Detection of Launch Frame in Long Jump Videos Using Computer Vision and Discreet Computation

by

Pablo E. Muniz

Submitted to the
Department of Mechanical Engineering
In Partial Fulfillment of the Requirements for the Degree of

Bachelor of Science in Mechanical Engineering
at the
Massachusetts Institute of Technology

June 2019

Signature redacted

Signature of Author: ______________________________

Department of Mechanical Engineering
May 10, 2019

Signature redacted

Certified by: ______________________________

Anette Hosoi
Associate Dean of Engineering/Professor
Thesis Supervisor

Signature redacted

Accepted by: ______________________________

Maria Yang
Associate Professor of Mechanical Engineering
Undergraduate Officer

Detection of Launch Frame in Long Jump Videos Using Computer Vision and Discreet Computation

by

Pablo E. Muniz

Submitted to the Department of Mechanical Engineering
On May 10, 2019 in Partial Fulfillment of the
Requirements for the Degree of

Bachelor of Science in Mechanical Engineering

ABSTRACT

Pose estimation, a computer vision technique, can be used to develop a quantitative feedback training tool for long jumping. Key performance indicators (KPIs) such as launch velocity would allow a long jumping athlete to optimize their technique while training. However, these KPIs need a prior knowledge of when the athlete jumped, referred to as the launch frame in the context of videos and computer vision. Thus, an algorithm for estimating the launch frame was made using the OpenPose Demo and Matlab. The algorithm estimates the launch frame to within 0.8±0.91 frames. Implementing the algorithm into a training tool would give an athlete real-time, quantitative feedback from a video. This process of developing an algorithm to flag an event can be used in other sports as well, especially with the rise of KPIs in the sports industry (e.g. launch angle and velocity in baseball).

Thesis Supervisor: Anette Hosoi
Title: Associate Dean of Engineering/Professor

Acknowledgements

The author would like to thank Prof. Anette (Peko) Hosoi for her indispensable guidance throughout the term of this project. He would also like to thank Phil Cheetham for advice regarding long jumping and for allowing the author to use the USOC's training videos. Additional thanks to Golda Nguyen and Jeremy Storming for their work with OpenPose and gathering the data for this application.

Table of Contents

List of Figures

List of Tables

1. Introduction

Long jumping is a track-and-field sport that consists of a horizontal jump following a running start with the goal of jumping the farthest. An Olympic sport since antiquity, it was part of the 708 BC Olympic games and it is still practiced to this day [1]. Long jumping has become more competitive in recent decades since professional Olympic committees have become the norm, leading to more accurate selection of the best athletes. As shown in Figure 1-1, technological advancements have increased the efficacy of training regiments, pushing athletes' bodies to their limit and lengthening world records. Nevertheless, there is still a dearth of quantitative, external feedback training tools [3]. This can be filled with the advent of computer vision programs.

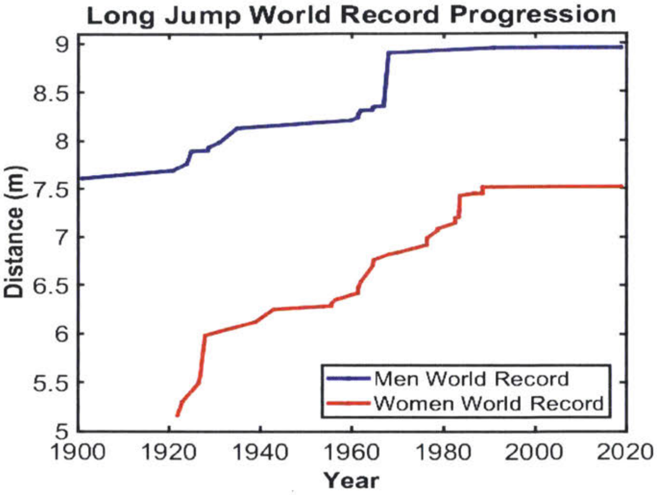


Figure 1-1: Long jumping world record progression since 1900. As in most sports, long jumping athletes have become more proficient with more efficient training, increasing the sport's competitiveness. Data. [2]

Computer Vision is a branch of computer science that attempts to give computers the ability to *see* videos and pictures. OpenPose, a computer vision program developed in Carnegie Mellon University, does pose estimation, allowing a computer to estimate the position of 25 key body parts. This program was used to acquire the data for this application. The KPIs representative of success in long jumping are velocity and angle when jumping: *launch velocity* and *launch angle,* which are calculated from the frame in the video where the athlete starts jumping, referred to as the *launch frame*. We develop an algorithm that provides an accurate estimate of the launch frame using six of these discreet time signals (left/right ankle, heel, and toe) in order to make these KPI calculations automatically and accurately.

2. Feedback, OpenPose, and Long Jumping

2.1 Feedback While Training

Traditionally, feedback in sports has taken the form of a coach's verbal feedback based on qualitative results or their own perception of an athlete's performance. This process is inherently qualitative and is dependent on a coach's ability to notice and communicate what they see. However, the development of sensory instrumentation has allowed athletes and their coaches to study their bodies' kinematics in newly quantitative terms. Along with an increased understanding of biomechanics, athletes have been able to use this data to improve their techniques.

In addition, athletes have benefitted from having access to this data in real-time (e.g. heart-rate monitor, accelerometers, etc.). A study by Anderson, Harrison, and Lyons (2005) tested the benefits of real-time feedback and summary feedback on 13 experienced rowers. They concluded that real-time feedback significantly increased the consistency of the rowers' technique over no feedback. In addition, summary feedback also provided more consistent results than training with no feedback [4]. A similar study by Eriksson, M., Halvorsen, K. A., and Gullstrand, L. (2011) tested how real-time feedback affected 18 trained runners' ability to maintain a consistently optimal running form. In almost all cases, the athlete was successful in correcting their running form [5]. Although quantitative external feedback can be invaluable to an athlete, it has to be accurate and relevant, appropriately timed, and decipherable to the athlete [6]. Therefore, key performance indicators (KPIs) that are indicative of good performance should be used to simplify the feedback data, ensuring these three conditions are met.

2.2 OpenPose and Computer Vision

Although sensory instrumentation has become smaller and lighter over time, it can still inhibit an athlete's movements and affect their performance. In order to optimize a technique for competition, it is important to recreate a competition's condition (i.e. no sensors). Therefore, there is a need for non-invasive quantitative feedback systems such as computer vision programs coupled with discreet-data algorithms. By giving the computer the ability to *see* how a person is moving, we can give it the ability to compute long jumping's KPIs by simply processing a video.

OpenPose was used to estimate the location of a long jumper's key body parts in a video frame. As seen in Figure 2-1, the OpenPose demo outputs the approximate location of 25 key body parts in each of the video's frames. The 10 videos used as data to develop this application were from this perspective: to the side of the athlete.

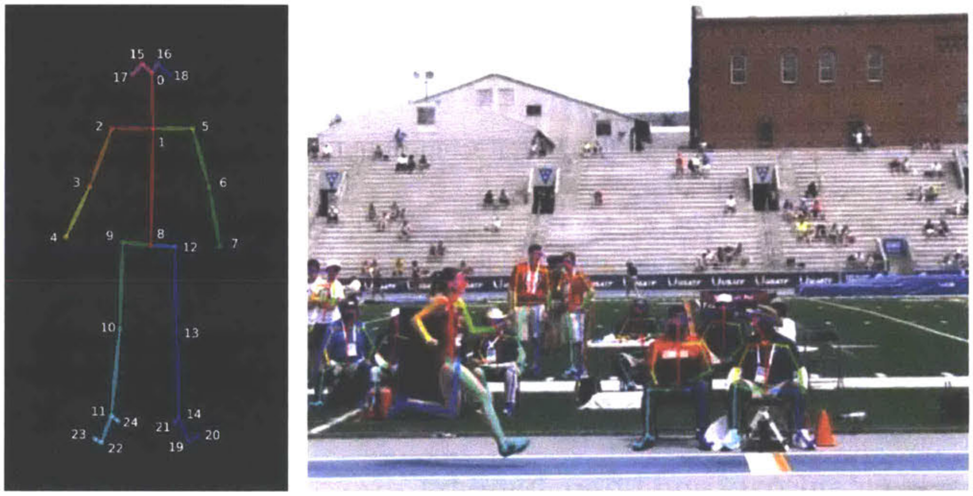


Figure 2-1: *(left) The 25 signals tracked by the OpenPose demo are shown [7]. We will use the foot coordinates for this application, but the rest can be used to estimate the center of mass's location. (right) Example of how the OpenPose demo works on one of our ten long jumping videos*

For general pose detection, OpenPose creates a 3-point vector of *<x, y, confidence level>* for each body part detected (total of 25 body parts detected), and the parts are ordered and named in the output file as follows [7]:

0) Nose
1) Neck
2) Right Shoulder (RShoulder)
3) Right Elbow (RElbow)
4) Right Wrist (RWrist)
5) Left Shoulder (LShoulder)
6) Left Elbow (LElbow)
7) Left Wrist (LWrist)
8) Mid Hip
9) Right Hip (RHip)
10) Right Knee (RKnee)
11) Right Ankle (RAnkle)
12) Left Hip(LHip)
13) Left Knee (LKnee)
14) Left Ankle (LAnkle)
15) Right Eye (REye)
16) Left Eye (LEye)
17) Right Ear (REar)
18) Left Ear (LEar)
19) Left Big Toe (LBigToe)
20) Left Small Toe (LSmallToe)
21) Left Heel (LHeel)
22) Right Big Toe (RBigToe)
23) Right Small Toe (RSmallToe)
24) Right Heel (RHeel)

2.3 Long Jumping Rules and Technique

The standard setup for a long jumping competition is a runway with a sand landing pit adjacent to it. As seen in Figure 2-2, there is a foul line at the end of the runway and the athlete's foot must be behind it for the attempt to be valid. The measurement begins where the foul line ends, so jumping as close to it as possible is ideal. In addition, long jumpers usually receive three opportunities to complete a valid attempt in each round. Therefore, it is essential for a long jumper to consistently perform their technique.

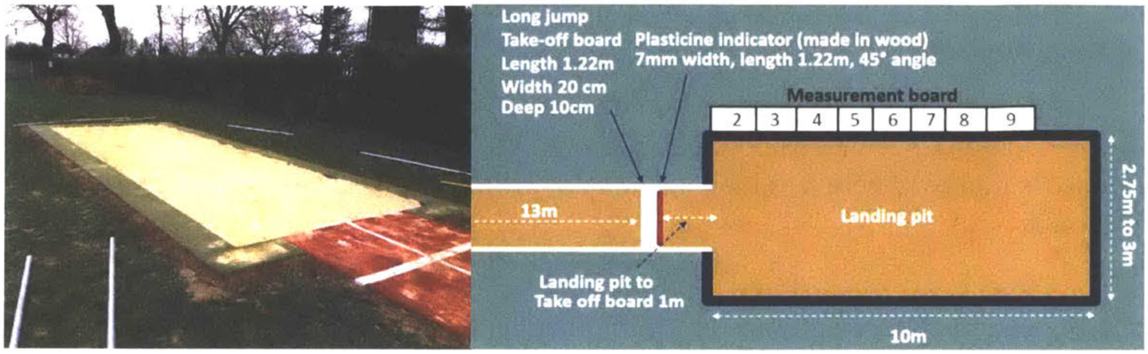


Figure 2-2: (left) Picture of long jumping setup.[8] (right) Model of long jumping setup, the measurement board displays the jumping distance the athlete is attempting to maximize.[9]

Modeling long jumping distance d_{final} requires knowledge of some key variables: center of mass (COM) position at launch $(0\ ,\ h_{launch})$, COM speed at launch v_{launch}, and angle of launch θ_{launch}. Accordingly, knowing when the take-off occurred is paramount. Since this application uses videos, this is referred to as the *launch frame*. Figure 2-3 is a graphical representation of our long jumping model with the defined parameters.

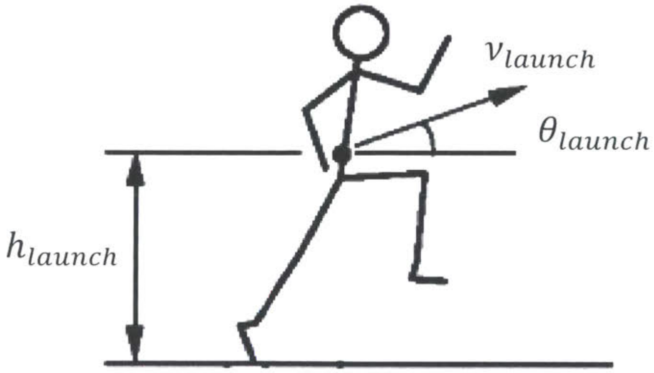


Figure 2-3: Model of long jumping at the launch frame, where our desired KPIs are displayed. [10]

The center of mass's height at launch h_{launch} is inherent to the athlete so it cannot be easily changed by changing techniques. However, v_{launch} and θ_{launch} can be altered with a change in technique, making them the relevant KPIs for long jumping performance. The long jumping distance d_{final} is then given by (Linthorne et al, 2005) [10]:

$$d_{final} = \frac{v_{launch}^2 \sin(2\theta_{launch})}{2g}\left[1 + (1 + \frac{2gh}{v_{launch}^2 \sin^2 \theta_{launch}}\right]^{\frac{1}{2}} \qquad (1)$$

Where g is the acceleration due to gravity ($9.81 \, {}^{m}/_{s^2}$) and relative h is given by:

$$h = h_{launch} - h_{landing} \tag{2}$$

According to Linthorne, Guzman, and Bridgett, the optimal launch angle is not the expected 45°. The launch velocity for a jumper decreases as the launch angle increases, so for most jumpers the optimal launch angle is approximately 22° [10].

2.4 Regressions

Regressions are the optimization of a model curve $\hat{Y}(T)$ such that it approximates the original data (T, Y) as best as possible. Since we are working with videos, T will be an array of the video's frames, where n is the total amount of frames in each video. This is shown in equation 3:

$$T = [1 \; 2 \ldots n] \tag{3}$$

Usually, regressions are optimized by minimizing the error between the model curve and the original signal. Since the error can be both positive and negative, it is commonplace to use the *Residual sum of squares* (RSS) to model the error. This can be seen in equation 4:

$$RSS = \sum_{i=1}^{n} [Y_i - \hat{Y}(i)]^2 \tag{4}$$

The model curve used can be composed of a linear combination of functions of T. To get the best results, it is ideal to choose the functions to use in any regression based on the data it is trying to approximate. In this application, we used linear and quadratic regressions.

2.4.1 Linear Regression

In a linear regression, the model curve will have the form of a linear equation, shown in equation 5:

$$\hat{Y}(T) = \beta_1 T + \beta_2 \tag{5}$$

In this case the RSS would be:

$$RSS_l = \sum_{i=1}^{n} [Y_i - (\beta_1 T_i + \beta_2)]^2 \tag{6}$$

This can be solved with matrix division. We define T_o as:

$$T_o = \begin{bmatrix} T_1 & 1 \\ T_2 & 1 \\ \vdots & \vdots \\ T_n & 1 \end{bmatrix} \tag{7}$$

To solve for the pair of coefficients (β_1, β_2) that minimize the RSS, we need to solve the relationship in equation 7:

$$Y = T_o \beta \text{ where } \beta = \begin{bmatrix} \beta_1 \\ \beta_2 \end{bmatrix} \tag{8}$$

The Matlab function "regress(Y, X)" returns an array β that minimizes any residual error RSS_l. An example of an implementation of the regress Matlab function can be seen in Figure 2-4.

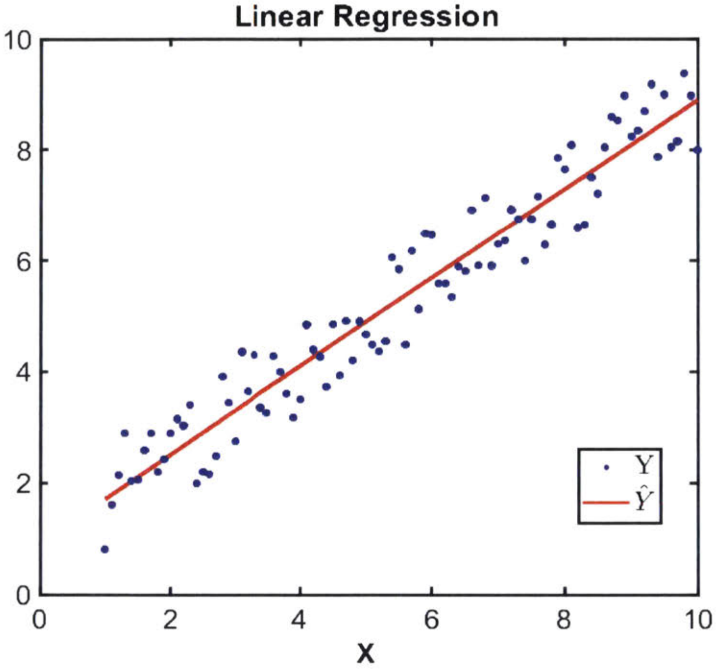


Figure 2-4: Result of linear regression on a discrete line where random noise was added results in an approximation of the original line.

2.4.2 Quadratic Regression

When making a quadratic regression, the model signal $\hat{Y}$ will take the form:

$$\hat{Y}(T) = \beta_1 T^2 + \beta_2 T + \beta_3 \tag{9}$$

Similar to the linear regression, the RSS is given by:

$$RSS_q = \sum_{i=1}^{n} \left[Y_i - \left(\beta_1 T_i^2 + \beta_2 T_i + \beta_3\right)\right]^2 \tag{10}$$

The Matlab function "fit(X, Y,'poly2')" will return a vector β that minimizes the error RSS_q where:

$$\beta = \begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix} \tag{11}$$

A second order polynomial is commonly used to model a projectile's center of mass y-coordinates. A second order regression is warranted when data collection leads to some error in data. An implementation of this using the Matlab function can be seen in Figure 2-5.

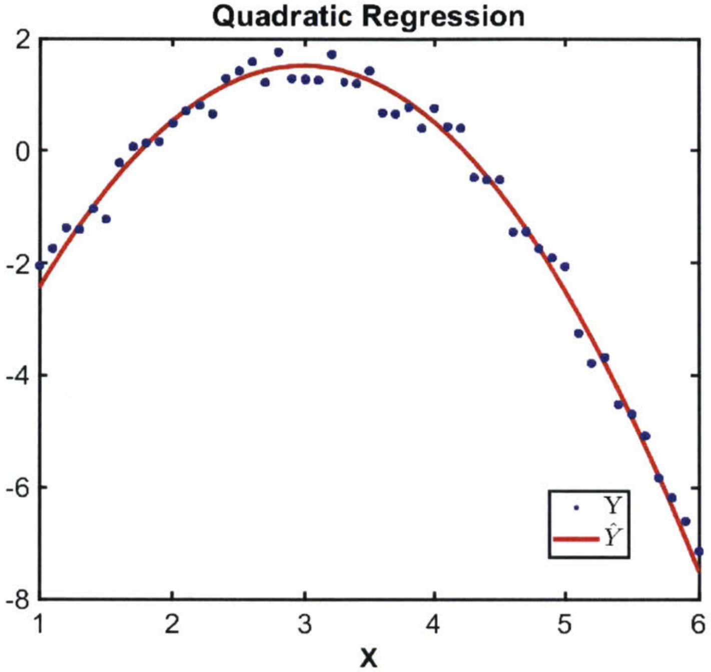


Figure 2-5: Result of quadratic regression on a discrete second order polynomial where random noise was added results in an approximation of the original polynomial.

3 Experimental Design

The goal was to develop an algorithm that will receive the athlete's coordinates as an input and output an estimate the launch frame. First, we needed accurate 'baseline' data to develop and corroborate the algorithm's efficacy. As discussed in Section 2.2, the OpenPose Demo was used as the data collection method and all the analysis was done in Matlab. Matlab was also used to produce the figures and the algorithm estimating the launch frame. The OpenPose data was processed to remove outliers. Only the foot signals for each foot (L/R Toe, L/R Ankle, L/R Heel) were used for the purpose of estimating the launch frame.

3.1 Manually Detect Launch Frame and Launch Foot

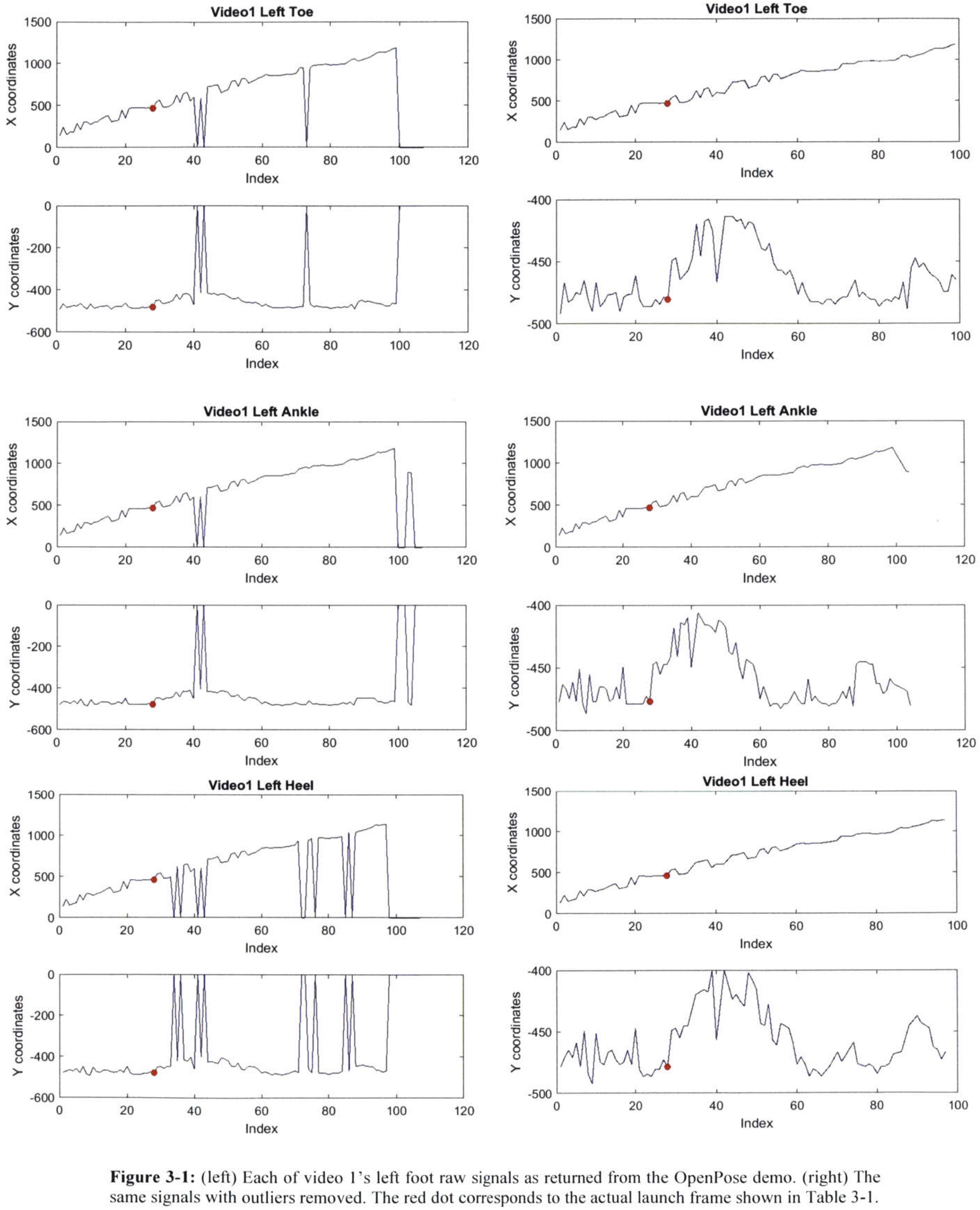
The true launch frame and launch foot were found by looking at the videos frame-by-frame using the Tracker Video Analysis and Modeling Tool [11]. The data points at the launch frame were highlighted by a red data point on all graphs to help identify trends that could be exploited in the algorithm. The actual launch frame and foot of the 10 videos analyzed are shown in Table 3-1.

Video #	*Actual Launch Frame*	*Actual Launch Foot*
1	28	Left
2	33	Left
3	26	Left
4	22	Left
5	34	Left
6	19	Left
7	22	Left
8	23	Left
9	30	Left
10	13	Left

Table 3-1: Actual launch frame and launch foot from each video, generated from looking at each video frame by frame.

3.2 Removing outliers

Since OpenPose's machine learning algorithm was not trained using long jump videos, it sometimes outputs data points that are incorrect for this application. Some coordinates are read at 0, resulting in a non-continuous curve. This happens in both x and y coordinates and could lead to inaccuracies when detecting the launch frame. Since the data is taken from a long jumping video, we expect the data to be continuous. Thus, the spikes in the raw OpenPose data are erroneous data points that were replaced by interpolation. A comparison of the OpenPose data before and after the outliers were removed and interpolated over is shown in Figure 3-1.


Video1 Left Toe

Video1 Left Toe

Video1 Left Ankle

Video1 Left Ankle

Video1 Left Heel

Video1 Left Heel

Figure 3-1: (left) Each of video 1's left foot raw signals as returned from the OpenPose demo. (right) The same signals with outliers removed. The red dot corresponds to the actual launch frame shown in Table 3-1.

These plots led us to believe the y-coordinates were more representative of the launch. That is, the data had distinctions around the launch frame that could be tested in an algorithm. In this case, all of the y-coordinate plots had a parabolic-shaped bump immediately after the launch frame, shown by the red dot. Therefore, we exclusively worked with y-coordinate data for the rest of the process.

4 Algorithm Development and Testing

We designed three algorithms and wrote Matlab code that implements each one. We then ran the OpenPose data from our ten long jump videos through this code to test each one's efficacy.

4.1 Breakpoints K

The y-coordinates graphs have 3 *phases*: pre-jump, jump, post-jump. *Breakpoints* are the frames where one phase ends and another begins. Most videos show the beginning of the second phase coinciding with the actual launch frame. An example of one such signal is shown in Figure 4-1.

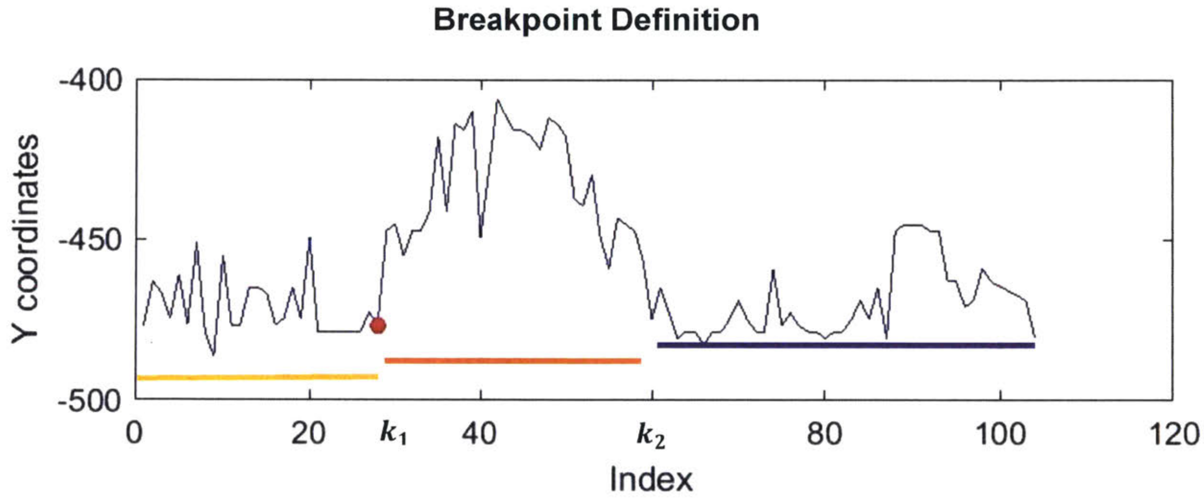

Breakpoint Definition

Figure 4-1: The actual launch frame is depicted by the red dot. In addition, the three defined phases are shown by the three colored, horizontal lines. Yellow is the first phase (1 to k_1), red is the second phase $(k_1 + 1\ to\ k_2)$, and purple is the third phase $(k_2 + 1\ to\ end)$. The breakpoints $k_1\ and\ k_2$ are shown on the horizontal axis.

4.2 Aggregate Phase Regression (2 Breakpoints per signal)

First, we chose to model the y-coordinates' phases by a line, a parabola, and a line, respectively. The goal was to choose optimizing breakpoints for each signal that would minimize the error between our estimated curve and each of the foot signals. Thus, we tested different combinations of possible breakpoints $k = (k_1, k_2)$ and performed two linear and one quadratic regressions at each combination k. In each trio of regressions, we stored the Aggregate Residual Sum of Squares (RSS_A), given by equation 12; where RSS_{l1} is the RSS of the first linear regression, RSS_q is the RSS of the quadratic regression, and RSS_{l2} is the RSS of the second linear regression.

$$RSS_A = RSS_{l1} + RSS_q + RSS_{l2} \tag{12}$$

This definition of the Aggregate Residual Sum of Squares can be expanded into equation 13 to better understand the algorithm.

$$RSS_A = \sum_{i=1}^{k_1} \left(Y_i - \widehat{Y_{l1}}(T_i)\right)^2 + \sum_{i=k_1+1}^{k_2} \left(Y_i - \widehat{Y_q}(T_i)\right)^2 + \sum_{i=k_2+1}^{n} \left(Y_i - \widehat{Y_{l2}}(T_i)\right)^2 \tag{13}$$

In this definition, n is the number of toe, heel, or ankle coordinates, $T = [1\ 2\ \dots\ n]$ is the vector of frames, $Y = [Y_1\ Y_2\ \dots\ Y_n]$ is the vector of y-coordinates at each frame, and $\widehat{Y_{l1}}$, $\widehat{Y_q}$, $\widehat{Y_{l2}}$ are the model curves given by each regression. After testing each possible combination of k, we chose the one with the smallest RSS_A as our optimized breakpoints k_{opt}. The launch frame is the first of these optimized breakpoints $k_{1,opt}$. Figure 4-2 shows the results of this algorithm on each foot signal in video 7.

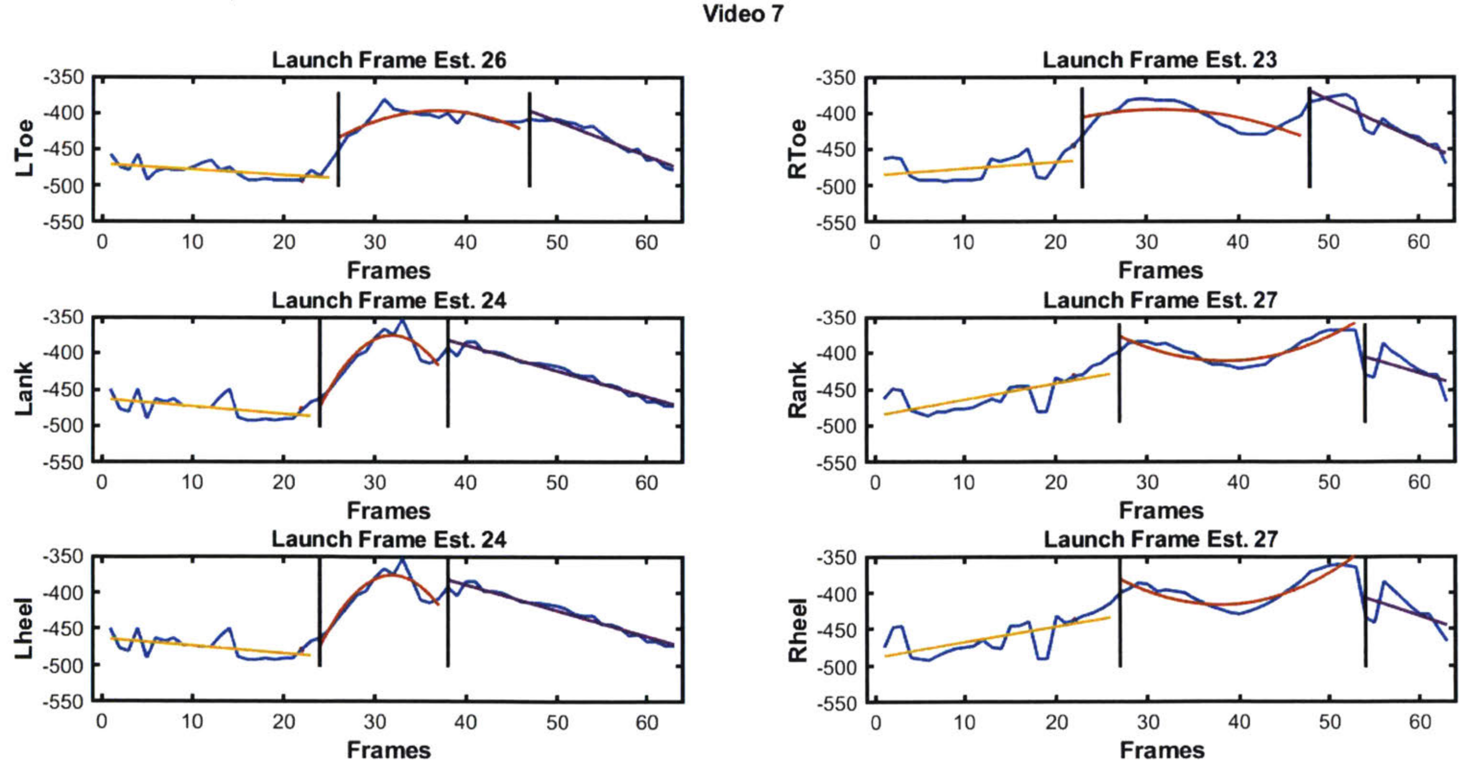


Figure 4-2: The results of running the data from video 7 through the algorithm. k_1 for each signal is shown at the top of each subplot. As can be seen, the quadratic region was not estimated how we expected. The estimated curve in each phase is depicted by the same colors as in Figure 4-1.

The optimizing break points were not as hypothesized in Figure 4-1. Ideally, the quadratic regression region, shown in red above, would be concave and symmetric around the vertex. However, as seen in the Right Heel plot (bottom right), this is not always the case. This is likely due to noise from pose estimation and modeling the foot coordinates instead of the center of mass coordinates. This algorithm was run on the 6 signals in each of the 10 videos and we compiled the Launch Frame Error (LFE) for each signal, given by equation 14.

$$LFE = k_1 - (Actual\ launch\ frame) \tag{14}$$

The Launch Frame Error magnitude for each signal is shown in Figure 4-3.

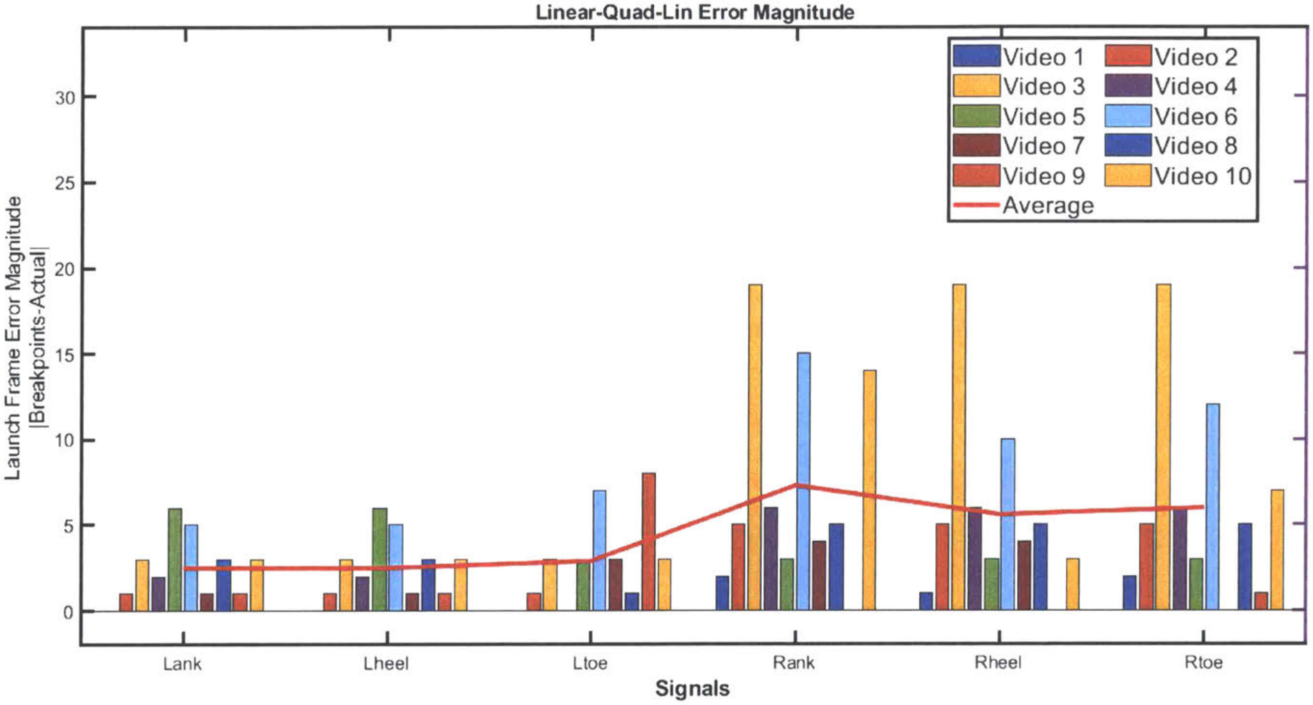


Figure 4-3: Launch Frame Error magnitude for each signal (horizontal axis) and each video (legend). Note that the left foot estimates are much better. It is also important to note that this is the absolute value error, as the error can be either positive or negative. The average error foe each signal is shown by the red line.

The estimated launch frame performed well ($|\overline{LFE}|$=3) as long as we look at signals given by the correct launch foot, which is the left foot in video 7. In addition, the average error magnitude was the same for each of the left foot signals, leading us to believe that none of the signals was more representative than the other two. There is also the need to decide how to estimate the video's launch frame from each signal's k_1.

4.3 Aggregate Phase Regression (3 Breakpoints per Signal)

We simplified the definition of phases and breakpoints to mitigate the pose estimation's noise. Instead of modeling the signal with 2 linear regressions and 1 quadratic regression, we chose to do so with 4 linear regressions. Therefore, we now expected 4 phases, defined in Figure 4-4.

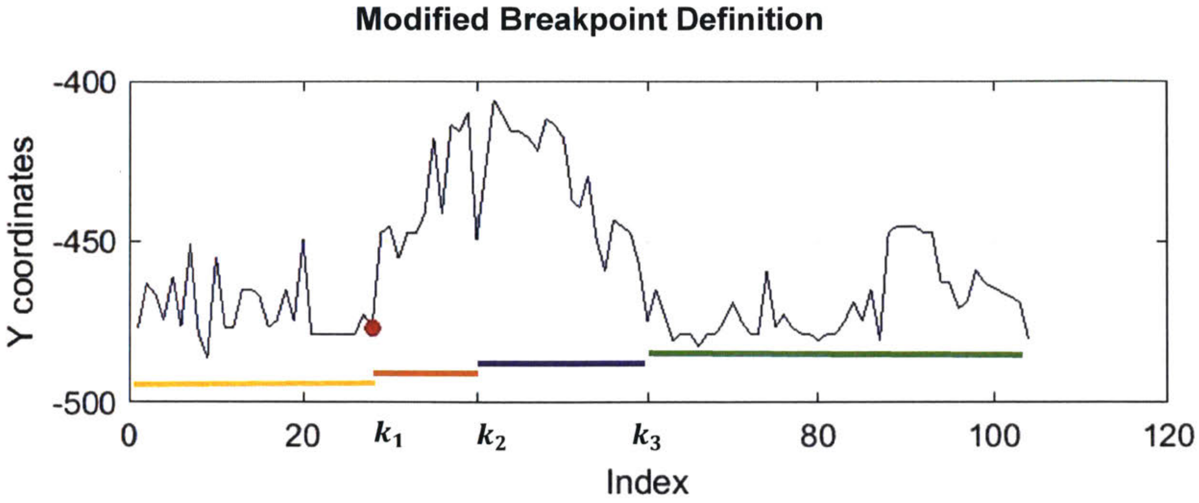


Figure 4-4: The actual launch frame is depicted by the red dot. In addition, the four defined phases are shown by the four colored, horizontal lines. Yellow is the first phase (1 to k_1), red is the second phase ($k_1 + 1\ to\ k_2$), purple is the third phase ($k_2 + 1\ to\ k_3$), and green is the fourth phase ($k_3 + 1\ to\ end$). The breakpoints $k_1,\ k_2\ and\ k_2$ are shown on the horizontal axis.

Similarly, our Aggregate Residual Sum of Squares definition was changed to reflect the new breakpoint definition. This is shown in equation 15, where $RSS_{l,i}$ is the linear regression in each phase.

$$RSS_A = RSS_{l1} + RSS_{l2} + RSS_{l3} + RSS_{l4} \tag{15}$$

Our expanded Aggregate Residual Sum of Squares definition is given by equation 16.

$$\begin{aligned} RSS_A = {} & \sum_{i=1}^{k_1} \left(Y_i - \widehat{Y_{l1}}(T_i)\right)^2 + \sum_{i=k_1+1}^{k_2} \left(Y_i - \widehat{Y_{l2}}(T_i)\right)^2 \\ & + \sum_{i=k_2+1}^{k_3} \left(Y_i - \widehat{Y_{l3}}(T_i)\right)^2 + \sum_{i=k_3+1}^{n} \left(Y_i - \widehat{Y_{l4}}(T_i)\right)^2 \end{aligned} \tag{16}$$

In this definition, n is the number of frames, $T = [1\ 2\ ...\ n]$ is the vector of frames, $Y = [Y_1\ Y_2\ ...\ Y_n]$ is the vector of y-coordinates at each frame, and $\widehat{Y_{l1}}, \widehat{Y_{l2}}, \widehat{Y_{l3}}, \widehat{Y_{l4}}$ are the model curves given by each regression. After testing each possible combination $k = (k_1, k_2, k_3)$, we chose the one with the smallest RSS_A as our optimized breakpoints k_{opt}. The launch frame is the first of these optimized breakpoints $k_{1,opt}$. Figure 4-5 shows the results of this algorithm on each foot signal in video 9.

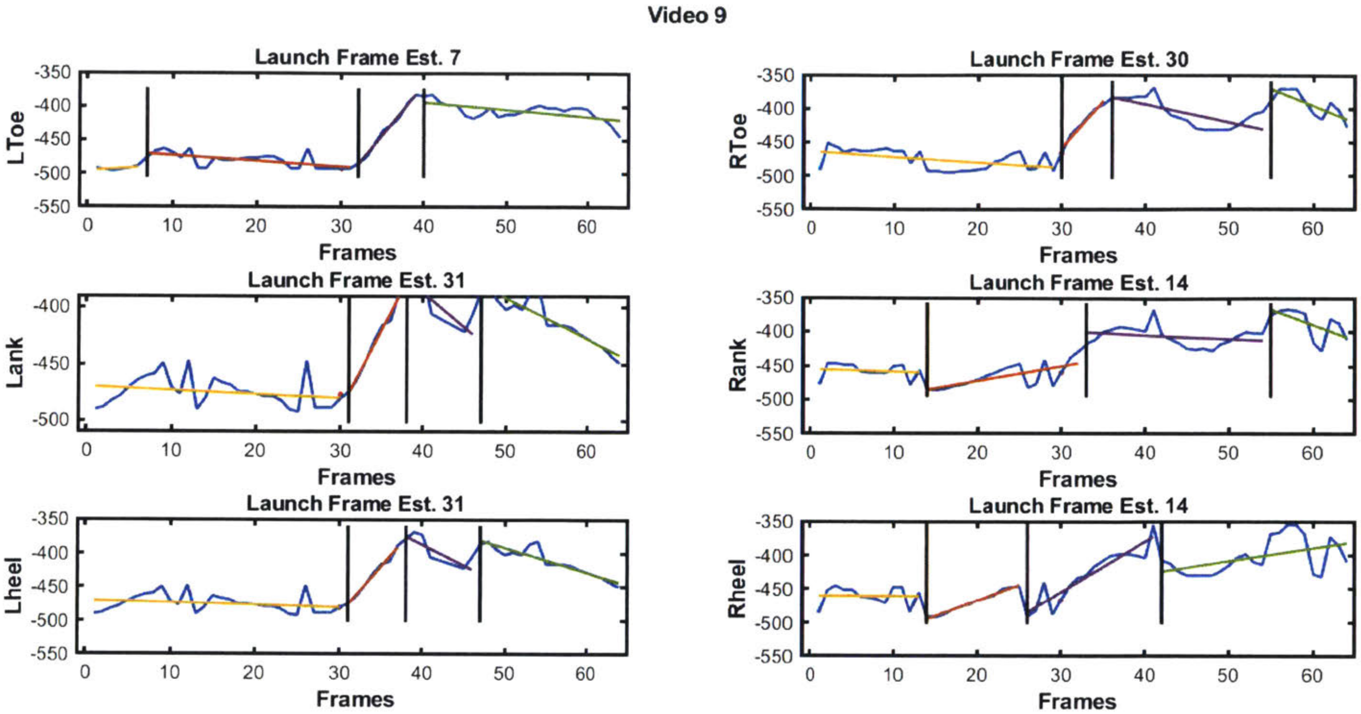


Figure 4-5: The results of running the data from video 9 through the algorithm. k_1 for each signal is shown at the top of each subplot. As can be seen, the linear breakpoints align with our theorized phase regions in Figure 4-4. The estimated curve in each phase is depicted by the same colors as in Figure 4-4. Note that the left toe (top left) k_1 is considerably different than the rest and that the signal does not clearly show the landing.

Using this algorithm, the optimized breakpoints k_{opt} consistently were at the theorized start of each phase. However, video 9, shown in Figure 4-5, had an outlier in the Left Toe coordinates. We ran the data from each of the 10 videos through this algorithm and compiled the Launch Frame Error, as defined in equation 14. This algorithm's LFE magnitude is shown in Figure 4-6.

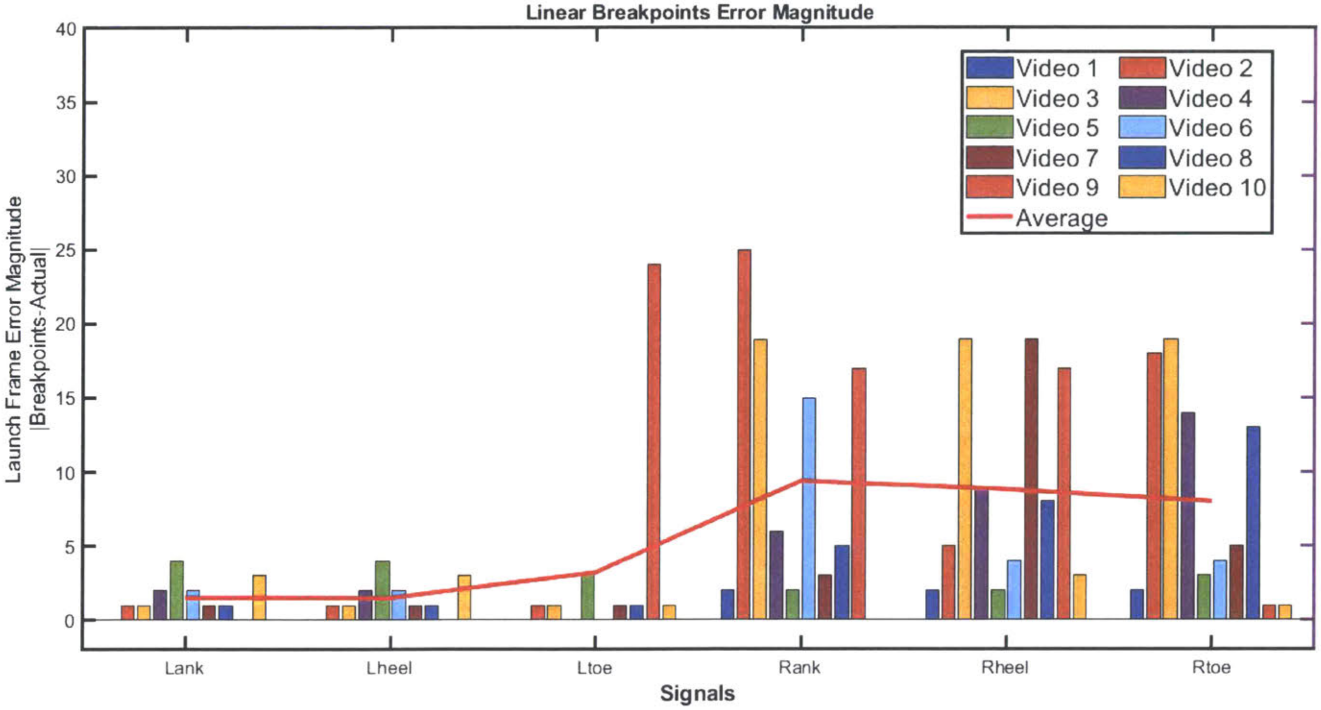


Figure 4-6: Launch Frame Error magnitude for each signal (horizontal axis) and each video (legend). Note that the left foot estimates are much better. It is also important to note that this is the absolute value error, as the error can be either positive or negative. The average error foe each signal is shown by the red line.

In video 8, the launch foot is the left foot, which is reflected in Figure 4-6. The algorithm performed better when looking at the left foot's coordinates. The LFE of the left foot's coordinates are shown in Figure 4-7.

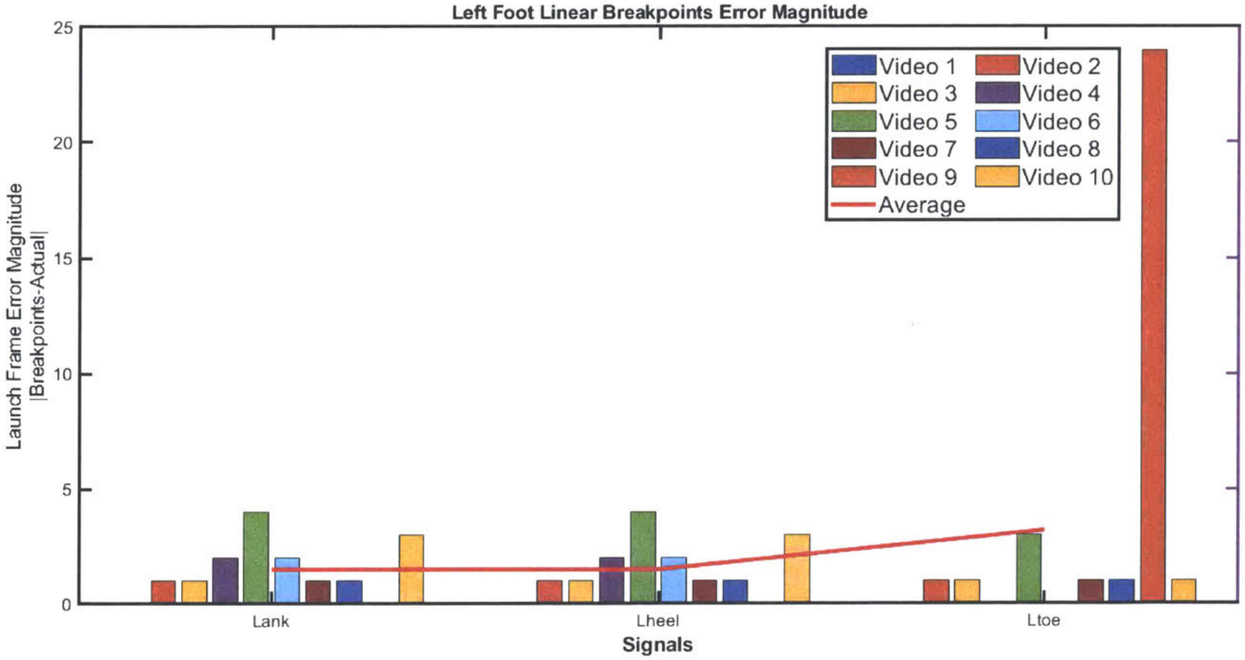


Figure 4-7: Left foot launch frame error magnitude. The outlier in video 9 could severely affect our launch frame estimate for the video. The average error is shown by the red line.

The algorithm returned a large outlier in video 9's left toe signal due to noise around the time the athlete landed, damaging the signal. The average LFE for the left toe signals was 2.8 frames while including the outlier and 2 frames not including the outlier. Thus, it performed better than the 2 Breakpoint Aggregate Regression algorithm, but we still have the problem of choosing one launch frame for the video while we have 3 k_1s. Bases on these results, we decided to just use the launch foot coordinates (left foot for our videos) as their results are more accurate.

4.4 Multiple Signal Aggregate Phase Regression (3 Breakpoints per Video)

We decided to design the algorithm such that it will return one combination of optimized breakpoints k_{opt} per video. We theorized that it would make the algorithm less sensitive to outliers and that it would be more be more accurate, as we would be using data from 3 signals. We used the same definition for breakpoints as discussed in Figure 4-4, $k = (k_1, k_2, k_3)$, but we modified our Aggregate Residual Sum of Squares definition to that in equation 17, where $RSS_{li,s}$ is the linear RSS for signal s.

$$RSS_A = \sum_{s=1}^{3} \left[RSS_{l1,s} + RSS_{l2,s} + RSS_{l3,s} + RSS_{l4,s} \right] \tag{17}$$

Again, we can expand this equation to show the algorithm in more detail. This expanded version is given by equation 18.

$$\begin{aligned} RSS_A = \sum_{s=1}^{3} \Bigg[& \sum_{i=1}^{k_1} \left(Y_{i,s} - \widehat{Y_{l1,s}}(T_{i,s}) \right)^2 + \sum_{i=k_1+1}^{k_2} \left(Y_{i,s} - \widehat{Y_{l2,s}}(T_{i,s}) \right)^2 \\ & + \sum_{i=k_2+1}^{k_3} \left(Y_{i,s} - \widehat{Y_{l3,s}}(T_{i,s}) \right)^2 + \sum_{i=k_3+1}^{n} \left(Y_{i,s} - \widehat{Y_{l4,s}}(T_{i,s}) \right)^2 \Bigg] \end{aligned} \tag{18}$$

In this definition, n is the number of toe, heel, or ankle coordinates, $s = [LToe, Lank, Lheel]$ is the signals we are using, $T = [1\ 2\ ...\ n]$ is the vector of frames, $Y = [Y_1\ Y_2\ ...\ Y_n]$ is the vector of y-coordinates at each frame, and $\widehat{Y_{l1}}$, $\widehat{Y_{l2}}$, $\widehat{Y_{l3}}$, $\widehat{Y_{l4}}$ are the model curves given by each regression. After testing each possible combination $k = (k_1, k_2, k_3)$, we chose the one with the smallest RSS_A as our optimized breakpoints k_{opt}. The launch frame is the first of these optimized breakpoints $k_{1,opt}$. Figure 4-8 shows the results of this algorithm on each foot signal in video 9.

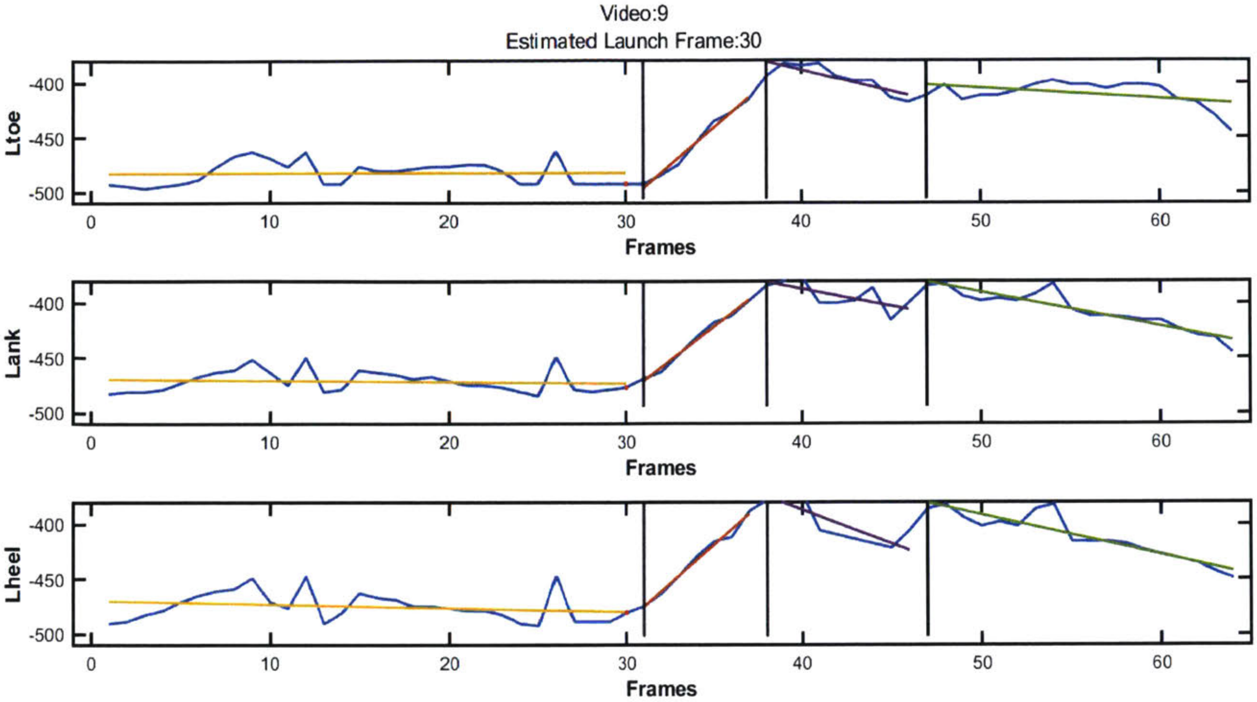


Figure 3-8: The results of running the data from video 9 through the algorithm. k_1 for each signal is shown at the top of each subplot. As can be seen, the linear breakpoints align with our theorized phase regions in Figure 4-4. The estimated curve in each phase is depicted by the same colors as in Figure 4-4. Note that the left toe (top left) k_1 is considerably different than the rest and that the signal does not clearly show the landing.

Adding the Aggregate Residual Sum of Squares between each of the launch foot's signals eliminated the algorithm's sensitivity to noise. In the previous algorithm, the video 9 results, Figure 4-5, had a large outlier in the left toe breakpoints, but this implementation fixed that. The Launch Frame Error magnitude of each video is shown in Figure 4-9.

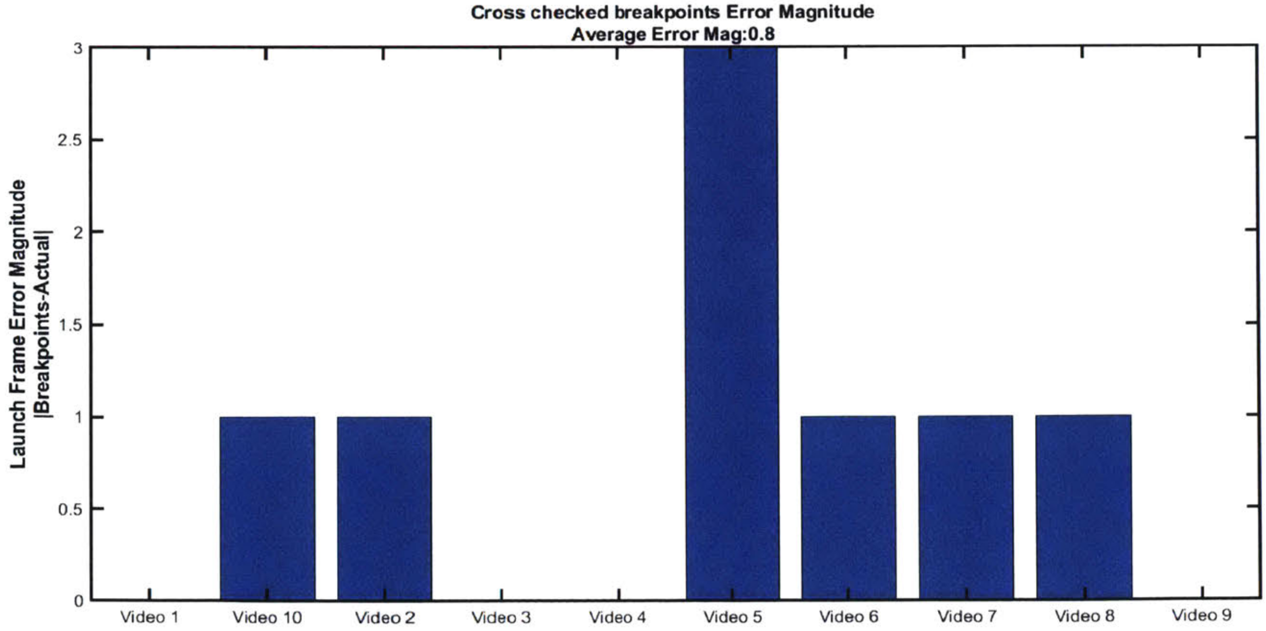


Figure 4-9: Average launch frame error magnitude for each video (x-axis). Note that the error can be either positive or negative.

The average launch frame error magnitude has now been reduced to 0.8±0.91 frames as long as the correct launch foot's signals are used. In this case, all the videos were of an athlete jumping with their left foot. Giving the algorithm access to an athlete's metadata (e.g. what foot they jump with) would give it the ability to be used while training.

5 Conclusions

An algorithm that consistently provides an accurate estimate for the launch frame from a long jumping video coordinates was made using computer vision and Matlab. With an average launch frame error magnitude of 0.8±0.91 frames, the algorithm can be used to accurately compute launch angle and launch velocity from a long jumping video while avoiding the tedious task of performing the calculations by hand. Consequently, a quantitative, external feedback training tool for long jumping may be within reach.

6 Appendices

Appendix A: Code

Find outliers and eliminate with interpolation

Author: Pablo E. Muniz

Date: 03/09/2019

This function takes a struct object of raw OpenPose data and interpolates over *outliers*. Outliers are data points that have coordinates at 0. OpenPose has the origin at the top left of a frame by default, so $(a, 0)$ is the top of a frame and $(0, a)$ is the left edge of a frame.

Contents

Function definition

```
function fixed_signal = fix_outliers(raw_signal)
```

Flagging X-coordinates

To find X-coordinate outliers check to see if they are at 0. Add indices where signal is at 0 to *index*

```
X = raw_signal.x;

X(isnan(X)) = [];

index = [];

for i = 1:length(X)
    if X(i) == 0
        index = [index; i];
    end
end
```

Interpolating X-coordinates

If there are *n* successive outliers, we want to interpolate over them with non-outliers.
Thus, if $X_i, X_{i+1}, ..., X_{i+n-1}$ are outliers,

We will use the preceding and following accurate data points: X_{i-1} and X_{i+n} to interpolate.

Find which outliers are successive and which ones are single. Create a vector *replace* with the staring index and ending index of successive outliers (the start and end index will be the same if it is a single outlier).
We will use recursive programming to make the vector current. The function *find_successive* is shown at the bottom of the document.

```
current = [];
```

```
replace = [];
```

Check the indeces in *index* and run them through *find_successive*

```
for i = 1:length(index)
    [replace,current] = find_successive(index(i),replace,current);
end
```

Add the last row from current from the *find_successive* funtion. This needs to be done because of the way find_successive was designed.

```
if isempty(current) == 0
    replace = [replace; current(1) , current(end)];
end
if isempty(replace) == 0
```

Sample the rows in replace to interpolate over single and successive outliers.

```
    for i = 1:length(replace(:,1))
```

If the second index in the sampled replace row is the last index, then eliminate these outliers from the signal since we are not interested in extrapolating.

```
        if replace(i,2) == length(X)
            X(replace(i,1):replace(i,2)) = [];
```

When the first and second index in the sampled replace row are different, this means there are successive outliers. Interpolate using the preceding and following accurate data points: X_{i-1} and X_{i+n} (Trapezoidal method).

```
        elseif replace(i,1) ~= replace(i,2)
            X(replace(i,1)-1:replace(i,2)+1) = linspace(X(replace(i,1)-
1),X(replace(i,2)+1),replace(i,2)-replace(i,1)+3);
```

When the first and second index in the sampled replace row are the same, this is a single outlier. We interpolate using the preceding and following data point (Trapezoidal method).

```
        else
            X(replace(i,1)) = (X(replace(i,1)-1)+X(replace(i,1)+1))/2;
        end
    end
end
```

Y-coordinates

The same process as described above is performed with the Y-coordinates.

```
Y = raw_signal.y;

Y(isnan(Y)) = [];

index = [];
```

```
for i = 1:length(Y)
    if Y(i) == 0
        index = [index ; i];
    end
end
replace = [];
current = [];
for i = 1:length(index)
    [replace,current] = find_successive(index(i),replace,current);
end
if isempty(current) == 0
    replace = [replace; current(1) , current(end)];
end

if isempty(replace) == 0
    for i = 1:length(replace(:,1))
        if replace(i,2) == length(Y)
            Y(replace(i,1):replace(i,2)) = [];
        elseif replace(i,1) ~= replace(i,2)
            Y(replace(i,1)-1:replace(i,2)+1) = linspace(Y(replace(i,1)-
1),Y(replace(i,2)+1),replace(i,2)-replace(i,1)+3);
        else
            Y(replace(i,1)) = (Y(replace(i,1)-1)+Y(replace(i,1)+1))/2;
        end
    end
end
```

Construct the *fixed* struct object to be returned.

```
fixed_signal.x = X;

fixed_signal.y = Y;

end
```

find_successive

Function definition. *num* is the index from *index* being sampled, *groups* is making *replace* and *current* is the last successive data sampled.

```
function [groups,current] = find_successive(num,groups,current)
```

If *current* is empty, add the current sampled *num* index to it.

```
if isempty(current) == 1
    current = [num];
```

If the index being sampled is right after the last index sampled, update *current*.

```
elseif current(end) + 1 == num
    current = [current ; num];
```

If the above conditions were not met (if *current* was not updated), add the last successive indices to *replace* and update *current*.

```
elseif current(end) ~= num
    groups = [groups ; current(1) , current(end)];
    current = [num];
end
```

```
end
```

Published with MATLAB® R2019a

Find Breakpoints (2 breakpoints: linear-quadratic-linear)

Author: Pablo E. Muniz

Date: 04/14/2019

This function takes a struct object and returns the aggregate Residual Sum of Squares (RSS) between the y-signal and 3 regressions: linear, quadratic, and linear. It is testing different break point permutations (k_1, k_2) and returning the optimal permutation along with the RSS (R_{min}). It also creates a plot a with the signal, each of the three regressions, and the breakpoints.

Contents

Funtion Definition

```
function [Rmin , k1 , k2 , a] = changepts(signal)
```

Eliminate outliers and isolate y-signal.

```
signal_no_out = fix_outliers(signal);

signaly_no_out = signal_no_out.y;
```

Test breakpoint permutations (k_1, k_2)

Lmin is the minimum distance between changepoints and can be modified below depending on choice of speed vs. accuracy. The vector *R* will store the RSS and its corresponding permutation of (k_1, k_2).

```
R = [];

Lmin = 5;

for k1 = 1:numel(signaly_no_out)-Lmin
    for k2 = k1+Lmin:numel(signaly_no_out)
```

Cut signals into three parts, testing changepoints.

```
        cut1 = signaly_no_out(1:k1-1);

        indeces1 = 1:k1-1;

        cut2 = signaly_no_out(k1:k2-1);
        indeces2 = k1:k2-1;

        cut3 = signaly_no_out(k2:end);

        indeces3 = k2:numel(signaly_no_out);
```

Regressions

We used Matlab's *regress* function to perform linear regression:
$Y = X * \beta$

where X is the indices column vector joined with a column vector of ones and $\beta =< \beta_1; \beta_2 >$.

We used Matlab's *fit* function to perform quadratic regression:
Regression of first cut

```
        [B1,~,R1] = regress(cut1 , [indeces1' , ones(numel(indeces1),1)]);

        R1 = sum(R1.^2);
```

Regression of second cut

```
        myfit2 = fit(indeces2' , cut2 , 'poly2');
        coeffs = coeffvalues(myfit2);
        poly2fit = coeffs(1)*indeces2'.^2 + coeffs(2)*indeces2' + coeffs(3)';
        R2 = sum((poly2fit-cut2).^2);
```

Regression of third cut

```
        [B3,~,R3] = regress(cut3 , [indeces3' , ones(numel(indeces3),1)]);

        R3 = sum(R3.^2);
```

Add error and its corresponding (k_1, k_2) pairing to existing R.

```
        R = [R ;

            R1+R2+R3 , k1 , k2];

    end
end
```

Picking best breakpoint permutation

The optimizing permutation of (k_1, k_2) will be the one with the lowest Residual Sum of Squares.

```
Rmin = min(R(:,1));
```

```
optimized_index_pair = find(R(:,1) == Rmin);

Ropt = R(optimized_index_pair,:);

k1 = Ropt(2);

k2 = Ropt(3);
```

Get the fits made with (k_1, k_2) to plot

```
cut1 = signaly_no_out(1:k1-1);

indeces1 = 1:k1-1;

cut2 = signaly_no_out(k1:k2-1);

indeces2 = k1:k2-1;

cut3 = signaly_no_out(k2:end);

indeces3 = k2:numel(signaly_no_out);

[B1,~,~] = regress(cut1 , [indeces1' , ones(numel(indeces1),1)]);

myfit2 = fit(indeces2' , cut2 , 'poly2');
coeffs = coeffvalues(myfit2);
poly2fit = coeffs(1)*indeces2'.^2 + coeffs(2)*indeces2' + coeffs(3)';
[B3,~,R3] = regress(cut3 , [indeces3' , ones(numel(indeces3),1)]);
```

Plotting

```
a = plot((1:numel(signaly_no_out))' , signaly_no_out , '-');
hold on
a = plot((indeces2)', poly2fit,'-');
a = plot((indeces1)' , (B1(1)*indeces1'+B1(2)),'-');
a = plot((indeces3)' , (B3(1)*indeces3'+B3(2)),'-');
a = plot(k1*ones(100,1),linspace(min(signaly_no_out)-
10,max(signaly_no_out)+10,100),'k-')
a = plot(k2*ones(100,1),linspace(min(signaly_no_out)-
10,max(signaly_no_out)+10,100),'k-')
improvePlot
ylim([-510,-390])
xlim([-1,numel(signaly_no_out)+1])
a
end
```

Published with MATLAB® R2019a

Find Break Points (3 breakpoints: $linear^4$)

Author: Pablo E. Muniz

Date: 04/14/2019

This function takes a struct object and returns the aggregate Residual Sum of Squares (RSS) between the y-signal and 4 linear regressions. It is testing different break point permutations (k_1, k_2, k_3) and returning the optimal permutation along with the RSS (R_{min}). It also creates a plot a with the signal, each of the four regressions, and the breakpoints.

Contents

Function Definition

```
function [Rmin , k1 , k2 , k3 , a] = linear_brkpts(signal)
```

Eliminate outliers and isolate y-signal

```
signal_no_out = fix_outliers(signal);

signaly_no_out = signal_no_out.y;
```

Test breakpoint permutations (k_1, k_2, k_3)

Lmin is the minimum distance between changepoints and can be modified below depending on choice of speed vs. accuracy. The vector *R* will store the RSS and its corresponding permutation of (k_1, k_2, k_3).
Note: Adding limits to what the breakpoints can be makes the function significantly faster, but it runs the chance of the limits being after each breakpoint's mark in the data. These limits were calibrated based on the data available.

```
R = [];

Lmin = 5;

for k1 = 1:round(1/2*numel(signaly_no_out))
    for k2 = k1+Lmin:round(6/10*numel(signaly_no_out))
        for k3 = k2+Lmin:round(9/10*numel(signaly_no_out))
% For no limits on breakpoints:
% for k1 = 1:numel(signaly_no_out)-2*Lmin
    % for k2 = k1+Lmin:numel(signaly_no_out)-Lmin
        % for k3 = k2+Lmin:numel(signaly_no_out)
```

Cut signals into four parts:

```
            cut1 = signaly_no_out(1:k1-1);
```

```
indeces1 = 1:k1-1;
cut2 = signaly_no_out(k1:k2-1);
indeces2 = k1:k2-1;
cut3 = signaly_no_out(k2:k3-1);
indeces3 = k2:k3-1;
cut4 = signaly_no_out(k3:end);
indeces4 = k3:numel(signaly_no_out);
```

Regressions

We used Matlab's *regress* function to perform linear regression:

$$Y = X * \beta$$

where X is the indices column vector joined with a column vector of ones and $\beta =< \beta_1; \beta_2 >$.

Regression of first cut:

```
[B1,~,R1] = regress(cut1 , [indeces1' , ones(numel(indeces1),1)]);
R1 = sum(R1.^2);
```

Regression of second cut:

```
[B2,~,R2] = regress(cut2 , [indeces2' , ones(numel(indeces2),1)]);
R2 = sum(R2.^2);
```

Regression of third cut:

```
[B3,~,R3] = regress(cut3 , [indeces3' , ones(numel(indeces3),1)]);
R3 = sum(R3.^2);
```

Regression of fourth cut:

```
[B4,~,R4] = regress(cut4 , [indeces4' , ones(numel(indeces4),1)]);
R4 = sum(R4.^2);
```

Add error and its corresponding (k_1, k_2, k_3) pairing to existing R.

```
            R = [R ;

                R1+R2+R3+R4 , k1 , k2 , k3];

        end
    end
end
```

Picking best breakpoint permutation

The optimizing permutation of (k_1, k_2, k_3) will be the one with the lowest Residual Sum of Squares.

```
Rmin = min(R(:,1));

optimized_index_pair = find(R(:,1) == Rmin);

Ropt = R(optimized_index_pair,:);

k1 = Ropt(2);

k2 = Ropt(3);

k3 = Ropt(4);
```

Get the fits made with (k_1, k_2, k_3) to plot

```
cut1 = signaly_no_out(1:k1-1);

indeces1 = 1:k1-1;

cut2 = signaly_no_out(k1:k2-1);

indeces2 = k1:k2-1;

cut3 = signaly_no_out(k2:k3-1);

indeces3 = k2:k3-1;

cut4 = signaly_no_out(k3:end);

indeces4 = k3:numel(signaly_no_out);

[B1,~,~] = regress(cut1 , [indeces1' , ones(numel(indeces1),1)]);

[B2,~,~] = regress(cut2 , [indeces2' , ones(numel(indeces2),1)]);

[B3,~,~] = regress(cut3 , [indeces3' , ones(numel(indeces3),1)]);
```

```
[B4,~,~] = regress(cut4 , [indeces4' , ones(numel(indeces4),1)]);
```

Plotting

```
a = plot((1:numel(signaly_no_out))' , signaly_no_out , '-');
hold on
a = plot((indeces2)', (B2(1)*indeces2'+B2(2)),'-');
a = plot((indeces1)' , (B1(1)*indeces1'+B1(2)),'-');
a = plot((indeces3)' , (B3(1)*indeces3'+B3(2)),'-');
a = plot((indeces4)' , (B4(1)*indeces4'+B4(2)),'-');
a = plot(k1*ones(100,1),linspace(min(signaly_no_out)-
10,max(signaly_no_out)+10,100),'k-');
a = plot(k2*ones(100,1),linspace(min(signaly_no_out)-
10,max(signaly_no_out)+10,100),'k-');
a = plot(k3*ones(100,1),linspace(min(signaly_no_out)-
10,max(signaly_no_out)+10,100),'k-');
improvePlot
ylim([-510,-390])
xlim([-1,numel(signaly_no_out)+1])
a
end
```

Published with MATLAB® R2019a

Find Break Points (3 signals, 3 breakpoints,: $linear^4$)

Author: Pablo E. Muniz

Date: 04/19/2019

This function takes three struct objects and returns the aggregate Residual Sum of Squares (RSS) between the 3 y-signals and 4 linear regressions per signal. It is testing different break point permutations (k_1, k_2, k_3) and returning the optimal permutation along with the RSS (R_{min}). It also creates a plot a with the signal, each of the four regressions, and the breakpoints.

Contents

Function Definition

```
function [Rmin , k1 , k2 , k3 , a] = xcheck_linear_brkpts(signal1,signal2,signal3,j)
```

Eliminate outliers and isolate y-signal

```
signalA_no_out = fix_outliers(signal1);

signalAy_no_out = signalA_no_out.y;

signalB_no_out = fix_outliers(signal2);

signalBy_no_out = signalB_no_out.y;

signalC_no_out = fix_outliers(signal3);

signalCy_no_out = signalC_no_out.y;
```

Test breakpoint permutations (k_1, k_2, k_3)

Lmin is the minimum distance between changepoints and can be modified below depending on choice of speed vs. accuracy. The vector *R* will store the RSS and its corresponding permutation of (k_1, k_2, k_3).
Note: Adding limits to what the breakpoints can be makes the function significantly faster, but it runs the chance of the limits being after each breakpoint's mark in the data. These limits were calibrated based on the data available.

```
R = [];

Lmin = 5;

for k1 = 1:round(1/2*numel(signalAy_no_out))
    for k2 = k1+Lmin:round(6/10*numel(signalAy_no_out))
```

```
        for k3 = k2+Lmin:round(9/10*numel(signalAy_no_out))
% For no limits on breakpoints:
% for k1 = 1:numel(signaly_no_out)-2*Lmin
    % for k2 = k1+Lmin:numel(signaly_no_out)-Lmin
        % for k3 = k2+Lmin:numel(signaly_no_out)
```

Cut signals into four parts:

```
            cut1A = signalAy_no_out(1:k1-1);

            indeces1A = 1:k1-1;

            cut2A = signalAy_no_out(k1:k2-1);

            indeces2A = k1:k2-1;

            cut3A = signalAy_no_out(k2:k3-1);

            indeces3A = k2:k3-1;

            cut4A = signalAy_no_out(k3:end);

            indeces4A = k3:numel(signalAy_no_out);

            cut1B = signalBy_no_out(1:k1-1);

            indeces1B = 1:k1-1;

            cut2B = signalBy_no_out(k1:k2-1);

            indeces2B = k1:k2-1;

            cut3B = signalBy_no_out(k2:k3-1);

            indeces3B = k2:k3-1;

            cut4B = signalBy_no_out(k3:end);

            indeces4B = k3:numel(signalBy_no_out);

            cut1C = signalCy_no_out(1:k1-1);

            indeces1C = 1:k1-1;

            cut2C = signalCy_no_out(k1:k2-1);

            indeces2C = k1:k2-1;

            cut3C = signalCy_no_out(k2:k3-1);

            indeces3C = k2:k3-1;
```

```
cut4C = signalCy_no_out(k3:end);

indeces4C = k3:numel(signalCy_no_out);
```

Regressions

We used Matlab's *regress* function to perform linear regression:
$Y = X * \beta$

where X is the indices column vector joined with a column vector of ones and $\beta =< \beta_1; \beta_2 >$.

Regression of first cut:

```
[B1A,~,R1A] = regress(cut1A , [indeces1A' , ones(numel(indeces1A),1)]);

R1A = sum(R1A.^2);

[B1B,~,R1B] = regress(cut1B , [indeces1B' , ones(numel(indeces1B),1)]);

R1B = sum(R1B.^2);

[B1C,~,R1C] = regress(cut1C , [indeces1C' , ones(numel(indeces1C),1)]);

R1C = sum(R1C.^2);

R1 = R1A+R1B+R1C;
```

Regression of second cut:

```
[B2A,~,R2A] = regress(cut2A , [indeces2A' , ones(numel(indeces2A),1)]);

R2A = sum(R2A.^2);

[B2B,~,R2B] = regress(cut2B , [indeces2B' , ones(numel(indeces2B),1)]);

R2B = sum(R2B.^2);

[B2C,~,R2C] = regress(cut2C , [indeces2C' , ones(numel(indeces2C),1)]);

R2C = sum(R2C.^2);

R2 = R2A+R2B+R2C;
```

Regression of third cut:

```
[B3A,~,R3A] = regress(cut3A , [indeces3A' , ones(numel(indeces3A),1)]);
```

```
            R3A = sum(R3A.^2);

            [B3B,~,R3B] = regress(cut3B , [indeces3B' , ones(numel(indeces3B),1)]);

            R3B = sum(R3B.^2);

            [B3C,~,R3C] = regress(cut3C , [indeces3C' , ones(numel(indeces3C),1)]);

            R3C = sum(R3C.^2);

            R3 = R3A+R3B+R3C;
```

Regression of fourth cut:

```
            [B4A,~,R4A] = regress(cut4A , [indeces4A' , ones(numel(indeces4A),1)]);

            R4A = sum(R4A.^2);

            [B4B,~,R4B] = regress(cut4B , [indeces4B' , ones(numel(indeces4B),1)]);

            R4B = sum(R4B.^2);

            [B4C,~,R4C] = regress(cut4C , [indeces4C' , ones(numel(indeces4C),1)]);

            R4C = sum(R4C.^2);

            R4 = R4A+R4B+R4C;
```

Add error and its corresponding (k_1, k_2, k_3) pairing to existing R.

```
            R = [R ;

                R1+R2+R3+R4 , k1 , k2 , k3 , B1A',B1B',B1C' , B2A',B2B',B2C' ,
B3A',B3B',B3C', B4A',B4B',B4C'];

        end
    end
end
```

Picking best breakpoint permutation

The optimizing permutation of (k_1, k_2, k_3) will be the one with the lowest Residual Sum of Squares.

```
Rmin = min(R(:,1));

optimized_index_pair = find(R(:,1) == Rmin);

Ropt = R(optimized_index_pair,:);
```

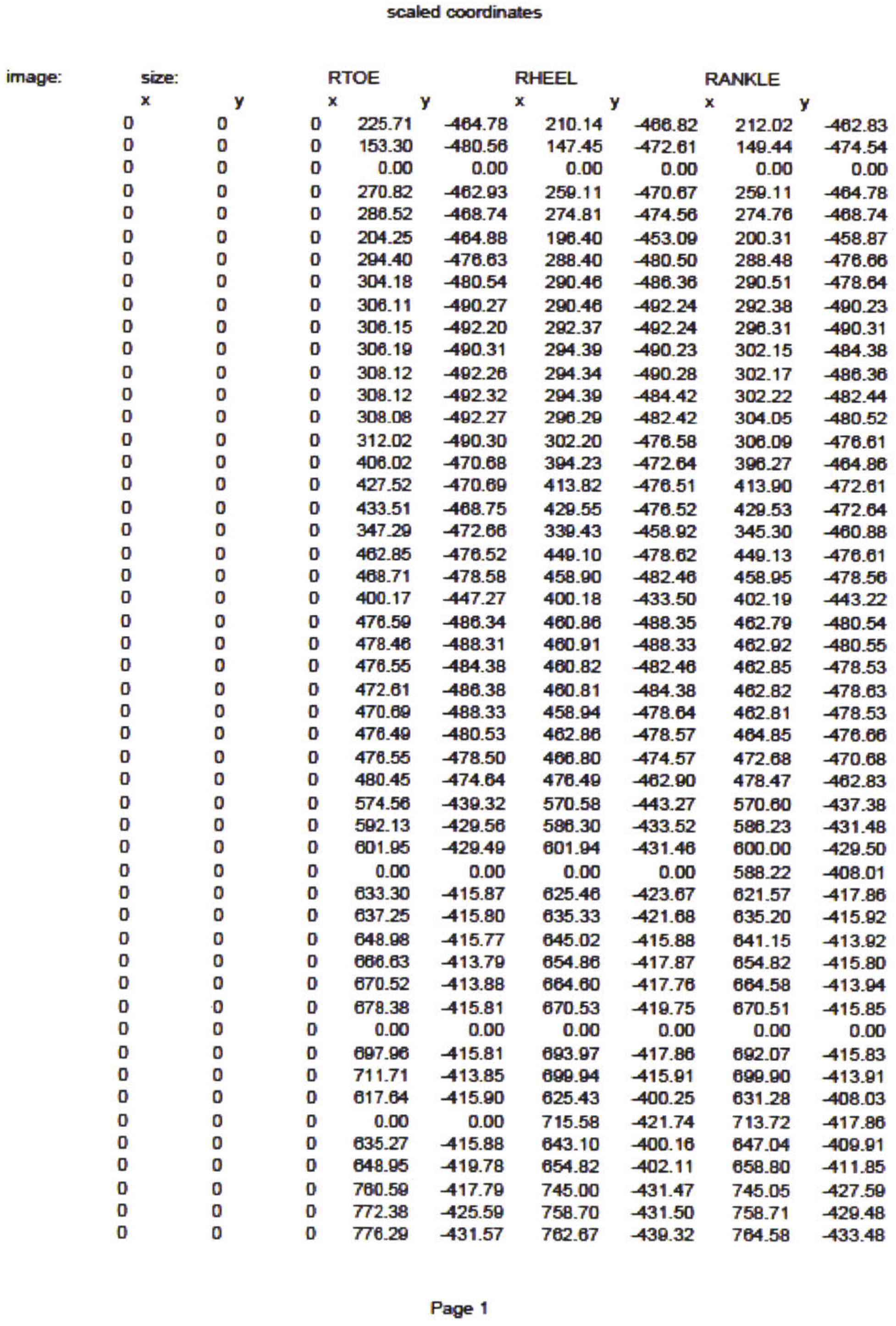
```
k1 = Ropt(2);

k2 = Ropt(3);

k3 = Ropt(4);

B1A = Ropt(5:6);

B1B = Ropt(7:8);

B1C = Ropt(9:10);

B2A = Ropt(11:12);

B2B = Ropt(13:14);

B2C = Ropt(15:16);

B3A = Ropt(17:18);

B3B = Ropt(19:20);

B3C = Ropt(21:22);

B4A = Ropt(23:24);

B4B = Ropt(25:26);

B4C = Ropt(27:28);



indeces1 = 1:k1-1;

indeces2 = k1:k2-1;

indeces3 = k2:k3-1;

indeces4 = k3:numel(signalAy_no_out);
```

Plotting

```
a = figure(j)

subplot(3,1,1)

plot((1:numel(signalAy_no_out))' , signalAy_no_out , '-');
hold on
plot((indeces2)', (B2A(1)*indeces2'+B2A(2)),'-');
plot((indeces1)' , (B1A(1)*indeces1'+B1A(2)),'-');
plot((indeces3)' , (B3A(1)*indeces3'+B3A(2)),'-');
plot((indeces4)' , (B4A(1)*indeces4'+B4A(2)),'-');
plot(k1*ones(100,1),linspace(min(signalAy_no_out)-10,max(signalAy_no_out)+10,100),'k-
');
plot(k2*ones(100,1),linspace(min(signalAy_no_out)-10,max(signalAy_no_out)+10,100),'k-
');
plot(k3*ones(100,1),linspace(min(signalAy_no_out)-10,max(signalAy_no_out)+10,100),'k-
');
improvePlot
ylim([-510,-390])
xlim([-1,numel(signalAy_no_out)+1])
ylim([-510,-380])

subplot(3,1,2)
plot((1:numel(signalBy_no_out))' , signalBy_no_out , '-');
hold on
plot((indeces2)', (B2B(1)*indeces2'+B2B(2)),'-');
plot((indeces1)' , (B1B(1)*indeces1'+B1B(2)),'-');
plot((indeces3)' , (B3B(1)*indeces3'+B3B(2)),'-');
plot((indeces4)' , (B4B(1)*indeces4'+B4B(2)),'-');
plot(k1*ones(100,1),linspace(min(signalBy_no_out)-10,max(signalBy_no_out)+10,100),'k-
');
plot(k2*ones(100,1),linspace(min(signalBy_no_out)-10,max(signalBy_no_out)+10,100),'k-
');
plot(k3*ones(100,1),linspace(min(signalBy_no_out)-10,max(signalBy_no_out)+10,100),'k-
');
improvePlot
ylim([-510,-390])
xlim([-1,numel(signalAy_no_out)+1])
ylim([-510,-380])

subplot(3,1,3)
plot((1:numel(signalCy_no_out))' , signalCy_no_out , '-');
hold on
plot((indeces2)', (B2C(1)*indeces2'+B2C(2)),'-');
plot((indeces1)' , (B1C(1)*indeces1'+B1C(2)),'-');
plot((indeces3)' , (B3C(1)*indeces3'+B3C(2)),'-');
plot((indeces4)' , (B4C(1)*indeces4'+B4C(2)),'-');
plot(k1*ones(100,1),linspace(min(signalCy_no_out)-10,max(signalCy_no_out)+10,100),'k-
');
plot(k2*ones(100,1),linspace(min(signalCy_no_out)-10,max(signalCy_no_out)+10,100),'k-
');
plot(k3*ones(100,1),linspace(min(signalCy_no_out)-10,max(signalCy_no_out)+10,100),'k-
');
improvePlot
ylim([-510,-390])
xlim([-1,numel(signalAy_no_out)+1])
ylim([-510,-380])
end
```

Published with MATLAB® R2019a

Appendix B: Sample of OpenPose Data, each row is a frame



scaled coordinates

image:	size:		RTOE		RHEEL		RANKLE	
	x	y	x	y	x	y	x	y
0	0	0	225.71	-464.78	210.14	-466.82	212.02	-462.83
0	0	0	153.30	-480.56	147.45	-472.61	149.44	-474.54
0	0	0	0.00	0.00	0.00	0.00	0.00	0.00
0	0	0	270.82	-462.93	259.11	-470.67	259.11	-464.78
0	0	0	286.52	-468.74	274.81	-474.56	274.76	-468.74
0	0	0	204.25	-464.88	196.40	-453.09	200.31	-458.87
0	0	0	294.40	-476.63	288.40	-480.50	288.48	-476.66
0	0	0	304.18	-480.54	290.46	-486.36	290.51	-478.64
0	0	0	306.11	-490.27	290.46	-492.24	292.38	-490.23
0	0	0	306.15	-492.20	292.37	-492.24	296.31	-490.31
0	0	0	306.19	-490.31	294.39	-490.23	302.15	-484.38
0	0	0	308.12	-492.26	294.34	-490.28	302.17	-486.36
0	0	0	308.12	-492.32	294.39	-484.42	302.22	-482.44
0	0	0	308.08	-492.27	296.29	-482.42	304.05	-480.52
0	0	0	312.02	-490.30	302.20	-476.58	306.09	-476.61
0	0	0	406.02	-470.68	394.23	-472.64	396.27	-464.86
0	0	0	427.52	-470.69	413.82	-476.51	413.90	-472.61
0	0	0	433.51	-468.75	429.55	-476.52	429.53	-472.64
0	0	0	347.29	-472.66	339.43	-458.92	345.30	-460.88
0	0	0	462.85	-476.52	449.10	-478.62	449.13	-476.61
0	0	0	468.71	-478.58	458.90	-482.46	458.95	-478.56
0	0	0	400.17	-447.27	400.18	-433.50	402.19	-443.22
0	0	0	476.59	-486.34	460.86	-488.35	462.79	-480.54
0	0	0	478.46	-488.31	460.91	-488.33	462.92	-480.55
0	0	0	476.55	-484.38	460.82	-482.46	462.85	-478.53
0	0	0	472.61	-486.38	460.81	-484.38	462.82	-478.63
0	0	0	470.69	-488.33	458.94	-478.64	462.81	-478.53
0	0	0	476.49	-480.53	462.86	-478.57	464.85	-476.66
0	0	0	476.55	-478.50	466.80	-474.57	472.68	-470.68
0	0	0	480.45	-474.64	476.49	-462.90	478.47	-462.83
0	0	0	574.56	-439.32	570.58	-443.27	570.60	-437.38
0	0	0	592.13	-429.56	586.30	-433.52	586.23	-431.48
0	0	0	601.95	-429.49	601.94	-431.46	600.00	-429.50
0	0	0	0.00	0.00	0.00	0.00	588.22	-408.01
0	0	0	633.30	-415.87	625.46	-423.67	621.57	-417.86
0	0	0	637.25	-415.80	635.33	-421.68	635.20	-415.92
0	0	0	648.98	-415.77	645.02	-415.88	641.15	-413.92
0	0	0	666.63	-413.79	654.86	-417.87	654.82	-415.80
0	0	0	670.52	-413.88	664.60	-417.76	664.58	-413.94
0	0	0	678.38	-415.81	670.53	-419.75	670.51	-415.85
0	0	0	0.00	0.00	0.00	0.00	0.00	0.00
0	0	0	697.96	-415.81	693.97	-417.86	692.07	-415.83
0	0	0	711.71	-413.85	699.94	-415.91	699.90	-413.91
0	0	0	617.64	-415.90	625.43	-400.25	631.28	-408.03
0	0	0	0.00	0.00	715.58	-421.74	713.72	-417.86
0	0	0	635.27	-415.88	643.10	-400.16	647.04	-409.91
0	0	0	648.95	-419.78	654.82	-402.11	658.80	-411.85
0	0	0	760.59	-417.79	745.00	-431.47	745.05	-427.59
0	0	0	772.38	-425.59	758.70	-431.50	758.71	-429.48
0	0	0	776.29	-431.57	762.67	-439.32	764.58	-433.48

Page 1

7 Bibliography

[1] Augustyn, Adam “Long Jump - Athletics.” Encyclopaedia Britannica, August 16, 2016. https://www.britannica.com/sports/long-jump.

[2] “Men's long jump world record progression” Wikipedia, February 2019. https://en.wikipedia.org/wiki/Men%27s_long_jump_world_record_progression.

[3] Statsulli, Dominique “The Benefits of Electronic Training Feedback.” Freelap, 2015. https://www.freelapusa.com/the-benefits-of-electronic-training-feedback/.

[4] Ross Anderson, Andrew Harrison & Gerard M. Lyons “Accelerometry-based Feedback – Can it Improve Movement Consistency and Performance in Rowing?“ *Sports Biomechanics*, *4*:2, 179-195

[5] Eriksson, M., Halvorsen, K. A., & Gullstrand, L. (2011). “Immediate effect of visual and auditory feedback to control the running mechanics of well-trained athletes.” *Journal of Sports Science, 29*:3, 253–262.

[6] Phillips, E., Farrow, D., Ball, K., & Helmer, R. (2013). “Harnessing and understanding feedback technology in applied settings. *Sports Medicine,* 43, 919-925.

[7] “OpenPose”. Perceptual Computing Lab (Carnegie Mellon University). Feb 2019. https://github.com/CMU-Perceptual-Computing-Lab/openpose.

[8] “Long Jump Services.” Sports and Safety Surfaces. https://www.sportsandsafetysurfaces.co.uk/athletics/long-jump.

[9] Agola, R. “Long jump and triple jump measurements in 3D.” YouTube. https://www.youtube.com/watch?v=dF-3NhWOM1o.

[10] Linthorne, N.P., Guzman, M.S., and Bridgett, L.A. (2005). Optimum take-off angle in the long jump. *Journal of Sports Sciences*, *23* :7, 703-712.

[11] Brown, Douglas. “Tracker Video Analysis and Modeling Tool.” *Open Source Physics*. https://www.compadre.org/osp/items/detail.cfm?ID=7365.