2010

Automatic object detection and tracking in video

Isaac Case

Automatic Object Detection and Tracking in Video

Isaac Case

isaaccase@gmail.com

A Thesis Submitted in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Computer Science

Department of Computer Science
Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, NY

May 2010

Thesis Chairman: Professor Roger S. Gaborski 20 May 2010

Thesis Reader: Professor Peter G. Anderson 20 May 2010

Thesis Observer: Yuheng Wang 20 May 2010

This page left intentionally blank.

Contents

List of Figures

Listings

Abstract

One ability of the human visual system is the ability to identify and track moving objects. Examples of this can easily be seen in any sporting event. Humans are able to find an object in motion and track its current path and even predict a trajectory based on its current motion. Computer vision systems exist that are able to track an object in video, but usually these systems need to be instructed what the object to track is. As a way to further the work done by these computer vision systems, I present two additions to the work in the form of Adaptive Thresholding, a way to dynamically discover a threshold value of difference images, and a new method of blob tracking to further improve the accuracy of tracking blobs in video.

1 Introduction

In the field of computer vision, systems exist that attempt to track objects in video. This is in fact two different problems. One problem is that of determining what is an object to track, and the other problem is to track an object once it has been determined that it is worth tracking.

For systems that need to determine what is an object worth tracking, there must be some way of determining what content in the video is foreground content and what is background content. One of the simplest way of doing this is to pick an image that is representative of the background. Then, in order to determine the foreground content, a difference is calculated between the current frame and the frame that has been designated background. If the video was a perfect image of the frame, this could be a simple task as all difference between the perfect background and the perfect current frame could be considered foreground content.

Unfortunately when dealing with natural images this is not the case. In the real world, with real world sensors, noise and variation exist everywhere. Even between two images that appear to be the same, there can be slight differences. These differences mean that there

exist areas in each of the images that are not exactly the same. This does not allow us to choose all differences to be foreground content. In order to deal with the issue of noise, one method is to choose a threshold point. Differences larger than this threshold are deemed foreground content. Differences less than the threshold point are considered background content.

Although simple in nature, picking a threshold value that works in all videos is not possible. Different cameras and different lighting conditions create different levels of noise. This also can be a problem within the same video if the lighting conditions change enough. Proposed in this paper is a method that is able to automatically determine a threshold value based on the differences between two images of the same video and this threshold value allows for a clean separation of foreground and background content.

The other problem mentioned is that of once an object is determined to be a foreground content image worth tracking then tracking the image becomes the next task. Many of the current techniques match objects from frame to frame, but don't attempt to create alignment between the objects frame to frame. A second proposal in this paper is a method of blob matching that not only is able to determine a match metric for use with determine matches from frame to frame, but also is able to provide an alignment vector to show how to align the objects in motion from one frame to the next.

2 Literature Search

There have been many examples of people using background subtraction as a part of a motion detection and tracking systems. Going as far back as the work done by Cai et al. in 1995, their system relied on using a threshold to display the difference between two frames [1]. PFinder developed by Wren et al also followed this model, although it had a more complex background modeling approach [19]. Even more modern systems such as the work done by

Fuentes et al. more recently in 2006 showed the need to threshold in modern systems [3]. Although Fuentes et al. claimed that there did exist an appropriate threshold that could be used, it was based on a much more complex model of the background [8]. Alternatives also include adapting the background for more accurate differencing as proposed by Huwer et al. [5]. Also discussed in the literature is a mixture Gaussian model by Stauffer et al. [18]. More and more complex models also include a technique called Sequential KD approximation [4] and Eigenbackgrounds [12], however these will not be discussed in detail as they have large memory and cpu requirements [14].

Spagnolo et al. [17] described a method to perform background subtraction using the radiometric similarity or a small neighborhood, or window within an image. The radiometric similarity is defined in figure 1. Using the radiometric similarity on windows of pixels within an image, according to Spagnolo et al. reduces the effects of pixel noise due to the fact that radiometric similarity uses a local interpretation of differences rather than a global pixel based interpretation. Unfortunately this technique also relies on a selection of a threshold value. This is necessary since the radiometric similarity is a continuous value, and needs to be thresholded, where if the radiometric similarity is above the threshold, then it must be foreground content, otherwise it must be background content. Also, the size of the windows used, according to Spagnolo et al. must be determined experimentally [17]. They further the task of background subtraction by including a frame differencing method, and combine the two for the highest degree of success. This is done by comparing the areas of motion detected by frame difference with the background model, using the radiometric similarity metric, and using that final result as the foreground content.

$$R = \frac{mean(W_1W_2) - mean(W_1)mean(W_2)}{\sqrt{variance(W_1)variance(W_2)}}$$

Figure 1: Radiometric similarity between two windows, W_1,W_2.

Matsuyama et al [11] also show the effectiveness of using a windowed view of the frames

for differencing. They use the window of size $N \times N$ as a vector of size N^2. The comparison is made between the current frame image $i(u, v)$ and the background image $b(u, v)$. The comparison made is the normalize vector distance. The normalized vector distance can be seen in figure 2. Matsuyama et al. also claim that the problem of thresholding is eliminated by the fact that an appropriate threshold value can be obtained by calculating the mean of the NVD and the variance of the NVD and the threshold value is $mean+2\times variance$. They also claim that due to the nature of the NVD, this metric is invariant to changes in lighting. The only stipulations presented by this method is the fact that for the calculations of the mean and variance of the NVD the standard deviation σ of the noise component of the imaging system must be calculated. This means that every time a new imaging system is introduced, a new σ must be calculated.

$$\text{NVD} = \frac{i(u, v)}{|i(u, v)|} - \frac{b(u, v)}{|b(u, v)|}$$

Figure 2: Normalized vector difference, where $|i(u, v)|$ and $|b(u, v)|$ are the magnitudes of the vectors i and b.

More recently Mahadevan et al. [9] presented a new method of background subtraction specifically targeted at dynamic scenes. These include scenes such as objects in the ocean, where there is a large amount of motion in the background, but humans are still able to detect non-background (foreground) motion in these scenes. Mahadevan et al.'s method relies on the calculation of the saliency of the image. Specifically the saliency of a sliding window is calculate for the image. Unfortunately, this method also requires the use of a threshold value for foreground detection. Although this method does improve the accuracy of foreground detection for scenes with dynamic backgrounds, it requires a threshold value to determine at what saliency level an area becomes foreground instead of background. One advantage of this method, however, is the fact that they Mahadevan et al. claim that there is no need to calculate a full background model and much simpler models can be used to

predict the motion of the background. They also claim that this method is invariant to camera motion, due to the fact that they are using saliency instead of pixel differencing.

One problem with these alternatives, however, according to Piccardi is that each one, as it gets further and further away from the simple background subtraction is that it incurs increased processing time and memory consumption[14]. Another problem with most of these techniques is the fact that a threshold value must be selected for some part of the calculations. This threshold value may change depending on the imaging system used and the lighting conditions of the video begin processed.

As far as blob matching goes, much of the work has been to just determine a best match, as in the work of Masoud et al. [10]. The goal in this case was not to search the blobs for exact overlap. For most cases this good enough, however, in some cases, it would be optimal to get a more precise location of each blob. Other examples of this coarse grained approximation is prevalent in Cai et al.'s method describing tracking as "... a matter of choosing moving subject with consistency in space, velocity, and coarse features such as clothing and width/height ratio."[1]. Fuentes et al. also made assumptions about tracking such as the fact that once individuals become part of a group, there is no attempt to separate the person from the whole [3]. No further attempt is made to separate the individual, or match an individual from within a group or occluded scene.

More recently Song et al. [16] described an updated method for object tracking in video. The method presented requires two different models for each object being tracked. The two models are an appearance model and a dynamic model. The appearance model used by Song et al is the color histogram of the object being tracked. The dynamic model used is the Kalman filter for location prediction of the object. According to Song et al.'s method, matching from frame to frame is as follows. First the location of all the objects are predicted based on the dynamic model. Then based on overlap with the objects in the current frame one of three things can be done. If there is only one object to be matched in the new location,

then that match is take. If there exist many objects in the current frame that match only one object in the previous frame, then the multiple objects in the current frame are combined into one large object and matched to the one object in the previous frame. If there exist multiple objects in the previous frame that match to one object in the current frame, then the one large object in the current frame must be broken down in to multiple objects to match with the previous frame objects. The breaking down and matching is performed by a mean-shift color tracking method as defined by Ramesh et al. [15].

Collins et al. [2] described another method of blob tracking using feature matching. The features described by Collins et al. are linear combinations of the RGB colors in the image. They are weighted by whole integer value coefficients in the set of $\{-2, -1, 0, 1, 2\}$ and normalized to the range 0-255. This creates features such as $R-G$, which is $1\times R+-1\times G+0\times B$, or $2G-B$. Collins et al. claims that there exist out of these combination of values 49 distinct features. Unfortunately, they also claim that the selection of the best features to be used for a image sequence must be chosen experimentally as one set of features will not be best for all types of images. Given an object to be tracked, the features for the object and the features for the area surrounding the object are calculated. Then a feature transform is computed that creates a more strict object feature list vs background feature list for this object. This helps isolate the image from the background. The most discriminating features are used to create a weight image, where the pixels of most weight are the pixels that most likely represent the object to be tracked. This weight image is used in a mean-shift algorithm to search for the object in the current frame.

Liang et al. [7] furthered the work done by Collins et al. to add adaptive feature selection and scale invariant, or scale adaption, tracking. The method for adding adaptive feature selection is based on the Bayes error rate between the current frame and the calculated background frame for each feature for each object. This allows an automatic selection of the N most important features to be used for tracking. The scale adaptation, or scale invariance

introduced is based on the use of four different correlation templates, one for each boundary of the objects rectangle, to attempt to search for the boundaries of the object in the current frame. This allows for the object to grow in all four directions, and still find a match.

3 Adaptive Thresholding

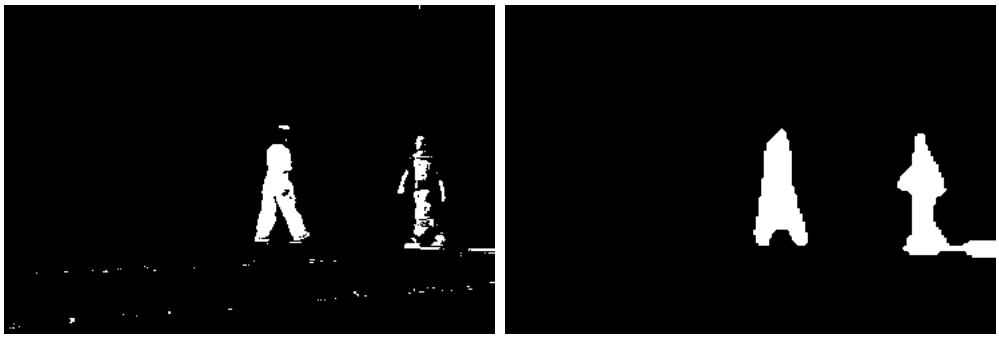
3.1 Design

The main goal of adaptive thresholding is to determine an appropriate threshold value to threshold a difference image, as in figure 3. This is somewhat similar in goal to Otsu's method for threshold selection[13] for which a threshold value is to be selected to convert grey scale images into black and white images. However, where Otsu's method is an attempt to preserve contrast in gray scale images when converted to black and white, the adaptive threshold method developed here is to determine appropriate threshold values when converting a difference image to black and white. Applying Otsu's thresholding methods to difference images produces non optimal results as can be seen in figure 4.

(a) Original (b) Diff

Figure 3: An example reference frame (a) and the difference image generated for that frame (b).

An alternative approach has been developed that can convert a gray scale difference

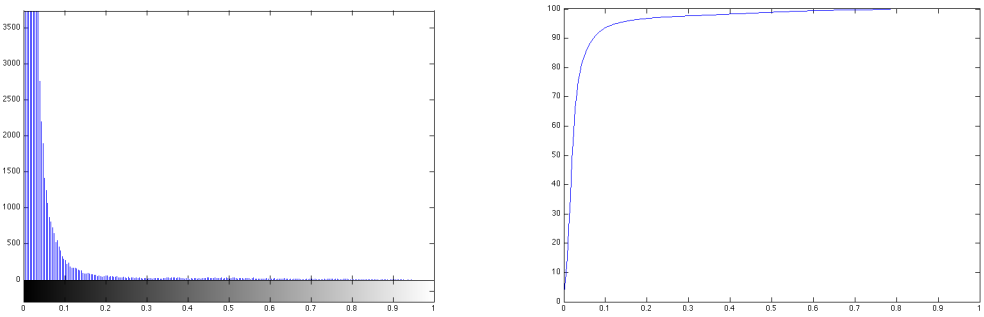


(a) Otsu's Method (b) Adaptive Thresholding

Figure 4: Comparing the results of using Otsu's method to threshold a difference image (a) to Adaptive Thresholding (b).

image into a one bit monochrome image that represents the areas of difference, or in this specific usage, areas of motion.

The only input to adaptive thresholding is a difference image. The difference image for two different images a and b can be represented as diff$= |a - b|$. An example of this can be seen in figure 3(b). A key characteristics of this difference image is the fact that a majority of its values are close to zero. This can be seen more easily in the histogram of the difference image as in figure 5(a).



(a) Difference Histogram (b) Difference Cumulative Sum

Figure 5: A sample histogram of a difference image (a) and its accompanying cumulative sum.

The search for the threshold point begins with the cumulative sum. After observing many of these difference images and their cumulative sums, there appeared to be a point where the change in slope appeared to reach zero. This is equivalent to saying that there was a point where the second derivative, $\frac{d^2y}{dx^2}$, approached zero. This is plainly visible when observing the second derivative as in figure 6. This is the optimal threshold point. The point at which the second derivative converges to zero is an optimal threshold point.

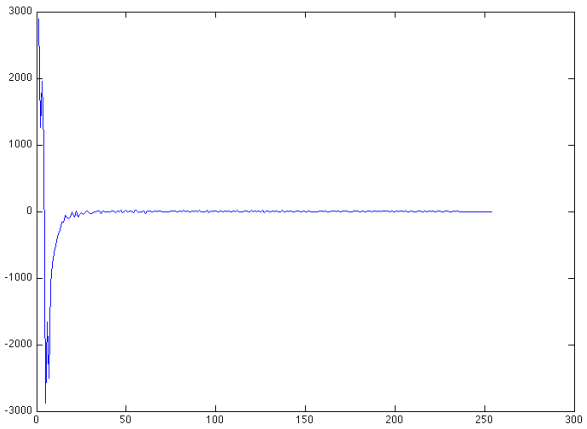


Figure 6: The second derivative of the cumulative sum.

Since the difference image cannot be guaranteed to be smooth, it needs to be smoothed before searching for the threshold value. This can be done with an averaging filter. An averaging filter is applied as the convolution of the image with the 11x7 matrix in figure 7. This forces this difference image to be smooth, which will produce a more connected outcome when thresholded, or in other words, the objects in motion will appear to be full objects and not partial objects. The difference can be seen comparing figure 8(a) and figure 8(b).

The final step in producing optimally thresholded difference images is to use morphological processing to reduce the size of each of the newly formed objects. This is due to the effect of the strong averaging filter. The average filter extends the area covered by each object. In order to get a better representation of each object, the image must be morphologically

$$
\begin{array}{c} \\ 11 \end{array}
\begin{array}{c} 7 \\ \left[\begin{array}{ccccc}
\frac{1}{77} & \frac{1}{77} & \cdots & \frac{1}{77} & \frac{1}{77} \\
\frac{1}{77} & \frac{1}{77} & \cdots & \frac{1}{77} & \frac{1}{77} \\
\vdots & \vdots & \ldots & \vdots & \vdots \\
\frac{1}{77} & \frac{1}{77} & \cdots & \frac{1}{77} & \frac{1}{77} \\
\frac{1}{77} & \frac{1}{77} & \cdots & \frac{1}{77} & \frac{1}{77}
\end{array}\right] \end{array}
$$

Figure 7: An averaging filter used to smooth difference images.

(a) Threshold without averaging filter

(b) Threshold with averaging filter

Figure 8: Comparing the filtered vs. unfiltered results of thresholding.

eroded. This results in the final thresholded difference image as presented in figure 4(b)

3.2 Analysis

Adaptive Thresholding is an attempt to solve two problems. One is the problem of determining an optimal threshold value across multiple videos. The other is the problem that an optimal threshold value may not be constant throughout a single video due to changes in lighting or other attributes.

For the first goal, that of picking a suitable threshold value for multiple different videos, three different videos were analyzed. One contained people walking outdoors, another people walking indoors in a lab environment, and people walking and sitting in an indoor living room setting.

Using Adaptive Thresholding, we can see the thresholds picked for each video in figure 9. The values chosen for adaptive thresholding range from approximately 0.035 to 0.120 where the range of possible values range between 0 and 1. This represents a range of possible threshold values of almost 10% of the total dynamic range of the image. If we were to use either of the extreme threshold values on the opposite image, the results would be very poor. An example of this is shown in figure 10 where the threshold value of 0.035, which was optimal for the indoor lab setting, was used for the indoor living room setting video.

Observing one of the outdoor videos, we can see that the threshold value chosen by Adaptive Thresholding can change dramatically over time, as in figure 12.

3.2.1 Failures

Adaptive Thresholding fails under certain conditions. One major condition is synthetic images. The problem that arises with synthetic images is the fact that there is no noise in the image. This causes a major problem due to the fact that with this type of data, the second derivative of the cumulative sum is almost always zero for the entire set of data due

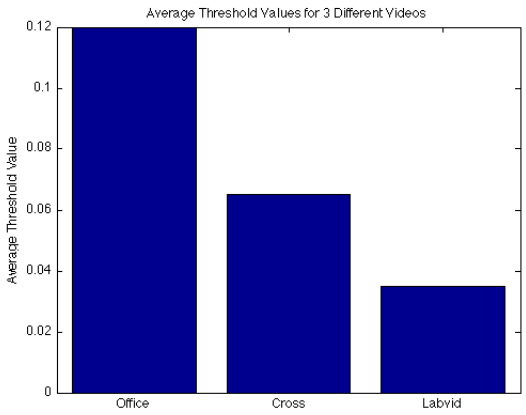


Figure 9: Average threshold values picked by Adaptive Thresholding.

(a) Threshold mask when using 0.035 as threshold value

(b) Threshold mask when using Adaptive Thresholding

(c) Frame image when using 0.035 as threshold value

(d) Frame image when using Adaptive Thresholding

Figure 10: Comparing the results of thresholding when using a suboptimal value that is too low (0.035) (a)(c) compared to Adaptive Thresholding (b)(d).

(a) Threshold mask when using 0.12 as threshold value

(b) Threshold mask when using Adaptive Thresholding

(c) Frame image when using 0.035 as threshold value

(d) Frame image when using Adaptive Thresholding

Figure 11: Comparing the results of thresholding when using a suboptimal value that is too high (0.12) (a)(c) compared to Adaptive Thresholding (b)(d).

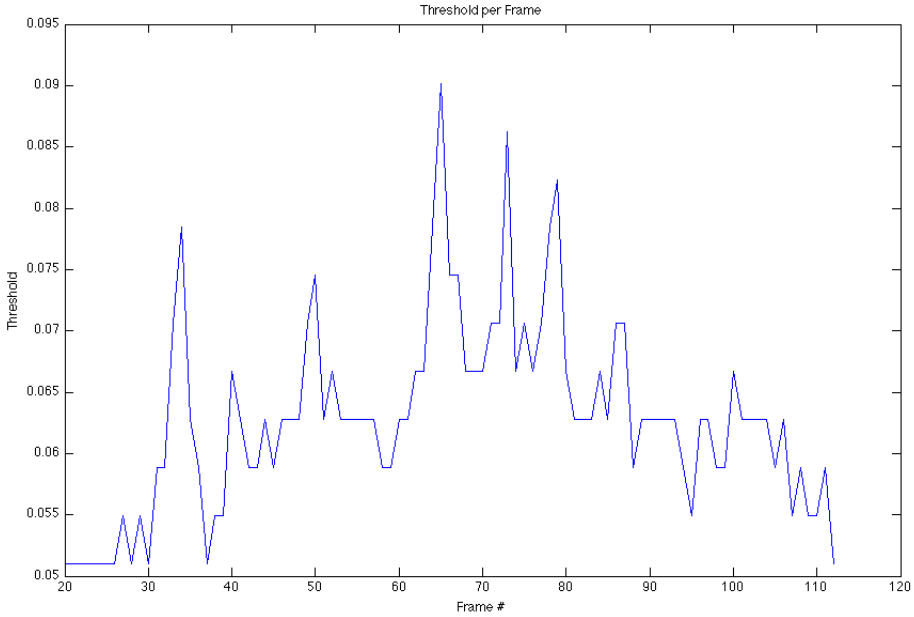


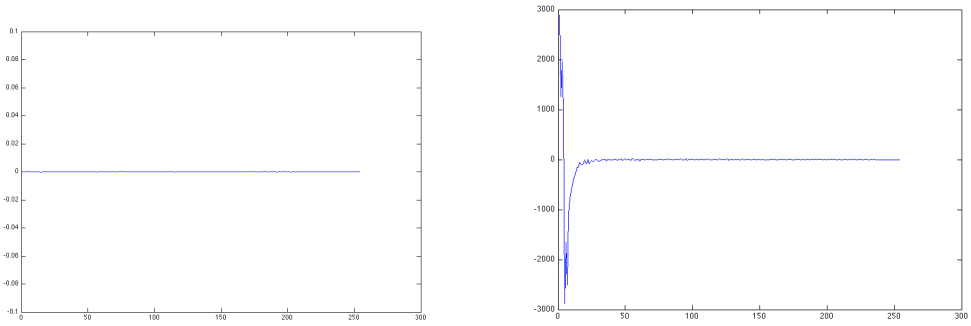
Figure 12: The threshold values chosen by Adaptive Thresholding over the series of frames.

to the harsh edges and lack of noise, see figure 13. In this case, a required failure test case must be checked for. In these cases, the search for a point at which the second derivative converges toward zero fails because the second derivative is always zero. In order to provide some threshold value, a value of 0.0001 may suffice in these cases.

4 New Method of Blob Matching

4.1 Design

Once an image has been segmented into foreground and background content, it then becomes the task of finding individual objects and matching them frame to frame. The individual objects in the foreground will hereafter be referred to as blobs. As part of this task it becomes necessary to match blobs. To accomplish this task a blob matching algorithm was designed



(a) Synthetic (b) Natural

Figure 13: Comparing the second derivative of the cumulative sum of the histogram for the difference image of synthetic images vs. natural images.

and constructed.

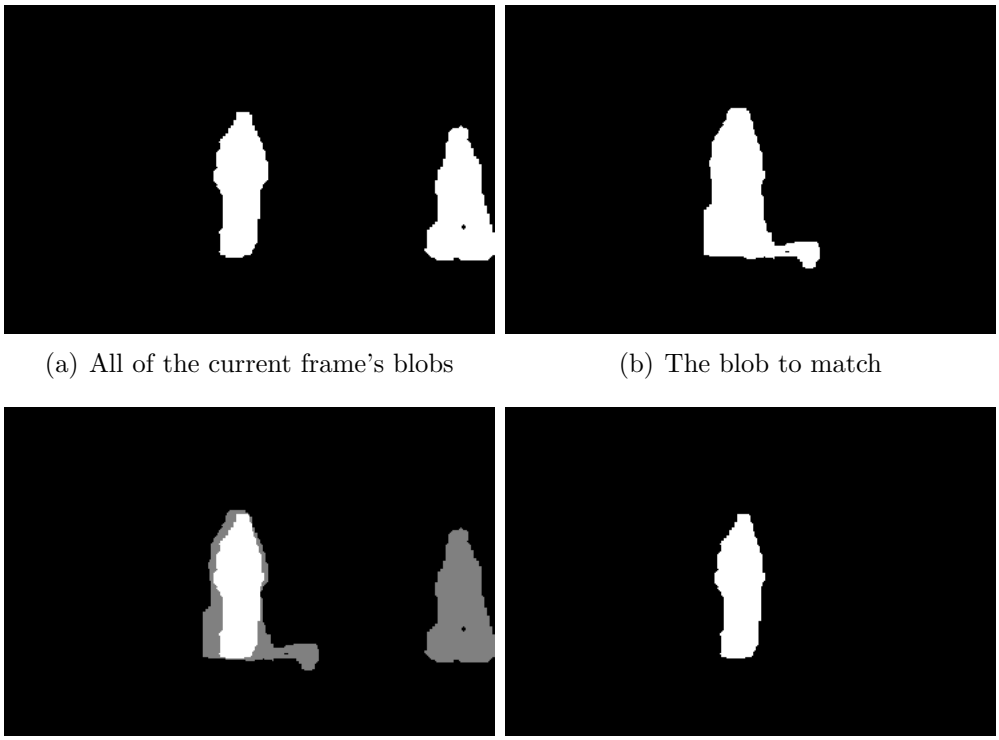
The main goal of the blob matching algorithm is to take two set of blobs and generate a new set of blobs that represents the combination of both input sets. Since there may exist overlap between the two sets of blobs, i.e. the 'same' blob may exist in both sets, when this overlap occurs, the blobs should be merged together into a new blob before being entered into the new set. This is similar to a set union, but where elements that are very similar, but not necessarily exactly the same will be merged into a new element that represents both of the original elements.

As stated, the input to algorithm should be two sets of blobs. Both sets are allowed to contain zero elements which is the trivial case, but most of the design is targeted for the case where both sets contain at least one element. For the trivial cases, if both of the sets are empty, then an empty set is returned. If only one of the sets is empty, then the set returned is exactly the same as the non-empty set.

Designing for the non-trivial case is as follows. One of the input sets of blobs is treated as a historical list of blobs, or the blobs that have existed in past frames, and will be referred to as B_p. The other set represents the blobs that exist in the current frame, referred to as

B_c. The newly created set of blobs, referred to as B_n where $B_n = B_p \cup B_c$, is initialized as the empty set {}. To add blobs to B_n, a forward matching is performed. The blobs in B_p are matched to B_c.

The matching is performed one blob, in B_p, at a time in two steps. First, a reduction is made in B_c so that only blobs for which there is spacial overlap are considered, see figure 14. Given a frame for which there are multiple blobs, this can greatly reduce the amount of computation time as the non overlapping blobs are not even considered for a match.



(a) All of the current frame's blobs (b) The blob to match

(c) Overlapping both images (d) The overlap reduction

Figure 14: Reducing a full frame of blobs (a) down to the region that overlaps with a previous frame's blob(b) produces (d).

Second, given a reduced number of blobs to match, the remaining blobs must be ranked for likely matches. This ranking is based on similarity where the metric is based on the relative similarity of the image data associated with the blob itself. This is done to provide

a more accurate match and also to provide alignment of image data from one frame's blob to the next.

The alignment of blobs is based on the normalized cross correlation [6]. The normalized cross correlation is a way to measure the correlation between two signals. This can be applied to images to search for the location where the two images have the highest correlation. The normalized cross correlation requires two images as input, one being called the 'image', I, and the other called the 'template', T. The only restrictions are that the template must be at most the same size as the image in all dimensions. Unfortunately the normalized cross correlation alone does not provide an adequate method for matching blobs. Two important changes need to be added.

Given the normalized cross correlation of two similar, but not identical images, it is likely that there will be regions that claim to have the highest correlation, but are mostly noise. This can be due to the fact that since the most blobs are not rectangular, but the inputs for the normalized cross correlation must be full in all dimensions, so for a 2D cross correlation it must be a rectangle. Since the blobs are not rectangular, but must be rectangular, there may exist empty borders, or the region from the blob to the boundary of the rectangle may be all zeros. In this region, it may create a noisy spike in the correlation results which is not desired. In order to reduce this type of noise in the normalized cross correlation, I first apply a median filter over the results, see figure 15 which removes this type of noise very well.

Another heuristic applied to the normalized cross correlation is that of a weight matrix. The assumption is that given the amount of time that passes between two frames there is not a lot of motion. Therefore for the weighting of the correlation, there is a center weighted Gaussian weight distribution created, see figure 16(a). This is the basis of the weights applied to the correlation of the image. If desired, the center weighted average can be shifted in any direction based on the last known location of one of the blobs, see figure

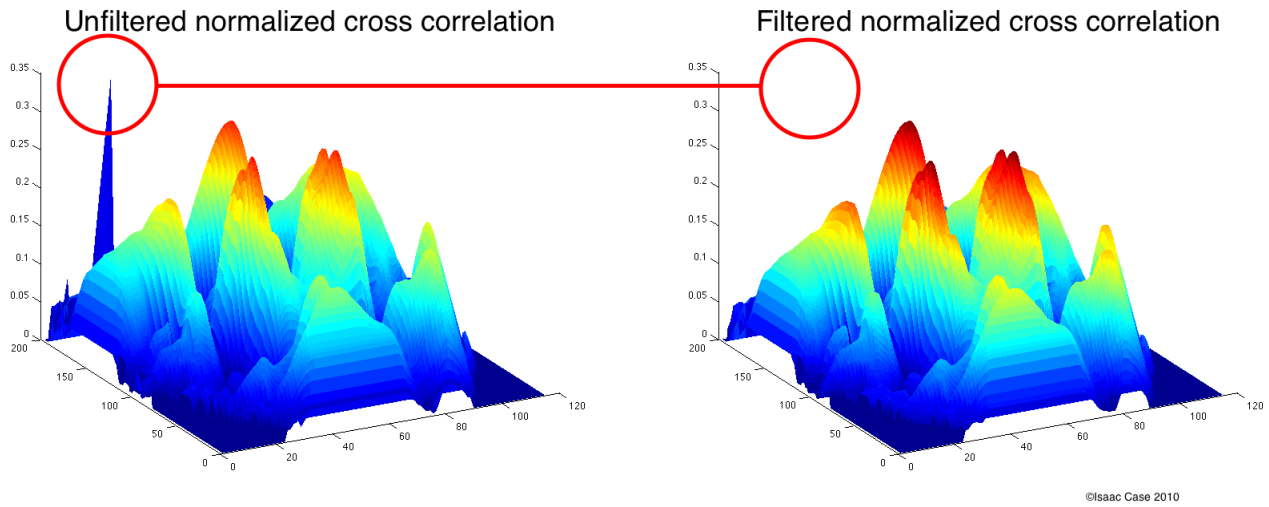
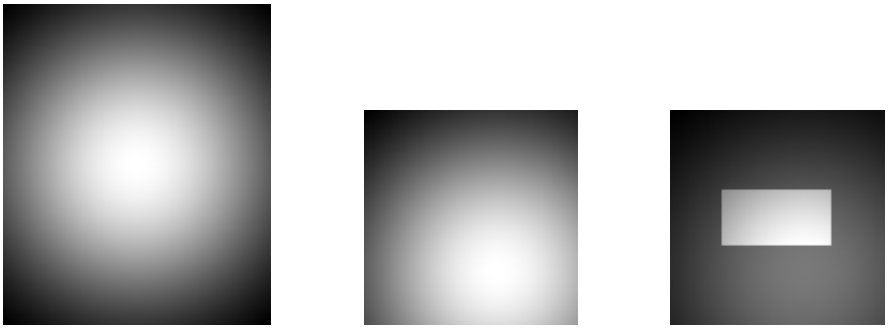


Figure 15: Comparison of an unfiltered results of the normalized cross correlation and the results after applying a median filter.

16(b). Also, the region of overlap, since the results of the normalized cross correlation does contain the potential for only a one pixel overlap so an additional weight or bias is added for the regions of full overlap, or the region of the result of the normalized cross correlation for which the template, when shifted, is completely inside of the image, see figure 16(c).



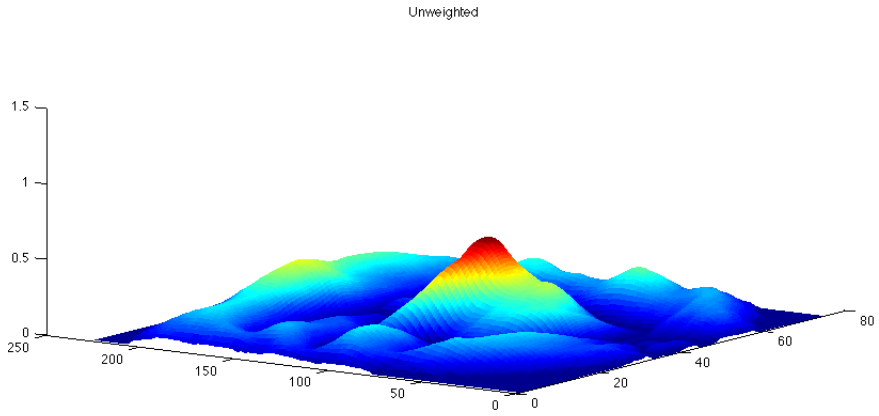
(a) A centered Gaussian (b) A shifted Gaussian (c) A shifted biased Gaussian

Figure 16: A centered Gaussian weight matrix (a), one that has been shifted in the X and Y direction (b), and one that has been shifted and biased for fully overlapping regions (c).

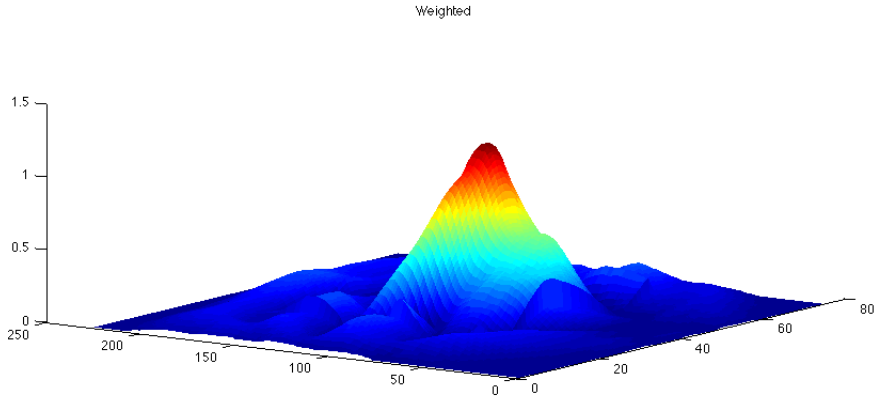
Given the weight matrix for the normalized cross correlation, the final matrix used for

$$M_{(i,j)} = N_{(i,j)} \cdot W_{(i,j)}$$

Figure 17: The match matrix based on the normalized cross correlation and a set of weights.



(a) Unweighted Normalized Cross Correlation



(b) Weighted Normalized Cross Correlation

blob matching, M, is the result of combining the filtered normalized cross correlation, N, with the weight matrix, W, element by element as in figure 17. The match matrix, or weighted normalized cross correlation can be seen visually in figure 4.1. From this match matrix, the necessary alignment vector, A, can be derived. This is accomplished by locating the element with the largest magnitude. The row and column of this element can be used to determine the offset as seen in figure 18

$$
\begin{aligned}
M_{(x,y)} &= \text{MAX\_POSITION}(M) \\
T_{(x,y)} &= \text{Template size} \\
A_{(x,y)} &= M_{(x,y)} - T_{(x,y)}
\end{aligned}
$$

Figure 18: Determining the alignment vector based on the match matrix.

From the alignment vector A, it is now possible to compare the image and template and see how close the match is between the two images. Based on some method of difference between the two images a match value can be assigned. The difference calculation can be any form of comparison that performs a pixel by pixel comparison between the two images for only the region which they overlap. In my current approach three different difference metrics were tried. One method was to take the absolute value of the difference of each color (RGB) for each pixel and then take the maximum value, or the color of most difference, for each pixel. Other alternatives are to take the Euclidean distance in RGB space for each pixel, or take the Euclidean distance in L*a*b* space for each pixel, see figure 19.

$$dE_{RGB} = \sqrt{(I_R - T_R)^2 + (I_G - T_G)^2 + (I_B - T_B)^2}$$

$$dE_{L*a*b*} = \sqrt{(I_{L*} - T_{L*})^2 + (I_{a*} - T_{a*})^2 + (I_{b*} - T_{b*})^2}$$

Figure 19: Computing the Euclidean distance in RGB and L*a*b* space.

From the difference between the image and the template, a match value can be associated with this alignment. The match value is calculated as a sum of each pixels difference

normalized on the size of the image. This is a value that represents how well the overlapping regions match based on some other difference function other than the normalized cross correlation.

$$\text{mv} = 1 - \frac{\sum_{x=1}^{m}\sum_{y=1}^{n} dE(x,y)}{m \times n}$$

Figure 20: The match value calculation for an image and template that overlap m $\times$ n pixels.

4.2 Analysis

Given this new technique of blob matching, how well did it perform? For most cases it performed fairly well and can be seen in figure 21 and results in a match value of 0.7073. This can be considered a successful match.

In most scenarios, if the blob that is trying to be matched has only changed slightly, then the probability of a successful match is greatly increased. If, however, the blob has changed dramatically since the last time a match was attempted, then the likelihood of a successful match decreases dramatically.

Unfortunately, there also exist many cases where this matching fails. In many cases this is due to occlusion by other objects or another blob. In these cases, the match of the full object may not be successful due to the fact that the whole object may not be visible.

Other failure cases include instances where the object has changed in size. If an object has changed in size, the normalized cross correlation cannot match the object with proper placement. If the texture of the object to match is too similar to the background, this can also cause problems as the normalized cross correlation does not take into account the color differences. This may be helped by adding the hue component of HSV color space, although this approach has not been investigated.

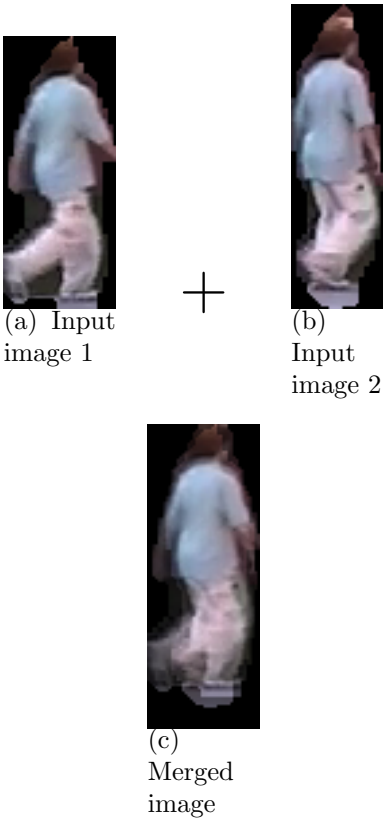


Figure 21: Merging two images (a) (b) using new blob matching and the resultant merged image (c).

5 Full System

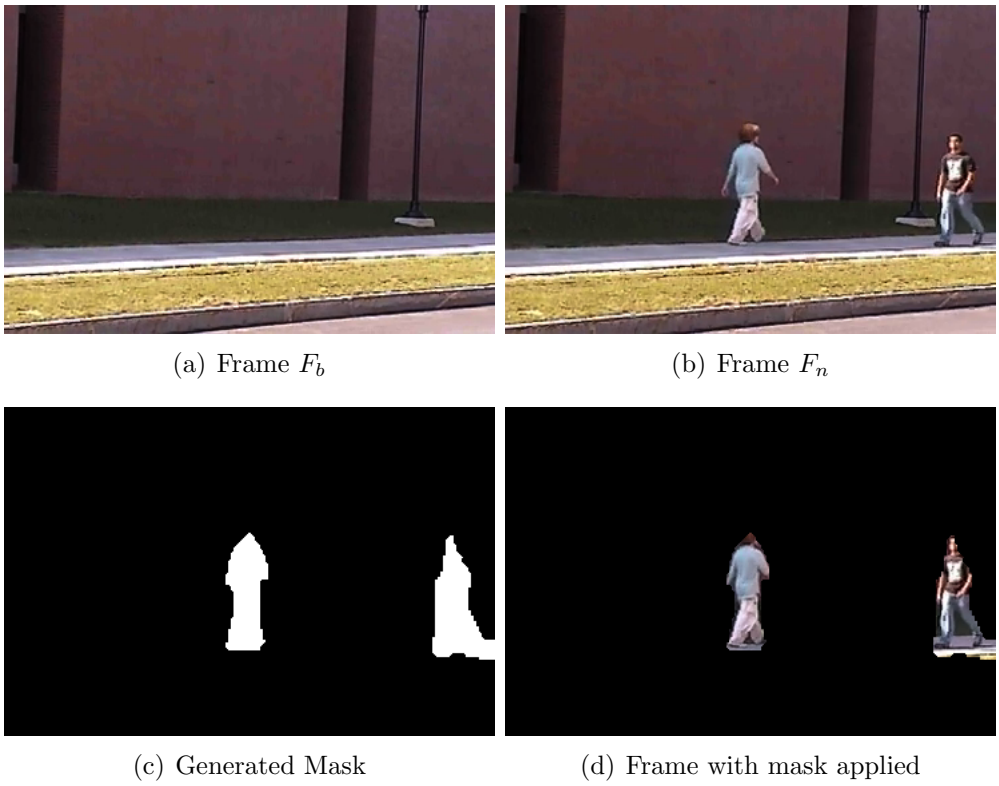
5.1 Design

In order to test out the adaptive thresholding and blob matching in a more real world setting an entire system needed to be created that effectively employed both of these techniques. The goal of the full system was to take a video as input and produce another video as output for which all the moving objects are detected and tracked. One other goal of the system was to create a tracking system that required few if any tunable parameters. In this system, the number of tunable parameters was reduced to one, which is the minimum size of a blob worth tracking. The tracking is displayed as a colored rectangle surrounding the tracked blob. Each blob is identified as a unique color. If a blob moves from one frame to the next and the color stays the same, then the tracking succeeded. If the color changes, then it is classified as a failure.

In order to process videos, it becomes necessary to view the video as a series of frames. Specifically, each frame must be analyzed individually. In theory, an entire video could be loaded and then decomposed into individual frames. In the implementation this work is done ahead of time, allowing memory conservation by only having to load a few frames at a time instead of the entire video or all of the frames at once.

In order to perform the adaptive thresholding at least two frames are needed to perform the difference operation on. For the system that was implemented, two different methods are supported. One is a static frame difference, or if there is a single frame that represents the background, it can be selected as the background image. All adaptive thresholding is performed on the difference of the current frame, F_n, and that background frame, F_b. An example can be seen in figure 22. The other alternative method allowed is temporal differencing or frame differencing. For this method the frames before and after, frames F_{n-1} and F_{n+1}, must be observed. The operation performed is similar, but instead of just

comparing two frames, there must be two comparisons, one between F_n and F_{n-1}, and a comparison between frame F_n and F_{n+1}. This can be seen in figure 23.

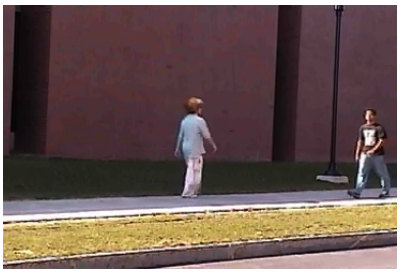


(a) Frame F_b (b) Frame F_n

(c) Generated Mask (d) Frame with mask applied

Figure 22: Background Subtraction between frame F_b(a) and F_n(b) results in the mask (c) shown with image (d).

Once a frame has been converted from the original input image to a frame mask, the individual blobs can be extracted. This is where the one parameter in the system is used. Any blobs that are smaller than the minimum size parameter are removed from the image before further processing. With the blobs from the current frame, merge those blobs with the blobs of the previous frames. If no blobs exist in the previous frames, then just accept the blobs from the current frame as the accepted set of blobs. Otherwise, perform a forward matching, or a matching from the blobs from previous frames to the blobs in the current frame. The matching algorithm is the same as described in section 4.1.



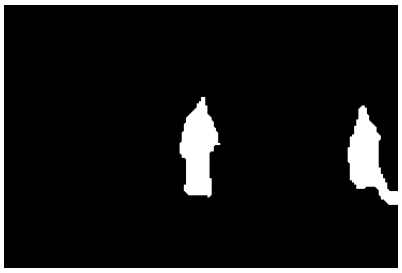
(a) Frame F_{n-1}

(b) Frame F_n

(c) Frame F_{n+1}

(d) Mask from F_{n-1} and F_n

(e) Mask from F_n and F_{n+1}



(f) Mask from the combination of $M_{n,n-1}$ and $M_{n,n+1}$

(g) Frame with mask applied

Figure 23: Temporal Differencing.

After a match has been performed on the blobs in the current frame, then construct a new image that will represent the current frame with matched and tracked content. This is done by first making a copy of the current image, then drawing a rectangle around each blob, see figure 24. The color of the rectangle drawn is representative of the blobs individuality.

Figure 24: A frame with the blobs that have been tracked draw with a surrounding rectangle.

5.2 Analysis

The design of the system employs two new techniques, or two extensions on currently used techniques. One advantage is that there are very few parameters to tune. The only directly tunable parameter is that of the minimum blob size. With only setting that one parameter I was able to take a new video and processes it with the system and get reasonable results. The video was somewhat simple, it was just one object moving at a constant velocity in one direction, but the system was able to detect and track the object without any tuning of threshold values.

There are still some unsolved problems. As described in section 3.2, synthetic images pose problems for the tracking system. Also, blobs that overlap for long periods of time, or blobs that are partially occluded for an extended period of time can confuse the system. This is due to the method of blob matching.

If temporal differencing is used instead of background subtraction, large objects may not be detected correctly. This can be seen with objects larger than approximately $\frac{1}{2}$ of the frame. The use of static background subtraction is also suboptimal as it does not take into consideration the changes that may occur to the background as time changes. The static background subtraction with adaptive thresholding does perform better than with a static threshold value, but overall the system may be able to perform better if a better background model was employed.

6 Software Design

6.1 Software design of Adaptive Thresholding

The adaptive threshold has been separated out to be a single function in matlab code. It can be seen in its entirety in Appendix A. The input is assumed to be a gray scale image (matrix) with values between 0 and 1. If the image is not 120 pixel high, the image is resized to be 120 pixels high using Matlab's *imresize* function. In order to apply the strong averaging

Listing 1: Resizing the input image

```
diff_im = imresize(input_image, [120 NaN]);
```

filter, a filter of size 11x7 is created with all values in the filter being $\frac{1}{77}$. The filter is applied with *imfilter* specifically in listing 2.

With the filtered difference image, the next step is to find the point in which the second derivative converges to 0. In order to calculate this, first a histogram is calculated from

Listing 2: Use of imfilter

```
filtered_image = imfilter(diff_im, averaging_filter, 'replicate');
```

the filtered image using *imhist*, then a cumulative sum is calculated with Matlab's *cumsum* function. As a way of normalization, each element in the cumulative sum is divided by the total sum. This results in a cumulative sum function for which the total is 1. Given the calculate cumulative sum of the histogram, an approximation of the second derivative is calculated with the *diff* operation. *diff* works by calculating the difference between neighboring elements in the vector that is given as input i.e.:

Listing 3: Matlab's approximation of a derivative

```
diff(X) = [X(2)–X(1) X(3)–X(2) ... X(n)–X(n–1)]
```

This is applied twice to the normalized cumulative sum to give an approximation of the second derivative of the cumulative sum. From this second derivative approximation, we want to determine the point at which the second derivative converges to 0. This is also approximated by searching for the last element in the approximated second derivative of the cumulative sum that is larger that 0.005. In Matlab it can be implemented as follows:

Listing 4: Using the second derivative to locate the convergence point

```
sec_der_thresh = sec_der;
sec_der_thresh(abs(sec_der)>0.005) = 1;
idx = max(find(sec_der_thresh == 1)) + 1;
```

This determines the point at which the second derivative converges to zero, but what is really desired is where that point is in the 0..1 range of values that is in the difference image. In order to accomplish this I use the newly found index into the second derivative, and use that as an index into the bins of the histogram. The only modification needed is to add

two to the index due to the fact that the *diff* function reduces the number of elements in a vector by one each time it is used. Therefore:

Listing 5: Location of the threshold

```
thresh_point = centers(idx+2);
```

where 'centers' is a vector containing the center points of the bins used in the histogram calculation.

Using this threshold point, I can convert the filtered gray scale difference image to a binary mask. This is done by using the threshold point as a threshold value.

Listing 6: Thresholding an image in Matlab

```
hist_based_diff = averaged_bw;
hist_based_diff(averaged_bw<thresh_point) = 0;

diff_mask = logical(hist_based_diff);
```

The main issue presented with using the threshold value with the filtered image is that due to the strong averaging filter, the threshold enlarges the area surrounding the moving object too much. To counteract this growth, the next step is to apply morphological processing to erode the binary image. For my implementation I used a structure element of a disk that had a $\sqrt{10}$ large radius.This is implemented in Matlab as:

Listing 7: Applying morphological processing

```
diff_mask = imerode(diff_mask, strel('disk', round(sqrt(10))));
```

In order to return a mask that was the same size as the original input image, we now resize the image back to the same size as the input image:

Listing 8: Restoring the mask to original size

```
image = imresize(diff_mask, restore_size);
```

6.2 Software design of Blob Matching

The new method of blob tracking has also been developed in Matlab. There are four inputs to this section. They are:

- A list of the previously detected blobs
- A list of the blobs detected this frame
- The current frame
- The motion mask of the current frame

Listing 9: Blob Matching Prototype

```
function people = compare_people(old_people, new_people, frame,
    full_mask)
```

The result of the function should be a list of the blobs that exist in the current frame as they have been matched to the previous frames blobs. Each blob is a structure that contains the following entries:

- 'im' - a RGB representation of the blob.
- 'mask' - a 1 bit representation of the blob.
- 'pos' - the (x,y) coordinates of the blob.
- 'count' - the number of times this blob has been seen.
- 'name' - a unique single character identifier.

- 'color' - the color used to draw a bounding box around this blob.

If the list of previously detected blobs is empty, the current list of blobs is returned as the current status since there was nothing to match. The only change made to the current frames detected blobs is that their count, or the number of frames that the blob was detected in is increased from zero to one.

If this is not the case, the main work is then to match the previous frames blobs to the current frame. Two arrays are maintained to show the connection from the previous frames blobs to the current frames blobs. Going forward, from the previous frame to the current frame, the array track which blob in the current frame is the best match for the previous frame's blob, if there is one. Going backwards, the array for the current frame's blobs tracks the number of previous frame blobs match each individual current frame blob. These are all initialized to zero and new list is created to represent the tracked blobs, both old and new, that exist for this frame.

In order to easily match the current frame's input mask to the currently found blobs I convert the frame's mask to a labeled image, see listing 10. The labeled image is exactly the same dimensions as the mask, but instead of only containing ones and zeros it contains integer values where each distinct integer is an entity. An example of this type of conversion can be seen in figure 25

Listing 10: Resizing the input image

```
full_mask_lab = bwlabel(full_mask);
```

For each of the previously found blobs that were passed in, the following is performed.

1. Create a blank monochromatic image that is exactly the same size as the current frame

```
mask_im = zeros(size(full_mask));
```

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 2 & 2 \\ 0 & 1 & 0 & 2 & 2 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 3 & 3 & 0 \end{bmatrix}$$

(a) Binary Image (b) Labeled Image

Figure 25: Conversion from a binary image(a) to a labeled image(b).

2. Place the mask of the blob we are trying to match in the blank image exactly where it was in the previous frame

```
mask_im(y1:y2,x1:x2) =
    old_people(x).mask(1:y2-y1+1,1:x2-x1+1,1);
```

3. Use the placed mask as an index to zero out all entries in the labeled image except for where the previous mask existed.

```
overlap = full_mask_lab;
overlap(mask_im==0) = 0;
```

4. Determine all unique non-zero values in the overlap image.

```
possible_matches = unique(overlap);
possible_matches(possible_matches==0)=[];
```

5. If there are multiple possible matches, then determine which of the possible matches is the best match.

The function *xcorr_match* is too large to describe here, but it is a major part of the new blob matching technique. The full source can be seen in listing 19 in appendix B and a full description in section 6.2.2.

```
if size(possible_matches,1) > 1
    match = possible_matches(1);
    match_val = 0;
    for ind = possible_matches'
        [c r match_v] = xcorr_match(old_people(x).im,
           new_people(ind).im);

        if match_v > match_val
            match_val = match_v;
            match = ind;
        end
    end
```

Once this has been done, the forward and backward match arrays are fully populated. The next step is to merge the matches. This is done in four steps.

1. If a blob in the current frame has no match from a previous frame, add it to the list of current blobs.

2. Split blobs where multiple previous frame blobs match to one current frame blob and merge them.

3. Merge the rest of the previous frame blobs to the blobs in the current frame that they were previously matched to.

4. If a blob from previous frames has no match to the current frame, decrease the number of times that the blob has been seen and add it to the current list of blobs.

If at any time there exist a blob that has not been seen in the last four frames, it is dropped from the list of current blobs.

Splitting the blobs in the current frame that had multiple matches from previous frames is done in two steps. Looking at all the previous blobs that match the current blob individually, first use *xcorr_match* with a shifted probability distribution that weights the previous location of the previously found blob the highest. This will find the region that most likely contains

the previously found blob in the new blob. Once this region is found it is merged with the previous frame's blob. The merge is completed with the *merge* function. The *merge* function will be covered in section 6.2.1.

For blobs from the previous frame where there is only one match the combining operation is as simple as one call to the merge function.

6.2.1 merge

The merge function is an attempt to merge two blobs based the image data contained in those blobs. First it uses *xcorr_match* to determine the exact location of the best match overlap. Depending on whether or not there is a considerable amount of overlap, specifically defined here in listing 6.2.1, there are two different options.

```
if sum(merged_mask(:)==1)/sum(old_person.mask(:)) < .4
```

If there is considered to be enough overlap, then the merged person is just the result of merging the two images at the correct overlap point as defined by *xcorr_match* and updating the correct x and y locations.

If this is not the case, then the merged blob will be the product of the merged image and the complete frame for where that blob would have been. This allows missing content to become part of the blob if the current frame blob is considerably smaller that it had been in the past.

The function *merge* contains an optional argument to force that no merged image is created. In this case, the old image is used to represent the blob, but the blob is updated with the best guess location of the blob in the new frame as calculated by *xcorr_match*.

6.2.2 xcorr_match

The function *xcorr_match* is the core of the new blob matching technique. The inputs for the function are two different images and an optional offset. The optional offset is a way to change the center of a Gaussian probability distribution, if there is no offset, then the probability will be centered at the center of the image. The full source can be seen in listing 19

Lines 2-48 perform some initialization and checks on the input. Based on the sizes of the images, one of the image must be the 'image' and the other be the 'template', see lines 16-25. Also, for the use of Matlab's normxcorr2 function we have to use gray scale images, so lines 27-37 convert to gray if needed. Also, in order to use normxcorr2 the 'image' image must be at least the same size if not larger in all dimensions compared with the 'template' image. If this is not the case, lines 39-45 create a new image that is padded with empty pixels in the deficient dimension.

Starting on line 50 the real heavy lifting begins. The basis for the match is the *normxcorr2* function in Matlab. *normxcorr2* performs a normalized cross correlation using an input image and a template. The resultant matrix can then be used to determine the location of most correlation, or in other words, the best match[6]. The two different options are whether or not we are processing each color channel separately and then combining, lines 50-60, or if we just deal with the image converted to gray scale only, line 62. Either way, at this point there exists a matrix c that is the result of running *normxcorr2* between both of the input images.

In order to make sure there are no unusual spikes in the results of the normalized cross correlation a median filter is applied to the matrix, see line 65. If it weren't for this median filter, some undesired results may appear that are noise and not the real maximum values in the matrix, see figure 15

From the results of the filtered normalized cross correlation a match could be made,

however, since the match should be weighted most heavily based on the previous location, a Gaussian weighting is applied in lines 67 - 103. The Gaussian weighting is constructed originally as a center weighted Gaussian, as in figure 16(a) and line 91. Based on an optional offset value, the center of the probability distribution can be shifted in either the X or Y direction, as in figure 16(b), and is performed by cropping the original Gaussian in the required direction in line 70-89 and 93. In order to weigh more heavily regions that are completely overlapping, i.e. where the smaller image is completely contained in the larger one, the section of overlap is multiplied by 2 and can be seen in figure 16(c) and line 100. This Gaussian weight distribution is applied to the results of the normalized cross correlation, see line 103.

Based on the results of the normalized cross correlation and the weight matrix that has been constructed, a maximum value can be searched for. Specifically, I am looking for the absolute value of the results of the normalized cross correlation, see lines 105-108.

Given this match, the rest of the function is a way to combine the two input images and calculate a difference image between the two input images. Lines 113-138 combine both images based on the previously calculated offset. Lines 160-191 calculate the difference between the two images. As an example, there are three different ways to perform the difference of the two images.

- Calculate the difference of each channel (R,G,B) separately and then takes the largest difference value out of the three channels, lines, lines 169-172.
- Calculate the difference as the Euclidean difference in RGB space, lines 175-178.
- Calculate the difference as the Euclidean difference in CIELAB space (dE), lines 181-185.

6.3 Software design of the rest of the system

In order to implement the system I relied heavily on Matlab's development environment. Specifically I required the signal processing toolkit and image processing toolkit. Since Matlab's processing of video is very memory intensive, the current approach is to decompose any video file that is to be processed into a series of images. That way individual images could be loaded without requiring all images or the entire video be loaded in memory at the same time.

For full system design see section 7.

7 Developer's Manual

As a developer, the areas of most interest are what the individual components are and what their functions are. The sequence diagrams describing the interactions between the components are in figures 26 and 27

The main components are:

- MotionTracker
- setup_video
- CreateDiffMask
- ApplyDynamicThreshold
- compare_blobs
- xcorr_match
- merge
- merge2im

- box

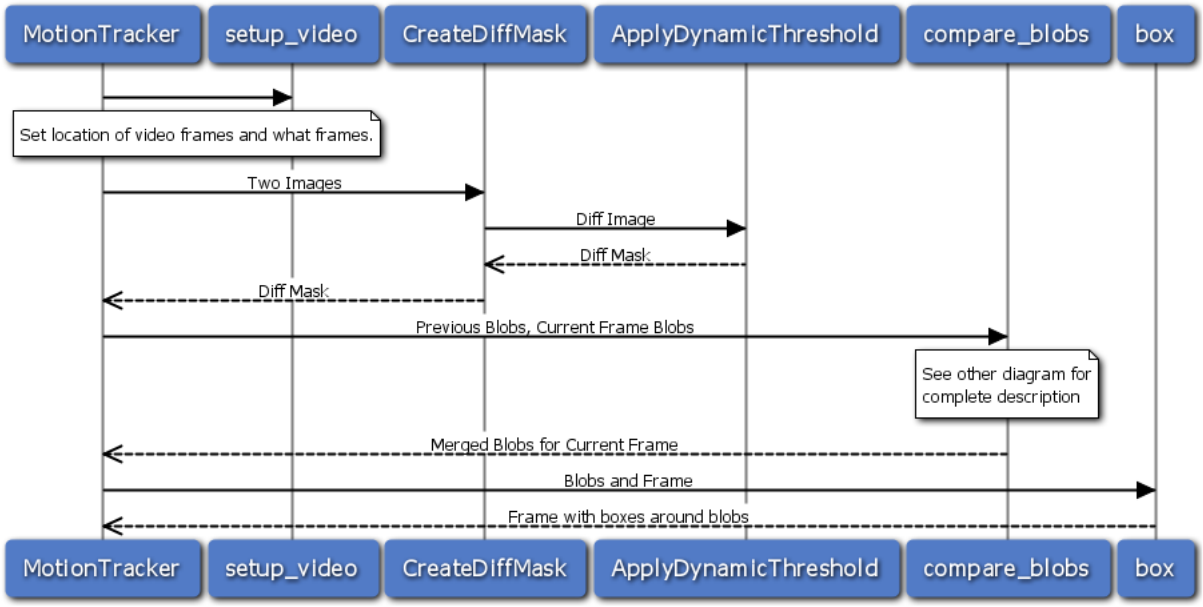


Figure 26: The full sequence diagrams of the Motion Tracker system without compare_blobs.

The *MotionTracker* module is the main starting point of the code. It is this module that loads in image data and iterates over all the frames of data. In this module it is possible to set the choice between background subtraction and frame differencing. The *MotionTracker* module does not anticipate any inputs, but can return a list of all the thresholds used for each frame for diagnostic purposes.

The *setup_video* module is a location to put the file paths to the image sequence, what the number of the starting frame is, and what number frame to stop at. This is also the location where you can set which frame is representative of the background if background subtraction is desired. *setup_video* is the only module that is not a function, therefore it does not have any input parameters or return values.

CreateDiffMask creates the mask used to extract blobs from the current frame. It does this by :

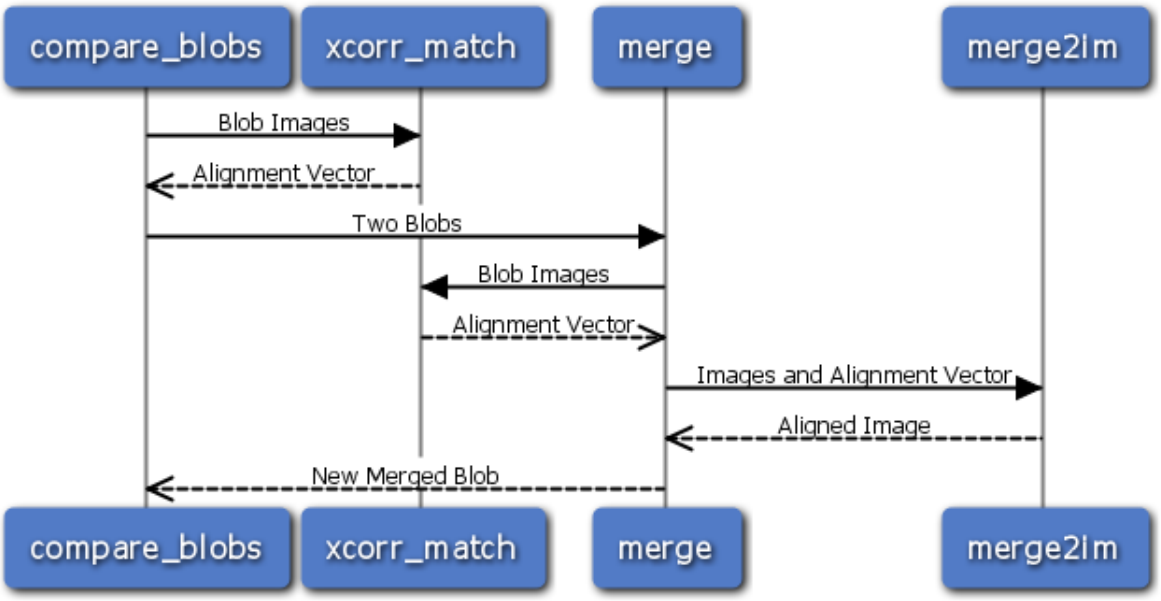


Figure 27: The sequence diagram of the compare_blobs function.

1. Convert the image to gray (if it is not already)
2. Sub sample the image so that it is at most 120 pixels high
3. Apply a 3×3 Gaussian filter to both of the images
4. Obtain the diff image $D = |F_1 - F_2|$ where D is the difference image, F_1 is one of the frames and F_2 is the other frame.
5. Threshold the difference image with *ApplyDynamicThreshold*

CreateDiffMask has the prototype of

Listing 11: *CreateDiffMask* function prototype

```
function [diff_mask Threshold]= CreateDiffMask(image1, image2)
```

ApplyDynamicThreshold determines the correct threshold value of a difference image and applies it. The correct threshold values is determined by the Adaptive Thresholding

algorithm described in section 3.1 and detailed in section 6.1. Its function prototype is: where the inputs are a difference image and an option to not filter the difference image. The

Listing 12: *ApplyDynamicThreshold* function prototype

```
function [image value] = ApplyDynamicThreshold(diff_im, NO_FILTER)
```

return values are both the thresholded image and the specific value of the threshold chose.

compare_blobs performs the blob matching of previous blobs to new blobs as described in section 4.1 and detailed in section 6.2. The function prototype is: where the input arguments

Listing 13: *compare_blobs* function prototype

```
function blobs = compare_blobs(old_blobs, new_blobs, frame,
    full_mask)
```

are the list of blobs from previous frames, the list of blobs from the current frame, the full RGB image of the current frame and the full blob mask for the current frame.

The *xcorr_match* module attempts to determine the best alignment given two blobs, or rather two images of blobs. It is fully described in section 6.2.2. Its functional prototype is: where the inputs are the two different images to align, and an optional *center_shift* vector

Listing 14: *xcorr_match* function prototype

```
function [x y match im diff_im switched] = xcorr_match(im1, im2,
    center_shift)
```

that is the offset from the center for the Gaussian weights. The function returns the shift in the x and y direction, the match value, a combined image that is the two input images combined using the shift values, a difference image between the aligned images and a flag to say whether or not the input images were 'switched' when detecting the shift. The original assumption is that we are matching im1 to im2, but if im1 is larger than im2, then we have to switch for the matching, and the flag is set that a switch happened.

merge merges two blobs as described in section 6.2.1 and has the prototype of: The input

Listing 15: *merge* function prototype

```
function blob = merge(old_blob, new_blob, frame, make_new_im)
```

arguments are the two blobs that are to be merged and the full image of the current frame. Also, a flag is passed in to determine whether or not we would like the new blob to have its image be the same as the previous *old_blob* image, or if it should be the result of combining the images between the *old_blob* and the *new_blob*. This function returns a new blob that represents the combination of both input blobs.

merge2im and *box* are two helper functions to perform a repeated task. *merge2im* merges two images based on an alignment vector and returns a new image, and a difference image, that is the combination of the two images aligned by the alignment vector. *box* draws a box around a given blobs on a frame, given a blob and an image. The function prototypes of these functions are as follows.

Listing 16: *merge2im* function prototype

```
function [merge_im diff_im] = merge2im(im1, im2, x_shift, y_shift)
```

Listing 17: *box* function prototype

```
function boxed_im = box(im, blob)
```

8 User's Manual

As as a user of the system there are only a handful of variables that need to be set to use the Motion Tracker. They are:

- impath
- start_frame
- end_frame
- min_size
- static
- BackgroundSubtraction
- WriteImage
- WriteImagePath

impath is the path where the input video, as a series of images resides. It is of the form of a standard *printf* format string where the location of the number for each image is embedded in the name. An example of this type of naming is the following:

```
impath = 'C:\MyVideoFiles\MyVideo_frame_%03d.png';
```

Would result in the names:

'C:\MyVideoFiles\MyVideo_frame_001.png';

'C:\MyVideoFiles\MyVideo_frame_002.png';

'C:\MyVideoFiles\MyVideo_frame_003.png';

...

'C:\MyVideoFiles\MyVideo_frame_101.png';

'C:\MyVideoFiles\MyVideo_frame_102.png';

start_frame and *end_frame* are the number of which frame to start looking at the video and when to stop. For example, if there is a 150 frame video that one would like analyzed from frames 50 to 125, the variables should be set as such:

```
start_frame = 50;
end_frame = 125;
```

If background subtraction is desired, then the variable *BackgroundSubtraction* should be set to 1, otherwise temporal differencing will be used. Also, the variable *static* should be set to the number of the frame that should be used as the representation of the background.

If the results of the Motion Tracking are desired to be written out to disk, then the variables *WriteImage* and *WriteImagePath* need to be set. *WriteImage* needs to be set to 1 and *WriteImagePath* needs to be set to the path where the images should be written to. The format for *WriteImagePath* is the same as the format of *impath*.

The only variable that actually effects the algorithm is that of *min_size*. *min_size* is the minimum size of an object that should be observed. All objects less that that size will be ignored. *min_size* says that all objects must be at least as large as *min_size* in one dimension, and if it is not, then the other dimension must be at least half as large as *min_size*

MotionTracker is run in the Matlab environment. It is simply with the command:

```
>> MotionTracker
```

It will display the current frame that it is processing in a window with the title of the image displayed as the number of frame that is begin processed. The frame displayed will be the current frame with the bounding boxes surrounding the blobs being tracked.

9 Future Work

Although the improvements made to the motion tracking system greatly help the work done in thresholding and matching, there is still a lot of work left to do.

One major area for improvement is that of performing the same actions, but with a non-stationary camera. Currently there is nothing in the Motion Tracking system that compensates for camera motion. This could potentially be done with an adaptation of the SIFT algorithm, or with an approximation based on tracking other features such as Harris corner points.

Also, the matching algorithm works very well when the object to match is not occluded. It would be very useful to extend the matching algorithms to work on partial matches. Possibly split up the image to be matched and perform multiple sub matches and then combine the results. Other options include attempting to incorporate the general shape, and search for the shape of the blob to be tracked.

The Adaptive Threshold algorithm has been shown to work well for natural images. It may be worthwhile to do a further analysis of why synthetic images generally fail, and see if there is a more appropriate algorithm for synthetic images.

Also, for the Adaptive Threshold algorithm an averaging filter of size 11×7 was chosen as an adequate filter. This has yet to be proven, and there may exist a simpler, optimal filter. It also may be the case that the filter chosen is optimal for tracking people, but may be suboptimal for tracking other objects, such as cars or animals.

10 Conclusion

Two improvements to the field of motion tracking have been suggested in this paper. One being Adaptive Thresholding which help eliminate the need for a chosen threshold value. It also help deal with cases where the threshold value could change from frame to frame

as other conditions change. Although other algorithms exist for separating foreground and background content that do not require thresholding, thresholding a difference image is the least memory intensive and one of the fastest methods. Improving upon this simple method can greatly help when speed and low memory usage is important.

Also discussed is a method for blob matching that attempts to get an accurate match and alignment of two blobs. It was shown that for images that include one blob with no occlusion it can perform well, but has problems when dealing with objects that are occluded. This can include complex scenes that include multiple people.

Future work in these areas has been discussed, and it would prove fruitful to research these areas further.

References

[1] Q. Cai, A. Mitiche, and J.K. Aggarwal. Tracking human motion in an indoor environment. *Image Processing, 1995. Proceedings., International Conference on*, 1:215–218 vol.1, Oct 1995.

[2] RT Collins, Y. Liu, and M. Leordeanu. Online selection of discriminative tracking features. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(10):1631–1643, 2005.

[3] Luis M. Fuentes and Sergio A. Velastin. People tracking in surveillance applications. *Image and Vision Computing*, 24(11):1165 – 1171, 2006. Performance Evaluation of Tracking and Surveillance.

[4] B. Han, D. Comaniciu, and L. Davis. Sequential kernel density approximation through mode propagation: applications to background modeling. In *Proc. ACCV*, volume 2004. Citeseer, 2004.

[5] Stefan Huwer and Heinrich Niemann. Adaptive change detection for real-time surveillance applications. *Visual Surveillance, IEEE Workshop on*, 0:37, 2000.

[6] JP Lewis. Fast normalized cross-correlation. In *Vision Interface*, volume 10, pages 120–123. Citeseer, 1995.

[7] Dawei Liang, Qingming Huang, Shuqiang Jiang, Hongxun Yao, and Wen Gao. Mean-shift blob tracking with adaptive feature selection and scale adaptation. In *Image Processing, 2007. ICIP 2007. IEEE International Conference on*, volume 3, pages III –369 –III –372, sept. 2007.

[8] S.A. Velastin L.M. Fuentes. Foreground segmentation using luminance contrast. *Proceedings of the WSES/IEEE Conference on Speech, Signal and Image Processing*, page 2231–2235, 2001.

[9] Vijay Mahadevan and Nuno Vasconcelos. Background subtraction in highly dynamic scenes. *Computer Vision and Pattern Recognition, IEEE Computer Society Conference on*, 0:1–6, 2008.

[10] O. Masoud and N.P. Papanikolopoulos. A novel method for tracking and counting pedestrians in real-time using a single camera. *Vehicular Technology, IEEE Transactions on*, 50(5):1267–1278, Sep 2001.

[11] T. Matsuyama, T. Wada, H. Habe, and K. Tanahashi. Background subtraction under varying illumination. *Systems and Computers in Japan*, 37(4):77, 2006.

[12] N.M. Oliver, B. Rosario, and A.P. Pentland. A bayesian computer vision system for modeling human interactions. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 22(8):831 –843, aug 2000.

[13] Nobuyuki Otsu. A threshold selection method from gray-level histograms. *IEEE Transactions on Systems, Man, and Cybernetics*, 9(1):62–66, 1979.

[14] M. Piccardi. Background subtraction techniques: a review. In *Systems, Man and Cybernetics, 2004 IEEE International Conference on*, volume 4, pages 3099 – 3104 vol.4, 10-13 2004.

[15] D.C.V. Ramesh and P. Meer. Real-time tracking of non-rigid objects using mean shift. In *i¿ Proc. IEEE Conf. Computer Vision and Pattern Recognition,¡/i*, volume 2, pages 142–149. Citeseer, 2000.

[16] X. Song and R. Nevatia. Robust vehicle blob tracking with split/merge handling. *Multimodal Technologies for Perception of Humans*, pages 216–222, 2006.

[17] P. Spagnolo, T.D. Orazio, M. Leo, and A. Distante. Moving object segmentation by background subtraction and temporal analysis. *Image and Vision Computing*, 24(5):411–423, 2006.

[18] C. Stauffer and W.E.L. Grimson. Adaptive background mixture models for real-time tracking. In *Computer Vision and Pattern Recognition, 1999. IEEE Computer Society Conference on.*, volume 2, page 252 Vol. 2, 1999.

[19] Christopher Richard Wren, Ali Azarbayejani, Trevor Darrell, and Alex Pentland. Pfinder: Real-time tracking of the human body. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(7):780–785, 1997.

Appendices

A Adaptive Thresholding Function

Listing 18: Dynamic Threshold Function

```
1  function [image value] = ApplyDynamicThreshold(diff_im, NO_FILTER)
2
3          if ~exist('NO_FILTER', 'var')
4              NO_FILTER = 0;
5          end
6
7          restore_size = [size(diff_im, 1) size(diff_im, 2)];
8          reduction = [120 NaN];
9
10         diff_im = imresize(diff_im, reduction);
11
12         %%Powerful averaging filter (helps remove small content and
               blend
13         %%large content
14
15         if NO_FILTER == 1
16             averaged_bw = diff_im;
17         else
18             f_avg = fspecial('average', [11 7]);
19
20             averaged_bw = imfilter(diff_im, f_avg, 'replicate');
21         end
```

```
22          %%Determine LAST inflection point in nomalized cumlative sum of
23          %%histogram
24          [n_elements centers] = imhist(averaged_bw);
25          c_elements = cumsum(n_elements);
26          sec_der = diff(diff(c_elements./sum(n_elements(:))));
27
28          %%Find inflection point as last point before second deretive
29          %%flattens (or close enough)
30          sec_der_thresh = sec_der;
31          sec_der_thresh(abs(sec_der)>0.005) = 1;
32          idx = max(find(sec_der_thresh == 1)) + 1;
33
34          if size(idx, 1) == 0
35              thresh_point = 0.0001;
36          else
37          %%Apply inflection point as threshold
38              thresh_point = centers(idx+2);
39          end
40          thresh_percent = c_elements(idx+2)./sum(n_elements(:));
41
42          %%%display(sprintf('Thresholding at %g which crops off
                %g%%\n', thresh_point, thresh_percent*100));
43
44          hist_based_diff = averaged_bw;
45          hist_based_diff(averaged_bw<thresh_point) = 0;
46
47          %%Apply morphological processing to image to reduce effects of
```

```
48          %%average filter
49
50          diff_mask = logical(hist_based_diff);
51
52          if NO_FILTER == 0
53              diff_mask = imerode(diff_mask, strel('disk',
                    round(sqrt(10))));
54          end
55          image = imresize(diff_mask, restore_size);
56          value = thresh_point;
57  end
```

B Blob Matching Function

Listing 19: Matching Function

```
1  function [x_shift y_shift match new_im diff_im switched] =
       xcorr_match(im1, im2, center_shift)
2
3  THREE_COLOR = 0;
4
5  [im1h im1w im1d] = size(im1);
6  [im2h im2w im2d] = size(im2);
7
8  if ~exist('center_shift', 'var')
9      center_shift = [0 0];
10 end
11
12 imw = max([im1w im2w]);
```

```
13 imh = max([im1h im2h]);
14
15
16 if im1w*im1h < im2w*im2h
17     template_c = im1;
18     im_c = im2;
19     switched = -1;
20     center_shift = center_shift .* -1;
21 else
22     im_c = im1;
23     template_c = im2;
24     switched = 1;
25 end
26
27 if size(template_c, 3) ~= 1
28     template = rgb2gray(template_c);
29 else
30     template = template_c;
31 end
32
33 if size(im_c, 3) ~= 1
34     im = rgb2gray(im_c);
35 else
36     im = im_c;
37 end
38
39 imr = zeros([imh imw]);
```

```
40 imr(1:size(im, 1), 1:size(im, 2)) = im;
41 im = imr;
42
43 imr = zeros([imh imw 3]);
44 imr(1:size(im_c, 1), 1:size(im_c, 2), :) = im_c;
45 im_c = imr;
46
47 y_shift = center_shift(2);
48 x_shift = center_shift(1);
49
50 if THREE_COLOR == 1
51
52     t_hsv = rgb2hsv(template_c);
53     im_hsv = rgb2hsv(im_c);
54
55     c1 = normxcorr2(t_hsv(:, :, 1), im_hsv(:, :, 1));
56     c2 = normxcorr2(t_hsv(:, :, 2), im_hsv(:, :, 2));
57     c3 = normxcorr2(t_hsv(:, :, 3), im_hsv(:, :, 3));
58
59     %c = c1 .* c2 .* c3;
60     c = c1 + c2 + c3;
61 else
62     c = normxcorr2(template, im);
63 end
64 %%%smooth c to remove erronious peaks... ?
65 filtered_c = medfilt2(c);
66
```

```
67 g_r = double(size(c, 1) + abs(y_shift));
68 g_c = double(size(c, 2) + abs(x_shift));
69
70 s_r1 = 1;
71 s_r2 = g_r;
72 s_c1 = 1;
73 s_c2 = g_c;
74
75 if (x_shift > 0)
76     s_c2 = s_c2 - x_shift;
77 else
78     if (x_shift < 0)
79         s_c1 = s_c1 - x_shift;
80     end
81 end
82
83 if (y_shift > 0)
84     s_r2 = s_r2 - y_shift;
85 else
86     if (y_shift < 0)
87         s_r1 = s_r1 - y_shift;
88     end
89 end
90
91 gaussian_c = fspecial('gaussian', [g_r g_c], max(g_r, g_c) /  3); %%
    Was 6
92 gaussian_c2 = gaussian_c.*(1/max(gaussian_c(:)));
```

```
93  gaussian_c3 = gaussian_c2(s_r1:s_r2, s_c1:s_c2);
94
95  r1 = uint8(size(template, 1)/2);
96  r2 = uint8(size(c, 1)-size(template, 1)/2);
97  c1 = uint8(size(template, 2)/2);
98  c2 = uint8(size(c, 2)-size(template, 2)/2);
99
100 gaussian_c3(r1:r2, c1:c2) = gaussian_c3(r1:r2, c1:c2).*2;
101 %%%gaussian_c3(r1:r2, c1:c2) = 1; %-- Assumes equal probability of
        matching if complete overlap
102
103 filtered_c = filtered_c .* gaussian_c3;
104
105 [max_c, imax] = max(abs(filtered_c(:)));
106 [ypeak, xpeak] = ind2sub(size(c), imax(1));
107 corr_offset = [(xpeak-size(template, 2))
108                (ypeak-size(template, 1))];
109
110
111
112
113 x_shift = corr_offset(1)+1;
114 y_shift = corr_offset(2)+1;
115 x_end = min(size(template, 2),size(im_c, 2));
116 y_end = min(size(template, 1),size(im_c, 1));
117
118 new_im = im_c;
```

```
119 diff_im = zeros(size(new_im));
120
121 match = max_c;
122
123 if x_shift < 1
124     im_c1 = 1;
125     t_c1  = abs( 1 - x_shift);
126
127     im_c2 = x_end + x_shift;
128     t_c2  = x_end;
129 else
130     im_c1 = x_shift;
131     t_c1  = 1;
132
133     im_c2 = min(x_shift+x_end-1, size(im, 2));
134     t_c2  = im_c2 - im_c1+1;
135 end
136
137 if y_shift < 1
138     im_r1 = 1;
139     t_r1  = abs( 1 - y_shift);
140
141         im_r2 = y_end + y_shift;
142     t_r2  = y_end;
143 else
144     im_r1 = y_shift;
145     t_r1  = 1;
```

```
146
147             im_r2 = min(y_shift+y_end-1, size(im, 1));
148         t_r2  = im_r2 - im_r1+1;
149 end
150
151
152 match = abs(new_im(im_r1:im_r2, im_c1:im_c2, :) -
        template_c(t_r1:t_r2, t_c1:t_c2, :));
153 match = sum(match(:));
154 match = match / ( size(new_im(im_r1:im_r2, im_c1:im_c2, :), 1) *
        size(new_im(im_r1:im_r2, im_c1:im_c2, :), 2) *
        size(new_im(im_r1:im_r2, im_c1:im_c2, :), 3));
155 match = 1 - match;
156
157 new_im(im_r1:im_r2, im_c1:im_c2, :) = new_im(im_r1:im_r2, im_c1:im_c2,
        :) + template_c(t_r1:t_r2, t_c1:t_c2, :);
158 new_im = new_im ./2;
159
160 %%%Calculate the absolute differece bettween the template and the newly
161 %%%aligned background
162 diff_im_mask = diff_im; %%%set to all zeros the same size as the
        diff_image
163 %%set the mask to be the template valuse (so we can mask out where
        template
164 %%is 0
165 diff_im_mask(im_r1:im_r2, im_c1:im_c2, :) = template_c(t_r1:t_r2,
        t_c1:t_c2, :);
```

```
166
167 %%%Calculate difference, a variety of methods%%%
168 if 0
169     %%%---Absolute difference, then take max---%%%
170     diff = abs(im_c(im_r1:im_r2, im_c1:im_c2, :) -
            template_c(t_r1:t_r2, t_c1:t_c2, :));
171     %%%Pick the largest of the error vectors in RGB space ?
172     diff = repmat(max(diff, [], 3), [1 1 3]);
173 end
174 if 1
175         %%%---Euclidean difference in RGB space---%%%
176     diff = (sum((im_c(im_r1:im_r2, im_c1:im_c2, :) -
            template_c(t_r1:t_r2, t_c1:t_c2, :)).^2, 3)).^.5;
177     %%%Pick the largest of the error vectors in RGB space ?
178     diff = repmat(diff, [1 1 3]);
179 end
180 if 0
181         %%%---Euclidean difference in LAB space---%%%
182     rgb2lab = makecform('srgb2lab');
183     diff = (sum((applycform(im_c(im_r1:im_r2, im_c1:im_c2, :),
            rgb2lab) - applycform(template_c(t_r1:t_r2, t_c1:t_c2, :),
            rgb2lab)).^2, 3)).^.5;
184     %%%Pick the largest of the error vectors in RGB space ?
185     diff = repmat(diff./max(diff(:)), [1 1 3]);
186 end
187
188
```

```
189 %%%Normalize to similarity so that largest difference is 0, most
        similar is 1
190 diff_im(im_r1:im_r2, im_c1:im_c2, :) = 1 -  diff;
191 diff_im(diff_im_mask == 0) = 0; %%%mask out areas where the template
        was 0
192
193 end
```