William & Mary
CHARTERED 1693

W&M ScholarWorks

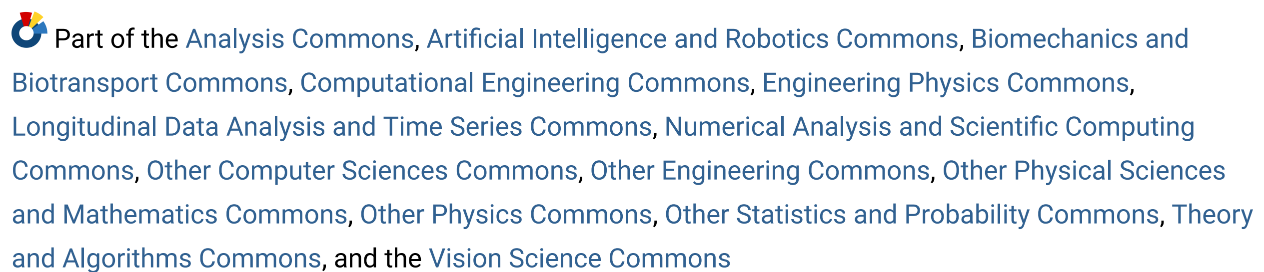
Undergraduate Honors Theses | Theses, Dissertations, & Master Projects

5-2020

Gait Characterization Using Computer Vision Video Analysis

Martha T. Gizaw
College of William and Mary

Follow this and additional works at: https://scholarworks.wm.edu/honorstheses



Part of the Analysis Commons, Artificial Intelligence and Robotics Commons, Biomechanics and Biotransport Commons, Computational Engineering Commons, Engineering Physics Commons, Longitudinal Data Analysis and Time Series Commons, Numerical Analysis and Scientific Computing Commons, Other Computer Sciences Commons, Other Engineering Commons, Other Physical Sciences and Mathematics Commons, Other Physics Commons, Other Statistics and Probability Commons, Theory and Algorithms Commons, and the Vision Science Commons

Recommended Citation

Gizaw, Martha T., "Gait Characterization Using Computer Vision Video Analysis" (2020). *Undergraduate Honors Theses.* Paper 1510.
https://scholarworks.wm.edu/honorstheses/1510

This Honors Thesis is brought to you for free and open access by the Theses, Dissertations, & Master Projects at W&M ScholarWorks. It has been accepted for inclusion in Undergraduate Honors Theses by an authorized administrator of W&M ScholarWorks. For more information, please contact scholarworks@wm.edu.

Gait Characterization Using Computer Vision Video Analysis

A thesis submitted in partial fulfillment of the requirement
for the degree of Bachelor of Science with Honors in
Interdisciplinary Studies from The College of William and Mary in Virginia,

by

Martha Gizaw

Accepted for Honors

Dr. William Cooke, Director

Dr. Keith Griffioen, Physics

Dr. Evie Burnet, Kinesiology & Health Sciences

Dr. Dennis Manos, Physics

Williamsburg, Virginia
Tuesday, May 5, 2020

Gait Characterization Using Computer Vision

Video Analysis

Table of Contents

Acknowledgments iii

Dedications vi

List of Figures vii

List of Tables ix

Abstract x

Chapter 1: Project Background and Objectives 1

Section 1.1: Falls in Human Health 1

Section 1.2: Current Studies in Greater Williamsburg 1

Section 1.3: Video Frame Illustration 2

Section 1.4: Description of Computer Vision 3

Section 1.5: Project Roadmap 4

Chapter 2: Standard Computer Vision Feature Detectors 5

Section 2.1: FAST 5

Section 2.2: ORB 7

Section 2.3: Canny Edges 7

Section 2.4: Hough Lines 8

Chapter 3: Properties & Behaviors of a Blob Detector 10

Section 3.1: Definition and Characterization of Blobs 10

Section 3.2: Choice of Parameters 11

Chapter 4: Correlation of Blobs Between Frames 17

Section 4.1: Blob Classification 17

Section 4.2: Descriptor Matching 19

Section 4.2.1: Blobs Already Identified 19

Section 4.2.2: Blobs Still Unidentified 20

Section 4.2.3: The Blob Renumbering Process 20

Section 4.3: Plotting Blob Trajectories 21

Section 4.3.1: Missing Trajectory Regions 23

Section 4.3.2: Blob Misidentification 24

Chapter 5: Association of Blobs with Human Features 26

Section 5.1: Combining Multiple Trajectories 26

Section 5.1.1: Blobs Appearing Twice in One Frame 29

Section 5.2: Stability of Trajectory Regions 30

Section 5.3: Left-Side Versus Right-Side Markers 31

Chapter 6: Project Conclusions 34

Section 6.1: Overall Review 34

Section 6.2: Future Directions 34

Appendix A: Blob Detection & Identification Flowchart 36

Appendix B: List of OpenCV-Python Programs 45

Appendix C: Full-Length Source Codes (Electronic Copy Exclusive) 49

Bibliography 84

Vita 87

Acknowledgments

First and foremost, I want to express my extreme gratitude to Dr. William Cooke for his patience, guidance, and constructive criticism throughout this highly significant investigation. I am also indebted to Dr. Jeffery Nelson for his never-ending assistance with finding my project advisor last year. I sincerely thank Dr. Keith Griffioen for his exceptional insights in each step of the thesis composition and presentation process. My thankfulness extends to Dr. Evie Burnet of Kinesiology & Health Sciences and Dr. Dennis Manos of Physics, both of whom agreed to join my Examining Committee along with Dr. Cooke and Dr. Griffioen to see my work go above and beyond the expectations of this endeavor. I give credit to the Center for Balance & Aging Studies for providing one of the videos to experiment with throughout the project.

The rest of the Department of Physics, including the Society of Physics Students, deserves my gratitude as well because it offers a welcoming and supportive environment for me to achieve my own goals and finish my bachelor's degree once and for all. Without its new Engineering Physics & Applied Design initiative, I would not be able to create a biomedical engineering concentration that draws courses from my previous programs of study: neuroscience, computer science, and mathematical biology. This full thesis created for the Department of Physics should stand as the final opportunity for me to demonstrate my appreciation toward fellow students, alumni, and friends near and far who eagerly donated to my video proposal during the William & Mary Honors Fellowship Campaign in March 2019.

Next, thank you to all the counselors, instructors, and friends who instill in me a sense of belonging beyond the classroom. Over the years, the Charles Center has become the academic advisor beyond my imagination, and I cannot appreciate it enough for guiding me through

independent scholarship. From the honors project to the engineering capstone to the journalism seminar, its opportunities make me realize that I learn best with interdisciplinary thought.

As a student with autism, anxiety, and attention deficit disorders, I sought the Neurodiversity Initiative as the best reason for coming to William & Mary. Besides receiving accommodations from Student Accessibility Services, I have been networking with John Elder Robison, who brings thoughtful insights toward how the disabled should shape our world. I want to take the time to offer my gratitude to him since his words have enabled me to speak up at Student Assembly on the common issues among people with disabilities who succeed in college and their careers.

Attaining a well-rounded education as a person from a single-mother household is not possible without the power of scholarships. When I received news from Northern Virginia Community College that the Honors Program and Pathway to the Baccalaureate will fully fund my tuition for my service and academic achievement, I took the risk of attending there as an ambitious high school graduate. Most of the time at my home campus, I felt like an outlier among the students who do not meet their goals the same way I did. However, I found a wonderful support group called Women in Search of Excellence, and it empowered me to sharpen my professional skills the right way. Never would I prove to my family that I can independently manage my career options without a Jack Kent Cooke Foundation scholarship lowering my stress of moving on to a new academic life. I thank the following people who invested in my success: Stacy Rice, Dr. Paul Fitzgerald, LeeAnn Thomas, Theana Kastens, Camisha Parker, Dr. John Sound, Dr. Rebecca Hayes, Alex Coppelman, Shannon Bobb, Kerry Coleman-Proksch, Tykesha Myrick, Dr. Scott Ralls, Jennifer Krasilovsky, and all of the current and former Cooke scholars.

Finally, I am more than excited to acknowledge the teachers and staff—past and present—at Freedom High School in Woodbridge, VA. They all are hard at work to inspire a diverse body of students and alumni to overcome their obstacles. My other alma mater schools in Virginia—Featherstone, Leesylvania, and Rippon—also deserve my gratitude since they lifted me toward the future I wanted.

This document cannot be complete without stating how much I love my mother Sophia, my brother Daniel, and my sister Hannah. I also love my father Tibebe, who currently lives in San Francisco, and my half-brother Brooke, who lives with his mother in Dallas. Other close relatives who comforted me and invested in my success include my aunt Nunu, my cousins Nigus and Sedenia, and my family friends Selam, Lili, Senti, Abraham, Rebecca, Grace, Etafi, Ojay, and Atoki.

Dedications

To my grandfathers Demissew (c. 1929-2014) and Mekuria (c. 1936-2017), who kept in touch with their families and supported their children and grandchildren's academic and personal endeavors.

To Drs. Harold O. Levy (1952-2018) and Stuart A. Haney (1957-2020), who last served as the executive director and charter board member, respectively, of the Jack Kent Cooke Foundation. Both gentlemen had a passion for helping disadvantaged students attain a world-class education.

List of Figures

Chapter 1

Figure 1.1: Original Video Frame 2

Chapter 2

Figure 2.1: How FAST Works 5

Figure 2.2: FAST Implementation 6

Figure 2.3: ORB Implementation 7

Figure 2.4: Canny Edge Implementation 8

Figure 2.5: Hough Line Implementation 9

Chapter 3

Figure 3.1: Visual Definitions of Blob Parameters 11

Figure 3.2: Poor Blob Detection 12

Figure 3.3: Blob Detection with Default Parameter Values 12

Figure 3.4: Sensitivity Chart for Minimum Area 13

Figure 3.5: Sensitivity Chart for Circularity 13

Figure 3.6: Keypoint Analysis with Threshold Step 14

Figure 3.7: Keypoint Analysis with Lack of Circularity 15

Figure 3.8: Blob Detection with Current Parameter Values 16

Chapter 4

Figure 4.1: Initial Versus New Blob Labeling 19

Figure 4.2: Tracking an Unidentified Blob 20

Figure 4.3: Video Axis Measurements 22

Figure 4.4: First Trajectory Example 23

Figure 4.5: How to Fill in Trajectory Gaps 24

Figure 4.6: Spotting Trajectory Discontinuities 25

Figure 4.7: Video Proof for Discontinuities 25

Chapter 5

Figure 5.1: Two Names for One Marker 26

Figure 5.2: Trajectories for Two Names 27

Figure 5.3: Combined Left Heel Trajectory 28

Figure 5.4: Left Knee Trajectory 29

Figure 5.5: Twice-Appearing Blobs at the Left Toe 30

Figure 5.6: Right Heel Trajectory 32

Figure 5.7: First Hiding Event 33

Figure 5.8: Second Hiding Event 33

List of Tables

Chapter 4

Table 4.1: Marker Name Placement 21

Abstract

The World Health Organization reports that falls are the second-leading cause of accidental death among senior adults around the world. Currently, a research team at William & Mary's Department of Kinesiology & Health Sciences attempts to recognize and correct aging-related factors that can result in falling. To meet this goal, the members of that team videotape walking tests to examine individual gait parameters of older subjects. However, they undergo a slow, laborious process of analyzing video frame by video frame to obtain such parameters. This project uses computer vision software to reconstruct walking models from residents of an independent living retirement community. Those subjects have agreed to be tested bi-annually and to report their fall history. Videos previously recorded demonstrate a variety of walks. Our procedures use several OpenCV-Python functions to detect, label, and follow markers that have been placed on the subjects' shoes and knees. The trajectories followed by these markers allow us to generate walking models with gait parameters, such as the step height and the ankle dorsiflexion angle. This computer vision video analysis runs unsupervised to reduce processing time dramatically while enhancing the accuracy of a variety of measurements. Therefore, our data processing techniques will enable our kinesiology investigators to quickly generate a more extensive data set to learn how falling problems develop. This outcome will allow them to develop and to test exercises that can reduce those problems and prevent future falls for older subjects.

Keywords: computer vision, falls, gait, seniors

Chapter 1: Project Background and Objectives

Section 1.1: Falls in Human Health

Kellog International defines a fall as an event that involves intentional or unintentional resting on the ground or floor [1]. The World Health Organization (WHO) states that falls are the second leading cause of accidental death worldwide [2]. In a 2017 issue of *IEEE Pulse*, falls are also the sixth leading cause of death among the elderly in the United States [3]. About 28-35 percent of adults over the age of 65 are falling each year [4]. These statistics are growing with age, frailty level, and gait variability. By 2050, over 1 in 5 individuals will be seniors, which results in higher rates of fall-related injuries at 20-30 percent and emergency visits at 10-15 percent [4][5].

With gait as a manner of walking and a measurable fall risk factor, developing fall prevention systems becomes possible. A research team under Dr. Kabalan Chaccour of Antonine University in Lebanon describes those systems as devices that can sense, process, and communicate essential data in the event of a fall [1]. To this day, falls prevention is one of the most pressing issues that require collaboration between a scientist and an engineer. In health science, measuring the gait of the senior-aged subject is crucial to establishing thresholds for when he or she is likely to fall.

Section 1.2: Current Studies in Greater Williamsburg

In 2018, the Department of Kinesiology and Health Sciences at the College of William & Mary established the Center for Balance and Aging Studies (CBAS) [6]. Since then, principal investigators Dr. Evie Burnet and Dr. Michael Deschenes, and student research assistants search for and evaluate factors that can increase fall risk among local senior citizens. Williamsburg Landing and the James City County Recreation Center have proposed to collaborate with CBAS on intervening with and evaluating participating senior adults to collect gait measurements [7]. So

far at Williamsburg Landing, CBAS utilizes the GAITRite walkway and high-speed camera systems to detect different motions that can lead to a fall [8]. GAITRite is a computer-connected, pressure-sensitive pathway system that records each footstep along a subject's walking path. A video camera films the passage through the GAITRite mat to track the subject's step height and ankle dorsiflexion angle [8].

The data from gait analysis tests at Williamsburg Landing suggest that future data should be automated. Currently, the kinesiology team observes and interprets videos frame by frame to manually measure step heights and angles. These tasks do not provide necessarily accurate information because the camera may not be viewing the limbs from a correct angle [9]. The development of new technologies is thus necessary to retest the subjects to gather measurements that are more accurate and readable. In fact, this project intends to use image processing methods, such as computer vision—which is defined in Section 1.4—to quickly collect and interpret gait data from desired video or photo elements.

Section 1.3: Video Frame Illustration

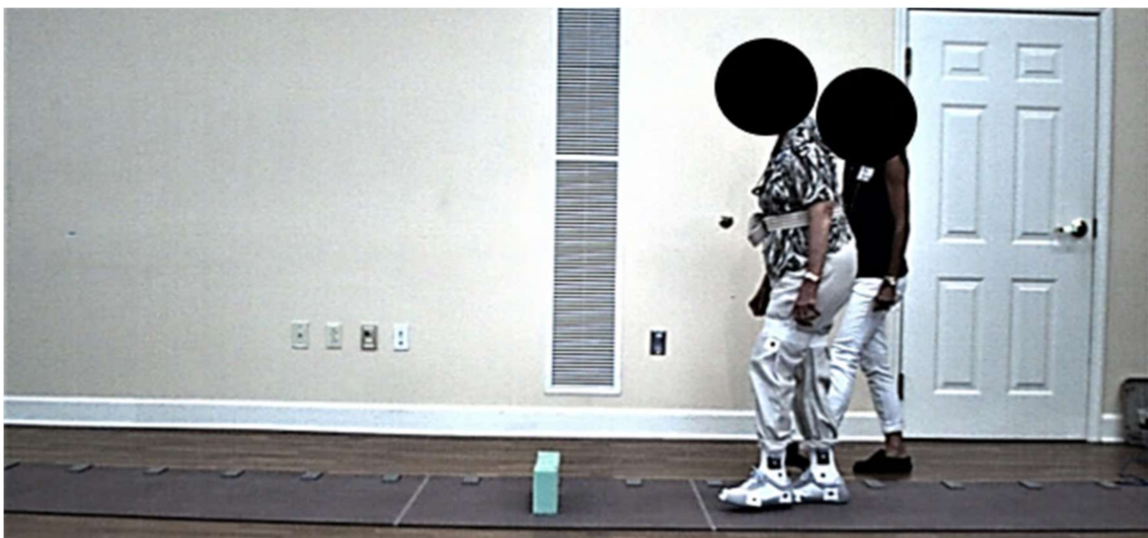


Figure 1.1: Original Video Frame. An original video labeled "S10.B1" is displayed to the user before any computer vision techniques can be applied [10].

Figure 1.1 is an original video frame from an AVI file titled "S10.B1," which shows a principal investigator and a test subject walking down the GAITRite mat, as well as that subject stepping over a foam block barrier [10]. The black dots described as hand-drawn gait markers were attached to the subject's legs because the kinesiology researchers elected to undergo the time-consuming procedure of measuring those markers in each frame. A possible argument for why videotaping a walking subject is necessary is that it not only syncs with the footprint path that GAITRite creates, but it also provides an opportunity for estimating step height and foot flex angle before considering any new technologies. Without image processing automating the video frame analysis, the researchers will spend more time producing complex data by frequently pausing videos, measuring objects with pixels, and observing how the feet extend.

Section 1.4: Description of Computer Vision

Computer vision is an interdisciplinary field of science that enables computers to see and process digital images and videos. At the forefront of such high-tech disciplines as electrical engineering and computer science, the significance of computer vision rests with the recognition, reconstruction, generation, and processing of images to solve vision problems. Likewise, developing algorithms and building models of the human visual system have sparked interest among computer scientists, neuroscientists, and physicists [11].

The most popular and well-documented open-source library for programmers who are new to image processing is OpenCV, which is short for Open Source Computer Vision. Started by Gary Bradsky in 1999 [12], OpenCV is expanding to support a variety of algorithms related to computer vision and machine learning. Also, it endorses several programming languages such as C++, Python, and Java, and it is available to download on Windows, Linux, Android, and Mac OS [13]. Several tutorials in the OpenCV documentation navigate this project to down-selection to specific

methods for video processing and feature detection deemed appropriate for gait characterization [14].

Section 1.5: Project Roadmap

The next few chapters lay out a computer vision procedure that CBAS can use in the future. First, Chapter 2 describes four standard feature detectors—FAST, ORB, Canny Edge, and Hough Line—that attempted to locate the subject's spots but expected unnecessary details from them and other video elements. Those methods were tried out before a blob detector was chosen to pinpoint the markers precisely. Next, Chapter 3 defines blobs and explains how one would prioritize the selection of blob detector video parameters and adjust their values to detect most of the subject's markers with similar size, shape, and color. Afterward, Chapter 4 delineates the process of matching and renumbering markers between frames based on changes in physical location so that one can establish useful trajectories during walking activity. In Chapter 5, if a blob is on the same location within the leg but its label changes, a few algorithms plot multiple trajectories at once and detect any stable regions indicating that the blob belongs to a shoe. Chapter 5 also notes the missing trajectory regions for the right-legged markers, which can impact the ability to carefully correspond the blobs to human features, such as the knee, ankle, toe, or heel. Lastly, Chapter 6 describes the outcomes of the project and the remaining tasks to include extracting step height and foot flexure from the blob positions and using the blob detector on other videos.

Chapter 2: Standard Computer Vision Feature Detectors

This chapter outlines some of OpenCV's image processing and feature detection methods aimed at analyzing the physical details of an image or video. Algorithms that quickly detect corners and other points of interest include FAST and ORB. Others, including the Canny edge detector and Hough line transform, would instead draw along the boundaries of some video elements. The FAST, ORB, Canny, and Hough methods meant to find but did not pay any more considerable attention to all the subject's markers. Instead, they look for objects with too many details and corners in the objects they are programmed to find.

Section 2.1: FAST

In 2006, Edward Rosten and Tom Drummond, who are researchers of engineering at Cambridge University, created the Features from Accelerated Segment Test (FAST) [15]. FAST is a corner detection algorithm that considers pixels along the keypoints for threshold analysis [16].

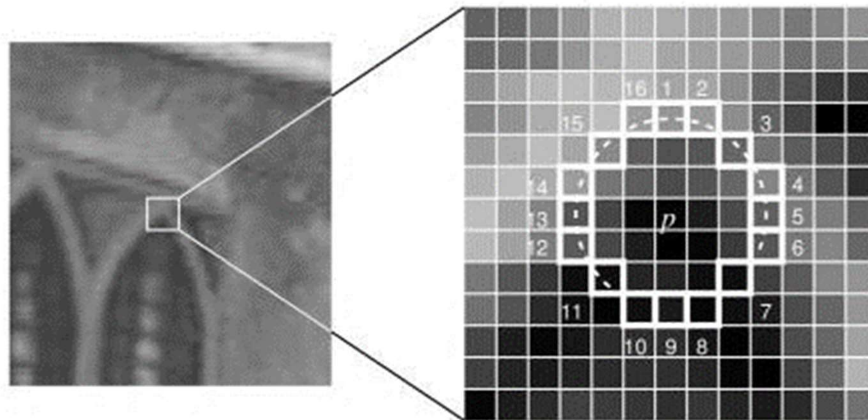


Figure 2.1: How FAST Works. The FAST algorithm is detecting a corner of a window and evaluates pixel *p* as either a corner or non-corner based on the brightness of 16 pixels around *p* [15].

In Figure 2.1, the algorithm considers a circle composed of 16 pixels (marked by white, highlighted squares) around corner candidate p. Let I_p be the intensity of pixel p, and t be the threshold value. Pixel p can be a corner if there are n continuous pixels in the circle that are brighter than $I_p + t$ or darker than $I_p - t$, as indicated by the white dashed lines in the figure. Let $n = 12$

because it allows for a high-speed test to discard a high number of non-corners to solely examine the following compass direction pixels: 1, 5, 9, and 13. For *p* to be a corner, at least three of the compass direction pixels must be brighter than $I_p + t$ or darker than $I_p - t$. FAST will then evaluate all pixels in the circle to apply to other corner candidates [15].

The FAST algorithm is essential for targeting and analyzing corners of features, which are small image patches that are independent of image scaling, rotation, and illumination changes. FAST is also successful at determining the status of corner candidate *p* so it can establish a keypoint, that is, the coordinate position where the feature has been detected.

Upon execution, FAST typically found over a hundred keypoints, or objects of interest, while only very few of them were the desired shoe markers. Even then, not all the markers were designated as keypoints (see Figure 2.2). In a Python program called TKinter_FAST.py, FAST interacts with TKinter, which is a standard Python interface to a GUI toolkit that opens a computer's file directory and enables the display of file types given in the source (e.g., jpeg and avi). TKinter also makes the program interactive by allowing the user to magnify regions of interest. The FAST algorithm will then apply to that region to calculate keypoint locations. Although it was one of the first steps toward detecting the desired video features, it returned an excessive number of keypoints. This situation can hinder the user's ability to know what the exact coordinates are for each desired shoe marker.

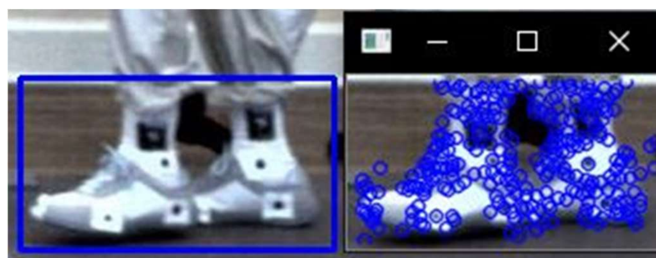


Figure 2.2: FAST Implementation. The FAST algorithm applies to a region of interest surrounding the shoes. A few out of the many keypoints detected the black shoe markers to be further analyzed.

Section 2.2: ORB

Along with the FAST detector, OpenCV offers BRIEF (Binary Robust Independent Elementary Features), which is a feature descriptor that is crucial for classifying and matching keypoints between frames to be discussed in Chapter 4. Another feature detector, ORB (Oriented FAST and Rotated BRIEF), combines the best aspects of FAST and BRIEF and adjusts them to enhance video analysis and improve keypoint detection performance [17]. First presented by Ethan Rublee et al. of the Willow Garage robotics company in 2011, ORB is rotation-invariant and noise-resistant while being more efficient and faster than other feature detectors [18].

This project's implementation of ORB can recognize a smaller number of video features and thus utilize fewer keypoints. In Figure 2.3, two ORB keypoints have successfully landed on the shoe markers. Selecting a region of interest via TKinter still helps with narrowing the number of keypoints down to the ones that are of interest to the user. Expanded that region would have helped target all the markers, yet the user may need to find ways to use the same number of keypoints as that of those spots.

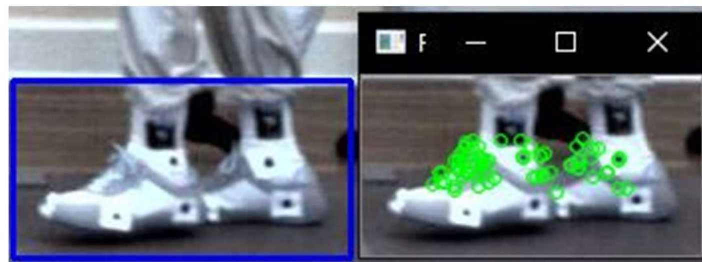


Figure 2.3: ORB Implementation. ORB improves video analysis with less image noise and rotation, thus reducing the keypoints, especially to such desired features as the black shoe markers.

Section 2.3: Canny Edges

In addition to the feature detectors, OpenCV provides the Canny edge detection algorithm for detecting a wide range of edges in an image. The algorithm goes through multiple stages to include reducing noise in the image, finding the intensity gradient of the image, removing pixels

that do not constitute an edge, and setting threshold values to classify edges [19]. Edges were considerable in attempting to detect shoe markers (see Figure 2.4). Moreover, they can shed light on the threshold and other factors that can impact the ability to draw keypoints along the boundaries of each spot.

Figure 2.4: Canny Edge Implementation. Canny edge detection helps to visualize the edges of each feature so the user can easily find the shoe markers.

Section 2.4: Hough Lines

Hough lines are comparable to Canny edges because they are detectable with any shape, even if it becomes distorted [20]. Figure 2.5 uses the probabilistic Hough transform, which is the optimization of a normal Hough transform that takes a randomized set of points suitable for line detection [20]. Although it is not a feature detector, the Hough line drawing method could be useful for asking questions about how far per second each subject could proceed on a walkway. The program, FindHoughLinesProb.py, holds the line-drawing algorithms that could be a beneficial source for generating and animating walking models to automate step height and angle measurements.

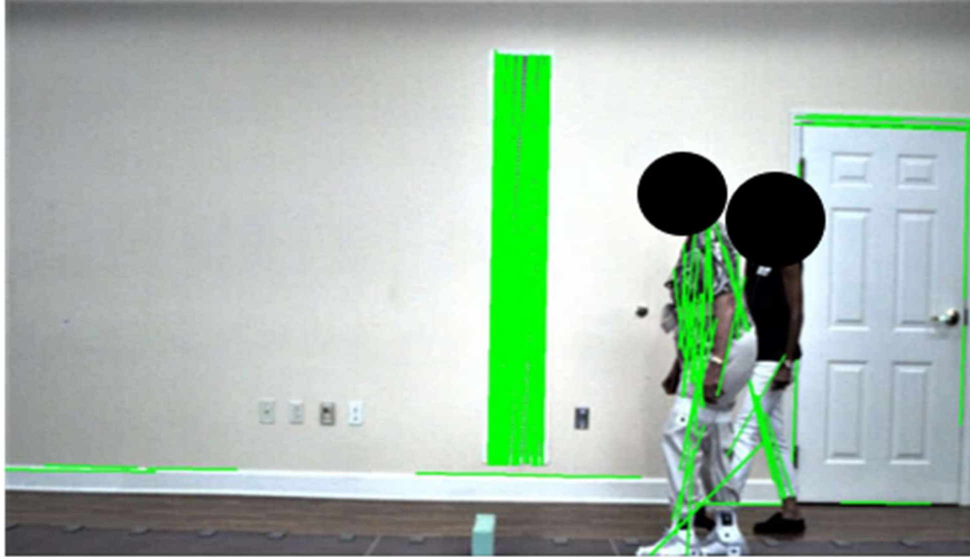


Figure 2.5: Hough Line Implementation. The probabilistic Hough transform paints straight lines on the subjects and the background to mark the edges of any object that it sees as essential.

Chapter 3: Properties & Behaviors of a Blob Detector

The objectives of this chapter include understanding what a blob is and how it can help with the detection of desired video features. One should also know how to choose among the five parameters—area, circularity, threshold, inertia, and convexity—that can impact the way a blob detector works to pinpoint desired features. Setting those parameters at appropriate values for the gait video can assist with increasing awareness of the subject's markers to be followed, as well as making the blobs located elsewhere less critical.

Section 3.1: Definition and Characterization of Blobs

The FAST, ORB, Canny Edge, and Hough Line detector algorithms did not adequately show the quantitative detection of all the black markers on a subject's shoes and knees. However, they served as important precursors to a method that precisely targets those spots based on their color, shape, and size. Blob detection is the method that achieves this task. A blob is a region of similar color or light that is localized within an image or video. Characterizing a blob requires an algorithm under a class called SimpleBlobDetector to extract the blob from an image [21]. First, the program, FindBlobs_OriginalParams.py, sets up the default parameters for SimpleBlobDetector described by

```
params = cv2.SimpleBlobDetector_Params()
```

Next, it reads five parameters associated with blob detection (see Figure 3.1): (1) threshold, which is the color intensity value of the pixel that can apply to grayscale images and can range between 0 and 255; (2) area, which is the number of pixels that constitute an element; (3) circularity, which is a measurement of how close to a perfect circle an element is; (4) convexity, which is a measurement of whether the element is convex or concave; and (5) inertia, which is a measurement of how elongated or elliptic an element is [22].

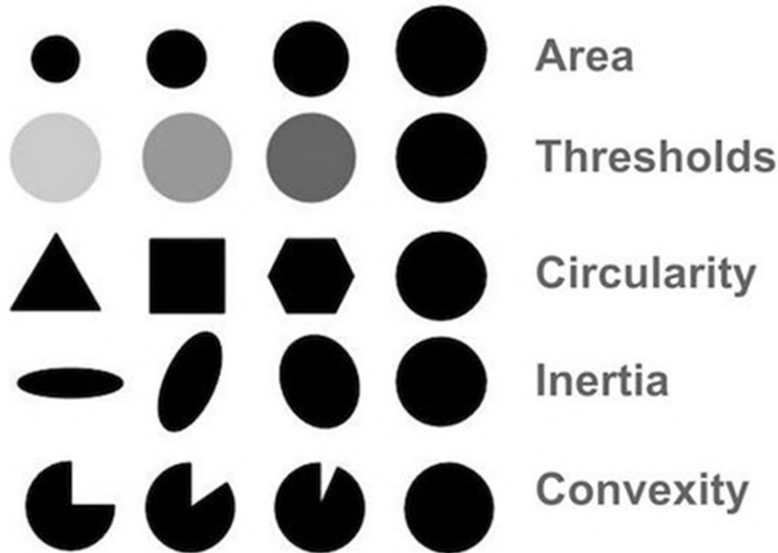


Figure 3.1: Visual Definitions of Blob Parameters. [22] uses visuals to define the blob detector parameters above. Note that inertia and convexity were neglected for this project.

Both threshold and area have minimum and maximum values that consider the number of pixels of different thresholds to determine the presence of a blob. Circularity, convexity, and inertia all carry only one value describing the variations of a typical shape, such as a circle. Since the spots on a subject's shoes and knees are all circular, the user can select and evaluate the values of circularity along with the area and threshold ranges. Inertia and convexity were turned off in all Python programs concerning blob detection because they do not support the identification of blobs whose roundness matters in collecting accurate gait data.

Section 3.2: Choice of Parameters

To consider desired image features as blobs, the user needs to choose detector parameters along with their values appropriate for those features with similar colors and shapes. Changing only one parameter while leaving others with default values will cause the image to display a doorknob, electrical outlets, a head of hair, and other objects as if they are blobs (see Figure 3.2). Setting any parameter closer to zero can lead to randomized blobs—large and small—that detect any shape with a stable or unstable structure.

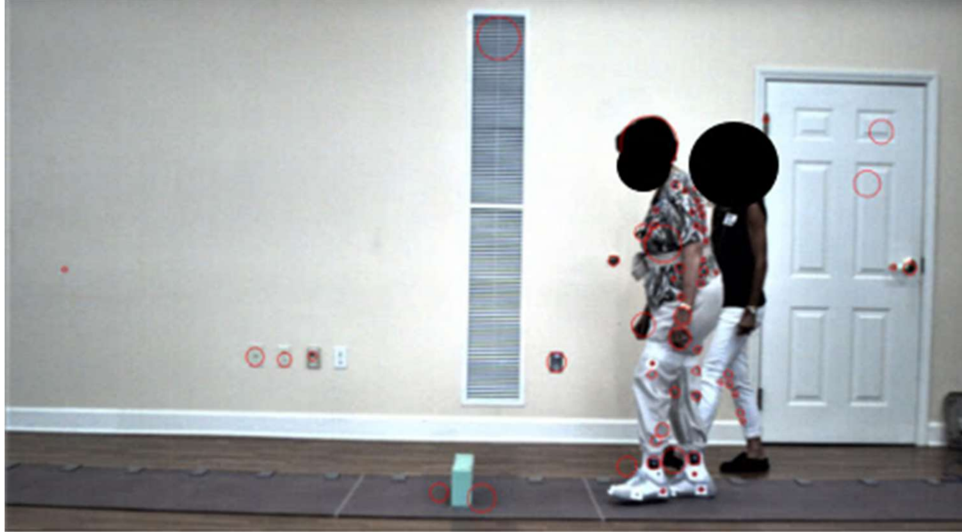


Figure 3.2: Poor Blob Detection. A poor selection of parameters and their values can ruin the blob detector's performance and produce keypoints other than those on the subject's knees and shoes.

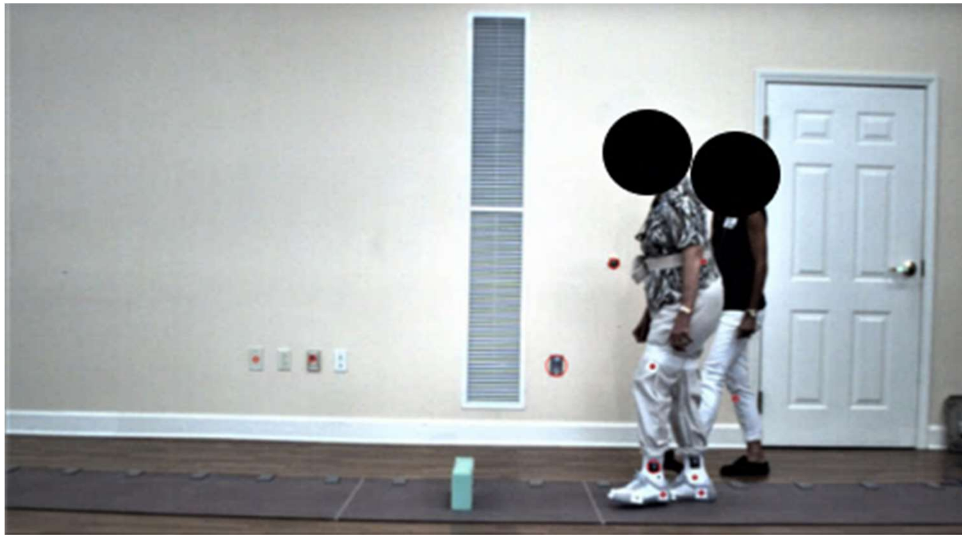


Figure 3.3: Blob Detection with Default Parameter Values. If the blob detector performs with default parameters, this could help the user with selecting the parameters that are important for targeting the subject's markers.

Selecting blob detector parameters begins with understanding the default values for three out of the five variables. The threshold was 50-220, the area was 25-5000, and the circularity was about 0.8 (see Figure 3.3). From there, the changes in area and circularity follow a similar trend as increasing their values results in fewer blobs in the image; decreasing them results in more blobs. Figures 3.4 and 3.5 display sensitivity graphs stressing the impact of minimum area and circularity on blob presence. The graph in Figure 3.4 suggests that a small value of the minimum area is

essential for detecting even the smallest markers on the subject that otherwise go unnoticed. To explain Figure 3.5, a high value for circularity is appropriate for targeting the subject's markers, which resemble hand-drawn, non-perfect circles.

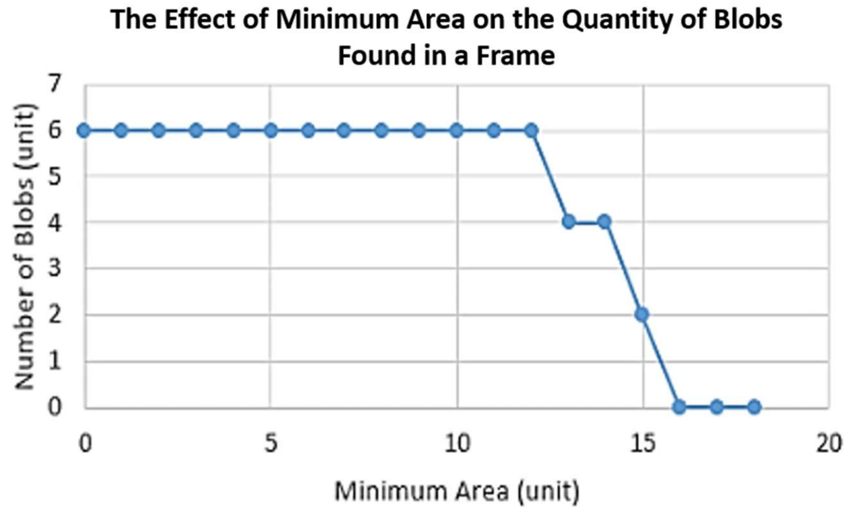


Figure 3.4: Sensitivity Chart for Minimum Area. Increasing the value for the minimum area creates a sensitivity graph for the total number of desired blobs. The maximum number for detecting all those blobs, in this case, was 12, and it was subject to change.

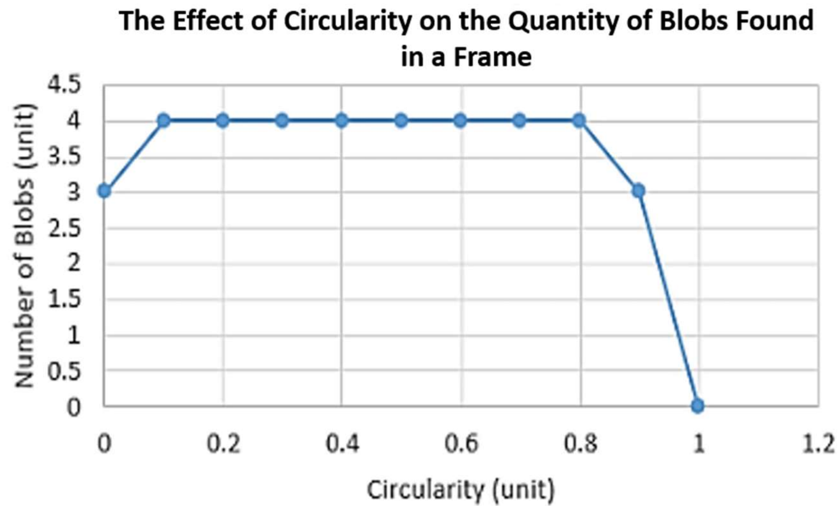


Figure 3.5: Sensitivity Chart for Circularity. Increasing the value for circularity creates a sensitivity graph for the total number of desired blobs. The maximum number for detecting all those blobs, in this case, was 0.83, and it was subject to change.

The user has the option to label detected keypoints with coordinate information. After drawing them as red circles, the Python algorithm below stores rounded values for the x- and y-coordinates of each blob before giving it a number label. Those labels are necessary for

distinguishing parameter information among multiple blobs and for determining the human features that each blob is associated with.

```
im_with_keypoints = cv2.drawKeypoints(frame, keypoints, np.array([]),
      (0,0,255), cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
for k in range(0,len(keypoints)):
      cx = round(keypoints[k].pt[0])
      cy = round(keypoints[k].pt[1])
      cv2.putText(im_with_keypoints, str(k), (cx, cy),
cv2.FONT_HERSHEY_SIMPLEX, .6,(0, 0, 255))
```

Additionally, the blob detector can assist with multiple thresholds of keypoints that change the area of the spots, which can also impact how circular the spots are. Therefore, two experiments were performed to discover the behavior of the detector when the user sets the threshold step to 10 and turns off the circularity.

A program called MultipleThresholds_Step10.py produces expanded two keypoints (namely Keypoints 1 and 4) in a specific frame with the application of threshold ranges with a threshold step of 10. It allows the user to observe how much the area of the blob is changing as the threshold increases. The pixel threshold matrices next to the keypoints help conclude that the threshold range can dictate the ability to locate blobs based on the number of black pixels that make up the area (see Figure 3.6).

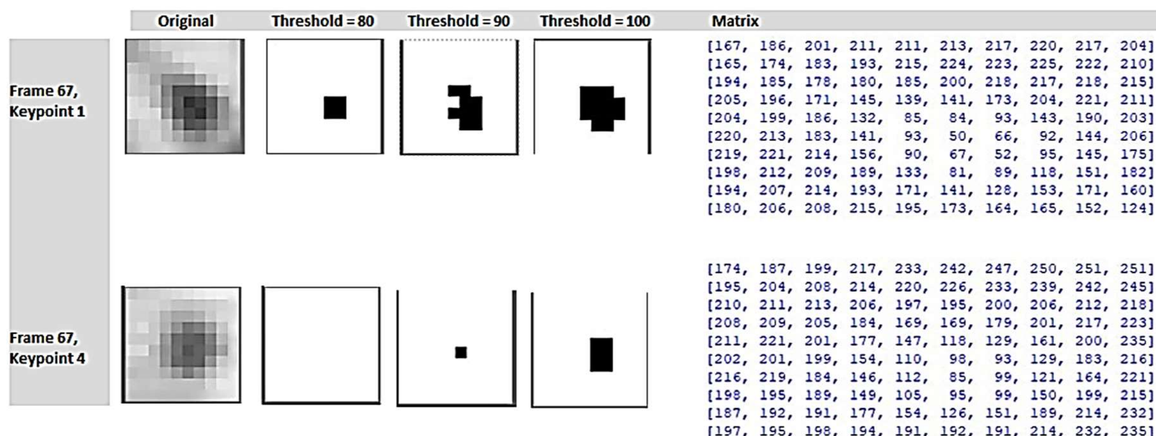


Figure 3.6: Keypoint Analysis with Threshold Step. MultipleThresholds_Step10.py presents the analysis of zoomed-in, pixelated blobs with binary threshold steps of 10. This procedure demonstrates how smaller threshold increases can reveal the best blob area and circularity to detect.

In another program called MultipleThresholds_CircOff.py, the user turns off the circularity of the blobs by setting it to False. That program expands and applies thresholds for two keypoints in a specific frame. It also allows for comparisons between a non-circular blob (namely Keypoint 1) and a circular blob (namely Keypoint 16) (see Figure 3.7). In a later experiment, Keypoint 16 was not detected despite being circular and having a smaller area. A noticeable difference between the two blobs is that Keypoint 1 has a darker center in the middle, which allows for more robust detection performance. Given this, the user should set the circularity back to True to detect blobs within a specific range of thresholds.

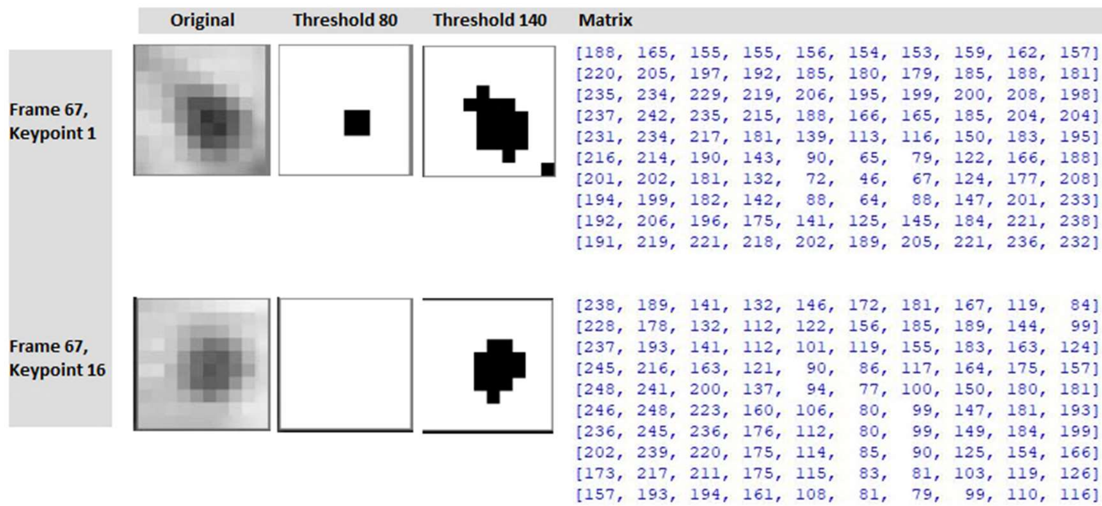


Figure 3.7: Keypoint Analysis with Lack of Circularity. MultipleThresholds_CircOff.py presents the analysis of zoomed-in, pixelated blobs with a lack of circularity. This procedure demonstrates how the detection of circular and non-circular blobs should be supported with threshold ranges to be used later in the project.

Today, the project uses the following parameters to assist with identifying and following the desired blobs on the subject: (1) the threshold ranges from 40-200 with a threshold step of 20, (2) the area ranges from 8-40, and (3) the circularity is equal to 0.85 (see Figure 3.8). With these values, the average number of detected markers per frame is 6.33, while the average number of blobs located away from the subject's legs is 1.72 per frame. With an average of 8.05 total blobs per frame, the blob detector can find, on average, 78.63 percent of the desired blobs per frame. In

this case, the current blob parameters allow for the flexibility of detecting the best blobs and the greater importance of following them frame by frame based on changes in coordinate positions.

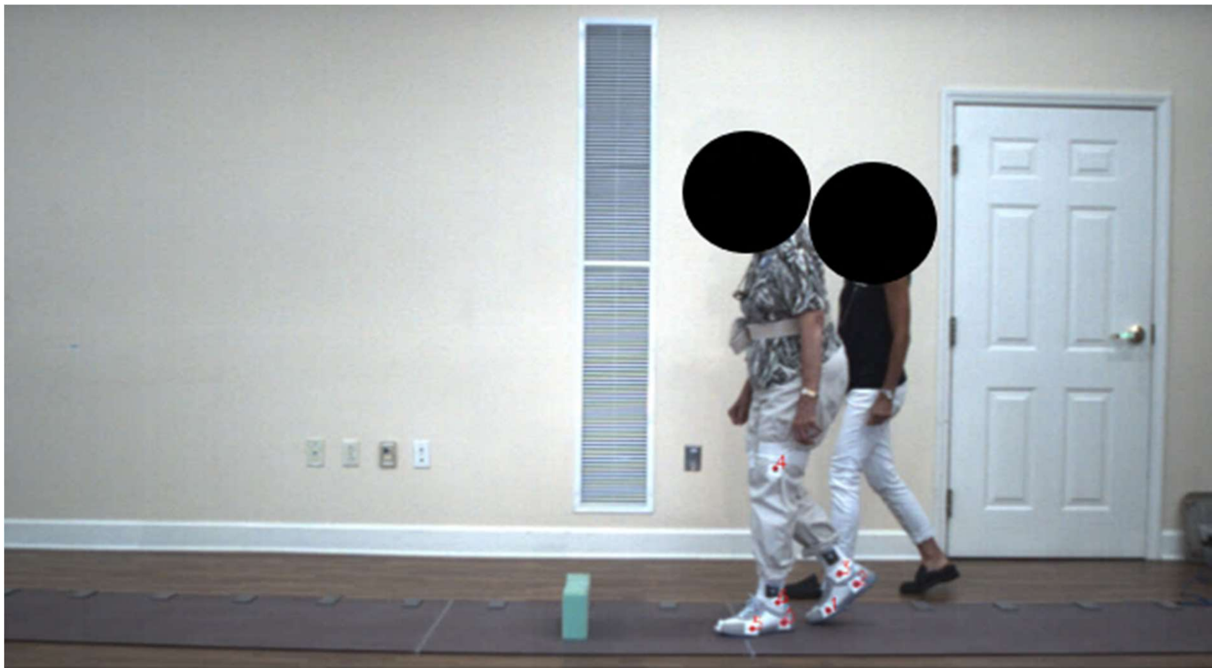


Figure 3.8: Blob Detection with Current Parameter Values. The current parameter values precisely pinpoint the subject's blobs that the user would like to identify and follow to establish trajectories.

Chapter 4: Correlation of Blobs Between Frames

This chapter provides the definitions of descriptors and descriptor matchers to help understand how the computer vision software works to classify and match blobs between frames. This project performs different matching procedures for two groups of blobs: those that were readily identified and those that remain unidentified. The usage of new keypoint labels allows for the creation of trajectories that measure (x, y)-coordinate position changes throughout the video.

Section 4.1: Blob Classification

After this project initializes the blob detector as a subroutine and sets up other video-related variables for displaying the images, it proceeds to classify and match keypoints with similar coordinates between two frames. OpenCV contains a framework for two-dimensional features called xfeatures2d and a class reference for creating BRIEF descriptor generators described by:

```
descriptor = cv2.xfeatures2d.BriefDescriptorExtractor_create()
```

Descriptors, in general, are characteristics of the keypoints that are numerical and point to the area of the image that the keypoint references. They usually describe the shape of a detected object, but they are insensitive to rotations and scale changes. BRIEF, which was previously mentioned in Chapter 2, is a feature descriptor that provides a shortcut to finding binary strings. It reads an image to carefully select a set of *n* (x, y) location pairs, compare those coordinates for pixel intensity and obtain an *n*-dimensional bitstring. Once a match has been made, BRIEF will use the Hamming distance to show that the location pairs are at least close to each other [23]. Hamming distance is defined as the number of different bits in corresponding positions in two bitstrings. For example, the distance between 011<u>1</u>0 and 011<u>0</u>0 is 1, and the distance between 10<u>1</u>0<u>0</u> and 10<u>0</u>0<u>1</u> is 2 [24].

Another object along with the BRIEF descriptor is the Brute-Force descriptor matcher, or BFMatcher, which finds one of the closest secondary descriptors by trying each one to create a match for each primary descriptor.

```
bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
```

The parameter cv2.NORM_HAMMING is the Hamming distance because the Python program is using the BRIEF descriptor in addition to the blob detector, and *crossCheck* is set to true if the BFMatcher will return the coordinate pairs that are consistent between multiple frames [25].

A program named VideoBlobsIdentifiedFinal.py first initializes the blob detector, BRIEF descriptor, brute-force matcher, and empty keypoint and descriptor lists. Afterward, it can start reading the frames to extract the keypoints for previous frame j and current frame $j + 1$ from a list of keypoints. Also, the program will use the BFMatcher to compute the descriptors for each blob found in the frame. Essentially, it proceeds to create a blob name list and a list of lists for keeping track of the blob identification numbers per frame. The next blob name to use depends on the current length of an array of keypoints. Based on the list of blobs per frame, the program intends to list the frames where a blob at the current frame $j + 1$ was last found. Suppose frame $j + 1 = 86$, where all desired blobs have been detected and are newly numbered after the program tries to match them to those found in frame $j = 85$ (see Figure 4.1b). This new way of numbering is different from the original, where those same blobs were not carrying the same labels then (see Figure 4.1a). Keeping evidence of the blobs' names and locations thus plays a significant role in giving the BFMatcher the ability to match most keypoints if they are slightly moving from one coordinate position to another.

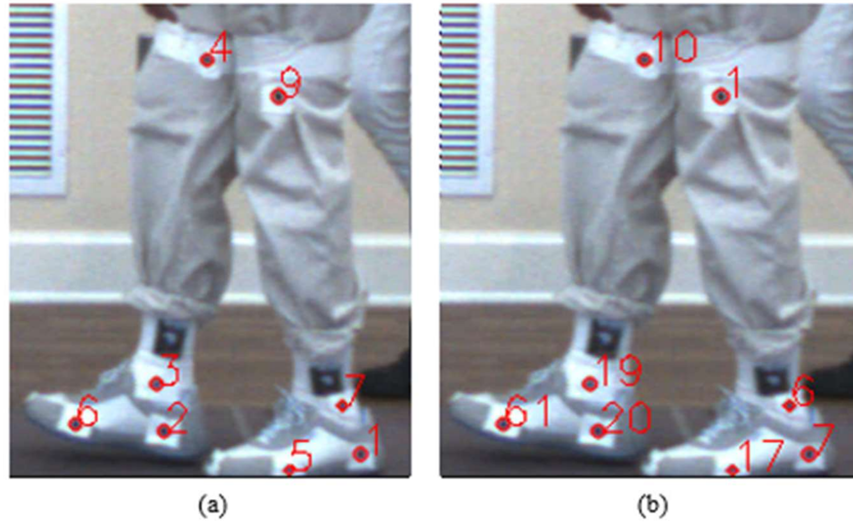


Figure 4.1: Initial Versus New Blob Labeling. (a) The initial labeling of blobs deals with numbers ranging up to the last keypoint index value in Frame 86. (b) A new way of numbering the blobs involves matching blobs to those found in the previous Frame 85.

Section 4.2: Descriptor Matching

Section 4.2.1: Blobs Already Identified

VideoBlobsIdentifiedFinal.py starts with retrieving keypoints for previous frame j from a list of keypoints and comparing them to those found in current frame $j + 1$ under the BFMatcher. It then builds a list of unmatched blobs and that of blob names in the current frame. An example of an unmatched blob to be included in the current blob list is Blob #10, which may appear in frame 86 if it was not last found in frame 85 but rather in frame 83 (see Figure 4.2). A matching test exists to compare the distance of keypoint descriptors to a user-specified distance threshold. The program will pass the matching test if the descriptor distances in coordinates are less than the pixel threshold values. Following that result, it can generate raw and corrected matches of blob descriptors between two frames while keeping track of the unmatched blobs and current blob lists. This procedure is increasingly useful for identifying blobs that were left unnoticed so the user can observe that a specific blob manages to move with the same label for most of the video.

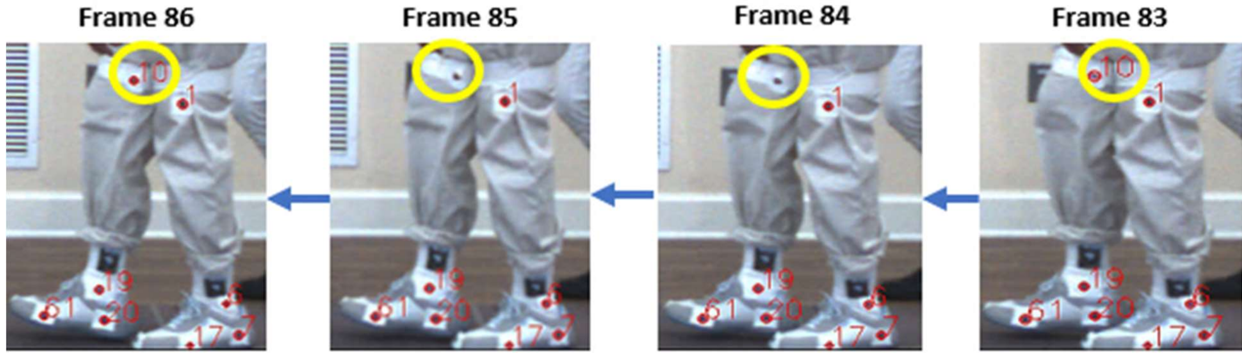


Figure 4.2: Tracking an Unidentified Blob. This visual representation of tracking Blob #10 shows that it went unnoticed two frames before it reappeared in Frame 86. Blob #10 can be included in the list of blobs in frame 86 if the program can match it with Frame 83.

Section 4.2.2: Blobs Still Unidentified

For any blob still not identified, the program gathers the frames where it is located and discards the current and previous ones. While the unmatched blobs and the most recently checked frames are available for further investigation, the program will carry the evidence of blobs to identify and frames to test with the most recent frame being used first. From there, it can match the descriptors between a current frame and a frame to check before it passes another match test. The BFMatcher will compute new matches with blobs still unidentified so the program can update for every current frame a list of blob identification numbers and a list of each blob's frame history. A blob without any matches will be discarded. If there are no more unmatched blobs to identify and no more frames to check, then the program can append the unmatched blobs to the current frame and renumber them.

Section 4.2.3: The Blob Renumbering Process

Suppose all the blobs have been renumbered within the first 170 frames. In this case, the list of blob names will hold 131 distinct numbers. Some of those names are not independent of each other because at least two of them belong to one of the subject's eight markers. For example, the video shows that Blob #7 and Blob #79 both represent the same blob within the left shoe. In an ideal scenario, if the program exactly matches all eight markers between frames, then the blob

naming list would not have more than eight numbers. Currently, 116 of the names in this project represent the least essential blobs, and 15 are placed onto the legs. About 87 percent of those desired names cooperate as two or more labels for the same marker (see Table 4.1). Although multiple names can help continue with following a blob throughout the walk, they can lead to such issues as misidentified blobs and multiple placements of a single blob label, which are discussed further in the next section.

Marker Location	Blob Name(s)	Quantity of Names on Marker
Left Heel	7, 79	2
Left Toe	5, 17, 87	3
Left Ankle	6, 83	2
Left Knee	1	1
Right Heel	20	1
Right Toe	45, 61	2
Right Ankle	19, 41	2
Right Knee	10, 76	2

Table 4.1: Marker Name Placement. Fifteen desired blob names are assigned physical marker locations on the subject.

Section 4.3: Plotting Blob Trajectories

A trajectory is usually defined as a path an object follows as it moves through space. An example of a trajectory can be a path an airplane takes as it flies into the open sky. In the context of this project, VideoBlobsIdentifiedFinal.py has each of the subject's eight blobs relocate to different coordinate positions throughout the video to create paths of walking motion. The resulting paths come as two separate plots: (1) one plot involving a blob's locations along the x-axis per frame, and (2) another involving that same blob's locations along the y-axis per frame. Both plots are imperative to analyze in that the user reviews the video to see whether the blob displayed in the frames is the same as the blob that leaves behind coordinate-based trajectories. Note that the y-axis of the video is measured from top to bottom, and the x-axis is measured from left to right (see Figure 4.3).

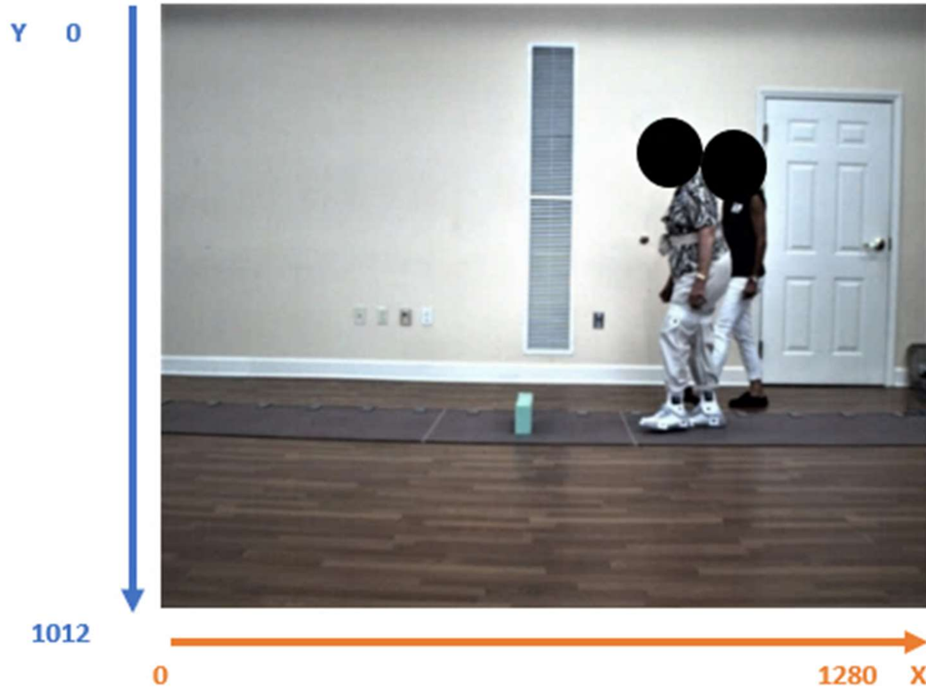


Figure 4.3: Video Axis Measurements. The way the video axes are measured will affect how the blobs' trajectories will appear along the axes.

VideoBlobsIdentifiedFinal.py contains an algorithm that records information about the blobs' (x, y)-coordinates with corresponding frames. Suppose one of those blobs is labeled #7, and the video is 240 frames long. If Blob #7 is present in any of the first 170 frames this project analyzes, the algorithm will extract a keypoint for that frame and match it with the location of Blob #7 within each list of blobs per frame. The frame numbers and (x, y)-coordinates for Blob #7 can then be stored into arrays that help plot trajectories over time. The plots with flat regions between frames 60 and 90 and steep regions between frames 30 and 50 indicate that Blob #7 belongs to the left shoe, particularly at the heel, which is stationary in between the swing stages of walking activity (see Figure 4.4).

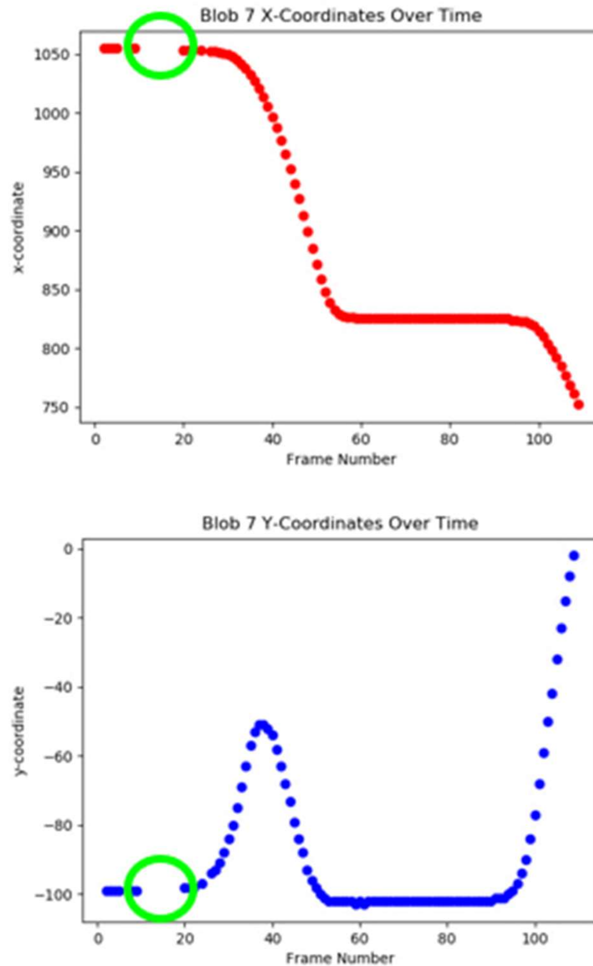


Figure 4.4: First Trajectory Example. The x- and y-coordinate trajectories for Blob #7 represent a shoe that is stationary between frames 60 and 90, as indicated by a flat trajectory region. Note that the points in Frames 10 through 19 are missing (circled) because the blob detector was unable to find the left heel blob even after some success at the beginning of the video.

Section 4.3.1: Missing Trajectory Points

Trajectories, including the one in Figure 4.4, may have points that are missing because a blob did not appear in one or more frames. If the missing trajectory region is small, the user can write an algorithm that draws a line determined by the mathematical distance between two points (see Figure 4.5). In contrast, if a trajectory is missing several points, it may not provide sufficient information for whether a blob is worth following during walking activity. In that regard, blobs that are not detected imply the need for unsupervised video analysis, which can help fix data-related flaws in the marker tracking process.

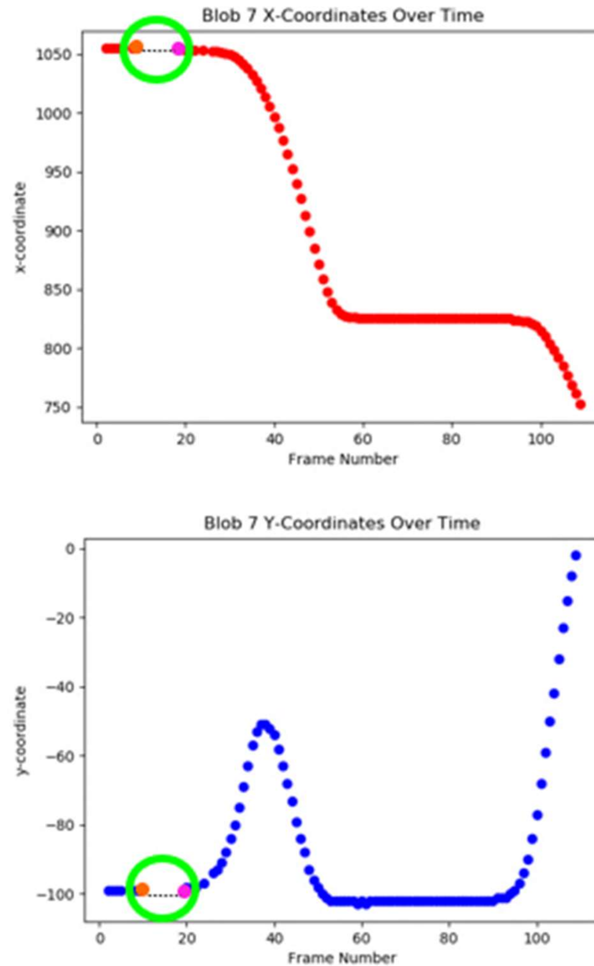


Figure 4.5: How to Fill in Trajectory Gaps. A new algorithm should be able to compute the mathematical distance between the last point before the missing region (orange) and the first point after it (magenta). That distance should be used to determine the length of the straight line (black and dashed), resolving the error of blob non-detection.

Section 4.3.2: Blob Misidentification

The discontinuities in any trajectory stress that a blob is misidentified. Figure 4.6, for instance, holds many misidentified points from the beginning of the video to Frame 45, where the x-coordinate for Blob #17 makes a sudden jump from a flat trajectory region. A numerical label that changed from 5 to 17 when Blob #17 switches marker positions at Frame 45 is the primary evidence supporting that error (see Figure 4.7). In searching for discontinuities, an algorithm could find the slope of the trajectories that could be impacted by the renumbering of blobs.

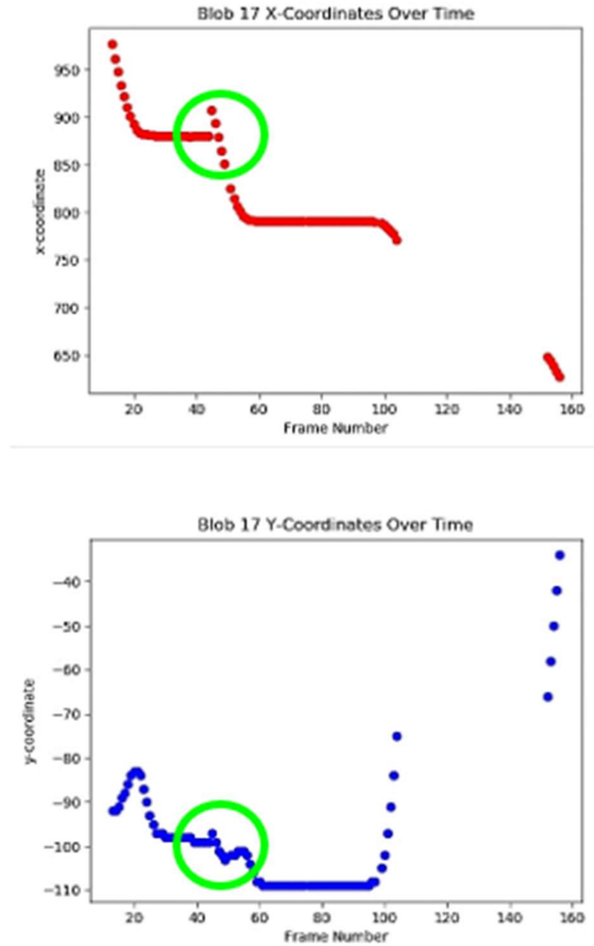


Figure 4.6: Spotting Trajectory Discontinuities. The discontinuities in the x- and y-coordinate trajectories for Blob #17 (circled) resulted from the renumbering and misidentification of blobs between frames.

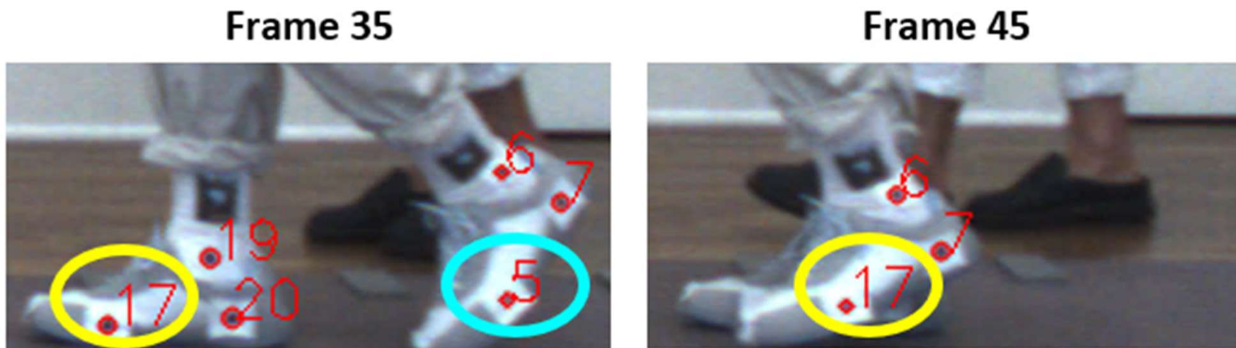


Figure 4.7: Video Proof for Discontinuities. To prove that discontinuities in the trajectory occurred in Figure 4.5, Frames 35 and 45 demonstrate a label change for the left toe marker. That spot started as Blob #5 (blue circle); by Frame 45, Blob #17 (yellow circle) suddenly leaped from the right toe to the left toe to renumber the marker. This event means that Blob #17 was misidentified on the right shoe in the earlier frames.

Chapter 5: Association of Blobs with Human Features

In this chapter, an algorithm attempts to plot the blob with different names to make extended trajectories of the toe, heel, ankle, or knee. The best trajectories of this project are associated with the shoe, which creates stable regions in between steps. Relabeling the blobs with human features can be useful if a program can mathematically detect the stable regions and acknowledge the differences between the left- and right-legged markers.

Section 5.1: Combining Multiple Trajectories

To review, the trajectory-drawing algorithm in VideoBlobsIdentifiedFinal.py checks for blobs with user-specified labels in each frame so it can plot their migrations between coordinate positions throughout the video. The user can also specify two blobs with labels differing from each other, and the algorithm can simultaneously follow each blob to plot separate trajectories in one graph. This situation allows for the combination of more than one blob to display a complete trajectory of a blob located on the subject's toe, heel, knee, or ankle.

As discussed in Chapter 4, Blob #7 is associated with the left heel, but the user wants to analyze another blob to produce a full trajectory of that human feature. Suppose that the second blob is labeled #79, which appeared at the heel after Blob #7 last appeared at frame 109 (see Figure 5.1).

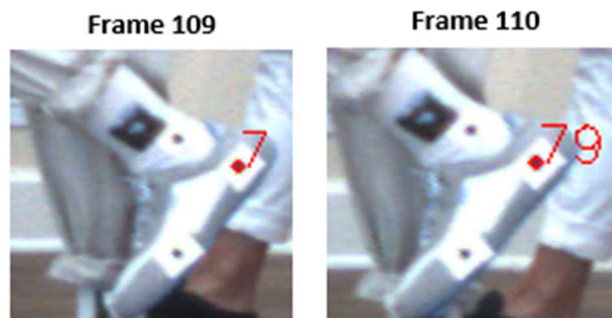


Figure 5.1: Two Names for One Marker. Blob #7 becomes Blob #79, which leads to the end of its trajectory in one plot and lets a different label take over the rest of the trajectory in another plot.

Comparing y-coordinate trajectories for Blobs #7 and #79 urges an algorithm to recognize that the peak of the walking path was split in half while the subject steps over the foam block (see Figure 5.2).

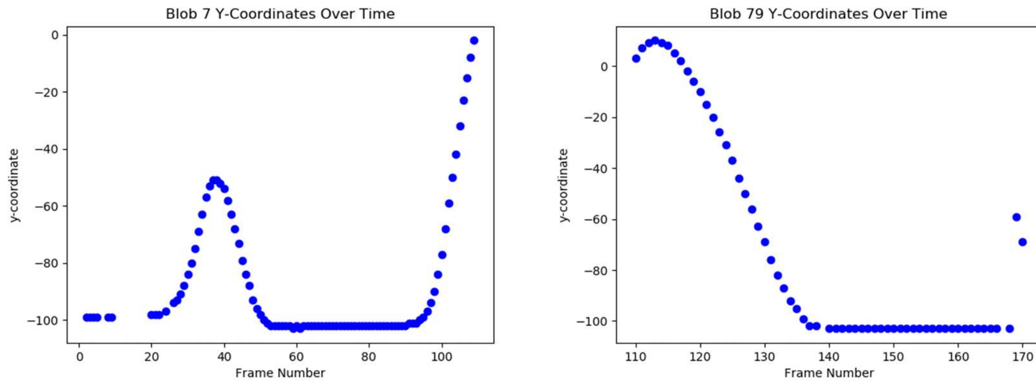


Figure 5.2: Trajectories for Two Names. The changes in blob labels have split the left heel trajectory near the peak of the walking motion. An algorithm is needed to handle the plotting of both blobs to piece the trajectories together.

The algorithm sees that Blob #7 was reaching toward the peak of the highest step before it disappeared, and Blob #79 appeared late to finish the rest of that step. Those scenarios help that algorithm ensure that none of the frames have both blobs, which in turn can be suitable for combining their trajectories. Also, the ending point of Blob #7 and the starting point of Blob #79 happened to share similar coordinates given that no other blob interferes with the last few frames for Blob #7 and the first few frames for Blob #79. If the algorithm sees this type of blob correlation between frames, it can proceed with merging the two trajectories to establish a complete path of the left heel marker (see Figure 5.3).

While x-coordinate trajectories move downward as the physical blob moves toward the left, y-coordinate trajectories provide an accurate measure of the distance each blob makes from the ground so long as it is near the bottom of the shoe. The changes in y-coordinate positions are also useful for differentiating shoes from the knees as the knee trajectories, such as those in Figure 5.4, represent the markers that are located between the thigh and the calf and are slightly moving during stationary leg activity. This situation suggests that, although they produce well-defined

trajectories, the knee markers do not establish as many stable regions as the shoe markers, which are much closer to the ground according to the video's y-coordinate scale (see Figure 4.2). A final implication for combining blobs is that, after the detection of any stable regions, the relative coordinates say where the blobs are located within the subject.

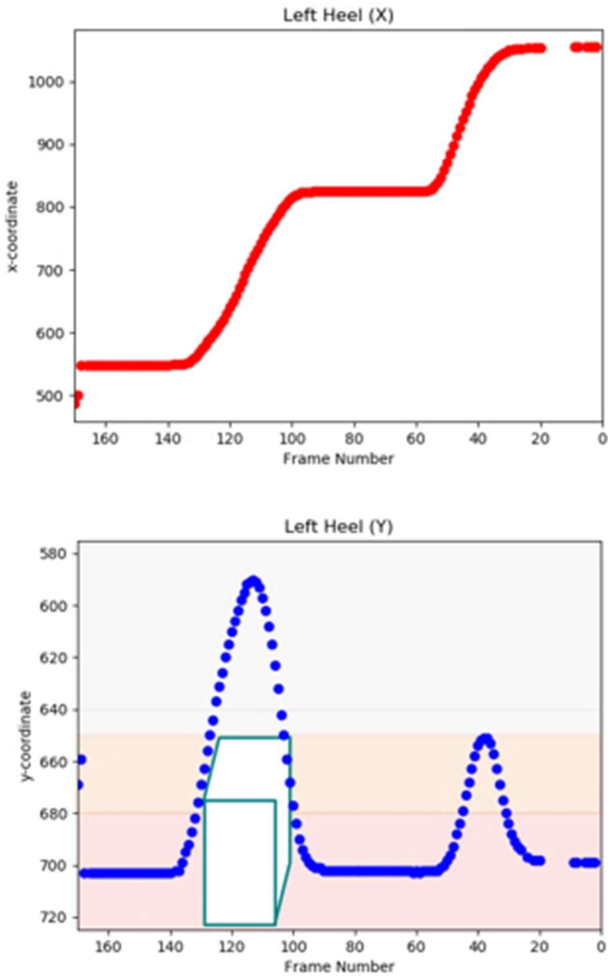


Figure 5.3: Combined Left Heel Trajectory. A y-coordinate trajectory shows the left heel rising and falling during the steps and staying put between them. The rise of the heel is highest when the subject steps over the block.

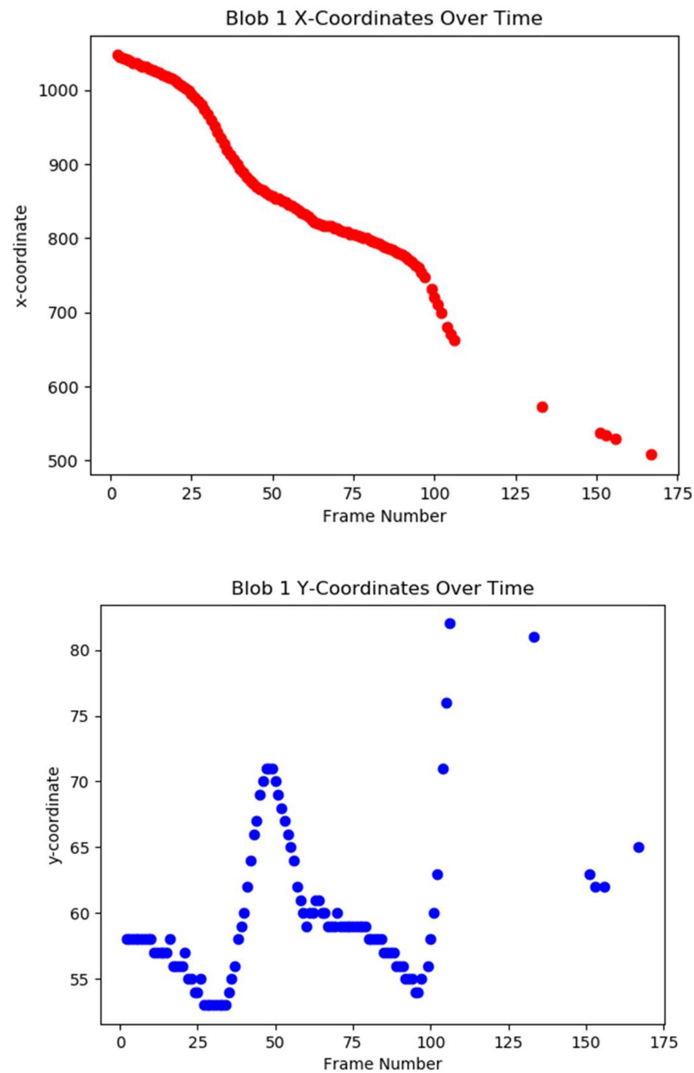


Figure 5.4: Left Knee Trajectory. The knee trajectories are different from the shoe trajectories because the knee is nowhere near the ground but is slightly flexing while the leg is standing upright between frames 0 and 25 and between frames 60 and 95.

Section 5.1.1: Blobs Appearing Twice in One Frame

In trajectories such as those in Figure 5.5, a blob can be found in at least two different places within a single frame. The name of that blob is also misidentified as if it appeared twice simultaneously. To resolve this error, the user would create an algorithm that keeps track of the frequency of a blob's appearance per frame. That program would then need to make sure that no frame has a blob appearing more than once. This proposed solution helps improve blob

identification further by avoiding the combination of trajectories that causes twice-appearing blobs per frame.

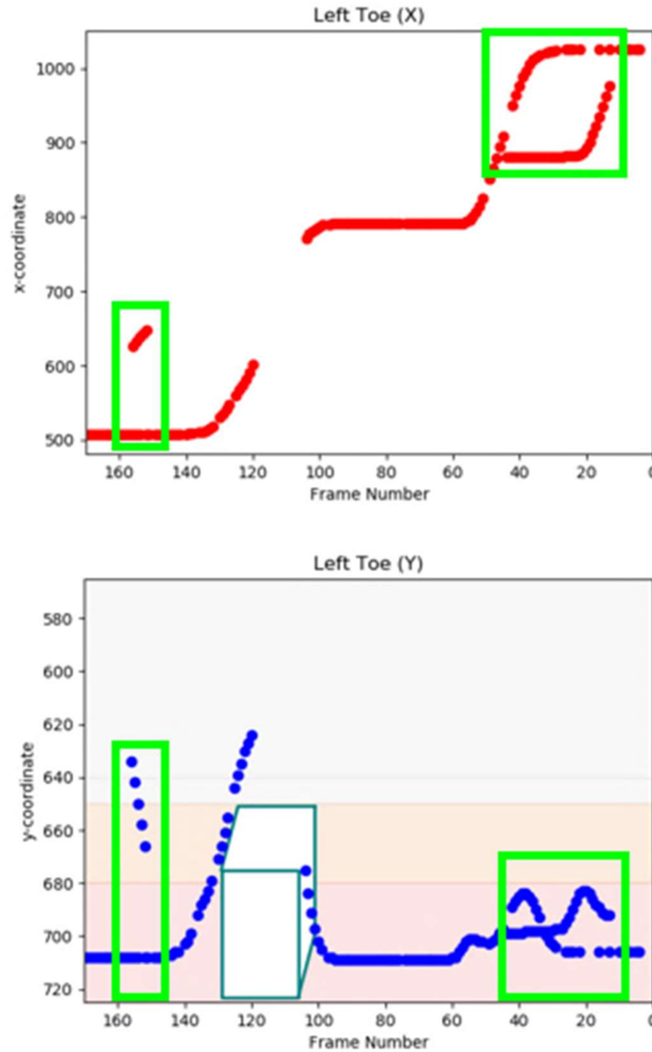


Figure 5.5: Twice-Appearing Blobs at the Left Toe. The left toe blob is a good example for illustrating twice-appearing blobs in many single frames (boxed). Three blobs were used to produce the above plots, which resulted in discontinuity and misidentification problems among blob names.

Section 5.2: Stability of Trajectory Regions

The best way to associate a blob with the subject's shoes is to find a region of its trajectory that is stable in between steps. Returning to the heel trajectory at Figure 5.3, the user can determine the start and end frames that constitute a region indicating that the shoe was planted firmly on the ground. In VideoBlobsIdentifiedFinal.py, another algorithm exists to calculate the standard

deviation of the (x, y)-coordinates for every ten frames within the entire trajectory. It then establishes a condition for which a frame belongs to the stable region should the standard deviation be less than or equal to 0.1. For instance, the heel trajectory enters the stable region at frame 62 when the subject finishes the first step. It will leave that region at frame 81 as the subject is about to start another step.

The significance of automating the detection of stable trajectory regions with standard deviations is that the blob should not make significant shifts between coordinates if it is considered a toe, heel, or ankle. Furthermore, its association with some human feature depends on its movement over time; no movement usually means the blob associates well with the shoe. This notion is also true when the heel and toe blobs create a stable trajectory region within the same frame interval because both are right next to the floor. A horizontal distance between them based on their coordinate locations is beneficial for which the user can argue that the shoes are best for determining whether a blob makes little to no movement against the floor.

Section 5.3: Left-Side Versus Right-Side Markers

The video is showing the left side of the subject, causing the blobs on the right leg to hide often as opposed to those on the left leg. This situation can lead to missing regions among the trajectories created for the right-sided blobs, regardless of their association with human features. Blob #20 is one of those blobs that is located on the right heel and stays in its current position even when the left leg covers it (see Figures 5.6-5.8). The ability for it to remain in its physical location requires that the blob detector adequately knows the size, shape, and color properties unique to Blob #20. This concept also applies to Blob #1, which is a left knee blob that is not difficult for the user to follow (see Figure 5.4). In the case of right-sided blobs, the detector should recognize that they are hiding often but reappear as if their characteristics are not significantly affected by

the coverage. If not, it will renumber the blobs, and the user will have to spend time piecing together multiple trajectories to claim that those blobs belong to any right-legged human feature.

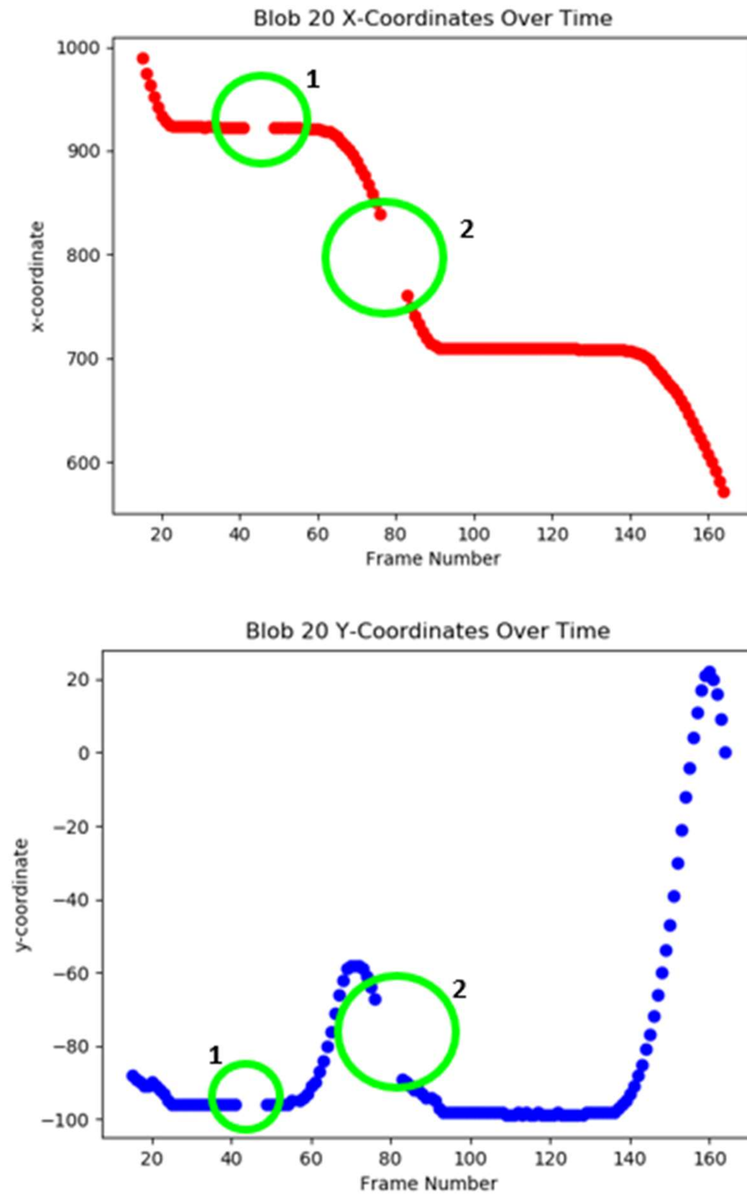


Figure 5.6: Right Heel Trajectory. There are two missing regions that have been labeled to track how many times the right heel was hiding while creating an effective trajectory.

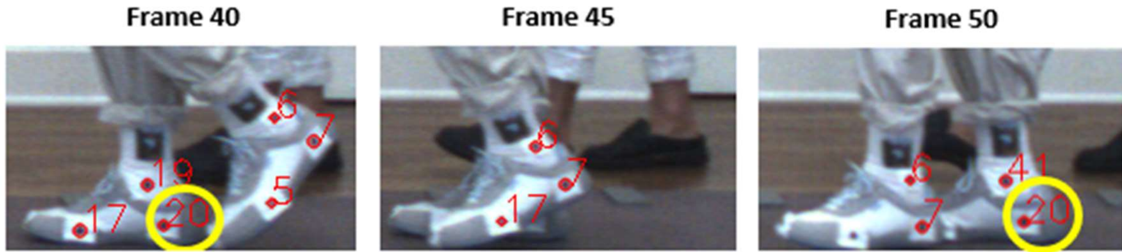


Figure 5.7: First Hiding Event. Frame 40 shows Blob #20 (circled) before the left shoe hid it during Frame 45, making this the first occurrence of blob hiding within the right heel trajectory. Frame 50 then shows Blob #20 in the same position.

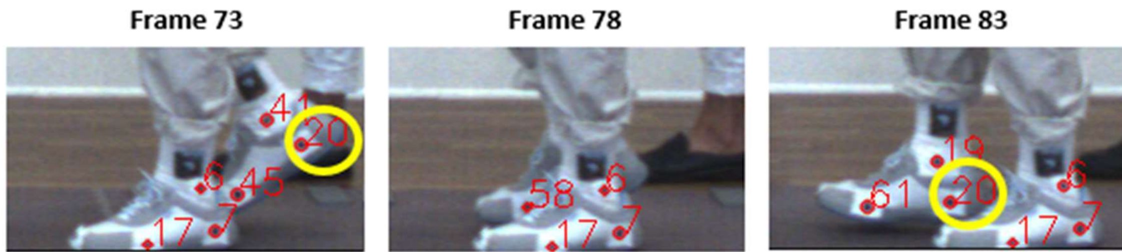


Figure 5.8: Second Hiding Event. Frame 73 shows Blob #20 (circled) before the left shoe hid it during Frame 78, making this the second occurrence of blob hiding within the right heel trajectory. Frame 83 then shows Blob #20 in the same position.

Chapter 6: Project Conclusions

Section 6.1: Overall Review

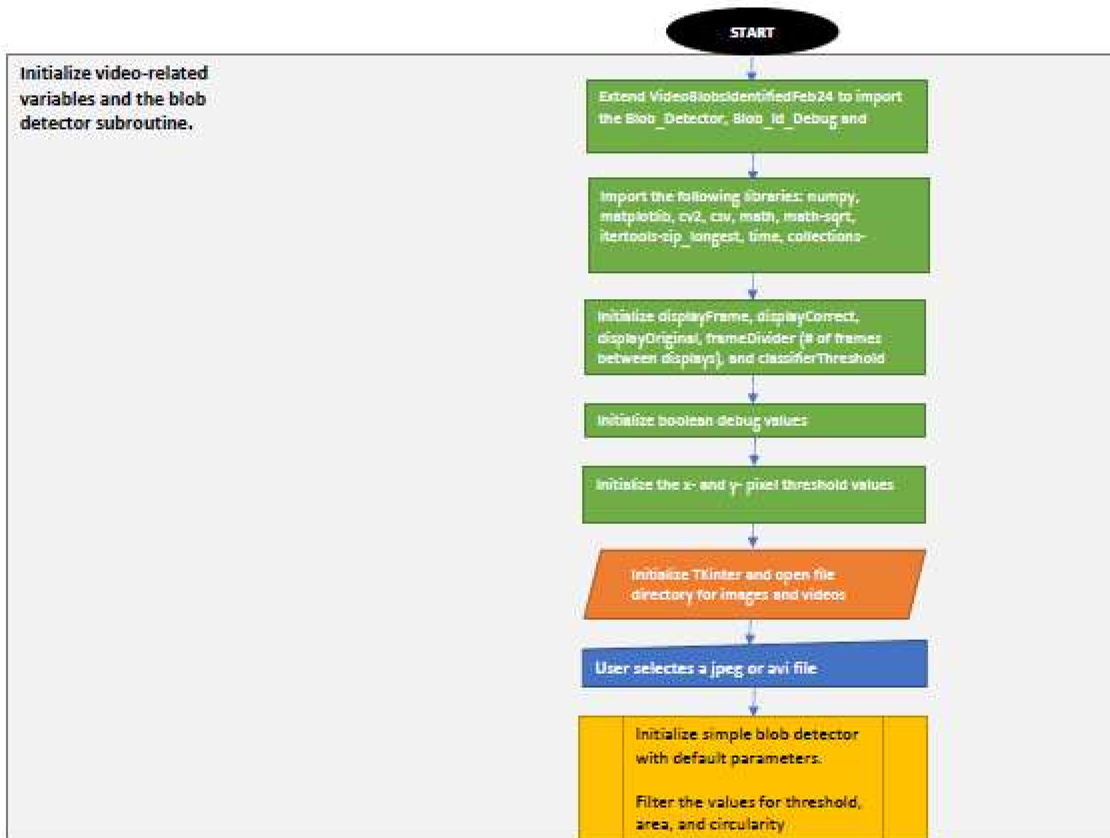
The project was supposed to utilize computer vision functions to find, identify, and follow the markers on the walking subject so that their trajectories can become useful for reconstructing walking models with automated step height and angle measurements. With appropriate user-selected values for area, circularity, and threshold; the blob detector can find up to 78.63 percent of the blobs located on the subject's shoes and knees. This outcome leads to the effective matching of most blob descriptors between frames while the blobs found within the room are being discarded. In constructing and comparing complete trajectories of the left heel and left toe, the user can observe and find ways to correct the plots misidentifying the blobs that were physically located elsewhere in the video. Trajectories with flat regions indicate that the left foot is stable on the ground so long as the standard deviation of (x, y)-coordinate pairs among ten frames is rarely changing, making the shoe blobs more identifiable. Lastly, the blob positions where the left shoe is on the ground are being reidentified as either the toe, heel, or ankle. The relabeling of markers depends on how a Python program compares them in terms of y-coordinate positions, extends the comparison of blobs to the right leg, and maps out pixels on the ground.

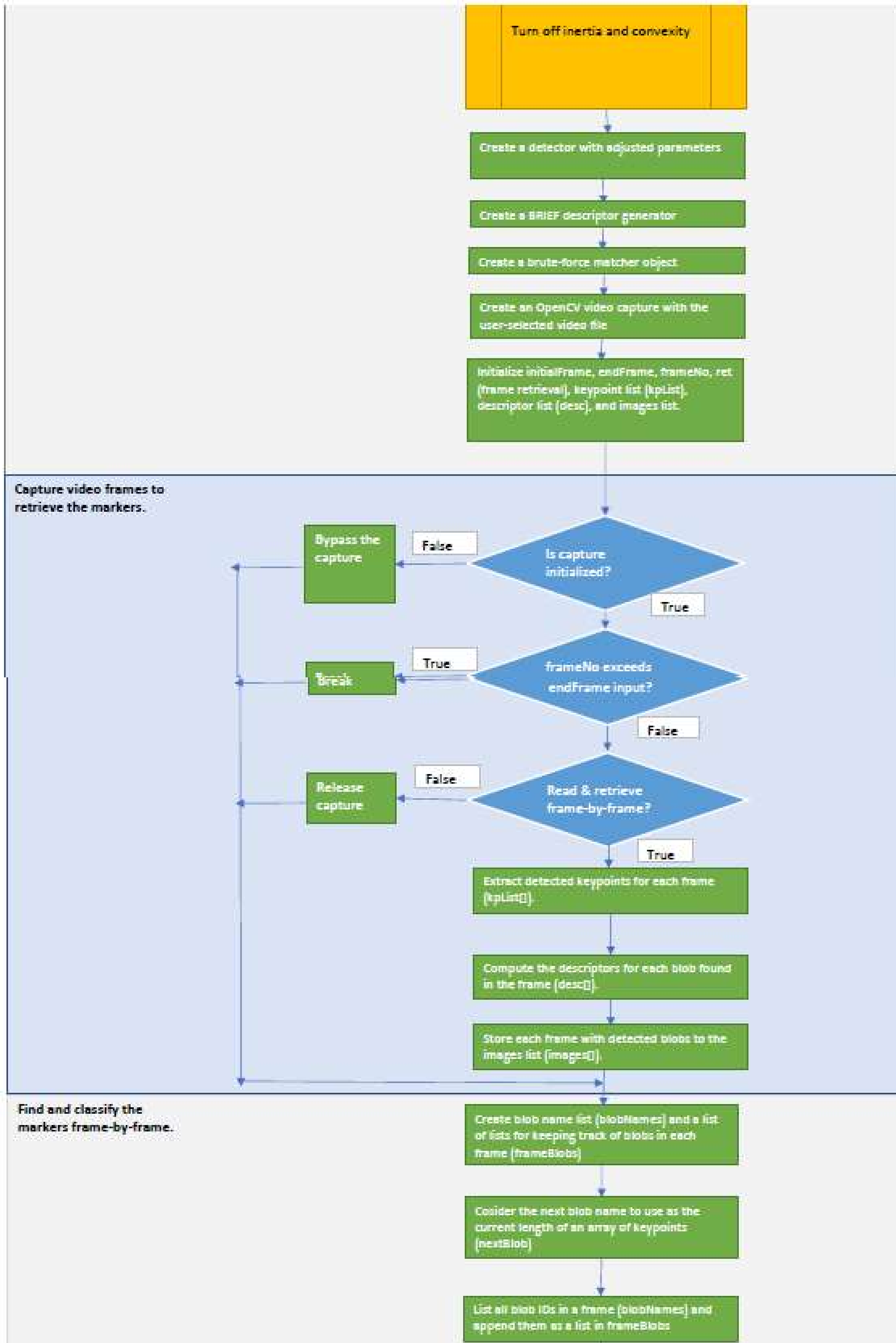
Section 6.2: Future Directions

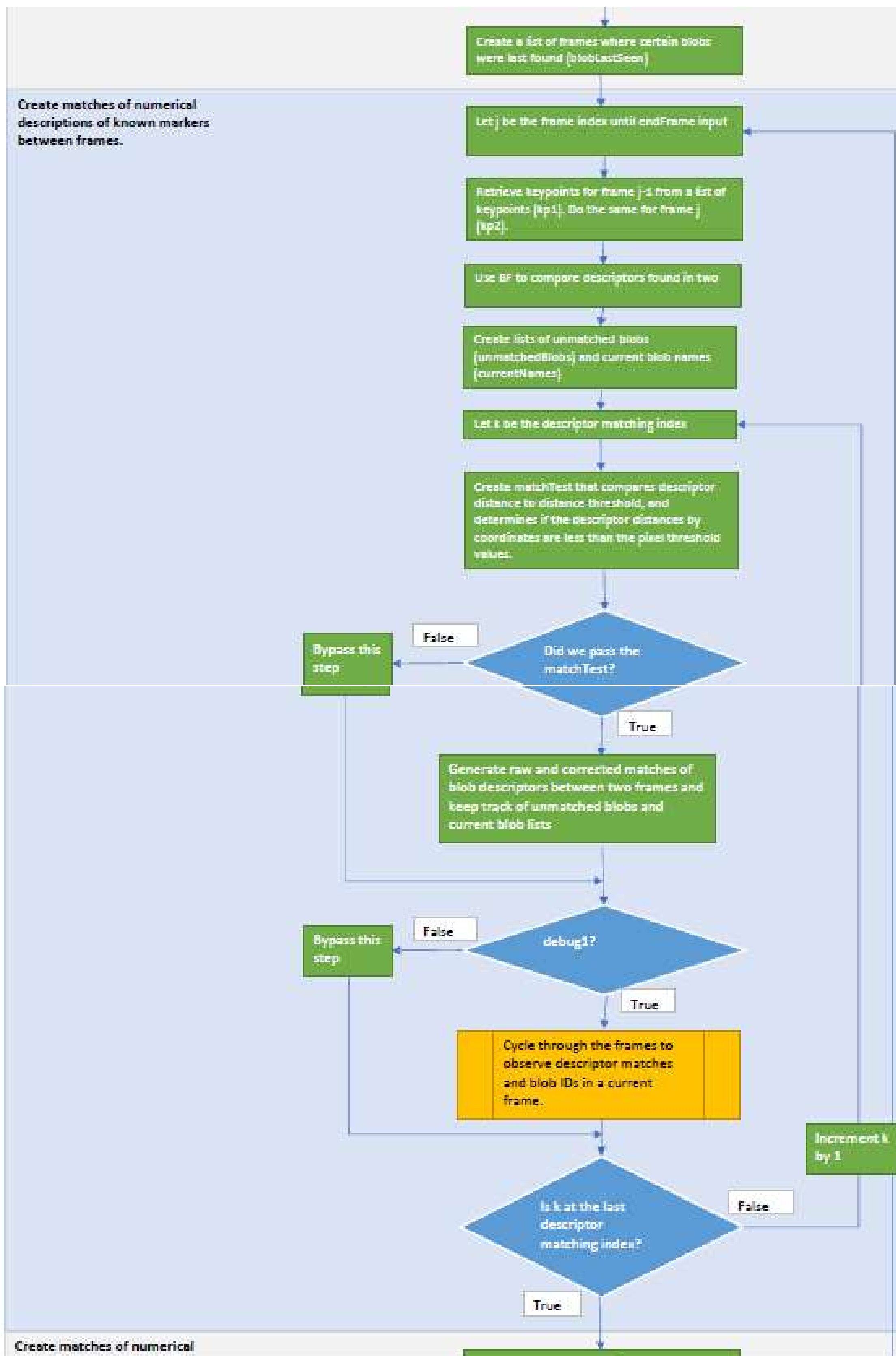
If this project continues, it will involve piecing together all the individual blob trajectories that associate with each toe, ankle, heel, and knee. Afterward, the user will be able to take advantage of the complete trajectories to extract the step height from the blob positions. Mapping the ankle dorsiflexion angles between the blobs is another task to be completed when the user reconstructs walking models. Once the step height and angle are both found from one video, the software can apply to other videos where they involve subjects wearing differently colored clothes

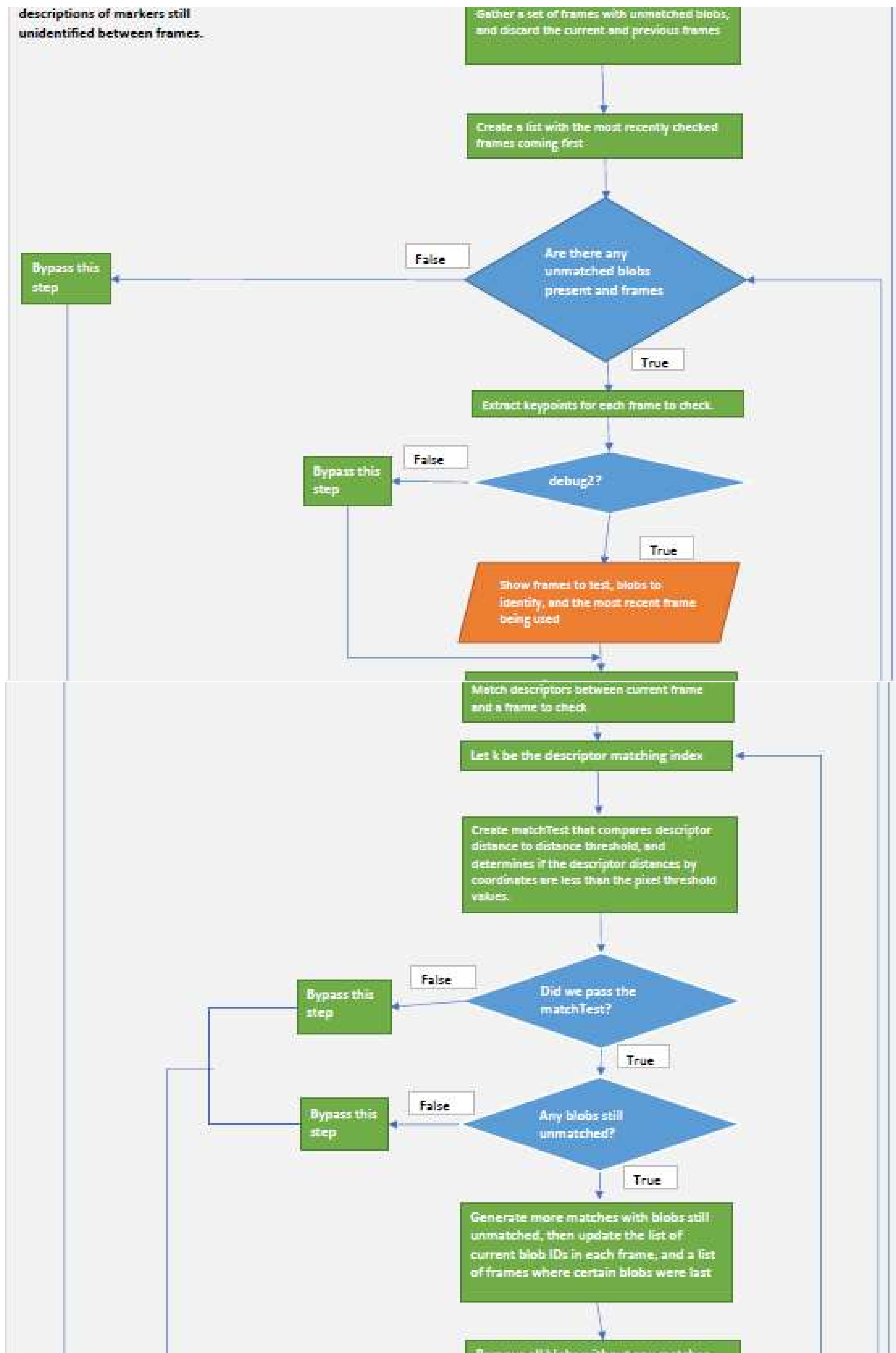
and differently drawn markers. Minor modifications to the software can include adjusting the thresholds and other blob detector parameters. The kinesiology investigators will be able to benefit from this project's data processing techniques because they need to generate a more comprehensive database to know what factors cause an older adult to fall. The software should be able to quickly process and return more accurate measurements that tell about the subjects at risk. The kinesiology team could also try out a more useful set of exercises to see whether the falls among the elderly become less frequent and are eventually preventable in the future.

Appendix A: Blob Detection & Identification Flowchart

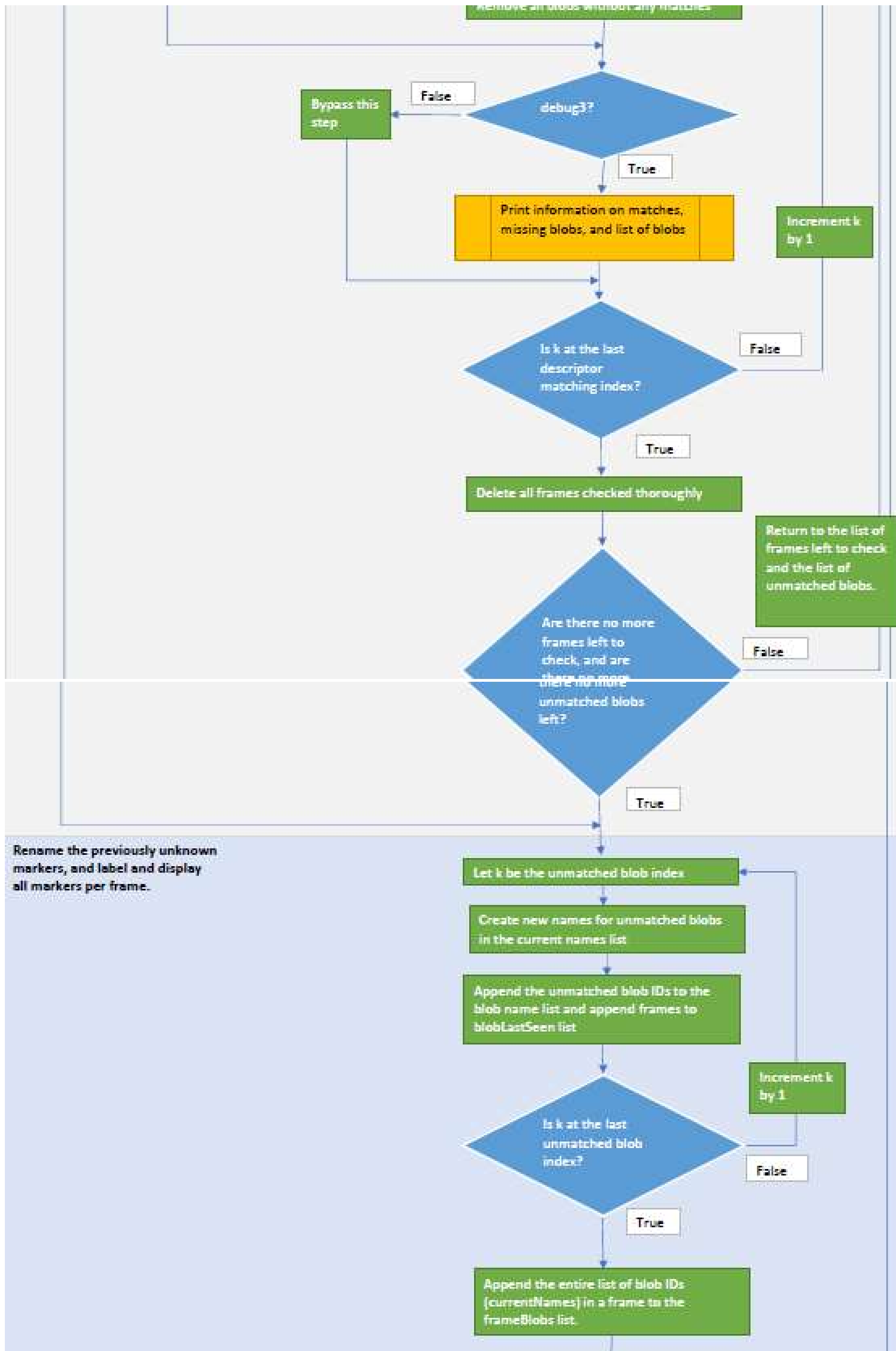


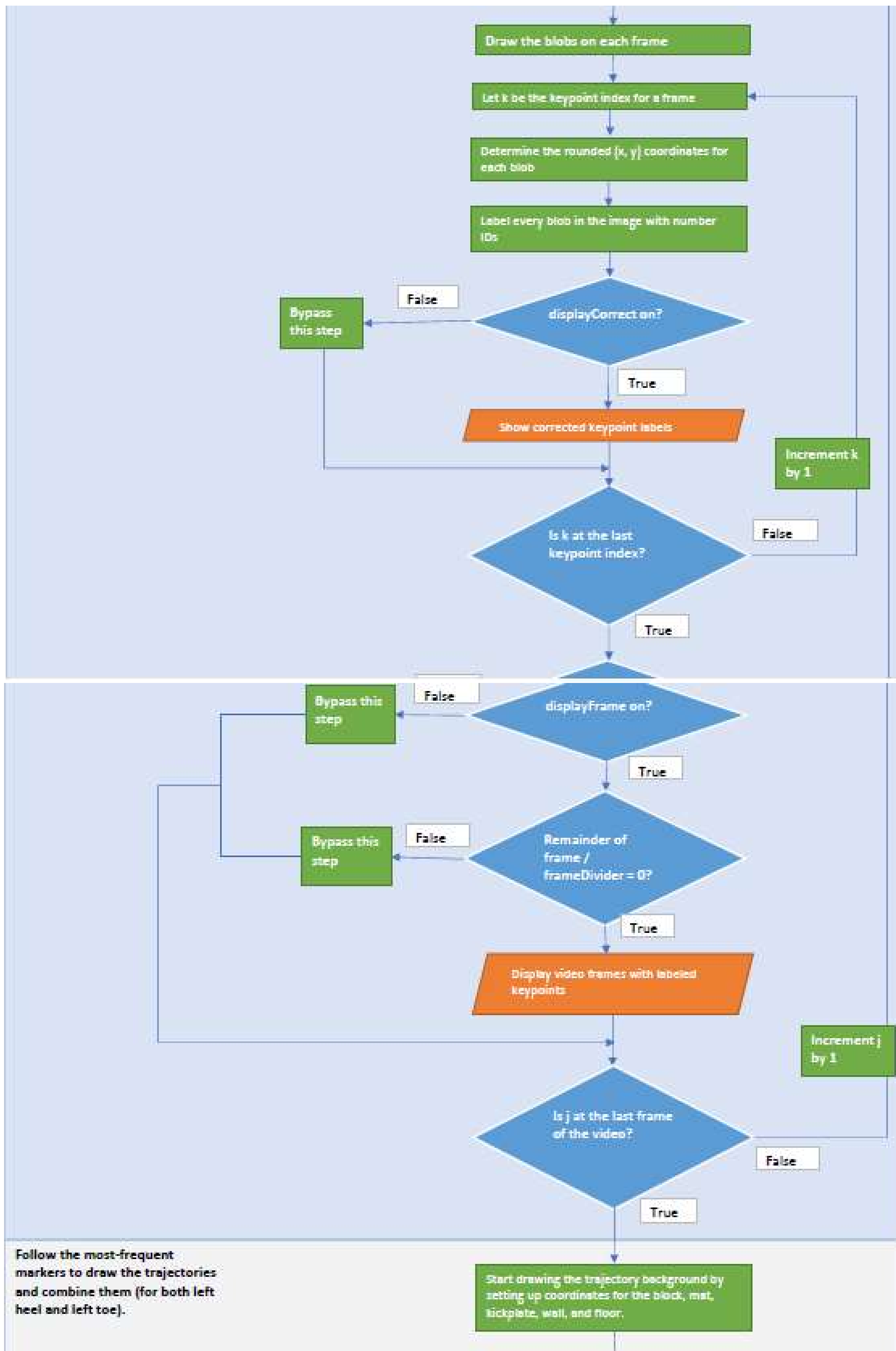

Turn off inertia and convexity
Create a detector with adjusted parameters
Create a BRIEF descriptor generator
Create a brute-force matcher object
Create an OpenCV video capture with the user-selected video file
Initialize initialFrame, endFrame, frameNo, ret (frame retrieval), keypoint list (kpList), descriptor list (desc), and images list.
Capture video frames to retrieve the markers.
Is capture initialized?
False
Bypass the capture
True
frameNo exceeds endFrame input?
True
Break
False
Read & retrieve frame-by-frame?
False
Release capture
True
Extract detected keypoints for each frame (kpList[]).
Compute the descriptors for each blob found in the frame (desc[]).
Store each frame with detected blobs to the images list (images[]).
Find and classify the markers frame-by-frame.
Create blob name list (blobNames) and a list of lists for keeping track of blobs in each frame (frameBlobs)
Cosider the next blob name to use as the current length of an array of keypoints (nextBlob)
List all blob IDs in a frame (blobNames) and append them as a list in frameBlobs


Create a list of frames where certain blobs were last found (blobLastSeen)
Create matches of numerical descriptions of known markers between frames.
Let j be the frame index until endFrame input
Retrieve keypoints for frame j-1 from a list of keypoints (kp1). Do the same for frame j (kp2).
Use BF to compare descriptors found in two
Create lists of unmatched blobs (unmatchedBlobs) and current blob names (currentNames)
Let k be the descriptor matching index
Create matchTest that compares descriptor distance to distance threshold, and determines if the descriptor distances by coordinates are less than the pixel threshold values.
False
Bypass this step
Did we pass the matchTest?
True
Generate raw and corrected matches of blob descriptors between two frames and keep track of unmatched blobs and current blob lists
False
Bypass this step
debug1?
True
Cycle through the frames to observe descriptor matches and blob IDs in a current frame.
Increment k by 1
Is k at the last descriptor matching index?
False
True
Create matches of numerical

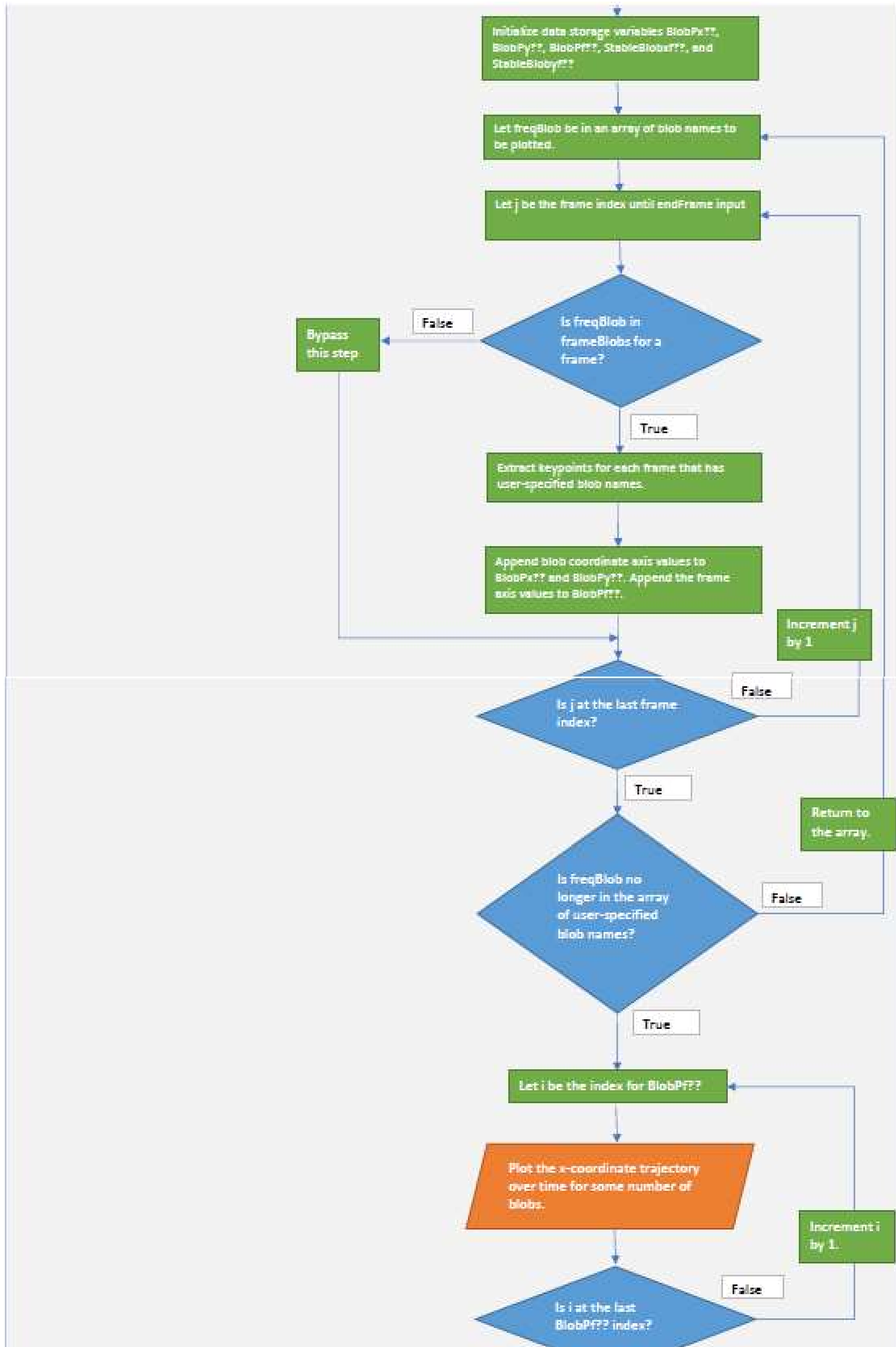


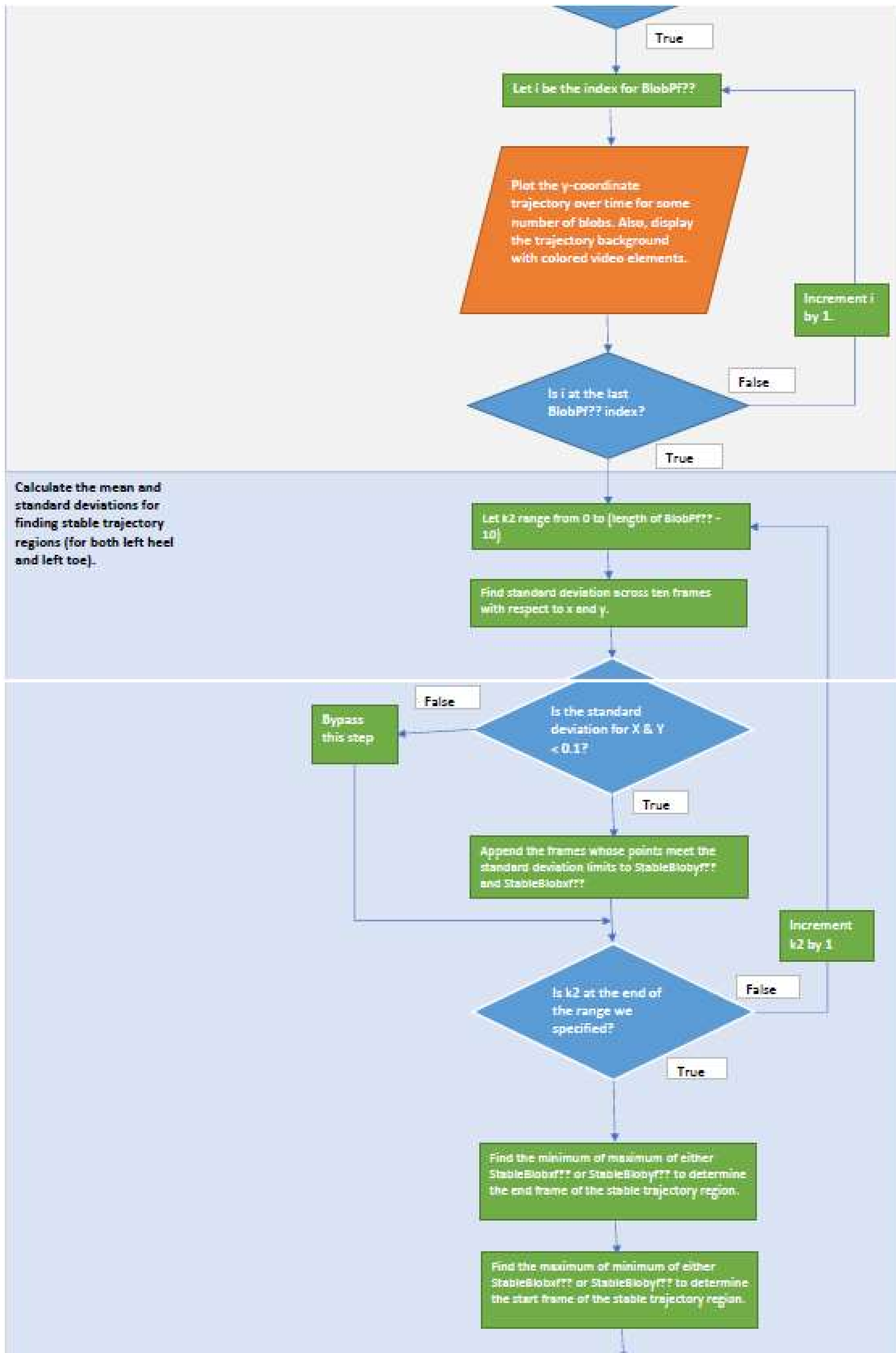
descriptions of markers still unidentified between frames.
Gather a set of frames with unmatched blobs, and discard the current and previous frames
Create a list with the most recently checked frames coming first
Are there any unmatched blobs present and frames
False
Bypass this step
True
Extract keypoints for each frame to check.
debug2?
False
Bypass this step
True
Show frames to test, blobs to identify, and the most recent frame being used
Match descriptors between current frame and a frame to check
Let k be the descriptor matching index
Create matchTest that compares descriptor distance to distance threshold, and determines if the descriptor distances by coordinates are less than the pixel threshold values.
Did we pass the matchTest?
False
Bypass this step
True
Any blobs still unmatched?
False
Bypass this step
True
Generate more matches with blobs still unmatched, then update the list of current blob IDs in each frame, and a list of frames where certain blobs were last

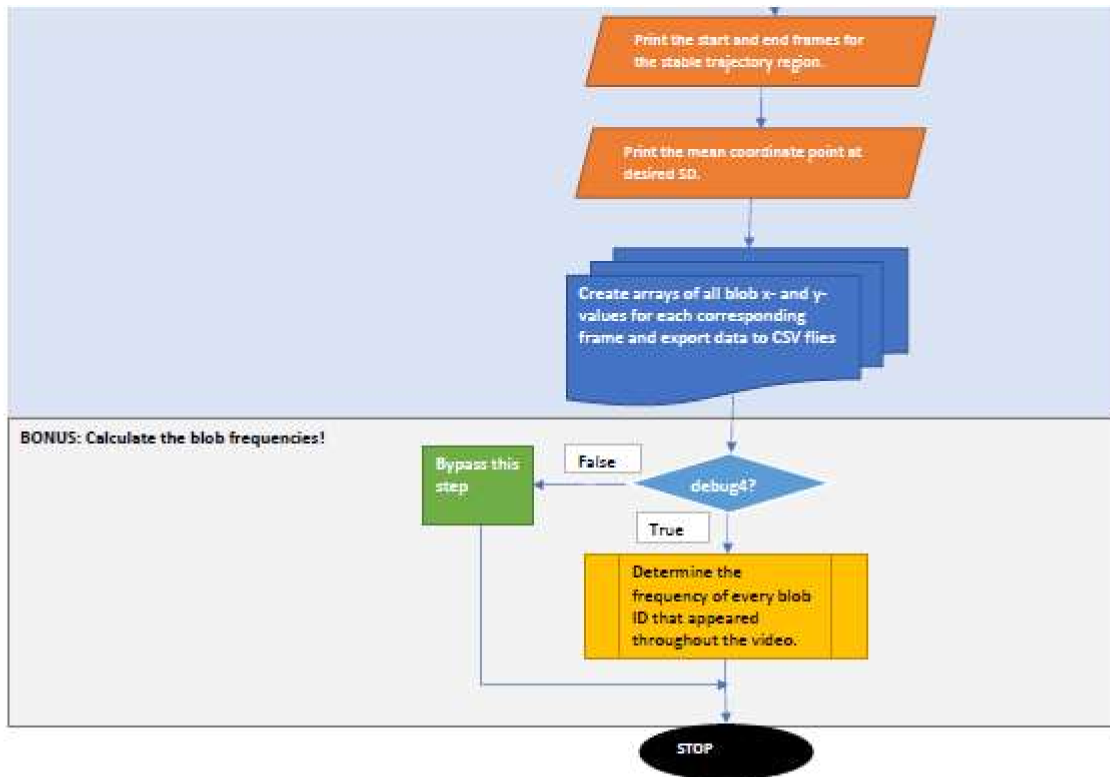

Remove all blobs without any matches
debug3?
False
Bypass this step
True
Print information on matches, missing blobs, and list of blobs
Increment k by 1
Is k at the last descriptor matching index?
False
True
Delete all frames checked thoroughly
Return to the list of frames left to check and the list of unmatched blobs.
Are there no more frames left to check, and are there no more unmatched blobs left?
False
True
Rename the previously unknown markers, and label and display all markers per frame.
Let k be the unmatched blob index
Create new names for unmatched blobs in the current names list
Append the unmatched blob IDs to the blob name list and append frames to blobLastSeen list
Increment k by 1
Is k at the last unmatched blob index?
False
True
Append the entire list of blob IDs (currentNames) in a frame to the frameBlobs list.



Draw the blobs on each frame
Let k be the keypoint index for a frame
Determine the rounded (x, y) coordinates for each blob
Label every blob in the image with number IDs
False
Bypass this step
displayCorrect on?
True
Show corrected keypoint labels
Increment k by 1
Is k at the last keypoint index?
False
True
False
Bypass this step
displayFrame on?
True
Bypass this step
False
Remainder of frame / frameDivider = 0?
True
Display video frames with labeled keypoints
Increment j by 1
Is j at the last frame of the video?
False
True
Follow the most-frequent markers to draw the trajectories and combine them (for both left heel and left toe).
Start drawing the trajectory background by setting up coordinates for the block, mat, kickplate, wall, and floor.


Initialize data storage variables BlobPx??, BlobPy??, BlobPf??, StableBlobxf??, and StableBlobyf??
Let freqBlob be in an array of blob names to be plotted.
Let j be the frame index until endFrame input
Is freqBlob in frameBlobs for a frame?
False
Bypass this step
True
Extract keypoints for each frame that has user-specified blob names.
Append blob coordinate axis values to BlobPx?? and BlobPy??. Append the frame axis values to BlobPf??.
Increment j by 1
False
Is j at the last frame index?
True
Return to the array.
Is freqBlob no longer in the array of user-specified blob names?
False
True
Let i be the index for BlobPf??
Plot the x-coordinate trajectory over time for some number of blobs.
Increment i by 1.
False
Is i at the last BlobPf?? index?


True
Let i be the index for BlobPf??
Plot the y-coordinate trajectory over time for some number of blobs. Also, display the trajectory background with colored video elements.
Increment i by 1.
False
Is i at the last BlobPf?? index?
True
Calculate the mean and standard deviations for finding stable trajectory regions (for both left heel and left toe).
Let k2 range from 0 to (length of BlobPf?? - 10)
Find standard deviation across ten frames with respect to x and y.
False
Bypass this step
Is the standard deviation for X & Y < 0.1?
True
Append the frames whose points meet the standard deviation limits to StableBlobyf?? and StableBlobxf??
Increment k2 by 1
False
Is k2 at the end of the range we specified?
True
Find the minimum of maximum of either StableBlobxf?? or StableBlobyf?? to determine the end frame of the stable trajectory region.
Find the maximum of minimum of either StableBlobxf?? or StableBlobyf?? to determine the start frame of the stable trajectory region.



Print the start and end frames for the stable trajectory region.
Print the mean coordinate point at desired SD.
Create arrays of all blob x- and y-values for each corresponding frame and export data to CSV files
BONUS: Calculate the blob frequencies!
Bypass this step
False
debug4?
True
Determine the frequency of every blob ID that appeared throughout the video.
STOP

Appendix B: List of OpenCV-Python Programs

Program Name	Last Modified	Steps to Execution	Page Number for Source Code
VideoReadKinesiologyTesting.py	6/1/2019	Upon execution, the program outputs the video file in grayscale.	49
CannyEdge.py	6/1/2019	Upon execution, the program produces an image under the canny edge detection method.	50
FindHoughLinesProb.py	6/2/2019	Upon execution, the program creates an image with probabilistic Hough lines.	51
TKinter_FAST.py	6/5/2019	1. User chooses an image. 2. User selects a region of interest (ROI) by dragging the box over the desired area with a mouse. 3. Program crops the image to match the ROI. 4. Program initiates the FAST method with default values. 5. Program finds and draws the keypoints. 6. Console shows all default parameters.	52
TKinter_ORB.py	6/10/2019	1. User chooses an image. 2. User selects a region of interest (ROI) by dragging the box over the desired area with a mouse.	53

		3. Program crops the image to match the ROI. 4. Program initiates the ORB method. 5. Program finds and draws the keypoints. 6. Console shows the number of keypoints and ROI coordinates.	
MultipleThresholds_Step10.py	7/10/2019	1. User selects a frame 2. Program produces a blob detector frame with labeled keypoints. 3. Program magnifies and crops the image for every keypoint detected. 4. Program provides each keypoint with a range of thresholds specified by the user within the source code.	54
MultipleThresholds_CircOff.py	7/19/2019	1. User selects a frame 2. Program produces a blob detector frame with labeled keypoints. 3. Program magnifies and crops the image for every keypoint detected. 4. Program provides each keypoint with a range of thresholds specified by the user within the source code.	58
FindBlobs_OriginalParams.py	7/25/2019	1. User chooses a jpeg image.	62

		2. Console prints the default and custom blob detector parameters. 3. Program locates and draws keypoints Console prints the (x, y) coordinates and size/area for each keypoint.	
FindVideoBlobs.py	10/25/2019	1. User chooses an AVI video file. 2. Program activates the blob detector, draws keypoints, and displays Frame 1. 3. User presses any key to move to the next frames. He or she will do so 240 times. 4. After 240 frames, the program terminates.	64
FindBlobs_NewParams.py	10/29/2019	1. User chooses a JPEG image. 2. Program locates and draws keypoints. 3. Program outputs the resulting image. 4. Console prints the (x, y) coordinates and size/area for each keypoint.	67
VideoBlobsIdentifiedFeb24.py	4/18/2020	Not applicable.	69
VideoBlobsIdentifiedFinal.py	5/8/2020	1. User selects an AVI video file. 2. User inputs the final frame to analyze. 3. Console prints the start and end frames for stable trajectory	72

		regions of the left heel and left toe. 4. If displayFrame = True, the program displays every *n*th frame, where *n* is the number of frames between displays. 5. If debug1 = True, the console prints generated matches from the first frame cycle for known blobs 6. If debug2 = True, the console prints frames to check, missing blobs to identify, and the current frame being checked. 7. If debug3 = True, the console prints generated matches from the second frame cycle for blobs still unidentified. 8. If debug4 = True, the console prints the tallies of all the distinct blob names. 9. With the Matplotlib library, the program displays x- and y-coordinate trajectories per human feature.	

Appendix C: Full-Length Source Codes (Electronic Copy Exclusive)

VideoReadKinesiologyTesting.py

```
import numpy as np
import cv2 as cv2
cap = cv2.VideoCapture('S10.B1.avi')
while cap.isOpened():
    ret, frame = cap.read()
    # if frame is read correctly ret is True
    if not ret:
        print("Can't receive frame (stream end?). Exiting ...")
        break
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    cv2.imshow('frame', gray)
    if cv2.waitKey(1) == ord('q'):
        break
cap.release()
cv2.destroyAllWindows()
```

CannyEdge.py

```
import numpy as np
import cv2 as cv2
#from matplotlib import pyplot as plt

img = cv2.imread('testframe50.jpg',0)
edges = cv2.Canny(img,100,200)
cv2.imshow('',edges)
```

FindHoughLinesProb.py

```
import cv2 as cv
import numpy as np
img = cv.imread('testframe50.jpg')
gray = cv.cvtColor(img,cv.COLOR_BGR2GRAY)
edges = cv.Canny(gray,50,150,apertureSize = 3)
lines =
cv.HoughLinesP(edges,1,np.pi/180,100,minLineLength=100,maxLineGap=10)
for line in lines:
    x1,y1,x2,y2 = line[0]
    cv.line(img,(x1,y1),(x2,y2),(0,255,0),2)
#cv.imwrite('houghlines5.jpg',img)
cv.imshow('houghlines',img)
```

TKinter_FAST.py

```
import numpy as np
import cv2 as cv2
import tkinter as tk
from tkinter import filedialog
from tkinter import messagebox
from tkinter import simpledialog

root = tk.Tk()
root.withdraw()
application_window = tk.Tk()
application_window.withdraw()
file_path = filedialog.askopenfilename(initialdir = ".",title =
"Select file",filetypes = (("jpeg files","*.jpg"),("all
files","*.*")))

img = cv2.imread(file_path)
img = img[50:770,:]
print(file_path)

cv2.imshow("Original",img)
# Select ROI
fromCenter = False
r = cv2.selectROI("Original",img,fromCenter)
# Crop image
img = img[int(r[1]):int(r[1]+r[3]), int(r[0]):int(r[0]+r[2])]
# Initiate FAST object with default values
fast = cv2.FastFeatureDetector_create()
# find and draw the keypoints
kp = fast.detect(img,None)
img2 = cv2.drawKeypoints(img, kp, None, color=(255,0,0))
#fast.setThreshold(1)
# Print all default params
print( "Threshold: {}".format(fast.getThreshold()) )
print( "nonmaxSuppression:{}".format(fast.getNonmaxSuppression()) )
print( "neighborhood: {}".format(fast.getType()) )
print( "Total Keypoints with nonmaxSuppression: {}".format(len(kp)) )
# Disable nonmaxSuppression
fast.setNonmaxSuppression(0)
kp = fast.detect(img,None)
print( "Total Keypoints without nonmaxSuppression: {}".format(len(kp))
)
img3 = cv2.drawKeypoints(img, kp, None, color=(255,0,0))
cv2.imshow('', img3)
```

TKinter_ORB.py

```
import numpy as np
import cv2 as cv
import tkinter as tk
from tkinter import filedialog
from tkinter import messagebox
from tkinter import simpledialog

root = tk.Tk()
root.withdraw()
application_window = tk.Tk()
application_window.withdraw()
file_path = filedialog.askopenfilename(initialdir = ".",title =
"Select file",filetypes = (("jpeg files","*.jpg"),("all
files","*.*")))

img = cv.imread(file_path)
#img =img[640:740,780:960]
print(file_path)

cv.imshow("Original",img)
# Select ROI
fromCenter = False
print("Select Region of Interest")
r = cv.selectROI("Original",img,fromCenter)
print(r)
# Crop image
img = img[int(r[1]):int(r[1]+r[3]), int(r[0]):int(r[0]+r[2])]
# Initiate ORB detector
orb = cv.ORB_create()
# find the keypoints with ORB
kp = orb.detect(img,None)
print(len(kp))
# compute the descriptors with ORB
kp, des = orb.compute(img, kp)
# draw only keypoints location,not size and orientation
img2 = cv.drawKeypoints(img, kp, None, color=(0,255,0), flags=0)
cv.imshow("Final",img2)

cv.waitKey(0)
cv.destroyAllWindows()
```

MultipleThresholds_Step10.py

```
import numpy as np
import cv2
import tkinter as tk
from tkinter import filedialog
from tkinter import messagebox
from tkinter import simpledialog

root = tk.Tk()
root.withdraw()
application_window = tk.Tk()
application_window.withdraw()
file_path = filedialog.askopenfilename(initialdir = ".",title =
"Select file",filetypes = (("jpeg files","*.jpg"),("all
files","*.*")))

# Reading the image
img = cv2.imread(file_path)
# img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
img = img[50:770,:]

# Setup SimpleBlobDetector parameters.
params = cv2.SimpleBlobDetector_Params()
print(params)
# Change thresholds
params.minThreshold = 20
params.maxThreshold = 220
params.thresholdStep = 20

# Filter by Area.
params.filterByArea = True
params.minArea = 5
params.maxArea = 50

# Filter by Circularity
params.filterByCircularity = True
params.minCircularity = 0.84

# Filter by Convexity
params.filterByConvexity = False
#params.minConvexity = 0.7

# Filter by Inertia
params.filterByInertia = False
```

```
#params.minInertiaRatio = 0.01

# Create a detector with the parameters
detector = cv2.SimpleBlobDetector_create(params)

# Detect blobs.
keypoints1 = detector.detect(img)

print(len(keypoints1))

# Draw detected blobs as red circles.
# cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS ensures
# the size of the circle corresponds to the size of blob

im_with_keypoints1 = cv2.drawKeypoints(img, keypoints1, np.array([]),
(0,0,255), cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

for k in range(0,9):
    cx1 = round(keypoints1[k].pt[0])
    cy1 = round(keypoints1[k].pt[1])
    cv2.putText(im_with_keypoints1, str(k), (cx1, cy1),
cv2.FONT_HERSHEY_SIMPLEX, .6,(0, 0, 255))

# Show blobs
cv2.imshow("Keypoints", im_with_keypoints1)
# cv2.imshow("Keypoints 2", im_with_keypoints2)
# cv2.imshow("Keypoints 3", im_with_keypoints3)

img1 = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
# Create thresholded images
rret,thresh1 = cv2.threshold(img1,130,255,cv2.THRESH_BINARY)
rret,thresh2 = cv2.threshold(img1,140,255,cv2.THRESH_BINARY)
rret,thresh3 = cv2.threshold(img1,150,255,cv2.THRESH_BINARY)
rret,thresh4 = cv2.threshold(img1,160,255,cv2.THRESH_BINARY)

threshImages = [thresh1,thresh2,thresh3,thresh4]
threshValues = [130,140,150,160]

# Cropping to Keypoint something

for k in range(0,9):
    # Cropped region is 2*halfWidth
    halfWidth = 5
    xVal1 = round(keypoints1[k].pt[0]-halfWidth)
    xVal2 = round(keypoints1[k].pt[0]+halfWidth)
    yVal1 = round(keypoints1[k].pt[1]-halfWidth)
```

```
    yVal2 = round(keypoints1[k].pt[1]+halfWidth)
    print(k,keypoints1[k].pt,keypoints1[k].size)
    #print(xVal1,':',xVal2)
    #columns: Top to bottom, then rows: Left to Right
        # This crops the orignal picture
    crop1 = img1[yVal1:yVal2,xVal1:xVal2]
    # Continue with this sequence!
    # Aim for a window size of 250 X 250 to show the title
    #    scalePixel1 = round(250/(xVal2_1-xVal1_1))
    scalePixel = 10;
    #print(scalePixel)
    pxFill = np.ones([scalePixel,scalePixel])
    #print(xVal2-xVal1)
    scaleUp1 = cv2.resize(crop1,(scalePixel*2*halfWidth,
scalePixel*2*halfWidth),cv2.INTER_NEAREST)
    scaleUp2 = cv2.resize(crop1,(scalePixel*2*halfWidth,
scalePixel*2*halfWidth),cv2.INTER_NEAREST)
    # Now add pixelization back to eliminate the interpolation
    # Remember that indices start at 0
    # Use this if you want to see the smoothed image from opencv
resize
    #cv2.imshow('Smoothed Image',scaleUp)

    # This repixelates the image to show the true data
    # Original
    for kRow in range(0,2*halfWidth-1):
        for kCol in range(0, 2*halfWidth-1):
            # Gray version

scaleUp1[kRow*scalePixel:(kRow*scalePixel+scalePixel),kCol*scalePixel:
(kCol*scalePixel+scalePixel)] = crop1[kRow,kCol]*pxFill

    windowName = 'Orig KeyPt '+str(k)
    cv2.imshow(windowName, scaleUp1)
    cv2.moveWindow(windowName, 200,200)

    for kThresh in range(0,4):
        currentThresh = threshImages[kThresh]
        # This is the thresholded version
            # This crops the first thresholded picture
        threshCrop1 = currentThresh[yVal1:yVal2,xVal1:xVal2]
        for kRow in range(0,2*halfWidth):
            for kCol in range(0, 2*halfWidth):
                # Gray version
```

```
scaleUp2[kRow*scalePixel:(kRow*scalePixel+scalePixel),kCol*scalePixel:
(kCol*scalePixel+scalePixel)] = threshCrop1[kRow,kCol]*pxFill

        windowName = 'Thresh '+str(threshValues[kThresh])+' KeyPt
'+str(k)
        cv2.imshow(windowName, scaleUp2)
        cv2.moveWindow(windowName, 200,200)

# cv2.waitKey(0)
# cv2.destroyAllWindows()
```

MultipleThresholds_CircOff.py

```
import numpy as np
import cv2 as cv2
import tkinter as tk
from tkinter import filedialog
from tkinter import messagebox
from tkinter import simpledialog

root = tk.Tk()
root.withdraw()
application_window = tk.Tk()
application_window.withdraw()
file_path = filedialog.askopenfilename(initialdir = ".",title =
"Select file",filetypes = (("jpeg files","*.jpg"),("all
files","*.*")))

# Reading the image
img = cv2.imread(file_path)
img = img[50:770,:]

# Setup SimpleBlobDetector parameters.
params = cv2.SimpleBlobDetector_Params()
print(params)
# Change thresholds
params.minThreshold = 80
params.maxThreshold = 140

params.thresholdStep = 20

# Filter by Area.
params.filterByArea = True
params.minArea = 5
params.maxArea = 50

# Filter by Circularity
params.filterByCircularity = False
params.minCircularity = 0.84

# params.minRepeatability = 1

# Filter by Convexity
params.filterByConvexity = False
#params.minConvexity = 0.7
```

```
# Filter by Inertia
params.filterByInertia = False
#params.minInertiaRatio = 0.01

# Create a detector with the parameters
detector = cv2.SimpleBlobDetector_create(params)

# Detect blobs.
keypoints1 = detector.detect(img)

print(len(keypoints1))

# Draw detected blobs as red circles.
# cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS ensures
# the size of the circle corresponds to the size of blob

im_with_keypoints1 = cv2.drawKeypoints(img, keypoints1, np.array([]),
(0,0,255), cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

for k in range(0,len(keypoints1)):
    cx1 = round(keypoints1[k].pt[0])
    cy1 = round(keypoints1[k].pt[1])
    cv2.putText(im_with_keypoints1, str(k), (cx1, cy1),
cv2.FONT_HERSHEY_SIMPLEX, .6,(0, 0, 255))

# Show blobs
cv2.imshow("Keypoints", im_with_keypoints1)
# cv2.imshow("Keypoints 2", im_with_keypoints2)
# cv2.imshow("Keypoints 3", im_with_keypoints3)

img1 = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
# Create thresholded images
rret,thresh1 = cv2.threshold(img1,80,255,cv2.THRESH_BINARY)
rret,thresh2 = cv2.threshold(img1,140,255,cv2.THRESH_BINARY)

threshImages = [thresh1,thresh2]
threshValues = [80,140]

# Cropping to Keypoint something

for k in range(0,len(keypoints1)):
    # Cropped region is 2*halfWidth
    halfWidth = 5
    xVal1 = round(keypoints1[k].pt[0]-halfWidth)
    xVal2 = round(keypoints1[k].pt[0]+halfWidth)
    yVal1 = round(keypoints1[k].pt[1]-halfWidth)
```

```
    yVal2 = round(keypoints1[k].pt[1]+halfWidth)
    print(k,keypoints1[k].pt,keypoints1[k].size)
    #print(xVal1,':',xVal2)
    #columns: Top to bottom, then rows: Left to Right
        # This crops the orignal picture
    crop1 = img1[yVal1:yVal2,xVal1:xVal2]
    # Continue with this sequence!
    # Aim for a window size of 250 X 250 to show the title
    #    scalePixel1 = round(250/(xVal2_1-xVal1_1))
    scalePixel = 10;
    #print(scalePixel)
    pxFill = np.ones([scalePixel,scalePixel])
    #print(xVal2-xVal1)
    scaleUp1 = cv2.resize(crop1,(scalePixel*2*halfWidth,
scalePixel*2*halfWidth),cv2.INTER_NEAREST)
    scaleUp2 = cv2.resize(crop1,(scalePixel*2*halfWidth,
scalePixel*2*halfWidth),cv2.INTER_NEAREST)
    # Now add pixelization back to eliminate the interpolation
    # Remember that indices start at 0
    # Use this if you want to see the smoothed image from opencv
resize
    #cv2.imshow('Smoothed Image',scaleUp)

    # This repixelates the image to show the true data
    # Original
    for kRow in range(0,2*halfWidth-1):
        for kCol in range(0, 2*halfWidth-1):
            # Gray version

scaleUp1[kRow*scalePixel:(kRow*scalePixel+scalePixel),kCol*scalePixel:
(kCol*scalePixel+scalePixel)] = crop1[kRow,kCol]*pxFill

    windowName = 'Orig KeyPt '+str(k)
    cv2.imshow(windowName, scaleUp1)
    cv2.moveWindow(windowName, 200,200)

    for kThresh in range(0,2):
        currentThresh = threshImages[kThresh]
        # This is the thresholded version
            # This crops the first thresholded picture
        threshCrop1 = currentThresh[yVal1:yVal2,xVal1:xVal2]
        for kRow in range(0,2*halfWidth):
            for kCol in range(0, 2*halfWidth):
                # Gray version
```

```
scaleUp2[kRow*scalePixel:(kRow*scalePixel+scalePixel),kCol*scalePixel:
(kCol*scalePixel+scalePixel)] = threshCrop1[kRow,kCol]*pxFill

        windowName = 'Thresh '+str(threshValues[kThresh])+' KeyPt
'+str(k)
        cv2.imshow(windowName, scaleUp2)
        cv2.moveWindow(windowName, 200,200)

# cv2.waitKey(0)
# cv2.destroyAllWindows()
```

FindBlobs_OriginalParams.py

```
import numpy as np
import cv2
import tkinter as tk
from tkinter import filedialog
from tkinter import messagebox
from tkinter import simpledialog
#from matplotlib import pyplot as plt

root = tk.Tk()
root.withdraw()
application_window = tk.Tk()
application_window.withdraw()
file_path = filedialog.askopenfilename(initialdir = ".",title =
"Select file",filetypes = (("jpeg files","*.jpg"),("all
files","*.*")))

img = cv2.imread(file_path)
img =img[50:770,:]
#print(file_path)

# Setup SimpleBlobDetector parameters.
params = cv2.SimpleBlobDetector_Params()

# Define the default values of the blob parameters.
print("Default Minimum Threshold: " + str(params.minThreshold))
print("Default Maximum Threshold: " + str(params.maxThreshold) + "\n")

print("Default Minimum Area: " + str(params.minArea))
print("Default Maximum Area: " + str(params.maxArea) + "\n")

print("Default Circularity: " + str(params.minCircularity) + "\n")

print("Default Convexity: " + str(params.minConvexity) + "\n")

print("Default Inertia Ratio: " + str(params.minInertiaRatio) + "\n")

# Change thresholds
params.minThreshold = 10 # Initial 10, Change to 1, then 180
params.maxThreshold = 200 # Initial 200, Change to 20, then 300
print("Custom Minimum Threshold: " + str(params.minThreshold))
print("Custom Maximum Threshold: " + str(params.maxThreshold) + "\n")

# Filter by Area.
```

```
params.filterByArea = True
params.minArea = 12 # Initial 12, Change to 1, then 17
params.maxArea = 18 # Initial 18, Change to 14, then 30
print("Custom Minimum Area: " + str(params.minArea))
print("Custom Maximum Area: " + str(params.maxArea) + "\n")

# Filter by Circularity
params.filterByCircularity = True
params.minCircularity = 0.83 # Intial 0.83, Change to 0, then 0.5,
then 1
print("Custom Circularity: " + str(params.minCircularity) + "\n")

# Filter by Convexity
params.filterByConvexity = False
params.minConvexity = 0.9 # Initial 0.9, change to 0.3, then 1
print("Custom Convexity: " + str(params.minConvexity) + "\n")

# Filter by Inertia
params.filterByInertia = False
params.minInertiaRatio = 0.6
print("Custom Inertia Ratio: " + str(params.minInertiaRatio) + "\n")

# Create a detector with the parameters
detector = cv2.SimpleBlobDetector_create(params)

# Detect blobs.
keypoints = detector.detect(img)

# Draw detected blobs as red circles.
# cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS ensures
# the size of the circle corresponds to the size of blob

im_with_keypoints = cv2.drawKeypoints(img, keypoints, np.array([]),
(0,0,255), cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

# Show blobs
cv2.imshow("Keypoints", im_with_keypoints)

# Print blob coordinates
for k in range(0,len(keypoints)):

print(round(keypoints[k].pt[0]),round(keypoints[k].pt[1]),round(keypoi
nts[k].size))

cv2.waitKey(0)
cv2.destroyAllWindows()
```

FindVideoBlobs.py

```
import numpy as np
import cv2 as cv2
import time
import tkinter as tk
from tkinter import filedialog
from tkinter import messagebox
from tkinter import simpledialog

root = tk.Tk()
root.withdraw()
application_window = tk.Tk()
application_window.withdraw()
# Setup SimpleBlobDetector parameters.
params = cv2.SimpleBlobDetector_Params()

# Change thresholds
params.minThreshold = 40
params.maxThreshold = 200
params.thresholdStep = 20


# Filter by Area.
params.filterByArea = True
params.minArea = 8
params.maxArea = 40

# Filter by Circularity
params.filterByCircularity = True
params.minCircularity = 0.85

# Filter by Convexity
params.filterByConvexity = False
params.minConvexity = 0.9

# Filter by Inertia
params.filterByInertia = False
params.minInertiaRatio = 0.01

# Create a detector with the parameters
detector = cv2.SimpleBlobDetector_create(params)
```

```
file_path = filedialog.askopenfilename(initialdir = ".",title =
"Select file",filetypes = (("jpeg files","*.avi"),("all
files","*.*")))

cap = cv2.VideoCapture(file_path)
outP = open("testOutNoRound.txt","w+")
frameNo = 0;
kpList = [];
while cap.isOpened():
    ret, frame = cap.read()
    #frame = cv2.bitwise_not(frame)
    if ret:
    # Detect Blobs
        #keypoints = detector.detect(frame)
        alpha = 1.1
        beta = -50
        frame = frame[50:770,:]
        imgNew = frame.copy()
        imgNew = cv2.convertScaleAbs(frame,imgNew, alpha,beta)

        kpList.append(detector.detect(frame));
        keypoints = kpList[frameNo]
        frameNo = frameNo + 1
# Draw detected blobs as red circles.
# cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS ensures
# the size of the circle corresponds to the size of blob
        im_with_keypoints = cv2.drawKeypoints(frame, keypoints,
np.array([]), (0,0,255), cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
# Show blobs
        #cv2.destroyAllWindows()
        if 'outText' in locals():
            cv2.destroyWindow(outText)
        outText = "Frame: "+str(frameNo)
        cv2.imshow(outText, im_with_keypoints)
        cv2.moveWindow(outText,0,0)

#        cv2.imshow('', im_with_keypoints)
#        cv2.moveWindow('',0,0)


        # This next step is necessary to force a draw.
        cv2.waitKey(2)
# Print blob coordinates
        outP.write(str(frameNo)+" ")
        for k in range(1,len(keypoints)):
```

```
            outP.write(str(keypoints[k].pt[0])+"
"+str(keypoints[k].pt[1])+" ")
        outP.write('\n')
# Delay for 1/4 second
# Only use this when writing to the same window
# Otherwise there is a lot of flashing
#        time.sleep(.25)
# Delay until key is pressed
        cv2.waitKey(0)
    else:
        outP.close()
        cap.release()
# cv2.destroyAllWindows()
# Now, kpList is the set of all keypoints
# For frame N, retrieve the keypoints by
# keypoints = kpList[N-1]
# To extract data from the third keypoint in frame 240, use
# [x ,y] = kpList[239][2].pt
# Or as
# x = kpList[239][2].pt[0]
# y = kpList[239][2].pt[1]


# Here's a method to sort the keypoints
# This assumes that the set of keypoints is a list called kpList
# Define a method for the key
#def sortSecond(val):
#    return val[1]
# Now, I am creating a list of the keypoint number and the coordinates
# I will use this on the first keypoint, kpList[N-1], and cycle
through all
# of the keypoints contained in it
#for k in range(0,range(kpList[N-1])):
#    test.append([k,kpList[N-1][k].pt])
# Finally, I can sort it
#test.sort(key = sortSecond)
# The first element is the old keypoint listing
# So test[0][0] is the index of the keypoint with the lowest x value
```

FindBlobs_NewParams.py

```
import numpy as np
import cv2 as cv2
import tkinter as tk
from tkinter import filedialog
from tkinter import messagebox
from tkinter import simpledialog
#from matplotlib import pyplot as plt

root = tk.Tk()
root.withdraw()
application_window = tk.Tk()
application_window.withdraw()
file_path = filedialog.askopenfilename(initialdir = ".",title =
"Select file",filetypes = (("jpeg files","*.jpg"),("all
files","*.*")))

img = cv2.imread(file_path)
img = img[50:770,:]
#print(file_path)

# Setup SimpleBlobDetector parameters.
params = cv2.SimpleBlobDetector_Params()

# Change thresholds
params.minThreshold = 40
params.maxThreshold = 200
params.thresholdStep = 20


# Filter by Area.
params.filterByArea = True
params.minArea = 8
params.maxArea = 40

# Filter by Circularity
params.filterByCircularity = True
params.minCircularity = 0.85

# Filter by Convexity
params.filterByConvexity = False
params.minConvexity = 0.9

# Filter by Inertia
```

```
params.filterByInertia = False
params.minInertiaRatio = 0.01

# Create a detector with the parameters
detector = cv2.SimpleBlobDetector_create(params)

# Detect blobs.
keypoints = detector.detect(img)

# Draw detected blobs as red circles.
# cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS ensures
# the size of the circle corresponds to the size of blob

im_with_keypoints = cv2.drawKeypoints(img, keypoints, np.array([]),
(0,0,255), cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

# Show blobs
cv2.imshow("Keypoints", im_with_keypoints)

# Print blob coordinates
for k in range(0,len(keypoints)):

print(round(keypoints[k].pt[0]),round(keypoints[k].pt[1]),round(keypoi
nts[k].size))

cv2.waitKey(0)
cv2.destroyAllWindows()
```

VideoBlobsIdentifiedFeb24.py

```
import numpy as np
import matplotlib.pyplot as plt
import cv2
import time
from collections import Counter

class Blob_Detector:

    def __init__(self, params = 0):
        self.params = params

    def initialize_defaults(self, params):
        # Define the default values of the blob parameters.
        print("Default Minimum Threshold: " +
str(params.minThreshold))
        print("Default Maximum Threshold: " + str(params.maxThreshold)
+ "\n")

        print("Default Minimum Area: " + str(params.minArea))
        print("Default Maximum Area: " + str(params.maxArea) + "\n")

        print("Default Circularity: " + str(params.minCircularity) +
"\n")

        print("Default Convexity: " + str(params.minConvexity) + "\n")

        print("Default Inertia Ratio: " + str(params.minInertiaRatio)
+ "\n")

    def adjust_parameters(self, params):
        # Change thresholds
        params.minThreshold = 40
        params.maxThreshold = 200
        params.thresholdStep = 20
        # Filter by Area.
        params.filterByArea = True
        params.minArea = 8
        params.maxArea = 40
        # Filter by Circularity
        params.filterByCircularity = True
        params.minCircularity = 0.85
        params.filterByConvexity = False
        # Filter by Inertia
```

```
        params.filterByInertia = False

class Blob_Id_Debug:

    # Initialize the variables for the main class
    def __init__(self, j = 1, k = 0, m = 0, n = 0, matches = [], desc
= [], currentNames = [],
                 framesToCheck = [], unmatchedBlobs = [], kpList = [],
frameBlobs = []):
        self.j = j
        self.k = k
        self.m = m
        self.n = n
        self.matches = matches
        self.desc = desc
        self.currentNames = currentNames
        self.framesToCheck = framesToCheck
        self.unmatchedBlobs = unmatchedBlobs
        self.kpList = kpList
        self.frameBlobs = frameBlobs

    # Define three fucntions: first frame cycle, frame check for
unmatched
    # blobs, and second frame cycle.
    def frame_cycle(self, j, k, m, n, matches, desc, currentNames):
        for j in range(1, m):
            bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
            matches = bf.match(desc[j],desc[j-1]);
            print("\nFRAME #"+str(j+1)+":")
            for k in range(0, n):
                print('Raw match frame',j+1,' to
frame',j,matches[k].queryIdx,matches[k].trainIdx,' Distance
',matches[k].distance)
                print('\n--> Current Frame',j+1,'Compare
Frame',j,'Names',currentNames)

    def frame_to_check(self, framesToCheck, unmatchedBlobs):
        print('Frames to test: ',framesToCheck)
        print('Blobs to identify: ',unmatchedBlobs)
        print('Using Frame: ',framesToCheck[0])

    def frame_cycle_two(self, j, k, m, n, matches, desc,
framesToCheck, kpList, frameBlobs):
        for j in range(1, m):
            bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
            matches = bf.match(desc[j],desc[framesToCheck[0]-1])
```

```
            for k in range(0, n):
                print('Raw match frame',j+1,' to
frame',framesToCheck[0],matches[k].queryIdx,matches[k].trainIdx,'
Distance ',matches[k].distance)
                print('Found missing blob',matches[k].queryIdx,' as
match',k)
            print('Frame', j+1,' has',len(kpList[j]),' keypoints,
labeled:',frameBlobs[j])

    # Indicate whether or not any function can be executed in the main
source code.
    def __str__(self):
        if (self.frame_cycle() == True):
            return str(self.j, self.k, self.m, self.n, self.matches,
self.desc, self.currentNames)
        if (self.frame_to_check() == True):
            return str(self.framesToCheck, self.unmatchedBlobs)
        if (self.frame_cycle_two() == True):
            return str(self.j, self.k, self.m, self.n, self.matches,
self.desc, self.framesToCheck, self.kpList, self.frameBlobs)
        else:
            return False

class Blob_Frequency:

    def __init__(self, i = 0, j = 1, frameBlobs = []):
        self.i = i
        self.j = j
        self.frameBlobs = frameBlobs

    def counter(self, i, j, frameBlobs):
        blobFreq = frameBlobs[0]
        for i in range(0,j):
            blobFreq.extend(frameBlobs[i])
        print(Counter(blobFreq))

    def __str__(self):
        return str(self.i, self.j, self.frameBlobs)
```

VideoBlobsIdentifiedFinal.py

```
from VideoBlobsIdentifiedFeb24 import Blob_Detector, Blob_Id_Debug,
Blob_Frequency
import numpy as np
import matplotlib.pyplot as plt
import cv2
import csv
import math
from math import sqrt
from itertools import zip_longest
import time
from collections import Counter
import tkinter as tk
from tkinter import filedialog
from tkinter import messagebox
from tkinter import simpledialog

displayFrame = False
displayCorrect = True;
displayOriginal = not displayCorrect;
frameDivider = 10 #109;  # Number of frames between display
classifierThreshold = 50
debug1 = False
debug2 = False
debug3 = False
debug4 = False

# Can we change these thresholds that are tailored to each blob?
xPixelThreshold = 20 #185 # 25
yPixelThreshold = 25 #10000 # 10
root = tk.Tk()
root.withdraw()
application_window = tk.Tk()
application_window.withdraw()

## BLOB DETECTOR SUB-ROUTINE ##
# Setup SimpleBlobDetector parameters.
params = cv2.SimpleBlobDetector_Params()
blobSubRoutine = Blob_Detector()
blobSubRoutine.adjust_parameters(params)

# Create a detector with the parameters
detector = cv2.SimpleBlobDetector_create(params)
```

```
# Create a descriptor generator
descriptor = cv2.xfeatures2d.BriefDescriptorExtractor_create()
# create BFMatcher object
bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)

# Get file from dialog
file_path = filedialog.askopenfilename(initialdir = ".",title =
"Select file",filetypes = (("jpeg files","*.avi"),("all
files","*.*")))
cap = cv2.VideoCapture(file_path)
#outP = open("testOutNoRound.txt","w+")
initialFrame = 0
#initialFrame = int(input('Starting Video Frame: '))-1;
#initialFrame = 1
endFrame = int(input('Final Video Frame: '));
#endFrame = 120;
frameNo = 0;
ret = cap.set(cv2.CAP_PROP_POS_FRAMES,initialFrame)
#print(ret)
kpList = [];
desc = [];
images = [];
while cap.isOpened():
    if (frameNo >= endFrame): break
    ret, frame = cap.read()
    if ret:
        frameNo = frameNo + 1
        if (frameNo > initialFrame):
    # Detect Blobs
            kpList.append(detector.detect(frame));
            keypoints = kpList[frameNo-1]
            keypoints, desc1 = descriptor.compute(frame,keypoints)
            desc.append(desc1);
    # Draw detected blobs as red circles.
    # cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS ensures
    # the size of the circle corresponds to the size of blob
        images.append(frame)
    else:
        cap.release()

# create BFMatcher object
# bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
# Classify blobs.
# Start the name list
blobNames = [];
# Make a list of lists to keep track of the blobs in each frame
```

```
frameBlobs = [];
# Keep track of next blob name to use
keypoints = kpList[0]
nextBlob = len(keypoints);
# print('Frame 1 has',len(keypoints),' total keypoints'); # Note that
frame counter starts at zero
#print('No matches possible')
#print('The next keypoint will be ',nextBlob)
blobNames = [i for i in range(0,nextBlob)];
# Note the [:] to get the current values, rather than the list
blobNames
# because the list itself will continue to change
frameBlobs.append(blobNames[:]); # This will only work for first frame
blobLastSeen = [1 for i in range(0,nextBlob)];
### print('All blobs in Frame',1,' are:',frameBlobs[0])

im_with_keypoints = cv2.drawKeypoints(images[0], keypoints,
np.array([]), (0,0,255), cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
for k in range(0,len(keypoints)):
    cx = round(keypoints[k].pt[0])
    cy = round(keypoints[k].pt[1])
    cv2.putText(im_with_keypoints, str(k), (cx, cy),
cv2.FONT_HERSHEY_SIMPLEX, .6,(0, 0, 255))
windowName = 'Frame 1'
if displayFrame:
    cv2.imshow(windowName, im_with_keypoints)
### This next step is necessary to force a draw.
cv2.waitKey(2)
##    if (debug1 == True):
##        debug1 = Blob_Id_Debug()
##        debug1.fc = Blob_Id_Debug().Frame_Count()
# Start cycling through the frames
for j in range(1,endFrame): # j is current frame index, so frameNo =
j+1
    kp1 = kpList[j-1]; # starts at kpList[0], frame 1
    kp2 = kpList[j]; # starts at kpList[1], frame 2
# Match descriptors.
    matches = bf.match(desc[j],desc[j-1]);
    # Check out https://opencv-python-
tutroals.readthedocs.io/en/latest/py_tutorials/py_feature2d/py_matcher
/py_matcher.html
    unmatchedBlobs = [i for i in range(0,len(kp2))];
# Create old and new Name lists
 #   currentBlobs = [i for i in range(0,len(kp2))]
    currentNames = [-1 for i in range(0,len(kp2))];
# Now we add the matched blobs
```

```
    for k in range(0,len(matches)):
        matchxDist = (abs(kp1[matches[k].trainIdx].pt[0] -
kp2[matches[k].queryIdx].pt[0]) < xPixelThreshold)
        matchyDist = (abs(kp1[matches[k].trainIdx].pt[1] -
kp2[matches[k].queryIdx].pt[1]) < yPixelThreshold)
        # matchxDist = True
        # matchyDist = True
        matchTest = (matches[k].distance < classifierThreshold) and
matchxDist and matchyDist
        # matchTest = matchTest and not (frameBlobs[j-
1][matches[k].trainIdx] in currentNames)
        if (matchTest):
            # print('Raw match frame',j+1,' to
frame',j,matches[k].queryIdx,matches[k].trainIdx,' Distance
',matches[k].distance)
            # print('Corrected
Match',k,currentNames[matches[k].queryIdx],frameBlobs[j-
1][matches[k].trainIdx])
            currentNames[matches[k].queryIdx] = frameBlobs[j-
1][matches[k].trainIdx];
            blobLastSeen[frameBlobs[j-1][matches[k].trainIdx]] = j+1;
            unmatchedBlobs.remove(matches[k].queryIdx)
        if (debug1 == True):
            debug1 = Blob_Id_Debug()
            print(debug1.frame_cycle(j, k, endFrame, (len(matches)-1),
matches, desc, currentNames[matches[k].queryIdx]))

# Next, we add matches to the other frames
# Collect all frames with unmatched blobs
# set gives no duplicates
    framesToCheck = set(blobLastSeen)
# Remove the current frame and the previous one
    framesToCheck.discard(j+1)
    framesToCheck.discard(j)
# reverse order to check the most recent frames first
# Also make it a list so you can use subscripts
    framesToCheck = list(sorted(framesToCheck, reverse = True))
# Now, do the checks when there are frames and unmatched blobs
    while ((len(framesToCheck) > 0) and(len(unmatchedBlobs) > 0)):
        kp1 = kpList[framesToCheck[0]-1];
        if (debug2 == True):
            print('Frames to test: ',framesToCheck)
            print('Blobs to identify: ',unmatchedBlobs)
            print('Using Frame: ',framesToCheck[0])
# Match descriptors.
        matches = bf.match(desc[j],desc[framesToCheck[0]-1])
```

```
# Finally, go through the matches looking to match blobs still
unidentified
        for k in range(0,len(matches)):
            #matchxDist = (abs(kp1[matches[k].trainIdx].pt[0] -
kp2[matches[k].queryIdx].pt[0]) < xPixelThreshold)
            matchxDist = (kp1[matches[k].trainIdx].pt[0] -
kp2[matches[k].queryIdx].pt[0] < xPixelThreshold)
            matchyDist = (abs(kp1[matches[k].trainIdx].pt[1] -
kp2[matches[k].queryIdx].pt[1]) < yPixelThreshold)
            matchxDist = True
            # matchyDist = True
            matchTest = (matches[k].distance < classifierThreshold)
and matchxDist and matchyDist
            matchTest = matchTest and not
(frameBlobs[framesToCheck[0]-1][matches[k].trainIdx] in currentNames)
            if (matchTest):
                # print('Raw match frame',j+1,' to
frame',framesToCheck[0],matches[k].queryIdx,matches[k].trainIdx,'
Distance ',matches[k].distance)
                if (unmatchedBlobs.count(matches[k].queryIdx) > 0):
                    currentNames[matches[k].queryIdx] =
frameBlobs[framesToCheck[0]-1][matches[k].trainIdx];
                    blobLastSeen[currentNames[matches[k].queryIdx]] =
j+1;
###                    print('Found missing
blob',matches[k].queryIdx,' as match',k)
                    unmatchedBlobs.remove(matches[k].queryIdx)
                    if (debug3 == True):
                        debug3 = Blob_Id_Debug()
                        print(debug3.frame_cycle_two(j, k, endFrame,
(len(matches)-1), matches, desc, framesToCheck, kpList, frameBlobs))

# Now finished this frame
        del(framesToCheck[0])
# Finally, create new names for the unmatched blobs
###########
#  Problem is here
# If an early identification is wrong
# We are not checking to see if a later match is better
# For example, frame 7 kp 6 gets matched to frame 6 kp 0 (called 0)
# But also frame 7 kp 6 gets matched to frame 5 kp 6 (called 5)
# The second match is better
#########
#########
#  Possible solution:
# Keep a vector of distances along with names
```

```
# If a later match has a smaller distance, use it
#
#########
    for k in range(0,len(unmatchedBlobs)):
        positionToFill = unmatchedBlobs[k];
        currentNames[positionToFill] = nextBlob;
        nextBlob +=1
        blobNames.append(unmatchedBlobs[k]);
        blobLastSeen.append(j+1); # Frame number, not index

# Next we add the whole list to the frame enumerator
    frameBlobs.append(currentNames);

    keypoints = kp2
# This shows the original keypoint labels
    im_with_keypoints = cv2.drawKeypoints(images[j], keypoints,
np.array([]), (0,0,255), cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
    for k in range(0,len(keypoints)):
        cx = round(keypoints[k].pt[0])
        cy = round(keypoints[k].pt[1])

        # Frame number, blob number, x-coord, y-coord
        # print(j+1, currentNames[k], cx, cy)

##            for k in range(0, len(matches)):
##                print("Frame:",j+1, j, currentNames[k], cx, cy,
matches[k].distance)
##                print()
        # cv2.putText(im_with_keypoints, str(frameBlobs[j][k]), (cx,
cy), cv2.FONT_HERSHEY_SIMPLEX, .6,(0, 0, 255))
# This shows the original keypoint labels
        if displayOriginal: cv2.putText(im_with_keypoints, str(k),
(cx, cy), cv2.FONT_HERSHEY_SIMPLEX, .6,(0, 0, 255))
### This shows the corrected keypoint labels
        if displayCorrect: cv2.putText(im_with_keypoints,
str(frameBlobs[j][k]), (cx, cy), cv2.FONT_HERSHEY_SIMPLEX, .6,(0, 0,
255))
    windowName = 'Frame '+str(j+1)
    windowName2 = 'testFrame'+str(j+1)+'.jpg'
# The next line shows frame
    if displayFrame:
        if (j%frameDivider == 0):
            cv2.imshow(windowName, im_with_keypoints)
            cv2.waitKey(2)
# This next step is necessary to force a draw.
##    if (j == 34):
```

```
##        displayFrame = True
##        frameDivider = 1
##        windowName = 'Frame '+str(j+1)
##        cv2.imshow(windowName, im_with_keypoints)
##        cv2.waitKey(2)
##    elif (j == 44):
##        displayFrame = False
##        cv2.waitKey(2)
##    if (j == 39):
##        displayFrame = True
##        frameDivider = 1
##        windowName = 'Frame '+str(j+1)
##        cv2.imshow(windowName, im_with_keypoints)
##        cv2.waitKey(2)
##    elif (j == 49):
##        displayFrame = False
##        cv2.waitKey(2)
    #print('Frame',j+1,currentNames)

# Background Drawings for Trajectories
topblock_x = [129,124,101]
topblock_y = [675,651,651]

sideblock_x = [129,106,106,129,129]
sideblock_y = [675,675,723,723,675]

faceblock_x = [106,101,101]
faceblock_y = [723,699,651]

mat_x = [0,  101,101,106,129,129,175,175,-5, -5, 0]
mat_y = [680,680,699,723,723,680,680,725,725,680,680]

kickplate_x = [-5,175,175,-5,-5]
kickplate_y = [650,650,640,640,650]

wall_x = [-5,175,175,-5,-5]
wall_y = [640,640,490,490,640]

hardwood_x = [-5,101,101,124,129,129,175,175,-5,-5]
hardwood_y = [680,680,651,651,675,680,680,650,650,680]

floor2_x = [175,-5, -5, 175, 175]
floor2_y = [1060, 1060, 725, 725, 1060]

# LEFT HEEL
BlobPxLH = [];
```

```
BlobPyLH = [];
BlobPfLH = [];
StableBlobxfLH=[];
StableBlobyfLH=[];

for freqBlob in [7, 79]:
    for j in range(1,endFrame):
        if (freqBlob in frameBlobs[j]):
            keypoints = kpList[j]
        # Print info on the frame number and blob coordinates to be
graphed.
        # print(str(j+1)+', '+str(round(keypoints[k].pt[0]))+',
'+str(round(keypoints[k].pt[1])))

        # Take the standard deviation of the data between Frames 60
and 90 for Blob 7,
        # and determine if it is small enough to detect the flat
regions of both
        # the X and Y trajectories.

        # Find the average of x-coordinates across the points between
frames 60 and 90.
        # Then do the same for y-coordinates
        # Compute the standard deviations and create logic for if
        # the result is less than 1 across 30 frames, for example,
        # the computer can detect that the shoe is on the ground.
            k = frameBlobs[j].index(freqBlob)
            BlobPfLH.append(j+1);
            BlobPxLH.append(round(keypoints[k].pt[0]));
            BlobPyLH.append(round(keypoints[k].pt[1]));

# At this point, BlobP? has information for left heel
# for i in range(0, len(BlobPf)):
fig, ax1 = plt.subplots()
ax1.set_title('Left Heel (X)')
ax1.set_xlabel('Frame Number')
ax1.set_ylabel('x-coordinate')
for i in range(0, len(BlobPfLH)):
    ax1.plot(BlobPfLH[i],BlobPxLH[i],'ro')
ax1.set_xlim(endFrame, 0)

fig, ax2 = plt.subplots()
ax2.set_title('Left Heel (Y)')
ax2.set_xlabel('Frame Number')
ax2.set_ylabel('y-coordinate')
```

```
#ax2.axvspan(125, 115, ymin=0.045, ymax=0.215, alpha=1, color='teal')
ax2.plot(topblock_x, topblock_y, linewidth=2, color='teal')
ax2.plot(sideblock_x, sideblock_y, linewidth=2, color='teal')
ax2.plot(faceblock_x, faceblock_y, linewidth=2, color='teal')

ax2.fill_between(hardwood_x,
hardwood_y,0,facecolor='sandybrown',color='sandybrown',alpha=0.2)
ax2.fill_between(mat_x,
mat_y,0,facecolor='lightcoral',color='lightcoral',alpha=0.2)
ax2.fill_between(kickplate_x,
kickplate_y,0,facecolor='white',color='gainsboro',alpha=0.2)
ax2.fill_between(wall_x,
wall_y,0,facecolor='white',color='gainsboro',alpha=0.2)
for i in range(0, len(BlobPfLH)):
    ax2.plot(BlobPfLH[i],BlobPyLH[i],'bo')
ax2.set_xlim(endFrame, 0)
ax2.set_ylim(725, 575)

for k2 in range(0, len(BlobPfLH)-10):
##        print(k2, np.std(Blob7x[k2:(k2+10)]))
    BPstdX = np.std(BlobPxLH[k2:(k2+10)])
    BPstdY = np.std(BlobPyLH[k2:(k2+10)])
    if (BPstdY < 0.1) and (BPstdX < 0.1):
        #print("Desired SD at Y found at frame", BlobPf[k2])
        StableBlobyfLH.append(BlobPfLH[k2]);
        StableBlobxfLH.append(BlobPfLH[k2]);
        #print(StableBlobxf)
        #print("Desired SD at X found at frame", BlobPf[k2])
    # [SOLVED] goodEnd turns out to be a float number, which is not
iterable.
    # Since the common end frame is supposed to be 81, we will have to
take
    # out the + 10
goodEndLH = min(max(StableBlobxfLH),max(StableBlobyfLH))
goodStartLH = max(min(StableBlobxfLH),min(StableBlobyfLH))
print("***FOR LEFT HEEL***")
print("ENTER stable region! Start frame:",goodStartLH)
print("LEAVE stable region! End frame:",goodEndLH)
##print("\n***STANDARD DEVIATIONS FOR Y TRAJECTORY***")

# If put into a for loop, the mean coordinate point at SD will repeat
multiple times.
for i in range(0, len(BlobPfLH)-10):
    k3 = i
print("Mean coordinate point at desired SD = (", BlobPxLH[k2],",",
BlobPyLH[k3],")")
```

```
total_listX = [BlobPfLH, BlobPxLH]
total_listY = [BlobPfLH, BlobPyLH]
export_dataX = zip_longest(*total_listX, fillvalue = '')
export_dataY = zip_longest(*total_listY, fillvalue = '')
try:
    with open('leftheel_trajectoryX.csv', 'w', newline='') as f1:
        wrX = csv.writer(f1)
        # column labels: "Frame, X"
        wrX.writerows(export_dataX)
    f1.close()
except PermissionError as e:
    print("File already open!")
try:
    with open('leftheel_trajectoryY.csv', 'w', newline='') as f2:
        wrY = csv.writer(f2)
        # column labels: "Frame, Y"
        wrY.writerows(export_dataY)
    f2.close()
except PermissionError as e:
    print("File already open!")

# END OF LEFT HEEL #

# BEGIN LEFT TOE #
BlobPxLT = []
BlobPyLT = [];
BlobPfLT = [];
StableBlobxfLT = []
StableBlobyfLT = []

for freqBlob in [17, 5, 87]:
    for j in range(1,endFrame):
        if (freqBlob in frameBlobs[j]):
            keypoints = kpList[j]
            k = frameBlobs[j].index(freqBlob)
            BlobPxLT.append(round(keypoints[k].pt[0]));
            BlobPfLT.append(j+1);
            BlobPyLT.append(round(keypoints[k].pt[1]));

fig, axx = plt.subplots()
axx.set_title('Left Toe (X)')
axx.set_xlabel('Frame Number')
axx.set_ylabel('x-coordinate')
for i in range(0, len(BlobPfLT)):
    axx.plot(BlobPfLT[i],BlobPxLT[i],'ro')
```

```
axx.set_xlim(endFrame, 0)

fig, axy = plt.subplots()
axy.set_title('Left Toe (Y)')
axy.set_xlabel('Frame Number')
axy.set_ylabel('y-coordinate')
#axy.axvspan(125, 115, ymin=0.0667, ymax=0.3125, alpha=1,
color='teal')
axy.plot(topblock_x, topblock_y, linewidth=2, color='teal')
axy.plot(sideblock_x, sideblock_y, linewidth=2, color='teal')
axy.plot(faceblock_x, faceblock_y, linewidth=2, color='teal')
axy.fill_between(hardwood_x,
hardwood_y,0,facecolor='sandybrown',color='sandybrown',alpha=0.2)
axy.fill_between(mat_x,
mat_y,0,facecolor='lightcoral',color='lightcoral',alpha=0.2)
axy.fill_between(kickplate_x,
kickplate_y,0,facecolor='white',color='gainsboro',alpha=0.2)
axy.fill_between(wall_x,
wall_y,0,facecolor='white',color='gainsboro',alpha=0.2)
for i in range(0, len(BlobPfLT)):
    axy.plot(BlobPfLT[i],BlobPyLT[i],'bo')
axy.set_xlim(endFrame, 0)
axy.set_ylim(725, 565)

for k2 in range(0, len(BlobPfLT)-10):
##        print(k2, np.std(Blob7x[k2:(k2+10)]))
    BPstdX = np.std(BlobPxLT[k2:(k2+10)])
    BPstdY = np.std(BlobPyLT[k2:(k2+10)])
    if (BPstdY < 0.1) and (BPstdX < 0.1):
        #print("Desired SD at Y found at frame", BlobPf[k2])
        StableBlobyfLT.append(BlobPfLT[k2]);
        StableBlobxfLT.append(BlobPfLT[k2]);
goodEndLT = min(max(StableBlobxfLT),max(StableBlobyfLT))
goodStartLT = max(min(StableBlobxfLT),min(StableBlobyfLT))
print("***FOR LEFT TOE***")
print("ENTER stable region! Start frame:",goodStartLT)
print("LEAVE stable region! End frame:",goodEndLT)
##print("\n***STANDARD DEVIATIONS FOR Y TRAJECTORY***")

# If put into a for loop, the mean coordinate point at SD will repeat
multiple times.
for i in range(0, len(BlobPfLT)-10):
    k3 = i
print("Mean coordinate point at desired SD = (", BlobPxLT[k2],",",
BlobPyLT[k3],")")
```

```
total_listX = [BlobPfLT, BlobPxLT]
total_listY = [BlobPfLT, BlobPyLT]
export_dataX = zip_longest(*total_listX, fillvalue = '')
export_dataY = zip_longest(*total_listY, fillvalue = '')
try:
    with open('lefttoe_trajectoryX.csv', 'w', newline='') as f1:
        wrX = csv.writer(f1)
        # column labels: "Frame, X"
        wrX.writerows(export_dataX)
    f1.close()
except PermissionError as e:
    print("File already open!")
try:
    with open('lefttoe_trajectoryY.csv', 'w', newline='') as f2:
        wrY = csv.writer(f2)
        # column labels: "Frame, Y"
        wrY.writerows(export_dataY)
    f2.close()
except PermissionError as e:
    print("File already open!")
# END OF LEFT TOE #

if (debug4 == True):
    debug4 = Blob_Frequency()
    debug4.counter(i, endFrame, frameBlobs)
        # print("No Blob #"+str(v)+" coordinates exist for
Frame",j+1,"\n")

plt.show(block=False)
```

Bibliography

[1] K. Chaccour, R. Darazi, A. H. El Hassani, and E. Andrès, "From Fall Detection to Fall Prevention: A Generic Classification of Fall-Related Systems," *IEEE Sensors Journal*, vol. 17, no. 3, pp. 812–822, Feb. 2017.

[2] "WHO Global Report on Falls Prevention in Older Age." *World Health Organization*, World Health Organization, 7-Mar-2008. [PDF report]. Available: https://www.who.int/ageing/publications/Falls_prevention7March.pdf

[3] J. Berglund, "A Balancing Act: Scientists Seek to Reduce the Risk of Falls in the Elderly," in *IEEE Pulse*, vol. 8, no. 2, pp. 21-24, March-April 2017.

[4] "Falls." *World Health Organization*, World Health Organization, 16-Jan-2018. [Online]. Available: www.who.int/news-room/fact-sheets/detail/falls.

[5] N. El-Bendary, Q. Tan, F. C. Pivot, A. Lam, "Fall detection and prevention for the elderly: A review of trends and challenges", *Int. J. Smart Sens. Intell. Syst.*, vol. 6, no. 3, pp. 1230-1266, Jun. 2013.

[6] M. Deschenes, "New Falls Prevention Program," *William & Mary*, 03-Sep-2018. [Online]. Available: https://www.wm.edu/as/kinesiology/news/new-falls-prevention-program.php.

[7] E.N. Burnet, M. Deschenes, "The Center for Balance and Aging Studies (CBAS)," *William & Mary*, 2018. [Online]. Available: https://www.wm.edu/as/kinesiology/research/lab-pages/cbas/index.php.

[8] J. McClain, "W&M Center for Balance & Aging Studies aims to reduce falls & injuries among the elderly," *William & Mary*, 08-Apr-2019. [Online]. Available:

https://www.wm.edu/news/stories/2019/wm-center-for-balance-aging-studies-aims-to-reduce-falls-injuries-among-the-elderly.php.

[9] M.W. Whittle, *Gait Analysis: An Introduction*, 2nd ed. Woburn, MA: Butterworth-Heinemann, 1996, pp. 106.

[10] E. Burnet, M. Deschenes, R. McCoy, *S10.B1.* Williamsburg Landing, Williamsburg, Virginia.

[11] K.W. Bowyer, J. Phillips, *Empirical Evaluation Techniques in Computer Vision*. Los Alamitos, CA: IEEE, 1998, pp. 1.

[12] "Introduction to OpenCV-Python Tutorials," *OpenCV.* [Online]. Available: https://docs.opencv.org/master/d0/de3/tutorial_py_intro.html

[13] "About," *OpenCV.* [Online]. Available: https://opencv.org/about/.

[14] "OpenCV-Python Tutorials," *OpenCV.* [Online]. Available: https://docs.opencv.org/master/d6/d00/tutorial_py_root.html.

[15] E. Rosten, T. Drummond, "Machine learning for high-speed corner detection," *In Proceedings of the 9th European conference on Computer Vision - Volume Part I (ECCV'06),* Berlin, 2006, pp. 430-443, doi: 10.1007/11744023_34.

[16] "OpenCV: FAST Algorithm for Corner Detection." *OpenCV.* [Online]. Available: https://docs.opencv.org/3.4/df/d0c/tutorial_py_fast.html.

[17] "OpenCV: ORB (Oriented FAST and Rotated BRIEF)." *OpenCV*. [Online]. Available: https://docs.opencv.org/3.4/d1/d89/tutorial_py_orb.html.

[18] E. Rublee, V. Rabaud, K. Konolige and G. Bradski, "ORB: An efficient alternative to SIFT or SURF," *2011 International Conference on Computer Vision*, Barcelona, 2011, pp. 2564-2571, doi: 10.1109/ICCV.2011.6126544.

[19] "OpenCV: Canny Edge Detection." *OpenCV.* [Online]. Available: https://docs.opencv.org/3.4/da/d22/tutorial_py_canny.html.

[20] "OpenCV: Hough Line Transform." *OpenCV.* [Online]. Available: https://docs.opencv.org/3.4/d6/d10/tutorial_py_houghlines.html.

[21] "cv::SimpleBlobDetector Class Reference," *OpenCV*. [Online]. Available: https://docs.opencv.org/3.4/d0/d7a/classcv_1_1SimpleBlobDetector.html

[22] S. Mallick, "Blob Detection Using OpenCV (Python, C++)," *Learn OpenCV,* 17-Feb-2015. [Online]. Available: https://www.learnopencv.com/blob-detection-using-opencv-python-c/.

[23] "cv::xfeatures2d::BriefDescriptorExtractor Class Reference," *OpenCV*. [Online]. Available: https://docs.opencv.org/3.4/d1/d93/classcv_1_1xfeatures2d_1_1BriefDescriptorExtractor.html.

[24] R.B. Fisher, K. Dawson-Howe, A. Fitzgibbon, C. Robertson, E. Trucco, *Dictionary of Computer Vision and Image Processing.* Hoboken, NJ: Wiley, 2005, pp. 117.

[25] "cv::BFMatcher Class Reference," *OpenCV.* [Online]. Available: https://docs.opencv.org/3.4/d3/da1/classcv_1_1BFMatcher.html.

Vita

Martha Gizaw was born in Abilene, Texas, in 1996 after her parents immigrated from Addis Ababa, Ethiopia. After her high school graduation in 2015, she began her post-secondary studies at Northern Virginia Community College as an honors student. She then transferred to the College of William & Mary in 2017 with an associate degree in science.

In May 2020, Gizaw graduated with a self-designed bachelor's degree in biomedical engineering and a minor in computer science. Her plans after college included, but were not limited to, taking a gap year to sit for the Graduate Record Examinations and apply for graduate programs in engineering and applied science. At the time of publication, she lived with her mother and two younger siblings in Woodbridge, Virginia.

© 2020 Martha T. Gizaw. All rights reserved.