor, and should be repeated to determine an optimal stopping point based on available memory and processor speed.

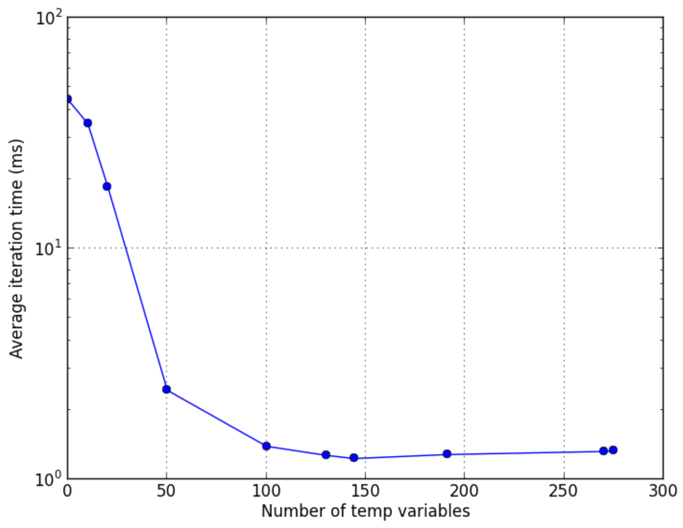


FIGURE 4.4: Number of temporary variables vs LM iteration time. A significant speedup is seen as we move from no temporary variables to a minimum located near 150 temporary variables, after which a small decrease in performance is seen. Note that without the speedups listed above the code takes approximately 75ms per iteration.

A plot of the number of temporary terms vs the average iteration time is shown in Figure 4.4. The optimal number of terms was found to be approximately 150 temporary terms, which corresponds to a minimum of nine occurrences of a expression to be factored out. These minimum value will depend on several factors such as processor and Floating Point Unit (FPU) speed and availability, and processor cache and Random Access Memory (RAM) speed and availability.

4.2 Parallelization

Segments of the algorithm can be parallelized in to improve performance on a multicore platform. These segments can be broken down into feature extraction, feature matching, distortion transfer function computation, and motion filtering and image warping. The algorithms presented here are tailored towards a platform with a large number of computation cores.

4.2.1 Feature Extraction

The use of sub-images in the feature extraction algorithm makes parallelization of this step simple. Each computation core is tasked with the feature extraction on a small number of sub-images. Cores do not need to communicate with each other, and the memory required to be transfered to each core is minimal.

4.2.2 Feature Matching

Efficient use of multiple cores in the KLT feature matching algorithm is more complex than the feature extraction step. To avoid recomputing the pyramid of images used by KLT, these images could be computed once. Each processor core would follow the pyramidal tracking procedure for each of the features previously calculated on that core in the feature extraction step. If care is taken to ensure that a single core contains features from adjacent sub-images only, memory transfer between cores can be reduced, since each core will only need portions of the lower pyramid images.

If the maximum displacement of features is limited, the memory transfer can be further reduced since each core will only need to consider features immediately surrounding the features it previously extracted.

4.2.3 Distortion Transfer Function Computation

The majority of the time spent computing transfer function parameters is spent calculating the Jacobian and Hessian matrices used in the LM optimization. For example, for the full model tuning it takes approximately $1230\mu s$ to calculate these matrices, but only $50\mu s$ to solve the linear system to calculate parameter updates.

With this in mind, the parallelization of this component was designed specifically to minimize the time required to compute the Hessian and Jacobian matrices.

Each computation core will calculate the Jacobian and Hessian matrices for the points previously extracted and matched on that core alone. Once this has been completed, the individual elements from matrices are added together to produce a single Hessian and Jacobian matrix from which the parameter updates will be computed. The addition of matrix elements can be added together using a binary tree topology to reduce the number of non-parallel additions. The solving of the linear system can then be performed on a single computation core, and new parameter values are then calculated. These parameter values are pushed to each computation core, where the matrix computations can begin for the next iteration.

4.2.4 Motion Filtering and Image Warping

The motion filtering and image warping algorithms can easily be parallelized to use a large number of cores efficiently. In the first step, the displacement of each pixel to correct distortion is calculated. This involves the evaluation of Equations 3.13 and 3.14 for each pixel of the frame. This operation can be performed in parallel on a large number of processing cores simultaneously. Next, the motion is filtered using an EMA moving filter that is once again applied to each pixel location, and can be performed in parallel by assigning a number of pixel locations to each processing core.

Image warping involves copying pixel values from a location in the raw video frame to a different location in the corrected output video. A number of pixel locations can be assigned to each computation core, allowing a large number of cores to correct video simultaneously.

4.3 Fixed Point Implementation

Many low power embedded platforms do not have hardware that allows for efficient operations on floating point numbers. A fixed point implementation of part of the algorithm was created to allow the system to be run on a parallel embedded system with no hardware to perform floating point operations. This work was focused around the calculation of the distortion model parameters using LM optimization. The extraction of feature points using fixed point math was not investigated in this work. The motion filtering and image warping stages were implemented by simply replacing floating point numbers with 32 bit fixed point numbers with good results. A semi-automated algorithm was developed to generate code to calculate full frame model parameters.

The fixed point implementation was intended to be used alongside the parallelization method discussed in Section 4.2.3. Each processing core calculates its own Hessian and Jacobian matrices using fixed point operations. When adding matrices together, they are both converted to floating point, and the linear system solving is performed using floating point LU decomposition to increase stability and accuracy.

Two large difficulties exist in implementing a floating point algorithm in fixed point: avoiding overflow while still maintaining numbers large enough that numerical precision does not suffer. 32 bit numbers were used in this fixed point implementation (the register size of the hardware for which this work was originally targeted), with 16 bits used for the integer portion of the number, and 16 bits of decimal precision. This produces a maximum range of numbers from -32768 to 32767.99998. The step between adjacent numbers is constant throughout this range, with a distance of approximately 3.1e-5 between adjacent values. Floating point numbers are able to represent a much larger range of numbers, and maintain consistent relative accuracy throughout their range. The fixed point implementation must keep numbers within the allowed range at all times, but also keep numbers as large as possible to increase relative accuracy.

Fixed point number sizes were kept reasonable through two methods: model parameters were normalized such that parameters averaged absolute values near 1, and intermediate values were scaled up or down. All normalization and scaling values were chosen to be powers of two. This can be implemented efficiently for both fixed point and floating point numbers: fixed point numbers can be arithmetically shifted left or right, and the exponent component of a floating point number can be added or subtracted from.

The full frame motion model was changed from the model presented in Section 3.3.1 to a more conventional affine model. While this does not model the camera motion as accurately, the parameter p_{r1} was found to be too small to be accurately used with fixed point numbers with 16 bits of decimal precision. The output x and y positions for the affine model used are shown in Equations 4.1 and 4.2 respectively

$$x_2 = (1 + p_{m1})x_1 + p_{m2}y_1 + p_{dx} \tag{4.1}$$

$$y_2 = p_{m3}x_1 + (1 + p_{m4})y_1 + p_{dy}. \tag{4.2}$$

Parameter normalization values were calculated by taking the inverse of the mean absolute parameter values over several frames. Rounding to the nearest powers of two, the values shown in Table 4.1 were obtained.

Next individual expressions had to be properly scaled to prevent overflow while maximizing numerical accuracy. To achieve this, a brute force approach was taken to achieve ideal variable scaling. An approach was taken starting from the first temporary variable, and swept through all temporary variables and then the expressions directly adding to the Jacobian and Hessian matries. The approach

Parameter	Scaling Value
p_{dx}	1
p_{dy}	1
p_{m1}	2^{10}
p_{m2}	2^{10}
p_{m3}	2^{10}
p_{m4}	2^{10}

TABLE 4.1: Parameter scaling values for fixed point implementation of the LM parameter optimization of full frame model parameters.

taken for each variable is explained below.

To determine optimal scaling values for operands in the calculation of a single temporary variable, a sweep of scaling values are tested for each operand. At each combination of scaling values, the accuracy of the temporary variable is compared to the value obtained using floating point math. Scaling values are swept in increasing order. Moving in this direction, a new solution is taken if the relative error decreases by at least 1% over the previous best solution. This results in a relatively accurate solution where values stay as far as possible from overflow conditions. An example of this process is shown in Table 4.2.

This process is iteratively repeated for each temporary variable and expression contributing to the Hessian and Jacobian matrices. In the case of temporary variables, when the optimal total scaling value is not 1, instances of that temporary variable are scaled by the inverse of the total scaling value. For expressions adding to the Jacobian or Hessian matrices, those matrix elements are scaled by the inverse of the total scaling value after being converted to floating point before solving the linear system.

Scaled Expressions	Total Scaling	Relative Error
$p_{m2}(y1 >> 4)$	2^{-4}	$9.3e-5$
$(p_{m2} >> 1)(y1 >> 3)$	2^{-4}	$7.3e-4$
$(p_{m2} >> 2)(y1 >> 2)$	2^{-4}	$7.3e-4$
$(p_{m2} >> 3)(y1 >> 1)$	2^{-4}	$7.3e-4$
$(p_{m2} >> 4)y1$	2^{-4}	$7.3e-4$
$p_{m2}(y1 >> 2)$	2^{-2}	$9.0e-5$
$(p_{m2} >> 1)(y1 >> 1)$	2^{-2}	$7.4e-4$
$(p_{m2} >> 2)y1$	2^{-2}	$7.4e-4$
$p_{m2}y1$	1	$8.9e-5$
$p_{m2}(y1 << 2)$	2^{2}	$8.9e-5$
$(p_{m2} << 1)(y1 << 1)$	2^{2}	$8.9e-4$
$(p_{m2} << 2)y1$	2^{2}	$8.9e-4$
$p_{m2}(y1 << 4)$	2^{4}	$8.9e-5$
$(p_{m2} << 1)(y1 << 3)$	2^{4}	$8.9e-5$
$(p_{m2} << 2)(y1 << 2)$	2^{4}	$8.9e-5$
$(p_{m2} << 3)(y1 << 1)$	2^{4}	$8.9e-5$
$(p_{m2} << 4)y1$	2^{4}	$7.3e-4$
$p_{m2}(y1 << 6)$	2^{6}	1.9
$(p_{m2} << 1)(y1 << 5)$	2^{6}	0.73
$(p_{m2} << 2)(y1 << 4)$	2^{6}	$8.9e-5$
$(p_{m2} << 3)(y1 << 3)$	2^{6}	$8.9e-5$
$(p_{m2} << 4)(y1 << 2)$	2^{6}	$8.9e-5$
$(p_{m2} << 5)(y1 << 1)$	2^{6}	$8.9e-5$
$(p_{m2} << 6)y1$	2^{6}	$8.9e-5$

Table 4.2: Testing to find optimal variable scaling values to calculate a temporary variable equal to $p_{m2}y1$. The expression $p_{m2}(y1 >> 2)$ is chosen as optimal, as it combines a low error with the largest average value. Note the overflow when y1 is shifted left by five or more places it overflows (y1 can be up to than 1080, multiplied by 32 or 64 can exceed 2**15 or 32767), resulting in large errors.

Chapter 5

Results

5.1 Code optimization speedup

Code optimized to perform iterations of LM optimization was found to be much faster than non-optimized generated code. Table 5.1 shows the results of speeding up the different model fitting algorithms. Note that there is a constant time associated with iterating through feature points that cannot be removed through this type of optimization. In each case a significant speedup was obtained using this code optimization.

Fitting Algorithm	Standard time (us)	Optimized time(us)	Speedup
Full Frame Motion	1804	90	20x
Sinusoidal Translation	361	47	7.7x
Full Model Tuning	75,000	1230	61x
Total (6 Sinusoids)	78970	1602	49x

Table 5.1: Speedup of LM iterations from code optimizations. Each standard iteration time was averaged over 1000 fits, while the optimized iteration times were averaged over 10,000 fits to obtain a more precise answer.

These optimizations allowed the model fitting time to be reduced to approximately 24ms per frame on a single core of an i7 processor. This includes performing 10 iterations of the full frame motion fit, 10 iterations on each of the six sinusoids, and 10 iterations of the full tuning fit.

5.2 Scalability

The proposed algorithm has four main sections: feature extraction, distortion transfer function calculation, motion smoothing and image warping. Here the scalability of each section is discussed with emphasis on the increasing video resolution appearing in electronics.

5.2.1 Feature Extraction

The feature extraction step is expensive to implement on a Central Processing Unit (CPU), and increases in complexity as video resolution increases. Two main steps are involved here: the computation of eigenvalues of the covariance matrix of each pixel, and the sorting of eigenvalues of the matrices. The complexity of the eigenvalue calculation scales linearly with the number of pixels in the image (ignoring edges which are negligible in a high resolution image). This can be easily implemented in parallel, and is a simple task which makes it well suited for implementation on an FPGA or ASIC if high performance is needed. The sorting process scales with complexity $\mathrm{O}\big(n_{pixels}\log\big(\frac{n_{pixels}}{n_{subImages}}\big)\big)$, which is improved by using the image division algorithm presented in Section 3.2.3. This complexity is manageable thanks to highly optimized sorting routines.

5.2.2 Distortion Transfer Function Calculation

The distortion transfer function calculation's complexity does not depend at all on the input video size. In much larger videos however, it may be useful to use greater than two sinusoidal distortion functions in one or more directions, or to use more feature pairs to match frames. Therefore, the complexity of the algorithm with respect to those parameters will be discussed.

The sinusoidal transfer functions are entirely self contained, and the overall time required to fit an arbitrary number of sinusoids is directly proportional to the number of sinusoids to be fit. The final model tuning, which already takes approximately 75% of transfer calculation time will become more costly as the number of sinusoids increases. The computation of the Hessian and Jacobian matrices will scale as $\mathrm{O}\left(n_{params}^2\right)$. The matrix inversion required scales as $\mathrm{O}\left(n_{params}^3\right)$ using the typical Gauss-Jordan elimination, though asymptotically more efficient algorithms have been developed. At the current number of parameters the matrix inversion time is negligible compared to the calculation of the Hessian and Jacobian matrices, taking approximately 3% of a single iteration time. Complexity with respect to the number of feature pairs is linear.

5.2.3 Motion Smoothing and Image Warping

Motion smoothing and image warping both scale linearly with the number of pixels. Both operations are fully parallelizable and could be efficiently implemented in parallel on a Graphics Processing Unit (GPU), FPGA or ASIC.

5.3 Artificially distorted, noisy video

During development, an artificial test video was created to allow for quantitative error analysis. This video was created by shifting and distorting a single image with added noise. This test video was not meant to provide a quantitative comparison of algorithms that would provide a real world error estimate, but rather a simple test for development purposes in a near-ideal environment.

A 720p (1280x720) video was created using a starting image with dimensions 1600x1067 pixels. By applying different transformations to the starting image before cropping, frames are generated and stored in a video format. To model true rolling shutter distortion, the rotation and translation parameters of the transformation are modified after every row of pixels is copied. Additionally, a delay is simulated between the final row of pixels in a frame and the first row of pixels in the next frame. Random gaussian noise is added to each pixel with a standard deviation of 20 (using 1-byte RGB values).

Translation and rotation parameters are modified using a combination of two sinusoids and a source of random noise. A slow moving sinusoid was used to simulate full frame camera motions with an amplitude of 4.0 pixels and 3.6 pixels in the x and y directions respectively, and a rotational magnitude of 0.0001 Rad. Frequencies of $0.002\frac{Rad}{row}$, $0.0015\frac{Rad}{row}$ and $0.001\frac{Rad}{row}$ were used in the x, y and rotational axis respectively. Higher frequency sinusoids were used to simulate rolling shutter distortion. Amplitudes of 3.0 pixels, 2.3 pixels and 0.01Rad were used for the x, y and rotational directions with frequencies of $0.007\frac{Rad}{row}$, $0.011\frac{Rad}{row}$ and $0.004\frac{Rad}{row}$ respectively. At each row, the translation and rotation were corrupted by gaussian random noise with standard deviations of 0.1 pixels, 0.05 pixels and 0.0002 Rad in the x, y and rotational directions. All phases were initialized with random values.

To evaluate the quality of distortion calculations, the true distortions relative to the first frame were stored. To reduce computation time and storage, the true distortions were stored only for pixels falling on a 5x5 grid. The true x and y shifts were stored for these values, and the estimate of these shifts was computed using the distortion transfer function. The RMSE was used to compute the overall accuracy, and was averaged over the first 50 frames.

The RMSE was compared between no distortion correction, the full frame portion of the model and the full model. The results are shown in Table 5.2.

Distortion Model	Average RMSE
No Correction	5.71
Full Frame Correction	1.82
Complete Distortion Model	0.058

Table 5.2: Artificial test video results. A single image was distorted to produce frames, and the calculated distortion was compared to the actual distortion on a pixel by pixel basis.

5.4 Real world video results

The stabilization algorithm was tested on several real world videos, each exhibiting different types of rolling shutter distortion and each with their own difficulties. Several videos and the result of their stabilization are described below.

5.4.1 Parliament Video

The Parliament video is an aerial shot of the Parliament buildings taken with a GoPro HD HERO2 mounted on a DJI Phantom quadcopter. This was recorded at 60fps at 1080p resolution. Constant rolling shutter wobble is seen in the original video due to the vibration of the quadcopter from the spinning rotor blades. There is some translational motion of the camera, but at a low enough rate that shear is not a significant issue. A screenshot of the video can be seen in Figure 5.1.

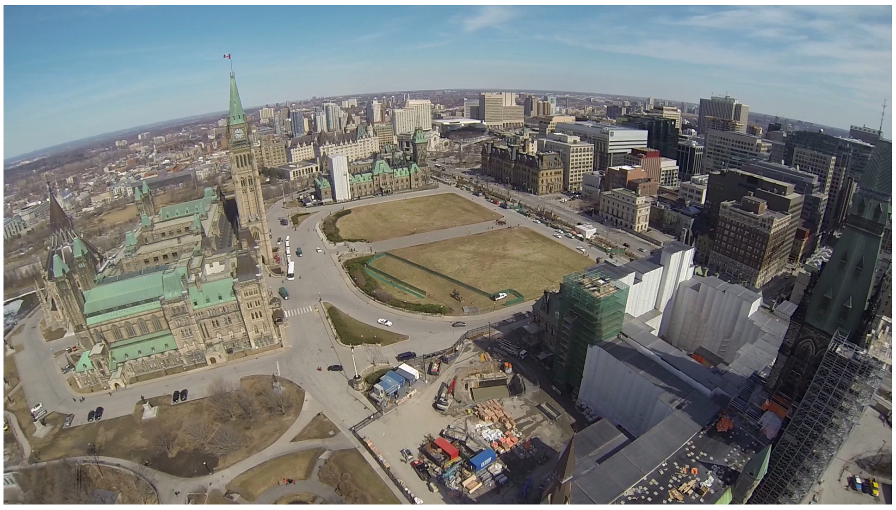


Figure 5.1: Screenshot of the Parliament video

The Parliament video has many strong corners distributed fairly evenly throughout the frame of view. The only exception to this is the small portion of sky at the top of the screen. The sinusoid based distortion model works well on this video due to the oscillatory motion of the quadcopter that can be modeled well using only two sinusoids. The algorithm performed well on this video, with small artifacts visible in the sky and the tower that extends into the sky, where compounded errors cause a slight bend in the tower. The correction of a significant horizontal translation in this video causes the curvature of the horizon (caused by a fisheye lens, not the curvature of the earth) to visibly change during this video. This effect, and a possible solution is discussed further in Section 6.2.6.

5.4.2 Driving Video

The Driving video was taken from the front seat of a car while driving down a country road. The camera used was a hand held LG Nexus 4 cellphone, and was recorded at 30 fps with 720p resolution. This example has more overall camera motion than the Parliament video, but the camera motion is much slower . Dirt and reflections on the windshield occlude small portions of the scene, and provide outlier feature points when they are tracked. This video also violates the assumption that camera motion is purely rotational, instead we are moving forward quickly. Sky covers approximately half of the screen, providing few high quality feature points. A screenshot of this video can be seen in Figure 5.2.

The full frame motion model accurately captures the shakiness of the camera during the Driving video. Low frequency rolling shutter skew can be seen at the corners of the screen, and are corrected using the distortion model. Some distortion can be seen in the road, especially noticeable is the movement of the dotted line. This is caused by the movement of the camera into the scene, and the model's inability to capture large camera movements relative to the scene.

FIGURE 5.2: Screenshot of the Driving video

5.4.3 Walking Video

The Walking video was taken from a handheld cell phone camera while walking down a path in the forest. This was taken using the LG Nexus 4 cellphone, recorded at 30fps with 1080p held in the portrait direction. A single large camera shake can be seen periodically with every step taken down the path. When full frame stabilization is applied to this video, high frequency distortions are still visible at each step. A portion of the video frames show the sky, but otherwise high quality tracking points are generally available throughout the frame. A screenshot can be seen in Figure 5.3

The sharp motion resulting from each walking step was removed from the video, with no visible artifacts produced. Small residual motions are seen at each step resulting from the parallax motion of the foreground trees compared to the background. The algorithm was also able to rotate the video so that pixels are scanned top to bottom, and then rotate it back, allowing the exact same distortion model to be used with a video taken in portrait.

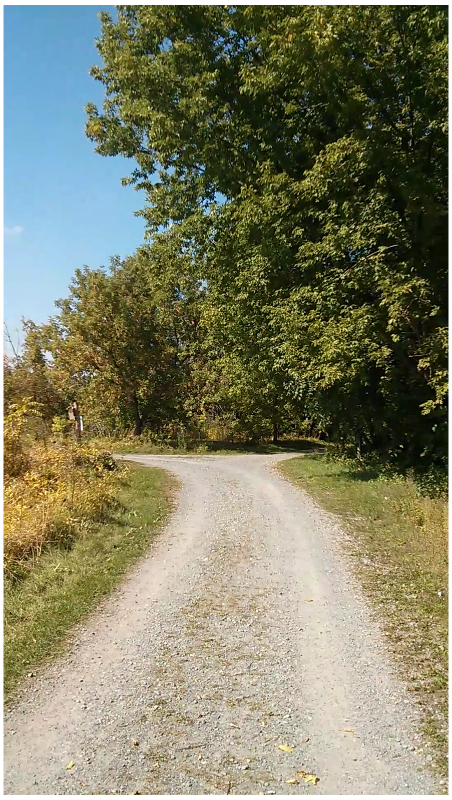


FIGURE 5.3: Screenshot of the Walking video

5.4.4 House Video

A video was recorded indoors, intentionally shaking the camera while slowly panning around the room. This was recorded using the LG Nexus 4 cellphone at 30fps with 720p quality. Due to the low lighting and small image sensor in the phone, motion blur can be seen due to the rapid movements and long exposure time. Large walls in the frame provide areas of low contrast which few high quality features to be tracked. A screenshot can be seen in Figure 5.4, showing the motion blur found in many frames.

The algorithm is able to filter the majority of the camera motion and frame distortion, but some rolling shutter artifacts are still visible in the output video. The motion blur becomes obvious once the corresponding motion has been removed from the video, and makes the output look somewhat unnatural.

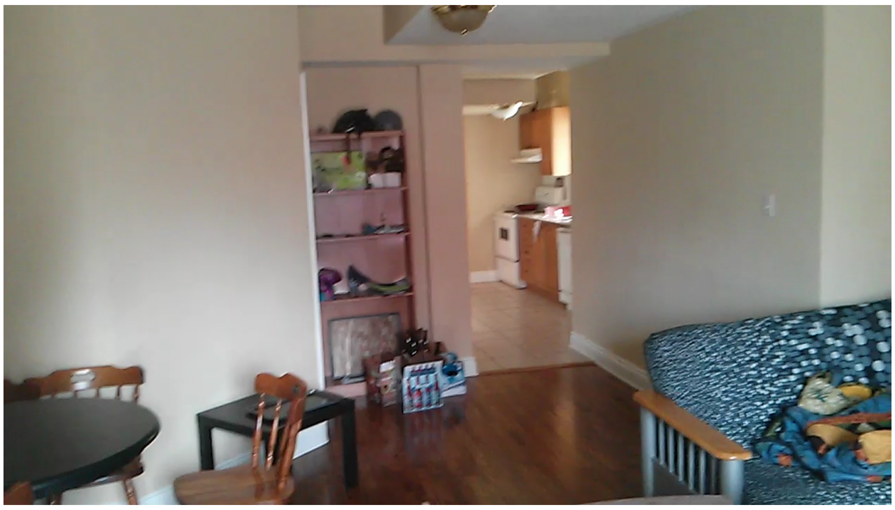


FIGURE 5.4: Screenshot of the House video

5.4.5 Street Video

A video was recorded by holding an LG Nexus 4 cellphone while riding a bicycle down the street in a neighborhood. This was recorded at 30fps and 1080p resolution. The difficulty in this video is caused by the camera motion causing objects to disappear and reappear from behind other objects. A screenshot can be seen in Figure 5.5.

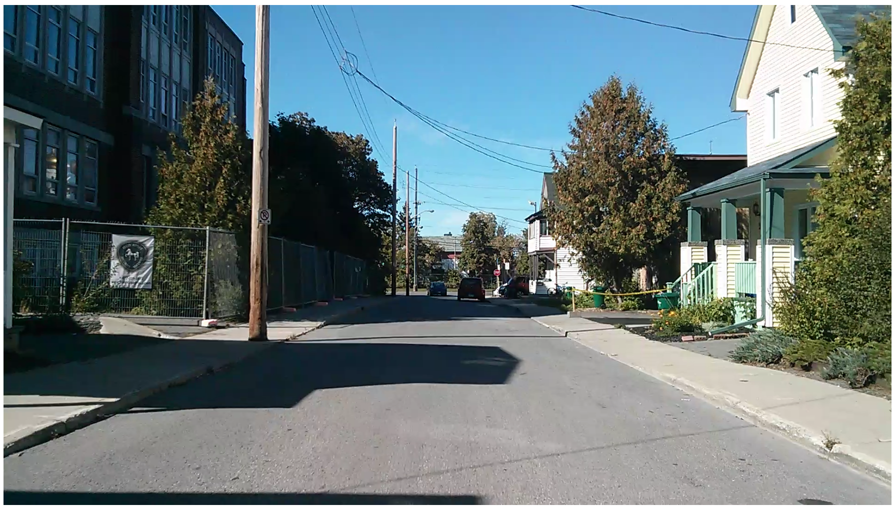


FIGURE 5.5: Screenshot of the Street video

The result of the algorithm on the Street video is quite poor. The algorithm fails to deal with the moving of objects in the frames due to the camera motion. Objects are often moving behind one another, but the outlier robust model optimization is able to deal with this. The horizontal stretching of features towards the edge of the screen causes the x-translation sinusoid to fit improperly. To improve the quality of the video result a 3D camera motion model could be used, as discussed in Section 6.2.1.

5.5 Fixed Point Implementation

The fixed point implementation of the full frame motion stabilization was not successful. Fixed point variables frequently overflowed even with the calculated scaling values. In many other cases, inaccuracies caused by the fixed point numbers resulted in incorrect model parameters.

The fixed point scaling parameters were tuned using a selection of frames from the Parliament video. In the majority of these frames the calculated model parameters matched the parameters calculated using floating point numbers. However, in some of these cases the resulting parameters were not accurate, and in other videos the results were poor.

Chapter 6

Conclusion

6.1 Contributions to Research

An algorithm has been developed that is able to identify and remove rolling shutter distortion from videos captured with a CMOS video camera. Additionally, some basic full frame camera stabilization is also provided. This algorithm can be efficiently implemented in parallel for improved performance, especially when power is a constraint.

The feature extraction algorithm of Grundmann *et al.* [1] using sub-images to provide more uniform features was extended by determining an optimal number of sub-images to use. This allows for an optimal balance between feature quality and feature uniformity to be found. A recursive feature extraction algorithm was also proposed that would determine optimal sub-image depth at runtime.

A sinusoidal based distortion model was developed, and tuned to work especially well for videos taken from a vehicle where motors, propellers or wheels create oscillatory camera motion. This model can easily be modified to incorporate any number of oscillation frequencies, allowing it to model arbitrary high frequency camera motion.

An efficient fitting method was developed to calculate model parameters in the presence of outliers. An iterative approach is used to fit re-descending m-estimators to data with decreasing kernel parameter to iteratively decrease the significance of outliers. To the author's knowledge, this approach has not been used before.

An automatic code optimization algorithm was developed to speed up iterations of LM optimization used to fit model parameters. Temporary variables are automatically created and pre-calculated for frequently evaluated sub-expressions. This code requires only the specification of a cost function and a list of parameters to generate a C code file containing optimized code, and tests the code for any accuracy decrease due to numerical issues. This addition resulted in speed increases between 7.7X and 61X for different problems it was applied to.

The stabilization algorithm presented here scales well with increases in image sizes. The only step that scales with the number of image pixels is the feature extraction step, which can be easily implemented in hardware. Additionally, the feature extraction algorithm used here requires less sorting complexity than standard feature extraction, since pixel eigenvalues are only sorted within small sub-images. All other stages of processing are constant in complexity as image sizes increase. Many other stabilization algorithms employ a separate motion model for several rows of pixels together, resulting in $\mathrm{O}\left(\sqrt{(n)}\right)$ scaling with respect to the number of pixels in images (assuming image proportions remain approximately constant). A particular video shot at different resolutions can be stabilized equally well using the same distortion model and number of sinusoids.

6.2 Future work

6.2.1 3D Camera Motion Model

The Street video shows a fault of the video stabilization algorithm: the violation of the assumption that camera motion is purely rotational results in poorly stabilized video. Objects enter and leave the frame of view as the camera moves forward into the scene. Additionally, the different depths of the scene for different image rows means that neither the full frame motion model described in Section 3.3.1 nor an affine transformation can accurately model the full frame camera motion. Instead, a 3D motion model is required to accurately stabilize the full frame camera motion before the rolling shutter distortions can be removed. Liu and Jin [19] developed a full frame camera stabilization algorithms able to calculate 3D motion and provide a smooth camera path in three dimensions. Such an algorithm could replace the full frame camera motion model used in this work to more accurately calculate camera motion. After the full frame motion is subtracted from pairs of feature points, the sinusoidal rolling shutter distortion model could be used to remove rolling shutter skew and wobble.

6.2.2 Bicubic Pixel Interpolation

Bilinear interpolation was used to calculate an image color located between pixels. This produces a slightly blurred image, but does so efficiently compared to other interpolation methods. Bicubic interpolation is another common interpolation method that uses the local 4x4 grid of pixels to interpolate a single value with a cubic convolution algorithm [39]. This method requires more computation than bilinear interpolation, but preserves fine detail better and increases edge contrast.

Bicubic interpolation, along with bilinear interpolation can easily be implemented in a more efficient manner than on a single CPU. Each pixel requires a computation, making it a parallel operation. This makes it ideal for implementation on a GPU, FPGA or ASIC.

6.2.3 M-estimator width reduction

The choice to use a predefined number of logarithmically spaced steps for the Welsch kernel parameter $w(x)$ was fairly arbitrary, but it was reasonably efficient and produced good results. Several possible alternatives to reduce $w(x)$ are discussed below.

The Welsch kernel associates the cost $w^2(1 - exp(-(\frac{x}{w})^2))$ for a point with error x and a kernel parameter w. This means that for the contribution of outliers to be reduced linearly, we should space out values of w such that the square root of parameters are logarithmically spaced. This would result in more parameters close to the final parameter value where the trust region of the global minimum tends to be smaller.

Though LM optimization was used instead of the Newton Raphson method for increased stability, if a situation is encountered where a large λ parameter is required for stability, we likely will not converge on the correct solution within the number of steps allowed. Allowing the number of steps to vary could produce more efficient optimization in most cases while allowing more difficult problems to take additional steps. The downside to this approach is that frames will no longer take a constant processing time, which can cause problems in a real time system.

Algorithms to determine the trust region radius have been developed for use in nonlinear programming [40]. This allows the radius of convergence of an algorithm such as the Newton Raphson method to be determined, and is often used with

LM optimization to update λ values. Such a calculation is performed by testing the agreement between the true objective function and the first or second order approximation used by trust region optimization methods. Such a method could be used to ensure that the w parameter varies at an optimal rate to converge reliably with as few iterations as possible. At each iteration a bisection search could be used to test the size of the trust region and find the largest value of w from which we could expect to continue to converge on the global maximum.

6.2.4 Motion Smoothing

The EMA filtering used to provide smooth camera motion is meant only to provide a simple and fast method to evaluate the distortion model. Two significant improvements could be made to provide smoother camera motion and reduce processing time.

To speed up computation, a mesh could be used to avoid computing pixel translations for every single pixel. At the output phase, the translation of a single point could calculated using linear or cubic interpolation of the grid. Since the sinusoidal model frequencies are limited to $0.1 Rad/row$, the largest errors are small provided the grid is dense, as shown in Figure 6.1. While the errors grow quickly, the complexity of filtering is $\mathrm{O}\left(\frac{n_{pixels}}{gridSize^2}\right)$ since a single tracked pixel would provide the shift for a square with edge length gridSize. This results in a small grid size causing a significant performance increase.

The rolling shutter distortion can be separated from the full frame motion and different smoothing procedures can be applied to each. The current decaying EMA works well for distortions and can be applied efficiently, but does not provide smooth camera motion when used on full frame movements. The L1 optimal camera paths algorithm proposed by Grundmann *et al.* [11] could replace the EMA algorithm used here and provide for more professional, visually pleasing camera

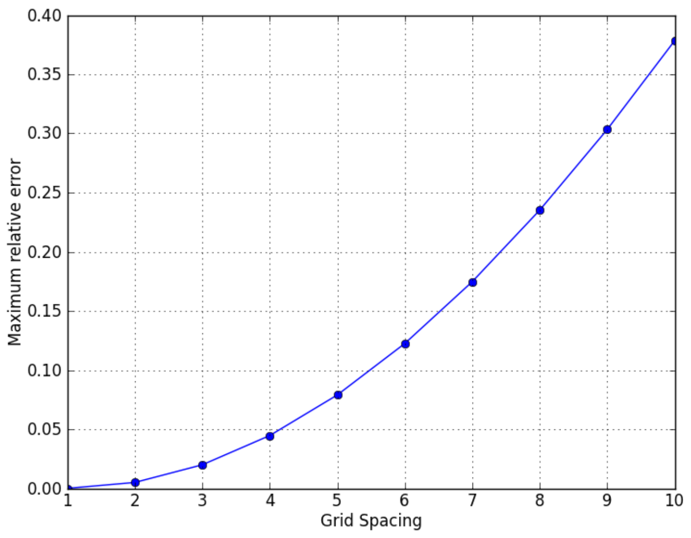


FIGURE 6.1: Maximum Relative Error vs Grid spacing if a grid were used to speed up motion smoothing. Note that this assumes a sinusoidal frequency at the limit that was allowed to be fit, and in most cases it would be much smaller.

motions. Camera motion is calculated by minimizing the L1 norm of the sum of the first, second and third derivatives of the camera motion. Weights applied to each derivative allow different camera paths to be taken. In this algorithm, a crop window is predetermined, and linear programming is used to find an optimal path under the crop window constraint.

6.2.5 Removal of Motion Blur

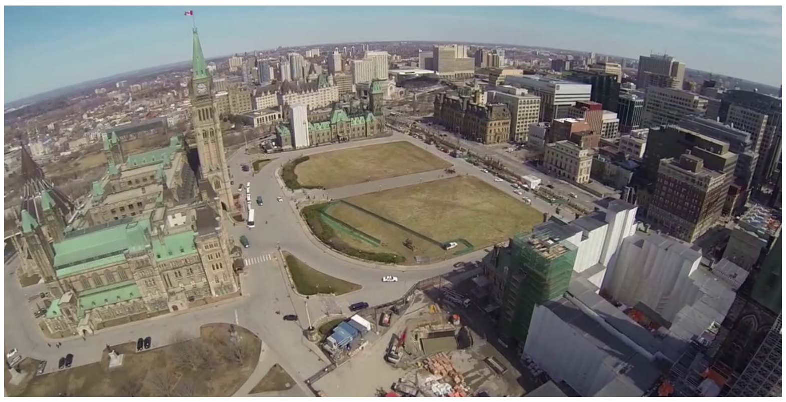
High speed camera motion in low light settings will often result in motion blur of frames due to the movement of the camera during exposure time. In a rolling shutter camera, this motion blur can even occur due to the high frequency wobbles of the camera during exposure, resulting in blurring that is spatially variant in each frame. Stabilized video with smooth camera motion may show blurring.

Algorithms to remove non-uniform motion blur have been developed to remove spatially variant motion blur from a single image [41]. The focus of this and other similar work is generally to estimate a blur kernel for different objects within a single image. In the case of video frame deblurring, the blurring motion can be extracted from the distortion transfer function. The blurring direction can be extracted directly from the transfer function. The size of the blur could be found directly in a single image similar to image motion deblurring algorithms, and averaged over several frames to achieve a more accurate result. Alternatively, the size of the blur could be calculated from the magnitude of the distortion and the exposure time, which can be estimated from video using an algorithm developed by Oth *et al.* [42]. Once a blur kernel has been estimated, a spatially variant Wiener filter could be applied to each frame to remove motion blur.

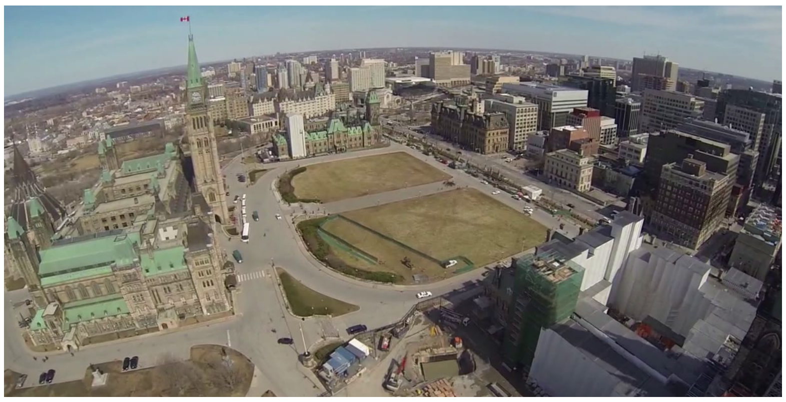
6.2.6 Fisheye lens correction

Many modern cameras use a fisheye lens to provide a wide panoramic image. This type of lens will distort an image to map a large angle of view into a rectangular frame. Depending on the lens used, this can result in straight lines becoming warped when they are not in the center of the frame. This effect can be seen in the Parliament video in Figure 5.1. The curvature of the horizon is not due to the curvature of the earth but instead the fisheye lens.

When transformations are applied to a video shot though a fisheye lens, the curvature of the lens should be corrected for. Otherwise videos will be distorted different amounts depending on the frame. This can be seen in the stabilized Parliament video. Two frames are shown in Figure 6.2 before and after a vertical translation of the frame to keep the video stable.



(A) Fisheye horizon curvature before panning.



(B) Fisheye horizon curvature after panning.

FIGURE 6.2: Fisheye distortion caused by stabilization. The horizontal translation required to stabilize the video results in different curvatures of the horizon caused by the fisheye lens. When the camera pans quickly, the change in curvature is not as noticeable. While this is difficult to see in still images, the earth is slightly more curved in Figure 6.2a than Figure 6.2b.

List of References

[1] M. Grundmann, V. Kwatra, D. Castro, and I. Essa, "Effective calibration free rolling shutter removal," *IEEE ICCP*, 2012.

[2] M. Bigas, E. Cabruja, J. Forest, and J. Salvi, "Review of {CMOS} image sensors," *Microelectronics Journal*, vol. 37, no. 5, pp. 433 – 451, 2006. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0026269205002764

[3] A. J. P. Theuwissen, "Cmos image sensors: State-of-the-art," *Solid-State Electronics*, 2008.

[4] E. Fossum, "Cmos image sensors: electronic camera on a chip," in *Electron Devices Meeting, 1995. IEDM '95., International*, 1995, pp. 17–25.

[5] N. Bock, A. Krymski, A. Sarwari, M. Sutanu, N. Tu, K. Hunt, and M. Cleary, "A wide-vga cmos image sensor with global shutter and extended dynamic range," Micron Imaging, Micron Technology Inc, Tech. Rep., 2005.

[6] S. Baker, E. Bennett, S. B. Kang, and R. Szeliski, "Removing rolling shutter wobble," in *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, 2010, pp. 2392–2399.

[7] D. Bradley, B. Atcheson, I. Ihrke, and W. Heidrich, "Synchronization and rolling shutter compensation for consumer video camera arrays," in *Computer*

Vision and Pattern Recognition Workshops, 2009. CVPR Workshops 2009. IEEE Computer Society Conference on, 2009, pp. 1–8.

[8] C. Corporation, "Cv2201 consumer pdk," 2013. [Online]. Available: http://www.cognivue.com/docs/PB-10259-01-00%20CV2201%20PDK_Consumer%20Product%20Brief_v2.pdf

[9] STABiLGO, "Stabilgo kickstarter project," 2013. [Online]. Available: http://www.kickstarter.com/projects/1969395374/stabilgo-ready-steady-go

[10] Y. Matsushita, E. Ofek, W. Ge, X. Tang, and H.-Y. Shum, "Full-frame video stabilization with motion inpainting," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 28, no. 7, pp. 1150–1163, 2006.

[11] M. Grundmann, V. Kwatra, and I. Essa, "Auto-directed video stabilization with robust l1 optimal camera paths," in *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*, 2011, pp. 225–232.

[12] K. Ratakonda, "Real-time digital video stabilization for multi-media applications," in *Circuits and Systems, 1998. ISCAS '98. Proceedings of the 1998 IEEE International Symposium on*, vol. 4, 1998, pp. 69–72 vol.4.

[13] A. Litvin, J. Konrad, and W. C. Karl, "Probabilistic video stabilization using kalman filtering and mosaicking," *Proceedings of the SPIE Conference on Electronic Imaging*, pp. 663–674, 2003.

[14] Y.-S. Wang, F. Liu, P.-S. Hsu, and T.-Y. Lee, "Spatially and temporally optimized video stabilization," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 19, no. 8, pp. 1354–1361, 2013.

[15] Y.-G. Kim, V. Jayanthi, and I.-S. Kweon, "System-on-chip solution of video stabilization for cmos image sensors in hand-held devices," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 21, no. 10, pp. 1401–1414, 2011.

[16] A. Karpenko, D. Jacobs, J. Baek, and M. Levoy, "Digital video stabilization and rolling shutter correction using gyroscopes," Stanford University Computer Science Tech Report CSTR 2011-03, Tech. Rep., 2011.

[17] G. Hanning, N. Forslow, P.-E. Forssen, E. Ringaby, D. Tornqvist, and J. Callmer, "Stabilizing cell phone video using inertial measurement sensors," in *Computer Vision Workshops (ICCV Workshops), 2011 IEEE International Conference on*, 2011, pp. 1–8.

[18] C.-K. Liang, L.-W. Chang, and H. Chen, "Analysis and compensation of rolling shutter effect," *Image Processing, IEEE Transactions on*, vol. 17, no. 8, pp. 1323–1330, 2008.

[19] F. Liu, M. Gleicher, H. Jin, and A. Agarwala, "Content-preserving warps for 3d video stabilization," *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2009*, 2009.

[20] J.-Y. Bouguet, "Pyramidal implementation of the lucas kanade feature tracker description of the algorithm," Intel Corporation, Microprocessor Research Labs, Tech. Rep., 2000.

[21] C. Tomasi and T. Kanade, "Detection and tracking of point features," Carnegie Mellon University, Tech. Rep., 1991.

[22] J. Shi and C. Tomasi, "Good features to track," in *Computer Vision and Pattern Recognition, 1994. Proceedings CVPR '94., 1994 IEEE Computer Society Conference on*, 1994, pp. 593–600.

[23] Z. Osman, F. Hussin, and N. Ali, "Hardware implementation of an optimized processor architecture for sobel image edge detection operator," in *Intelligent and Advanced Systems (ICIAS), 2010 International Conference on*, 2010, pp. 1–4.

[24] B. D. Lucas and T. Kanade, “An iterative image registration technique with an application to stereo vision,” in *Proceedings of Imaging Understanding Workshop*, 1981, pp. 121–130.

[25] D. Lowe, “Object recognition from local scale-invariant features,” in *Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on*, vol. 2, 1999, pp. 1150–1157 vol.2.

[26] H. Bay, A. Ess, T. Tuytelaars, and L. V. Gool, “Speeded-up robust features (surf),” *Computer Vision and Image Understanding*, vol. 110, no. 3, pp. 346 – 359, 2008, ¡ce:title¿Similarity Matching in Computer Vision and Multimedia¡/ce:title¿. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1077314207001555

[27] W. L. Briggs and V. E. Henson, *The DFT: An Owners' manual for the Discrete Fourier Transform.* Library of Congress Cataloging-in-Publication Data, 1995.

[28] J. Matas, “Ransac in 2011 (30 years after),” *Computer Vision and Pattern Recognition (CVPR)*, 2011.

[29] R. C. B. Martin A. Fischler, “Random sample consensus: A paradigm for model fitting with apphcatlons to image analysis and automated cartography,” *Communications of the ACM*, pp. 381–395, 1981.

[30] K. G. Derpanis, “Overview of the ransac algorithm,” York University, Tech. Rep., 2010.

[31] O. Chum, J. Matas, and J. Kittler, “Locally optimized ransac,” in *Pattern Recognition*, ser. Lecture Notes in Computer Science, B. Michaelis and G. Krell, Eds. Springer Berlin Heidelberg, 2003, vol. 2781, pp. 236–243. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-45243-0_31

[32] O. Chum and J. Matas, "Matching with prosac - progressive sample consensus," in *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, vol. 1, 2005, pp. 220–226 vol. 1.

[33] M. Zuliani, "Ransac for dummies," University of California, Santa Barbara, Tech. Rep., 2012.

[34] Z. Zhang, "Parameter estimation techniques: A tutorial with application to conic fitting," *Image and Vision Computing*, vol. 15, pp. 59–76, 1997.

[35] P. J. Huber, *Robust Statistical Procedures.* CBMS-NSF regional conference series in applied mathematics, 1996.

[36] ——, "Robust estimation of a location parameter," *The Annals of Mathematical Statistics*, vol. 35, no. 1, pp. pp. 73–101, 1964. [Online]. Available: http://www.jstor.org/stable/2238020

[37] G. Shevlyakov, S. Morgenthaler, and A. Shurygin, "Redescending m-estimators," *Journal of Statistical Planning and Inference*, vol. 138, no. 10, pp. 2906 – 2917, 2008. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0378375807004120

[38] M. K. Transtrum and J. P. Sethna, "Improvements to the levenberg-macquardt algorithm for nonlinear least-squares minimization," Department of Computational Physics, Cornerll University, Tech. Rep., 2012.

[39] R. Keys, "Cubic convolution interpolation for digital image processing," *Acoustics, Speech and Signal Processing, IEEE Transactions on*, vol. 29, no. 6, pp. 1153–1160, 1981.

[40] A. Sartenaer, "Automatic determination of an initial trust region in nonlinear programming," *SIAM Journal on Scientific Computing*, 1995.

[41] S. Cho, Y. Matsushita, and S. Lee, “Removing non-uniform motion blur from images,” in *Computer Vision, 2007. ICCV 2007. IEEE 11th International Conference on*, 2007, pp. 1–8.

[42] L. Oth, P. Furgale, L. Kneip, and R. Siegwart, “Rolling shutter camera calibration,” in *Computer Vision and Pattern Recognition (CVPR), 2013 IEEE Conference on*, 2013, pp. 1360–1367.