Machine Learning Assisted Non-Rigid Surface Tracking in Biological Systems

by William Henry Hempstead, M.S.B.E.

July 2020

Directed by: Zhen Zhu, Ph.D.

Department of Engineering

Occlusions, obstructions, and lighting changes that occur in a camera's field-of-view (FOV) during a medical procedure can cause tracking algorithms to lose track of a particular region-of-interest (ROI). Various approaches to reacquire the tracking of rigid objects have been developed, however, the non-rigid nature of biological structures requires more complicated approaches. The purpose of this research is to improve the performance of an existing non-rigid tracking algorithm under these types of adverse conditions. This existing algorithm was previously shown to be accurate and efficient under ideal conditions but exhibited a high rate of tracking failures due to the aforementioned occlusions, obstructions, and lighting changes. To improve tracking under these conditions, a tissue motion machine learning model was developed to provide predictions of future ROI grid motion. The combination of this machine learning technique along with various improvements to the base algorithm was shown to greatly reduce the number of tracking resets and allow the tracking grid to briefly follow an expected motion pattern during a simulated occlusion.

Machine Learning Assisted Non-Rigid Surface Tracking in Biological Systems

A THESIS

Presented to the Faculty of the Department of Engineering

College of Engineering and Technology

East Carolina University

In Partial Fulfillment of the Requirements For the Degree

Master of Science in Biomedical Engineering

by

William Henry Hempstead

July, 2020

Machine Learning Assisted Non-Rigid Surface Tracking in Biological Systems

By

William Henry Hempstead

APPROVED BY:

DIRECTOR OF
THESIS: ____________________

Zhen Zhu, Ph.D.

COMMITTEE MEMBER: ____________________

Sunghan Kim, Ph.D.

COMMITTEE MEMBER: ____________________

Rui Wu, Ph.D.

GRADUATE PROGRAM DIRECTOR: ____________________

Sunghan Kim, Ph.D.

CHAIR OF THE
DEPARTMENT OF ENGINEERING: ____________________

Barbara J. Muller-Borer, Ph.D.

DEAN OF THE
GRADUATE SCHOOL: ____________________

Paul J. Gemperline, Ph.D.

"Those who have knowledge, don't predict. Those who predict, don't have knowledge."

-Lao Tzu

ACKNOWLEDGMENTS

I would like to give my highest thanks to my thesis advisor, Dr. Zhen Zhu, of the Department of Engineering at East Carolina University for his insight and guidance during the development of this research.

In addition, I would like to thank the other members of my thesis committee, Dr. Sunghan Kim of the Department of Engineering and Dr. Rui Wu of the Department of Computer Science, for their continued support.

Thank you to Dr. Barbara Muller-Borer for your encouragement and support in my return to study in the field of biomedical engineering.

Thanks to Dr. Cheng Chen and Dr. Bruce Ferguson of RFPi, Inc. for providing the data set used in this research.

Finally, thank you to all of my friends, new and old, near and far, who have been supportive of me during the course of this research.

TABLE OF CONTENTS

Page

ACKNOWLEDGMENTS . . . . . **v**

LIST OF TABLES . . . . . **x**

LIST OF FIGURES . . . . . **xi**

CHAPTER

1. INTRODUCTION **1**

1.1 Problem Statement and Proposed Work 1
1.2 Organization of Thesis 3

2. BACKGROUND INFORMATION **5**

2.1 Previous Work 5
2.2 Time Series Forecasting 8

2.2.1 Characteristics of Time Series Data 8
2.2.2 Deterministic Models 10
2.2.3 Machine Learning Methods 10

3. METHODS **15**

3.1 Programming Framework 15
3.2 Software Packages and Development Environment 16

3.2.1 MATLAB 17
3.2.2 Linux 17
3.2.3 Python 18

3.2.4 OpenCV 18
3.2.5 TensorFlow 19
3.2.6 Hardware 19

3.3 Improvements \ Changes to the Original Algorithm 20

3.3.1 Frame Pre-processing 20
3.3.2 Feature Detection Algorithm/Parameters 24
3.3.3 Tracking Algorithm/Parameters 25
3.3.4 Initial Frame Selection 25
3.3.5 Pulse Detection 26
3.3.6 Determination of Number of Patches 27
3.3.7 Neighboring Patch Smoothness Constraint 27
3.3.8 Tracking Re-initialization 29
3.3.9 Parameter Values 30
3.3.10 Occlusion Detection 31

3.4 Feature Extraction 32

3.4.1 Data Set 32
3.4.2 Extraction Method / Algorithm 34
3.4.3 Sequence Length 35
3.4.4 Scaling 36

3.5 Neural Network 36

3.5.1 Architecture 36
3.5.2 Training 38

3.6 Testing Scenarios 39

4. RESULTS 41

4.1 Results for 20160725T124323.avi 43
4.2 Results for 20160725T124609.avi 46
4.3 Results for 20160725T124909.avi 49
4.4 Results for 20160725T132840.avi 52
4.5 Results for 20160725T133113.avi 55
4.6 Neural Network Training 57
4.7 Speed of Execution 59

5. DISCUSSION **61**

5.1 Tracking Performance 61
5.2 Prediction Performance 62
5.3 Speed of Execution 63
5.4 Limitations / Future Improvements 65

5.4.1 Tracking Algorithm 65
5.4.2 Data Collection 66
5.4.3 Neural Network 67
5.4.4 Speed of Execution 68

6. CONCLUSIONS **70**

REFERENCES **71**

APPENDICES

APPENDICES **77**
A. SOURCE CODE **78**

A.1 Main Algorithm 79

A.1.1 Overview 80
A.1.2 Code 81

A.2 Feature Point Sequence Extraction 92

A.2.1 Overview 92
A.2.2 Code 93

A.3 Data Set Creation 99

A.3.1 Overview 99
A.3.2 Code 100

A.4 Neural Network Training 105

A.4.1 Overview 105

A.4.2 Code 106

A.5 Supporting Classes / Functions 109

A.5.1 Classes 109
A.5.2 Functions 112

A.6 MAT to AVI Conversion 151

LIST OF TABLES

Table **Page**

3.1 Extracted Sequence Example. 35

3.2 RNN Model Layers. 37

4.1 Tracking Performance Summary (20160725T124323.avi). 44

4.2 Tracking Performance Summary (20160725T124609.avi). 47

4.3 Tracking Performance Summary (20160725T124909.avi). 50

4.4 Tracking Performance Summary (20160725T132840.avi). 53

4.5 Tracking Performance Summary (20160725T133113.avi). 56

4.6 Execution Speed Summary. 60

5.1 Tracking Failure Summary and Comparison. 61

5.2 Grid Prediction Error Summary. 62

5.3 Profiler Summary for 20160725T124323.avi. 64

LIST OF FIGURES

Figure **Page**

1.1 Example of Occlusion from Data Set 3

2.1 Example of Region-of-Interest (ROI) 6

2.2 Simplified Block Diagram of Original Algorithm 7

2.3 Components of a Time Series 9

2.4 Multi-Layer Perceptron (MLP) Diagram 11

2.5 Recurrent Neural Network (RNN) Diagram 12

2.6 Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) Nodes 13

2.7 Convolutional Neural Network (CNN) Diagram 14

3.1 Block Diagram of Improved Algorithm 16

3.2 Comparison of Histogram Equalization Methods 22

3.3 Comparison of Denoising Methods 24

3.4 Optical Flow from Feature Point Movement 27

3.5 Smoothness Criteria Constraint 28

3.6 Sample Frame From Data Set 33

3.7 Sample of Extracted Feature Points 34

3.8 ROI Selection Landmarks 39

4.1 Representative Screenshots (20160725T124323.avi). 43

4.2 Analysis Timeline (20160725T124323.avi) 44

4.3 Average Grid Prediction Error (x, y, and x-y (absolute)) - (20160725T124323.avi) 45

4.4 Representative Screenshots (20160725T124609.avi). 46

4.5 Analysis Timeline (20160725T124609.avi) 47

4.6 Average Grid Prediction Error (x, y, and x-y (absolute)) - (20160725T124609.avi) 48

4.7 Representative Screenshots (20160725T124909.avi). 49

4.8 Analysis Timeline (20160725T124909.avi) 50

4.9 Average Grid Prediction Error (x, y, and x-y (absolute)) - (20160725T124909.avi) 51

4.10 Representative Screenshots (20160725T132840.avi) 52

4.11 Analysis Timeline (20160725T132840.avi) 53

4.12 Average Grid Prediction Error (x, y, and x-y (absolute)) - (20160725T132840.avi) 54

4.13 Representative Screenshots (20160725T133113.avi). 55

4.14 Analysis Timeline (20160725T133113.avi) 56

4.15 Average Grid Prediction Error (x, y, and x-y (absolute)) - (20160725T133113.avi) 57

4.16 Neural Network Training Loss 58

4.17 Neural Network Training Accuracy 59

CHAPTER 1
INTRODUCTION

1.1 Problem Statement and Proposed Work

The increase in machine-assisted and robotic-assisted medical procedures over the past several years has driven a need for more robust tracking techniques in computer vision. One of the most well-known robotic surgical systems, the da Vinci Surgical System, has grown to 5,582 total installations since entering the market in 2000 with approximately 1,229,000 procedures performed in 2019 [1]. These robotic, as well as, traditional laparoscopic surgeries are increasingly dependent on computer vision technologies for procedures such as tumor or cardiac ablations [2, 3] and tumor resections [4] where the surgeon's visibility and frame of reference are limited.

Other areas that can benefit from image-based tracking of organ surfaces are: fusion of various diagnostic images and endoscopic camera feeds [4, 5], instrument guidance and motion compensation [6, 7], and deformation/strain assessment of cardiac structures [8].

Fusion of different types of medical imagery can be useful in various types of robotic and endoscopic surgery. For example, by projecting an ultrasound image of a liver tumor onto a live camera feed of the moving organ surface, the surgeon can make better decisions about instrument positioning and tissue removal [5]. These types of augmented reality (AR) overlays have shown great promise in assisting with tasks such as tumor location, but accurate, precise tracking to the sub-millimeter scale continues to be a challenge [4, 9].

Many endoscopic procedures, such as ablation of specific tissue regions to destroy tumors or reduce internal hemorrhaging have become commonplace [10], while more complicated procedures such minimally-invasive Coronary Artery Bypass Grafting (CABG) continue to develop and improve [11]. These procedures require precise instrument movements synchronized with moving tissue surfaces or normal breathing motions. A robust tissue tracking algorithm could be used in such procedures to either mechanically compensate for organ movements [11, 12], or to trigger timing of ultrasound therapies to direct energy to a specific target [6].

Tracking the deformation and strain of cardiac tissues has become an important monitoring tool during complex heart surgeries. Techniques have been developed to calculate tissue strain values via the displacements derived from optical flow in cardiac videos [8, 13]. These types of measurements can serve as a "virtual" strain gauge indicating a relative load on the heart during surgery.

The original algorithm that provides the basis for the research presented in this thesis has demonstrated that accurate tracking of these non-rigid organ surfaces in 2-D imagery without *a priori* understanding of the underlying motion is possible [14]. However, obscured surface features cannot be tracked without some understanding of the underlying motion during extended periods of surface occlusion.

Throughout any typical surgical procedure, numerous occlusions and obstructions can enter the camera's field-of-view (FOV) including those caused by instruments, surgical materials, glare, or, in the case of open surgery, the surgeon's hands. Figure 1.1 shows an example of a clamping instrument and surgical thread in the FOV that could block a region-of-interest (ROI).

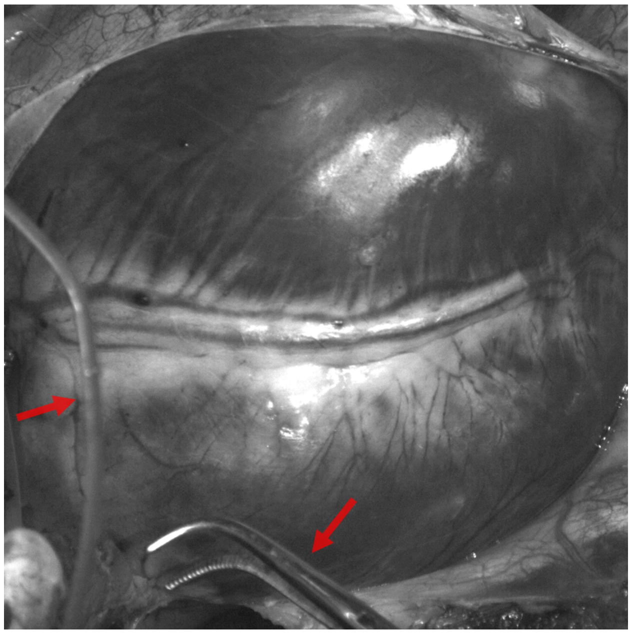


Figure 1.1: Example of Occlusions from Data Set. The left arrow points to a surgical thread and the right arrow points to a clamping instrument.

This research aims to improve upon the original work through various refinements to the algorithm and through implementation of a more optimized computer vision software library. A machine learning model will be used to model and predict the motion of the ROI.

The proposed algorithm framework will:

- Modify the image pre-processing in the original tracking algorithm to improve identification of image features.
- Improve the ROI re-acquisition strategy based on the cyclic motion of the target.
- Predict the future motion of an occluded region based on a window of previous motion.

1.2 Organization of Thesis

This research is presented as follows: Chapter 2 will review background relevant to the original algorithm and sequence prediction. The underlying methods used for algorithm development and the experimental design will be presented in Chapter 3. Lastly, results, discussion, and conclusions will be presented in Chapters 4, 5, and 6 respectively. All of the source code used for this project is listed in the Appendices.

CHAPTER 2
BACKGROUND INFORMATION

This chapter provides a review of topics relevant to the proposed work. The first section will highlight some key points from the original research [14]. The second will discuss point sequence prediction as a time series forecasting problem.

2.1 Previous Work

The research that established the original algorithm developed a method for tracking non-rigid surfaces based on the establishment of constraints (statistical tests) for the optical flow of distinct feature points within small, piecewise areas (patches) of a particular region (See Figure 2.1). Feature points are distinctive locations in an image that are detected through a particular detection algorithm (in this case Shi-Tomasi [15]). Figure 2.2 shows a simplified block diagram of the original algorithm for reference.

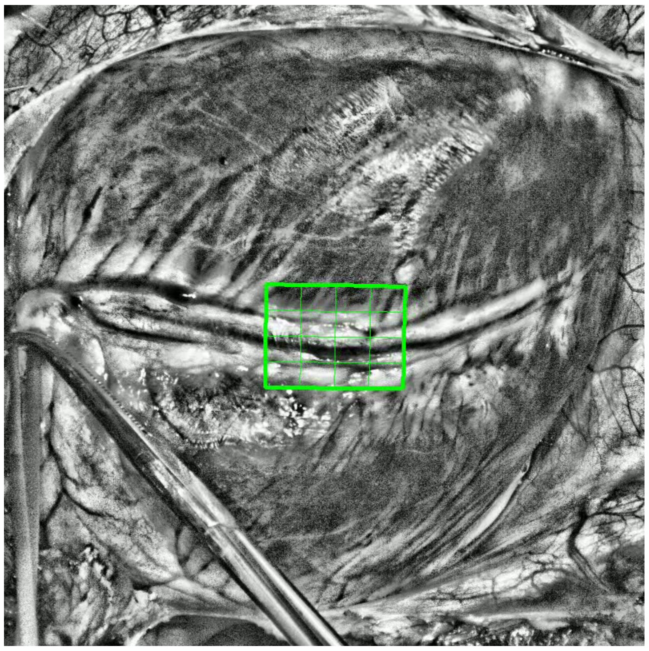


Figure 2.1: Example of Region-of-Interest (ROI). The grid area is divided into smaller, piecewise areas (patches).

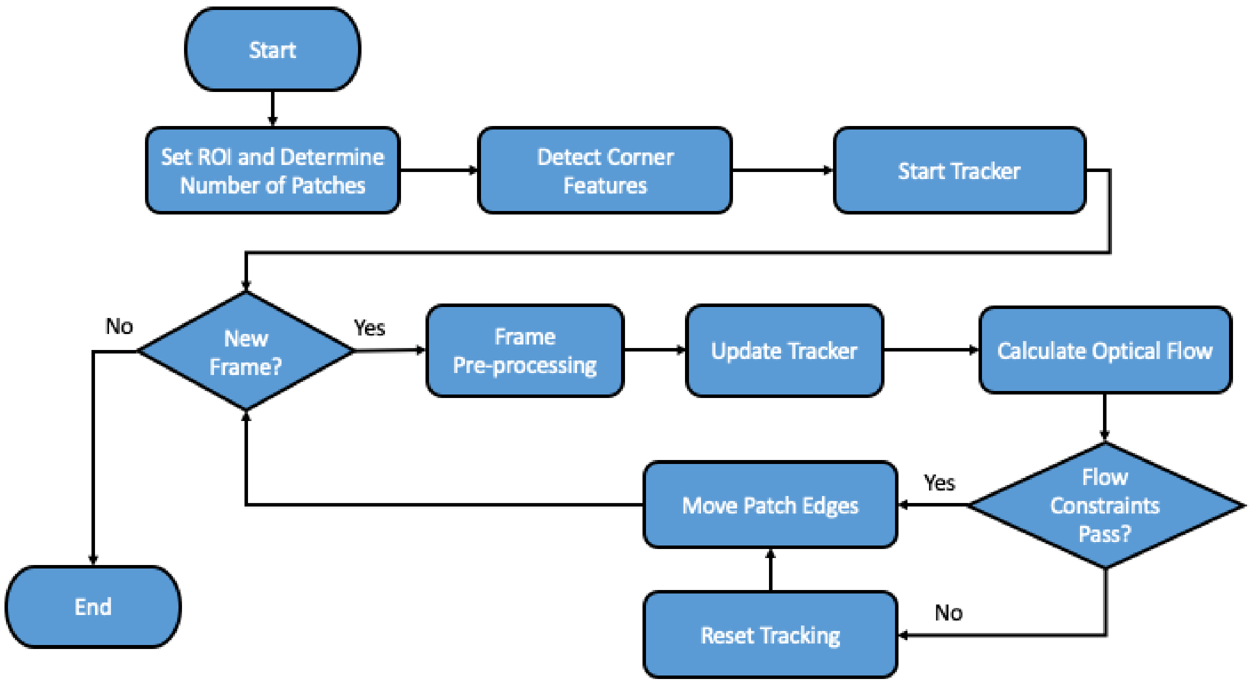


Figure 2.2: Simplified Block Diagram of Original Algorithm [14].

These tracking constraints were: 1) the motion of feature points must follow a normal distribution within a patch and 2) the motion of a particular patch must follow a normal distribution relative to its neighboring patches. If these constraints are not met, tracking is re-initialized using a more computationally expensive homography transform which warps the perspective of the grid points to match a reference frame [16].

The original algorithm was able to continue tracking under some cases of partial occlusion, but failed when the ROI was completely obscured. To improve the performance during a total occlusion, the original research proposed the addition of machine learning-based ROI prediction. By using some knowledge of the previous motion history of the ROI grid points before occlusion, a prediction of future grid points could be made until the occlusion is cleared. This translates into a multiple point sequence prediction problem.

2.2 Time Series Forecasting

The topic of point sequence prediction can be grouped under the larger category of time series forecasting. Research has been done on many different types of data to predict future values based on past history. Much of this research focuses on financial data analysis [17] and scientific sensor data [18].

Most of these prediction approaches fall into two categories: deterministic models or machine learning methods. The following sections will discuss these methods in greater detail along with some of the characteristics of time series data important to forecasting.

2.2.1 Characteristics of Time Series Data

One of the fundamental assumptions for forecasting a time series with traditional techniques is stationarity. If a data set is said to have stationarity, one of more of the statistical distribution measures (mean, variance, and co-variance) are constant. Without at least one constant statistical property, a signal would be considered noise and a reliable prediction would not be possible [19].

A typical time series can be separated into three components: a trend, seasonal (cyclic) variation, and random variation. Figure 2.3 demonstrates how each of these components contribute to the overall series.

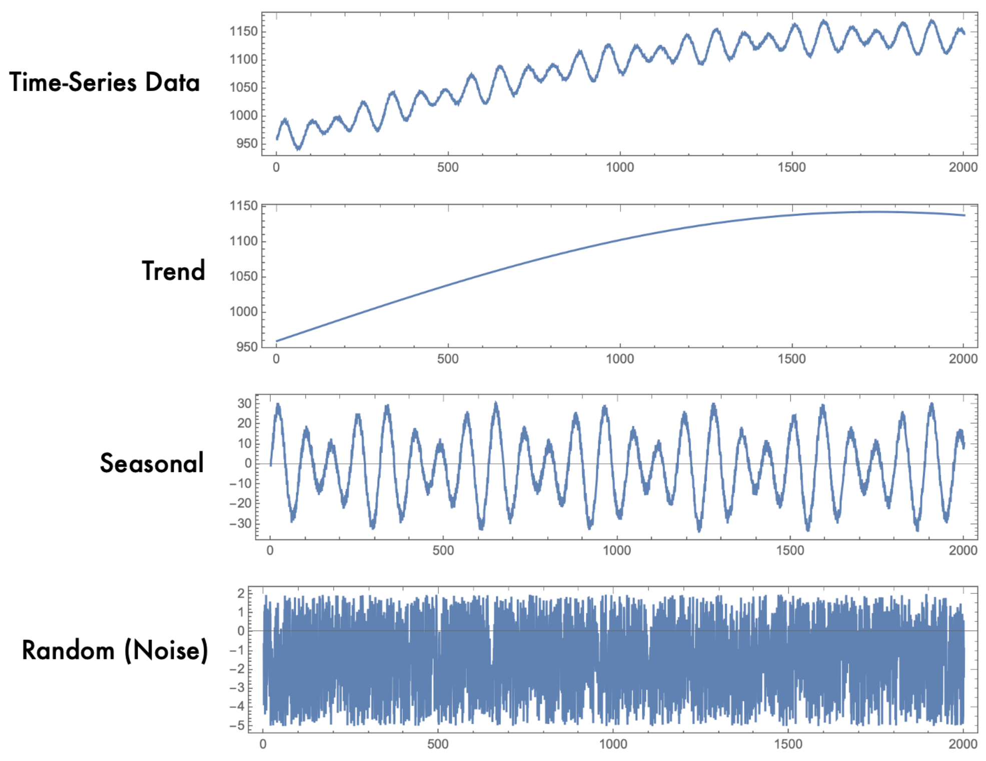


Figure 2.3: Components of a Time Series. The trend, seasonal, and random parts of the series sum to form the composite signal.

A trend is a long term directional change in the data that can be either linear or nonlinear. Seasonal (periodic cyclic) components are repeated, long or short-term patterns that occur throughout the data series. On top of this trend/seasonal structure, a degree of randomness or noise from error and unpredictability in measurements is typically present [19].

To achieve stationarity in a time series, mathematical transformations are performed to remove the trend and seasonal components leaving an aperiodic data series. This can be achieved with various filtering and differencing methods [19].

Time series data can also be classified as univariate or multivariate. Univariate time series data consists of the values of a single variable over a period of time such as the daily temperature over a year. Multivariate time series data consists of multiple variables such as temperature, rainfall, and atmospheric pressure over a concurrent period of time.

2.2.2 Deterministic Models

A deterministic model would attempt to use mathematical formulas to strictly define the motion based on expected patterns. The development of deterministic models is difficult in problems dealing with non-rigid tissue motion because of the complex spatial dynamics of the surface, thus this research will not focus on the use of deterministic models.

2.2.3 Machine Learning Methods

Various machine learning techniques have been developed to perform time series forecasting on collected data. The two main methods found in the literature are Autoregression and Neural Networks.

Autoregression

Autoregression analyses are a form of state-space models that use linear equations to predict future values based on differences with previous values at various lag times. The state-space model consists of two equations: an observation equation describing the output of a system and a state equation describing a system's current state based on a previous observation [19].

The Autoregressive Moving Average (ARMA) model is one of the standard analyses for stationary data and the Autoregressive Integrated Moving Average (ARIMA) model is used for non-stationary data [19]. Univariate and multivariate versions of both of these techniques have been developed.

A multivariate autoregression could be used to analyze several time sequences at once and make a prediction for each sequence. An advantage of this approach would be that spatial relationships between the time sequences could be preserved if the spatial locations were fixed.

Neural Networks

MLP Multi-layer perceptrons (MLPs) are one of the most basic types of neural networks consisting of a series of input nodes connected to one or more layers of hidden nodes and output nodes (Figure 2.4). MLPs are classified as feed-forward networks because the flow of information is always from the input nodes through the hidden layers to the output nodes. Therefore, the MLP network is said to have no "memory" or feedback of previous states or inputs [20].

The MLP network is initialized with a rigid input/output structure of a fixed size. While these sizes are fixed, variable size inputs can be used with an MLP structure if the input size is chosen to be greater than the longest individual input and the inputs are "zero-padded" [21], however, the output size will remain a fixed length.

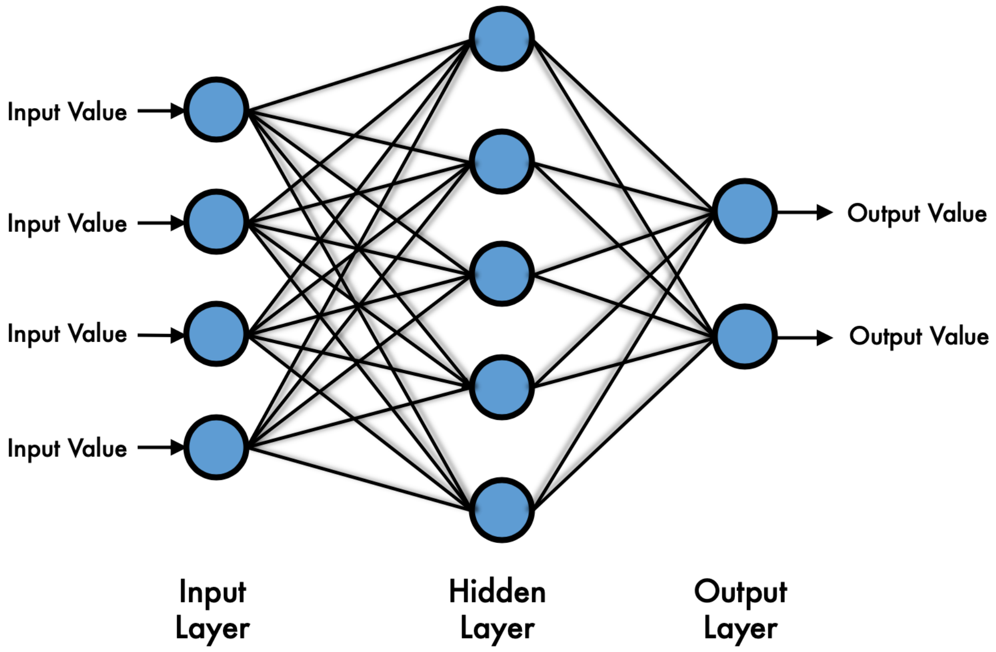


Figure 2.4: Multi-Layer Perceptron (MLP) Diagram.

RNN Recurrent Neural Networks (RNNs) have a structure similar to MLPs with the addition of feedback connections to add a form of "memory" that can be useful in sequence analysis (See

Figure 2.5) [20]. One advantage of this type of network over a standard MLP is that it can fed with variable length inputs to generate variable length outputs.

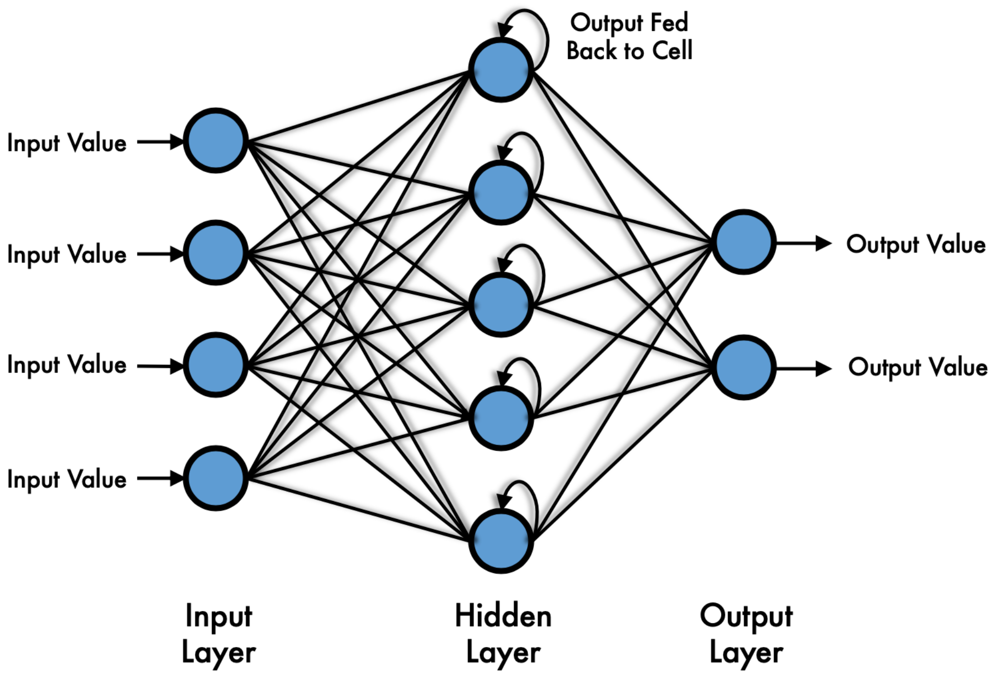
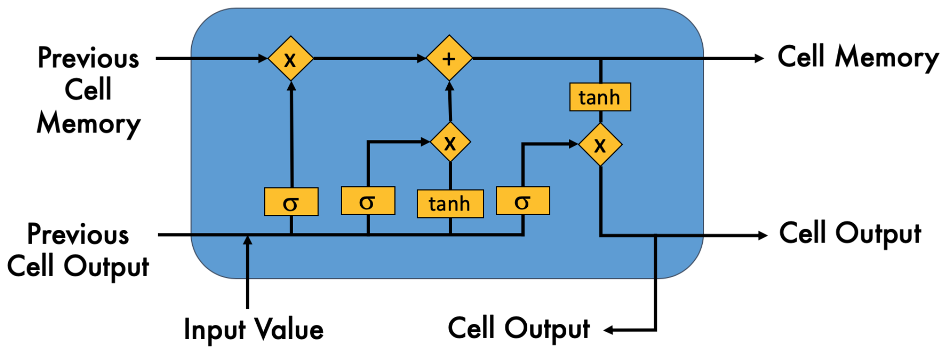
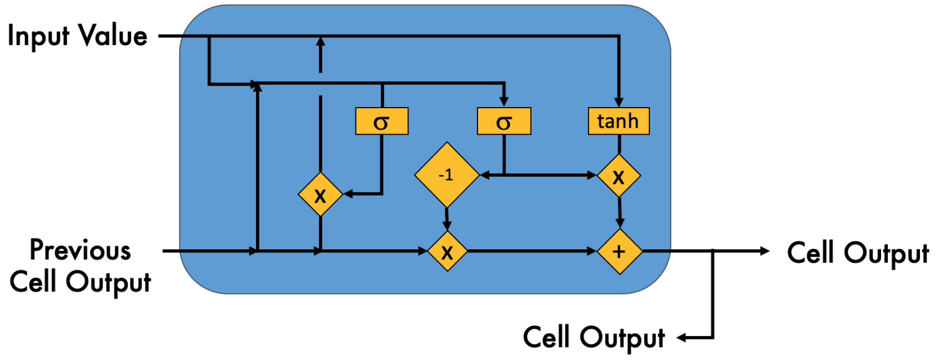


Figure 2.5: Recurrent Neural Network (RNN) Diagram.

One large disadvantage of RNNs is that the memory feedback loop can lead to the problem of exploding or vanishing gradients during backprogation in the training phase. The errors in the derivative calculations which determine the weighting of the nodes can, in some cases, drive the weight gradients to very small values (vanishing) or to very large values (exploding) [21]. Either case results in a network of limited value that cannot be effectively trained.



(a) LSTM Cell



(b) GRU Cell

Figure 2.6: Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) Nodes.

The addition of alternate information pathways, gates, within the network nodes can help to better control the flow of the information through the network and eliminate the exploding/vanishing gradient issue. The two most well-known modifications to the RNN node are the Long Short-Term Memory (LSTM) cell and the Gated Recurrent Unit (GRU) cell (Figure 2.6) [22]. The LSTM cell has the addition of three gates: the forget gate, the input gate, and the output gate. The GRU is a more recent invention intended to improve processing speed by reducing the number of gates and, therefore, the number of operations. It only has two gates: the update gate and the reset gate. These two network types can have similar performance in many situations, but research to establish the best case uses is ongoing.

CNN A Convolutional Neural Network (CNN) is optimized for learning patterns in two-dimensional data occurring in a regular spatial pattern [20]. This data could take the form of image data or any other 2-D array of values in a grid-like structure. Data flows forward through the network (See Figure 2.7) and is downsampled through a series of convolutions to extract essential features and perform tasks such as classification and segmentation [23].

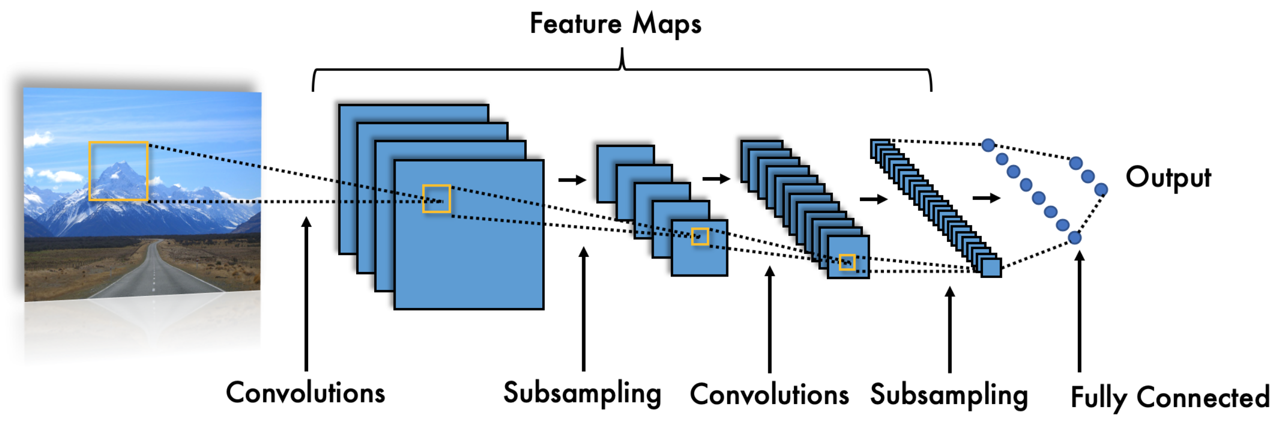


Figure 2.7: Convolutional Neural Network (CNN) Diagram.

CNNs have come into greater prominence after the success of the Krizhevsky et al. team in the 2012 ImageNet competition where image classification was improved by over 10% compared to the previous year's winner [24]. Recurrent CNNs (R-CNNs) have also been successfully applied to many types of video analyses such as video captioning [25], object tracking [26], and frame prediction [27]. The hardware requirements for network training are high [27], but inference operations have been able to generate new complete frames in real-time (20-30 ms per image [~30 fps]) on a consumer level GPU [28].

CHAPTER 3
METHODS

The following sections describe the modifications to the original MATLAB code, the overall framework for incorporating the occlusion prediction into the original algorithm, the rationale for the selected software packages and neural network, and the structure of the experimental tests of the framework.

3.1 Programming Framework

The overall structure of the research will be arranged as follows:

1. Rewrite the original algorithm [14] to improve tracking performance and support a future real-time implementation.
2. Develop a feature extraction method to create a point motion sequence data set.
3. Train the selected machine learning model using the point motion sequence data set.
4. Integrate the machine learning model into the main algorithm for prediction of the ROI.
5. Continue ROI tracking using the machine learning prediction.
6. Assess the performance of the tracking prediction.

Figure 3.1 shows a simplified block diagram of the improved algorithm.

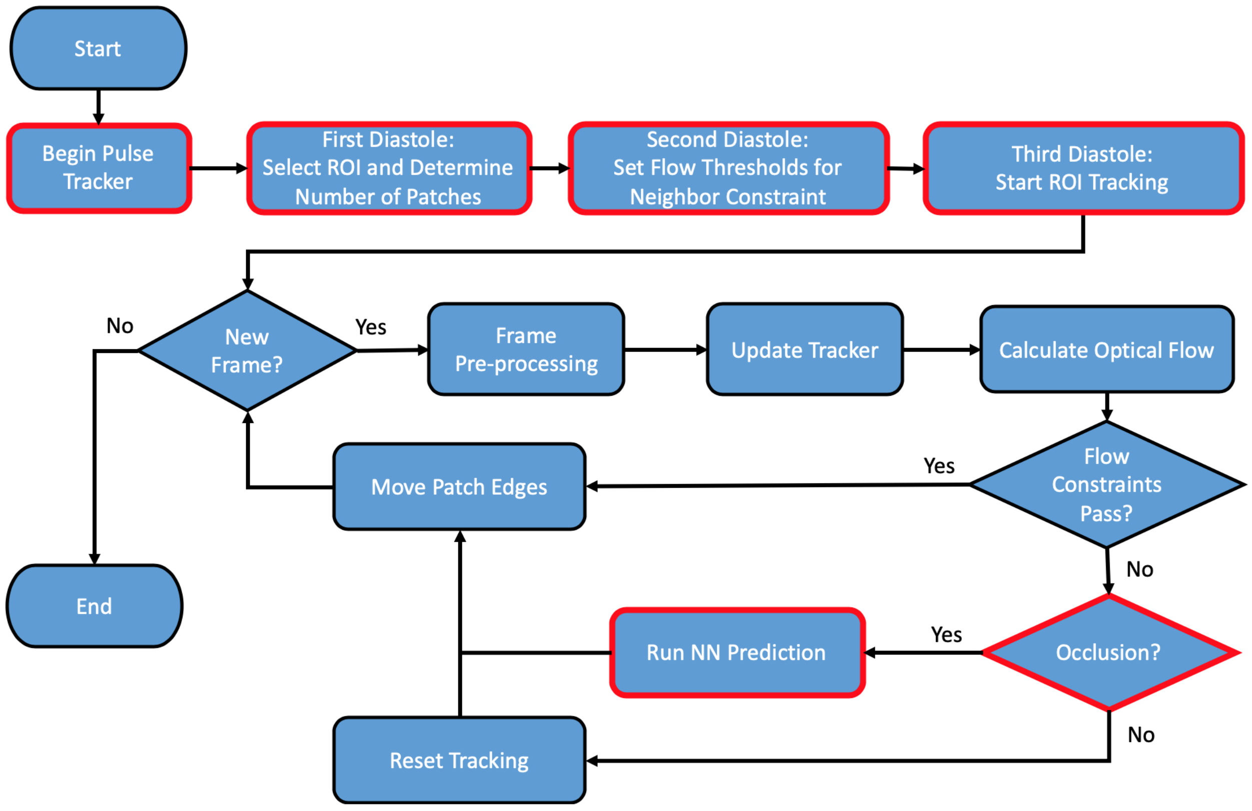


Figure 3.1: Block Diagram of Improved Algorithm. The red outlines indicate added routines.

3.2 Software Packages and Development Environment

The following is a list of the main software packages and versions used in this research:

- MATLAB R2019a
- Linux - Ubuntu 18.04 LTS
- Python 3.6.9
- OpenCV 4.2.0-dev
- NumPy 1.18.1
- TensorFlow 2.0.0

The next sections give some background on each package and rationale for its use in this research.

3.2.1 MATLAB

MATLAB is a programming environment/language developed by Mathworks that provides the ability to create programs and run computational code for an array of scientific analyses. It has the capability of running various image processing and computer vision algorithms and was used in the original algorithm research [14].

In recent years, MATLAB has added more capabilities to interface with C++, NVIDIA Graphics Processing Units (GPUs), and Python. Mathworks continues to improve the speed of its algorithms by providing access to C++ functions through MEX [29], GPU-enabled algorithms, and machine learning models that are claimed to be up to 7x faster than TensorFlow [30]. These approaches may be viable, but the practical implementation of combining and optimizing all of these methods was beyond the scope of this research.

Therefore, a more streamlined approach was taken to keep everything open-source (free) and cohesive by using the Python language combined with NumPy, SciPy, OpenCV Python, and TensorFlow. In addition, many of the MATLAB image processing functions are not optimized for a GPU and the transfer of data between the CPU and GPU requires extra programming attention to optimize speed. Mismanagement of the CPU-GPU data transfer can make the code run slower than without a GPU [31].

3.2.2 Linux

Development for this project was done under the Linux operating system (Ubuntu 18.04 LTS [Long Term Stable]). Linux is a free, open-source operating system and its ability to scale to many different types of hardware could be useful for future implementation in a real-time system or actual product.

3.2.3 Python

Python has become more popular as a programming language in recent years for scientific computing due to its open-source nature, clear syntax, and the proliferation of add-on packages that provide access to many essential programming functions [32].

C and C++ are traditionally the most popular languages for performance-driven computing applications especially in embedded systems [33]. However, increasingly Python is being used to create bindings to faster C++ functions to achieve similar performance with more intuitive, understandable code. The Cython language, a superset of Python, can also be used to generate C or C++ code from regular Python code for faster performance [34]. These are additional options that may improve the performance of the code but were not implemented in this research.

Packages such as NumPy [35], SciPy [36], Scikit-learn [37], and Matplotlib [38] have given Python expanded functionality for various mathematical and scientific computing operations. NumPy adds support for array generation and manipulation similar to that found in MATLAB. SciPy adds many scientific computation modules including modules for signal and image processing and statistics. Matplotlib adds graphical plotting capabilities similar to those available with MATLAB.

3.2.4 OpenCV

OpenCV (Open-source Computer Vision) was developed as a cross-platform real-time computer vision library and has both Python and C++ versions [39].

The conversion of the original MATLAB code to OpenCV provides several advantages. OpenCV Python allows easy integration with other Python-based packages such as TensorFlow and NumPy. OpenCV was developed for real-time systems and runs on Linux allowing applications to be more easily scaled and ported to various types of hardware. Finally, OpenCV is open-source and free-to-use compared to the cost of a non-educational MATLAB license.

At this time, the Python version of OpenCV is not optimized for the CUDA language. Compute Unified Device Architecture (CUDA) is a software interface for graphical processing units (GPUs) created by NVIDIA, the leading manufacturer of GPUs [40]. CUDA allows many operations to be performed in parallel on NVIDIA GPUs including the types of matrix operations common in image processing and machine learning.

OpenCV Python can, however, make use of OpenCL acceleration via the Transparent API (T-API) added in OpenCV Version 3 [41]. OpenCL is a more general GPU language that can run as an alternative to CUDA on non-Nvidia GPUs. It can also run on Nvidia GPUs, but has much slower performance than CUDA. It is possible to create Python bindings for the C++ functions and achieve increased speed using OpenCV with CUDA instead of OpenCL.

Due to the frame-by-frame nature of the tracking algorithm, the GPU was not used for processing during tracking as these image processing tasks are better performed in large batches that minimize the transfer of data between the system and GPU memory.

3.2.5 TensorFlow

TensorFlow is an open-source machine learning library developed and maintained by Google that is compatible with Python, C++, and CUDA. Version 2.0 included the popular high-level machine learning interface Keras as part of the core TensorFlow package. TensorFlow Lite allows the flexibility to run models on less powerful devices [42].

3.2.6 Hardware

Image processing, neural network training, and final algorithm testing were performed on a custom-built workstation containing the following components:

- Intel Core i9-9900X 3.5 GHz 10-Core Processor
- Zotac NVIDIA GeForce RTX 2080 Ti 11 GB Video Card
- Asus WS X299 SAGE SSI CEB LGA2066 Motherboard
- Corsair Vengeance LPX 32 GB (2 x 16 GB) DDR4-2133 CL13 Memory
- Samsung 960 EVO 500 GB M.2-2280 NVME Solid State Drive

These components are near the top of the performance ladder available for workstation desktop PCs at the time of this research.

3.3 Improvements \ Changes to the Original Algorithm

The following sections describe the improvements and modifications that were made to the original algorithm including changes to frame pre-processing, a new method to determine the initial feature detection frame and detect the pulse cycle, a modification to the neighboring patch constraint, and modifications to the tracking reset routine.

3.3.1 Frame Pre-processing

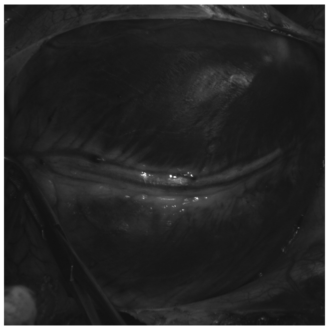
To improve feature point detection and tracking in the original algorithm, two standard imaging processing techniques were applied to each frame: a histogram equalization and denoising (blur) filter. These techniques are still used in the new framework, but they have been modified to give better results as described in the following paragraphs.

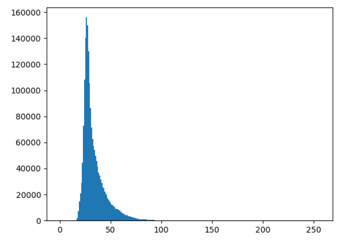
Equalization

Histogram equalization enhances the contrast of the image by spreading the pixel intensity levels across the full spectrum of grayscale levels. By improving contrast, features on the surface will be more distinct and more easily found by the feature detection and tracking algorithms.

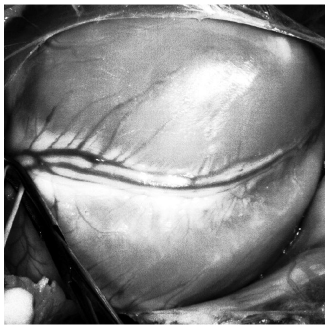
The histogram equalization algorithm used in the original research was the Matlab function "histeq" with its default settings [43]. For this research, improved performance was found by using the Contrast Limited Adaptive Histogram Equalization (CLAHE) algorithm of OpenCV [44]. This algorithm performs a similar function as the "histeq" algorithm multiple times on smaller tiled sections of the image with an additional option to limit the amount of level changes. A 20x20 pixel tile with a level clip threshold of 30 was found to give good performance on this data set.

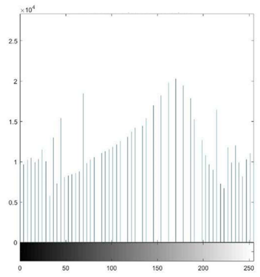
Figure 3.2 shows an example of the difference in enhancement between these two equalization algorithms. The original images were very dark near the low end of the intensity scale. The basic Matlab and OpenCV equalization algorithms give different results but both spread the intensity levels across the full grayscale range. The CLAHE algorithm gives much greater contrast across the image than the basic equalization algorithm with a more dense histogram. The main coronary artery is much more defined in the equalized image using this method. It is expected to provide better point feature tracking performance. The highlights in the areas of glare (upper right center) are also greatly reduced helping to improve tracking performance.


(a) Unprocessed Image

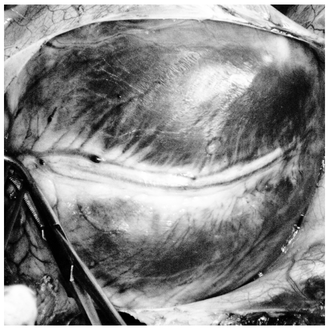


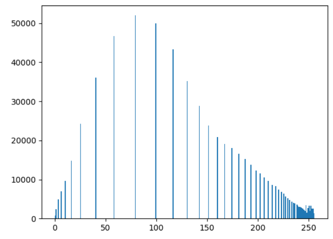
(b) Unprocessed Image - Histogram


(c) MATLAB Basic Equalization [14]

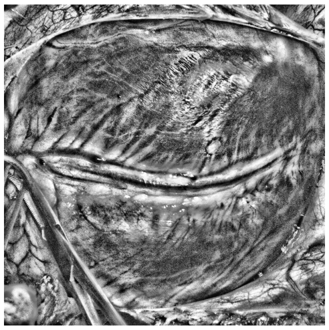


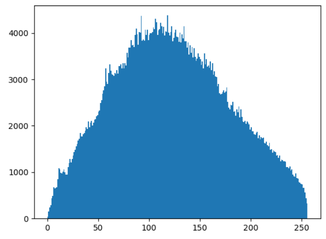
(d) MATLAB Basic Equalization - Histogram [14]


(e) OpenCV Basic Equalization



(f) OpenCV Basic Equalization - Histogram


(g) OpenCV CLAHE Equalization



(h) OpenCV CLAHE Equalization - Histogram

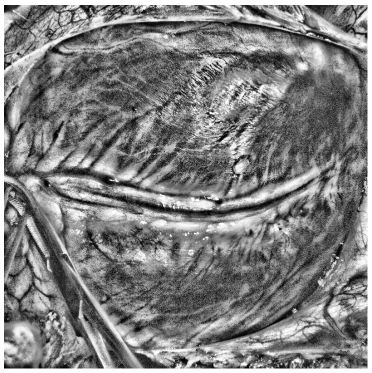
Figure 3.2: Comparison of Histogram Equalization Methods.

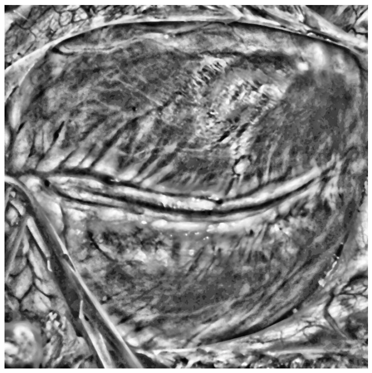
Denoising

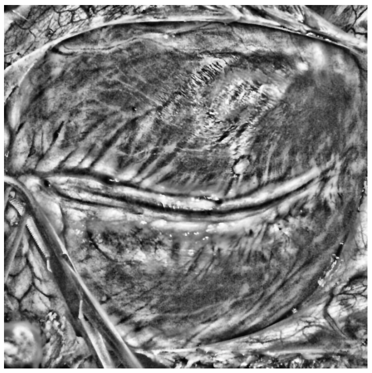
Denoising, or blurring, can remove or reduce small, speckle-like artifacts in the image by applying a smoothing filter. This helps the feature detector or tracker to focus on the strongest image features instead of the undesired noise artifacts.

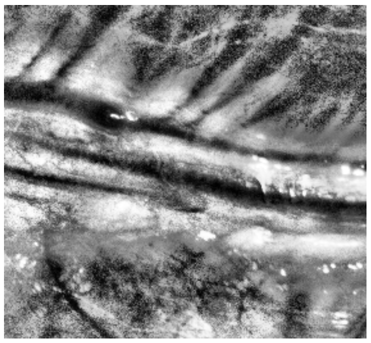
OpenCV has multiple denoising algorithms: the box filter, gaussian filter, median filter, and bilateral filter. The first three of these algorithms are effective in reducing noise but can also smooth over the edges in the image [45]. The bilateral filter takes the pixel intensity of neighboring pixels into account to reduce noise while better preserving edges [45].

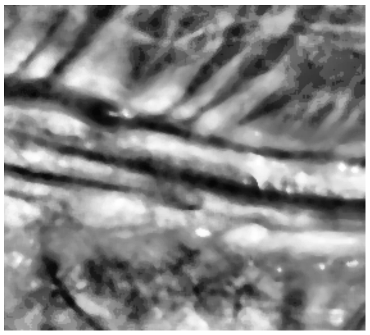
The original research used the median filter in MATLAB. After some experimentation using the different blurring filters, this research chose to use the bilateral filter for its slight improvement in retaining edges which are important for optimal feature detection. Figure 3.3 shows a comparison of the unblurred image, median filter, and bilateral filter. The difference is subtle, but there is a slight increase in edge contrast with the bilateral filter (more apparent in the magnified images).


(a) No Filter


(b) Median Filter


(c) Bilateral Filter


(d) No Filter - Magnified


(e) Median Filter - Magnified

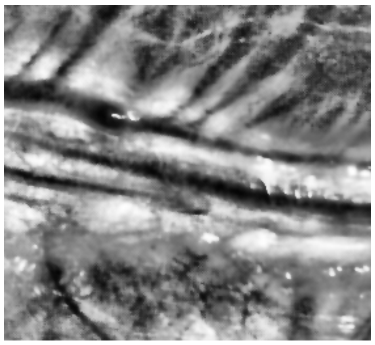

(f) Bilateral Filter - Magnified

Figure 3.3: Comparison of Denoising Methods. The Bilateral filter reduces speckling while better preserving edges.

3.3.2 Feature Detection Algorithm/Parameters

The original algorithm uses the MATLAB function “detectMinEigenFeatures” to detect the initial feature points [14]. This function incorporates the Shi-Tomasi minimum eigenvalue algorithm to find feature points [46].

The equivalent OpenCV function is “goodFeaturesToTrack” which uses the Shi-Tomasi algorithm [47]. By setting the “useHarrisDetector” parameter to “False” or “0,” this function will use the minimum eigenvalue calculation and operate similar to the MATLAB function.

The “qualityLevel” parameter was decreased to 0.001 from the 0.005 value used in the original research which gave a better distribution of points within the ROI.

3.3.3 Tracking Algorithm/Parameters

The original algorithm uses the MATLAB function “vision.PointTracker” to track the set of detected points throughout the image sequence [48]. This function incorporates the Kanade-Lucas-Tomasi (KLT) feature-tracking algorithm [49].

The equivalent OpenCV function is “calcOpticalFlowPyrLK” which uses the iterative Lucas-Kanade method with pyramids [50].

The implementation of the OpenCV KLT tracker is different from the MATLAB KLT tracker and does not include the error parameter, “MaxBidirectionalError,” that is set to “1” in the original research. This parameter does a calculation forward and backward between frames to double-check the tracking calculation and uses this difference as an error criterion at the cost of additional computational overhead [48]. With a setting of “1”, the tracking would be considered invalid if the difference in the forward and backward tracking location was greater than 1. According to the OpenCV documentation, this backward check is implemented into “calcOpticalFlowPyrLK” but control of this error criterion is not directly accessible [50].

3.3.4 Initial Frame Selection

A recommendation was made in the previous research to use machine learning to assist in the selection of the initial feature point detection frame. It was also found that the diastole point of the cycle was the best frame to use for initialization of the feature points [14]. This finding makes intuitive sense because the tissue surface is most relaxed and stretched during diastole and thus covers more surface area (pixels) providing more information to the feature detector.

While a machine learning approach may be viable, it would require an effort to collect and label many examples of heart images at the diastole phase and training of another classification algorithm. A more simple approach can make use of the pulse detection algorithm described in the next section to determine the approximate frame corresponding to the diastole phase.

3.3.5 Pulse Detection

A method for detecting the pulse cycle was implemented in order to assist in the initialization of feature detection described in Section 3.3.4 and the re-initialization of tracking. An additional benefit of pulse cycle detection is the calculation of an approximate heart rate based only on the video imagery.

To detect the cycle, a sparse KLT feature point tracker (using the 10 strongest features across the entire frame) is started at the beginning of the video sequence. The average of the absolute Euclidean distance of all of the tracked points from their initial location is calculated for each frame. This distance should have a rough correlation to the overall motion in the frame.

The peaks of this average distance should correspond to the approximate systole and diastole points of the cardiac cycle where motion is at a minimum and changing direction. By tracking the direction of the distance changes, the type of peak can be determined (i.e. increasing distance before a peak would correspond to a diastole [relaxation expansion] and decreasing distance would correspond to a systole [contraction]).

This cycle detection runs continuously in the background independently of the main feature tracker for the selected ROI. An example of this overlay of the optical-flow based cycle is shown in the red graph at the bottom of Figure 3.4.

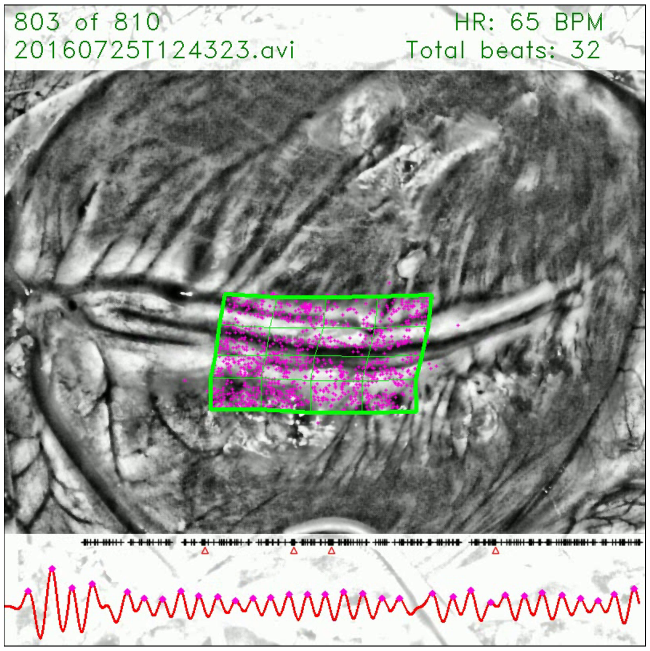


Figure 3.4: Optical Flow from Feature Point Movement.

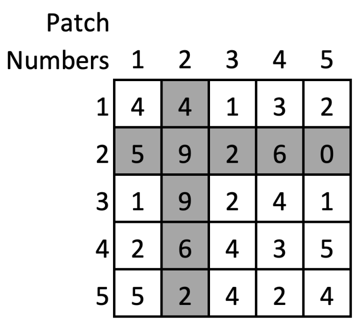
3.3.6 Determination of Number of Patches

The function to determine the number of patches in the original algorithm relied on incrementally increasing the number of patches until the two constraint conditions described in Section 2.1 were met. The patches are set to increase as a square matrix (i.e. 2x2, 3x3) to a maximum of 36 patches (6x6). This is similar to the method in the original algorithm.

If the maximum number of patches is reached without satisfying the constraint criteria, the calculation will proceed, but a warning will be displayed.

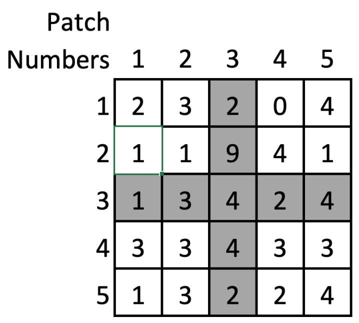
3.3.7 Neighboring Patch Smoothness Constraint Criteria

The neighboring patch smoothness constraint (Section 2.1, Constraint 2) from the original algorithm tested the normality of the optical flow difference values of each patch's corresponding row and column and reinitialized tracking if this condition was not met (See Examples in Fig. 3.5 a, b).



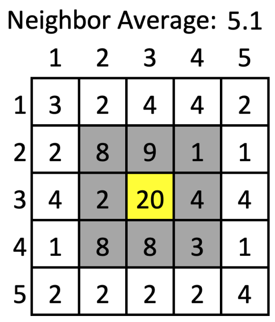
Patch Numbers	1	2	3	4	5
1	4	4	1	3	2
2	5	9	2	6	0
3	1	9	2	4	1
4	2	6	4	3	5
5	5	2	4	2	4

(a) Original Constraint 2 Comparison - Example 1



Patch Numbers	1	2	3	4	5
1	2	3	2	0	4
2	1	1	9	4	1
3	1	3	4	2	4
4	3	3	4	3	3
5	1	3	2	2	4

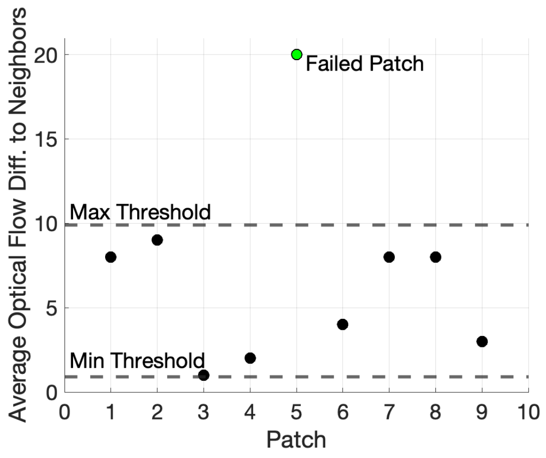
(b) Original Constraint 2 Comparison - Example 2



Neighbor Average: 5.1

	1	2	3	4	5
1	3	2	4	4	2
2	2	8	9	1	1
3	4	2	20	4	4
4	1	8	8	3	1
5	2	2	2	2	4

(c) New Constraint 2 Comparison to Surrounding Neighbors



(d) Outside Threshold Range -> Failed Constraint Check

Figure 3.5: Smoothness Criteria Constraint.

Because this algorithm consisted of several nested loops that were slow, unclear, and difficult to modify, a different measure of smoothness was implemented in this research. Instead of calculating multiple U (horizontal) and V (vertical) flow differences between each patch in the entire row and column and then checking normality, the average U and V flow for each patch will be calculated and compared to the average of the neighboring patch flows (See Figure 3.5 c).

This allows for a simpler constraint check as a comparison of the individual patch flow to the expected range of neighbor flow. In addition, the influence of the immediate diagonal neighboring patches is now considered without the influence of distant patches. The next paragraph describes how the flow threshold range was determined.

Threshold Range

After the number of patches are determined during the first full heart cycle, the difference between the average optical flow of each patch and the average optical flow of its neighbors is monitored during the second heart cycle. Each patch is assigned a maximum and minimum threshold range (plus margin) based on the observed difference values for later comparison during tracking. Any patch with an extreme difference from its neighboring patches would violate these thresholds and the patch would be flagged as failing the constraint check (See Fig. 3.5 d).

The original algorithm included analysis of several frames in order to determine a maximum threshold for the smoothness across a cycle. This was performed in the same routine used to determine the number of patches and slowed program execution during calculation. In this research, this threshold determination procedure was moved into a separate routine to allow for more options to calculate the threshold while the video sequence is moving forward.

3.3.8 Tracking Re-initialization

The original algorithm re-initialized tracking after the constraint check failed on three consecutive frames. Once three failures were detected, another routine began to search backward from the current frame through previous frames to find a best homography match. This homography was used to transform the region of interest boundaries and reacquire tracking points.

This algorithm has a long execution time as implemented and could benefit from several improvements. The backward search through multiple frames performs the same calculations multiple times which is inefficient for use in a real-time application. Performing operations in a forward direction would be more desirable in a live video stream application. Thus, in this research, the original ROI selection frame and ROI grid are saved and used to obtain a homography matrix when tracking is lost. The homography matrix is used to warp the original ROI grid to the current frame.

In scenarios where frames with occluded ROIs could be compared to the non-occluded selection frame, re-initialization error is likely to be high, triggering more constraint failures and attempts to re-initialize tracking, but tracking should return to normal when the occlusion is cleared. Future work could add changes to limit this undesired behavior.

A homography re-initialization is also done at each diastole cycle whether or not there has been a tracking failure. This modification should help to reduce tracking failures in general and periodic tracking resets are recommended by the OpenCV development team for more robust tracking [51]. While each added re-initialization slows performance, it was anecdotally observed that less total re-initializations were needed with periodic resets compared to waiting for tracking failure.

3.3.9 Parameter Values

Optimization of the tracking parameters is not the focus of this research, but several observations were made while re-factoring the algorithm. The quality of unobstructed tracking depends on a balance of the number of initial points tracked, the tracking failure threshold number, size of the ROI, and the presence of glare.

Ensuring that there are a minimum number of points within each patch to meet the statistical normality tests is a good guideline for setting the number of corner feature points to detect. If the maximum number of patches is set to 36 (6 x 6) and the minimum number of points to meet the normality test is 20, a minimum of 36 x 20 = 720 points would be required. More points than the minimum required should lead to better tracking that is less sensitive to outliers. The detection threshold was set to 1000 features in this research to give 720 points plus a margin. Care should be taken to avoid using an excessive number of points which would slow down the overall calculation time. A dynamic patch point threshold of 20 points per patch (plus a margin) could be set to reduce excess feature points and improve run times.

The threshold for the number of tracking failures before a reset can be increased if the tracking error is relatively low. Setting the failure limit too low can lead to frequent, perhaps unnecessary, re-initializations that can cause slow-downs and potential re-initialization errors. The failure limit setting is also closely related to the frame rate of the video under analysis. If the frame rate is very high, multiple failures may occur over several frames that represent a very short length of time. Setting the failure limit high could be warranted in this case to prevent resets from occurring too often. Likewise, if the frame rate is very low, movement (and error) could be higher from frame to frame and a lower failure limit would be justified. The tracking failure limit in this research was able to be increased to five consecutive failures without excessive tracking resets.

In general, larger ROIs with larger patch sizes were tracked better than smaller overall ROIs or ROIs with smaller patches. This is likely due to a larger ROI containing a greater number of distinctive features overall and more detected features within each patch. This may vary with the resolution of the frame as well.

ROI areas with glare seemed to generate more overall tracking resets. Glare is similar to an occlusion in the sense that it hides the tissue surface information from the view of the camera. However, some amount of glare is likely always visible in biological images due to moisture and ambient lighting. This may be reduced using the CLAHE equalization described in Section 3.3.1, however, it will not be completely eliminated because the pixel intensity has been saturated to a maximum value and texture detail will be lost in the overexposed areas.

3.3.10 Occlusion Detection

No changes were made in this research to improve the detection of occlusions in the original algorithm. The original research assumed that a partial occlusion would trigger constraint failure and an eventual re-initialization. Failure of constraints on all of the patches would be considered a full, non-recoverable occlusion and may require a pause of tracking until the occlusion is removed [14].

In this research, a full occlusion will be simulated by switching the ROI grid from the tracking algorithm to a prediction from the machine learning model for a specified selection of frames. A shaded rectangle will be displayed to indicate an occlusion, but this is cosmetic and does not trigger the prediction or affect the tracking algorithm.

In a final implementation, additional work may be needed to define criteria for switching to a neural network prediction or triggering a tracking re-initialization. Actual occlusions would have several variables such as the speed across the ROI, the size relative to the ROI, and the period of occlusion which could trigger unwanted and inaccurate re-initialization behavior.

Expected types of real occlusions fall into three categories: glare, shadows, and objects. Routines could be created to deal with the unique properties of each of these separately. Glare causes a complete saturation of intensity levels, shadows cause a partial decrease in intensity levels, and an object introduces new features and edges that can confuse the feature detector/tracker.

3.4 Feature Extraction

3.4.1 Data Set

Five data files containing image sequences of cardiac motion were provided by Dr. Bruce Ferguson and Dr. Cheng Chen for the previous research [14]. These images were obtained during an unrelated, previously conducted porcine (pig) animal study conducted at ECU facilities under Institutional Animal Care and Use Committee (IACUC) supervision and approval.

The image sequences contain an upright, anterior view of the heart with the coronary artery in a horizontal orientation within the image frame as shown in Figure 3.6. The heart rate for all of the sequences is approximately 60-70 beats per minute (bpm). The image intensity levels are very low (dark) without any level adjustments.

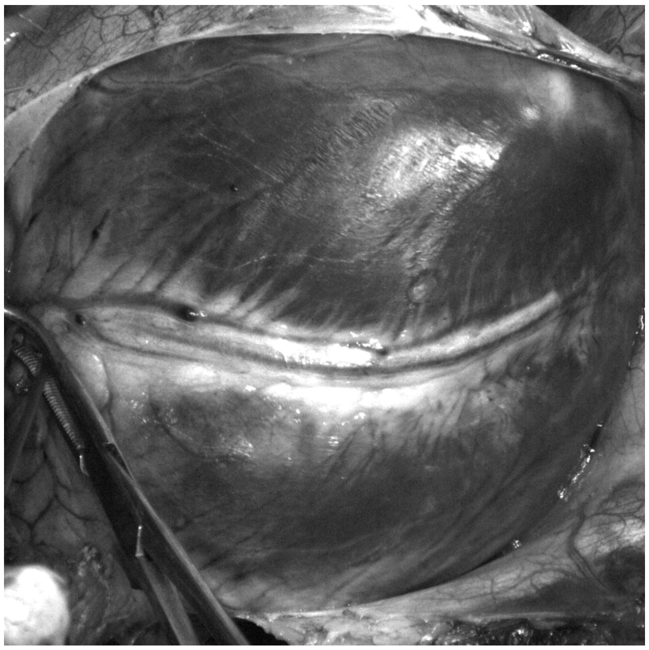


Figure 3.6: Sample Data Set Image Frame. Raw image levels adjusted for clarity.

For the majority of the heart cycles, most of the heart surface is in clear view with minimal occlusion of the moving surface. Occasionally, a pair of forceps or surgical threading material moves across the heart surface temporarily obstructing a part of the surface. This occurs mainly in the lower left area of the frame.

The files were provided in a MATLAB data file format (.mat). Each file contains approximately 2,430 grayscale, 8-bit image frames at 800 x 800 resolution captured at 81 frames per second (fps). This corresponds to a total image sequence time of 30 seconds.

The original filenames are: 20160725T124323.mat, 20160725T124609.mat, 20160725T124909.mat, 20160725T132840.mat, 20160725T133113.mat. The filenames correspond to the date and time of image capture.

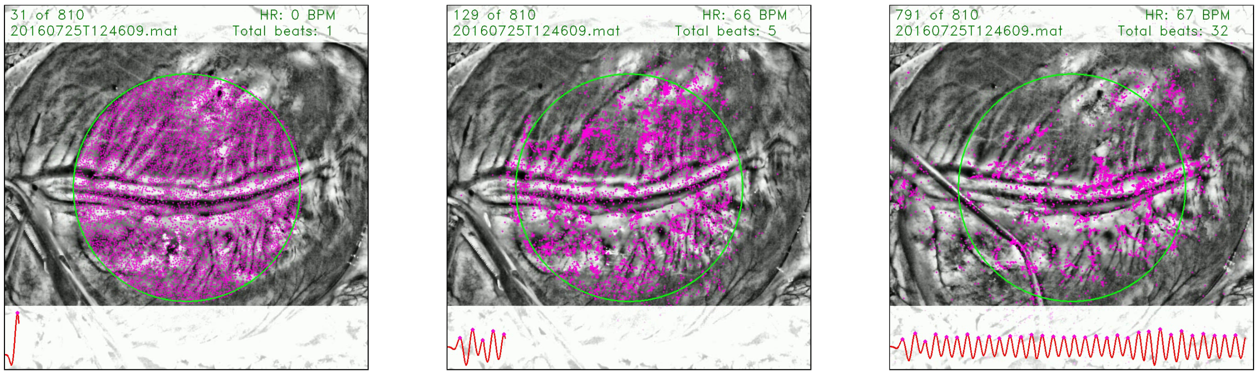
A MATLAB script (Appendix A.6) was created to convert the .mat image sequence files into uncompressed audio-video interleave (.avi) video files that can be read by the OpenCV file input algorithms.

To increase the speed of processing, each file was downsampled by a factor of 3 to 27 fps. The original files were captured at a high camera frame rate that is not necessary for accurate tracking and 27 fps is closer to the 30 fps framerate that is typical of many commonly available cameras.

3.4.2 Extraction Method / Algorithm

An OpenCV Python script (Appendix A.2) was created to perform feature point detection and tracking on each video and save each point sequence to a NumPy data file (.npy). This tracking script uses the pulse detection method described in Section 3.3.5 to locate the peak values of the flow corresponding the systole and diastole phases of the cycle.

Initial feature detection will begin at the first diastole frame and tracking will continue through the remainder of the video as shown in Figure 3.7. Point sequences that the tracking algorithm designates with a "lost" status will not be saved.



(a) Start of Tracking (b) Middle of Tracking (c) End of Tracking

Figure 3.7: Sample of Extracted Feature Points Across Tracking Period.

The points are detected in a circular region in the center of the image to avoid collecting unwanted data from tracking points on the surgical instruments or from stationary tissues. This will limit predictions to the central area of the image frame.

Each frame is pre-processed before feature detection and tracking using the equalization and denoising techniques described in Section 3.3.

3.4.3 Sequence Length

The continuous data sequence for each point can be divided up into sub-sequences of arbitrary length. These smaller sub-sequences will be used as training data for the neural network.

A sub-sequence length of 10 time steps (frames) was chosen. At a heart rate of approximately 70 bpm and frame rate of 27 fps, each cardiac cycle would cover approximately 32 frames and each 10 step sub-sequence would contain information for approximately 1/3 of a cardiac cycle.

With approximately 7000 points detected in each video and 810 frames per video, this would generate over 5.6 million training sequences per video and over 28 million sequences for the entire data set.

Table 3.1 shows a typical example of an extracted point sequence that will used to train the neural network. Out of the 10 steps captured, the first nine points will be the training features (used to set the weights of the network), and the last point will be a training label (target predicted output of the network). Note that even though pixel image coordinates are normally expressed in integer values, the tracking algorithm uses subpixel (decimal) values to estimate pixel locations. Training and tracking did not perform well when these values were rounded to integers likely due to accumulation of rounding errors.

Table 3.1: Extracted Sequence Example.

Data Type	Sequence Step	Coordinate Values (x,y)
Feature	1	[381.06005859, 410.21243286]
	2	[380.56082153, 415.64233398]
	3	[380.24166870, 424.26119995]
	4	[387.43267822, 427.93359375]
	5	[396.51394653, 432.11978149]
	6	[400.73373413, 434.64465332]
	7	[389.73266602, 428.98709106]
	8	[369.53717041, 414.84133911]
	9	[349.72045898, 402.50061035]
Label	10	[342.07440000, 400.10553000]

3.4.4 Scaling

While not always required, neural networks typically perform better with normalized data (scaled to -1 to 1 or 0 to 1) or standardized data with a zero mean and unit variance [52].

The entire data set will be standardized using the StandardScaler function of Scikit-learn before it is used in the neural network. Outputs of the neural network will thus be standardized and require inverse scaling to return to image coordinates. The scaler is fit only on the training data set to maintain data independence.

Data should also normally be transformed to remove stationarity [52], but, in this case since the time sequences cover a short period without significant trends no stationarity transform is added.

3.5 Neural Network

This section describes the selected neural network architecture and training parameters.

3.5.1 Architecture

As described in Section 2.2.3, MLPs, RNNs, and CNNs can all be applied to learn patterns in sequential data. This research will focus on the use of the RNN due to its strengths in dealing with variable length inputs and the capability to generate variable outputs. While we will only predict one future sequence point at a time with each new frame, multiple-step predictions could be useful in further experiments that expand upon this research.

The multivariate autoregression and CNN networks should have an advantage in their ability to better capture spatial dependencies between features compared to a single sequence prediction model. However, because of the variable ROI selection and non-ordered nature of feature point detection used in the baseline algorithm, these networks are not practical. The autoregression and CNN networks assume a fixed number of inputs or grid points, but the baseline algorithm can have a variable number of patches and grid points determined at the time of ROI selection. Collection of training data would be complicated by the large number of grid sequences that would need to be collected in order to cover the entire range of possible grid permutations. In addition, if the prediction of a series of tracked point features is desired rather than ROI grid points, there would be no *a priori* knowledge of the number or order of features nor any consistent structure to the feature locations since any ROI location could be selected. Individual grid point prediction could be desirable in a future iteration where the ROI is partially obscured and additional statistical checks are required for a single patch.

The RNN will be structured with GRU cells to improve speed due to the reduced number of operations (See Section 2.2.3). The model will consist of two GRU layers and one dense layer as shown in Table 3.2.

Table 3.2: RNN Model Layers.

Number	Layer Type	Activation	Output Shape	Parameter Number
1	GRU	ReLU	(None, None, 64)	13,056
2	GRU	ReLU	(None, 32)	9,408
3	Dense	Linear	(None, 2)	66
			Total Parameters:	22,530
			Trainable Parameters:	22,530
			Non-trainable Parameters:	0

The input GRU layer will have 64 units (nodes), the second GRU layer will have 32 units, and the final Dense layer has 2 units to match the size of the desired output (x and y point coordinates).

Each GRU layer uses the default "ReLU" activation function. ReLU stands for Rectified Linear Unit and is a type of function applied to transform each node summation value to an output value in a way that minimizes computations and allows for larger networks that do not suffer from vanishing gradients [20]. The final layer uses a "Linear" activation function which is recommended for regression outputs [53].

Initial experiments added dropout layers after each GRU layer which can aid in reducing overfitting via a sampling process that simulates an average of many possible network configurations [54]. However, prediction performance for this data set was improved by removing these dropout layers. Predictions using the dropout layer model had large deformation and location errors.

3.5.2 Training

The network will be trained using a split of 80% of the data set as training data and 20% of the data set as testing (validation) data. The testing data will be held back during each training iteration (epoch) and used as a test of how well the network was trained on the other 80%. One measure of optimal training for a neural network is when the overall network losses on the training data are approximately equal to the losses on the testing data [55]. Before the data split, the point sequences are shuffled so that the training and test data are randomly distributed.

The losses will be calculated using the Mean Squared Error (MSE) metric which emphasizes large errors in the predictions and results in worse loss values for these cases [56]. Because the network is trained on individual point sequences, these losses would be related to an individual point prediction and not the overall grid.

The accuracy of the model is a percentage value calculated based on a comparison of how well the model has predicted the last sequence point during inference compared with the training label.

The Adaptive Momentum (AdaM) optimization algorithm was chosen for training as this is becoming more of an industry standard [57]. The optimization algorithm is the particular convergence method used to set the network weights while minimizing the overall loss through the network. AdaM adds some enhancements to the traditional gradient descent approaches to dampen the convergence to a system minimum and thereby greatly speed up training [58].

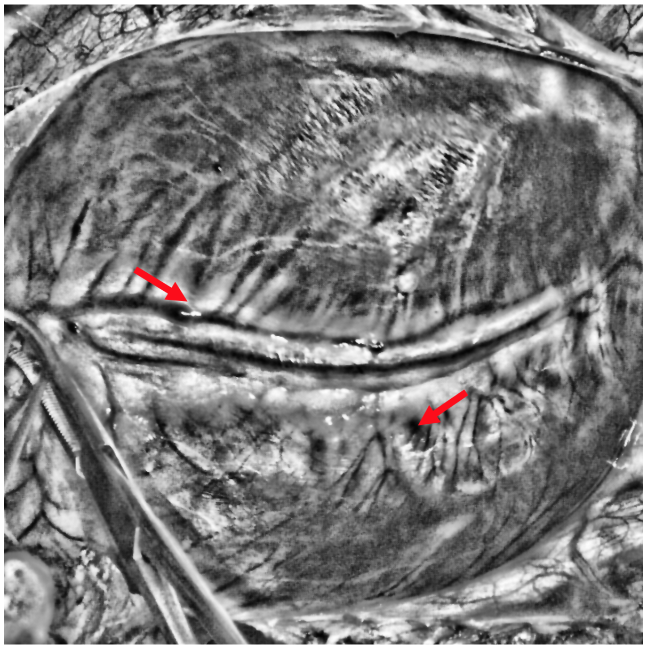


Figure 3.8: ROI Selection Landmarks. Repeatable points were chosen for use in the manual ROI selection step for each video. The left arrow points to the intersection of the larger artery with a smaller branch. The right arrow points to the middle of a large, circular dark region.

3.6 Testing Scenarios

Because each video will have some variation in the heart surface and orientation at diastole, each tracking sequence will be initiated with manual selection of a ROI based on repeatable landmarks. Figure 3.8 shows the landmarks chosen for the experiments presented in this research. These landmarks were chosen to be near the center of the frame (within the feature extraction ROI - See Section 3.4) and contain a region that may be more clinically relevant (coronary artery). The upper left landmark is at the intersection of the coronary artery and the indicated vessel branch and the lower right landmark is near the center of the dark circular region indicated.

The performance of the grid prediction will be measured over 30 frames. At each frame, the predicted grid points will be added to the grid point sequence history and used as the next input sequence for prediction. Frames 201 to 230 will be selected for the prediction period in each video. The heart cycles are not synchronized across the videos so this frame range will begin at different points in the cardiac cycle. After the prediction period, normal tracking will continue until the end of the video.

CHAPTER 4

RESULTS

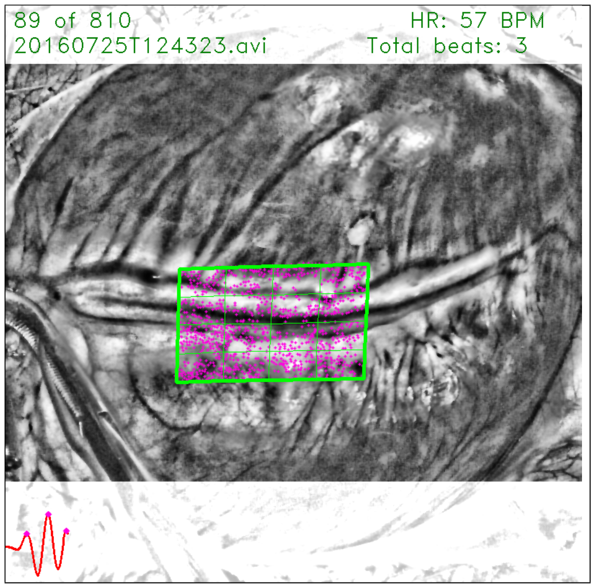
The following sections provide a summary of the tracking and grid prediction results for each data set.

A series of representative screenshots from normal tracking (green grid) and predictive tracking (yellow grid) are shown for each data set. At the top of each screenshot is an overlay that displays the current frame number, the data set name, the estimated heart rate, and the total number of beats counted (diastole peaks). At the bottom of each screenshot is an overlay showing a plot of the pulse cycle (based on overall optical flow) with frame markers above. Each black (+) marker on the bottom overlay represents a frame with failed constraints. Each red (△) marker represents a tracking re-initialization due to multiple consecutive constraint failures.

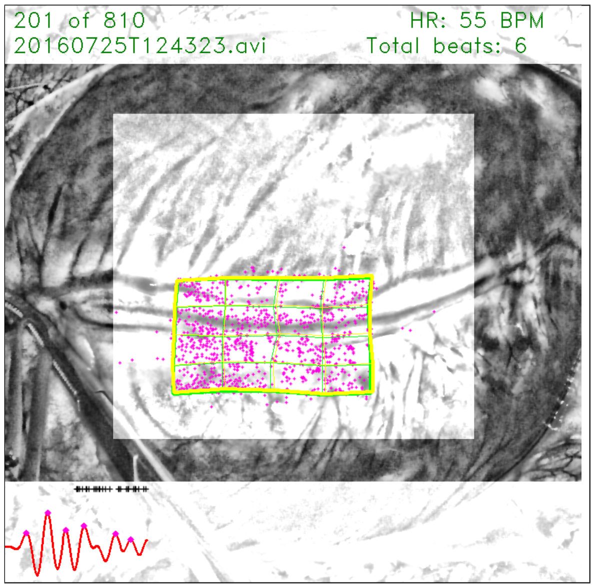
A summary of tracking results is presented in a table and in a timeline graph. Each of these figures show: the total number of frames in the video, the number of frames where the tracking algorithm was active, the number of diastole peaks detected (and corresponding homography re-initialization), the number tracking resets (undesired homography re-initialization due to consecutive failure of the tracking constraints), the number of frames that had a failure of one or more constraints (flag raised), and the number of frames where the grid prediction was active. The tracking resets and failed constraints percentages are relative to the number of tracked frames.

The prediction performance is presented in a graph that shows the distance between the predicted grid points and the actively tracked grid points (ground truth) versus the predicted frame number. These values are shown as differences in the x-direction only, the y-direction only, and the absolute Euclidean distance in the x-y plane. The average of all the differences across the entire grid is shown by the main marker. Error bars have been added to each marker to show the maximum and minimum grid differences for each ROI prediction. This allows the graph to show the overall spread of differences for the entire ROI without showing each individual grid difference.

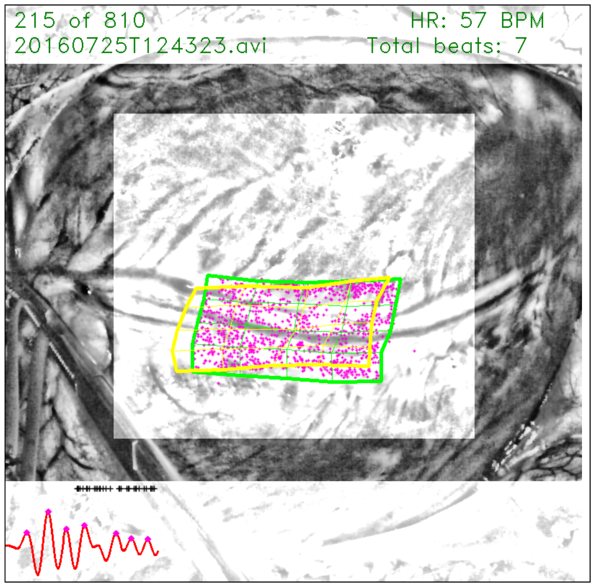
4.1 Results for 20160725T124323.avi



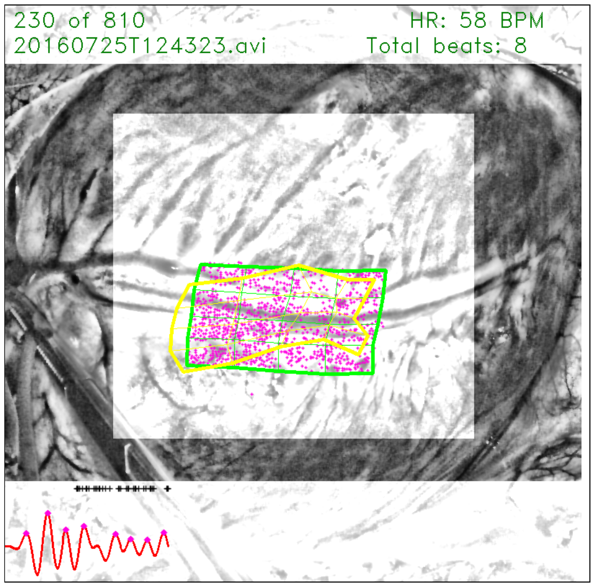
(a) Tracking Screenshot - Start



(b) Prediction Screenshot - Start



(c) Prediction Screenshot - Middle



(d) Prediction Screenshot - End

Figure 4.1: Representative Screenshots (20160725T124323.avi).

Figure 4.1 shows that the prediction begins close to the tracked ROI and continues to deviate as the prediction continues. In the middle of the prediction, the predicted grid has a similar shape as the ground truth. By the last prediction frame, the predicted grid has remained centered in the same area, but it is further deformed compared to the ground truth.

Table 4.1: Tracking Performance Summary (20160725T124323.avi).

Parameter	Total	% of Total
Total Frames	810	-
Tracked Frames	721	89.0 %
Diastole Peaks	32	4.0 %
Tracking Resets	4	0.6 %
Flag Raised	234	32.5 %
Predicted Frames	30	4.0 %

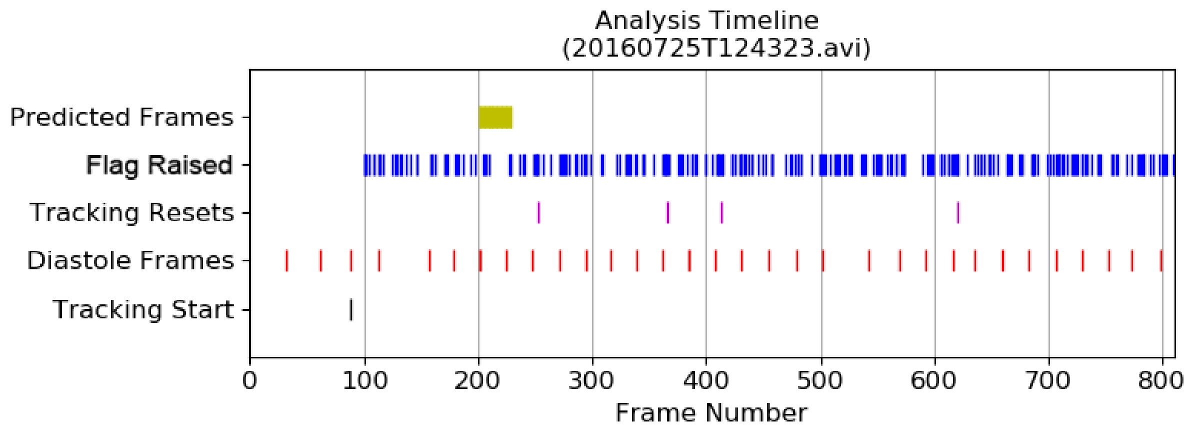


Figure 4.2: Analysis Timeline (20160725T124323.avi).

Figure 4.2 shows that there are a large number of frames where the optical flow has not passed both of the tracking constraint criteria (flag raised) but only 4 frames where the tracking was forced to reset.

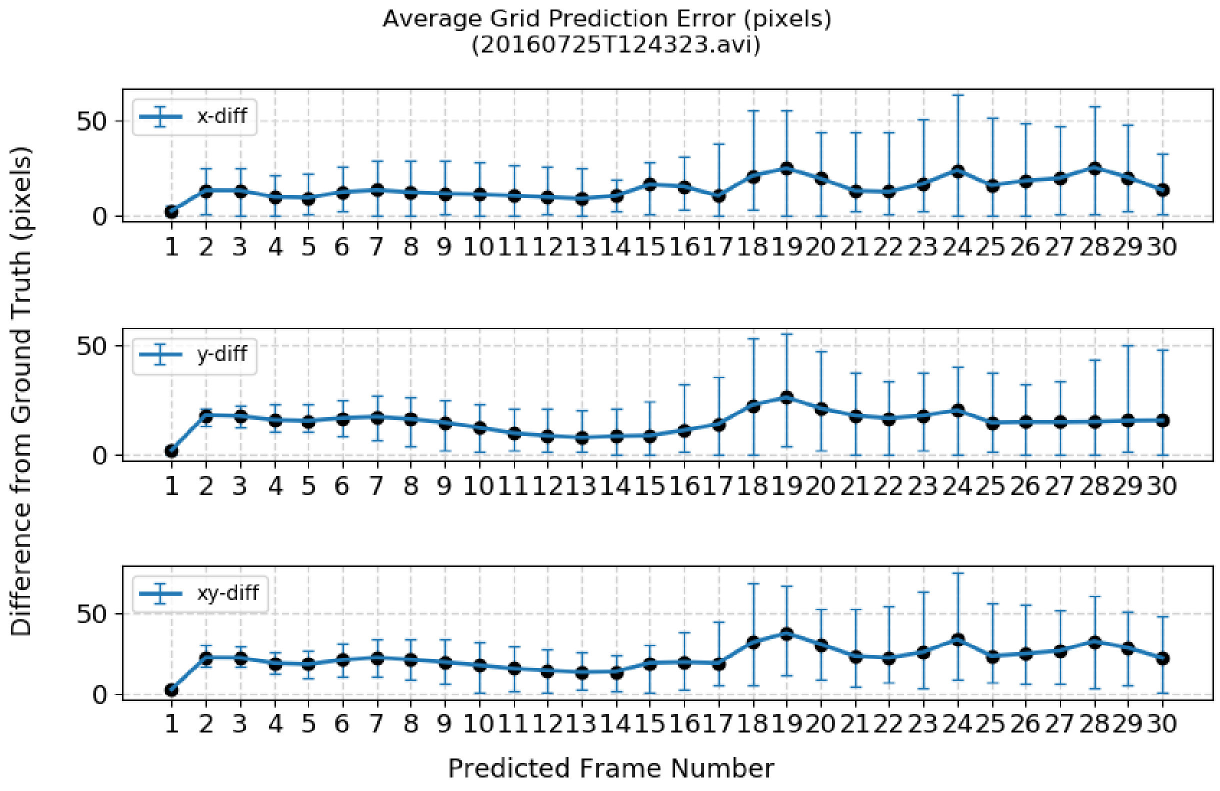
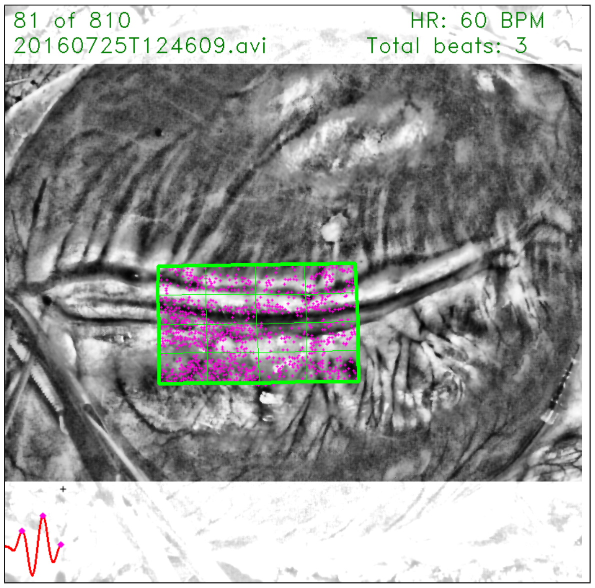


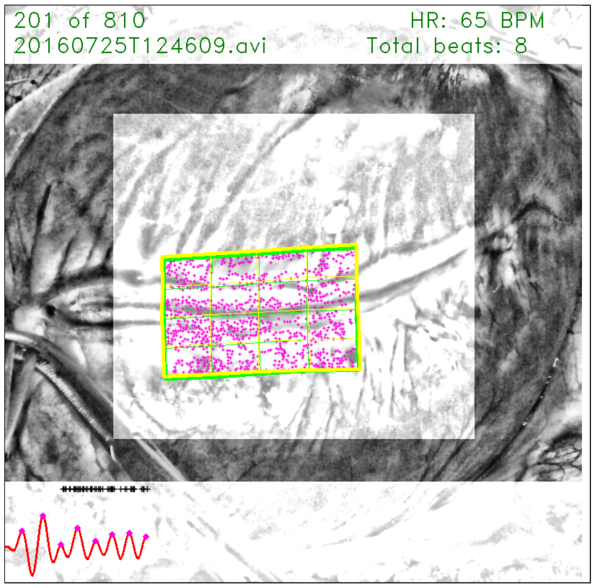
Figure 4.3: Average Grid Prediction Error (x, y, and x-y [absolute]). Error bars show max and min error values. Image is 800 x 800 pixels. Initial ROI is ~160 x 280 pixels. (20160725T124323.avi).

Figure 4.3 shows that the average difference, as well as the spread of the differences, generally increases as the prediction time increases. However, the graph shows that there are periods during which the errors decrease as the cyclic motion changes direction. This is more evident in the video output files. The average error remains below 50 pixels throughout the prediction and remains consistent in the x- and y-directions.

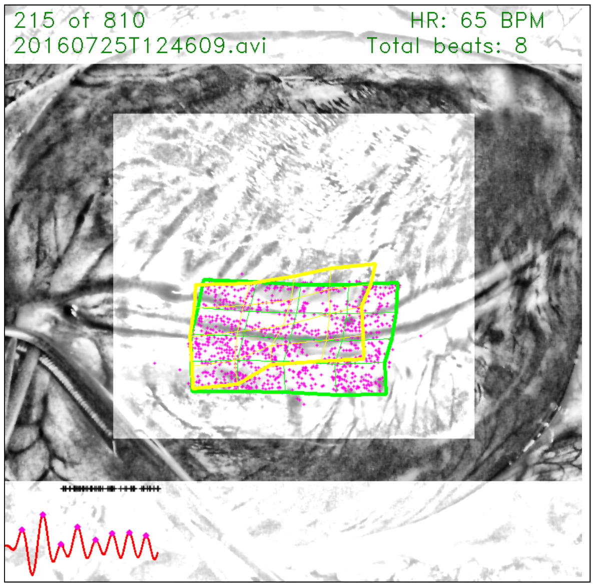
4.2 Results for 20160725T124609.avi



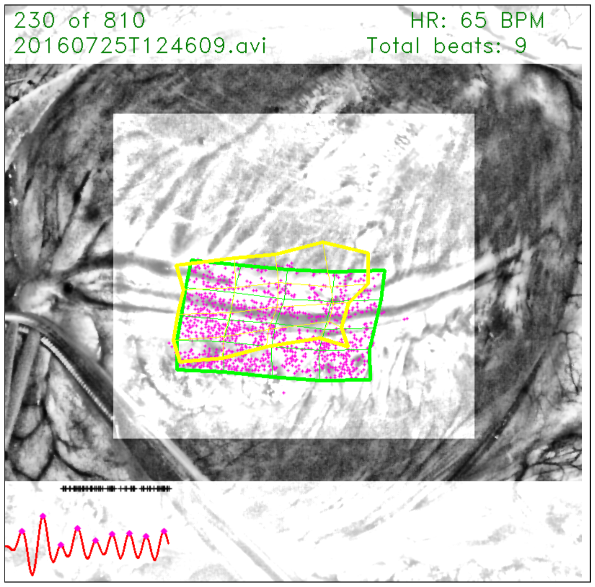
(a) Tracking Screenshot - Start



(b) Prediction Screenshot - Start



(c) Prediction Screenshot - Middle



(d) Prediction Screenshot - End

Figure 4.4: Representative Screenshots (20160725T124609.avi).

Figure 4.4 shows that the prediction begins close to the tracked ROI and continues to deviate as the prediction continues. In the middle of the prediction, the predicted grid has moved in a similar direction as the ground truth, but it shows a large amount of deformation. By the last prediction frame, the predicted grid has a similar amount of deformation as the middle frame.

Table 4.2: Tracking Performance Summary (20160725T124609.avi).

Parameter	Total	% of Total
Total Frames	810	-
Tracked Frames	729	90.0 %
Diastole Peaks	34	4.2 %
Tracking Resets	1	0.1 %
Flag Raised	255	35.0 %
Predicted Frames	30	4.0 %

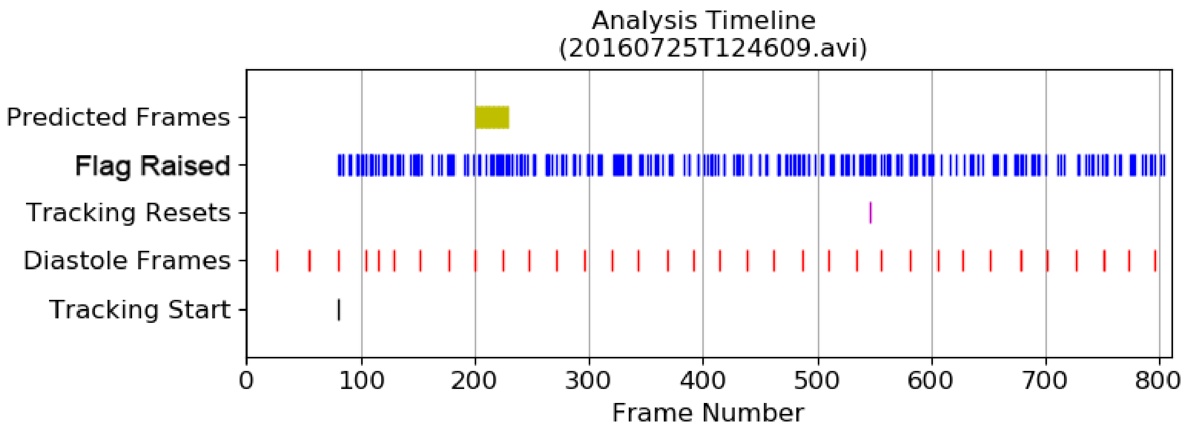


Figure 4.5: Analysis Timeline (20160725T124609.avi).

Figure 4.5 shows that there are a large number of frames where the optical flow has not passed both of the tracking constraint criteria (flag raised) but only 1 frame where the tracking was forced to reset.

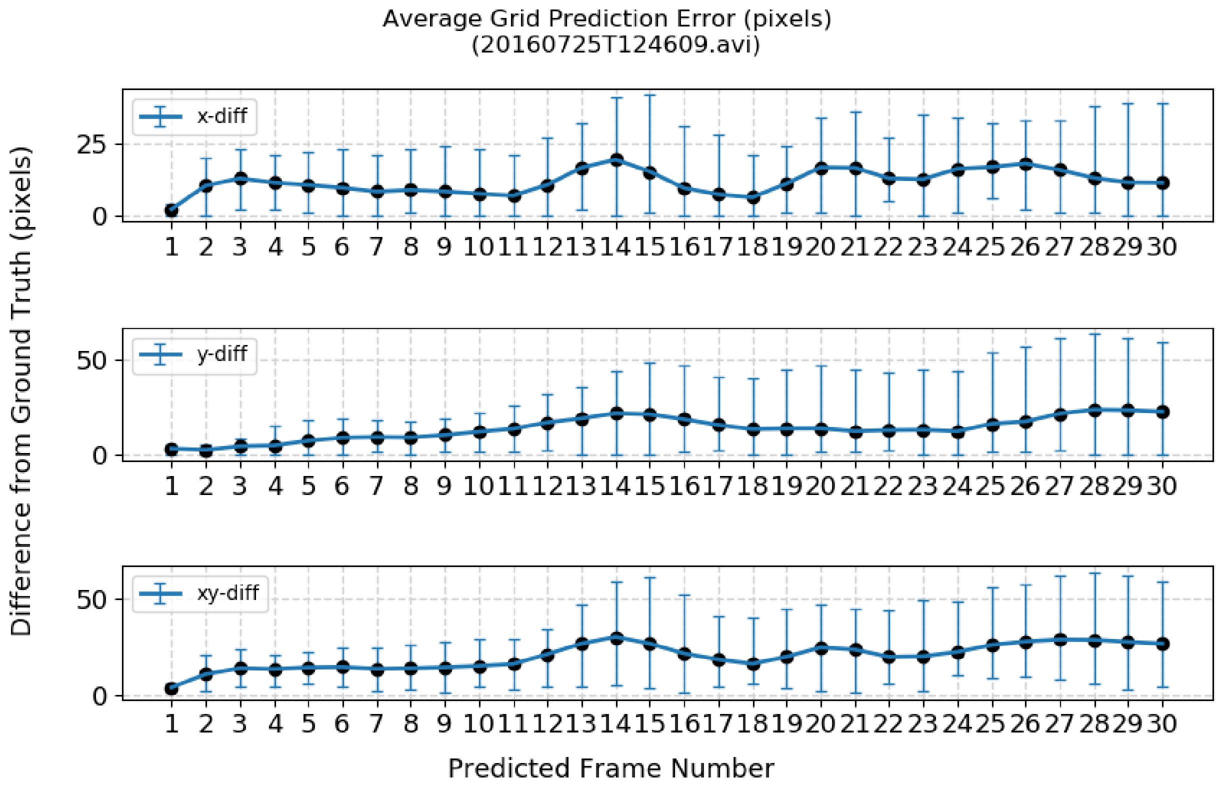
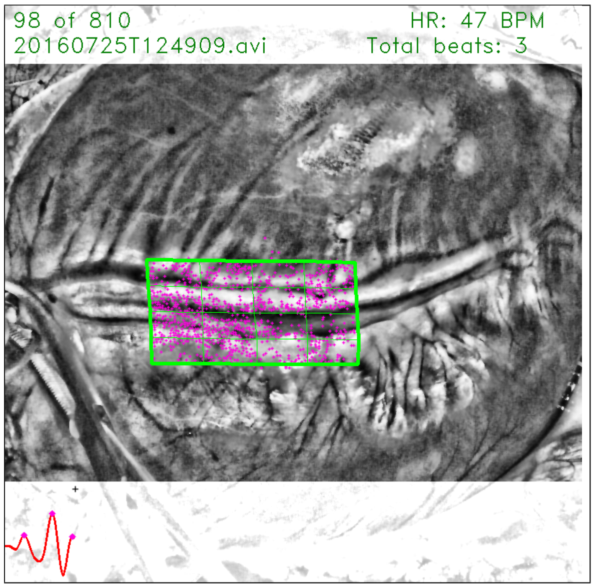


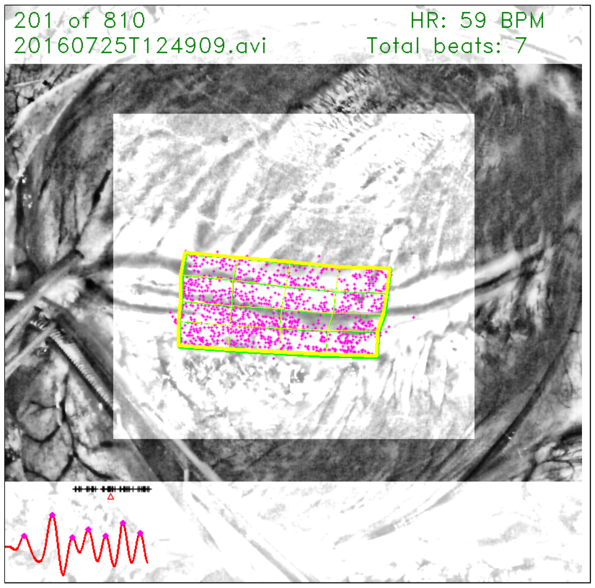
Figure 4.6: Average Grid Prediction Error (x, y, and x-y [absolute]). Error bars show max and min error values. Image is 800 x 800 pixels. Initial ROI is ~160 x 280 pixels. (20160725T124609.avi).

Figure 4.6 shows that the average difference, as well as the spread of the differences, generally increases as the prediction time increases. However, the graph shows that there are periods during which the errors decrease as the cyclic motion changes direction. This is more evident in the video output files. The average error remains below 50 pixels throughout the prediction. The errors are lower in the x-direction than in the y-direction.

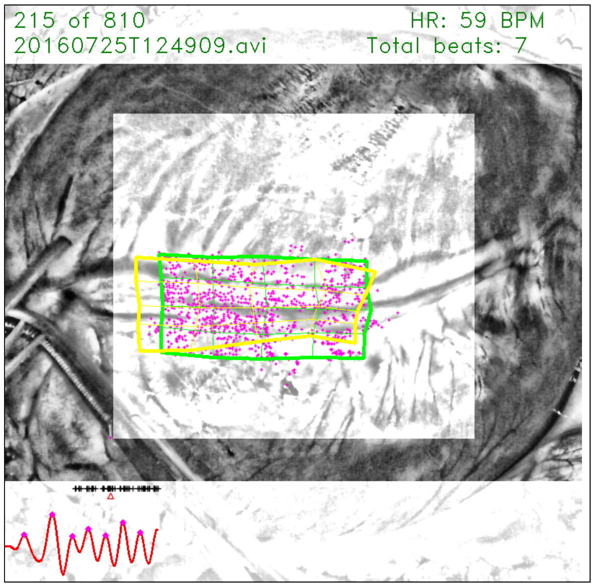
4.3 Results for 20160725T124909.avi



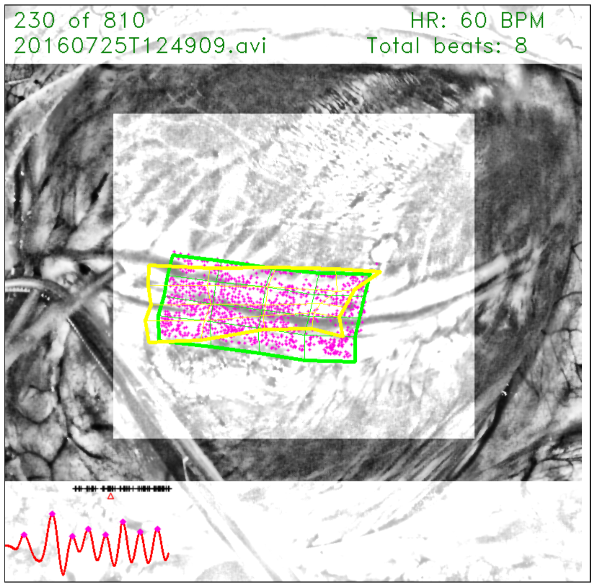
(a) Tracking Screenshot - Start



(b) Prediction Screenshot - Start



(c) Prediction Screenshot - Middle



(d) Prediction Screenshot - End

Figure 4.7: Representative Screenshots (20160725T124909.avi).

Figure 4.7 shows that the prediction begins close to the tracked ROI and continues to deviate as the prediction continues. In the middle of the prediction, the predicted grid has moved in a similar direction as the ground truth with some deformation. By the last prediction frame, the predicted grid error and deformation has further increased.

Table 4.3: Tracking Performance Summary (20160725T124909.avi).

Parameter	Total	% of Total
Total Frames	810	-
Tracked Frames	712	87.9 %
Diastole Peaks	35	4.3 %
Tracking Resets	6	0.8 %
Flag Raised	304	42.7 %
Predicted Frames	30	4.1 %

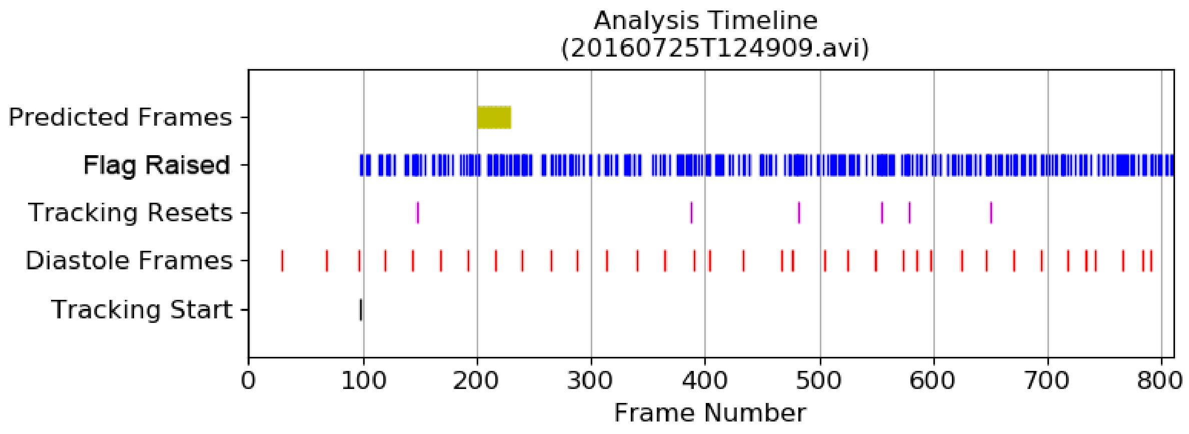


Figure 4.8: Analysis Timeline (20160725T124909.avi).

Figure 4.8 shows that there are a large number of frames where the optical flow has not passed both of the tracking constraint criteria (flag raised) but only 6 frames where the tracking was forced to reset.

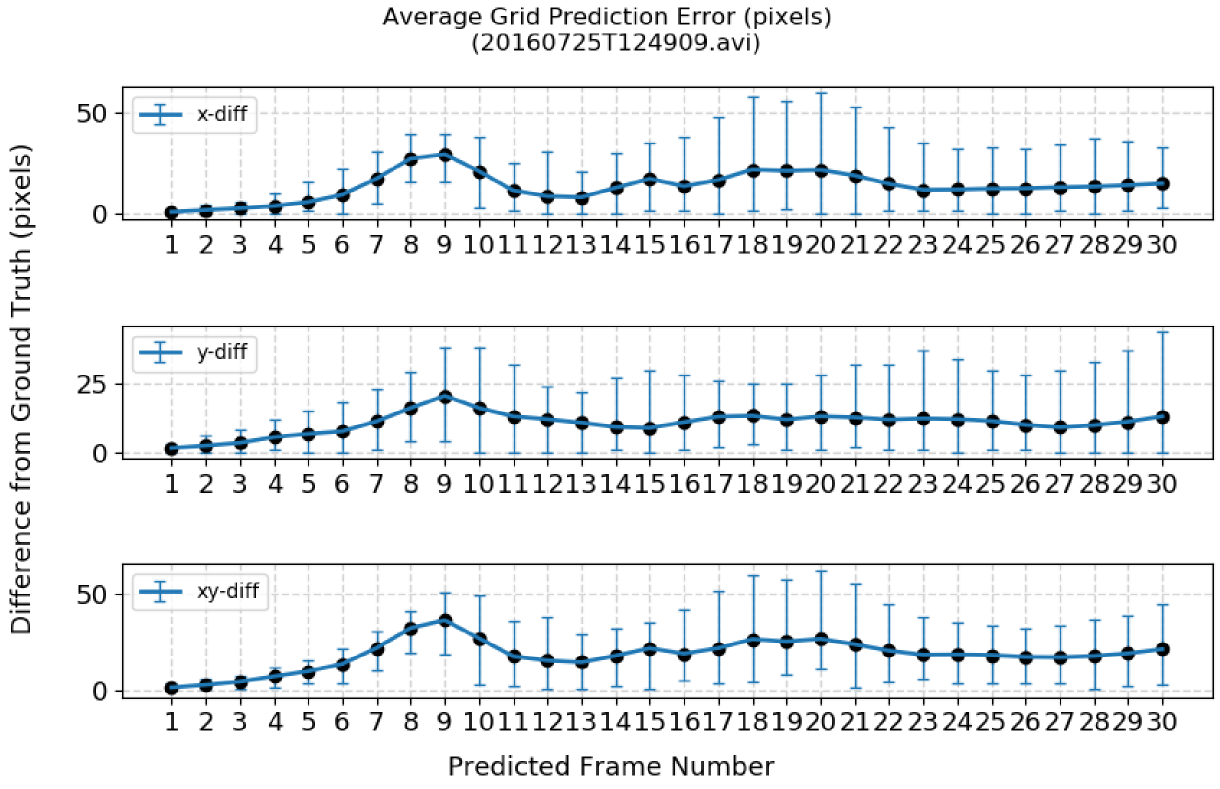
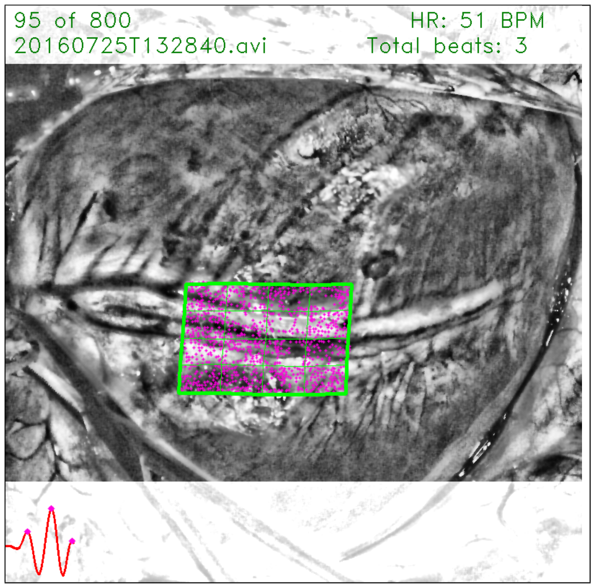


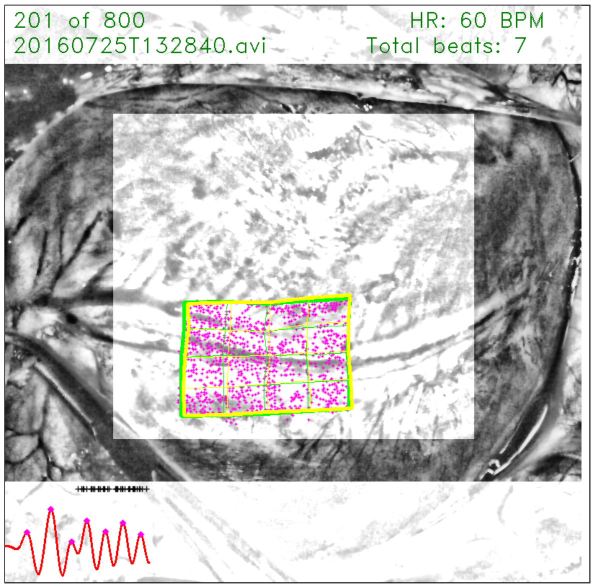
Figure 4.9: Average Grid Prediction Error (x, y, and x-y [absolute]). Error bars show max and min error values. Image is 800 x 800 pixels. Initial ROI is ~160 x 280 pixels. (20160725T124909.avi).

Figure 4.9 shows that the average difference, as well as the spread of the differences, generally increases as the prediction time increases. However, the graph shows that there are periods during which the errors decrease as the cyclic motion changes direction. This is more evident in the video output files. The average error remains below 50 pixels throughout the prediction. The errors are lower in the y-direction than in the x-direction.

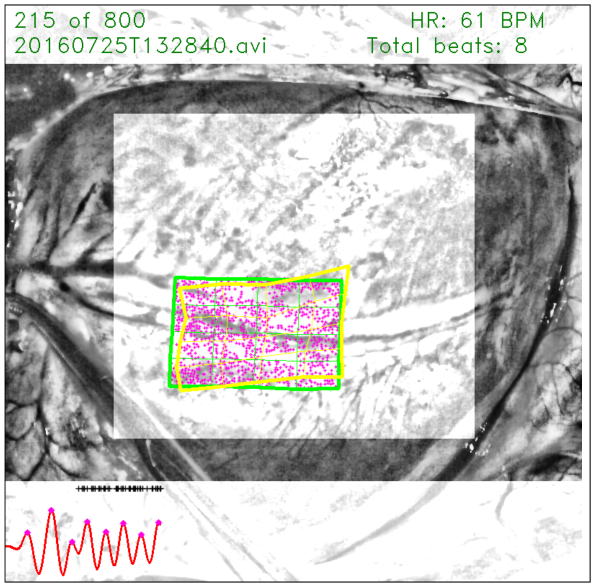
4.4 Results for 20160725T132840.avi



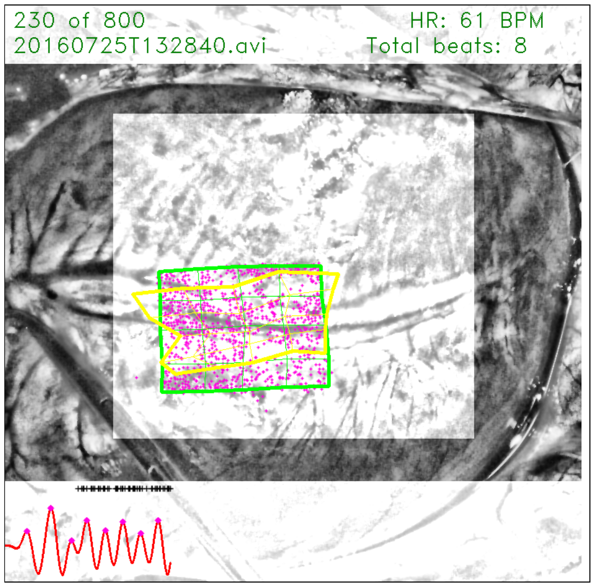
(a) Tracking Screenshot - Start



(b) Prediction Screenshot - Start



(c) Prediction Screenshot - Middle



(d) Prediction Screenshot - End

Figure 4.10: Representative Screenshots (20160725T132840.avi)

Figure 4.10 shows that the prediction begins close to the tracked ROI and continues to deviate as the prediction continues. In the middle of the prediction, the predicted grid has moved in a similar direction as the ground truth and has a very similar shape. By the last prediction frame, the predicted grid error has increased, and the grid shape shows larger deformation.

Table 4.4: Tracking Performance Summary (20160725T132840.avi).

Parameter	Total	% of Total
Total Frames	810	-
Tracked Frames	705	88.1 %
Diastole Peaks	40	5.0 %
Tracking Resets	4	0.6 %
Flag Raised	257	36.5 %
Predicted Frames	30	4.1 %

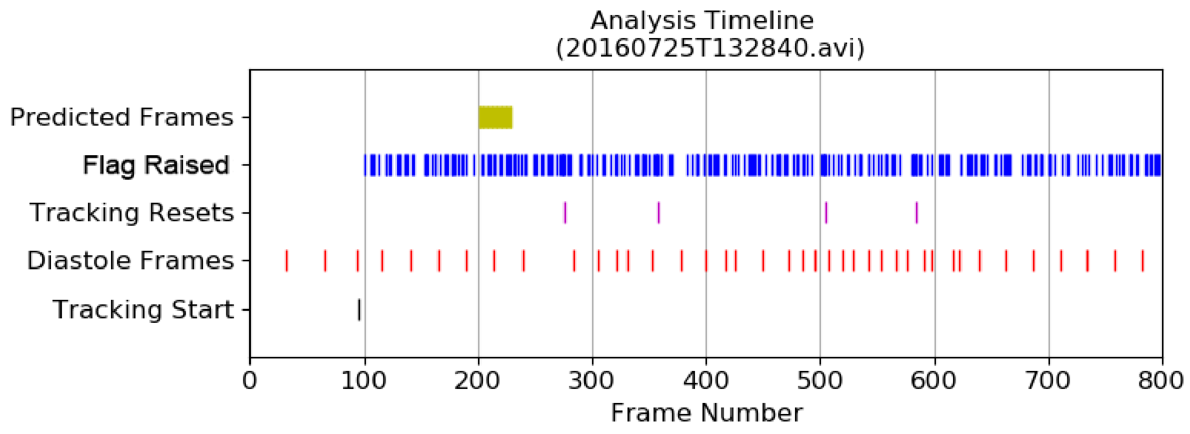


Figure 4.11: Analysis Timeline (20160725T132840.avi).

Figure 4.11 shows that there are a large number of frames where the optical flow has not passed both of the tracking constraint criteria (flag raised) but only 4 frames where the tracking was forced to reset.

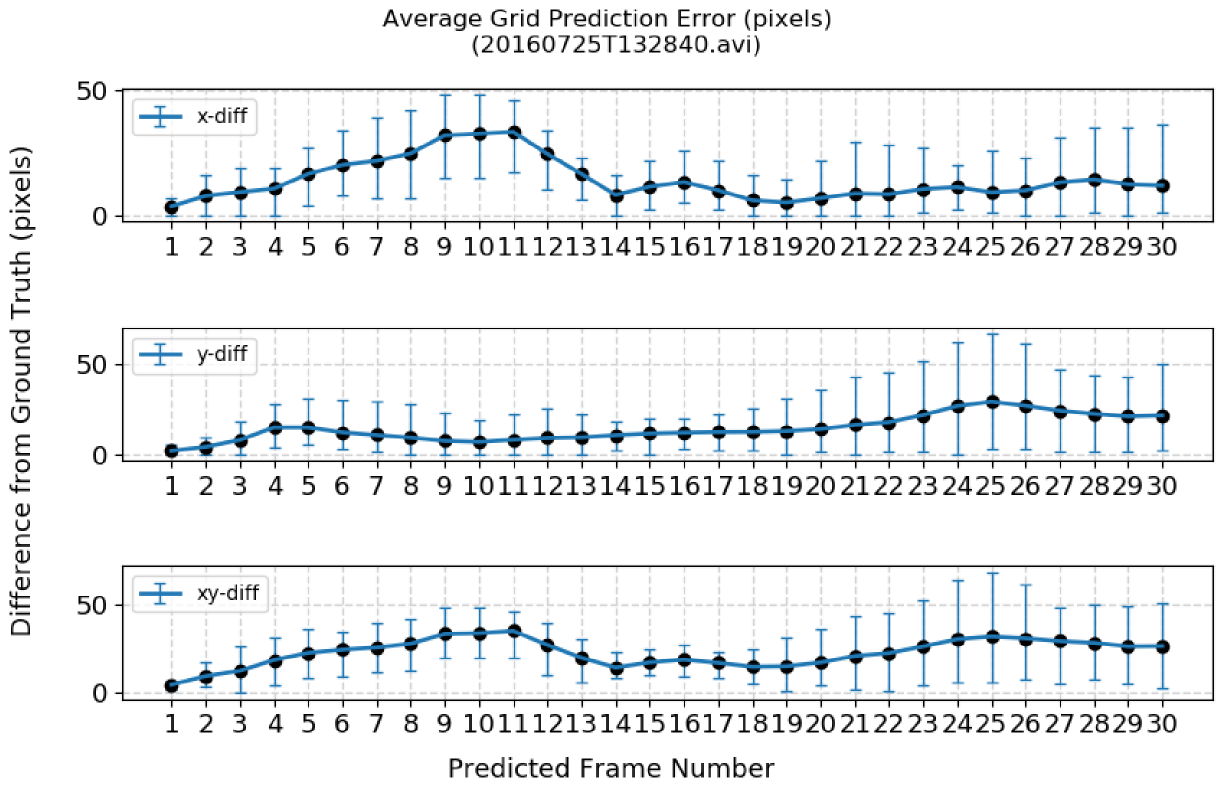
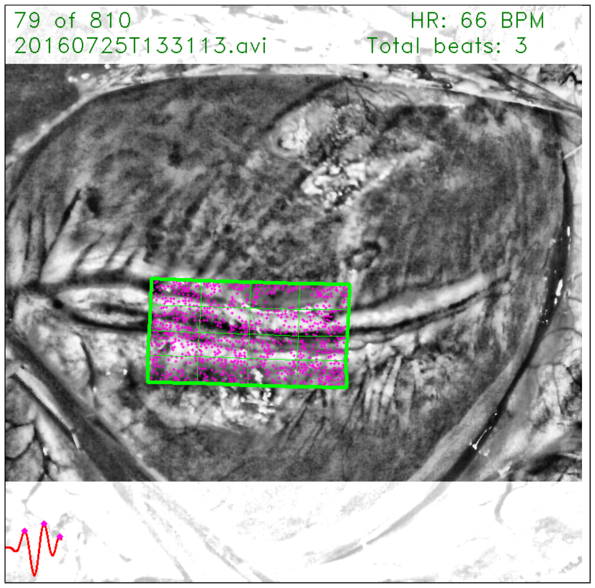


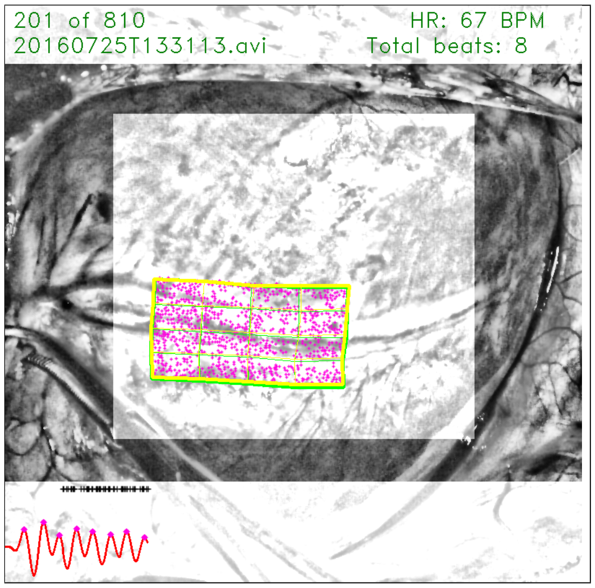
Figure 4.12: Average Grid Prediction Error (x, y, and x-y [absolute]). Error bars show max and min error values. Image is 800 x 800 pixels. Initial ROI is ~160 x 280 pixels. (20160725T132840.avi).

Figure 4.12 shows that the average difference, as well as the spread of the differences, generally increases as the prediction time increases. However, the graph shows that there are periods during which the errors decrease as the cyclic motion changes direction. This is more evident in the video output files. The average error remains below 50 pixels throughout the prediction. The errors are lower in the y-direction than in the x-direction.

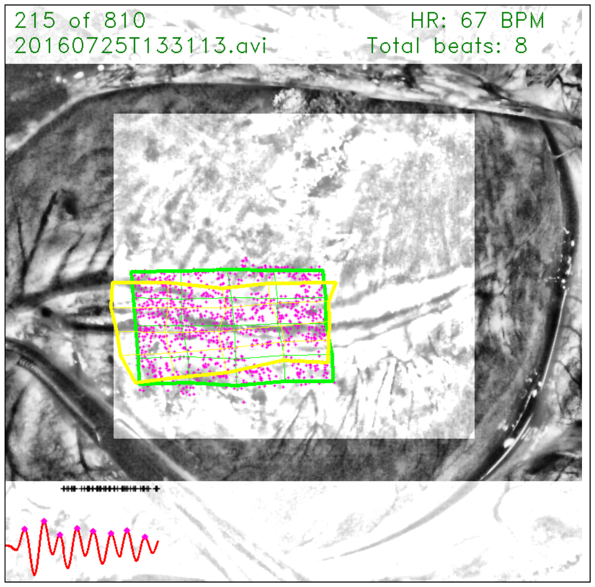
4.5 Results for 20160725T133113.avi



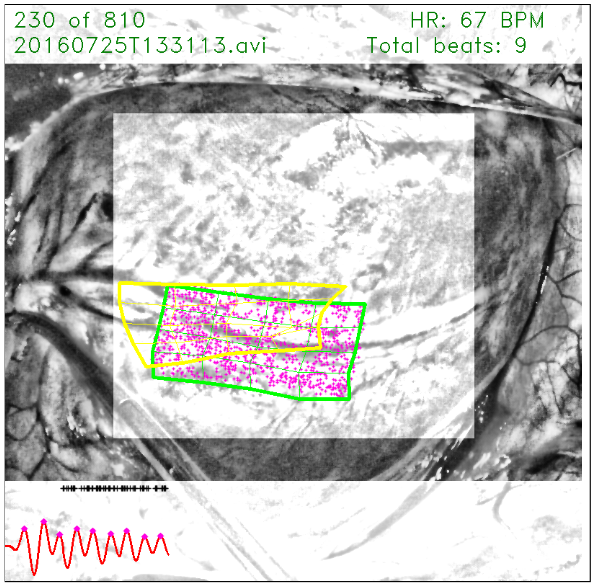
(a) Tracking Screenshot - Start



(b) Prediction Screenshot - Start



(c) Prediction Screenshot - Middle



(d) Prediction Screenshot - End

Figure 4.13: Representative Screenshots (20160725T133113.avi).

Figure 4.13 shows that the prediction begins close to the tracked ROI and continues to deviate as the prediction continues. In the middle of the prediction, the predicted grid has moved in a similar direction as the ground truth and shows a moderate amount of deformation. By the last prediction frame, the predicted grid has a large error and deformation.

Table 4.5: Tracking Performance Summary (20160725T133113.avi).

Parameter	Total	% of Total
Total Frames	810	-
Tracked Frames	731	90.2 %
Diastole Peaks	33	4.1 %
Tracking Resets	6	0.8 %
Flag Raised	297	40.6 %
Predicted Frames	30	4.0 %

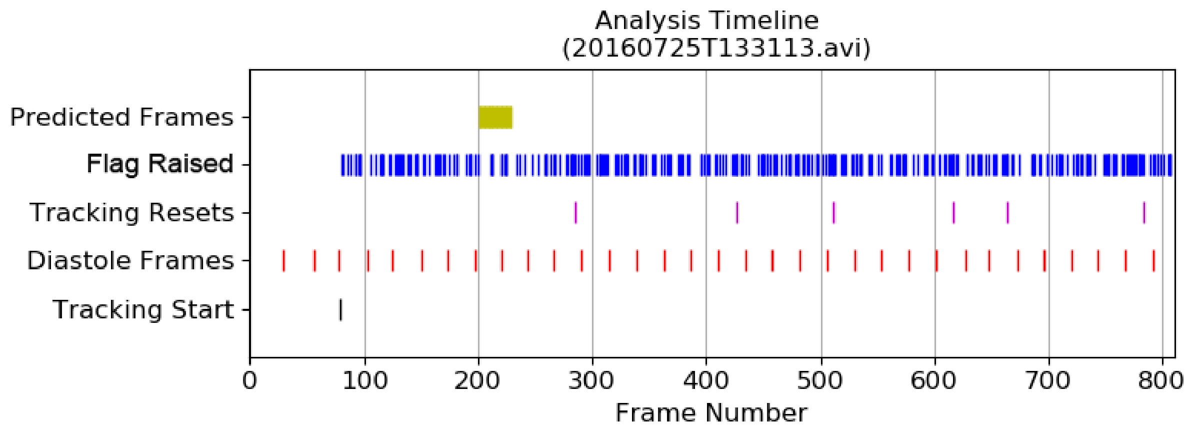


Figure 4.14: Analysis Timeline (20160725T133113.avi).

Figure 4.14 shows that there are a large number of frames where the optical flow has not passed both of the tracking constraint criteria (flag raised) but only 6 frames where the tracking was forced to reset.

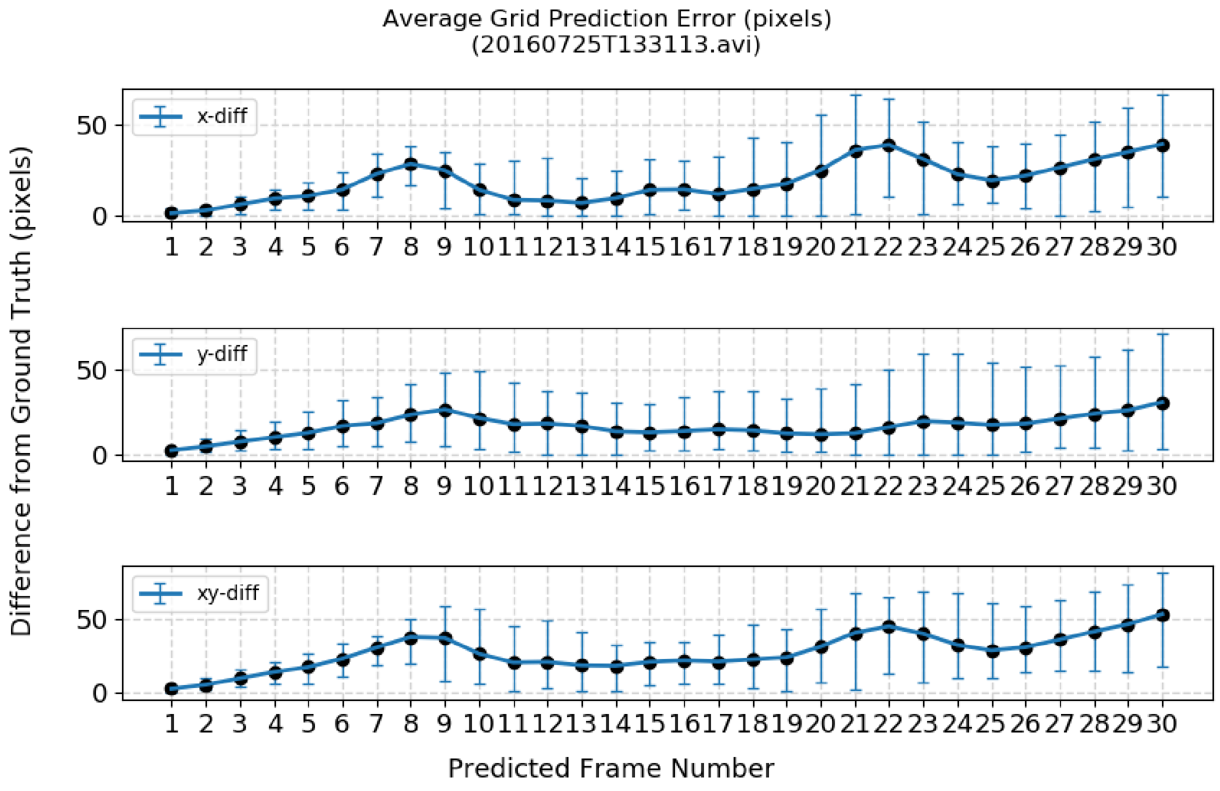


Figure 4.15: Average Grid Prediction Error (x, y, and x-y [absolute]). Error bars show max and min error values. Image is 800 x 800 pixels. Initial ROI is ~160 x 280 pixels. (20160725T133113.avi).

Figure 4.15 shows that the average difference, as well as the spread of the differences, generally increases as the prediction time increases. However, the graph shows that there are periods during which the errors decrease as the cyclic motion changes direction. This is more evident in the video output files. The average error remains below 50 pixels throughout the prediction. The errors are in the x- and y-direction are of similar magnitude, but the x-direction shows more cyclical motion.

4.6 Neural Network Training

The neural network was trained for 5 epochs with a batch size of 512. After 5 epochs, the losses on the training and testing data sets were approximately equal as shown in Figure 4.16. The accuracy of both data sets begins to plateau after 4 epochs as shown in Figure 4.17. Training time was a total of 14 minutes. See Section 3.5.2 for a discussion of the loss and accuracy calculation.

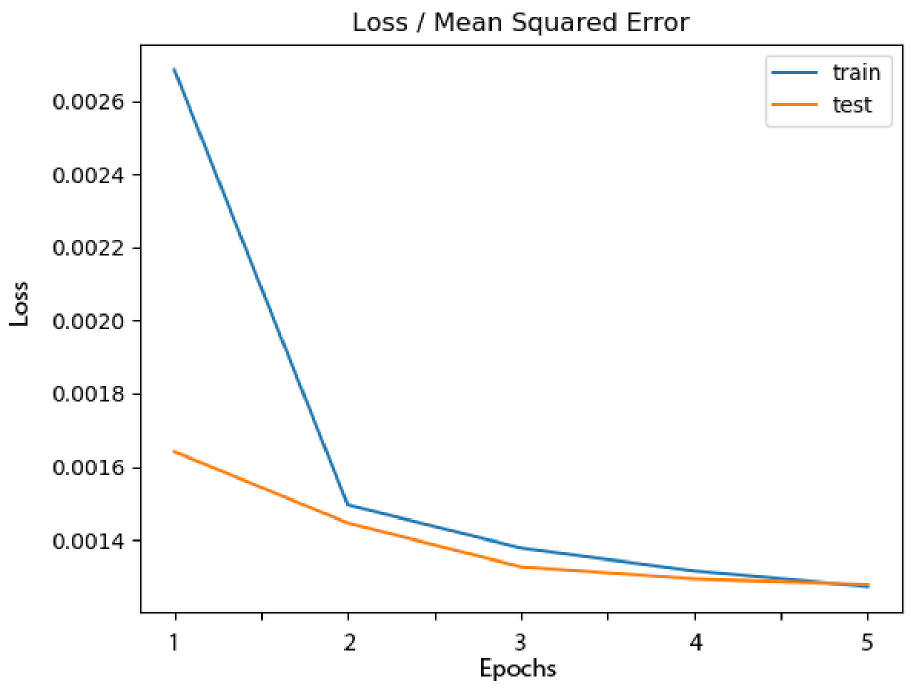


Figure 4.16: Neural Network Training Loss.

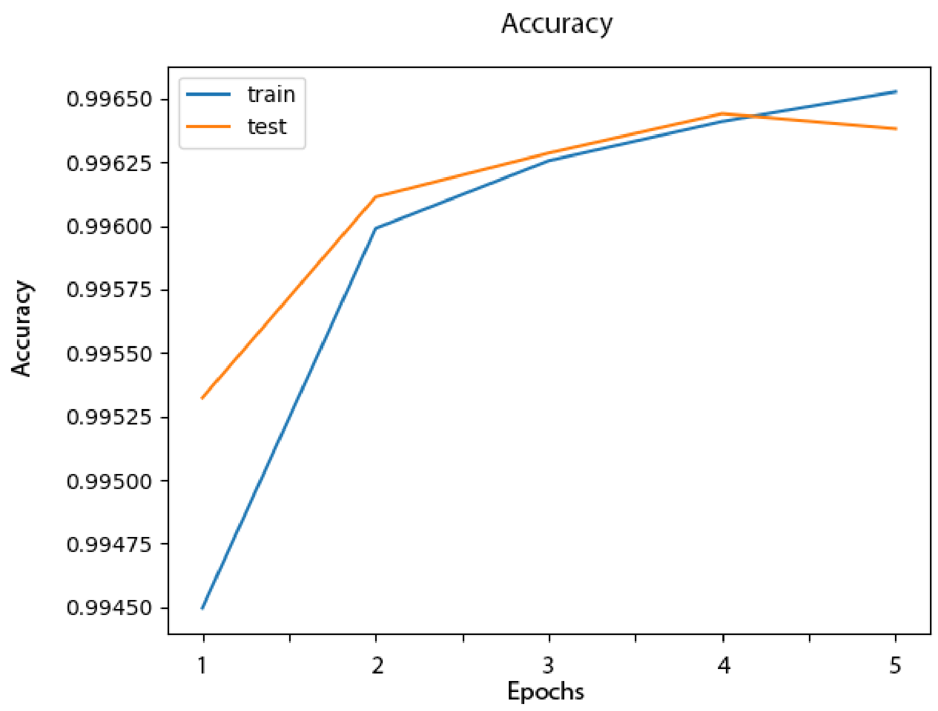


Figure 4.17: Neural Network Training Accuracy.

4.7 Speed of Execution

Table 4.6 provides a summary of the run time for each file. The file read/write time was subtracted from the total time and frame rate calculation to give a better estimate of the performance that might be expected from a live camera with better frame buffering (See Section 5.4.4).

Table 4.6: Execution Speed Summary.

Data Set	Execution Time (s)	Approximate Frame Rate (FPS)
20160725T124323.avi	55.1	13.1
20160725T124609.avi	47.0	15.5
20160725T124909.avi	51.4	13.9
20160725T132840.avi	49.1	14.4
20160725T133113.avi	47.2	15.5
Average	50.0	14.5

CHAPTER 5

DISCUSSION

This chapter discusses the results presented in Chapter 4.

5.1 Tracking Performance

While a direct comparison to the original research is not made (due to the difference in the selected ROI and change in constraint failure limit), the results from Chapter 4 demonstrate that the tracking failure rate is improved in the new implementation. Table 5.1 below gives a summary and comparison of the failure rates between the new and original algorithm.

Table 5.1: Tracking Failure Summary and Comparison.

	Tracking Resets		
	Original	New	
Data Set	Failures	Diastole	Failures
20160725T124323.avi	22.2%	4.0%	0.6%
20160725T124609.avi	16.8%	4.2%	0.1%
20160725T124909.avi	17.2%	4.3%	0.8%
20160725T132840.avi	30.8%	5.0%	0.6%
20160725T133113.avi	11.7%	4.1%	0.8%
Average	19.7%	4.3%	0.6%

Table 5.1 demonstrates that the changes to re-initializing tracking at every diastole peak greatly reduces the overall reset rate from almost 20% to 5%. If tracking can be maintained through

multiple diastole peaks without resetting, the reset rate may be further reduced. The equalization, denoising, and failure limit changes drive the resets due to failure close to zero.

5.2 Prediction Performance

The results shown in Chapter 4 demonstrate that the motion of cardiac images contained in the data set was learned by the neural network model. Table 5.2 below gives a summary and comparison of the x-y error across the range of predicted frames.

Table 5.2: Grid Prediction Error Summary.

	x-y Errors (pixels)								
	Frame 1			Frame 15			Frame 30		
Data Set	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
20160725T124323.avi	1.0	3.2	5.1	1.0	19.4	22.5	5.1	30.6	48.0
20160725T124609.avi	2.2	3.8	5.7	3.2	26.7	61.6	4.2	26.7	59.0
20160725T124909.avi	0.0	1.7	3.6	1.0	22.0	35.4	3.0	21.5	45.1
20160725T132840.avi	2.2	4.7	7.1	9.9	17.3	24.8	2.8	26.5	50.8
20160725T133113.avi	0.0	2.8	4.5	5.4	20.9	34.0	18.0	53.2	81.5
Average	1.1	3.2	5.2	4.1	21.3	35.7	6.6	31.7	56.9

Table 5.2 shows that the neural network model was able to predict the grid point locations for the overall data set with an average maximum distance error of 56.9 pixels after a 30 frame prediction period. The mean average error for all of the data sets gradually increased throughout the prediction period from 3.2 pixels to 31.7 pixels. The overall maximum error was below 81.5 pixels across the prediction period for all data sets.

The errors quickly accumulate and become large across the prediction period which may limit the practical usefulness of the prediction depending on the application. An average 50 pixel offset could mean the grid point is off by ~15-30% of the grid dimensions. In a precise tumor tracking application this offset may not be acceptable, but in longer term monitoring of tissue strain, for example, it may be acceptable. However, further training with better data or different network configurations may reduce the errors.

5.3 Speed of Execution

One of the original goals of this research was to improve the implementation of the baseline algorithm so that it could run in real-time on a live camera feed. The original research did not include any information on the algorithm execution time, but, in informal testing, the frame rate was around 5 fps. The results shown in Section 4.7 demonstrate that the new implementation improves the frame rate to around 14.5 fps.

By examining the output of the code profiling tool included with Python, other improvements can be identified. Table 5.3 shows a summary of the most time consuming functions in the algorithm grouped by type for one video file. Results were typical across the data set.

Table 5.3: Profiler Summary for 20160725T124323.avi.

Function Name	Number of Calls	Cumulative Time (s)	Total Time (s)
TensorFlow Functions:			
TFE_Py_execute	792	15.076	15.076
TFE_Py_FastPathExecute	2560	1.116	1.116
NumPy Functions:			
implement_array_function	7085963/ 2741625	18.85	5.416
append	1932354	10.892	1.73
ravel	2082699	3.821	1.155
array	4824238	2.408	2.404
reduce	465898	1.014	1.011
OpenCV Functions:			
calcOpticalFlowPyrLK	1592	4.561	4.561
Write cv2.VideoCapture object	809	4.534	4.534
Bilateral Filter	810	2.72	2.72
norm	6472	1.787	1.787
detectAndCompute	35	1.69	1.69
Read cv2.VideoCapture object	811	1.654	1.654
Other:			
update_patch_flow	783	25.359	2.428

The second column shows the total number of times the function is called. The third column shows the total time for that function along with all of the functions called within that function. The last column shows the total time used within that function only.

Overall, the NumPy functions use the most amount of runtime with millions of calls to functions that manipulate the feature point lists. The "update_patch_flow" function shown on the last row is where many of these operations occur.

The TensorFlow inference operations also take a significant amount of the total run time as inference is performed one point at a time.

The last category of time-consuming functions relate to OpenCV. Two of these functions are used to read from the input video file and write to a new file and may be unnecessary with a live camera feed. The optical flow tracker, feature detection, denoising filter, and norm operation (used for calculating the pulse tracker optical flow) are the most intensive image processing operations.

5.4 Limitations / Future Improvements

Several improvements could be added in future research to improve the tracking algorithm, neural network prediction accuracy, and execution speed.

5.4.1 Tracking Algorithm

Further work could be done to develop additional constraints for the tracking algorithm. Additional criteria could be added to help distinguish non-rigid motion from the motion of occluding objects to trigger a switch from tracking to prediction or detect corner cases where the tracking algorithm does not fail but has an undesired shape (grid lines overlapping, etc.).

The smoothness constraint threshold could be improved by addition of an extreme event detection algorithm using Extreme Value Theory (EVT) which could continuously monitor and update the threshold range dynamically as tracking progresses [59, 60]. The current method assumes that the motion during the second diastole is representative of the rest of the video and may require periodic threshold resets as motion changes over longer periods.

A tracking re-initialization that occurs within the 10-step sequence history just prior to a prediction could cause an abrupt change within the sequence that may introduce additional errors. An adjustment to smooth large re-initialization changes may reduce errors and improve the predictions.

The tracked feature points can sometimes exhibit unstable jumps to other image areas when details are suddenly lost (e.g. glare) or when objects enter the ROI. A more robust method of handling outlier motion could help to screen out these points and improve the tracking performance.

The switch to triggering re-initialization during cardiac diastole peaks required some knowledge of the period and type of tissue motion. Future implementations for other organs may require changes specific to that tissue motion.

5.4.2 Data Collection

As can be seen in Figure 3.7, the extracted data point features tended to cluster over the length of the tracking sequence. Performing a periodic re-initialization of the features to reduce this clustering could result in better training data. However, because these sequences were sliced into relatively small sequences, any long term clustering effects would only minimally affect the final sequences. There could be a bias of more sequences based on the clustered areas, but this might not necessarily be detrimental since the clusters are likely to form in more well-defined areas of the image.

A large number of points (10,000 maximum) were used in the feature detection step in order to obtain more points along the coronary artery. This area did not show many detected features without a high maximum point threshold setting. Further changes to the feature detection routine to improve detection along the artery, divide the initial detection into zones with different detection settings, or eliminate excess feature points in certain regions could reduce the size of the data set. The total number of point sequences obtained from the combined data set after slicing the 30 second time series numbered approximately 28 million sequences. Out of memory errors were encountered during training with this data set so only 40% of the collected sequences (after shuffling) were used.

Valid predictions would be limited to the central area of the image due to the circular feature extraction area chosen. The feature extraction method should be tailored to each specific tissue and data set as necessary.

Finally, all videos in this data set had a similar field of view and similar, regular heart rates. An increase in the variation of angles, heart rates, and beat patterns would allow better prediction in other scenarios. On another data set with different heartrates, prediction accuracy might be reduced due to the difference in distance moved (tissue speed) during each frame. More data could be collected at several heart rates or, assuming the cyclic motion is similar across all heart rates, a scaling algorithm could be developed to scale the predicted output based on the detected heart rate. Collection of data with no foreign objects in the frame would also improve the quality of the data and require less post-processing.

5.4.3 Neural Network

Adopting a different neural network structure may yield improved results by better capturing the spatial relationships between grid points. As described in Section 2.2.3, a CNN network can learn these spatial relationships, but it typically performs operations on entire images or fixed grids unlike the moving, user-selected ROI grids created in this research. However, a CNN-LSTM combination trained on the entire set of image frames could predict regional optical flows and adjust grid point locations [61, 62]. Other recent research into using Convolutional LSTMs or LSTMs with attention mechanisms for analysis of non-uniform sensor data may provide a framework for using the variable grids generated in this research with a CNN [63, 64]. Further exploration of the addition of dropout layers may yield better, more generalized prediction results.

In this research, a simple 80% (training) / 20% (testing) data split was performed after shuffling the entire data set. Thus, the network model accuracy calculated during training is limited to this single data split configuration. A more robust approach to calculating accuracy is to use k-fold cross-validation in which several data splits (folds) are generated and used to train the model multiple times. This helps to ensure that no single portion of data is over-emphasized in the results. The average accuracy of multiple training runs on a single model is also better suited for use in comparisons to different network models with different hyper-parameters [20]. In this instance, 20% of the data is 1/5 of the data set, hence, a 5-fold cross-validation would train the model 5 times with different 20% test data split.

5.4.4 Speed of Execution

Achieving real-time performance for use during a surgery is an eventual goal for this type of tissue tracking system. While real-time performance was not achieved in this research due to several factors (See Section 5.3), several improvements can be added that would significantly increase performance. Simply adding additional computing power (faster CPUs, GPUs) may provide improved performance (and additional hardware cost), but optimization of the programming routines could provide a similar or greater performance boost while allowing the algorithm to run on less powerful hardware.

Various programming improvements could increase the speed of calculations. Loops should be re-examined and optimized to take advantage of NumPy vector operations when possible as vectorization can speed up array operations by orders of magnitude [65]. Additional performance may be possible on large data sets by using CyPy, a CUDA accelerated version of NumPy [66, 67]. Conversion of double-precision values to single-precision may improve performance if the reduction in precision can be tolerated [31]. Image processing functions should generally perform better on GPU hardware, but care must be taken to balance the performance decrease due to the transfer of data to and from the GPU with the performance improvement of a specific routine [31]. However, various OpenCV functions may or may not necessarily always benefit from being executed on a GPU [68]. Parallel threads to perform the grid point prediction may greatly improve the speed of inference and would be necessary for a real-time grid prediction.

One bottleneck identified in Section 5.3 is in the file reading and writing routines. This is consistent with the experience of other software developers and is a known issue when reading image frames from video files [69]. Similar issues may be encountered when quickly reading frames from a video camera [70]. Frame buffering and multi-threaded operations are recommended to improve the input/output performance in both cases [69, 70].

A final way to dramatically increase speed is to simply reduce the frame rate to a lower value. For example, a frame rate reduction of 30 fps to 20 fps should reduce the overall amount of computation by about 1/3. The frame rate would need to high enough that each motion step does not increase to a level that causes tracking to fail and the neural network would need to be retrained on data adjusted to that frame rate.

CHAPTER 6

CONCLUSIONS

A method was successfully developed to predict a non-rigid tracking grid location from a neural network model that has learned similar types of motion. This method was applied to a cardiac video data set and integrated with an existing non-rigid tissue tracking algorithm.

Several improvements were made to increase the performance of the baseline algorithm. A complete rewrite from Matlab into Python code was finished to simplify the code and increase readability. This rewrite also allowed the use of industry-standard, open-source software libraries.

A pulse tracking routine based on optical flow was added to aid in feature detection and re-initialization of tracking. This gives the additional benefit of adding an estimate of heart rate to the algorithm. If developed further, this optical flow measurement could be used as an additional diagnostic parameter during surgery to monitor overall tissue strain as described in Chapter 1.

The final speed of execution was about one-half of that required for a real-time system. The performance bottlenecks identified in Section 5.4.4 have great potential for optimization suggesting that real-time performance is achievable on the current generation of high-end desktop/laptop computers.

Other future improvements identified in Section 5.4 include developing additional tracking constraints, improving occlusion detection, improving the quality of the data collection, and experimenting with more neural network configurations.

Finally, this technique need not be limited to cardiac images. It could be applied to other types of non-rigid motion tracking outside of biological tissues such as ocean surface tracking for weather and energy research or fabric motion tracking for augmented reality applications.

REFERENCES

[1] Intuitive Surgical Inc. Intuitive surgical, inc. - annual report 2019, 2019. `https://investor.intuitivesurgical.com/static-files/31b5c428-1d95-4c01-9c85-a7293bac5e05`, Accessed 05/21/2020.

[2] Koichiro Yamakado. Image-guided ablation of adrenal lesions. *Seminars in Interventional Radiology*, 31(2):149–156, Jun 2014.

[3] Nebojša Mujović, Milan Marinković, Radoslaw Lenarczyk, Roland Tilz, and Tatjana Potpara. Catheter ablation of atrial fibrillation: An overview for clinicians. *Advances in Therapy*, 34(8):1897–1917, Aug 2017.

[4] Marlène C. H. Hekman, Mark Rijpkema, Johan F. Langenhuijsen, Otto C. Boerman, Egbert Oosterwijk, and Peter F. A. Mulders. Intraoperative imaging techniques to support complete tumor resection in partial nephrectomy. *European Urology Focus*, 4(6):960–968, 2017.

[5] Michele Solis. New frontiers in robotic surgery: The latest high-tech surgical tools allow for superhuman sensing and more. *IEEE Pulse*, 7(6):51–55, Nov 2016.

[6] O. Lorton, P. C. Guillemin, N. Mori, L. A. Crowe, S. Boudabbous, S. Terraz, C. D. Becker, P. Cattin, R. Salomir, and L. Gui. Self-scanned hifu ablation of moving tissue using real-time hybrid us-mr imaging. *IEEE transactions on bio-medical engineering*, 66(8):2182–2191, August 01 2019.

[7] T. Bader, A. Wiedemann, K. Roberts, and U. D. Hanebeck. Model-based motion estimation of elastic surfaces for minimally invasive cardiac surgery. pages 2261–2266, 2007.

[8] A. Soltani, J. Lahti, K. Jarvela, S. Curtze, J. Laurikka, M. Hokka, and V. T. Kuokkala. An optical method for the in-vivo characterization of the biomechanical response of the right ventricle. *Scientific reports*, 8(1):6831–z, May 01 2018.

[9] Archie Hughes-Hallett, Erik K. Mayer, Hani J. Marcus, Thomas P. Cundy, Philip J. Pratt, Ara W. Darzi, and Justin A. Vale. Augmented reality partial nephrectomy: Examining the current status and future perspectives. *Urology*, 83(2):266–273, 2014.

[10] T. R. McCarty and T. Rustagi. New indications for endoscopic radiofrequency ablation. *Clinical gastroenterology and hepatology : the official clinical practice journal of the American Gastroenterological Association*, 16(7):1007–1017, July 01 2018.

[11] R. Richa, A. P. Bo, and P. Poignet. Towards robust 3d visual tracking for motion compensation in beating heart surgery. *Medical image analysis*, 15(3):302–315, June 01 2011.

[12] G. R. Hale, F. Pesapane, S. Xu, I. Bakhutashvili, N. Glossop, B. Turkbey, P. A. Pinto, and B. J. Wood. Tracked foley catheter for motion compensation during fusion image-guided prostate procedures: a phantom study. *European radiology experimental*, 4(1):24–4, April 16 2020.

[13] A. Soltani, J. Lahti, K. Jarvela, J. Laurikka, V. T. Kuokkala, and M. Hokka. Characterization of the anisotropic deformation of the right ventricle during open heart surgery. *Computer methods in biomechanics and biomedical engineering*, 23(3):103–113, February 01 2020.

[14] Bryent Tucker. Development of a heart motion tracking system using non-invasive imaging data. Master's thesis, 2017. Copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works.; Last updated - 2017-11-07.

[15] Jianbo Shi and Tomasi. Good features to track. In *1994 Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, pages 593–600, 1994.

[16] Basic concepts of the homography explained with code. `https://docs.opencv.org/master/d9/dab/tutorial_homography.html`, Accessed 05/21/2020.

[17] A. Sharaff and M. Choudhary. Comparative analysis of various stock prediction techniques. In *2018 2nd International Conference on Trends in Electronics and Informatics (ICOEI)*, pages 735–738, 2018.

[18] N. Muralidhar, S. Muthiah, K. Nakayama, R. Sharma, and N. Ramakrishnan. Multivariate long-term state forecasting in cyber-physical systems: A sequence to sequence approach. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 543–552, 2019.

[19] Peter J. Brockwell and Richard A. Davis. *Introduction to time series and forecasting*. Springer, New York, 2002.

[20] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[21] J. Brownlee. *Deep Learning With Python: Develop Deep Learning Models on Theano and TensorFlow Using Keras*. Machine Learning Mastery, 2017. v1.8.

[22] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, abs/1406.1078, 2014.

[23] R. Yamashita, M. Nishio, R. K. G. Do, and K. Togashi. Convolutional neural networks: an overview and application in radiology. *Insights into imaging*, Jun 22, 2018.

[24] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, page 1097–1105, Red Hook, NY, USA, 2012. Curran Associates Inc.

[25] M. Amaresh and S. Chitrakala. Video captioning using deep learning: An overview of methods, datasets and metrics. In *2019 International Conference on Communication and Signal Processing (ICCSP)*, pages 0656–0661, 2019.

[26] L. Zhang and P. N. Suganthan. Visual tracking with convolutional neural network. In *2015 IEEE International Conference on Systems, Man, and Cybernetics*, pages 2072–2077, 2015.

[27] Ting-Chun Wang, Ming-Yu Liu, Jun-Yan Zhu, Guilin Liu, Andrew Tao, Jan Kautz, and Bryan Catanzaro. Video-to-video synthesis, 2018.

[28] Ting-Chun Wang, Ming-Yu Liu, Jun-Yan Zhu, Andrew Tao, Jan Kautz, and Bryan Catanzaro. High-resolution image synthesis and semantic manipulation with conditional gans. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018.

[29] Choosing mex applications - matlab. `https://www.mathworks.com/help/matlab/matlab_external/choosing-mex-applications.html`, Accessed 05/21/2020.

[30] Bill Chou. Implement deep learning applications for nvidia gpus with gpu coder. `https://www.mathworks.com/videos/implement-deep-learning-applications-for-nvidia-gpus-with-gpu-coder-1512748950189.html`, Accessed 05/21/2020.

[31] Mathworks. Measure and improve gpu performance. `https://www.mathworks.com/help/parallel-computing/measure-and-improve-gpu-performance.html`, Accessed 05/21/2020.

[32] Python. `https://www.python.org/`.

[33] Tom Radcliffe. Python vs. c/c++ in embedded systems, 08/29/16 08/29/16. `https://opensource.com/life/16/8/python-vs-cc-embedded-systems`, Accessed 05/21/2020.

[34] Cython. `https://www.cython.org`.

[35] Numpy. https://numpy.org/.

[36] Scipy. https://www.scipy.org/.

[37] Scikit-learn. https://scikit-learn.org/.

[38] Matplotlib. https://matplotlib.org/.

[39] Opencv: About. https://opencv.org/about/, Accessed 05/21/2020.

[40] Cuda zone | nvidia developer. https://developer.nvidia.com/cuda-zone, Accessed 05/21/2020.

[41] Opencv: Opencl. https://opencv.org/opencl/, Accessed 05/21/2020.

[42] Why tensorflow. https://www.tensorflow.org/about, Accessed 05/21/2020.

[43] Mathworks. Enhance contrast using histogram equalization - matlab histeq. https://www.mathworks.com/help/images/ref/histeq.html, Accessed 05/21/2020.

[44] OpenCV. Histograms - 2: Histogram equalization. https://docs.opencv.org/master/d5/daf/tutorial_py_histogram_equalization.html, Accessed 05/21/2020.

[45] Opencv: Smoothing images. https://docs.opencv.org/master/dc/dd3/tutorial_gausian_median_blur_bilateral_filter.html, Accessed 05/21/2020.

[46] Mathworks. Detect corners using minimum eigenvalue algorithm and return cornerpoints object - matlab detectmineigenfeatures. https://www.mathworks.com/help/vision/ref/detectmineigenfeatures.html, Accessed 05/21/2020.

[47] OpenCV. Shi tomasi corner detector and good features to track. https://docs.opencv.org/master/d4/d8c/tutorial_py_shi_tomasi.html, Accessed 05/21/2020.

[48] Mathworks. Track points in video using kanade-lucas-tomasi (klt) algorithm - matlab. https://www.mathworks.com/help/vision/ref/vision.pointtracker-system-object.html, Accessed 05/21/2020.

[49] Bruce D. Lucas and Takeo Kanade. An iterative image registration technique with an application to stereo vision. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'81, page 674–679, San Francisco, CA, USA, 1981. Morgan Kaufmann Publishers Inc.

[50] OpenCV. Opencv: Object tracking. https://docs.opencv.org/master/dc/d6b/group__video__track.html#ga473e4b886d0bcc6b65831eb88ed93323, Accessed 05/21/2020.

[51] OpenCV. Opencv: Optical flow. `https://docs.opencv.org/master/d4/dee/tutorial_optical_flow.html`, Accessed 05/21/2020.

[52] Jason Brownlee. Time series forecasting with the long short-term memory network in python, April 7, 2017. `https://machinelearningmastery.com/time-series-forecasting-long-short-term-memory-network-python/`, Updated August 5, 2019, Accessed 05/21/2020.

[53] Stacey Ronaghan. Deep learning: Which loss and activation functions should i use?, July 26, 2018. `https://towardsdatascience.com/deep-learning-which-loss-and-activation-functions-should-i-use-ac02f1c56aa8`, Accessed 05/21/2020.

[54] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, January 2014.

[55] Jason Brownlee. How to diagnose overfitting and underfitting of lstm models, September 1, 2017. `https://machinelearningmastery.com/diagnose-overfitting-underfitting-lstm-models/`, Updated on January 8, 2020, Accessed 05/21/2020.

[56] Jason Brownlee. Time series forecasting performance measures with python, February 1, 2017. `https://machinelearningmastery.com/time-series-forecasting-performance-measures-with-python/`, Updated on August 21, 2019, Accessed 05/21/2020.

[57] John Pomerat, Aviv Segev, and Rituparna Datta. On neural network activation functions and optimizers in relation to polynomial regression. pages 6183–6185, 12 2019.

[58] Rochak Agrawal. Optimization algorithms for deep learning, July 23, 2019. `https://medium.com/analytics-vidhya/optimization-algorithms-for-deep-learning-1f1a2bd4c46b`, Accessed 05/21/2020.

[59] Daizong Ding, Mi Zhang, Xudong Pan, Min Yang, and Xiangnan He. Modeling extreme events in time series prediction. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, page 1114–1122, 2019.

[60] Alban Siffer, Pierre-Alain Fouque, Alexandre Termier, and Christine Largouet. Anomaly detection in streams with extreme value theory. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, page 1067–1075, 2017.

[61] Pengpeng Liu, Michael Lyu, Irwin King, and Jia Xu. Selflow: Self-supervised learning of optical flow, 2019.

[62] A. M. Fiorito, A. Østvik, E. Smistad, S. Leclerc, O. Bernard, and L. Lovstakken. Detection of cardiac events in echocardiography using 3d convolutional recurrent neural networks. In *2018 IEEE International Ultrasonics Symposium (IUS)*, pages 1–4, 2018.

[63] Chaoyun Zhang, Marco Fiore, Iain Murray, and Paul Patras. Cloudlstm: A recurrent neural model for spatiotemporal point-cloud stream forecasting, 2019.

[64] Yuxuan Liang, Songyu Ke, Junbo Zhang, Xiuwen Yi, and Yu Zheng. Geoman: Multi-level attention networks for geo-sensory time series prediction. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 3428–3434. International Joint Conferences on Artificial Intelligence Organization, 7 2018.

[65] George Seif. One simple trick for speeding up your python code with numpy, Jun 5, 2019. `https://towardsdatascience.com/one-simple-trick-for-speeding-up-your-python-code-with-numpy-1afc846db418`, Accessed 05/21/2020.

[66] Ryosuke Okuta, Yuya Unno, Daisuke Nishino, Shohei Hido, and Crissman Loomis. Cupy: A numpy-compatible library for nvidia gpu calculations. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*, 2017.

[67] George Seif. Here's how to use cupy to make numpy over 10x faster, August 22, 2019. `https://towardsdatascience.com/heres-how-to-use-cupy-to-make-numpy-700x-faster-4b920dda1f56`, Accessed 05/21/2020.

[68] James Bowley. Opencv 3.4 gpu cuda performance comparison (nvidia vs intel), February 28, 2018. `https://jamesbowley.co.uk/opencv-3-4-gpu-cuda-performance-comparison-nvidia-vs-intel/`, Accessed 05/21/2020.

[69] Adrian Rosebrock. Faster video file fps with cv2.videocapture and opencv, February 6, 2017. `https://www.pyimagesearch.com/2017/02/06/faster-video-file-fps-with-cv2-videocapture-and-opencv/`, Accessed 05/21/2020.

[70] Najam R Syed. Multithreading with opencv-python to improve video processing performance, July 5, 2018. `https://nrsyed.com/2018/07/05/multithreading-with-opencv-python-to-improve-video-processing-performance`, Accessed 05/21/2020.

APPENDICES

APPENDIX A

SOURCE CODE

A.1 Main Algorithm

The following code is a re-factored version of the original MATLAB algorithm [14] into Python with the various improvements described in Chapter 3.

A.1.1 Overview

Algorithm 1: Main Tracking.

```
Result: Track a region-of-interest through an image sequence
begin
    pre-load neural network model and scaler
    open data set input video
    while new frame available do
        pre-process frame (equalize histogram, de-noise)
        if first frame then
            begin pulse tracker
        else if first diastole detected then
            prompt for ROI selection determine number of patches
        else if second diastole detected then
            set neighbor flow difference thresholds
        else if normal tracking is started (third diastole detected) then
            update optical flow
            TODO add criteria to identify occlusion and switch to prediction
            if neural net prediction frames active then
                run neural net grid point prediction
            else
                check constraints
                re-initalize tracking on each diastole or constraint failure
            end
            record grid point history sequences
            write and display current frame with overlays
        end
    end
    close data set input video
    close tracking output video
end
```

A.1.2 Code

```python
# ==================================================================================
#
# file:    main.py
# version: v1.0.0
# author:  Billy Hempstead
#
# summary: The purpose of this program is to loop through an entire data set
#          to track the boundary edges of a selected ROI, fix failed estimate,
#          and quantify current track estimate.
#
#          Piecewise tracking algorithm loop for data set with ML tracking
#          improvements
#
#          Based on the code from the thesis work:
#          "Development of a Heart Motion Tracking System using Non-invasive
#          Imaging Data" by Bryent Tucker, July, 2017
#
# ==================================================================================

# Initialization ===================================================================
import cv2
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
import os, time, math
import functions as f
import classes as c
import cProfile, io, pstats
from scipy import stats as spstats
from sklearn.externals import joblib
from tkinter import filedialog
from tkinter import *

# Global Variables =================================================================

# algorithm selection
# the 'Canny' and 'approxcanny' methods are not supported on a GPU.
edge_method = "Canny"
edge_threshold = (0.025, 0.3)

```

```
# plot options
edges_on = True
patches_on = True
points_on = False
ROI_on = True

# tracking reset parameters
tracking_enabled = True
n_patches_max = 36
n_patch_rows = 3
n_patches_found = False
ML_enabled = True
pulse_detection_enabled = False
min_corners_in_patch = 8
sufficient_corners = False
last_peak = 0

# pulse tracking parameters
pulse_max_corners = 10
pulse_quality_level = 0.5
pulse_min_distance = 25
pulse_block_size = 3
pulse_use_harris = 0
pulse_k = 0.04

# optical flow settings
OF_win_size = (15, 15)
OF_max_level = 2

# colors
colors = c.Colors()

# drawing options
show_contours = False

# Main Loop ======================================================================

# start profiler
pr = cProfile.Profile()
pr.enable()

# get filename
```

```
root = Tk()
root.fullpath = filedialog.askopenfilename(initialdir="./data",
                      filetypes =(("AVI FIle", "*.avi"),("All Files","*.*")),
                      title = "Choose a file.")
fullpath = root.fullpath
root.destroy()

pathname = os.path.dirname(fullpath)
input_file = os.path.basename(fullpath)
basename = os.path.splitext(input_file)[0]
output_path = "./output/"
output_file = output_path + basename + "_output.avi"
text_output_file = output_path + basename + "_summary.txt"
ML_model_path = "./models/sequence_model.h5"
scaler_model_path = "./models/scaler_model.pkl"

# initialize local variables
corners = []
corners_pulse = []
peaks = []
flow_avg = []
contours = []
hierarchy = []
diastole_frames = []
new_peak = False
main_tracking_start = False
main_tracking_start_frame = -1
peak_count = 0
screenshot_count = 0
t_read = 0

# filtering parameters
lowcut = 30/60 # Hz
highcut = 70/60 # Hz

# initialize constraints object
constraints = c.Constraints()

# set length of sequence for prediction
sequence_history_length = 10

# pre-load neural network
```

```
if ML_enabled == True:
    model = tf.keras.models.load_model(ML_model_path)
    scaler = joblib.load(scaler_model_path)

# load input / initialize output video
cap, output_vid, frame_width, frame_height, n_last_frame, FPS = \
                                        f.import_video(fullpath, output_file)

# create region of interest rectangle and mask
ROI = c.Rect(100, 100, 200, 200) # (x, y, w, h)

# define area of occlusion and frame range
occluded_area = c.Rect(150, 150, 500, 450)
occluded_frames = (201, 230)

# define frame range for analysis
frame_range = ( 0, n_last_frame )
n_frames = frame_range[1] - frame_range[0]
cap.set(cv2.CAP_PROP_POS_FRAMES, frame_range[0])

# loop through frames
while(cap.isOpened()):

    # get current frame
    frame_count = int(cap.get(cv2.CAP_PROP_POS_FRAMES))

    # update progress
    # os.system('clear') # slows down execution a lot
    print('Processing frame: ', int(frame_count), ' of ', int(n_last_frame))

    # exit loop early if desired
    if frame_count > frame_range[1]:
        break

    # get a new frame from video
    t1_read = time.time()
    ret, frame = cap.read()
    t2_read = time.time()
    t_read = t_read+(t2_read-t1_read)

    if ret == True:

```

```python
        # equalize and de-noise frame
        frame_gray = f.preprocess_frame(frame, 'adaptive', 'bilateral')
        frame_gray_original = np.copy(frame_gray)

        if main_tracking_start == False:
            if frame_count == frame_range[0]:

                # initalize corners for pulse tracking on first frame
                corners_pulse = cv2.goodFeaturesToTrack(frame_gray,
                              maxCorners = pulse_max_corners,
                              qualityLevel = pulse_quality_level,
                              minDistance = pulse_min_distance, mask = None,
                              blockSize = pulse_block_size,
                              useHarrisDetector = pulse_use_harris, k = pulse_k )

            else:

                # update pulse detection tracker
                hr, flow_avg, flow_avg_scaled, peaks, new_peak, corners_pulse =
                    f.detect_pulse(frame_gray, frame_gray_old, corners_pulse,
                                  peaks, flow_avg, lowcut, highcut, FPS,
                                      frame_count, n_frames)

                # determine number of patches based on first diastole
                if len(peaks) == 1:

                    # save reference frame and SURF features for reinitialization
                    if 'frame_ref' not in locals():
                        frame_ref = np.copy(frame_gray)
                        detector = cv2.xfeatures2d.SURF_create( hessianThreshold =
                            1000, nOctaves = 4,
                                                           nOctaveLayers = 3,
                                                               extended = True,
                                                           upright = True)
                        pts_ref, dsc_ref = detector.detectAndCompute(frame_ref, None)

                        t1_roi = time.time()
                        # select tracking ROI
                        ROI = f.ROI_selection(frame_gray, ROI)
                        t2_roi = time.time()
                        t_roi = t2_roi - t1_roi

```

```
    if n_patches_found == False:
        if 'frame_diastole_1' not in locals():
            # detect corner points for use in tracker on first peak
            frame_diastole_1 = frame_gray_old
            diastole_frames.append(frame_count)

        patches, n_patch_rows, n_patches, n_patches_found =
            f.determine_n_patches(frame_diastole_1, frame_ref, ROI,
            n_patch_rows, n_patches_max )

        if n_patches_found == True:
            last_peak = len(peaks)
            x_grid_ref, y_grid_ref = f.create_grid(patches)
            x_grid_ML = np.zeros_like(x_grid_ref)
            y_grid_ML = np.zeros_like(y_grid_ref)
            x_grid_hist = [ [] for _ in range(x_grid_ref.size)]
            y_grid_hist = [ [] for _ in range(y_grid_ref.size)]
            x_grid_ML_hist = [ [] for _ in range(x_grid_ref.size)]
            y_grid_ML_hist = [ [] for _ in range(y_grid_ref.size)]

# set neighbor flow threshold based on cycle after next diastole
if len(peaks) > last_peak and n_patches_found == True:

    if 'frame_diastole_2' not in locals():
        # reinitialize on peak
        diastole_frames.append(frame_count)
        frame_diastole_2 = frame_gray_old
        patches, corners, _, _, _ =
            f.reinitialize_tracking(frame_ref, frame_gray,
            frame_gray_old, patches, x_grid_ref, y_grid_ref,
            pts_ref, dsc_ref)

    # set neighbor flow thresholds
    patches = f.set_flow_thresholds(frame_gray, frame_gray_old,
        corners, patches)

# start main tracker on next diastole
if len(peaks) > last_peak + 1:

    main_tracking_start = True
    main_tracking_start_frame = frame_count+1
    diastole_frames.append(frame_count)
```

```

                    # initialize timers
                    t0_sec = time.time()
                    t0 = time.ctime()
                    print(t0)
                    t_write = 0
                    t_read = 0
                    frame_start_tracking = frame_count

                    patches, corners, _, _, _ = f.reinitialize_tracking(frame_ref,
                        frame_gray, frame_gray_old, patches, x_grid_ref, y_grid_ref,
                        pts_ref, dsc_ref)

                    # initialize grid sequences
                    grid_sequences = [ [] for _ in range(x_grid_ref.size)]

                frame_final_RGB = f.write_frame(frame_gray_original, ROI, [], [],
                                                [], [], output_vid, [], [],
                                                [], [], frame_count, frame_range,
                                                occluded_frames, n_frames,
                                                    n_last_frame,
                                                peaks, flow_avg_scaled, hr,
                                                    constraints, basename)

        elif main_tracking_start == True:

            last_peak = len(peaks)

            # draw occluded rectangle on prediction frames
            if occluded_frames[0] <= frame_count <= occluded_frames[1] and
                "occluded_frames" in locals():
                frame_gray_original = f.rectangle_occlusion(frame_gray_original,
                    occluded_area, alpha = 0.5 )

            # analyze frame
            patches, corners, grid_sequences, constraints, x_grid, y_grid,
                x_grid_ML, y_grid_ML = \
                    f.analyze_frame(frame_gray, frame_gray_old, frame_ref, ROI,
                        patches,
                                corners, grid_sequences, constraints,
                                    frame_count, cap,
                            ML_enabled, model, scaler,
                               sequence_history_length, occluded_frames,
                            x_grid_ref, y_grid_ref, x_grid_ML, y_grid_ML,
                               pts_ref, dsc_ref)

            # detect edges
            if show_contours == True:
                contours, hierarchy = f.find_contours(frame_gray, x_grid, y_grid)

            # update pulse detection tracker
            hr, flow_avg, flow_avg_scaled, peaks, new_peak, corners_pulse =
                f.detect_pulse(frame_gray, frame_gray_old, corners_pulse,
                        peaks, flow_avg, lowcut, highcut, FPS, frame_count,
                           n_frames)

            # reset corners on diastole
            if len(peaks) > last_peak:
                patches, corners, corners_new, x_grid, y_grid =
                    f.reinitialize_tracking(frame_ref,
                        frame_gray, frame_gray_old, patches, x_grid_ref,
                           y_grid_ref, pts_ref, dsc_ref)
                diastole_frames.append(frame_count)

            # save corner sequences
            if occluded_frames[0] <= frame_count <= occluded_frames[1]:
                grid_sequences = f.append_grid_sequences(grid_sequences, x_grid_ML,
                    y_grid_ML)
                x_grid_hist, y_grid_hist, x_grid_hist_ML, y_grid_hist_ML =
                    f.append_grid_prediction_history(x_grid, y_grid, x_grid_ML,
                                  y_grid_ML, x_grid_hist, y_grid_hist,
                                     x_grid_ML_hist, y_grid_ML_hist)
            else:
                grid_sequences = f.append_grid_sequences(grid_sequences, x_grid,
                    y_grid)

            # write frame to file
            t1_write = time.time()

            frame_final_RGB = f.write_frame(frame_gray_original, ROI, patches,
                corners,
                                  contours, hierarchy, output_vid,
                                  x_grid, y_grid, x_grid_ML, y_grid_ML,
```

```python
                                        frame_count, frame_range, occluded_frames, n_frames,
                                        n_last_frame, peaks, flow_avg_scaled,
                                        hr, constraints, basename)

            t2_write = time.time()
            t_write = t_write + (t2_write - t1_write)

        # show current frame
        if frame_count == frame_range[0]:
            cv2.imshow('Current Frame', frame_gray)
        else:
            cv2.imshow('Current Frame', frame_final_RGB)
            if cv2.waitKey(1) & 0xFF == ord('q'):
                break

        # save frame at desired frames
        if frame_count in [main_tracking_start_frame, occluded_frames[0], occluded_frames[1], int((occluded_frames[1]+occluded_frames[0])/2)]:
            screenshot_count +=1
            screenshot_filename = output_path + 'plots/' + 'screenshot_' + str(screenshot_count) + '_' + basename + '.png'
            cv2.imwrite(screenshot_filename, frame_final_RGB)

        # save current frame for next cycle
        frame_gray_old = frame_gray.copy()

    # break the loop
    else:
        break

# Post-processing =================================================================

# close video files
cap.release()
cv2.destroyAllWindows()
output_vid.release()

# update timer
t1_sec = time.time()
t1 = time.ctime()
```

```python
total_secs = t1_sec - t0_sec - t_roi - t_read - t_write # subtract out frame read
    / write time
actual_FPS = (n_frames-frame_start_tracking)/total_secs

# print info
os.system('clear')

n_tracked_frames = int(n_frames-main_tracking_start_frame+1)
n_diastole_frames = len(diastole_frames)
n_failed_frames = len(constraints.failed_frames)
n_reinit_frames = len(constraints.reinitialized_frames)
n_predicted_frames = int(occluded_frames[1]-occluded_frames[0]+1)

fout = open(text_output_file,'w')
print("Filename:", input_file, file=fout)
print("\n", file=fout)
print("ROI: Upper Left:", ROI.start_pt, "Lower Right:", ROI.end_pt)
print("End Time:", t1, file=fout)
print("Total Tracking Time:", round(total_secs,1), " secs", file=fout)
print("Tracking Framerate:", round(actual_FPS, 1), " FPS", file=fout)
print("Percent of Full Speed:", round(actual_FPS/FPS*100, 1), "%", file=fout)
print("\n", file=fout)
print("Total Frames:", n_frames, file=fout)
print("Tracked Frames:", n_tracked_frames, '(',
    round(n_tracked_frames/n_frames*100, 1), '% )', file=fout)
print("Re-Initalized Frames (Diastole):", n_diastole_frames, '(',
    round(n_diastole_frames/n_frames*100, 1), '% )', file=fout)
print("Re-Initalized Frames (Failures):", n_reinit_frames, '(',
    round(n_reinit_frames/n_tracked_frames*100, 1), '% )', file=fout)
print("Failed Frames (Diastole):", n_failed_frames, '(',
    round(n_failed_frames/n_tracked_frames*100, 1), '% )', file=fout)
print("Predicted Frames:", n_predicted_frames, '(',
    round(n_predicted_frames/n_tracked_frames*100, 1), '% )', file=fout)
print("ROI: Upper Left:", ROI.start_pt, "Lower Right:", ROI.end_pt)
fout.close()

print("Filename:", input_file)
print("\n")
print("ROI: Upper Left:", ROI.start_pt, "Lower Right:", ROI.end_pt)
print("End Time:", t1)
print("Total Tracking Time:", round(total_secs,1), " secs")
print("Tracking Framerate:", round(actual_FPS, 1), " FPS")
print("Percent of Full Speed:", round(actual_FPS/FPS*100, 1), "%")
print("\n")
print("Total Frames:", n_frames)
print("Tracked Frames:", n_tracked_frames, '(',
    round(n_tracked_frames/n_frames*100, 1), '% )')
print("Re-Initalized Frames (Diastole):", n_diastole_frames, '(',
    round(n_diastole_frames/n_frames*100, 1), '% )')
print("Re-Initalized Frames (Failures):", n_reinit_frames, '(',
    round(n_reinit_frames/n_tracked_frames*100, 1), '% )')
print("Failed Frames (Diastole):", n_failed_frames, '(',
    round(n_failed_frames/n_tracked_frames*100, 1), '% )')
print("Predicted Frames:", n_predicted_frames, '(',
    round(n_predicted_frames/n_tracked_frames*100, 1), '% )')

# create plots
f.create_grid_plots(x_grid_hist, y_grid_hist, x_grid_ML_hist, y_grid_ML_hist,
    basename, output_path, text_output_file)
f.create_timeline_plots(constraints, n_last_frame, diastole_frames,
    main_tracking_start_frame, occluded_frames, basename, output_path)

# end profiler
pr.disable()
s = io.StringIO()
ps = pstats.Stats(pr, stream=s).sort_stats('tottime')
ps.print_stats()

with open(text_output_file, 'a') as fout:
    fout.write(s.getvalue())
```

A.2 Feature Point Sequence Extraction

A.2.1 Overview

Algorithm 2: Feature Point Sequence Extraction.

```
Result: Obtain a sequence of feature points for each motion cycle in a video and save to
        file

begin
    open data set input video
    while new frame available do
        pre-process frame (equalize histogram, denoise)
        if first frame then
            begin pulse tracker
        if first diastole detected then
            start feature tracker
        update optical flow for pulse and feature tracker
        append new feature locations to sequence array
        write and display current frame with overlays
    end
    save sequence array to file
    close data set input video
    close feature extraction output video
end
```

A.2.2 Code

```
# ==================================================================================
#
# file:    feature_extraction.py
# version: v1.0.0
# author:  Billy Hempstead
#
# summary: Opens an .avi file, extracts the feature points, and saves to .csv
#
# ==================================================================================

# Imports ==========================================================================

import cv2
import numpy as np
import matplotlib.pyplot as plt
import time, os, math
import functions as f
import classes as c
from sklearn import preprocessing
from scipy.signal import find_peaks
from tkinter import filedialog
from tkinter import *

# Global Variables =================================================================

# ROI size
ROI_radius = 250

# optical flow settings
OF_win_size = (15, 15)
OF_max_level = 2

# filtering parameters
lowcut = 30/60 # Hz
highcut = 70/60 # Hz

# colors
colors = c.Colors()

# output folder
```

```
output_path = './data_feature_extraction/'

# initialize heart rate parameters
hr = 0
new_peak = False
peaks = []
corners_1 = []
corners_new_1 = []
corners_new_pulse = []
status_1 = []
status_1_lost_index = []
err_1 = []
flow_avg = []

# initialize sequence lists
sequences = []

# tracking parameters
trk_max_corners = 10000
trk_quality_level = 0.001
trk_min_distance = 3
trk_block_size = 3
trk_use_harris = 0
trk_k = 0.004

# pulse tracking parameters
pulse_max_corners = 10
pulse_quality_level = 0.005
pulse_min_distance = 25
pulse_block_size = 3
pulse_use_harris = 0
pulse_k = 0.004

# Main Loop ======================================================================

# get filename
root = Tk()
root.fullpath = filedialog.askopenfilename(initialdir="./data/",
                     filetypes =(("AVI FIle", "*.avi"),("All Files","*.*")),
                     title = "Choose a file.")
fullpath = root.fullpath
root.destroy()
```

```

pathname = os.path.dirname(fullpath)
input_file_name = os.path.basename(fullpath)
basename = os.path.splitext(input_file_name)[0]
output_csv = pathname + '/' + basename + '.csv'
output_npy = output_path + basename + '.npy'
input_file = fullpath
output_file = output_path + basename + '_w_features.avi'

# load input video and initialize output video
cap, output_vid, frame_width, frame_height, n_frames, FPS = \
                                    f.import_video(input_file, output_file)

# set sampling frequency
fs = FPS

# initialize timer
t0_sec = time.time()
t0 = time.ctime()
print(t0)

# create ROI mask
mask = np.zeros((frame_width, frame_height), dtype=np.uint8)
mask = cv2.circle(mask, center = (int(frame_width/2), int(frame_height/2)),
                                radius = ROI_radius, color = colors.white,
                                    thickness = -1)

# loop through frames
while(cap.isOpened()):

    # get current frame
    frame_count = int(cap.get(cv2.CAP_PROP_POS_FRAMES))

    # update progress
    os.system('clear')
    print('Processing frame: ', int(frame_count), ' of ', int(n_frames))

    # get a new frame from video
    ret, frame = cap.read()

    if ret == True:

        # adjust frame
        frame_gray = f.preprocess_frame(frame, 'adaptive', 'bilateral')

        if frame_count == 0:

            # detect pulse corners
            corners_pulse = cv2.goodFeaturesToTrack( frame_gray, pulse_max_corners,
                    pulse_quality_level, pulse_min_distance, mask, pulse_block_size,
                    pulse_use_harris, pulse_k )

            # convert first frame to RGB
            frame_RGB = cv2.cvtColor(frame_gray, cv2.COLOR_GRAY2RGB)

            # save current frame for pulse calculation
            frame_gray_old = frame_gray.copy()

        elif len(peaks) == 1 and new_peak == True:

            # detect feature corners on first diastole
            corners_1 = cv2.goodFeaturesToTrack( frame_gray, trk_max_corners,
                        trk_quality_level, trk_min_distance, mask, trk_block_size,
                        trk_use_harris, trk_k )

            # convert frame to RGB
            frame_RGB = cv2.cvtColor(frame_gray, cv2.COLOR_GRAY2RGB)

            # initialize corner sequences
            corners_1_seq = [ [] for _ in range(corners_1.shape[0])]

            for i, corner in enumerate(corners_1):
                corners_1_seq[i] = np.append(corners_1_seq[i], corner[0])

        elif frame_count > 0:

            # update pulse tracker
            corners_new_pulse, _, _ = cv2.calcOpticalFlowPyrLK(frame_gray_old,
                            frame_gray, corners_pulse, OF_win_size, OF_max_level)

            # update feature tracker
            if len(corners_1) > 0:
                corners_new_1, status_1, err_1 = cv2.calcOpticalFlowPyrLK(
```

```python
                        frame_gray_old, frame_gray, corners_1, OF_win_size, OF_max_level)

                # collect lost feature indices
                if status_1.min() == 0:
                    status_1_lost_index = np.append(status_1_lost_index, np.where(status_1 == 0)[0])
                    status_1_lost_index = np.unique(status_1_lost_index)

                # append new corners to sequence
                for i, corner in enumerate(corners_new_1):
                    corners_1_seq[i] = np.append(corners_1_seq[i], corner)

            # save new corners for next cycle
            corners_1 = corners_new_1
            corners_pulse = corners_new_pulse

        # calculate flow, heart rate, phase
        hr, flow_avg, flow_avg_scaled, peaks, new_peak, corners = f.detect_pulse(
                    frame_gray, frame_gray_old, corners_pulse, peaks, flow_avg,
                    lowcut, highcut, fs, frame_count, n_frames )

        # draw overlays
        frame_RGB = f.draw_overlays_extraction(frame_gray, corners_new_1,
                        status_1, frame_width, frame_height, frame_count,
                        n_frames, peaks, new_peak, flow_avg_scaled, basename, hr, ROI_radius)

        # save current frame for next cycle
        frame_gray_old = frame_gray.copy()

        # write frame to file
        output_vid.write(frame_RGB)

        # show converted frame
        cv2.imshow('Current Frame',frame_RGB)
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break

    # break the loop
    else:
        break
```

```python

# Post-processing ==================================================================

# close video files
cap.release()
cv2.destroyAllWindows()
output_vid.release()

# stop timer
t1_sec = time.time()
t1 = time.ctime()
total_secs = t1_sec-t0_sec

# reshape sequences
sequence_length = int(len(corners_1_seq[0])/2)
for i, sequence in enumerate(corners_1_seq):
    corners_1_seq[i] = sequence.reshape((sequence_length, 2))

# remove lost sequences
for ele in sorted(status_1_lost_index, reverse = True):
    del corners_1_seq[int(ele)]

# save data
np.save(output_npy, corners_1_seq)

# print info
os.system('clear')
print("Number of Point Sequences Extracted:", len(corners_1_seq))
print("End Time:", t1)
print("Total Seconds:", round(total_secs,3))
print('\n')
```

A.3 Data Set Creation

A.3.1 Overview

Algorithm 3: Data Set Creation.

```
Result: Consolidate and process extracted data into training and testing data sets
begin
    open each point sequence file
    append to data set array
    shuffle data set
    create array of shorter sub-sequences
    shuffle data set
    split data set into training and test data sets
    fit a Standard scaler to training data set only
    scale training and test data set values
    separate training and test data sets into features and labels
    save training and test data sets and scaler model
end
```

A.3.2 Code

```python
# ================================================================================
#
# file:    create_dataset.py
# version: v1.0.0
# author:  Billy Hempstead
#
# summary: Compiles extracted data sequences into a single dataset.
#
# ================================================================================

# Initialization ==================================================================

from __future__ import absolute_import, division, print_function, unicode_literals

import numpy as np

from sklearn.model_selection import train_test_split
from sklearn.externals import joblib
from sklearn import preprocessing
from sklearn.utils import shuffle

import os, argparse, platform, math, time, io, glob

# Functions =======================================================================

def load_dataset(data_file_dir):

    files = sorted(glob.glob(data_file_dir + '/*.npy'))
    arrays = []
    data_lengths = []

    for f in files:
        temp_data = np.load(f, allow_pickle=True)
        data_lengths.append(temp_data.shape[1])

    min_data_length = min(data_lengths)

    for f in files:
        temp_data = np.load(f, allow_pickle=True)
        temp_data = temp_data[:, :min_data_length]
        arrays.append(temp_data)

    data = np.concatenate(arrays)

    return data

# ---------------------------------------------------------------------------------

def get_scaler(data, scaler_type):

    os.system('clear')
    print("Fitting Scaler")

    step_size = data[0].shape[0]

    if scaler_type == "minmax":
        scaler = preprocessing.MinMaxScaler()
    elif scaler_type == "standard":
        scaler = preprocessing.StandardScaler()
    elif scaler_type == "robust":
        scaler = preprocessing.RobustScaler()

    data_flat = np.concatenate(data)
    data_flat = np.concatenate(data_flat)

    scaler.fit(data_flat.reshape(-1,1))

    return scaler

# ---------------------------------------------------------------------------------

def scale_dataset(data, scaler, transform_type):

    os.system('clear')
    print("Scaling Data")

    data_flat = np.concatenate(data)
    data_flat = np.concatenate(data_flat)

    if transform_type == 'transform':
        data_flat = scaler.transform(data_flat.reshape(-1,1))

    elif transform_type == 'inverse':
        data_flat = scaler.inverse_transform(data_flat.reshape(-1,1))

    os.system('clear')

    return data_flat

# -----------------------------------------------------------------------------

def shift_dataset(data, step_size):

    # create shifted versions of each sequence
    n_seq = data.shape[0]
    data_shifted = []
    sequence_size = int(step_size*2)

    for i, sequence in enumerate(data):

        os.system('clear')
        print("Creating shifted copies - Processing sequence number", i+1, 'of',
            n_seq)

        for j in range(len(sequence)):
            shifted_seq = sequence[ j:j+step_size ]
            if shifted_seq.size == sequence_size:
                data_shifted.append(shifted_seq)

    os.system('clear')

    return data_shifted

# -----------------------------------------------------------------------------

def split_data(data, validation_split):

    # should be done after shuffling

    data_length = len(data)
    split_point = int(data_length * (1-validation_split))

    train_data = data[:split_point]
    test_data = data[split_point:]
```

```python

    return train_data, test_data

# ---------------------------------------------------------------------------------

def separate_dataset(data, step_size):

    n_seq = int(len(data)/step_size/2)
    data = data.reshape((n_seq, step_size, 2))

    features = data[:,:-1]
    labels = data[:,-1]

    return features, labels

# Main ============================================================================

# display version info
print('Python Version:  ', platform.python_version())
print('Numpy Version:   ', np.__version__)

step_size = 10
data_reduction_ratio = 0.4

data_file_dir = './data_feature_extraction'

# load and compile datafiles
data = load_dataset(data_file_dir)

# create shifted versions of sequences
data_split = int(data_reduction_ratio*len(data))
data = shuffle(data[:data_split])
data = shift_dataset(data, step_size = step_size)

# shuffle all of the sequences
data = shuffle(data, random_state = 0 )

# separate shuffled sequences into training and testing data
train_data, test_data = split_data(data, validation_split = 0.2)

# scale data based on training data only
# train_split = int(train_split_ratio*len(train_data)) # reduce amount of data to
    scaler to avoid out of memory
scaler = get_scaler(train_data, 'standard')
train_data = scale_dataset(train_data, scaler, 'transform')
test_data = scale_dataset(test_data, scaler, 'transform')

# separate features and labels
train_data_x, train_data_y = separate_dataset(train_data, step_size)
test_data_x, test_data_y = separate_dataset(test_data, step_size)

# save files
joblib.dump(scaler, './models/scaler_model.pkl')
joblib.dump([train_data_x, train_data_y, test_data_x, test_data_y],
    './models/dataset.pkl')

# print sample
sample_number = np.random.randint(0, train_data_x.shape[0])
print('Features: ', '\n', train_data_x.shape)
print(train_data_x[sample_number])
print('Labels: ', '\n', train_data_y.shape)
print(train_data_y[sample_number])

print('Features: ', '\n', train_data_x.shape)
# print(scale_dataset(train_data_x[sample_number], scaler, 'inverse'))
print(scaler.inverse_transform(train_data_x[sample_number]))

print('Labels: ', '\n', train_data_y.shape)
# print(scale_dataset(train_data_y[sample_number], scaler, 'inverse'))
print(scaler.inverse_transform(train_data_y[sample_number].reshape(1,-1)))
```

A.4 Neural Network Training

A.4.1 Overview

Algorithm 4: Neural Network Training.

```
Result: Train the neural network on the point sequence data set
begin
    open training and test data sets and scaler model
    create NN model
    for number of epochs do
        train model with training data set
        check performance on test data set
    end
    run a sample prediction to check output
    save trained model
end
```

A.4.2 Code

```
# ==================================================================================
#
# file:    nn_training.py
# version: v1.0.0
# author:  Billy Hempstead
#
# summary: Neural Network training code for extracted heart feature point
#          sequences.
#
# ==================================================================================

# Initialization ===================================================================

from __future__ import absolute_import, division, print_function, unicode_literals

import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split
from sklearn.externals import joblib
from sklearn import preprocessing
from sklearn.utils import shuffle

import os, argparse, platform, math, time, io, glob

# Main =============================================================================

# display version info
print('Python Version:  ', platform.python_version())
print('Tensorflow Version:', tf.__version__)
print('Numpy Version:   ', np.__version__)

earlystop_callback = tf.keras.callbacks.EarlyStopping(monitor='val_loss',
    patience=3)

# import dataset and scaler
train_data_x, train_data_y, test_data_x, test_data_y =
    joblib.load('./models/dataset.pkl')
scaler = joblib.load('./models/scaler_model.pkl')
```

```

# start timer
t0_sec = time.time()

# create model
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.GRU(64, input_shape=(None, 2), return_sequences = True))
# model.add(tf.keras.layers.Dropout(0.2))
model.add(tf.keras.layers.GRU(32))
# model.add(tf.keras.layers.Dropout(0.2))
model.add(tf.keras.layers.Dense(2, activation = 'linear'))

# train model
# opt = tf.keras.optimizers.SGD(lr=0.01, momentum=0.9, decay=0.01)
opt = tf.keras.optimizers.Adam()
model.compile(loss='mean_squared_error', optimizer=opt, metrics=['accuracy'])
model.summary()
history = model.fit(train_data_x, train_data_y, validation_data=(test_data_x,
    test_data_y), batch_size=512, epochs=5)#, callbacks=[earlystop_callback])

# update timer
t1_sec = time.time()
total_min = (t1_sec - t0_sec)/60

# save model
model.save('./models/sequence_model.h5')

# plot loss during training
plt.title('Loss / Mean Squared Error')
plt.plot(history.history['loss'], label='train')
plt.plot(history.history['val_loss'], label='test')
plt.legend()
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.savefig('./models/loss.png')
plt.show()

# plot accuracy during training
plt.title('Accuracy')
plt.plot(history.history['accuracy'], label='train')
plt.plot(history.history['val_accuracy'], label='test')
plt.legend()
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.savefig('./models/accuracy.png')
plt.show()

# check output
sample_number = np.random.randint(0, test_data_x.shape[0])
sample = test_data_x[sample_number].reshape((1,
    test_data_x[sample_number].shape[0], test_data_x[sample_number].shape[1]))
pred = model.predict(sample)
sample = scaler.inverse_transform(sample[-1])
pred = scaler.inverse_transform(pred)

print(sample)
print(pred)

# print info
print()
print("Total Training Time:", round(total_min,1), "min")
```

A.5 Supporting Classes / Functions

The following code provides classes and supporting functions that are called in the source files in previous sections.

A.5.1 Classes

```python
# =================================================================================
#
# file:    classes.py
# version: v1.0.0
# author:  Billy Hempstead
#
# summary: Contains classes for use with the main piecewise tracking algorithm
#
# =================================================================================

import numpy as np
import math

class Patch:

    def __init__(self):

        self.index = 1           # overall patch number

        # corners points of patch (UL, UR, LR, LL)
        self.corners = np.empty([4,2], dtype='int')

         # index of points inside patch
        self.point_idx = np.empty([0], dtype='int')

        self.u_avg = 0                      # average horizontal flow
        self.v_avg = 0                      # average vertical flow
        self.u_avg_neighbors = 0
        self.v_avg_neighbors = 0
        self.u_diff = 0
        self.v_diff = 0
        self.u_diff_min = 0
        self.u_diff_max = 0
        self.v_diff_min = 0
        self.v_diff_max = 0
        self.normal_corner_dist = False     # do points have normal dist.?
        self.continuous_neighbor_flow = False # is neighbor flow continuous?
        self.u_std_dev = 0
        self.v_std_dev = 0
        self.u_outliers = 0
        self.v_outliers = 0

    # method to move the corners of each patch by the average of the optical flow
    def update_corners(self):

        if not math.isnan(self.u_avg):
            for i in range(4):
                self.corners[i][0] = round(float(self.corners[i][0]) + self.u_avg)

        if not math.isnan(self.v_avg):
            for i in range(4):
                self.corners[i][1] = round(float(self.corners[i][1]) + self.v_avg)

        return self.corners

class Rect:

    def __init__(self, x, y, width, height):
        self.x = x
        self.y = y
        self.width = width
        self.height = height
        self.start_pt = (self.x, self.y)
        self.end_pt = (self.x + self.width, self.y + self.height)

class Constraints:

    def __init__(self):
        self.all_normal = False
        self.neighbor_flow_continuous = False
        self.passed = False
        self.failed_frames = []
        self.reinitialized_frames = []
        self.fails = 0
        self.fail_limit = 5

class Colors:

    def __init__(self):
        self.blue      = (255,   0,   0)
        self.red       = (  0,   0, 255)
        self.green     = (  0, 255,   0)
        self.white     = (255, 255, 255)
        self.black     = (  0,   0,   0)
        self.dark_green = ( 0, 125, 0)
        self.magenta   = (225,   0, 255)
        self.cyan      = (255, 255,  0)
        self.yellow    = (  0, 255, 255)
```

A.5.2 Functions

```
# ==================================================================================
#
# file:    functions.py
# version: v1.0.0
# author:  Billy Hempstead
#
# summary: Contains functions for use with the main piecewise tracking algorithm
#          and feature extraction
#
# ==================================================================================

# Initialization ===================================================================

import cv2
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.ticker import FormatStrFormatter
from matplotlib.ticker import PercentFormatter
import time
import os
import math
import classes as c
from scipy import stats as spstats
from scipy import ndimage
from scipy.signal import butter, lfilter
from sklearn import preprocessing
from scipy.signal import find_peaks
from scipy.spatial import distance

# Global Variables =================================================================

# colors
colors = c.Colors()

# SURF detector and matcher
detector = cv2.xfeatures2d.SURF_create(hessianThreshold = 1000, nOctaves = 4,
                                       nOctaveLayers = 3, extended = True,
                                       upright = True)

```

```
matcherBF = cv2.BFMatcher(cv2.NORM_L2, crossCheck=False)

# detection parameters
det_max_corners = 1000
det_quality_level = 0.001
det_min_distance = 3
det_block_size = 3
det_use_harris = 0
det_k = 0.04

# convolution kernel for neighbor flow average
kernel = np.ones((3, 3))
kernel[1, 1] = 0

# equalization / blur settings
CLAHE_clip_limit = 30.0
CLAHE_tile_size = (20, 20)
median_blur_level = 5
BLF_dia = 5
BLF_sigma_color = 60
BLF_sigma_space = 60

# tracker settings
OF_win_size = (15, 15)
OF_max_level = 2

# constraint settings
flow_threshold_margin = 0.1
min_corners_in_patch = 8

# contours settings
contours_max_level = 2
contours_lower_th = 0
contours_upper_th = 255

# grid settings
grid_outline_thickness = 4
grid_color_normal = colors.green
grid_color_ML = colors.yellow

# drawing options
draw_ROI = False
```

```python
draw_grid = True
draw_grid_ML = True
draw_corners = True
draw_contours = False
draw_patch_corners = False
draw_overlays = True
draw_failed_frames = True

# Functions =================================================================

# === Input/Output ==========================================================

def import_video(input_file, output_file):

    # open input file
    cap = cv2.VideoCapture(input_file)

    # check if file opened successfully
    if (cap.isOpened() == False):
        print("Error opening video stream or file")

    # get video properties
    frame_width = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
    frame_height = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))
    n_last_frame = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))
    FPS = cap.get(cv2.CAP_PROP_FPS)

    # create writer object
    fourcc = cv2.VideoWriter_fourcc(*'XVID')
    out = cv2.VideoWriter(output_file,fourcc, FPS, (frame_height, frame_width) )

    return cap, out, frame_width, frame_height, n_last_frame, FPS

# === Image Processing ======================================================

def preprocess_frame(frame, equalization_type, blur_type):

# operations performed on each frame before tracking

    # convert frame to grayscale
    if len(frame.shape) == 3:
        frame = cv2.cvtColor(frame, cv2.COLOR_RGB2GRAY)

    # equalize frame
    if equalization_type == 'normal':
        frame = cv2.equalizeHist(frame)

    elif equalization_type == "adaptive":
        # apply Contrast Limited Adaptive Histogram Equalization (CLAHE)
        clahe = cv2.createCLAHE(clipLimit = CLAHE_clip_limit, tileGridSize =
            CLAHE_tile_size)
        frame = clahe.apply(frame)

    else:
        frame = frame

    # blur to reduce noise to focus on actual features
    if blur_type == 'median':
        frame = cv2.medianBlur(frame, median_blur_level)

    elif blur_type == 'bilateral':
        # better retains edges
        frame = cv2.bilateralFilter(frame, BLF_dia, BLF_sigma_color,
            BLF_sigma_space)

    else:
        frame = frame

    return frame

# === Analysis ===================================================================

def analyze_frame(frame_curr, frame_prev, frame_ref, ROI, patches, corners,
                grid_sequences, constraints, frame_count, cap, ML_enabled, model,
                scaler, sequence_history_length, occluded_frames,
                x_grid_ref, y_grid_ref, x_grid_ML, y_grid_ML, pts_ref, dsc_ref):

# tracking/flow update and patch analysis for current frame

    # update tracker
    corners_new, _, _ = cv2.calcOpticalFlowPyrLK(frame_prev, frame_curr, corners,
        OF_win_size, OF_max_level)

    # calculate flow of features between frames
    patches = update_patch_flow(patches, corners, corners_new)

    # check constraints
    constraints.all_normal, constraints.neighbor_flow_continuous =
        check_constraints(patches)

    if constraints.all_normal or constraints.neighbor_flow_continuous == False:
        constraints.passed = False
        constraints.fails += 1
        constraints.failed_frames.append(frame_count)

    else:
        constraints.passed = True
        constraints.fails = 0

    # trigger reinitialization
    if constraints.fails == constraints.fail_limit:
        constraints.reinitialized_frames.append(frame_count)
        patches, corners, corners_new, _, _ = reinitialize_tracking(frame_ref,
                                        frame_curr, frame_prev, patches,
                                            x_grid_ref, y_grid_ref, pts_ref,
                                            dsc_ref)

    # move patch boundaries by flow
    for patch in patches:
        patch.update_corners()

    # save corners for next cycle
    corners = corners_new

    # get updated grid locations
    x_grid, y_grid = create_grid(patches)

    # ML prediction loop
    if occluded_frames[0] <= frame_count <= occluded_frames[1]+1:

        x_grid_ML = np.zeros(len(grid_sequences))
        y_grid_ML = np.zeros_like(x_grid_ML)
        n_rows = int(np.sqrt(len(x_grid_ML)))

        for i, sequence in enumerate(grid_sequences):
            print('Predicting point ', i, ' of ', len(grid_sequences) )
            output = run_NN(sequence[-(int(sequence_history_length*2)):], model,
               scaler)
            x_grid_ML[i] = output[0][0]
            y_grid_ML[i] = output[0][1]

        x_grid_ML = x_grid_ML.reshape(n_rows, n_rows).astype(int)
        y_grid_ML = y_grid_ML.reshape(n_rows, n_rows).astype(int)

    return patches, corners, grid_sequences, constraints, x_grid, y_grid,
        x_grid_ML, y_grid_ML

# ---------------------------------------------------------------------------------

def ROI_selection(frame, ROI):

# prompts user for selection of ROI

    cv2.destroyAllWindows()
    ROI_sel = cv2.selectROI("Select Region-of-Interest", frame)
    cv2.waitKey(0)
    cv2.destroyAllWindows()

    ROI.x = ROI_sel[0]
    ROI.y = ROI_sel[1]
    ROI.width = ROI_sel[2]
    ROI.height = ROI_sel[3]
    ROI.start_pt = (ROI_sel[0], ROI_sel[1])
    ROI.end_pt = (ROI_sel[0]+ROI_sel[2], ROI_sel[1]+ROI_sel[3])

    return ROI

# ---------------------------------------------------------------------------------

def determine_n_patches(frame_diastole, frame_next, ROI, n_patch_rows,
    n_patches_max ):

# gradually increases the number of patches until all patches meet constraint 1

    print('Determining Number of Patches within ROI')

    n_patch_rows +=1
    patches, corners, n_patches = create_patches(frame_next, ROI, n_patch_rows)

    # calculate optical flow between diastole frame and prev frame
    corners_new, _, _ = cv2.calcOpticalFlowPyrLK(frame_diastole, frame_next,
       corners, OF_win_size, OF_max_level)

    # update patch flow
    patches = update_patch_flow(patches, corners, corners_new)

    constraint_1, _ = check_constraints(patches)

    for patch in patches:
        if len(patch.point_idx) >= min_corners_in_patch:
            sufficient_corners = True
        else:
            sufficient_corners = False
            break

    # constraint 1 and minimum number of points per patch must be met
    # continue on if constraints met or number of patches exceeds max
    # TODO determine how to handle constraints never met
    if constraint_1 == True and sufficient_corners == True or n_patches ==
       n_patches_max:
        n_patches_found = True
    else:
        n_patches_found = False

    return patches, n_patch_rows, n_patches, n_patches_found

# ---------------------------------------------------------------------------------

def set_flow_thresholds(frame_curr, frame_prev, corners, patches):

# adjusts the neighbor difference thresholds based on latest tracking update

    print('Setting Flow Thresholds')

    # calculate optical flow
    corners_new, _, _ = cv2.calcOpticalFlowPyrLK(frame_curr, frame_prev, corners,
       OF_win_size, OF_max_level)

    # update patch flow
    patches = update_patch_flow(patches, corners, corners_new)

    for i, patch in enumerate(patches):
        patch.u_diff = patch.u_avg - patch.u_avg_neighbors
        patch.v_diff = patch.v_avg - patch.v_avg_neighbors

        if patch.u_diff < patch.u_diff_min:
            patch.u_diff_min = patch.u_diff * (1 + flow_threshold_margin)
            #TODO should subtract margin in next version
        elif patch.u_diff > patch.u_diff_max:
            patch.u_diff_max = patch.u_diff * (1 + flow_threshold_margin)

        if patch.v_diff < patch.v_diff_min:
            patch.v_diff_min = patch.v_diff * (1 + flow_threshold_margin)
            #TODO should subtract margin in next version
        elif patch.v_diff > patch.v_diff_max:
            patch.v_diff_max = patch.v_diff * (1 + flow_threshold_margin)

    return patches

# --------------------------------------------------------------------------------

def detect_pulse( frame_curr, frame_prev, corners, peaks, flow_avg, lowcut,
    highcut,
                  fs, frame_count, n_frames ):

# updates pulse detection history

    frame_height = frame_curr.shape[0]
    frame_width = frame_curr.shape[1]

    flow = 0
    hr = 0

   # calculate optical flow
    corners_new, _, _ = cv2.calcOpticalFlowPyrLK(frame_prev, frame_curr, corners,
        OF_win_size, OF_max_level)

    # calculate average point flow
    for i in range(len(corners_new)):
        p1 = corners_new[i]
        p2 = corners[i]
        dist = cv2.norm(p1,p2)
```

```
            flow += dist

    flow_avg = np.append(flow_avg, flow/len(corners_new))

    # filter/scale average flow
    flow_avg_filtered = butter_bandpass_filter(flow_avg, lowcut, highcut, fs,
        order=5)
    flow_avg_scaled = preprocessing.scale(flow_avg_filtered)

    # plot graph
    # normalization to fit to canvas dimension
    flow_avg_scaled = np.asarray(flow_avg_scaled, dtype = np.float32)

    # detect peaks
    num_peaks = len(peaks)
    peaks, _ = find_peaks(-flow_avg_scaled, height = 0.1)

    if len(peaks) > num_peaks:
        new_peak = True
    else:
        new_peak = False

    # calculate heart rate
    if len(peaks) > 1:
        average_frames_per_cycle = sum(np.diff(peaks))/(len(peaks)-1)
        time_per_frame = 1/fs
        time_per_cycle = average_frames_per_cycle * time_per_frame
        hr = int(60 / time_per_cycle) # bpm

    return hr, flow_avg, flow_avg_scaled, peaks, new_peak, corners_new

# ---------------------------------------------------------------------------------

def find_contours(frame, x_grid, y_grid):

# finds the contours (edges) within the ROI

    pts = get_grid_outline(x_grid, y_grid)
    pts = np.asarray( pts, dtype = np.int32).reshape((-1,1,2))

    mask = np.zeros((frame.shape[0], frame.shape[1], 1), np.uint8)
    mask = cv2.polylines(mask, [pts], True, colors.white, thickness = 2)
    mask = cv2.fillPoly(mask, [pts], [255])

    frame = cv2.bitwise_and(frame, frame, mask = mask)

    # # apply blur
    # frame = cv2.medianBlur(frame, median_blur_level)

    # apply thresholding
    retval, frame = cv2.threshold(frame, contours_lower_th, contours_upper_th,
        cv2.THRESH_OTSU)

    # detect contours
    contours, hierarchy = cv2.findContours(frame, cv2.RETR_CCOMP,
        cv2.CHAIN_APPROX_SIMPLE)

    return contours, hierarchy

# === Filtering ==================================================================

def butter_bandpass(lowcut, highcut, fs, order=5):
    nyq = 0.5 * fs
    low = lowcut / nyq
    high = highcut / nyq
    b, a = butter(order, [low, high], btype='band')
    return b, a


def butter_bandpass_filter(data, lowcut, highcut, fs, order=5):
    b, a = butter_bandpass(lowcut, highcut, fs, order=order)
    y = lfilter(b, a, data)
    return y

# === Neural Network ==============================================================

def run_NN(sequence, model, scaler):

# run a new point prediction

    # scale sequence
    x_input = np.asarray(sequence)
    x_input = scaler.transform(x_input.reshape(-1,1))
    x_input = x_input.reshape(1, int(len(sequence)/2), 2)

    # run prediction
    yhat = model.predict(tf.cast(x_input,tf.float32), verbose=0)

    # inverse scale prediction
    yhat = scaler.inverse_transform(yhat)

    return yhat

# === Re-initialization ===========================================================

def reinitialize_tracking(frame_ref, frame_curr, frame_prev, patches,
                        x_grid_ref, y_grid_ref, pts_ref, dsc_ref):

# performs a homography operation to regain tracking

    # detect features
    pts_prev, dsc_prev = detector.detectAndCompute(frame_prev, None)

    # match descriptors
    matches = matcherBF.match(dsc_prev, dsc_ref)
    matches_d_sort = sorted(matches, key = lambda x:x.distance)

    # extract the matched keypoints
    pts_prev_match = np.float32([pts_prev[m.queryIdx].pt for m in
        matches_d_sort]).reshape(-1,1,2)
    pts_ref_match = np.float32([pts_ref[m.trainIdx].pt for m in
        matches_d_sort]).reshape(-1,1,2)

    # find homography matrix using RANSAC
    H, _ = cv2.findHomography(pts_ref_match, pts_prev_match, \
        method = cv2.RANSAC, ransacReprojThreshold = 100.0)

    # transform grid points with homography matrix
    x = x_grid_ref.flatten()
    y = y_grid_ref.flatten()
    grid = np.array([list(zip(x,y))], dtype=np.float32)
    grid_warp = cv2.perspectiveTransform(grid, H)

    # unravel warped combined grid back to x and y grid arrays
    w = x_grid_ref.shape[0]
    h = x_grid_ref.shape[1]
    x_grid_new = grid_warp[0][:,0]
    y_grid_new = grid_warp[0][:,1]
    x_grid_new = np.reshape(x_grid_new.astype(int), (w,h))
    y_grid_new = np.reshape(y_grid_new.astype(int), (w,h))

    # mask to patch border
    pts = get_grid_outline(x_grid_new, y_grid_new)
    pts = np.asarray( pts, dtype = np.int32).reshape((-1,1,2))

    mask = np.zeros((frame_prev.shape[0], frame_prev.shape[1], 1), np.uint8)
    mask = cv2.polylines(mask, [pts], True, colors.white, thickness = 2 )
    mask = cv2.fillPoly(mask, [pts], [255])

    corners = cv2.goodFeaturesToTrack(frame_prev, maxCorners=det_max_corners,
                                   qualityLevel=det_quality_level,
                                   minDistance=det_min_distance,
                                   blockSize = det_block_size,
                                   useHarrisDetector = det_use_harris, mask=mask,
                                   k = det_k)

    corners_new, _, _ = cv2.calcOpticalFlowPyrLK(frame_prev, frame_curr, corners,
        OF_win_size, OF_max_level)

    # reassign corner points after adjusting patch boundaries
    patches = move_patch_corners(patches, x_grid_new, y_grid_new)
    patches = assign_patch_points(patches, corners)
    patches = update_patch_flow(patches, corners, corners_new)

    return patches, corners, corners_new, x_grid_new, y_grid_new

# === Patch-Related ===============================================================

def assign_corners_to_patches(corners, ROI, n_patches):

# calculates initial boundary points of patches based on region of interest
# and number of patches

    patches = []

    # calculate the width and height of each patch
    patch_width = math.floor(ROI.width / np.sqrt(n_patches))
    patch_height = math.floor(ROI.height / np.sqrt(n_patches))

    # assume the number of rows and columns are equal
    n_cols = np.sqrt(n_patches)
    n_rows = n_cols
    row_ct = 0
    col_ct = 0

    # assign point locations
    for i in range(n_patches):
        patches.append( c.Patch() )
        patches[i].index = i + 1
        patches[i].corners[0][0] = ROI.x + patch_width * col_ct
        patches[i].corners[0][1] = ROI.y + patch_height * row_ct
        patches[i].corners[1][0] = patches[i].corners[0][0] + patch_width
        patches[i].corners[1][1] = patches[i].corners[0][1]
        patches[i].corners[2][0] = patches[i].corners[1][0]
        patches[i].corners[2][1] = patches[i].corners[1][1] + patch_height
        patches[i].corners[3][0] = patches[i].corners[0][0]
        patches[i].corners[3][1] = patches[i].corners[0][1] + patch_height

        if col_ct == n_cols-1:
            col_ct = 0
            row_ct += 1
        else:
            col_ct += 1

    return patches

# ------------------------------------------------------------------------------

def assign_patch_points(patches, corners):

    # checks which corners are within the patches

    for idx, point in enumerate(corners):
        for patch in patches:

            if idx == 0:
                patch.point_idx = np.empty([0], dtype='int')

            point_in_patch = cv2.pointPolygonTest(patch.corners, tuple(point[0]),
                                                  False)
```

```python

            # 1 means inside, 0 means on edge, -1 means outside
            if point_in_patch == 1 or 0:
                patch.point_idx = np.append(patch.point_idx, idx)

    return patches

# -----------------------------------------------------------------------------------

def update_patch_flow(patches, corners, corners_new):

    # updates the optical flow based on the new tracked corner values and checks
    # constraints

    # initialize flow vectors
    u_point_all = []
    v_point_all = []

    # generate list of horizontal and vertical optical flow values for each corner
    for i, point in enumerate(corners):
        u_point_all = np.append(u_point_all, corners_new[i][0][0] -
            corners[i][0][0])
        v_point_all = np.append(v_point_all, corners_new[i][0][1] -
            corners[i][0][1])

    # calculate parameters for each patch
    # only update flow if there are sufficient number of samples (10)
    for patch in patches:

        # get points within current patch
        if patch.point_idx.size > 0:

            u_p_value = 0
            v_p_value = 0

            u_point_patch = u_point_all[patch.point_idx]
            v_point_patch = v_point_all[patch.point_idx]

            # calculate patch flow after excluding outliers
            z_u = np.abs(spstats.zscore(u_point_patch))
            z_v = np.abs(spstats.zscore(v_point_patch))

            u_outlier_idx = np.where(z_u < 2)[0]
            v_outlier_idx = np.where(z_v < 2)[0]

            u_point_patch = u_point_patch[u_outlier_idx]
            v_point_patch = v_point_patch[v_outlier_idx]

            if len(u_point_patch) > 0:
                patch.u_avg = np.mean(u_point_patch)
            else:
                patch.u_avg = 0
            if len(v_point_patch) > 0:
                patch.v_avg = np.mean(v_point_patch)
            else:
                patch.v_avg = 0

            # calculate stats if there are enough points for comparison
            # test for normal distribution

            if u_point_patch.size > 8: # skewtest minimum
                _, u_p_value = spstats.skewtest(u_point_patch)

            if v_point_patch.size > 8: # skewtest minimum
                _, v_p_value = spstats.skewtest(v_point_patch)

            alpha = 0.005

            if u_p_value >= alpha and v_p_value >= alpha:
                patch.normal_corner_dist = True
            else:
                patch.normal_corner_dist = False

        # set the patch flow to zero if fewer than specified points in patch
        else:
            patch.u_avg = 0
            patch.v_avg = 0
            patch.u_std_dev = 0
            patch.v_std_dev = 0
            patch.normal_corner_dist = True

    # set patches with zero flow to average neighbor flow after all patch flows
    # have been calculated

    # create matrices for u and v avg
    x_dim = int(np.sqrt(len(patches)))
    u_avg_all = np.zeros(len(patches))
    v_avg_all = np.zeros(len(patches))

    for i, patch in enumerate(patches):
        u_avg_all[i] = patch.u_avg
        v_avg_all[i] = patch.v_avg

    u_avg_all = np.reshape(u_avg_all, (x_dim, x_dim))
    v_avg_all = np.reshape(v_avg_all, (x_dim, x_dim))

    # find average of neighbors by convolution
    u_avg_neighbors = ndimage.generic_filter(u_avg_all, np.nanmean,
        footprint=kernel, mode='constant', cval=np.NaN)
    v_avg_neighbors = ndimage.generic_filter(v_avg_all, np.nanmean,
        footprint=kernel, mode='constant', cval=np.NaN)

    # flatten for iteration
    u_avg_neighbors = np.ndarray.flatten(u_avg_neighbors)
    v_avg_neighbors = np.ndarray.flatten(v_avg_neighbors)

    # set zero flows to neighbor average
    for i, patch in enumerate(patches):
        patch.u_avg_neighbors = u_avg_neighbors[i]
        patch.v_avg_neighbors = v_avg_neighbors[i]

        if patch.u_avg == 0 and patch.v_avg == 0:
            patch.u_avg = patch.u_avg_neighbors
            patch.v_avg = patch.u_avg_neighbors

    # check for continuous flow between neighbors
    for patch in patches:
        patch.u_diff = patch.u_avg - patch.u_avg_neighbors
        patch.v_diff = patch.v_avg - patch.v_avg_neighbors

        if patch.u_diff_min <= patch.u_diff <= patch.u_diff_max and \
           patch.v_diff_min <= patch.v_diff <= patch.v_diff_max:

            patch.continuous_neighbor_flow = True
        else:
            patch.continuous_neighbor_flow = False

    return patches

# ------------------------------------------------------------------------------

def create_patches(frame, ROI, n_patch_rows):

# creates a patch object with corners

    patches = []

    n_patches = int(n_patch_rows**2)
    # rows, cols are always equal; done for convenience in this project but
    # could be changed to allow non-square grid; would require more changes to the
        functions

    # detect initial corners
    mask = np.zeros(frame.shape, dtype=np.uint8)
    mask = cv2.rectangle(mask, ROI.start_pt, ROI.end_pt, (255,255,255), -1)

    corners = cv2.goodFeaturesToTrack(frame, maxCorners = det_max_corners,
                                      qualityLevel = det_quality_level,
                                      minDistance = det_min_distance,
                                      blockSize = det_block_size,
                                      useHarrisDetector = det_use_harris,
                                      mask = mask,
                                      k = det_k)

    # run function to determine initial patch corner points
    patches = assign_corners_to_patches(corners, ROI, n_patches)

    # assign points to patches
    patches = assign_patch_points(patches, corners)

    return patches, corners, n_patches

# ------------------------------------------------------------------------------

def move_patch_corners(patches, x_grid, y_grid):

# updates patch boundaries to match grid

```

```
    n_rows = int(np.sqrt(len(patches)))
    n_cols = n_rows

    row = 0
    col = 0

    for patch in patches:
       patch.corners[0] = [ x_grid[row, col ], y_grid[row, col]  ]
       patch.corners[1] = [ x_grid[row, col+1], y_grid[row, col+1] ]
       patch.corners[2] = [ x_grid[row+1, col+1], y_grid[row+1, col+1] ]
       patch.corners[3] = [ x_grid[row+1, col ], y_grid[row+1, col] ]

       if patch.index % n_rows == 0:
          row +=1
          col = 0
       else:
          col += 1

    return patches

# --------------------------------------------------------------------------------

def check_constraints(patches):

# checks that the conditions for piecewise flow are met

    # initialize constraints
    constraint_1 = True
    constraint_2 = True

    # check constraints
    for patch in patches:

       # 1. all corners within each patch must be normally distributed
       if patch.normal_corner_dist == False:
          constraint_1 = False
          break

       # 2. the average flow measure between neighboring patches is continuous
       if patch.continuous_neighbor_flow == False:
          constraint_2 = False
          break
```

```python

    return constraint_1, constraint_2

# === Grid-Related ===============================================================

def append_grid_sequences(grid_sequences, x_grid, y_grid):

# adds the current grid points to the sequence history array

    x_grid = np.ndarray.flatten(x_grid)
    y_grid = np.ndarray.flatten(y_grid)

    for i, sequence in enumerate(grid_sequences):
        sequence.append(x_grid[i])
        sequence.append(y_grid[i])

    return grid_sequences

# -------------------------------------------------------------------------------

def get_grid_outline( x_grid, y_grid ):

# returns the outline shape (points) of the grid

    n_rows = x_grid.shape[0]
    n_cols = x_grid.shape[1]

    pts = []

    # top row
    for col in range(n_cols):
        pts.append([x_grid[0, col], y_grid[0, col]])
    # right col
    for row in range(n_rows):
        pts.append([x_grid[row, n_cols-1], y_grid[row, n_cols-1]])
    # bottom row
    for col in reversed(range(n_cols)):
        pts.append([x_grid[n_rows-1, col], y_grid[n_rows-1, col]])
    # left col
    for row in reversed(range(n_rows)):
        pts.append([x_grid[row, 0], y_grid[row, 0]])

    return pts

# ----------------------------------------------------------------------------------

def append_grid_prediction_history(x_grid, y_grid, x_grid_ML, y_grid_ML,
                                   x_grid_hist, y_grid_hist,
                                   x_grid_ML_hist, y_grid_ML_hist, ):

# creates history arrays for the ground truth grid and predicted grid

    rows = x_grid.shape[0]
    cols = x_grid.shape[1]

    x_grid = x_grid.flatten()
    y_grid = y_grid.flatten()
    x_grid_ML = x_grid_ML.flatten()
    y_grid_ML = y_grid_ML.flatten()

    for i in range(x_grid.size):
        x_grid_hist[i] = np.append(x_grid_hist[i], x_grid[i])
        y_grid_hist[i] = np.append(y_grid_hist[i], y_grid[i])
        x_grid_ML_hist[i] = np.append(x_grid_ML_hist[i], x_grid_ML[i])
        y_grid_ML_hist[i] = np.append(y_grid_ML_hist[i], y_grid_ML[i])

    return x_grid_hist, y_grid_hist, x_grid_ML_hist, y_grid_ML_hist

# ----------------------------------------------------------------------------------

def create_grid(patches):

# creates a grid array from the patch boundaries

    # determine size of grid - 1 more dimension than number of patches
    n_patches = len(patches)
    n_rows_patch = int(np.sqrt(n_patches))
    n_cols_patch = n_rows_patch
    n_rows_grid = n_rows_patch + 1
    n_cols_grid = n_rows_grid
    x_grid = np.zeros((n_rows_grid, n_cols_grid), dtype = 'int')
    y_grid = np.zeros((n_rows_grid, n_cols_grid), dtype = 'int')

    # start counting rows/cols of patch
    patch_row = 1
    patch_col = 1

    # assign grid points based on corners in each patch
    for i, patch in enumerate(patches):

        patch_num = i + 1

        # grid point indices start at 0
        grid_row = patch_row - 1
        grid_col = patch_col - 1

        # assign top left point
        x_grid[grid_row, grid_col] = patch.corners[0][0]
        y_grid[grid_row, grid_col] = patch.corners[0][1]

        # assign bottom left point on last row
        if patch_row == n_rows_patch:
            x_grid[grid_row+1, grid_col] = patch.corners[3][0]
            y_grid[grid_row+1, grid_col] = patch.corners[3][1]

        # assign top right point on last column
        if patch_col == n_cols_patch:
            x_grid[grid_row, grid_col+1] = patch.corners[1][0]
            y_grid[grid_row, grid_col+1] = patch.corners[1][1]

        # assign bottom left point at end
        if patch_num == n_patches:
            x_grid[grid_row+1, grid_col+1] = patch.corners[2][0]
            y_grid[grid_row+1, grid_col+1] = patch.corners[2][1]
            break

        # increment counters (patchNum indexed to 0)
        if not patch_num == n_patches:
            if patch_num % n_cols_patch == 0:
                patch_col = 1
                patch_row += 1
            else:
                patch_col += 1

    return x_grid, y_grid

```

```
# ---------------------------------------------------------------------------------

def update_grid_flow(patches, x_grid, y_grid):

# moves grid points based on patch movements

    n_patches = len(patches)
    n_cols = int(np.sqrt(n_patches))
    n_rows = n_cols

    # "current" patch will be lower left relative to the grid
    curr_patch = 0

    # move patch corners by average patch U and V flow
    for row in range(n_rows):
        for col in range(n_cols):

            neighbors = []
            u_vals = 0
            v_vals = 0

            # find neighboring patches (LR, LL, UR, UL)

            # top row left
            if row == 0 and col == 0:
                neighbors.append(curr_patch)

            # top row center
            elif row == 0 and 0 < col < n_cols:
                neighbors.append(curr_patch)
                neighbors.append(curr_patch-1)

            # top row right
            elif row == 0 and col == n_cols:
                neighbors.append(curr_patch-1)

            # interior points
            elif 0 < row < n_rows and 0 < col < n_cols:
                neighbors.append(curr_patch)
                neighbors.append(curr_patch-1)
                neighbors.append(curr_patch-1-n_cols)
                neighbors.append(curr_patch-n_cols)

            # bottom row left
            elif row == n_rows and col == 0:
                neighbors.append(curr_patch-n_cols)

            # bottom row center
            elif row == n_rows and 0 < col < n_cols:
                neighbors.append(curr_patch-n_cols)
                neighbors.append(curr_patch-n_cols)

            # bottom row right
            elif row == n_rows and col == n_cols:
                neighbors.append(curr_patch-n_cols-1)

            # left column center
            elif col == 0 and 0 < row < n_rows:
                neighbors.append(curr_patch)
                neighbors.append(curr_patch-n_rows)

            # left column right
            elif col == n_cols and 0 < row < n_rows:
                neighbors.append(curr_patch-1)
                neighbors.append(curr_patch-1-n_rows)

            # get average flow
            for n in neighbors:
                u_vals += patches[n].u_avg
                v_vals += patches[n].v_avg

            x_grid[row, col] += u_vals/len(neighbors)
            y_grid[row, col] += v_vals/len(neighbors)

            curr_patch += 1

    return x_grid, y_grid

# === Drawing ======================================================================

def rectangle_occlusion(frame, ROI, alpha):

# adds a rectangle overlay to a frame

```

```
    # convert frame to RGB
    if len(frame.shape) == 2:
        frame = cv2.cvtColor(frame,cv2.COLOR_GRAY2RGB)

    overlay = np.zeros(frame.shape, dtype=np.uint8)
    overlay = cv2.rectangle(overlay, ROI.start_pt, ROI.end_pt, color =
        colors.white, thickness = -1)

    frame = cv2.addWeighted(overlay, alpha, frame, 1, 0, frame)

    return frame

# --------------------------------------------------------------------------------

def draw_overlays_main(frame, frame_count, frame_range, n_frames, n_last_frame,
    peaks,
                flow_avg_scaled, hr, constraints, basename):

# draws top and bottom information overlays for main analysis

    frame_height = frame.shape[0]
    frame_width = frame.shape[1]
    font_scale = 1

    # apply the transparent header/footer shaded overlay
    overlay = np.zeros(frame.shape, dtype=np.uint8)
    overlay = cv2.rectangle(overlay, (0, 660), (frame_width, frame_height), color
        = colors.white, thickness = -1)
    overlay = cv2.rectangle(overlay, (0, 0), (frame_width, 80), color =
        colors.white, thickness = -1)
    frame = cv2.addWeighted(overlay, 0.7, frame, 1, 0, frame)

    # scale pulse values and draw to frame
    t = []
    flow_avg_scaled_max = max(flow_avg_scaled)

    # had some issues with divide by zero here hence the conditional statements
    # scale to 5% of frame height then offset to bottom
    if abs(flow_avg_scaled_max) > 0:
        flow_avg_scaled = frame_height*0.05 * flow_avg_scaled /
            flow_avg_scaled_max + (frame_height - 50)
    else:
        flow_avg_scaled = frame_height*0.05 * flow_avg_scaled / 0.1 +
            (frame_height - 50)

    for i, point in enumerate(flow_avg_scaled):
        t_increment = (frame_width * i / n_frames)
        t = np.append(t, t_increment)

    flow_avg_scaled = np.vstack((t,flow_avg_scaled)).T.astype(np.int)

    frame = cv2.polylines(frame, [flow_avg_scaled], color = colors.red, thickness
        = 2, isClosed = False)

    # plot cycle peaks and sequence numbers
    if len(peaks) > 0:
        for i, peak in enumerate(peaks):
            x = flow_avg_scaled[peak][0]
            y = flow_avg_scaled[peak][1]
            frame = cv2.drawMarker(frame, tuple(flow_avg_scaled[peak]),
                colors.magenta, cv2.MARKER_CROSS, 5, 4, cv2.LINE_AA)

    # draw failed frame markers
    if draw_failed_frames == True:
        if len(constraints.failed_frames) > 0:
            for frame_number in constraints.failed_frames:
                frame_number_scaled = round((frame_number -
                    frame_range[0])/n_frames * frame.shape[0])
                cv2.drawMarker(frame, (frame_number_scaled, 670 ), colors.black,
                    cv2.MARKER_CROSS, 8, 1, cv2.LINE_AA)

        if len(constraints.reinitialized_frames) > 0:
            for frame_number in constraints.reinitialized_frames:
                frame_number_scaled = round((frame_number -
                    frame_range[0])/n_frames * frame.shape[0])
                cv2.drawMarker(frame, (frame_number_scaled, 680 ), colors.red,
                    cv2.MARKER_TRIANGLE_UP, 8, 1, cv2.LINE_AA)

    # print header text
    # heart rate
    text_hr = "HR: " + str(hr) + ' BPM'
    frame = cv2.putText(frame, text_hr, (560,30), fontFace =
        cv2.FONT_HERSHEY_DUPLEX,
                            fontScale = font_scale, color = colors.dark_green)

    # frame number
    text_fn = str(frame_count) + " of " + str(n_last_frame)
    frame = cv2.putText(frame, text_fn, (10,30), fontFace = cv2.FONT_HERSHEY_DUPLEX,
                        fontScale = font_scale, color = colors.dark_green)

    text_tb = "Total beats: " + str(len(peaks))
    frame = cv2.putText(frame, text_tb, (500,65), fontFace = cv2.FONT_HERSHEY_DUPLEX,
                        fontScale = font_scale, color = colors.dark_green)

    text_fn = basename + ".avi"
    frame = cv2.putText(frame, text_fn, (10,65), fontFace = cv2.FONT_HERSHEY_DUPLEX,
                        fontScale = font_scale, color = colors.dark_green)

    return frame

# ----------------------------------------------------------------------------------

def draw_overlays_extraction_old(frame, corners_1, corners_2, status_1, status_2, frame_width, frame_height,
                 frame_count, n_frames, peaks, new_peak, flow_avg_scaled, basename, hr, ROI_radius):

# draws top and bottom information overlays for feature extraction

    frame_height = frame.shape[0]
    frame_width = frame.shape[1]
    font_scale = 1

    # separate points into lost and found
    if new_peak == False and len(corners_1) > 0 and len(corners_2) > 0:
        status_found_idx_1 = np.where(status_1 == 1)[0]
        status_lost_idx_1 = np.where(status_1 == 0)[0]

        corners_found_1 = corners_1[status_found_idx_1]
        corners_lost_1 = corners_1[status_lost_idx_1]

        if len(peaks) > 1:
            status_found_idx_2 = np.where(status_2 == 1)[0]
```

```
                status_lost_idx_2 = np.where(status_2 == 0)[0]

                corners_found_2 = corners_2[status_found_idx_2]
                corners_lost_2 = corners_2[status_lost_idx_2]

        else:
            if len(peaks > 0):
                corners_found_1 = corners_1
                corners_lost_1 = []

            if len(peaks) > 1:
                corners_found_2 = corners_2
                corners_lost_2 = []

        # draw corners within ROI
        frame = cv2.cvtColor(frame, cv2.COLOR_GRAY2RGB)

        if len(peaks) > 0:
            if (len(corners_found_1) > 0):
                for point in corners_found_1:
                    cv2.drawMarker(frame, tuple(point[0]), colors.magenta,
                        cv2.MARKER_CROSS, 5, 1, cv2.LINE_AA)

            if (len(corners_lost_1) > 0):
                for point in corners_lost_1:
                    cv2.drawMarker(frame, tuple(point[0]), colors.green,
                        cv2.MARKER_CROSS, 20, 2, cv2.LINE_AA)

        if len(peaks) > 1:
            if (len(corners_found_2) > 0):
                for point in corners_found_2:
                    cv2.drawMarker(frame, tuple(point[0]), colors.cyan,
                        cv2.MARKER_CROSS, 5, 1, cv2.LINE_AA)

            if (len(corners_lost_2) > 0):
                for point in corners_lost_2:
                    cv2.drawMarker(frame, tuple(point[0]), colors.green,
                        cv2.MARKER_CROSS, 20, 2, cv2.LINE_AA)

        # draw ROI
        frame = cv2.circle(frame,
                          center = (int(frame_height/2), int(frame_width/2)),
                          radius = ROI_radius,
                          color = colors.green,
                          thickness = 2)

        # apply the header/footer shaded overlay
        overlay = np.zeros(frame.shape, dtype=np.uint8)
        overlay = cv2.rectangle(overlay, (0, 660), (frame_width, frame_height), color
            = colors.white, thickness = -1)
        overlay = cv2.rectangle(overlay, (0, 0), (frame_width, 80), color =
            colors.white, thickness = -1)

        alpha = 0.7
        frame = cv2.addWeighted(overlay, alpha, frame, 1, 0, frame)


        # scale pulse values to frame
        t = []
        flow_avg_scaled_max = max(flow_avg_scaled)

        # had some issues with divide by zero here hence the conditional statements
        # scale to 5% of frame height then offset to bottom
        if abs(flow_avg_scaled_max) > 0:
            flow_avg_scaled = frame_height*0.05 * flow_avg_scaled /
                flow_avg_scaled_max + (frame_height - 50)
        else:
            flow_avg_scaled = frame_height*0.05 * flow_avg_scaled / 0.1 +
                (frame_height - 50)

        for i, point in enumerate(flow_avg_scaled):
            t_increment = (frame_width * i / n_frames)
            t = np.append(t, t_increment)

        flow_avg_scaled = np.vstack((t,flow_avg_scaled)).T.astype(np.int)
        frame = cv2.polylines(frame, [flow_avg_scaled], color = colors.red, thickness
            = 2, isClosed = False)

        # plot cycle peaks and sequence numbers
        if len(peaks) > 0:
            for i, peak in enumerate(peaks):
                x = flow_avg_scaled[peak][0]
                y = flow_avg_scaled[peak][1]

                if i % 2 == 0:
                    peak_color = colors.magenta
                    frame = cv2.line(frame, (x, y-10), (x, 670), colors.black, 2)
                    if i != 0:
                          frame = cv2.putText(frame, str(int(i-1)), (x-5,795), fontFace =
                              cv2.FONT_HERSHEY_DUPLEX,
                                          fontScale = .5, color = colors.black)
                    frame = cv2.drawMarker(frame, tuple(flow_avg_scaled[peak]),
                          peak_color, cv2.MARKER_CROSS, 5, 4, cv2.LINE_AA)

                else:
                    peak_color = colors.cyan
                    frame = cv2.line(frame, (x, y+10), (x, 800), colors.black, 2)
                    frame = cv2.putText(frame, str(int(i-1)), (x-5,680), fontFace =
                          cv2.FONT_HERSHEY_DUPLEX,
                                      fontScale = .5, color = colors.black)
                    frame = cv2.drawMarker(frame, tuple(flow_avg_scaled[peak]),
                          peak_color, cv2.MARKER_CROSS, 5, 4, cv2.LINE_AA)


        # print header text
        text_fn = str(frame_count) + " of " + str(n_frames)
        frame = cv2.putText(frame, text_fn, (10,30), fontFace =
            cv2.FONT_HERSHEY_DUPLEX,
                              fontScale = font_scale, color = colors.dark_green)

        text_hr = "HR: " + str(hr) + ' BPM'
        frame = cv2.putText(frame, text_hr, (560,30), fontFace =
            cv2.FONT_HERSHEY_DUPLEX,
                              fontScale = font_scale, color = colors.dark_green)

        text_tb = "Total beats: " + str(len(peaks))
        frame = cv2.putText(frame, text_tb, (500,65), fontFace =
            cv2.FONT_HERSHEY_DUPLEX,
                              fontScale = font_scale, color = colors.dark_green)

        text_fn = basename + ".mat"
        frame = cv2.putText(frame, text_fn, (10,65), fontFace =
            cv2.FONT_HERSHEY_DUPLEX,
                              fontScale = font_scale, color = colors.dark_green)

        return frame
```

```

# ------------------------------------------------------------------------------

def draw_overlays_extraction(frame, corners_1, status_1, frame_width, frame_height,
                    frame_count, n_frames, peaks, new_peak, flow_avg_scaled,
                      basename, hr, ROI_radius):

# draws top and bottom information overlays for feature extraction

    frame_height = frame.shape[0]
    frame_width = frame.shape[1]
    font_scale = 1

    # separate points into lost and found
    if len(corners_1) > 0 and new_peak == False:
        status_found_idx_1 = np.where(status_1 == 1)[0]
        status_lost_idx_1 = np.where(status_1 == 0)[0]

        corners_found_1 = corners_1[status_found_idx_1]
        corners_lost_1 = corners_1[status_lost_idx_1]

    else:
        if len(peaks > 0):
            corners_found_1 = corners_1
            corners_lost_1 = []

    # draw corners within ROI
    frame = cv2.cvtColor(frame, cv2.COLOR_GRAY2RGB)

    if len(peaks) > 0:
        if (len(corners_found_1) > 0):
            for point in corners_found_1:
                cv2.drawMarker(frame, tuple(point[0]), colors.magenta,
                    cv2.MARKER_CROSS, 5, 1, cv2.LINE_AA)

        if (len(corners_lost_1) > 0):
            for point in corners_lost_1:
                cv2.drawMarker(frame, tuple(point[0]), colors.green,
                    cv2.MARKER_CROSS, 20, 2, cv2.LINE_AA)

    # draw ROI
    frame = cv2.circle(frame,
                      center = (int(frame_height/2), int(frame_width/2)),
                      radius = ROI_radius,
                      color = colors.green,
                      thickness = 2)

    # apply the header/footer shaded overlay
    overlay = np.zeros(frame.shape, dtype=np.uint8)
    overlay = cv2.rectangle(overlay, (0, 660), (frame_width, frame_height), color = colors.white, thickness = -1)
    overlay = cv2.rectangle(overlay, (0, 0), (frame_width, 80), color = colors.white, thickness = -1)

    alpha = 0.7
    frame = cv2.addWeighted(overlay, alpha, frame, 1, 0, frame)

    # scale pulse values to frame
    t = []
    flow_avg_scaled_max = max(flow_avg_scaled)

    # had some issues with divide by zero here hence the conditional statements
    # scale to 5% of frame height then offset to bottom
    if abs(flow_avg_scaled_max) > 0:
        flow_avg_scaled = frame_height*0.05 * flow_avg_scaled / flow_avg_scaled_max + (frame_height - 50)
    else:
        flow_avg_scaled = frame_height*0.05 * flow_avg_scaled / 0.1 + (frame_height - 50)

    for i, point in enumerate(flow_avg_scaled):
        t_increment = (frame_width * i / n_frames)
        t = np.append(t, t_increment)

    flow_avg_scaled = np.vstack((t,flow_avg_scaled)).T.astype(np.int)
    frame = cv2.polylines(frame, [flow_avg_scaled], color = colors.red, thickness = 2, isClosed = False)

    # plot cycle peaks and sequence numbers
    if len(peaks) > 0:
        for i, peak in enumerate(peaks):
            x = flow_avg_scaled[peak][0]
            y = flow_avg_scaled[peak][1]

            peak_color = colors.magenta
            frame = cv2.drawMarker(frame, tuple(flow_avg_scaled[peak]), peak_color,
                cv2.MARKER_CROSS, 5, 4, cv2.LINE_AA)

    # print header text
    text_fn = str(frame_count) + " of " + str(n_frames)
    frame = cv2.putText(frame, text_fn, (10,30), fontFace =
        cv2.FONT_HERSHEY_DUPLEX,
                          fontScale = font_scale, color = colors.dark_green)

    text_hr = "HR: " + str(hr) + ' BPM'
    frame = cv2.putText(frame, text_hr, (560,30), fontFace =
        cv2.FONT_HERSHEY_DUPLEX,
                          fontScale = font_scale, color = colors.dark_green)

    text_tb = "Total beats: " + str(len(peaks))
    frame = cv2.putText(frame, text_tb, (500,65), fontFace =
        cv2.FONT_HERSHEY_DUPLEX,
                          fontScale = font_scale, color = colors.dark_green)

    text_fn = basename + ".avi"
    frame = cv2.putText(frame, text_fn, (10,65), fontFace =
        cv2.FONT_HERSHEY_DUPLEX,
                          fontScale = font_scale, color = colors.dark_green)

    return frame

# --------------------------------------------------------------------------------

def draw_grid_lines(frame, x_grid, y_grid, color):

# draw gridlines on frame

    # plot grid lines over an image
    n_cols = len(x_grid)
    n_rows = n_cols

    if n_cols > 1:

        for row in range(n_rows):
            for col in range(n_cols):

                # draw vertical lines
                if row < n_rows - 1:
                    cv2.line(frame,
                        tuple( (x_grid[row, col], y_grid[row, col]) ),
                        tuple( (x_grid[row+1, col], y_grid[row+1, col]) ),
                        color)

                    # draw left/right outline
                    if col == 0 or col == n_cols - 1:
                        cv2.line(frame,
                            tuple( (x_grid[row, col], y_grid[row, col]) ),
                            tuple( (x_grid[row+1, col], y_grid[row+1, col]) ),
                            color, grid_outline_thickness)


                # draw horizontal lines
                if col < n_cols - 1:
                    cv2.line(frame,
                        tuple( (x_grid[row, col], y_grid[row, col]) ),
                        tuple( (x_grid[row, col+1], y_grid[row, col+1]) ),
                        color)

                    # draw top/bottom outline
                    if row == 0 or row == n_rows - 1:
                        cv2.line(frame,
                            tuple( (x_grid[row, col], y_grid[row, col]) ),
                            tuple( (x_grid[row, col+1], y_grid[row, col+1]) ),
                            color, grid_outline_thickness)

    return frame

# -----------------------------------------------------------------------------------

def create_grid_plots(x_grid_hist, y_grid_hist, x_grid_ML_hist, y_grid_ML_hist,
    basename, output_path, text_output_file):

    # calculations

    # absolute distances
    x_grid_hist = np.asarray(x_grid_hist)
    y_grid_hist = np.asarray(y_grid_hist)
    x_grid_ML_hist = np.asarray(x_grid_ML_hist)
    y_grid_ML_hist = np.asarray(y_grid_ML_hist)

    x_grid_diff_abs = np.abs(x_grid_hist - x_grid_ML_hist)
    y_grid_diff_abs = np.abs(y_grid_hist - y_grid_ML_hist)
    grid_diff_abs = np.zeros_like(x_grid_diff_abs)

    for i in range(len(x_grid_hist)):
        for j in range(len(x_grid_hist[0])):
            point1 = (x_grid_hist[i][j], y_grid_hist[i][j])
            point2 = (x_grid_ML_hist[i][j], y_grid_ML_hist[i][j])
            grid_diff_abs[i][j] = np.abs(distance.euclidean(point1, point2))

    x_max_idx = np.argmax(x_grid_hist, axis=0)

    x_grid_avg_diff_abs = np.mean(x_grid_diff_abs, axis=0)
    x_grid_max_diff_abs = np.max(x_grid_diff_abs, axis=0)
    x_grid_min_diff_abs = np.min(x_grid_diff_abs, axis=0)

    y_grid_avg_diff_abs = np.mean(y_grid_diff_abs, axis=0)
    y_grid_max_diff_abs = np.max(y_grid_diff_abs, axis=0)
    y_grid_min_diff_abs = np.min(y_grid_diff_abs, axis=0)

    grid_avg_diff_abs = np.mean(grid_diff_abs, axis=0)
    grid_max_diff_abs = np.max(grid_diff_abs, axis=0)
    grid_min_diff_abs = np.min(grid_diff_abs, axis=0)

    f_nos = np.arange(1, len(x_grid_avg_diff_abs)+1)

    yerr_x_abs = [np.abs(x_grid_min_diff_abs-x_grid_avg_diff_abs),
        np.abs(x_grid_max_diff_abs-x_grid_avg_diff_abs)]
    yerr_y_abs = [np.abs(y_grid_min_diff_abs-y_grid_avg_diff_abs),
        np.abs(y_grid_max_diff_abs-y_grid_avg_diff_abs)]
    yerr_abs = [np.abs(grid_min_diff_abs-grid_avg_diff_abs),
        np.abs(grid_max_diff_abs-grid_avg_diff_abs)]

    # plot grid differences - average
    fig, axs = plt.subplots(3, figsize=(8.5,5.5))
    title_string = 'Average Grid Prediction Error (pixels) \n (' + basename +
        '.avi)'
    fig.suptitle(title_string)
    axs[0].errorbar(f_nos, x_grid_avg_diff_abs, yerr = yerr_x_abs, label =
        'x-diff', linewidth=2, elinewidth = 1, capsize = 3)
    axs[0].plot(f_nos, x_grid_avg_diff_abs, 'ok')
    axs[0].set_xticks(f_nos, minor=False)
    axs[0].tick_params(labelsize=13)
    axs[0].legend(loc = 2, ncol = 1)
    axs[0].grid(color = 'lightgray', linestyle = '--', linewidth = 1)

    axs[1].errorbar(f_nos, y_grid_avg_diff_abs, yerr = yerr_y_abs, label =
        'y-diff', linewidth=2, elinewidth = 1, capsize = 3)
    axs[1].plot(f_nos, y_grid_avg_diff_abs, 'ok')
    axs[1].set_xticks(f_nos, minor=False)
    axs[1].tick_params(labelsize=13)
    axs[1].legend(loc = 2, ncol = 1)
    axs[1].grid(color = 'lightgray', linestyle = '--', linewidth = 1)

    axs[2].errorbar(f_nos, grid_avg_diff_abs, yerr = yerr_abs, label = 'xy-diff',
        linewidth=2, elinewidth = 1, capsize = 3)
    axs[2].plot(f_nos, grid_avg_diff_abs, 'ok')
    axs[2].set_xticks(f_nos, minor=False)
    axs[2].tick_params(labelsize=13)
    axs[2].legend(loc = 2, ncol = 1)
    axs[2].grid(color = 'lightgray', linestyle = '--', linewidth = 1)

    fig.text(0.5, 0.01, 'Predicted Frame Number', ha='center', fontsize=13)
    fig.text(0.01, 0.5, 'Difference from Ground Truth (pixels)', va='center',
        rotation='vertical', fontsize=13)

    fig.subplots_adjust(hspace=0.8, left=0.1, right=0.99)

    plt.savefig(output_path + 'plots/' + 'grid_diff_avg_' + basename + '.png')
    plt.show()

    # plot grid differences - histogram - first frame
    n_bins = 10
    plt.hist(x_grid_diff_abs[:,0], bins='auto', facecolor='blue', alpha=0.5,
        edgecolor='black')
    title_string = 'First Frame Differences (pixels) \n (' + basename + '.avi)'
    plt.title(title_string)
    plt.xlabel('Difference from Ground Truth (pixels)')
    plt.ylabel('Number of Values')
    plt.savefig(output_path + 'plots/' + 'grid_diff_hist_patches_' + basename +
        '.png')
    plt.show()

    # plot grid differences - boxplot - patches
    patch_nos = np.arange(x_grid_diff_abs.shape[0])+1
    plt.boxplot(np.transpose(x_grid_diff_abs), meanline = True)
    title_string = 'Per Patch Grid Prediction Error (pixels) \n (' + basename +
        '.avi)'
    plt.title(title_string)
    plt.grid(color = 'lightgray', linestyle = '--', linewidth = 1)
    plt.xlabel('Patch Number')
    plt.ylabel('Difference from Ground Truth (pixels)')
    plt.xticks(patch_nos,patch_nos-1)
    plt.savefig(output_path + 'plots/' + 'grid_diff_patches_' + basename + '.png')
    plt.show()

    f1 = 0
    f3 = len(grid_avg_diff_abs)-1
    f2 = int(f3/2)

    fout = open(text_output_file,'a')
    print("\n", file=fout)
    print("x-y Distances", file=fout)
    print("   F", f1, "|F", f2, "|F", f3, file=fout)
    print("Min:", round(grid_min_diff_abs[f1], 1), round(grid_min_diff_abs[f2],1),
        round(grid_min_diff_abs[f3],1), file=fout)
    print("Avg:", round(grid_avg_diff_abs[f1], 1), round(grid_avg_diff_abs[f2],1),
        round(grid_avg_diff_abs[f3],1), file=fout)
    print("Max:", round(grid_max_diff_abs[f1], 1), round(grid_max_diff_abs[f2],1),
        round(grid_max_diff_abs[f3],1), file=fout)
    print(" ")
    fout.close()

    return

# --------------------------------------------------------------------------------

def create_timeline_plots(constraints, n_last_frame, diastole_frames,
    main_tracking_start_frame, occluded_frames, basename, output_path):

    # calculations
    predicted_frames = np.arange(occluded_frames[0], occluded_frames[1])
    frame_numbers = np.arange(0, n_last_frame)

    diastole_frames_y = np.empty(len(diastole_frames))
    diastole_frames_y.fill(2)
    reinitialized_frames_y = np.empty(len(constraints.reinitialized_frames))
    reinitialized_frames_y.fill(3)
    failed_frames_y = np.empty(len(constraints.failed_frames))
    failed_frames_y.fill(4)
    predicted_frames_y = np.empty(len(predicted_frames))
    predicted_frames_y.fill(5)

    # plot timeline
    plt.figure(figsize=(8, 3))
    plt.plot(main_tracking_start_frame, 1, '|k', markersize = 10)
    plt.plot(diastole_frames, diastole_frames_y, '|r', markersize = 10)
    plt.plot(constraints.reinitialized_frames, reinitialized_frames_y, '|m',
        markersize = 10)
    plt.plot(constraints.failed_frames, failed_frames_y, '|b', markersize = 10)
    plt.plot(predicted_frames, predicted_frames_y, '|y', markersize = 10)

    title_string = 'Analysis Timeline \n (' + basename + '.avi)'
    plt.title(title_string)
    plt.xlabel('Frame Number', fontsize = 12)
    ax = plt.gca()
    ax.xaxis.grid()
    plt.xticks(fontsize=12)
    ax.set_xlim(0, n_last_frame)
    ax.set_ylim(0, 6)
    ax.set_yticks([1, 2, 3, 4, 5])
    ax.set_yticklabels(['Tracking Start', 'Diastole Frames', 'Tracking Resets',
        'Failed Constraints', 'Predicted Frames'], fontsize = 12)
    plt.tight_layout()
    plt.savefig(output_path + 'plots/' + 'timeline_' + basename + '.png',
        bbox_inches='tight')
    plt.show()

    return

# ---------------------------------------------------------------------------------

def write_frame(frame, ROI, patches, corners, contours, hierarchy, vid, x_grid,
                y_grid, x_grid_ML, y_grid_ML, frame_count, frame_range,
                    occluded_frames,
                n_frames, n_last_frame, peaks, flow_avg_scaled, hr, constraints,
                    basename):

# draws on current frame and writes to file

    # convert frame to RGB
    if len(frame.shape) == 2:
        frame_RGB = cv2.cvtColor(frame,cv2.COLOR_GRAY2RGB)
    else:
        frame_RGB = frame

    # draw overlays including pulse
    if draw_overlays == True:
        frame_RGB = draw_overlays_main(frame_RGB, frame_count, frame_range,
            n_frames, n_last_frame, peaks, flow_avg_scaled, hr, constraints,
            basename)

    # draw corner points
    if draw_corners == True:
        # draw corners within ROI
        if len(corners) > 0:
            for point in corners:
                cv2.drawMarker(frame_RGB, tuple(point[0]), colors.magenta,
                    cv2.MARKER_CROSS, 5, 1, cv2.LINE_AA)

    # draw contours
    if draw_contours == True:
        if len(contours) > 0:
            frame_RGB = cv2.drawContours(frame_RGB, contours, -1, hierarchy =
                hierarchy, \
                                    maxLevel = contours_max_level, color =
                                        colors.yellow, thickness = 1)

    # draw patch corners
    if draw_patch_corners == True:
        for i in range(0, len(patches)):
            cv2.line(frame_RGB, tuple(patches[i].corners[0]),
                tuple(patches[i].corners[1]), colors.yellow, 1, 1)
            cv2.line(frame_RGB, tuple(patches[i].corners[1]),
                tuple(patches[i].corners[2]), colors.yellow, 1, 1)
            cv2.line(frame_RGB, tuple(patches[i].corners[2]),
                tuple(patches[i].corners[3]), colors.yellow, 1, 1)
```

```python
            cv2.line(frame_RGB, tuple(patches[i].corners[3]), tuple(patches[i].corners[0]), colors.yellow, 1, 1)

    # draw grids
    if draw_grid == True:
        frame_RGB = draw_grid_lines(frame_RGB, x_grid, y_grid, grid_color_normal)

    if draw_grid_ML == True and occluded_frames[0] <= frame_count <= occluded_frames[1]:
        frame_RGB = draw_grid_lines(frame_RGB, x_grid_ML, y_grid_ML, grid_color_ML)

    # draw ROI
    if draw_ROI == True:
        cv2.rectangle(frame_RGB, ROI.start_pt, ROI.end_pt, colors.green, thickness = 2)

    # write frame to file - needs RGB
    vid.write(frame_RGB)

    return frame_RGB
```

A.6 MAT to AVI Conversion

```
% ==================================================================================
%
% file:    mat2avi.m
% version: v1.0.0
% author:  Billy Hempstead
%
% summary: Converts a .mat video file to a .avi file
%
% ==================================================================================

clear all
close all
clc

frameskip = 2;

% select and load data file
load_msg = 'Loading Datafile';
disp(load_msg)

[filename, pathname] = uigetfile('*.mat');
fullpath = strcat(pathname,filename);
load(fullpath)

% setup output video
output_file = strcat(pathname,filename(1:end-4),'.avi');
output_vid = VideoWriter(output_file, 'Grayscale AVI');
fps = IDS.CAM.FPS;
output_vid.FrameRate = fps/frameskip;
open(output_vid);

% write frames to output video
n_frames = length(IDS.ImgVIS);

for i = 1:frameskip:n_frames
    clc
    update_msg = strcat({'Processing frame '}, num2str(i), {' of '},
        num2str(n_frames));
    disp(update_msg)
    frame = IDS.ImgVIS(:,:,i);
    writeVideo(output_vid,frame);
end

close(output_vid);
```