DETECTING AND SUMMARIZING SALIENT EVENTS IN COASTAL VIDEOS

Daniel J. Cullen

14 May 2012

Boston University
Department of Electrical and Computer Engineering

Technical Report No. ECE-2012-06

BOSTON UNIVERSITY

DETECTING AND SUMMARIZING SALIENT EVENTS IN COASTAL VIDEOS

Daniel J. Cullen

Boston University
Department of Electrical and Computer Engineering
8 Saint Mary's Street
Boston, MA 02215
www.bu.edu/ece

14 May 2012

Technical Report No. ECE-2012-06

Acknowledgements

Advisors
Professor Janusz Konrad
Professor Thomas D.C. Little

Sponsor
The Consortium for Ocean Sensing
of the Nearshore Environment (COSINE)
(funded by the MIT SeaGrant Program)

Summary

Coastal environment sensing is an application of video surveillance that is of great interest to many biologists, ecologists, environmentalists, and law enforcement officials. We present a practical approach to detection and summarization of three salient events in coastal videos, namely the appearances of boats, motor vehicles, and people near the shoreline. Our approach consists of three fundamental steps: object detection through background subtraction and connected-components analysis, object classification using covariance of features in the detected regions, and summarization by means of video condensation. The goal is to distill hours of video down to a few short segments containing only salient events, allowing human operators to expeditiously study a coastal scene. We demonstrate the effectiveness of our approach on long videos taken of the beach on Great Point, Nantucket, Massachusetts.

Contents

1 Introduction **1**

2 Literature Review **4**

2.1 Background Subtraction 5
2.2 Behavior Subtraction 7
2.3 Covariance Matrix-Based Object Detection and Classification 8
2.4 Automatic Threshold Selection 10
2.5 Image Gradients using Cubic Convolution Interpolation 11
2.6 Seam Carving 12
2.7 Video Condensation 15

3 Methods **20**

3.1 System Architecture Overview 21
3.2 Implementation Details 21
3.3 Background Subtraction 22
3.4 Behavior Subtraction 24
3.5 Masking 27
3.6 Obtaining the Regions of Interest 27
3.7 Covariance Matrix-Based Object Classification 28
3.8 Improvements 38

4 Experimental Results and Discussion **42**

5 Conclusions and Future Work **46**

References **48**

Appendices **51**

A MS Project Symposium Poster **51**

B Source Code **53**

C Covariance Matrix-Based Detection Reports **53**

D AVSS2012 Draft Paper **74**

E Draft of System Specification **81**
E.1 Introduction . . . 81
E.2 System Block Diagram . . . 81
E.3 Usage . . . 81
E.4 Configuration File . . . 83
E.5 Output . . . 88
E.6 Caveats . . . 88
E.7 Other Observations . . . 89
E.8 Miscellaneous . . . 90

F Preliminary C++ Video Condensation Benchmarks **91**

List of Figures

1 Case study: Great Point, Nantucket, Massachusetts. The Great Point Lighthouse (a) has a video camera (b) mounted at the top (from a prior BU project), which captures scenes of the beach (c). 2
2 Illustration of each step in the proposed approach. 4
3 Original and background-subtracted frames. Top: Pedestrians in Marsh Plaza on BU campus. Bottom: Traffic on Interstate-90 near BU campus. 6
4 Similarity (i.e., *1/distance*) to cars dictionary for fixed-size rectangles across image. 9
5 Examples of seam carving borrowed from Avidan and Shamir 2007 [2]. 13
6 Video Condensation Example. Two images from two totally different moments in time merged into same frame. 16
7 Illustration of vertical ribbons (a) and horizontal ribbons (b), borrowed from [15]. 17
8 Example showing silhouette tunnels (in blue) and ribbons (in red). Image borrowed from [15]. 18
9 Video condensation demonstration. Frames are from a video of Marsh Plaza on BU Campus. 19
10 Example of video condensation. The three boats are from three different instants in time, but video condensation allows us to summarize this by merging them into the same frames. 20
11 System Block Diagram 21
12 Background-subtracted frame before (left) and after (right) enabling the Markov random field (MRF) model. Notice that the MRF model helps to “fill in” the interiors of the silhouettes of the cars, the truck, and the pedestrians. 23
13 Ripples on a pond: original image, background subtraction, and behavior subtraction 26
14 Beach: original image, background subtraction, and behavior subtraction 26
15 Three samples from each of the dictionaries: (a) boats, (b) cars, and (c) people used in feature-covariance detection. Images have been obtained from a search on Google Images. 30
16 Multithreaded, Pipelined Implementation 37

17 Aspect ratio technique. (a) shows the original behavior-subtracted frame, whose connected components give a large bounding rectangle around the wake, seen in (b). The result of applying the aspect ratio threshold technique is shown in (c). 40
18 Samples of typical input video frames (top row) and outputs from the processing blocks in Figure 11: (row 2) background subtraction, (row 3) behavior subtraction, (row 4) object detection, (row 5) video condensation. 44
19 SEAL System Block Diagram 82
20 Object Detection Block Diagram 82

List of Tables

1 Number of frames after each flex-step and cumulative condensation ratios (CR) for 38-minute, 5 fps video with boats and people (11,379 frames after behavior subtraction). 45
2 Number of frames after each flex-step and cumulative condensation ratios (CR) for 22-minute, 5 fps video with boats, cars and people (6,500 frames after behavior subtraction). 45
3 Average execution time for each stage of processing. 46

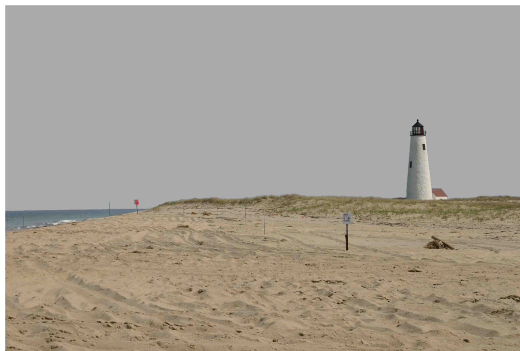
1 Introduction

In recent years, technological improvements have made digital cameras ubiquitous. They have become physically smaller, less expensive, more efficient in power consumption, wirelessly-networked, more readily available, and altogether easier for the average consumer to use. Even low-end, off-the-shelf personal digital cameras can capture hours of high-definition video nowadays, thanks to fast embedded processors and high-density storage media. For these reasons, cameras are finding increasing use in many new surveillance applications outside of the more-traditional public safety applications. One example of this is environmental monitoring. Scientists and researchers can deploy networks of cameras, such as the one described in [16], to gather video data to study wildlife and the environment.

This project focuses on one particular coastal environment sensing case study. We have collected hundreds of hours of video data from a networked camera located at the beach on Great Point, Nantucket, Massachusetts (see Figure 1). Many organizations are interested in using this data to answer questions about the wildlife, erosion, and how humans impact the environment. For example, biologists would like to learn more about how many seals (marine mammals) are on the beach at any given time and whether or not humans have gotten too close to them. Environmental protection agencies need to know how many people and automobiles have been on the beach beach and whether or not they have disturbed the fragile sand dunes. Law enforcement officials also have concerns about automobiles on the beach. Other organizations would like to keep track of the boats passing through the harbor. These are just a few of the many uses of coastal video data.

A fundamental difficulty in many surveillance applications is that there is simply too much data for human operators to watch. A single camera recording continuously all day long, seven days a week produces over one hundred hours of video data per week, and if multiple cameras are used, this number can grow to thousands of hours. Even if the videos were played back at high speed, the amount of data to visually inspect is enormous, and there is still a chance of missing an important event (due to frame skipping, operator fatigue, etc.). Many organizations simply do not have the resources to hire humans to study the data.

Therefore, the goal of this project is to develop automatic algorithms to identify and summarize the salient events in the videos. In particular, this project focuses on events involving three specific classes of objects, namely cars, boats, and people. This



(a) Great Point Lighthouse

(b) Camera on Great Point Lighthouse

(c) View of beach from the camera in (b)

Figure 1: Case study: Great Point, Nantucket, Massachusetts. The Great Point Lighthouse (a) has a video camera (b) mounted at the top (from a prior BU project), which captures scenes of the beach (c).

choice of objects of interest is dictated by our specific application but the approach we propose is general and can be applied in other scenarios as well.

Our proposed approach has three key steps: object detection, object classification, and video summarization. First, since motion in videos is generally considered interesting, we apply a technique known as *background subtraction* [18] to identify areas of motion. Unfortunately, the presence of ocean waves, dune grass blowing in the wind, and other spurious background activity results causes problems in the subsequent processing steps, namely increased false positives in object classification and poor degrees of video summarization. Therefore, we apply a technique known as *behavior subtraction* [12] to remove some of this uninteresting, stochastically-stationary background activity. We next apply connected-components analysis to obtain bounding rectangles around the blobs of the detected objects. Next, in the object classification step, we apply the *region covariance* approach of [21] to determine whether a rectangular region likely contains a boat, a car, a person, or none of these. Finally, we summarize the video using a technique called *video condensation* [15], adapting it to our needs by using the outputs of behavior subtraction and object classification as the cost function for condensation. The end result of our approach is that we have a system that automatically distills hours of video down to a few short segments containing only the salient events.

A block diagram of the proposed system is given in Figure 11. A sequence of sample images illustrating each of the steps is shown in Figure 2. For a quick overview of the project, we refer the reader to the project poster given in Appendix A, as well as the *Reader's Digest* version of the project in our AVSS conference paper draft reproduced in Appendix D.

Another reason why object detection and classification is desirable is that once the objects are identified, we can run algorithms to track the objects and generate statistics about the quantities and behaviors of objects. The first step in a more-sophisticated analysis of the scene is detecting and identifying the objects involved. However, this is outside the scope of this project. For now, a human operator is still required to analyze the condensed video due to the complexity of the scene and the fact that that the researcher may not have a clear idea of what constitutes an anomalous event. Since the human user is unavoidable, video condensation is indispensable because it greatly expedites his or her job.

In our approach, we place a strong emphasis on designing the system for fast execution speed. For example, we modified the background subtraction described in

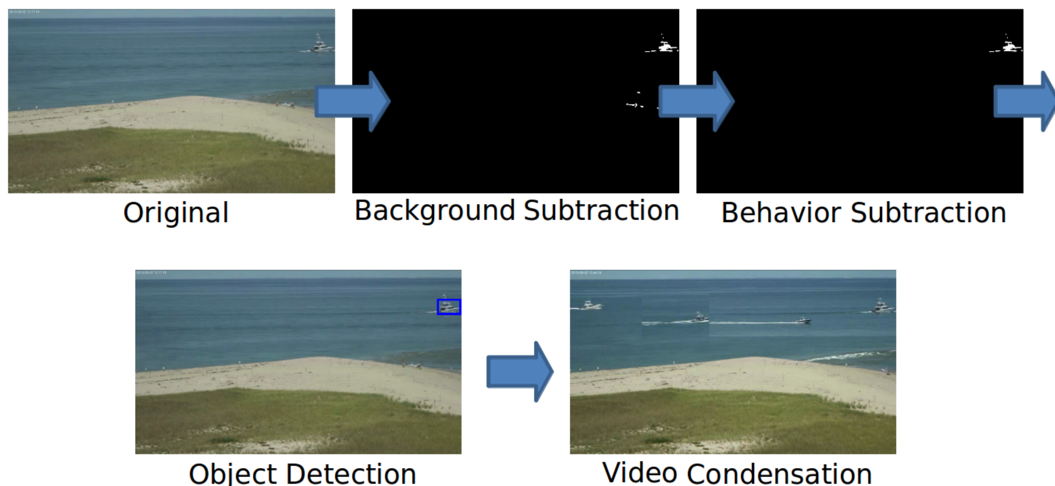


Figure 2: Illustration of each step in the proposed approach.

[18] to use a simple, exponentially-smoothed moving average for speed, plus a Markov Random Field (MRF) model to improve accuracy. Speed is critical when there are many hours of video to process, or if slower embedded systems will be running the algorithms, or if the observers would like to quickly respond (in real time) to the events in the video.

Of course, there are several assumptions that we make about our input videos. We assume that the camera is fixed (stationary) and does not move or zoom. We assume that there is no camera shaking, no water on the lens, constant lighting (i.e., no change in cloud cover and no sudden changes in camera camera gain), and so on.

This paper is organized as follows. First, the *Literature Review* section provides background about each of the techniques we apply in this project. Next, we describe our system in further detail in the *Methods* section, explaining how we implemented and tested each component. After that, we discuss some of the experimental results from running the complete system. Finally, we offer a few concluding remarks and suggestions for future improvement.

2 Literature Review

In this section, we discuss the theoretical background of each of the algorithms that we employ in our system. We hope that this section serves as a useful starting point for the reader and helps to explain the key points and design decisions of our approach. We try to keep this section concise and practical, summarizing and relevant

referencing prior work while avoiding esoteric detail; for more explanation on each topic, refer to the cited papers.

2.1 Background Subtraction

Background subtraction is a much-studied and well-understood problem in the literature. The idea is to separate the salient *foreground* objects in a scene from the uninteresting *background*. For example, consider a video of an urban street corner captured using a fixed camera. Every so often, pedestrians and cars (the foreground objects) move past the camera, but the buildings and environment (the background objects) remain relatively unchanged. Many background subtraction algorithms measure the change in pixel intensity values in order to classify each pixel as belonging to either the foreground or background. Thus, these techniques capture motion in an image.

Figure 3 provides some examples of background subtraction. The original frames are shown on the left and the background-subtracted frames are given on the right. As can be seen, the background-subtracted frames are binary images in which the foreground pixels are marked with logic ones (white pixels) and the background pixels are marked with logic zeros (black pixels).

One approach to background subtraction is presented by Mike McHugh et al. in [17] and [18]. This approach uses kernel density estimation (KDE) to estimate the probability that a pixel belongs to the background. [1] In addition, [18] uses a Markov random field (MRF) model to greatly improve accuracy. It also presents a foreground model that can be used for a marginal increase in performance.

Unfortunately, although the KDE approach yields excellent results, it is also very computationally intensive and is therefore quite slow. Since our system will be used to process many hours of video, it is important that we perform the background subtraction as efficiently as possible. A simpler, faster alternative is to use an exponentially-smoothed moving average (i.e., a recursive filter) to estimate the mean and variance of each of the pixels in the image. This approach also requires less memory than KDE, as it only requires storing an average image and a variance image, rather than a long buffer of images. Yifan Yu explored this moving average approach in his Master's project [25] and found that it works reasonably well, although it is not quite as accurate as KDE. One reason for this is that KDE intrinsically models multi-modal

[1] More information about using KDE for modeling the background can be found in Elgammal et al. [4].

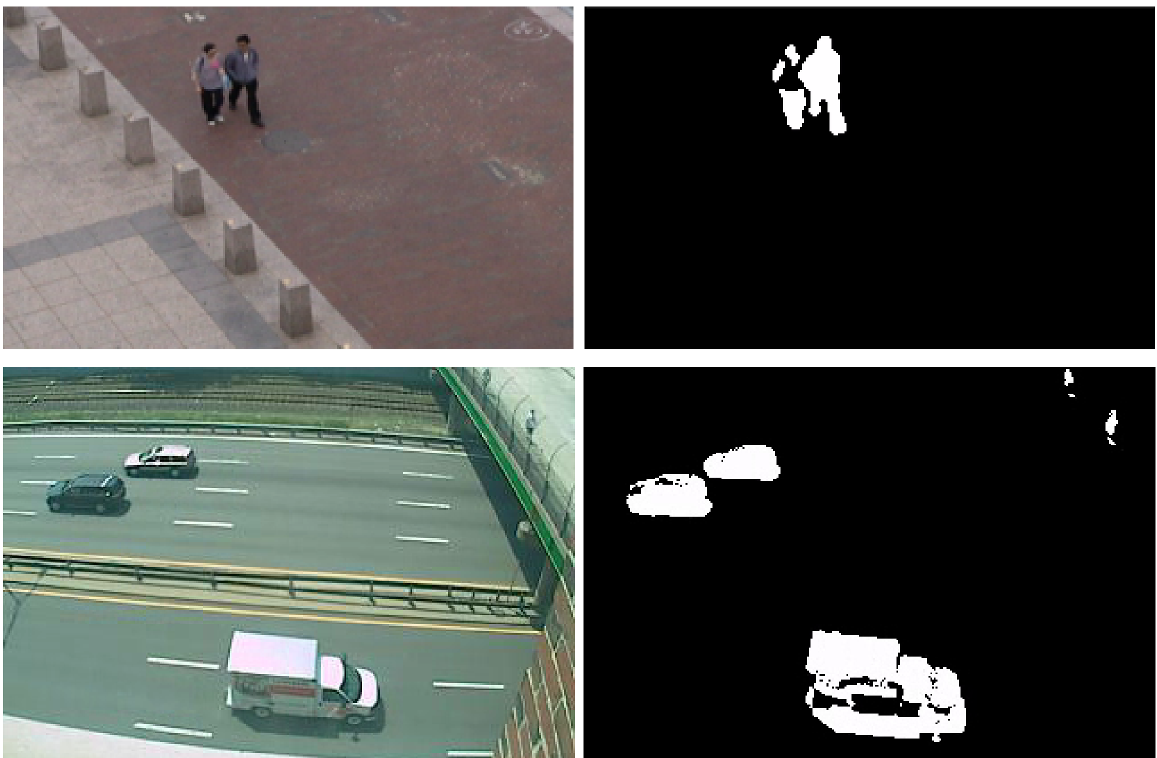


Figure 3: Original and background-subtracted frames. Top: Pedestrians in Marsh Plaza on BU campus. Bottom: Traffic on Interstate-90 near BU campus.

probability distributions, whereas the moving average approach assumes uni-modal distributions.

In this project, we implemented both the KDE and recursive moving average approaches. Specific details of our implementation are discussed in the *Methods* section. Results of our code are discussed in the *Experimental Results and Discussion* section.

2.2 Behavior Subtraction

In the coastline videos that we wish to process, there is a great deal of uninteresting, repetitive background motion such as ocean waves and dune grass. This results in many spurious detections when we perform background subtraction. One simple thing that we can do is select a region of interest, ignoring or "masking-out" all activity in regions of the image that we don't care about. While this primitive method works well in situations such as monitoring people on the sandy part of the beach but ignoring all activity in the dune grass and in the ocean, it does not work when we need to analyze objects in the the same region as the false detections, such as boats on the water. Furthermore, this approach requires a human to manually select the region of interest, but ideally we would like to have a method that requires minimal human intervention.

We implemented an algorithm called behavior subtraction to reduce the number of false detections. Behavior subtraction [12] is an algorithm that removes stationary motion from a video. We run background subtraction to create the binary cost video and then run behavior subtraction to remove the false detections. The behavior subtraction algorithm has two phases: training and processing. In the training phase, we examine a window of N frames from the M frames of training data (where M $\geq$ N). The training data should exhibit the stationary behavior that we want to remove, but it should not have any "interesting" moving objects. First, we sum up the cost at each pixel in the window. Then we shift our window by one frame and perform another summation. We record the maximum of each of the summations at at each pixel at each window location. This completes the training phase. The next phase is the processing phase. We perform the same kind of summations over the sliding window as we did during the training phase, except that we compare the sum against the maximum, rather than updating the maximum. If the difference between the sum and the maximum is greater than a certain threshold, we detect the pixel as interesting motion and write a white pixel to the output video file; otherwise, we write a black

pixel. The rationale behind this is that an "interesting" object will occupy those pixels for more frames than the maximum frames occupied by stationary behavior, thereby allowing us to discriminate between interesting and uninteresting motion.

2.3 Covariance Matrix-Based Object Detection and Classification

We base our object detection and classification approach on the works [21] and [20]. We have selected this approach because of of its success in identifying objects and also because its robustness to non-idealities such as partial occlusion and illumination changes. Others, such as [10] and [9], have also had a great deal of success in applying similar covariance matrix-based techniques to other applications, such as action recognition and classification.

The approach described in [21] entails computing a d-dimensional vector of simple features for every pixel in a region of n pixels in an image, and then generating the $d \times d$ covariance matrix from all n of the feature vectors. For simplicity, rectangular regions of pixels are used. The similarity of two regions can be computed using a distance metric. Since covariance matrices lie on a Riemannian manifold in non-Euclidean space, we cannot use a simple Euclidean distance measure. [21] recommends using the distance metric proposed in [5], in which the square root of the sum of the squared logarithms of the generalized eigenvalues of the covariance matrices gives the distance. [9] uses another distance metric in which the matrix logarithms of the covariance matrices are computed and the Frobenius norm gives the distance between them.

To detect the objects in the image, [21] describes a brute-force search using fixed-size search rectangles of different scales. The distance between the target covariance matrix and the covariance matrix of each query location is computed and compared to a threshold; if the distance is less than the threshold, the object is detected. To increase the speed of the search, they use a technique called "integral images", which allows them to quickly obtain the covariance matrix for any arbitrary rectangular region by performing simple arithmetic on precomputed integral images.

An example of such a brute-force search is shown in Figure 4. On the left is a 3D surface that shows the similarity (i.e., *1/distance*) between the cars dictionary and fixed-size query rectangles scanned all over the image. The regions around the two jeeps are most similar (i.e., their distance is minimized) to the cars dictionary. Clearly, thresholding this 3D surface allows us to detect these vehicles as cars. For

more examples of 3D plots such as this one, please refer to C.

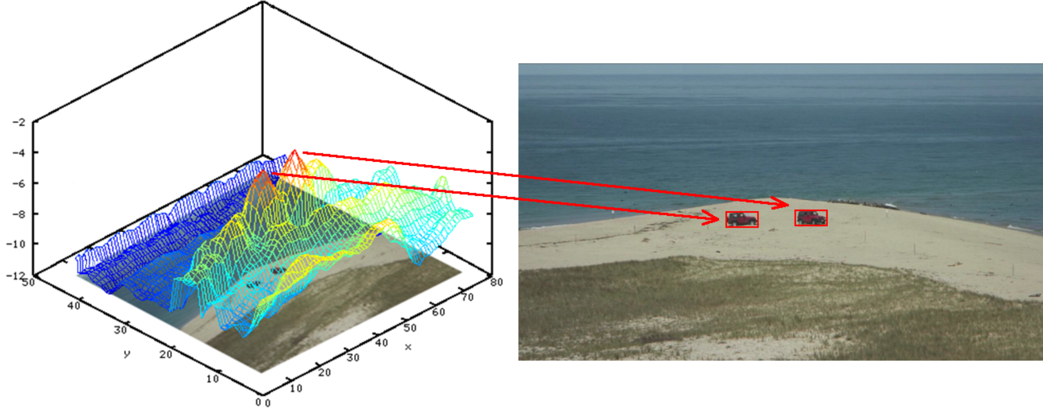


Figure 4: Similarity (i.e., *1/distance*) to cars dictionary for fixed-size rectangles across image.

The main drawback to the brute-force search is that even with integral images, it is computationally expensive, especially when many frames of video must be searched for many different classes of objects. Since we want our own system to run in nearly real-time, we do not use a brute-force search. Instead, we use the the bounding boxes from connected components of the behavior-subtracted video as the set of regions that we need to test. This greatly reduces the number of locations that must be checked, greatly increasing the search speed and eliminating the need to use integral images. Another advantage of our own approach is that it is more automatic because the size of the search windows comes from the connected components; in contrast, [21] uses several fixed-size search windows, so there is an underlying assumption about the sizes of the target objects.

We used a feature vector recommended by [21] that contains six features: the x and y pixel coordinates relative to the top-left corner of the region of interest and the first-order and second-order gradients in each direction, calculated using simple derivatives of cubic convolution interpolation [14] of the pixel intensity I. Equation 1 shows this vector:

$$\zeta(x,y) = \left[\; x, \; y, \; \left|\frac{\partial I(x,y)}{\partial x}\right|, \; \left|\frac{\partial I(x,y)}{\partial y}\right|, \; \left|\frac{\partial I^2(x,y)}{\partial x^2}\right|, \; \left|\frac{\partial I^2(x,y)}{\partial y^2}\right| \; \right] \qquad (1)$$

Note that for some applications, [21] also proposes augmenting this feature vector to also contain RGB color components. However, in our system, we do not use the color-augmented feature vector because we want our system to consider only the

shapes of the objects. Many of the types of objects that we wish to test, such as cars can come in many different colors. Rather than create more databases, one for each color of car, we simply ignore color and consider only shape. Our preliminary preliminary experiments have shown that an augmented feature vector can improve detection accuracy for certain classes of objects, but

We use a nearest-neighbor approach similar to the one described in [9] to detect the objects. We first compute the similarity between a given test region in the image and each of the images in our dictionary using the distance metric and then we find the minimum of these distances. If the minimum distance between the test region and the dictionary is less than a given threshold, the object is detected. For each class of object that we want to compute, we have a different dictionary. We repeat this procedure for each class of object that we wish to detect, comparing the region against all of the images in the dictionary corresponding to that particular class of object.

Since covariance matrices do not lie in a Euclidean space, the Euclidean distance between covariance matrices is a poor measure of similarity. Therefore, for our distance metric, we instead use the the Frobenius norm of matrix logarithms, as proposed by [1]. This method is efficient because we can precompute and store the log covariance matrix for each image in each dictionary. To check if a given target region contains an object, we simply compute the Frobenius norm between the target log covariance matrix and the item from the dictionary. This is more efficient than the [5] method because at each step, all we need to do is take the sum of squared differences between elements of two matrices, rather than having to apply eigenvalue decomposition at every step.

The matrix logarithm of a covariance matrix C is computed as follows. Suppose that the eigen-decomposition of C is given by $C = VDV'$, where the columns of V are orthonormal eigenvectors and D is the diagonal matrix of (non-negative) eigenvalues. Then $\log(C) := V\widetilde{D}V'$, where $\tilde{D}$ is a diagonal matrix obtained from D by replacing D's diagonal entries with their logarithms.

2.4 Automatic Threshold Selection

Ideally, one would like the detection threshold for the covariance matrix-based technique to be automatically established based on a desired confidence level (e.g., 95% likelihood of detection), so that the system does not require any human intervention. One

technique to accomplish this called *leave-one-out cross-validation (LOOCV)*. This technique involves generating a distribution of pairwise distances between elements of the training set, or in our case, between elements of the dictionaries of reference images. It has been applied successfully for other applications using similar covariance matrix-based metrics in [6, p.75-76] and [8].

2.5 Image Gradients using Cubic Convolution Interpolation

We want to use object shape as one of the features when building our covariance matrices. One way to characterize shape is to find the edges in the image. There are many different approaches to edge detection, but one of the simplest is to compute derivatives of the pixel intensity in the image. This can be accomplished by taking simple pairwise differences between pixels along the horizontal and vertical directions (treating each direction independently). (The pairwise differences can be obtained by convolving each row and column of the image with the kernel [-1, 0, 1]; second derivatives can be obtained using the kernel [-1, 2, 1].) Typically, some type of lowpass filtering is applied before taking the pixel differences in order to minimize the effects of high frequency noise in the gradient.

However, in this project, we decided to take a slightly more sophisticated approach. Rather than filtering and convolving with a difference kernel, we use the cubic convolution interpolation approach suggested by Robert G. Keys in [14] to model the image with cubic functions passing through each set of four consecutive points in the rows and columns. The coefficients of the cubic function are obtained using these sample points. Symbolic differentiation of the equation for this cubic function yields symbolic equations for the first and second derivatives, which we can easily use to find the gradients at any point in the image.

The equation for cubic convolution interpolation from [14] is as follows:

$$g(s) = \frac{f(a)(-s^3 + 2s^2 - s) + f(b)(3s^3 - 5s^2 + 2) + f(c)(-3s^3 + 4s^2 + s) + f(d)(s^3 - s^2)}{2}$$

where a, b, c, and d are four consecutive points, $f(x)$ is the value of the function (i.e., pixel intensity) at point x, and s is a value on the interval $[0.0, 1.0]$ indicating the location at which we want to obtain the interpolated value $g(s)$. Note that we are interpolating between points b and c, so s is the fraction of the linear distance between b and c. Furthermore, if we wanted to interpolate between points a and b or between points c and d instead, we would simply shift to the left or to the right by

one point so that the desired interpolated point would be centered between b and c again. If you want to interpolate a point near either the left or right boundary of the data, but the point a or d is located outside the range of data, you simply use one of the following equations to estimate the values at a or d using three leftmost or three rightmost points from the data set:

$$f(a) = 3f(b) - 3f(c) + f(d)$$

$$f(d) = 3f(c) - 3f(b) + f(a)$$

Finally, note that this algorithm is designed for interpolation, not extrapolation of data past the boundaries of the data set.

Differentiating, we obtain the following expression for the first derivative:

$$\frac{d}{ds}g(s) = \frac{f(a)(-3s^2+4s-1) + f(b)(9s^2-10s) + f(c)(-9s^2+8s+1) + f(d)(3s^2-2s)}{2}$$

Differentiating again, we obtain the following expression for the second derivative:

$$\frac{d^2}{ds^2}g(s) = \frac{f(a)(-6s+4) + f(b)(18s+4) + f(c)(18s+8) + f(d)(6s-2)}{2}$$

It is worth noting that other edge-detection kernels are also popular, such as the 3x3 Sobel operator, which performs computes pixel differences while performing some averaging in the orthogonal direction. Other edge detection algorithms are also possible, such as the popular Canny edge detector. Although these methods were not explored in this project, they are potential areas for future study.

2.6 Seam Carving

The goal of this project is to distill many hours of video down to a few minutes showing just the most interesting events in order to help humans to analyze the data. One algorithm for doing this is called video condensation. However, before we explain video condensation, we first need to discuss an algorithm called seam carving because it will facilitate discussion of video condensation in the next section.

The main goal with seam carving is to resize images while preserving image content. When the dimensions of images are scaled down, they are typically downsampled and interpolated. Unfortunately, downsampling results in a loss of information in the salient regions of the image. If we could selectively remove uninteresting areas

within the image, we could decrease the dimensions of the image while preserving all of the visual information in the important regions. This principle referred to as *content-aware image resizing* in [2].

Cropping an image is another technique to reduce the size of an image. When an image is cropped, pixels are discarded, resulting in a smaller image. However, the main drawback to cropping areas from the interior of the image is that visible discontinuities will occur along the edges of the removed regions when the remaining pixels are fitted together. These discontinuities can be greatly minimized if the pixels to be removed are carefully chosen; this is one of the motivations for the seam carving algorithm presented in [2].

Figure 5: Examples of seam carving borrowed from Avidan and Shamir 2007 [2].

Please refer to the image of the lake in the Figure 5. For the moment, ignore the two red lines (called *seams*) that run from top to bottom and right to left. One might argue that the most interesting features of this image are the building and perhaps the shoreline, whereas of the lake itself is not very interesting. (One way to quantify this is that there are many edges in the building and shoreline, but relatively few edges in the interior of the lake.) Now suppose that we want to reduce the size of the image in the vertical direction by removing pixels. Since much of the lake is not very interesting, suppose we remove pixels from the lake area. Note that we must remove the same number of pixels from each column, so that all columns of the resulting image have the same number of pixels. Unfortunately, if we remove straight horizontal rows, we might cut through some of the interesting reflections in the water. However, if we allow some flexibility in the path of pixels from left to right across the

image, we can avoid cutting through interesting objects while still removing one pixel from each column. An example of such a path is shown by the red line (*seam*) in the image that runs from left to right across the image; notice how it bends to avoid the interesting reflections in the water. The image that results from removing many left-to-right seams of pixels from the image of the lake is shown below it; notice that much of the uninteresting area of the lake has been removed but the features have been preserved.

Avidan and Shamir [2] define a *seam* to be a "connected path of low-energy pixels crossing the image from top to bottom or from left to right." The "energy" or "cost" function is what allows the algorithm to weight certain pixels with more importance than other pixels. For the image of the lake in Figure 5, a pixel intensity gradient cost function was used to automatically select left-to-right seams of pixels to remove from the image. A gradient cost function works well for many applications because it allows pixels that are part of edges to be marked with high cost, thus preserving interesting shapes in an image. The seam carving algorithm finds paths that avoid cutting through the gradients, thus minimizing the total accumulated cost of the pixels along those paths. This helps to minimize visual discontinuities in the image with the seams removed. The seam carving algorithm can be used for other applications if a different cost function is used. For example, consider the two images of people on the right-hand side of Figure 5. If high cost is assigned to the man and low cost is assigned to the woman (indicated by the red and green shading in the inset image), the woman can be removed from the image, as shown on the right. [2] refers to this procedure as "object removal." Video condensation via ribbon carving is another application based on the seam carving algorithm, in which we use a binary cost function to preserve certain features in the video frames, such as moving objects.

How does it work? The seam carving algorithm finds a seam through the image that minimizes the accumulated cost of the pixels along that seam. In order to efficiently find the minimum-cost path, this algorithm uses a dynamic programming approach. The following example explains this in further detail. The main idea is to look for a seam, out of all possible seams, with minimum energy. Suppose that we have a 3×3 image and we compute e, the energies at each pixel:

```
e = +---+---+---+
    | 1 | 2 | 3 |
    +---+---+---+
    | 2 | 1 | 2 |
    +---+---+---+
    | 1 | 2 | 3 |
    +---+---+---+
```

Next is the dynamic programming step. Here we compute M, which contains the cumulative minimum energy for the minimum-energy seam. We compute it by starting at the top and working our way down (i.e., start with the top row and work our way down). In other words, we're looking for vertical seams.

```
M = +---+---+---+
    | 1 | 2 | 3 |
    +---+---+---+
    | 3 | 2 | 4 |
    +---+---+---+
    | 3 | 4 | 5 |
    +---+---+---+
```

Then, if you traverse starting with the bottom (i.e., last) row and working your way back up to the first row, you can find the minimum energy path P:

```
P = +---+---+---+
    | X |   |   |
    +---+---+---+
    |   | X |   |
    +---+---+---+
    | X |   |   |
    +---+---+---+
```

The ribbon carving video condensation algorithm is an extension of seam carving from a 2D to 3D, since video is comprised of a sequence of images, rather than a single image. As mentioned above, ribbon carving uses a binary cost function to indicate which pixels must be preserved in the condensed video. Video condensation will be discussed further in the next section.

2.7 Video Condensation

In this project, we have collected hours upon hours of video data from cameras located at the beach on Great Point, Nantucket. Although many organizations are interested in this data, there is too much video for human researchers to watch. This problem

is common to many types of surveillance applications. The goal of our research is to use computers to analyze this data, automatically distilling hours of video down to a few short segments of only the salient events. One technique for doing this is known as *video condensation*.

The main idea behind video condensation is to remove frames of data without activity, preserving only the salient events. Furthermore, objects from different moments in time are merged into the same moment in time, as illustrated in Figure 6, as long as the regions occupied by two objects at the different times do not overlap. This greatly reduces the amount of data that a human observer must watch. In addition, if we tag the objects with number of the frame at which that object appeared in the original video, it is easy for the observer observer to jump back to the original video clip for closer study.

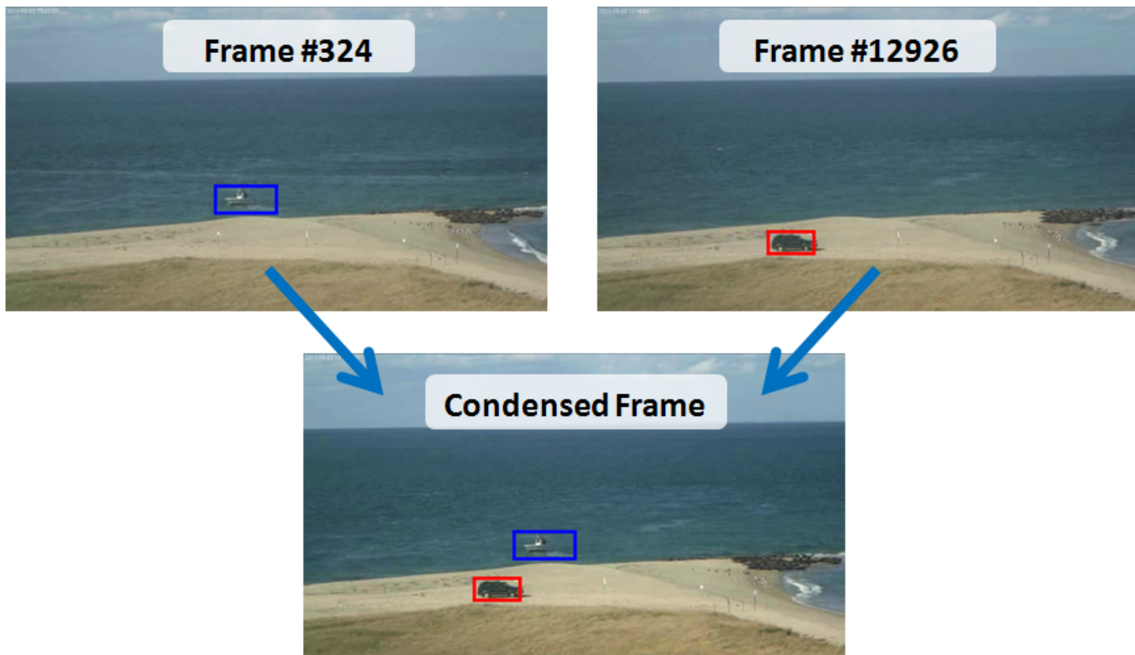


Figure 6: Video Condensation Example. Two images from two totally different moments in time merged into same frame.

The algorithm we use in this project to perform video condensation is called *ribbon carving*, introduced in [15]. This algorithm is based on the seam carving algorithm discussed in Section 2.6 (*Seam Carving*). Seam carving performs quite well on two-dimensional images, but extending it to a third dimension (time) for sequences of images (videos) becomes computationally intractable [15]. However, if either the x-dimension or the y-dimension is constrained, we get ribbon shapes instead of 3D

video seams, known as *vertical ribbons* and *horizontal ribbons*, respectively, as shown in Figure 7. This constraint reduces the task to solving a simple 2D seam carving problem, hence the name *ribbon carving*. Using ribbons instead of 3D seams makes the algorithm computationally tractable, easier to code, and still produces nice results for many practical situations.

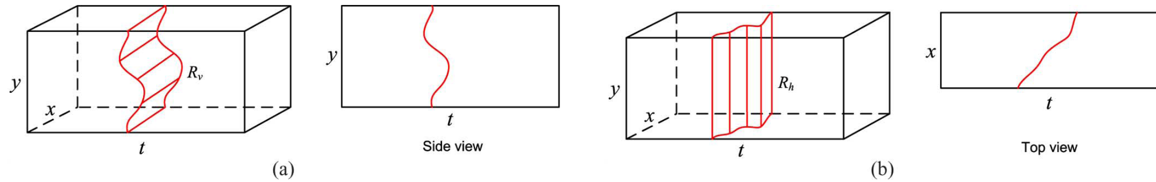


Figure 7: Illustration of vertical ribbons (a) and horizontal ribbons (b), borrowed from [15].

Just like seam carving, ribbon carving requires a cost function to determine which pixels should be preserved and which pixels can be removed. We typically use a binary background subtraction cost video because we are interested in analyzing moving objects in a video. By assigning a high cost to the pixels in moving areas, we can preserve those objects and allow ribbons to cut through other areas. Higher compression ratios can be obtained if we allow some of the ribbons to partially carve though some of the objects that we want to preserve, or in other words, if we allow some ribbons to have a nonzero accumulated cost. In our code for this project, we have created a parameter θ for this cost threshold. [15] refers to this threshold as the *stopping criterion* because the algorithm will carve ribbons until it reaches this stopping point. It is generally undesirable to allow ribbons to cut through objects because this can cause display artifacts, but in some situations, a nonzero threshold may help to improve robustness to noise or to imperfect background subtraction [15].

Visualizing the volumes occupied by objects moving in 3D video space helps in understanding how ribbon carving works. These regions are referred to as *object tunnels* in [15] or as *silhouette tunnels* in [7], and they show us which pixels from which frames correspond to which moving objects. The images in Figure 8 are basically 3D plots of the frames of background subtraction videos and show the silhouette tunnels for several objects. The figure also shows some examples of ribbons carving through the 3D space around the silhouette tunnels. The idea behind ribbon carving is that the closer we can move the silhouette tunnels together (without touching), the better condensation we will achieve. The way that we move the silhouette tunnels closer together is by carving ribbons. However, we must make sure that the ribbons do not cut through the tunnels and that tunnels do not touch each other. From this figure,

it is also evident that ribbons that are more flexible can better fit between tunnels, carving out more pixels and yielding better compression.

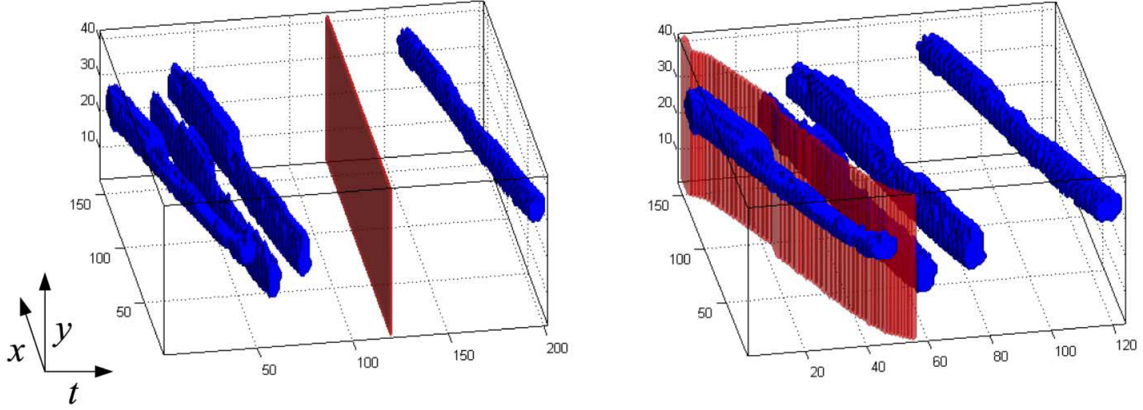


Figure 8: Example showing silhouette tunnels (in blue) and ribbons (in red). Image borrowed from [15].

The ribbon carving algorithm runs in several passes. At each pass, an increasing amount of flexibility is allowed for the ribbon along the time dimension. The amount of flexibility is known as the *flex parameter*, often denoted by ϕ. As explained in [15], it is important to perform the flex passes in the order of increasing flex parameter so that "event anachronism is kept to a minimum".

In the first pass of the algorithm, known as *flex 0*, the ribbon is not allowed to bend at all. Thus, *flex 0* is equivalent to removing entire frames. For example, if the stopping criterion threshold is zero, the *flex 0* pass will remove all of the frames with zero cost. In the next pass of the algorithm, known as *flex 1*, the maximum allowed slope of the ribbon is 1 pixel per frame. In other words, *flex 1* means that the maximum allowed deviation in the time dimension between rows (for vertical ribbons) or columns (for horizontal ribbons) is 1 pixel. Similarly, *flex 2* has a maximum deviation of 2 pixels and so on. Greater values of the flex parameter result in better compression ratios because the ribbons can more curve more to carve out more pixels while avoiding the silhouette tunnels. As explained in [15], the amount of condensation follows the law of diminishing returns; higher flex values result in marginal condensation. In practice, a good upper limit on the number of passes is *flex 3* because it is a good tradeoff between computation time and condensation performance, as was empirically determined in [15] and in [24].

It is important to understand that ribbon carving does more than just remove frames without any activity. Removing frames without activity is just the first pass

(*flex 0*). Each subsequent pass (*flex* $\geq$ *1*) merges events from different moments of time into the same frames, yielding better and better condensation. Figure 9 illustrates this with frames of a video of people walking down a city street. First, a girl enters the scene from the top, walks down the street, then exits at the bottom of the scene. Figure 9a gives one frame from this sequence. Some time passes without any people in the scene. A while later, two people enter the scene and walk down the street, as shown in Figure 9b. If we run *flex 0* through *flex 3* on the video, we will obtain a condensed video showing the girl entering the scene and walking down the street, immediately followed by the two people. Figure 9c shows one of the frames from the condensed video sequence. The *flex 0* pass removed all of the empty frames without any people in them, but it was the *flex 1* through *flex 3* passes that actually merged the girl from one frame and the two people from another frame into the same frame, as shown in Figure 9c. If you look closely, you can see a horizontal boundary across the middle of the image where the pixels from the two moments of time come together. (This line is noticeable because the ambient lighting or camera gain changed slightly between the time that the girl walked across these scene and the time that the two people walked across the scene.)

(a) Original frame (t=9 seconds) (b) Original frame (t=36 seconds) (c) Condensed frame (flex 3)

Figure 9: Video condensation demonstration. Frames are from a video of Marsh Plaza on BU Campus.

Another example that shows how video condensation merges events from different moments in time is shown in Figure 10. The three boats in the image

The ribbon carving algorithm is designed to process a sequence of N frames. However, for very long videos, the computational costs of processing all of the frames at once are prohibitive. In order to efficiently process long sequences, [15] outlines a sliding window approach. The minimum and suggested numbers of frames for the sliding block, as well as the algorithms for adding and removing frames to and from this block, are discussed in detail in [15] and [24]. Huan-Yu Wu [24] implemented this

Figure 10: Example of video condensation. The three boats are from three different instants in time, but video condensation allows us to summarize this by merging them into the same frames.

technique in MATLAB and developed a C code implementation of the ribbon carving steps using MATLAB's MEX interface in order to speed up the ribbon carving step.

The amount of condensation that can be achieve depends upon the amount of activity in the scene and the behavior of that activity; thus, the compression ratio depends on the particular application. For example, [15] and [24] have found that the condensation algorithm works better when objects tend to move across the screen in the same direction, such as cars traveling one way along a highway. Another example of this is boats traveling in the same direction across a harbor, one right after another, as shown in Figure 10. The reason situations like this work so well is that the silhouette tunnels can be tightly packed together when ribbons are carved out between them. Another example of a scenario that yields good condensation results is one in which activity occurs at opposite sides of the screen at different moments in time, but the objects involved stay on their respective sides of the screen; after condensation, the two activities can be played back at the same time, side by side. In this project, we found that for typical videos of the beach, one hour of video can be condensed to about three minutes. For more discussion of our results, please refer to the *Experimental Results and Discussion* section.

3 Methods

This section explains how the system works and how we developed and tested it. It also describes some of the challenges we encountered as we developed the system, as well as some of the improvements we implemented to overcome them.

3.1 System Architecture Overview

The block diagram of the entire system is given in Figure 11.

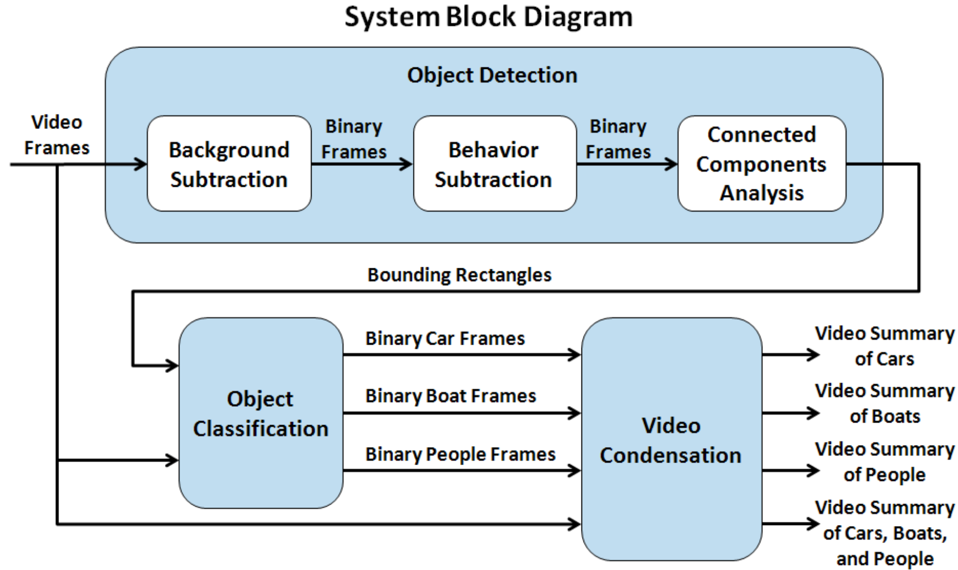


Figure 11: System Block Diagram

An early draft of my system specification can be found in Appendix E. It is a bit out of date, but still provides some useful information about the system parameters and how to tune them.

3.2 Implementation Details

All code was written in C++ for fast performance and portability so that it may be readily deployed on embedded sensing platforms in the near future.

Our code uses only free, open-source libraries. For the video I/O (and some image I/O) we used *FFMPEG*. We used *CImg* for image I/O. In order to solve for the eigenvalues in computing the matrix logarithm of the covariance matrix, we used the *Armadillo C++ Linear Algebra Library*. To save some time and effort implementing the connected components, we used the simple *OpenCV* library function *cvFindContours*. All of the other algorithms were implemented "from scratch," using only standard C++ code.

The code has been developed under Linux, but can easily be ported to Microsoft Windows (future work).

Since there is far too much source code to include in this report, it is provided as

a zipped archive to be distributed along side this report. For more details about the source code, please refer to the Doxygen documentation in the source.

In the near future, we hope to increase the simplicity and portability of the code by removing the dependency on OpenCV. The OpenCV libraries require many dependencies, which makes it more difficult to port to different platforms. Currently, the only pieces that we need OpenCV for are displaying the images (which we can easily implement with *CImg* or any GUI toolkit) and performing the connected-components analysis. We therefore plan to re-write the connected-components code from scratch or to borrow code for it from another, lightweight library. This should allow eliminate our dependence on OpenCV, making our code much easier to compile and distribute.

3.3 Background Subtraction

Background subtraction is a much-studied and well-understood problem in the literature. The idea is to separate the salient *foreground* objects in a scene from the uninteresting *background*. For example, consider a video of an urban street corner captured using a fixed camera. Every so often, pedestrians and cars (the foreground objects) move past the camera, but the buildings and environment (the background objects) remain relatively unchanged. Many background subtraction algorithms measure the change in pixel intensity values in order to classify each pixel as belonging to either the foreground or background. Thus, these techniques capture motion in an image.

[18] presents one approach to background subtraction that uses kernel density estimation (KDE) to estimate the probability that a pixel belongs to the background. (More information about using KDE for modeling the background can be found in [4].) KDE works well because thresholding a a probability distribution is much more robust than thresholding actual intensity values. [18] also presents a foreground model can be used for a marginal increase in performance. More importantly, these authors also describe a Markov random field (MRF) model that considers neighboring pixels in determining the probability. The MRF model tends to improve detection of objects by filling in any “missing” pixels within the silhouettes of objects, as shown in Figure 12.

Unfortunately, although the KDE approach yields excellent results, it is also very computationally intensive and is therefore quite slow. Since our system will be used to process many hours of video, it is important that we perform the background subtrac-

Figure 12: Background-subtracted frame before (left) and after (right) enabling the Markov random field (MRF) model. Notice that the MRF model helps to "fill in" the interiors of the silhouettes of the cars, the truck, and the pedestrians.

tion as efficiently as possible. A simpler, faster alternative is to use an exponentially-smoothed moving average (i.e., a recursive filter) to estimate the mean and variance of each of the pixels in the image. This approach also requires less memory than KDE, as it only requires storing an average image, rather than a long buffer of images. This recursive update equation is as follows:

$$m(x, y) = (1 - \alpha) * m(x, y) + \alpha * I(x, y) \tag{2}$$

where m is the mean intensity image, I is intensity image for the new frame, x and y are the pixel coordinates, and α is the smoothing parameter. This model is much faster than KDE, simpler to implement, and still yields reasonable results.

In this project, I implemented both the KDE and recursive moving average algorithms for background subtraction. Yifan Yu [25] wrote some C code that demonstrated several different background subtractions. I leveraged his code as a starting point, but I ported it from Microsoft Windows to Linux, cleaned it up and improved it, added the MRF model, and adapted it to the needs of our system.

Our final system uses the moving average approach with an MRF model in our system. The formula for this variable-threshold hypothesis test is as follows:

$$|I^k[x, y] - B^{k-1}[x, y]| \underset{S}{\overset{M}{\gtrless}} \theta \exp\left((Q_S[x, y] - Q_M[x, y])/\gamma\right) \tag{3}$$

where k is the frame number, x and y are the pixel coordinates, B^K is the estimated background image (i.e., $m(x, y)$ when the moving average approach is used), Q_M and Q_S are the numbers of moving and static neighbors, respectively, and *gamma* is a

tuning parameter that adjusts the impact of the Markov model.

I have written a C++ class called *ProcessingBlock* that defines an interface for a black box that takes a frame of input, processes it, and produces a frame of output. Deriving the background-subtraction (and also the behavior-subtraction) classes from this abstract *ProcessingBlock* base class makes these blocks simple to use and easy to reuse in any design. Furthermore, using this interface makes it easy to swap one block for another. Along these lines, I have created the *MovingAverage* and *KDE* classes, each of which inherit from the *ProcessingBlock* class, which makes them completely interchangeable.

3.4 Behavior Subtraction

In the coastline videos that we wish to process, there is a great deal of uninteresting, repetitive background motion such as ocean waves and dune grass. This results in many spurious detections when we perform background subtraction. One simple thing that we can do is select a region of interest, ignoring or "masking-out" all activity in regions of the image that we don't care about. While this primitive method works well in situations such as monitoring people on the sandy part of the beach but ignoring all activity in the dune grass and in the ocean, it does not work when we need to analyze objects in the the same region as the false detections, such as boats on the water. Furthermore, this approach requires a human to manually select the region of interest, but ideally we would like to have a method that requires minimal human intervention.

We implemented an algorithm called behavior subtraction to reduce the number of false detections. Behavior subtraction [12] is an algorithm that removes stationary motion from a video. We run background subtraction to create the binary cost video and then run behavior subtraction to remove the false detections. The behavior subtraction algorithm has two phases: training and processing. In the training phase, we examine a window of N frames from the M frames of training data (where $M \geq N$). The training data should exhibit the stationary behavior that we want to remove, but it should not have any "interesting" moving objects. First, we sum up the cost at each pixel in the window. Then we shift our window by one frame and perform another summation. We record the maximum of each of the summations at at each pixel at each window location. This completes the training phase. The next phase is the processing phase. We perform the same kind of summations over the sliding

window as we did during the training phase, except that we compare the sum against the maximum, rather than updating the maximum. If the difference between the sum and the maximum is greater than a certain threshold, we detect the pixel as interesting motion and write a white pixel to the output video file; otherwise, we write a black pixel. The rationale behind this is that an "interesting" object will occupy those pixels for more frames than the maximum frames occupied by stationary behavior, thereby allowing us to discriminate between interesting and uninteresting motion. Overall, the behavior subtraction algorithm was relatively quick and simple to implement.
The behavior subtraction code takes three parameters:

1. the number of frames in the buffer (a.k.a. sliding window) over which to average
2. the number of frames in the training sequence
3. the detection threshold

The parameters are pretty easy to tune, since there is a pretty wide range of what works. Generally, longer training sequences and longer buffer lengths give better performance. In some of my tests, typical training sequences ranged from 500 to 2000 frames and typical buffer lengths ranged from 100 to 500, although this depends on the type of motion in your data, as well as how much of your data you have available. For example, the buffer should be long enough to completely capture a full cycle of the stationary behavior, but not too much longer than necessary, especially if you don't have much training data available. When selecting your training data, you want enough frames to sufficiently capture the stationary behavior that you want to remove, but you must make sure that the training frames do not contain any of the objects that you wish to preserve, or else those objects will be removed as well. The detection threshold I used in my tests was typically very small, ranging from 0 to 10. A higher threshold will remove more pixels since it means that the desirable motion must last for that many frames longer than the maximum number of frames in the unwanted motion. Generally, one should keep the threshold at 0 unless trying to remove noise.

Here are some images from my tests. Figure 13 shows a frame from a video of ripples on a pond, and Figure 14 shows a frame captured from the camera on Great Point, Nantucket, on 2010-04-23. In each figure, the image on the left shows the original frame, the image in the center shows the results of background subtraction, and the image on the right shows the result of performing behavior subtraction on the background-subtracted image.

Notice that the ripples and small waves on the pond in Figure 13 are almost

Figure 13: Ripples on a pond: original image, background subtraction, and behavior subtraction

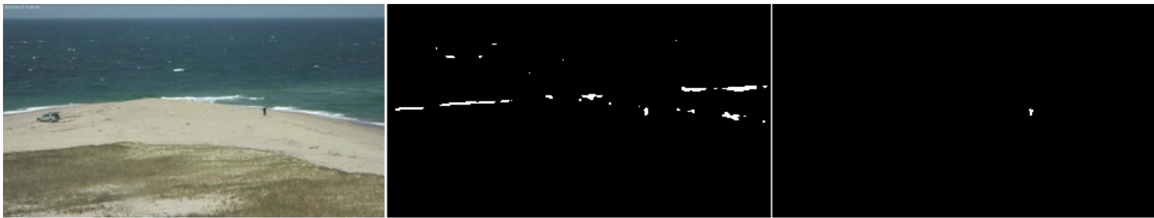


Figure 14: Beach: original image, background subtraction, and behavior subtraction

entirely eliminated. Behavior subtraction performed well here because the motion of the ripples was very stationary and we had plenty of training data.

Notice that if the objects in the video are too small, they can easily be mistaken for noise. We spent a lot of time carefully tuning the parameters to obtain the result in 14 in order to remove the waves while preserving the human. We were lucky in this particular example that we were able to preserve the person; in many other similar videos we processed, the tiny humans on the beach were completely removed. This is particularly true if the people are walking quickly, since moving people don't occupy many pixels for very long.

It is also important to make sure that the people and cars not present in the training sequence; otherwise, behavior subtraction will remove them from the processed data at the places that they were located in the training video.

The behavior subtraction code performed reasonably well for the ocean data. Obviously, it was not able to completely remove all waves because the waves are not entirely stationary, but the behavior subtraction still accomplished what it was supposed to do: it reduced the amount of tests that we must perform using the somewhat costly covariance matrix object detection.

As an interesting aside, It was difficult to find a video of a boat with lots of waves because boats generally don't go out on the ocean where on days when there are lots of waves.

As expected, the code for behavior subtraction runs very quickly. The execution

time benchmarks can be found in the *Experimental Results and Discussion* section.

3.5 Masking

For this project, we have also written some code that takes takes a binary image as a mask and applies this mask to each of the input frames. When it applies the mask, it basically sets any pixels from the input frame that correspond to black pixels in the mask to zero, while leaving any pixels in the input frame that correspond to white pixels in the mask unchanged. This allows the system to ignore specified regions of the image. For example, in an application that only cares about activity on the beach, you could mask out the regions of the image corresponding to the ocean, so that any activity caused by waves or boats would be ignored. Obviously, this approach is only viable when there is a specific region of the scene that can be ignored; if some activity occurs in the masked-out region, we will miss detecting it.

3.6 Obtaining the Regions of Interest

We use a simple connected components algorithm to identify the regions in the behavior-subtracted video corresponding to where the objects of interest are potentially located. Since the covariance matrix-based detection algorithm operates on rectangular regions of pixels, we circumscribe an axis-aligned bounding box (AABB) around each of the contours we detect with connected components.

We discard any AABB smaller than a given threshold (say 5 pixels by 5 pixels) as too small to contain any of the target objects. This helps to reduce the effects of any noise in the behavior-subtracted video.

We also increase the size of each box by a constant scale factor, say 20 percent. This helps to ensure that that bounding box completely captures the target object. This margin is important because the edges of the objects obtained from background subtraction are often imperfect, and the covariance matrix-based detection algorithm is robust and still works well when the bounding box is a little bigger than the object. When the bounding box is smaller than the object, we risk losing the edges of the objects; this is a big problem since our feature vectors are based on object shape.

The resulting set of boxes is passed along to object detection block.

In this project, we also explored an alternate approach for obtaining the rectangular regions of interest that involved counting the active pixels in each row and in each column, and thresholding the peaks of their joint histogram to find areas of

high active pixel density. We started with this approach because we wanted a fast and simple algorithm, but we soon decided to use OpenCV for connected-components using the *cvFindContours* function because it is robust, easy to use, and relatively fast.

Sometimes, gaps in the background-subtracted image cause connected components analysis to segment the object into distinct pieces. One approach I tried for merging these pieces was to scale up the size of the returned rectangles so that adjacent rectangles overlapped slightly and then use this to generate a new binary image. Repeating connected-components analysis on the new binary image would return contours that now encompassed the all of the sub-rectangles of the objects. However, this approach is slow because it requires two passes at connected-components, and increasing the sizes of rectangles for the second pass reduces the accuracy of the outlines. Ultimately, I found that this technique is unnecessary because if we simply tune the background-subtraction step to be more sensitive, it does a much better job at filling in the entire object, thereby eliminating the gaps that cause the segmentation.

3.7 Covariance Matrix-Based Object Classification

As explained in the *Literature Review* section, we use a covariance matrix-based approach to determine whether the rectangular region from connected components contains a car, a boat, a person, or none of these. The main idea is that we compare the covariance matrix for the given rectangular region to a reference image from a library or dictionary of images and check if their similarity is within a specified threshold.

The covariance matrix is easy to compute. First, for each pixel in a given region of pixels, compute the 6-feature vector from Equation 1. Then, take the covariance of all of these vectors from the region to obtain the $6{\times}6$ covariance matrix. Once the covariance matrix is computed, the matrix logarithm of the covariance matrix can be found via eigendecomposition, following the procedure explained in *Literature Review*.

For all of the images in each of our dictionaries, we first pre-compute the log covariance matrices C_i^d, where i is the image index and d is the dictionary index. Then, for each query region in the images from our bounding boxes, we compute the

log covariance matrix Q. Classification is then achieved using the following formula:

$$\text{class}(Q) = d^* \qquad \textit{if} \quad (d^*, i^*) = \arg\min_{d,i}(R_i^d) \quad \& \quad \min_{d,i}(R_i^d) < \psi^d \tag{4}$$

where

$$(R_i^d) = \| \log(Q) - \log(C_i^d) \|_2 \tag{5}$$

where $\|\cdot\|_2$ denotes the Frobenius norm. Obviously, if the distance between the Q and all C_i^d is greater than the threshold for all d, the region is determined to be neither a car, a boat, nor a person.

We decided not to use "integral images" technique (see [23]) because we only have to check relatively few locations and scales per frame (since our regions of interest come from background subtraction; we do not perform a brute-force sweep over the entire image). Therefore, it is not worth precomputing the integral image for covariance over entire images.

3.7.1 Distance Metrics

As already explained, this project considered two different metrics for similarity between two covariance matrices: the Forstner-Moonen approach [5] and the Frobenius norm of matrix logarithms approach [1]. While the literature has shown success with both approaches, we decided to adopt the latter for several practical reasons. First, the log-matrix approach is easier to implement because it only requires solving a standard eigendecomposition, not solving a generalized eigenvalue problem as is required by the Forstner-Moonen approach. Although MATLAB comes with routines for solving generalized eigenvalue problems, many simpler linear algebra libraries, such as the *Armadillo C++ Linear Algebra Library* adopted in this project, only support simple eigendecomposition. Furthermore, the Frobenius norm approach is more efficient for our application because we can pre-compute the log covariance matrices for our dictionary; thus, each query means taking only a summed difference of squares between the query log covariance matrix and each of the log covariance matrices in the dictionary. This is in contrast to the Forstner-Moonen approach that requires solving an eigenvalue problem for every comparison between the query matrix and the dictionary, which is clearly much more expensive.

3.7.2 Dictionaries

For each class of object that we wish to detect, we created a library or dictionary of many different images of that class. We downloaded the images from Google Images, rather than extract the images from our video data, so as not to bias our results. The images are of assorted sizes and we have been cropped as necessary to remove extraneous background clutter. The dictionaries we have used in most of our tests each contain roughly 10 to 100 images. A few examples from our dictionaries can be found in Figure 15. Our system easily can be scaled to detect more types of objects simply by adding more dictionaries of images.

Figure 15: Three samples from each of the dictionaries: (a) boats, (b) cars, and (c) people used in feature-covariance detection. Images have been obtained from a search on Google Images.

Within each object class, we tried to be fairly diverse in our selection of images to make our dictionaries more representative and thus more robust at detecting a variety of objects within that class. For example, our database of cars consists a variety of different automobiles (sedans, pick-up trucks, Jeeps, etc.) with several different orientations. Since our feature vector does not contain color, it does not matter what color these objects are; only the shape matters. Our people database contains primarily black-and-white silhouettes of people standing in different poses, and this works quite well because only the silhouettes of the people are discernible

at the far distance between the camera and the people in our test videos. In all, the dictionaries must be fairly representative of the objects found in the videos. For example, if the boats in the videos have wake, the dictionaries should also include boats with some wake.

Our research has shown that detection works better if objects with significantly different shape are treated as separate objects. For example, the sails of a sailboats have large triangular shape, which is quite different than the hull and cabin of most motorboats, so detection works best if we use a separate dictionary for each type of boat. For more information about these results, please refer to the reports (particularly the “Boats” report) in Appendix C.

We also ran a few experiments to ascertain how much of a difference the direction in which the objects in the dictionary are facing matters. The question arose as we were creating the dictionaries because we did not know how many objects we need to face to the left or to the right, or how much this really matters. In our tests, we created an augmented dictionary of boats by taking all of the images from our boats dictionary and adding a horizontally-mirrored copy of each image to the dictionary. We also created two dictionaries in which all of the boats face to the left or to the right. We found that our original dictionary, which consists of some boats facing to the left and some facing to the right, performs approximately the same as the the augmented dictionary, and that both the original and augmented dictionaries performed slightly better than either the all-face-left or all-face-right dictionaries. For more information about these tests, please refer to the “Boats” report in Appendix C. It is unclear why there was not a significant difference between the original and augmented dictionaries, or even why the differences between these and the same-direction-facing dictionaries were not more significant. This could be due to the fact that even with a small cabin or conning tower, a side-view of a boat is roughly symmetric. Obviously, more work behind the math about how symmetry in features affects the covariance matrix needs to be done in order to better understand this, but we have left this for future work.

3.7.3 Testing

We performed extensive testing 1) to make sure that the covariance-based classification algorithm is coded properly, 2) to better understand how the technique works, and 3) to test the accuracy of this technique. One of our greatest worries throughout the project was whether or not the covariance-based detection approach is discriminative enough to differentiate between cars, boats, people, and anything else going

on in the scene.

As an initial sanity check, we also tested how well the algorithm performs when using a dictionary consisting of samples of an object taken from earlier frames of the same video. For example, we manually extracted a region around a boat for the first few frames of a video, and then tested the detections in the subsequent frames as the boat drove across the screen, in order to make sure that we could reliably detect the boat. Although unrealistic, this test gave us confidence that our code was doing what we expect, thus corroborating the correctness of our code.

We also tested each of our three dictionaries from Google Images with frames from the coastal videos. The results of these tests can be found in the reports in Appendix C.

Overall, we fear that our feature vector simply does not contain enough information to be able to discriminate between different types of objects and between an object of interest and other. The simple, 6-feature vector in Equation 1 really does not contain very much information about an object, and the algorithm can get confused between two different classes of objects with similar texture and roughly the same number of edges. Fortunately, the implicit information about the object's motion from the background subtraction allows us to disregard many regions in the scene that may otherwise have caused false positives, greatly helping with detection accuracy. Furthermore, our preliminary tests have shown that augmenting the feature vector to include more information, such as color data, can improve detection accuracy. Much future work is still needed in this area.

3.7.4 Choosing Exactly One Class

Under the formulation shown above in Equation 4, it is possible that a rectangular region will be detected as belonging to two different classes of objects. For example, a region may be detected as both a car and a boat. This may happen if the two classes are very similar or if the detection threshold is too high. Since only one object an occupy a region at a time (i.e., our system is not really designed to handle occlusions), we would like our system to select only the single, most-likely class of object.

We can select the class of object that matches the best by selecting the class whose dictionary yields the lowest minimum distance, or alternatively, the lowest average distance to all elements in the dictionary. We have tried both approaches in this project and have adopted the former.

However, the problem with the minimum distance approach is it is susceptible to

misclassifications. In other words, since we are only basing the decision on a single minimum-distance object, any outliers in the dictionaries can easily cause the system to label a region as the wrong class of object. We therefore propose using a majority vote of the K nearest neighbors to make the class decision. In other words, out of the K minimum distances between the query region and all of the dictionaries, we determine which dictionary produced the most votes and we use this to select the object class accordingly. We have left this as an area of future work.

3.7.5 Automatic Threshold Selection

Selecting a suitable distance threshold to balance between false detections (threshold too high) and misses (threshold too low) can sometimes be a a difficult task. As discussed in the *Literature Review*, one approach for selecting a suitable threshold is to generate a histogram of the pairwise distances between all of the images in each dictionary. The threshold can be chosen to be the distance corresponding to a specified confidence level, the percentage of area under the cumulative distribution function. A similar leave-one-out cross-validation (LOOCV) technique has been used successfully by [9].

In this project, we spent a little time experimenting with this LOOCV technique, generating and analyzing distributions for different dictionary data sets. A few of these histograms can be found in the reports in C. However, selecting a suitable threshold from these histograms is not as intuitive and simple as one might think, so our final approach uses manually-set thresholds. We still believe this LOOCV approach has merit, but we did not have time to study it further, and therefore we have left it as a direction for future work.

3.7.6 Augmented Feature Vectors

We also ran a few experiments in which we augmented the 6-feature vector from Equation 1 to a 9-feature vector that also contains the red, green, and blue pixel color components. Our preliminary tests showed that this additional color information significantly improves detection robustness for certain classes of objects. However, we did not perform enough experimentation to see how this affects the choice of dictionaries. For example, since cars come in a variety of colors, it is probably necessary to make sure that a wide variety of cars are represented. Of course, this makes the size of the dictionary much larger, as combinations of different orientations

and different colors must be represented. One question that arises is how do we ensure that sufficient, but not too much, object diversity is captured in the dictionary? Moreover, should we create separate dictionaries for separate colors of cars? These are important questions, but they are outside the scope of this project. There is still a significant amount of future work that needs to be done with regards to how to best select images for the dictionaries.

3.7.7 Scaling and Weighting Features

Several authors, including [21, p.6] and [6, p.38] recommend scaling or normalizing the features, in order to make the covariance of the features invariant to the size of the image.

This makes sense, particularly for the gradients. For example, if the pixel value changes from 0 to 100 across two images, but one of the images is twice as wide, then the change in intensity divided by the change in position will be twice as large for the smaller image than the larger image.

However, we ran a few empirical tests, comparing detection accuracy between runs using the scaled and unscaled features, and found that scaling had little effect on the detection accuracy. For our scaled features, we used $sqrt(rectangle_width * rectangle_height)$ to normalize both directions. (A naive approach would be to normalize the y-dimension by the height and the x-dimension by the width, but this might lead to problems when comparing rectangles of different aspect ratios due to unequal scaling between of the dimensions.) Since scaling seemed to make little difference in our tests, we abandoned the scaling and thus use only unscaled features in our final design.

We also performed some simple experiments in which we tried weighting some of the features to make certain features more significant than others. For example, we tried multiplying the color values (note that here we are using the RGB-augmented feature vector) with large constants and the gradient values with small constants to make color relatively more significant than shape. However, we empirically found this weighting to make little difference on the detection accuracy, or even on the shape of the 3D plot of distances to the dictionary (see Figure 4 as an example).

In a related-but-different test, we experimented to see whether or not taking the absolute value of the gradients, as done in the feature vector proposed in [21], actually makes a difference. We found that taking the absolute values did not make a difference; we observed no change in detection accuracy when we did not take the

absolute value versus when we took it. In our final design, our code still takes the absolute value, for no better reason than to agree with the literature.

In order to better understand why scaling and weighting have made little difference in detection performance, we need to take a closer look into the mathematics behind the covariance of the features, but we have left this as future work.

3.7.8 Eigenvalue Considerations

An important edge case to consider is what happens when you are computing the log covariance matrix and the eigenvalues of the covariance matrix are non-positive. If we attempt to take the logarithm of zero, we get negative infinity, and if we attempt to take the logarithm of a negative value, the result is imaginary. Thus, we handle negative and zero eigenvalues by replacing them with the value 1.0. Thus, taking $log(1.0)$ gives us zero, and so this component of log covariance matrix does not contribute to the Frobenius norm distance. This makes sense logically because these problematic eigenvalues arise where there is a linear dependence, or in other words, when a particular feature does not contribute any new information about the similarity of the objects. Setting the log of the eigenvalue to zero makes sense because the zero entry does not contribute to the Frobenius norm. In other words, we are just discarding useless information when we make the comparison between the matrices.

But the question still remains: what causes these negative or zero eigenvalues to arise? First, recall that covariance matrices are positive semi-definite. However, only strictly positive-definite matrices have nonzero eigenvalues. So, clearly, it is possible that the covariance matrix can have zero eigenvalues. But what causes a negative eigenvalues? Floating-point rounding error. The negative values that arise are very tiny, near-zero negative values, and due to error in the eigenvalue computation, they are negative rather than zero or tiny positive values.

Furthermore, what causes an eigenvalue of zero? An eigenvalue of zero implies that the rank of the matrix is wrong, or in other words, that there is some linear dependence between the entries. Recall that a matrix is invertible if-and-only-if its rows and columns are linearly independent. Therefore, if the columns are linearly dependent, the matrix is invertible. The contrapositive of this states that if a matrix is NOT invertible, then the rows and columns are NOT linearly independent. Thus, if a matrix is singular (i.e., non-invertible), its determinant is zero and thus there is some dependence between values. It makes sense if we ignore the linearly dependent factors.

More concretely, zero eigenvalues in the covariance matrix occur when the variance of a feature is zero, which occurs if all of the pixels in a region have the same value. You can easily simulate this in MATLAB or GNU Octave by creating a small matrix of identical values. However, in practice, having identical pixel values seldom happens, since the pixels in a region are rarely all the same value. My code that replaces the non-positive eigenvalues with 1.0 will also print warning messages if this ever occurs, but in the entire twelve months of working on this project, I have yet to see this happen for real data.

3.7.9 Video Condensation

In this project, we implemented video condensation entirely in C++. We leveraged the work [24] of a previous student, Huan-Yu, who had created a single-threaded MATLAB implementation of video condensation that used MATLAB's MEX interface to optimize the most time-consuming steps of the algorithm (ribbon carving) in C code. We first ported this code entirely to C++, creating a baseline, single-threaded implementation, which we compared against the MATLAB implementation to make sure that the outputs were identical, frame-for-frame, pixel-for-pixel. (We obviously used lossless compression for the output videos to ensure a valid comparison.)

Video condensation is an extremely computationally-intensive algorithm and we wanted to speed up the processing time by utilizing multiple processor cores. We developed a pipelined, multithreaded algorithm for the video condensation to allow us to process each pass (flex 0 through flex 3) using separate processor cores. We found that this reduced processing time by up to a factor of three for typical hour-long beach videos, compared to a traditional single-core approach. However, note that it is only effective for videos that are long enough to fill up the pipeline. A report detailing some of our early benchmark experiments can be found in Appendix F.

The reason why we decided to use a pipelined approach, rather than finding some other way to partition the problem, is that it is a relatively simple way to leverage multiple cores, achieving noticeable gains while investing little time and effort; the video condensation algorithm does not need to be restructured or redesigned at all to accommodate the pipeline.

The multithreaded pipeline is basically an extension of the producer-consumer design pattern. The algorithm that I developed is original work; I could not find anything in the literature that discussed pipelining processing blocks using multiple threads, so I designed it myself, though I would be surprised if someone has not

already thought up the same technique. I use a very large circular buffer of memory frames, large enough to keep all the threads busy once the pipeline is full. I use semaphores to pass pointers to the frames from one stage of the pipeline to the next. Once a frame is no longer being used (either because it was a frame of data that was carved out by a ribbon, or because it is the output of the last stage of the pipeline), it gets passed back to the first pipeline stage so that it can be recycled and filled with the next input frame of data. Overall, this general multithreaded pipline approach is very clean, simple, and can be applied to other applications as well, not just video condensation.

I have designed the multithreaded pipeline for a system with four cores. The first core is responsible for computing all of the preprocessing (which may include background subtraction, behavior subtraction, connected-components analysis, object classification, etc.) and flex 0. The second, third, and forth cores are responsible for flex 1, flex 2, and flex 3, respectively. This is illustrated in Figure 16. Note that this system is designed only to process up through flex 3; we do not bother to process higher amounts of flex because each subsequently greater flex provides only a diminishing marginal benefit.

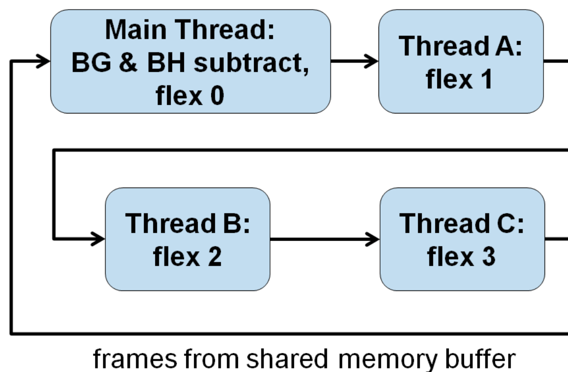


Figure 16: Multithreaded, Pipelined Implementation

Each pair of successive pipeline stages can be viewed as a producer-consumer pair, with a shared buffer of pointers to memory frames (not shown in the diagram). The shared resources (the pointers to memory frames) are coordinated using semaphores.

The memory requirements of the multithreaded, pipelined approach are very large, much larger than the memory requirements for the single-threaded approach, because we must allocate enough memory frames to process multiple flex passes simultaneously. In contrast, the single-threaded approach stores the results of each flex pass to temporary files, and thus only needs a buffer large enough to store the frames for only

the flex pass that is currently being processed. The high memory requirements for the multithreaded, pipelined approach restrict the maximum resolution of the video frames that can be processed.

As long as you allocate enough frames in the circular buffer to keep all of the stages busy simultaneously, everything works as intended. There is no need to worry about one thread hogging all of the memory. In general, when you write multithreaded programs, you worry about one thread hogging all of the memory. However, by the nature of the fact that we're pipelining all of our threaded stages, everything works itself out. Each stage needs the results of the previous stage before it can run, which causes the stages to wait until they have the resources they need. There is a limit to the maximum amount of resources that each stage will ever consume, and there are plenty of resources available. The threads simply wait for the data from the previous stage to become available before they start to process it. Thus, no stage will ever starve any other stage.

3.8 Improvements

After we finished coding and unit-testing all of the components, we tested the entire system and found a few things that needed improvement. First, sometimes the detectors do not steadily detect objects as they move across the scene; in other words, some moving objects are occasionally be missed in a frame here or there. Another problem we encountered was that the bounding boxes around the boats were very wide because they captured the wake as well as the boat itself. A third problem is that with our current background-subtraction algorithm, moving objects that stop moving and remain motionless for a while eventually cease to be detected. We developed techniques to address each of these issues, as described in the next few subsections.

3.8.1 Variable threshold adjustment for improved temporal consistency

The occasional missed detections over a sequence of frames happens for a variety of reasons: camera vibrations, occlusions, slight changes in orientation of the object, poor background-subtracted silhouettes in some of the frames that adversely affect the size of the bounding boxes, etc.– any of which may cause the object to be occasionally missed. The boats are especially susceptible tho this problem, especially as the wake of the boat changes and the boat rises and dips in the waves of the ocean. Fortunately, notice that all of the objects of interest are moving at relatively low velocities; in other

words, their displacements between subsequent frames is pretty small; they are not moving fast enough to be in the middle scene one moment and gone in the next. We exploit this principle by implementing an adaptive threshold that encourages temporal consistency between objects of the same class in subsequent frames:

$$\min_{d,i}(R_i^d) < \psi^d * max(1, exp(-(\delta_{k-1}^d - \delta_{max})/\delta_\gamma)) \tag{6}$$

where δ_{k-1}^d is the distance to the nearest detection of class d in the previous frame and δ_{max}, δ_γ are parameters.

We found through experimentation that this method is quite effective at improving detections. However, care must be used when selecting the parameters because a detection encourages detections in the surrounding region; if the threshold is too large, the number of false detections in surrounding regions may increase (this is particularly the case with boats and false detections in the surrounding wake).

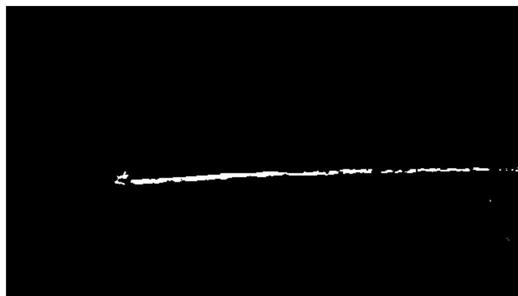
3.8.2 Aspect ratio thresholding to truncate elongated boxes around boat wake

The motion of the water in the wake behind fast-moving boats creates a long trail of activity behind the boat in the background-subtracted and behavior-subtracted videos, as shown in Figure 17a. As a result, connected components analysis gives a bounding rectangle that encompasses the boat plus the wake, as shown in Figure 17b, but we really would rather put a rectangle around roughly only the boat.

We implemented a simple-but-effective heuristic to accomplish this goal. We check the aspect ratio of all of the detected bounding rectangles, and if it exceeds a certain threshold, it means that the rectangle is very wide and likely encompasses boat wake. In this case, we replace the wide box with two smaller boxes, one at either end, of the same height as the original, but with a smaller width (specified by a predetermined new box aspect ratio). Our experimental results showed that this worked quite well, as shown in Figure 17c. For these experiments, we used an aspect ratio threshold of 4:1 and a new box aspect ratio of 2:1.

3.8.3 Handling stopped objects

The reason why objects that have stopped moving for a long period of time are eventually no longer detected is inherent to our algorithm for background subtraction.



(a) Behavior-subtracted frame

(b) Aspect ratio technique not applied

(c) Aspect ratio technique applied

Figure 17: Aspect ratio technique. (a) shows the original behavior-subtracted frame, whose connected components give a large bounding rectangle around the wake, seen in (b). The result of applying the aspect ratio threshold technique is shown in (c).

As the average background image adapts, the stopped objects become the new background and hence cease to be marked with active pixels. As a result of the eroding object silhouettes, the bounding rectangles found by connected-components analysis shrink until they disappear altogether, and when there are no bounding rectangles, there are no candidate regions for the detectors to check.

In some applications, this behavior might be desired. For example, maybe the customer only cares about watching the clips of cars when they enter or leave the scene, but the frames of parked cars can be removed. Nevertheless, we still want to have a technique capable of preserving these stopped objects so that our system is flexible; we can always disable it later for these types of applications.

We overcome the problem of missed detections of stopped objects by continuing to check the regions corresponding to the stopped objects in every subsequent frame, until those stopped objects are no longer detected. A naive first approach that we tried was to basically recheck any region from any previous frame that had a detection. The problem with this was that it produced a lot of false positives, particularly in the wake of the boat. After much thought, we developed a new approach that does not have this problem because it only rechecks an object if that object is not moving.

Our experimental results showed that this algorithm worked fairly well in practice. The following is a description of this algorithm:

1. Loop through each detection d from the list of detections from the previous (i.e., $(k-1)$th) frame. Check to see if the rectangle from d overlaps any detections from the prior frames (i.e., frame $k-2$, frame $k-3$, ... until you reach end of the circular buffer).
   (a) Note that we do not check the class of the objects involved, since we only are interested in finding regions to re-check. So if a car from one frame overlaps with a boat from another frame, that is fine. We just want to see if any type of object in that region is stopped, so that we can continue to recheck it.
2. Let us consider frame $k-2$. If the rectangle from d overlaps a rectangle from the detections in frame $k-2$, we set the displacement for frame $k-2$ equal to the distance between the centers of the boxes.
   (a) If no boxes overlap, we set the displacement for frame $k-2$ equal to some reasonably large size (i.e. larger than the threshold, but of the same order of magnitude), for example, twice the length of the diagonal of d's rectangle, to penalize frames that missed detections.
   (b) If there are multiple overlapping boxes, then we arbitrarily just take the first one we come across. But this should never happen because code from the end of the previous iteration always gets rid of overlapping detections.
3. After we have computed the displacements for frames $k-2$, $k-3$, $k-4$, etc., we find the average of these displacements and we compare it to a threshold. This threshold is basically a function of the diagonal of the box (so that it varies appropriately with the size of the box), times a constant scale factor parameter (`STOPPED_OBJECT_THRESHOLD_SCALE`).
   (a) If this average displacement is less than the threshold, it means that the object is stopped.
   (b) If this average displacement is greater than the threshold, it means that the object is moving.

4. If the object is stopped, we need to recheck d's rectangle in the current (i.e., kth) frame. To do this, we add d's rectangle to the list of bounding boxes to check in the current frame. Otherwise, if the object is NOT stopped (i.e., moving), we should NOT check this rectangle in the current frame, so do NOT add d's rectangle to the list.

Note that we use circular buffer of the frame history, not just a single frame $k-2$, because we want to make our system more robust to missed detections. Without a buffer of frames, if one or two frames misses the object (which is possible, even for stopped objects, if there are camera vibrations or changes in gain, for example), the system will lose track of the stopped object. However, with a buffer, even a miss in a frame or two does not pose a problem; the detections will be maintained. We empirically found that a circular buffer length of 5 frames works well in practice.

Also note that at the end of each iteration, we check all pairs of detected rectangles for overlap. If any detection overlaps any other detection of the SAME class of object, remove the one with greater distance to the dictionary. If any detection overlaps with any other detection of a DIFFERENT class of object, we keep only the object that has the lower average error.

The reason why we discard the overlapping rectangles is so that a stopped object does not generate an infinite number of rectangles to check over time (i.e., redundant boxes to check). Also, our system does not handle occluded objects very well (after all, our connected-components analysis cannot distinguish the difference between two adjacent moving objects; they look like a single blob), and so we assume that two different objects cannot occupy the same location at the same time, which is a reasonable assumption most of the time in the coastal scenes.

It is important to disable the variable threshold technique (described in Section 3.8.1) because we do not want to get false positives in the same location after a stopped object moves.

After we implemented this code, we tested it with a video of the car whose background subtraction silhouette erodes over time, in order to make sure that it accomplishes our goal of maintaining consistent detections. We also tested it by running it with a video of the boat to make sure that it did not cause a lot of false detections.

4 Experimental Results and Discussion

This section summarizes our final results and key findings.

We have collected hundreds of hours of coastal videos at 640×360 resolution and 5fps from cameras mounted at Great Point, Nantucket Island, MA. Figure 18 shows two examples from one of the videos we processed. We used the following ranges of parameters:

- background subtraction: $\alpha = 0.005$, $\theta = 20 - 25$, $\gamma = 1.0$, 2nd-order Markov model with 2 iterations
- behavior subtraction: $300 - 2500$, $N = 50 - 100$, $\Theta = 0.0 - 1.0$
- connected-component analysis: 5×5 bounding box threshold, 20% box enlargement
- object classification: for boats, cars and people consisting of 30, 80 and 62 objects, respectively, with corresponding thresholds ψ between 2.5 and 4.0.

Note the relative resilience of background subtraction to the presence of waves (second row in Figure 18). Although behavior subtraction provides only a slight additional wave suppression in this case, for videos with choppy water surface behavior subtraction offers a significant improvement for larger values of M. Also, note the detection of the boat on left despite a long wake behind it. The condensed frame on left (bottom row) shows two boats together that were never visible in the field of view of the camera at the same time. Similarly, on right, four boats have been combined into one condensed frame thus shortening the overall video.

The effectiveness of our joint detection and summarization system can be measured by the condensation ratio (CR) achieved for each class of objects (or combination thereof). Table 1 shows detailed results with cumulative condensation ratios (after flex-3) of over 18:1 for boats, 9:1 for people, but only about 5:1 for boats or people. Clearly, when one wants to capture a larger selection of objects, the condensation ratio suffers. Condensation ratios for another video with boats, cars and people are shown in Table 2.

Note the last row in both tables labeled "behavior subtraction" with very low condensation ratios. These are results for the whole-frame behavior subtraction output being used as the cost in video condensation instead of being limited to the bounding boxes of detected objects. Clearly, the spurious detections (e.g., due to ocean waves) at the output of behavior subtraction reduce the condensation efficiency.

Table 3 provides the average execution time for each stage of processing in a single-threaded C++ implementation on an Intel Core i5 CPU with 4GB of RAM running

Figure 18: Samples of typical input video frames (top row) and outputs from the processing blocks in Figure 11: (row 2) background subtraction, (row 3) behavior subtraction, (row 4) object detection, (row 5) video condensation.

Table 1: Number of frames after each flex-step and cumulative condensation ratios (CR) for 38-minute, 5 fps video with boats and people (11,379 frames after behavior subtraction).

Cost	# of frames after each step				CR
	flex-0	flex-1	flex-2	flex-3	
boats only	1346	743	662	614	18.53:1
people only	3544	2411	1772	1265	9.00:1
combined boats and people	4666	3225	2887	2414	4.71:1
behavior subtraction	11001	8609	8147	7734	1.47:1

Table 2: Number of frames after each flex-step and cumulative condensation ratios (CR) for 22-minute, 5 fps video with boats, cars and people (6,500 frames after behavior subtraction).

Cost	# of frames after each step				CR
	flex-0	flex-1	flex-2	flex-3	
cars only	768	598	513	439	14.81:1
boats only	835	741	692	589	11.04:1
people only	1729	1528	1527	1519	4.28:1
combined cars, boats, and people	2125	2045	2013	1969	3.30:1
behavior subtraction	6437	5843	5619	5460	1.19:1

Ubuntu 11.04 Linux. Note that the execution times for background subtraction and behavior subtraction depend only on the video resolution (in this case, 640×360). On the contrary, the execution times of object detection and video condensation vary depending upon the data (e.g., more activity means that more regions must be checked for the presence of objects and also that fewer frames can be dropped in the flex-0 stage of condensation). The benchmarks reported in the table were obtained for detections of cars in the video from Table 2 and are representative of typical execution times. Since video condensation operates on blocks of frames, we computed the average processing time of each "flex" pass by dividing the execution time for that pass by the number of input frames to that pass. For the background subtraction, we used second-order Markov neighborhood and two update iterations for each frame. Disabling the MRF model reduces the execution time to 0.156 sec/frame but significantly lowers the quality of the output. The object detection benchmark in the table includes the time for the connected components computation, the bounding box extraction, and the tests of each candidate region against three dictionaries (cars, people, and boats).

Clearly, our single-threaded implementation can process only 0.2 fps. Even for a 5 fps input video this is far from real time. One possibility to close this gap is by leveraging parallelism. For example, we have experimented with a pipelined, multithreaded implementation of video condensation for different flex parameters. We

Table 3: Average execution time for each stage of processing.

Processing Technique	Average Execution Time
Background Subtraction	0.292 sec/frame
Behavior Subtraction	0.068 sec/frame
Detection and Classification	0.258 sec/frame
Video Condensation	
flex 0	0.034 sec/frame
flex 1	2.183 sec/frame
flex 2	1.229 sec/frame
flex 3	0.994 sec/frame
Total	5.058 sec/frame

found that on a quad-core CPU this reduced the processing time by up to a factor of three for typical hour-long beach videos, compared to a traditional single-core approach. Similar parallelism can be applied to the other steps.

5 Conclusions and Future Work

In this project, we combined multiple video processing algorithms to create a robust coastal surveillance system that should be useful for biologists, ecologists, environmentalists, and law enforcement officials. We tested our approach extensively using real coastal video sequences and showed that our system can reduce the length of typical videos up to about 20 times without losing any salient events. Our system can dramatically reduce human operator involvement in search for events of interest. Currently, our system does not operate in real time but with a careful implementation on a multicore architecture, real-time performance is within reach.

The approach is successful, though imperfect. For instance, the detection block works well, but ocean waves still cause some false positives. The classification block attains reasonable accuracy, but there are still some misses and still some instances of class confusion. The summarization block achieves a 20x reduction in frame count for scenes with high activity, although greater rates of condensation are possible if the detection and classification accuracy were improved. Nevertheless, our system is still a vast improvement over using only the raw background subtraction or behavior subtraction as the cost input to video condensation, as it allows researchers to observe ONLY the events involving objects of interest and filters out spurious activity caused by ocean waves.

In addition to the development of the algorithms, this project contributes a complete software package written in C++ for speed, portability, and ease of deployment on embedded platforms. In particular, our *VideoWriter* and *VideoReader* wrappers

for *FFMPEG* will be especially useful to other students who need C++ video I/O in future projects, since, to our best knowledge, there do not exist any other simple interfaces to *FFMPEG*. All code is clean and thoroughly documented (using *Doxygen* and plenty of source code comments) so that it can be easily leveraged. Another key innovation in this project is the novel use of a pipelined, multithreaded algorithm to speed up the computationally-intensive video condensation algorithm by a factor of three. In addition to an efficient C++ version of video condensation, this project contributes updated, improved versions of the original MATLAB implementation of video condensation for both Linux and Microsoft Windows platforms.

As is always the case in research, a single question leads to exponentially many more, and so it is not surprising that there are many possible directions for future study. One direction for future work is augmenting feature vectors with additional features, such as RGB, or even some SIFT-inspired features. Our preliminary results showed that this greatly improved the ability of the system to discriminate between objects, but it also undoubtedly affects the choice of dictionaries (e.g., need to capture many different colors of cars since color now matters). More work must also go into determining a method for selecting a suitable, sufficiently-diverse-but-still-discriminative dictionary of images. Once the images are chosen, a probability distribution can be calculated, which can then be used to automatically select a detection threshold, but more work is clearly needed in this area. In addition to augmenting our choice of features, we should also consider how to best use those features, looking more closely into things such as scaling, weighting, and symmetry. Another future improvement to the system's ability to differentiate between different types of objects is to select the object class via a best-out-of-K-nearest-neighbors approach.

As a final step to making the system more user-friendly to scientists, we propose creating a graphical user interface (GUI) that allows the user to select the input video files, specify and test parameters for the processing, and batch-process entire directories of videos.

Object detection and classification opens the gateway to performing more-sophisticated, higher-level analysis of the scene, such as object tracking, counting, and generation of other statistics. Future work in these areas will allow scientists, law enforcement, and researchers to better study the behavior of the objects and the interactions between them.

References

[1] Vincent Arsigney, Piere Fillard, Xavier Pennec, and Nicholas Ayache. Log-Euclidean Metrics for Fast and Simple Calculus on Diffusion Tensors. *Magnetic Resonance in Medicine*, 2006.

[2] Shai Avidan and Ariel Shamir. Seam Carving for Content-Aware Image Resizing. *ACM Transactions on Graphics*, 26(3), July 2007. Article 10.

[3] Gary Bradski and Adrian Kaehler. *Learning OpenCV - Computer Vision with the OpenCV Library*. O'Reilly Media, 2008.

[4] Ahmed Elgammal, Ramani Duraiswami, David Harwood, and Larry S. Davis. Background and Foreground Modeling Using Nonparametric Kernel Density Estimation for Visual Surveillance. *Proceedings of the IEEE*, 90(7), July 2002.

[5] W. Förstner and B. Moonen. A Metric for Covariance Matrices. *Technical Report, Dept. of Geodesy and Geoinformatics, Stuttgart University*, 1999.

[6] Kai Guo. *Action recognition using log-covariance matrices of silhouette and optical-flow features*. PhD thesis, Boston University, September 2011.

[7] Kai Guo, Prakash Ishwar, and Janusz Konrad. Action Recognition in Video by Covariance Matching of Silhouette Tunnels. *Proc. XXII Brazilian Symp. on Computer Graphics and Image Processing*, pages 299–306, October 2009.

[8] Kai Guo, Prakash Ishwar, and Janusz Konrad. Action change detection in video by covariance matching of silhouette tunnels. *Proc. IEEE Int. Conf. Acoustics, Speech, and Signal Processing*, pages 1110–1113, March 2010.

[9] Kai Guo, Prakash Ishwar, and Janusz Konrad. Action recognition in video by sparse representation on covariance manifolds of silhouette tunnels. *Proc. Int. Conf. Pattern Recognition*, August 2010. (Semantic Description of Human Activities Contest).

[10] Kai Guo, Prakash Ishwar, and Janusz Konrad. Action recognition using sparse representation on covariance manifolds of optical flow. *Proc. IEEE Int. Conf. Advanced Video and Signal-Based Surveillance*, pages 188–195, August 2010.

[11] Pierre-Marc Jodoin, Janusz Konrad, and Venkatesh Saligrama. Modeling background activity for behavior subtraction. *ACM/IEEE Int. Conf. Distributed Smart Cameras*, September 2008.

[12] Pierre-Marc Jodoin, Venkatesh Saligrama, and Janusz Konrad. Behavior Subtraction. *Proc. SPIE Visual Communications and Image Processing*, 6822:10.1–10.12, January 2008.

[13] Pierre-Marc Jodoin, Venkatesh Saligrama, Janusz Konrad, and Vincent Veilleux-Gaboury. Motion Detection with an Unstable Camera. *Proc. IEEE Int. Conf. Image Processing*, October 2008.

[14] Robert G. Keys. Cubic Convolution Interpolation for Digital Image Processing. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-29(6), December 1981.

[15] Zhuang Li, Prakash Ishwar, and Janusz Konrad. Video Condensation by Ribbon Carving. *IEEE Transactions on Image Processing*, 18(11), November 2009.

[16] Thomas D.C. Little, Janusz Konrad, and Prakash Ishwar. A Wireless Video Sensor Network for Autonomous Coastal Sensing. *Proc. Conference on Coastal Environmental Sensing Networks (CESN)*, 2007.

[17] J. Mike McHugh. Probabilistic Methods for Adaptive Background Subtraction. Master's thesis, Boston University, 2008.

[18] J. Mike McHugh, Janusz Konrad, Venkatesh Saligrama, and Pierre-Marc Jodoin. Foreground-Adaptive Background Subtraction. *IEEE Signal Processing Letters*, 16(5), May 2009.

[19] Robert Laganière. *OpenCV 2 - Computer Vision Application Programming Cookbook*. Packt Publishing, 2011.

[20] Oncel Tuzel, Fatih Porikli, and Peter Meer. Covariance Tracking using Model Update Based on Means on Riemannian Manifolds. *Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, June 2006.

[21] Oncel Tuzel, Fatih Porikli, and Peter Meer. Region Covariance: A Fast Descriptor for Detection and Classification. *Proc. Ninth European Conf. Computer Vision (ECCV 06)*, 2:589–600, May 2006.

[22] Oncel Tuzel, Fatih Porikli, and Peter Meer. Pedestrian Detection via Classification on Riemannian Manifolds. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, October 2008.

[23] Paul Viola and Michael Jones. Rapid Object Detection Using a Boosted Cascade of Simple Features. *Technical Report, Mitsubishi Research Laboratories*, May 2004.

[24] Huan-Yu Wu. Sliding-Window Ribbon Carving for Video Condensation. Technical report, Boston University, December 2009. Master's project.

[25] Yifan Yu. Robust Real-Time Background Subtraction in C++. Technical report, Boston University, November 2010. Master's project.

Appendices

A MS Project Symposium Poster

This is the poster prepared for the BU ECE MS Project Symposium held on 3 May 2012. The original file format for the poster was MS Powerpoint (as that was the format of the department-provided template). Note that unfortunately, some of the formatting for the equations got a little bit messed up when converting to PDF for inclusion in this report.

Detecting and Summarizing Salient Events in Coastal Videos

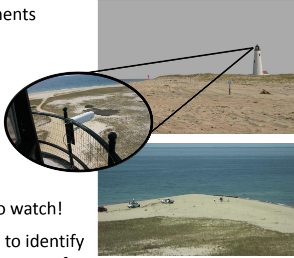
Daniel Cullen, Janusz Konrad, and Thomas D.C. Little {dcullen, jkonrad, tdcl}@bu.edu
Department of Electrical & Computer Engineering, Boston University, Boston, MA

BOSTON UNIVERSITY

BU ECE

Introduction

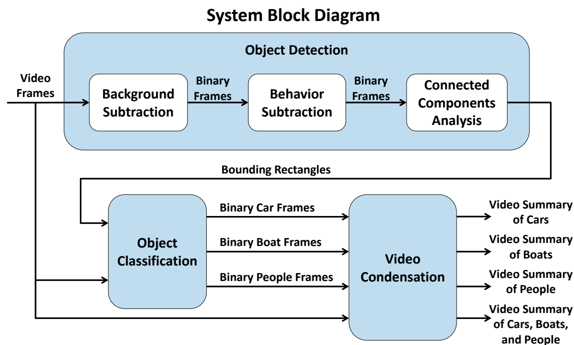
- Motivation: Monitoring coastal environments
  - Case study: Great Point, Nantucket, MA
- Method: Surveillance video cameras
- Applications:
  - Scientific research: ecology, biology
  - Environmental protection
  - Law enforcement
- Challenge: Too many hours for humans to watch!
- Objective: Develop automatic algorithms to identify and summarize salient events (e.g., appearance of cars, boats, people)



Approach

Key components: Object Detection, Object Classification, Video Summarization

System Block Diagram



Object Detection

1. Background Subtraction
   - Finds regions of interest
   - Recursive moving average

$$B^k[x,y] = (1-\alpha) * B^{k-1}[x,y] + \alpha * I^k[x,y]$$

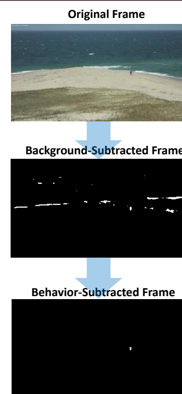
I^k, B^k - image and background intensities in frame # k

   - Variable-threshold hypothesis test uses MRF [2]

$$\left|I^k[x,y] - B^{k-1}[x,y]\right| \underset{S}{\overset{M}{\gtrless}} \theta \exp(Q_S[x,y] - Q_M[x,y]) / \gamma)$$

Q_S, Q_M - number of static and moving neighbors

2. Behavior Subtraction [3]
   - Removes stochastically-stationary motion (e.g., waves)
3. Connected-Components Analysis
   - Finds bounding rectangles around objects



Object Classification

- Detects cars, boats, or people in each bounding rectangle
- Covariance matrix-based approach [4]
  - Feature vector:

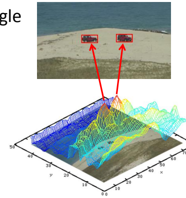
$$\xi(x,y) = \left[x, y, \left|\frac{\partial I}{\partial x}\right|, \left|\frac{\partial I}{\partial y}\right|, \left|\frac{\partial I^2}{\partial x^2}\right|, \left|\frac{\partial I^2}{\partial y^2}\right|\right]$$

  - Pre-compute log covariance matrices C_i^d where i = image index and d = dictionary index
  - Compute log covariance matrix Q for each bounding box
  - Assign object class:

$$class(Q) = d^* \text{ if } (d^*, i^*) = \arg\min_{d,i}(R_i^d) \ \& \ \min_{d,i}(R_i^d) < \psi^d$$

where $R_i^d = \left\|\log(Q) - \log(C_i^d)\right\|_2$

  - If multiple classes found, accept the class with minimum R_i^d



Similarity (*i.e., 1/distance*) to cars dictionary for fixed-size rectangles across image

- Dictionaries
  - 10 to 100 images per class
  - Orientation matters; many captured
  - Only shape matters; color is ignored
  - Require different databases for classes with distinct shapes (e.g., motorboats vs. sailboats)
  - Source: Google Images

- Improvements
  - Adaptive threshold in class assignment encourages temporal class consistency:

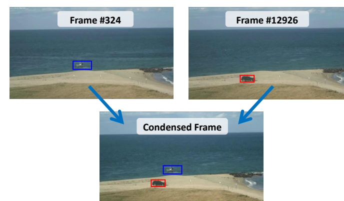
$$\min_{d,i}(R_i^d) < \psi^d * \max(1, \exp(-(\delta_{k-1}^d - \delta_{max}) / \delta_\gamma))$$

δ_{k-1}^d - distance to nearest detection of class d in previous frame; δ_{max}, δ_γ - parameters

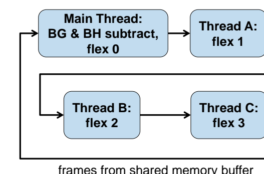
  - Handle elongated rectangles produced from the wake of boats
  - Prevent stopped objects from disappearing as background subtraction adapts

Video Summarization

- Apply the video condensation algorithm [1] to each object class
  - Creates summary of cars, boats, people, or any combination of these
  - High computational complexity
- Contribution: Multi-threaded, pipelined implementation
  - 3x speed improvement over single thread
  - Only effective if videos are long enough to fill pipeline

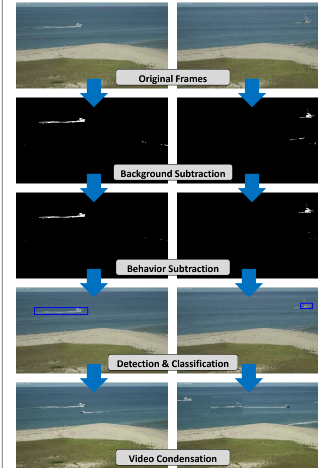


Multi-threaded Implementation



Experimental Results

Outputs of processing steps



Classification examples

Rectangle colors: cars = red, boats = blue, people = green

Summarization results

Results for the Great Point 2010-08-02 Video
640x360 resolution. Video contains boats and people.

Cost function for video condensation	Number of frames in each video					input to flex 3 ratio
	input	flex 0	flex 1	flex 2	flex 3	
behavior subtraction	11379	11001	8609	8147	7734	1.47
boats only	11379	1346	743	662	614	18.53
people only	11379	3544	2411	1772	1265	9.00
boats and people	11379	4666	3225	2887	2414	4.71

Execution time benchmarks

Processing Technique	Typical Execution Time
Background Subtraction	0.292 sec/frame
Behavior Subtraction	0.068 sec/frame
Detection and Classification	0.258 sec/frame
Summarization flex 0	0.034 sec/frame
Summarization flex 1 + flex 2 + flex 3	4.406 sec/frame
Total	5.058 sec/frame

Setup: C++, single-thread, Intel Core i5, 4GB RAM, Ubuntu 11.04

Discussion

- Execution times and summarization ratios vary with the amount of activity in the scene. Results shown are for typical video sequences.
- Our approach yields better condensation ratios than using raw cost from background or behavior subtraction due to fewer false detections (e.g., waves).

See video results at vip.bu.edu

Conclusions

- Approach successful, though imperfect
  - Detection: Works well, but ocean waves still cause some false positives
  - Classification: Achieves reasonable accuracy, but some misses and class confusion
  - Summarization: 20x reduction in frame count for scenes with high activity
- Contribution: Complete software package
  - Written in C++ for speed, portability, and ease of deployment on embedded platforms
  - Uses only free, open-source libraries: *FFMPEG* (video I/O), *CImg* (image I/O), *Armadillo C++ Linear Algebra Library* (solving for eigenvalues), *OpenCV* (connected components)
- Future work
  - Augment feature vector (e.g., with RGB color) to improve accuracy

References

[1] Z. Li, P. Ishwar, and J. Konrad. Video condensation by ribbon carving. *IEEE Trans. Image Processing*, 18(11):2572-2583, Nov. 2009.

[2] J. McHugh, J. Konrad, V. Saligrama, and P.-M. Jodoin. Foreground-adaptive background subtraction. *IEEE Signal Processing Letters*, 16(5):390-393, May 2009.

[3] V. Saligrama, J. Konrad, and P.-M. Jodoin. Video anomaly identification: A statistical approach. *IEEE Signal Processing Magazine*, 27(5):18-33, Sept. 2010.

[4] O. Tuzel, F. Porikli, and P. Meer. Region covariance: A fast descriptor for detection and classification. In *Proc. European Conf. Computer Vision*, May 2006.

MS Project Symposium, 3 May 2012

B Source Code

Please refer to the zipped repository distributed along with this report. There are far too many lines of source code to include in this report.

C Covariance Matrix-Based Detection Reports

This section provides three short reports that I created as I was testing the covariance matrix-based object detection and classification algorithm. There is one report for cars, one for boats, and one for people.

Results of covariance matrix detection using frames of beach video and a library of cars

Written by D.Cullen
Last updated 2012-02-02

Status update

Not much new over break. Been writing final report and cleaning up code and resolving known issues. Started work with detecting cars (similar to boats, but it's good to test our algorithm with other databases, and also a good refresher for me).

Experiment setup (*Contains excerpts from my 2011-11-08 email. Obviously revised for the new experiments.)*

For the distance computation, I used the Frobenius norm between the log covariance matrices of the two images. I used feature vectors containing six features: raw x and y pixel coordinates (where (0,0) is the top-left corner of the rectangular query region) and the absolute values of the first-order and second-order gradients in each direction (where the gradients have been calculated using Keys' cubic convolution interpolation). In other words: features = [x, y, abs(dI/dx), abs(dI/dy), abs(d2I/dx2), abs(d2I/dy2)]. This is the same feature vector suggested by Porikli for detecting shape and ignoring color components.

In order to create my cars database, I downloaded a set of images of automobiles from Google images, cropping them as necessary. (More information about this database will be given later.) I next computed the minimums of the pairwise distances, following the method that Kai explains in his thesis (p.75-76) and in his ICASSP paper (i.e., leave-one-out cross-validation) to build a probability distribution of the distances between the cars. From this, I built a cumulative distribution, which I used to select a threshold distance (i.e., I chose a confidence level and picked the threshold distance that corresponded to this percentage of the area under the pdf). (For the tests described in this document, I set several hard thresholds so that I could better analyze the behavior, but my approach for the real system uses the threshold computed from the CDF.)

I also extracted several frames from the Great Point videos that contain cars and trucks on the beach. I ran my code on each frame in order to detect the vehicles. I performed a brute force search for the vehicles in each frame, sliding a fixed-size search window (80x50 pixels, which is roughly the same size as the cars in most of the frames) all over each frame, stepping by 4 pixels each time in each direction. (For this simple test, I only used a single search window size; I did not check for cars of different sizes.) At each search position, I computed the log covariance matrix, and compared it to the log covariance matrix of each of the images in the database. If the minimum of these distances was less than or equal to my threshold, I marked that position as a detection, and drew a red rectangle in the output video to mark it as such. I have attached a thumbnail of a frame from this video to help you to visualize this.

Observations from these experiments

- Notice that the datestamp causes many false detections, just like we observed with our boat experiments.
- When we are trying to see if our algorithm for setting threshold from the library is good, it's good to test it on lots of data. Since the same threshold was good for different cars and background images, it suggests that our library algorithm is good.

Goals and next steps

- Goal for Prof. Little: Try to answer this question: How many per day? (boats, cars, seals, people, etc.)
- Figure out why datestamp produces such a large peak.
- Is thresholding based on CDF the best approach? Database diversity? Separate classes of objects?
- Feature vectors. Are these features sufficient? Test different features. Also study covariance matrix theory.
- Detecting people?
- Discriminate between object classes. Collect statistics. Make confusion matrix.
- How will I perform my performance (speed & accuracy) benchmarks? How will I generate my tables of statistics?
- Create good database: iss:*/mnt/data/misc/visualdata/video/beach*. Create thumbnails. Document parameters used.
- Symposium in May. Final report. Final code.
- Loose ends: speed up video condensation. More benchmarks. Batch processing. GUI/visualizations?
- I have plenty of things to keep me busy. I'll e-mail in a few weeks when I'm ready to meet or if I have questions.

We created a database of 30 images using a Google image search and cropping where appropriate. The resolution of the images varies, but it is roughly 500x300 on average.

Here are thumbnails of all of the images in our library.

car_001.jpg

car_002.jpg

car_003.jpg

car_004.jpg

car_005.jpg

car_006.jpg

car_007.jpg

car_008.jpg

car_009.jpg

car_010.jpg

car_011.jpg

car_012.jpg

car_013.jpg

car_014.jpg

car_015.jpg

car_016.jpg

car_017.jpg

car_018.jpg

car_019.jpg

car_020.jpg

car_021.jpg

car_022.jpg

car_023.jpg

car_024.jpg

car_025.jpg

car_026.jpg

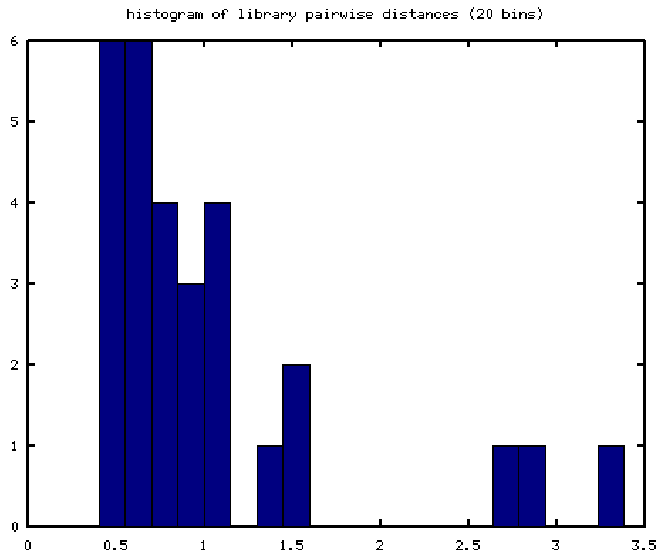
car_027.jpg

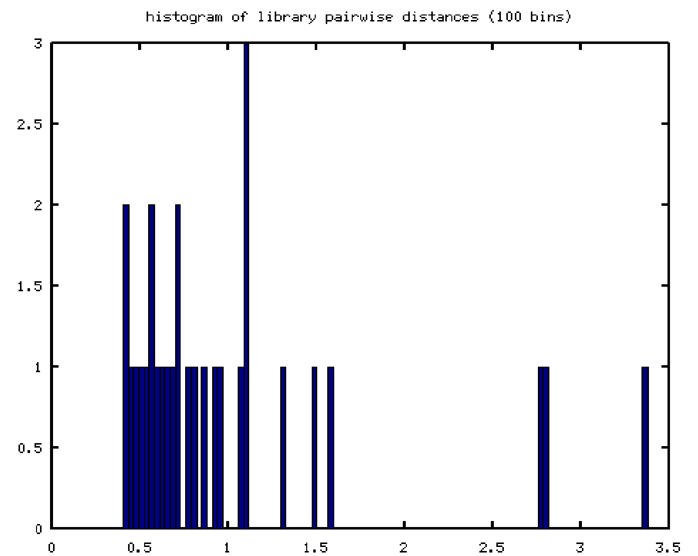
car_028.jpg

car_029.jpg

car_030.jpg

Histogram of the pairwise distances between images in our library:





Relevant source code files:

- Code/src/processing/CovarianceDetection.h
- Code/src/testbench/tb_CovarianceDetection.cpp
- Code/src/testbench/tb_CovarianceDetection_plots.m

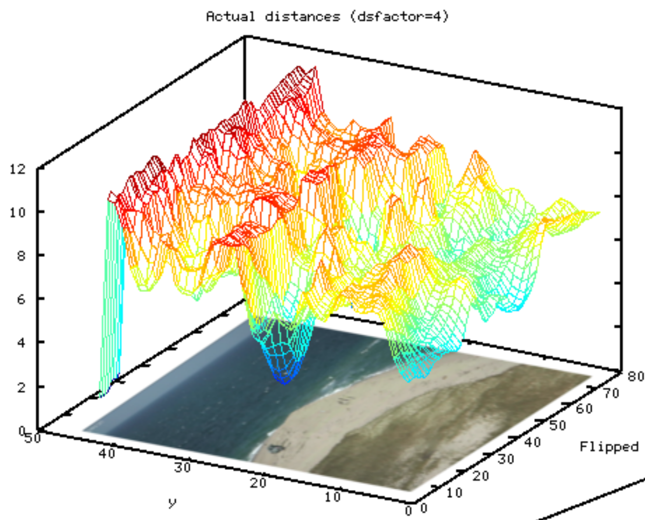
Feature vector used for all tests:

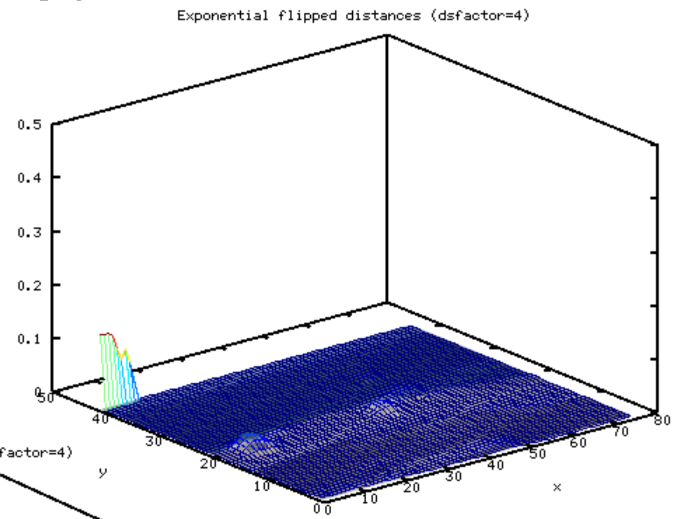
```
inline arma::mat feature_matrix(const FeatureImages &f, RROI roi)
{
  const int X = roi.x;
  const int Y = roi.y;
  const int W = roi.w;
  const int H = roi.h;
  arma::mat result = arma::zeros(W*H, 6);
  for (int j=0; j < H; j++)
  {
    for (int i=0; i < W; i++)
    {
     int index = j*W+i;
     result(index,0) = (float) i;
     result(index,1) = (float) j;
     result(index,2) = fabs(f.dx1->get(i+X, j+Y));
     result(index,3) = fabs(f.dy1->get(i+X, j+Y));
     result(index,4) = fabs(f.dx2->get(i+X, j+Y));
     result(index,5) = fabs(f.dy2->get(i+X, j+Y));
    }
  }
  return result;
} // feature_matrix()
```

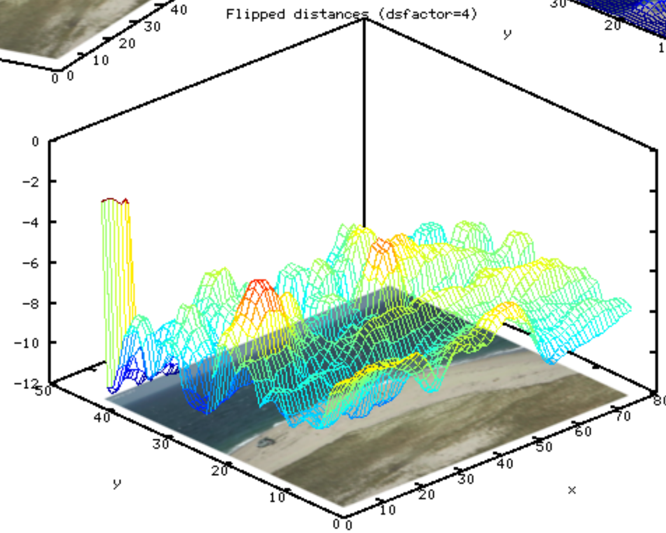
Parameters of the covariance window (a.k.a. "scan window" or "search window" or "query window") are given below. I chose the width and height to be that of each of the jeeps in *GP-2010-05-01_twojeeps.png*.

```
const int search_width  = 80;     // width of search window.
const int search_height = 50;     // height of search window.
const int step_x = 4;             // number of pixels to step in x direction
const int step_y = 4;             // number of pixels to step in x direction
```

GP-2010-04-23_silversubaru.png (1280x720)



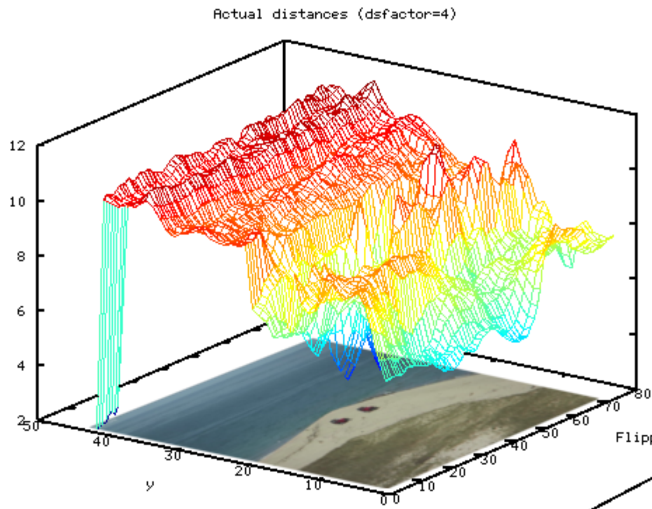


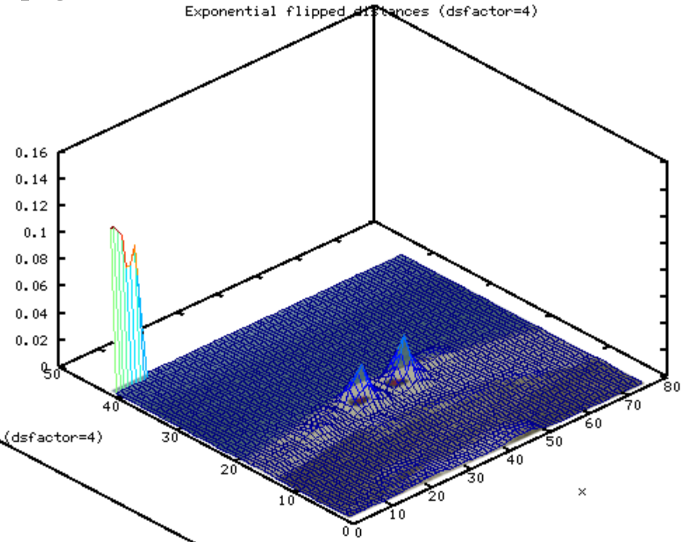


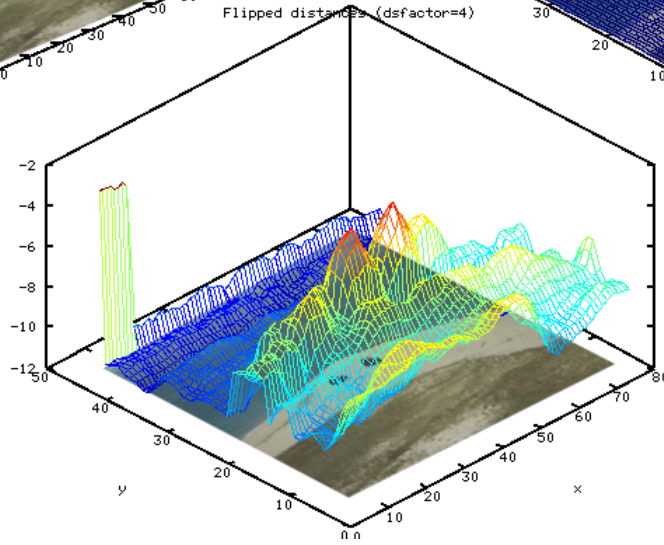
3.4 threshold
109 detections

threshold=3.8
203 detections

GP-2010-05-01_twojeeps.png (1280x720)







threshold = 3.4
118 detections

threshold = 3.8
196 detections

GP-2010-05-01_twopickups.png (1280x720)

Actual distances (dsfactor=4)

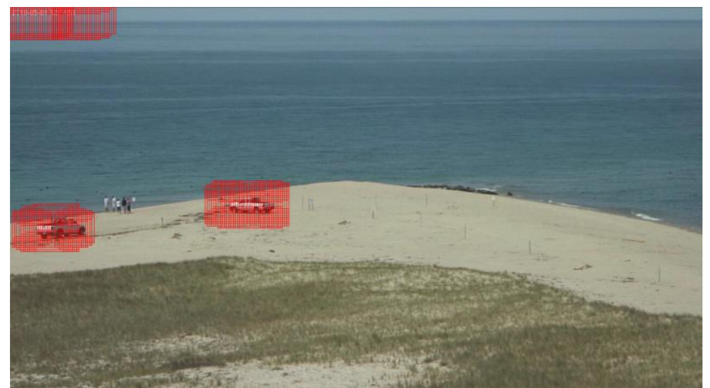
Exponential flipped distances (dsfactor=4)

Flipped distances (dsfactor=4)

Left:
threshold=3.2
129 detections

Below left:
threshold=3.4
230 detections

Below right
threshold=3.8
452 detections



Results of covariance matrix detection using frames of beach video and a library of boats

Written by D.Cullen
Last updated 2012-02-22

Goals of the following tests:

- Test to see how well the covariance matrix-based object detection works for boats.
- Determine whether it's better to have separate dictionaries for motorboats and sailboats, or if is okay to put them all in the same database. In other words, should motorboats and sailboats be treated as a separate class of objects? Which method yields the best detections?
- Does the direction of the boats in the dictionary image matter? In other words, if I have a dictionary of boats all facing to the left, with the system be able to detect boats that face to the right? Taking this one step further, if I augment my dictionary by adding flipped copies of all of the boats to the dictionary, does detection accuracy improve? Also take a look at the mathematics behind the covariance of flipped images. Is there any mathematical basis for any conclusions that I make?

We created a database of 80 images of motorboats and another database of 25 images of sailboats. We also combined these two sets of images to create a third database. All images were obtained from a Google Images search and cropping where appropriate. The resolution of the images variables, but it is roughly 200x100 on average.

Below are the thumbnails of all of the images in each database. Note that even though a square border is drawn around each of the thumbnail images, the actual sizes of the images have been cropped to tightly fit the boundaries of each image.

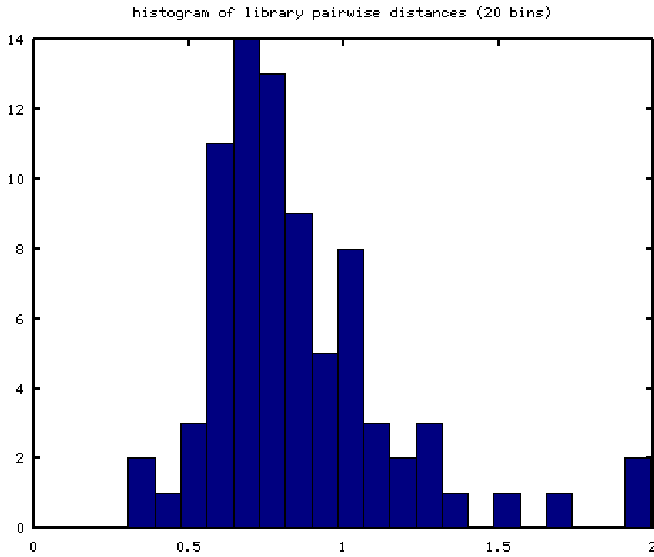
Here are the sailboats:

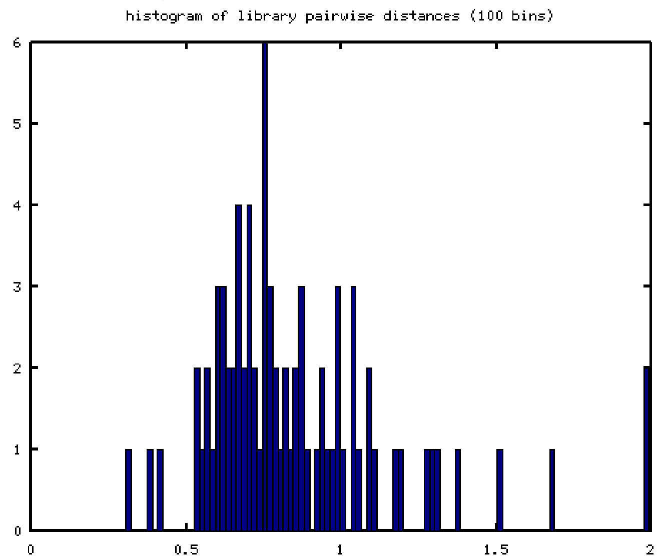
Here are the motorboats:

motor 001.jpg	motor 002.jpg	motor 003.jpg	motor 004.jpg	motor 005.jpg	motor 006.jpg	motor 007.jpg	motor 008.jpg
motor 009.jpg	motor 010.jpg	motor 011.jpg	motor 012.jpg	motor 013.jpg	motor 014.jpg	motor 015.jpg	motor 016.jpg
motor 017.jpg	motor 018.jpg	motor 019.jpg	motor 020.jpg	motor 021.jpg	motor 022.jpg	motor 023.jpg	motor 024.jpg
motor 025.jpg	motor 026.jpg	motor 027.jpg	motor 028.jpg	motor 029.jpg	motor 030.jpg	motor 031.jpg	motor 032.jpg
motor 033.jpg	motor 034.jpg	motor 035.jpg	motor 036.jpg	motor 037.jpg	motor 038.jpg	motor 039.jpg	motor 040.jpg
motor 041.jpg	motor 042.jpg	motor 043.jpg	motor 044.jpg	motor 045.jpg	motor 046.jpg	motor 047.jpg	motor 048.jpg
motor 049.jpg	motor 050.jpg	motor 051.jpg	motor 052.jpg	motor 053.jpg	motor 054.jpg	motor 055.jpg	motor 056.jpg
motor 057.jpg	motor 058.jpg	motor 059.jpg	motor 060.jpg	motor 061.jpg	motor 062.jpg	motor 063.jpg	motor 064.jpg
motor 065.jpg	motor 066.jpg	motor 067.jpg	motor 068.jpg	motor 069.jpg	motor 070.jpg	motor 071.jpg	motor 072.jpg
motor 073.jpg	motor 074.jpg	motor 075.jpg	motor 076.jpg	motor 077.jpg	motor 078.jpg	motor 079.jpg	motor 080.jpg

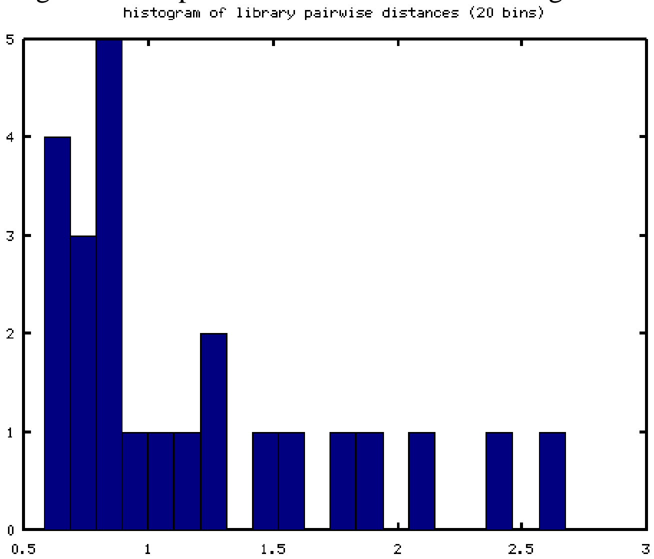
The combined library is just the union of the above two libraries.

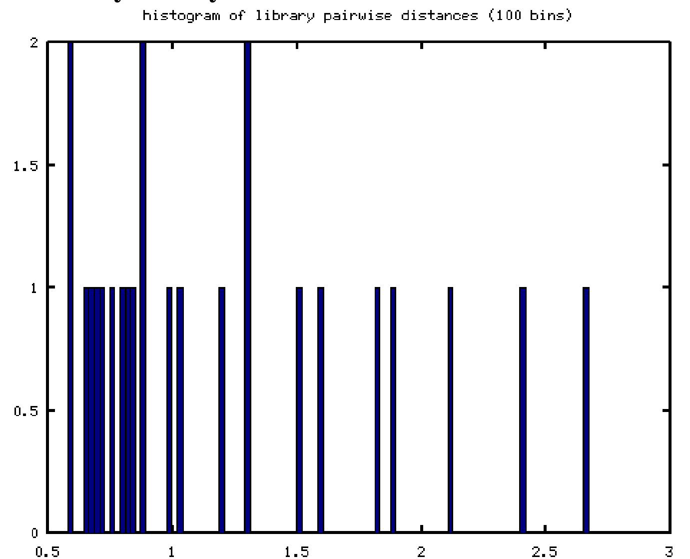
Histogram (with two different bin sizes) of the pairwise distances between images in our **motorboats-only library:**



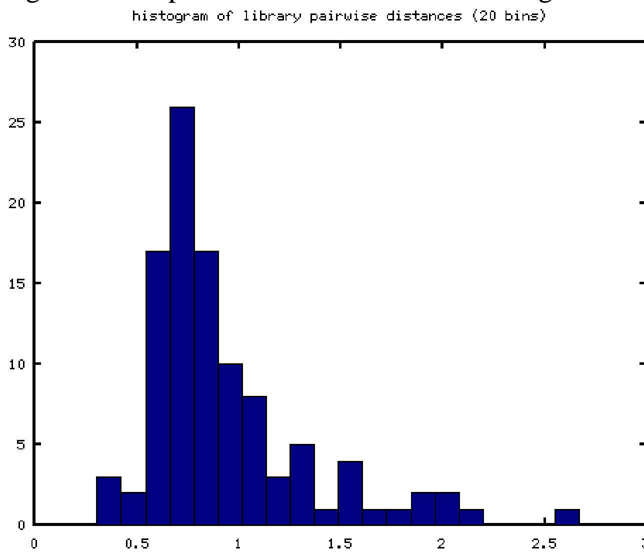


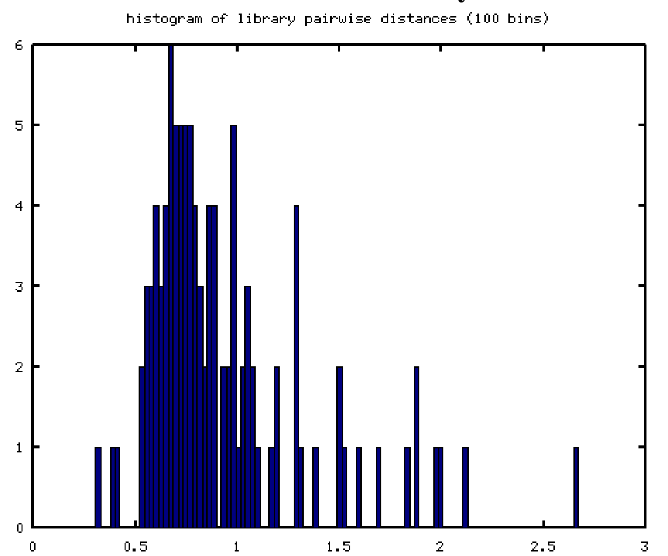
Histogram of the pairwise distances between images in our **sailboats-only library:**





Histogram of the pairwise distances between images in our **combined motorboats and sailboats library:**





Relevant source code files:

- Code/src/processing/CovarianceDetection.h
- Code/src/testbench/tb_CovarianceDetection.cpp
- Code/src/testbench/tb_CovarianceDetection.m

Feature vector used for all tests:

This is the same feature vector used in [Porikli, May 2006]. Note that *i* and *j* are relative to the region of interest.

I used the following parameters for the search window in all of my tests:

```
  const int search_width  = 100;   // width of search window.
  const int search_height = 50;    // height of search window.
  const int step_x = 8;            // number of pixels to step in x direction
  const int step_y = 8;            // number of pixels to step in x direction
```

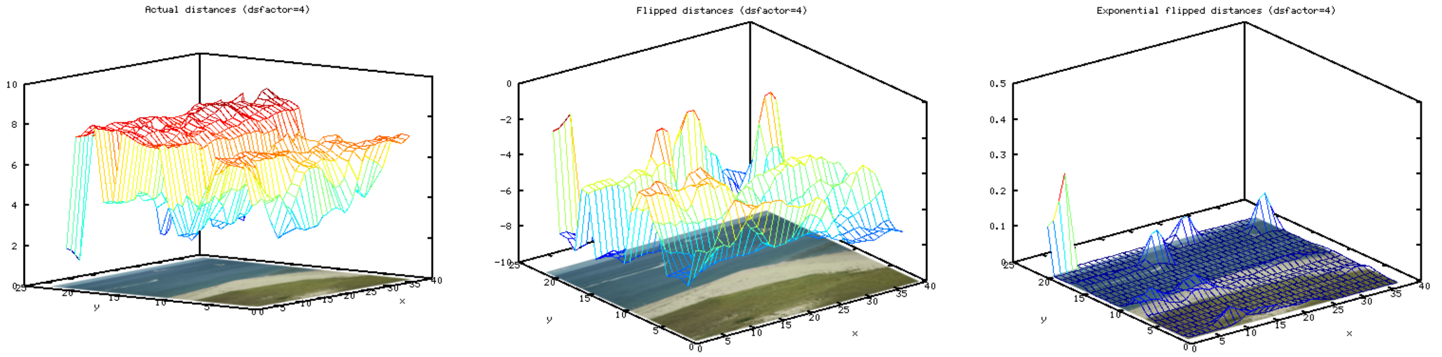
I selected these dimensions by looking at the data and choosing a reasonable size for the bounding box of a boat.
Note that I step by 8 pixels in each direction, not 4 pixels or anything smaller, so that the tests won't take as long to run.

In "tb_CovarianceDetection.cpp", I used the following "debug" lines to ignore the datestamp in the top-left corner:

```
    // Perform detections. Also store distances for debugging purposes.
    std::vector< std::pair<int,int> > detect_coords;
    std::vector<DistanceSample> distance_samples;
    for (int j=0; j < gray->height()-search_height; j+=step_y)
    {
      for (int i=0; i < gray->width()-search_width; i+=step_x)
      {
       RROI search_roi(i, j, search_width, search_height);
       arma::mat search_logcov = get_log_cov_matrix(feature_images, search_roi);
       float min_dist = cd.min_distance_to_library(search_logcov);
       // DEBUG  ----------------------------------------------------------
       // IGNORE THE DATESTAMP IN THE TOPLEFT CORNER BY FORCING DISTANCE TO A CONSTANT VALUE.
       if (j <= 20 && i < 125)
         min_dist = 10;
       // END DEBUG ------------------------------------------------------
       distance_samples.push_back(DistanceSample(i,j,min_dist));
       if (min_dist <= cd.get_threshold())
         detect_coords.push_back(std::pair<int,int>(i,j));
      } // for ...
      printf("  Just finished row %d\n", j);  // Debug print lets you know how much time is remaining.
    } // for ...
```

GP-2010-08-02_boats008.png with motorboats library

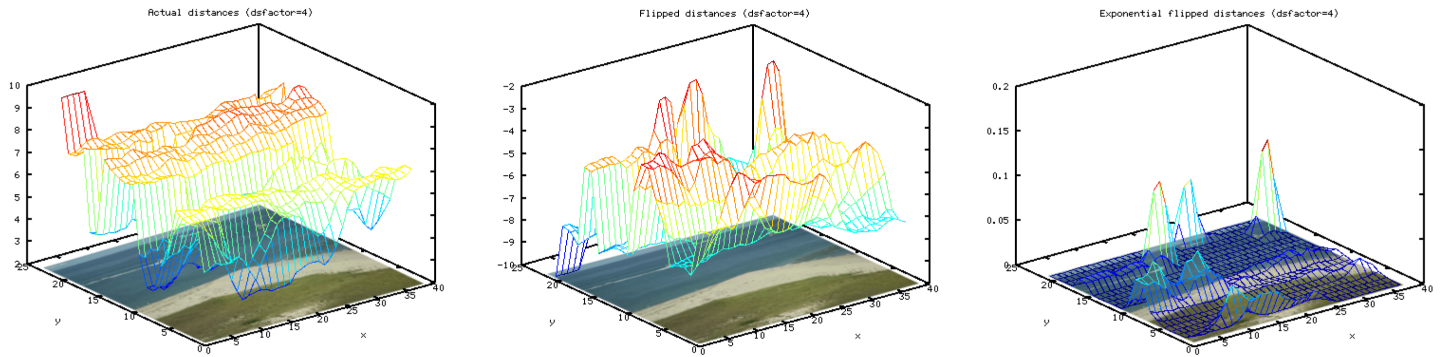
Datestamp kept:



threshold=2.6; 96 detections

threshold=2.7, 132 detections

Datestamp ignored:



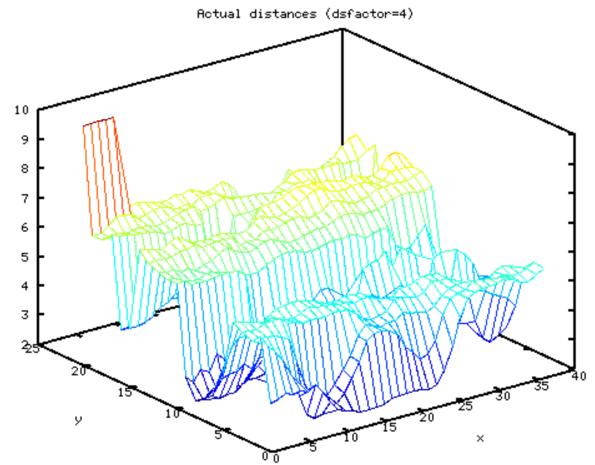
Why is there a high peak where the date stamp is on the actual (non-flipped) plot above? It's because I set it to an arbitrarily and sufficiently large, constant distance to prevent anything from being detected here.

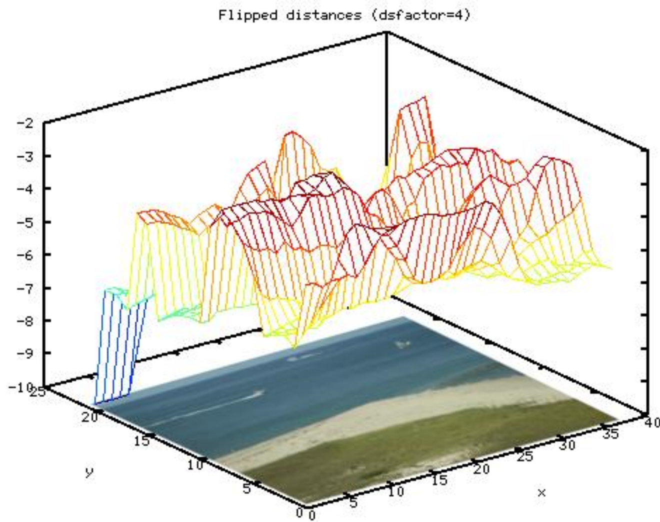
threshold=2.6; 67 detections

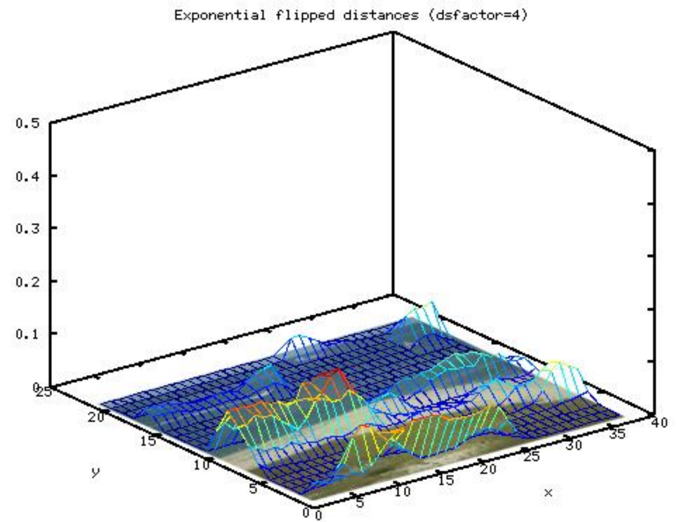
threshold=2.7; 102 detections

GP-2010-08-02_boats008.png with sailboats library

threshold=2.7; 939 detections



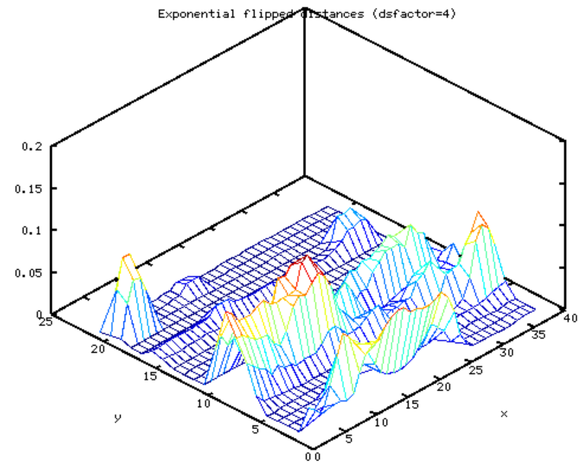




Notice that only one of the boats was detected, and barely so.

GP-2010-08-02_boats005.png with sailboats library

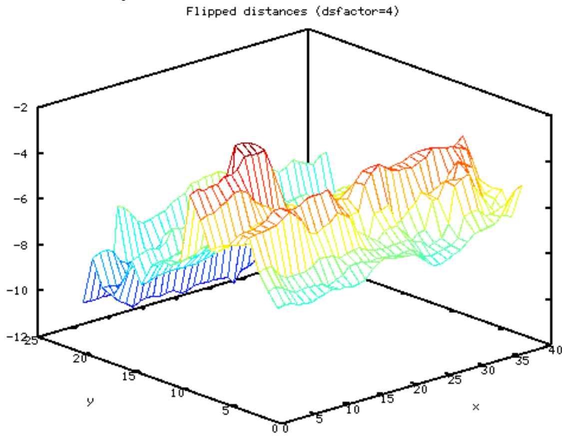
Next we tried the sailboats library on an image with an actual sailboat, although the sails were lowered. It does better than the previous image, although there are still many false detections in the grass.



threshold=2.4; 326 detections

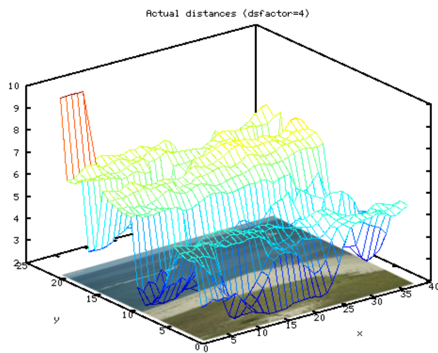
GP-2011-10-23_boats001.png with sailboats library

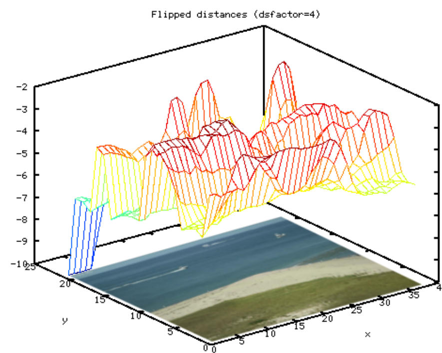
Very few sailboats travel near Great Point, so I had difficulty finding any frames of sailboats. The following image has a boat with a tall conning tower and antennas, which gives it a triangular shape similar to that of a sailboat's sails, and thus it performs very well, as shown below.

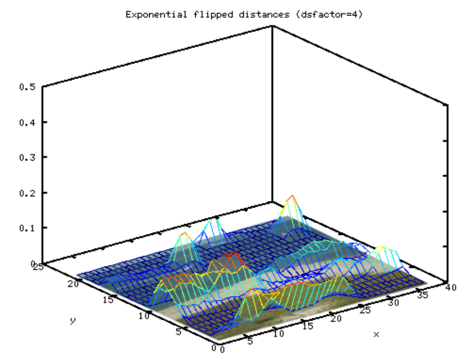


2.4 threshold; 20 detections

GP-2010-08-02_boats008.png with combined library







threshold=2.6; 819 detections

threshold=2.7; 1034 detections

Notice that the combined library performs much worse than the motorboats-only library. Prior to running these tests, I thought that perhaps adding more diversity to the database would be good because it might make the detections more robust for handling a larger variety of different boats. However, it appears that the contrary is true; adding more diversity to the database essentially just adds noise and makes the system less discriminative. Therefore, we recommend treating motorboats and sailboats as different classes of objects, with separate databases for each.

Now we will perform some experiments to determine how much the direction of the boats in the database matters.

Here is how I prepared these tests:

1. Copied all of the images from the original *library_motorboats* database to a new folder called *library_motorboats_all_face_left*. I manually went through the database and made all of the right-facing images face to the left using the *mogrify* command. For example:

   ```
   mogrify -flop "motor 052.jpg"
   ```

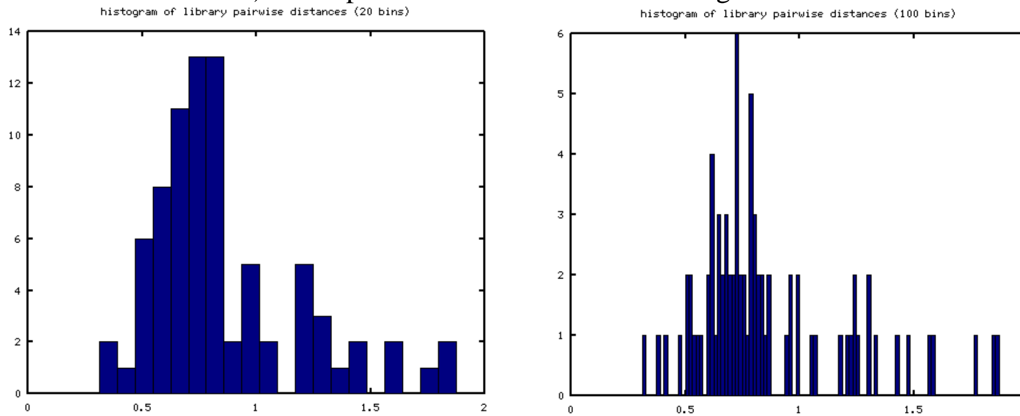
   (I used the “flop” option to mirror horizontally; the “flip” option would have meant to mirror vertically.)

   The result is a database of 80 images of boats that all face to the left.
2. I copied all of these left-facing images from *library_motorboats_all_face_left* into a new folder called *library_motorboats_all_face_right*. Than I ran the following command to flip all the images:

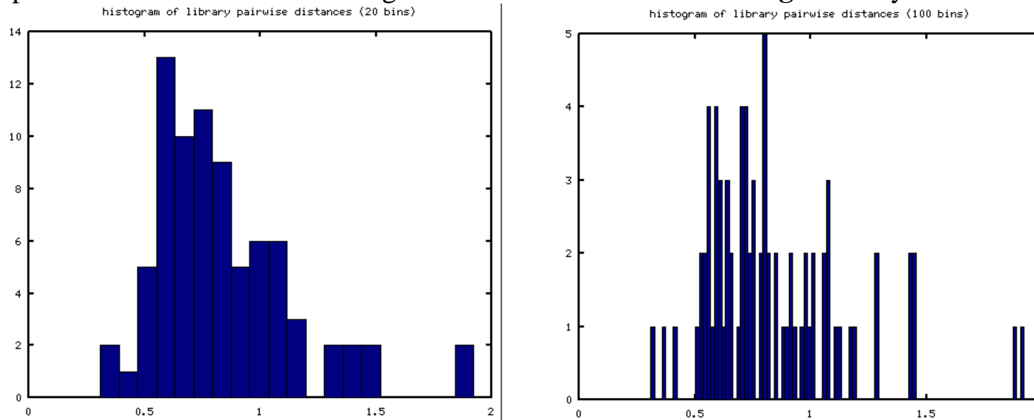
   ```
   mogrify -flop *
   ```

   The result is a database of 80 images of boats that all face to the right.
3. Next, I copied all of the images from *library_motorboats_all_face_left* and *library_motorboats_all_face_right* into a new folder called *library_motorboats_face_both_ways*. The result is a database of 160 images of boats, in which there are 80 different boats and 80 mirrored copies of those boats.

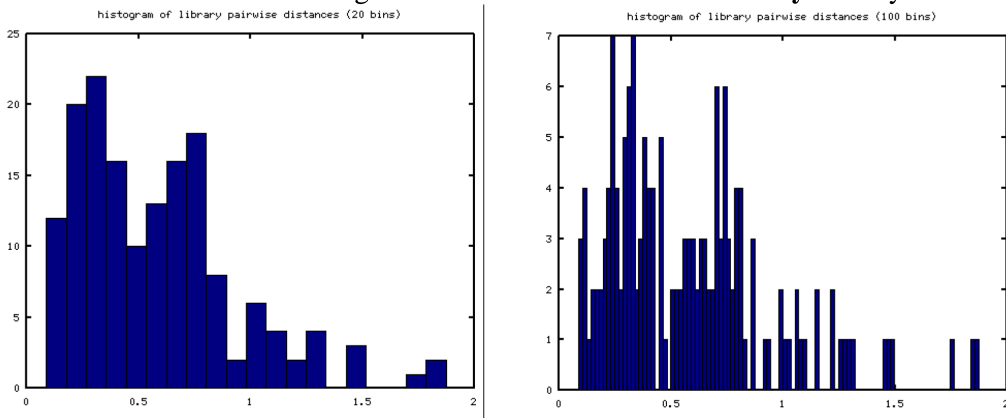
Histogram (with two different bin sizes) of the pairwise distances between images in our **motorboats all-face-left** library:



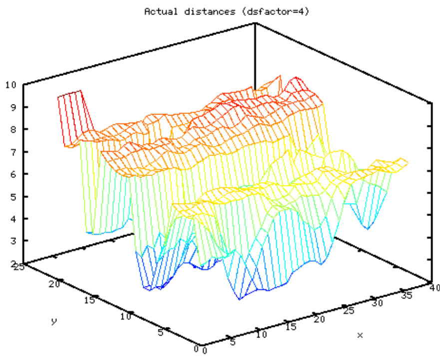
Histogram of the pairwise distances between images in our **motorboats all-face-right** library:

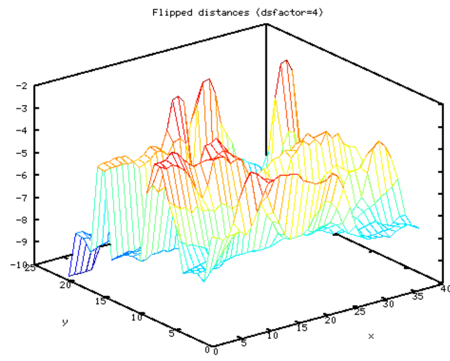


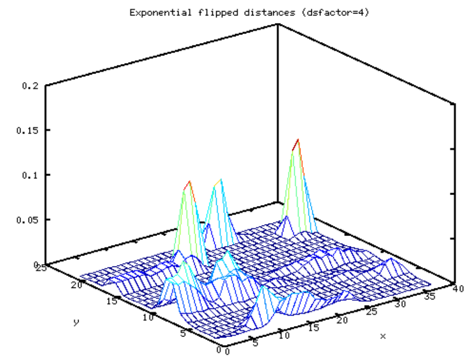
Histogram of the pairwise distances between images in our **motorboats face-both-ways** library:



GP-2010-08-02_boats008.png with library_motorboats_all_face_left library



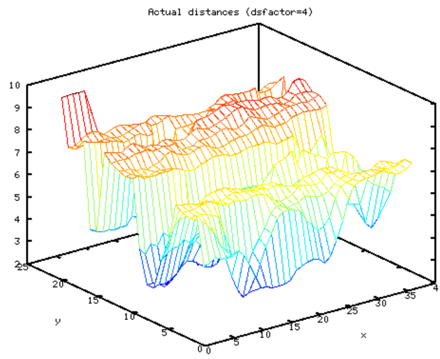


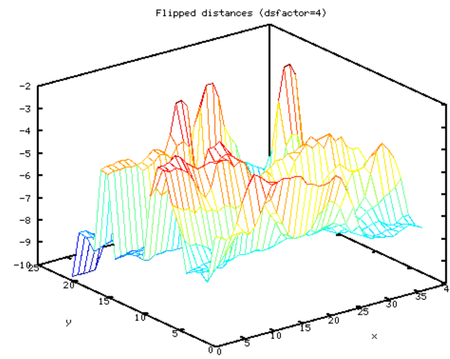


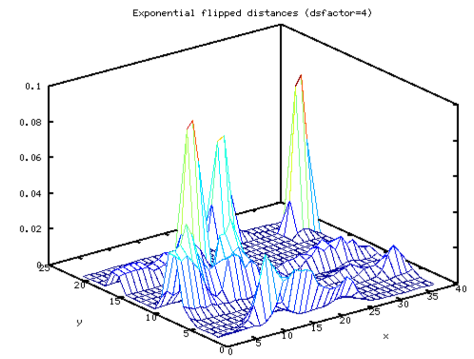
threshold=2.6; 67 detections

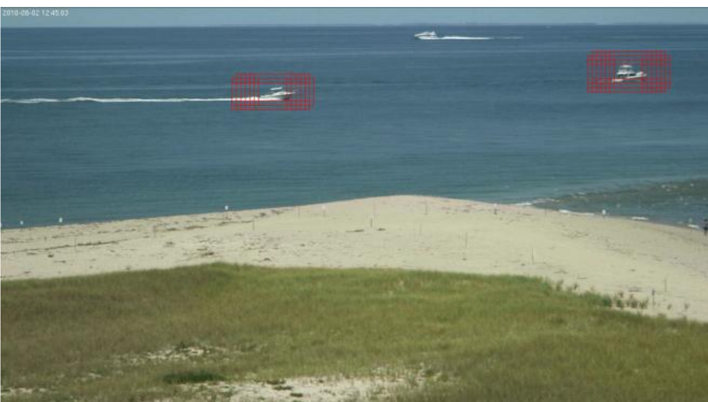
threshold=2.7; 102 detections

GP-2010-08-02_boats008.png with library_motorboats_all_face_right library

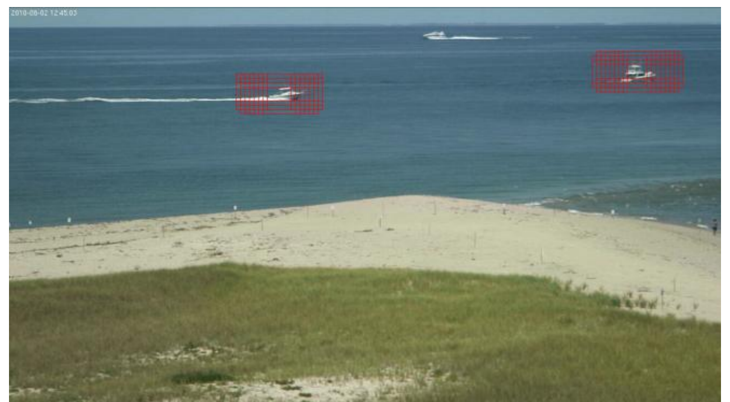






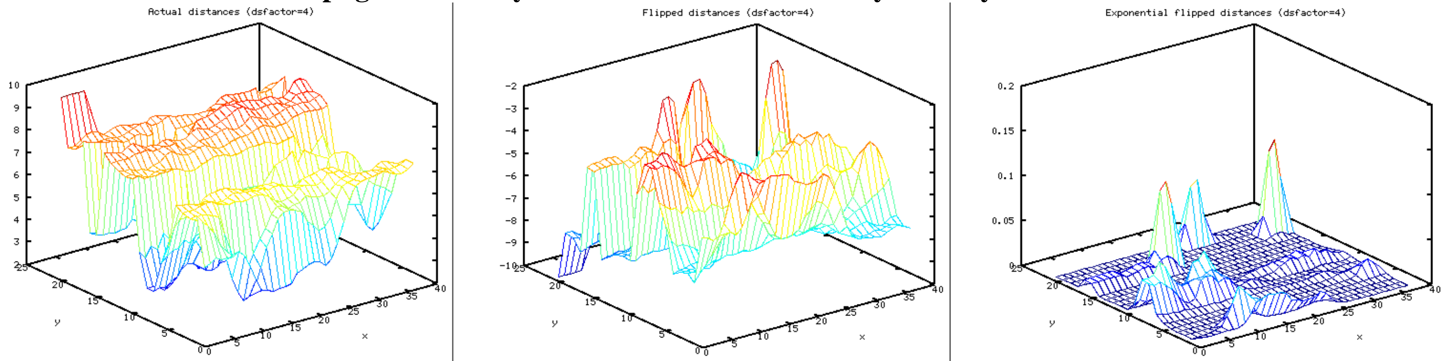


threshold=2.6; 43 detections



hreshold=2.7; 62 detections

GP-2010-08-02_boats008.png with library_motorboats_face_both_ways library



threshold=2.6; 67 detections

threshold=2.7; 102 detections

The results from the above tests suggest that the direction of the motorboats does not matter much. This could be caused by any of the following:

1. Motorboats, especially at far distance, look somewhat similar regardless of whether they face right or left.
2. Does the way the covariance matrix is computed make the direction not matter much? This is still TBD.

The “all face left” library did slightly better than the “all face right” library, but this was probably just due to random chance for this particular image; the boats were oriented in just such a way that favored the “all face left” library. The “face both ways” library performs just as well as the “all face left” library. Interestingly, if we look back to our first set of results for this image on page 5, in which our database consisted of just 80 images of boats, some facing left and some facing right, we see that this had performance equivalent to our “face both ways” library.

We conclude that when creating a database, it is sufficient to just select half of the images to be facing to the right and half of the images to be facing to the left, just to give sufficient diversity, but that ultimately, the direction does not matter a whole lot. It is not really worth the trouble to duplicate all of the images and flip the duplicates, so that there are left-facing and right-facing versions of each image; this just makes the database twice as big for marginal benefit.

It would be interesting to study the mathematics of the covariance matrices for the original and flipped version of the image, in order to try to better understand why the direction does not matter much, but we’ll save this analysis and discussion for the final project report. Another test that would be interesting to do would be to select another class of objects that doesn’t have the approximate left-to-right symmetry that the boats exhibit, and then run more experiments to see if a library of images facing all one way or the other makes a significant difference.

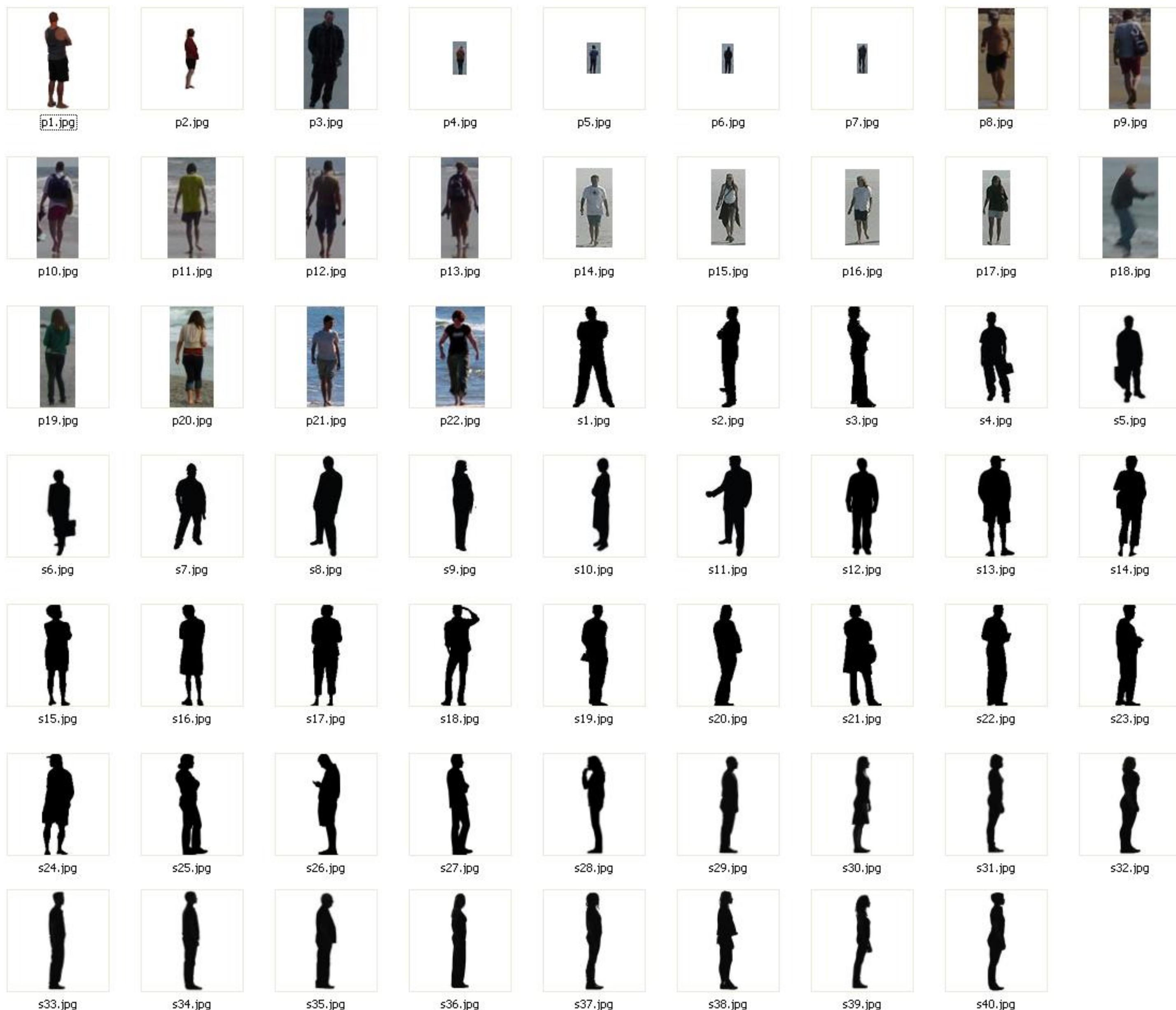
Results of covariance matrix detection using frames of beach video and a library of people

Written by D.Cullen

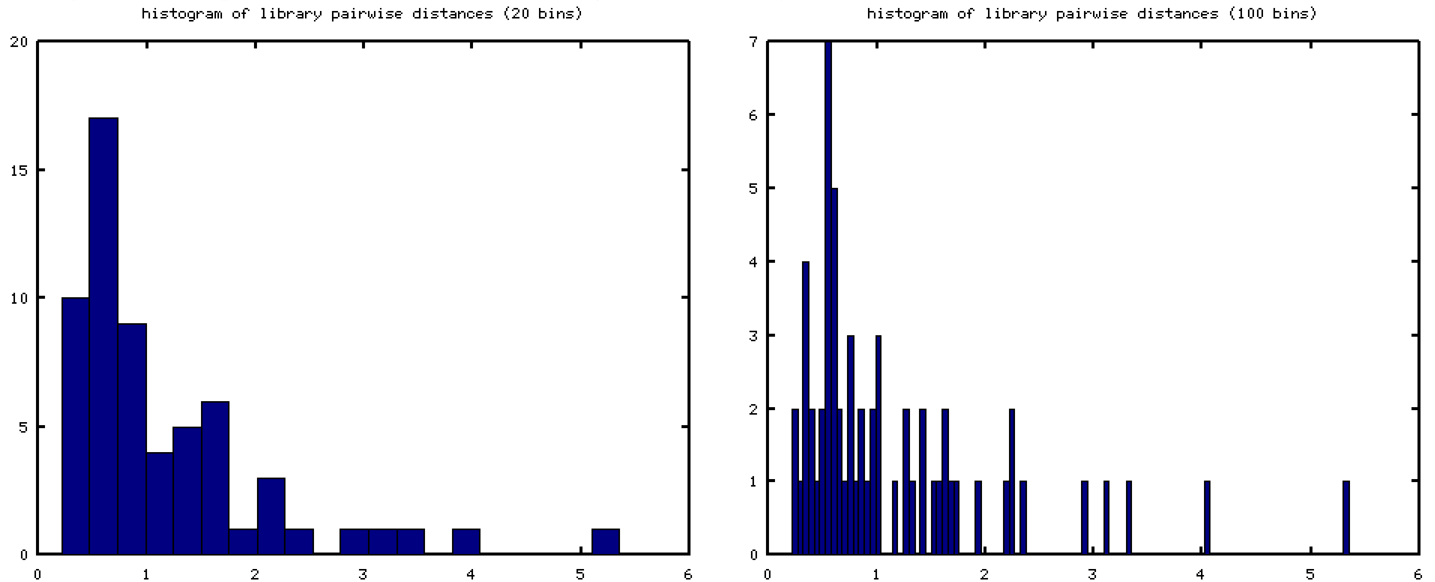
Last updated 2012-02-07

We created a database of 62 images of the silhouettes of people standing using a Google Images search and cropping where appropriate. The resolution of images varies, but it is roughly 50x200 on average.

Below are thumbnails of all the images in our library. Note that even though a square border is drawn around each of the thumbnail images, the actual size of the image has been cropped to tightly fit the boundary of each person.



Histograms of the pairwise distances between images in our library:



Relevant source code files:

- Code/src/processing/CovarianceDetection.h
- Code/src/testbench/tb_CovarianceDetection.cpp
- Code/src/testbench/tb_CovarianceDetection_plots.m

Feature vector used for all tests:

This is the same feature vector used in [Porikli, May 2006]. Note that *i* and *j* are relative to the region of interest.

I used the following parameters for the search window in all of my tests:

```
const int search_width  = 20;     // width of search window.
const int search_height = 40;     // height of search window.
const int step_x = 4;             // number of pixels to step in x direction
const int step_y = 4;             // number of pixels to step in x direction
```

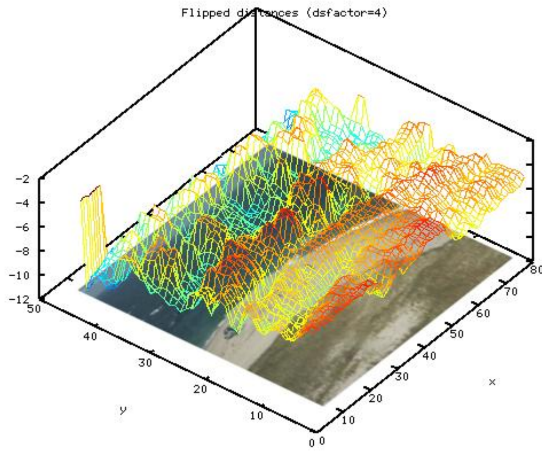
(I selected these dimensions by looking at the data and choosing a reasonable size for the bounding box of a person.)

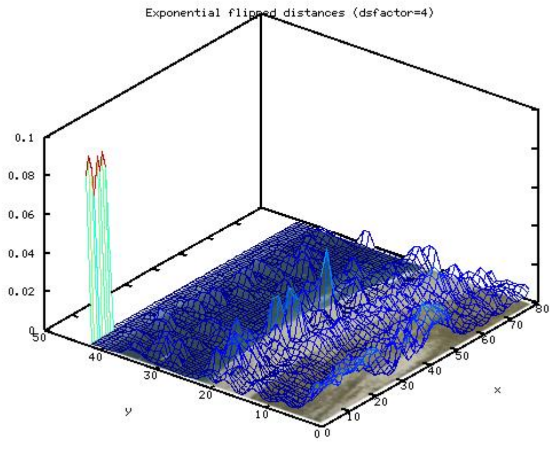
In "tb_CovarianceDetection.cpp", I used the following "debug" lines to ignore the datestamp in the top-left corner:

```
// Perform detections. Also store distances for debugging purposes.
std::vector< std::pair<int,int> > detect_coords;
std::vector<DistanceSample> distance_samples;
for (int j=0; j < gray->height()-search_height; j+=step_y)
{
  for (int i=0; i < gray->width()-search_width; i+=step_x)
  {
   RROI search_roi(i, j, search_width, search_height);
   arma::mat search_logcov = get_log_cov_matrix(feature_images, search_roi);
   float min_dist = cd.min_distance_to_library(search_logcov);
   // DEBUG  ---------------------------------------------------------
   // IGNORE THE DATESTAMP IN THE TOPLEFT CORNER BY FORCING DISTANCE TO A CONSTANT VALUE.
   if (j <= 20 && i < 125)
     min_dist = 10;
   // END DEBUG ------------------------------------------------------
   distance_samples.push_back(DistanceSample(i,j,min_dist));
   if (min_dist <= cd.get_threshold())
     detect_coords.push_back(std::pair<int,int>(i,j));
  } // for ...
  printf("  Just finished row %d\n", j);  // Debug print lets you know how much time is remaining.
} // for ...
```

GP-2010-04-23_people002.png

Datestamp kept:

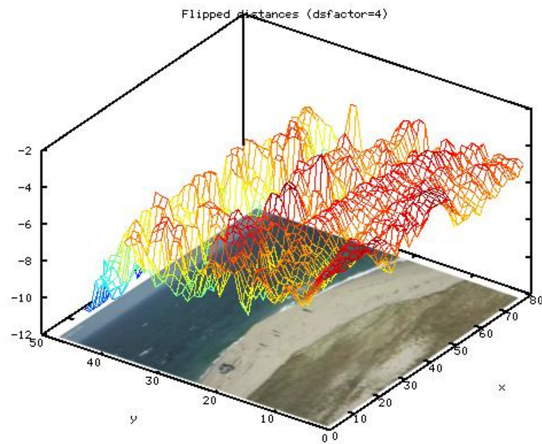


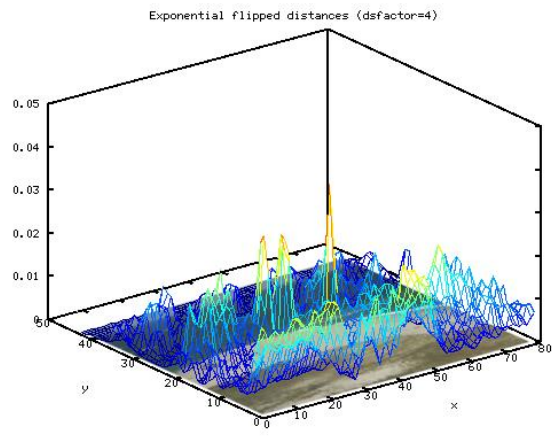


threshold=3.4; 123 detections

threshold=3.6; 154 detections

Datestamp ignored:

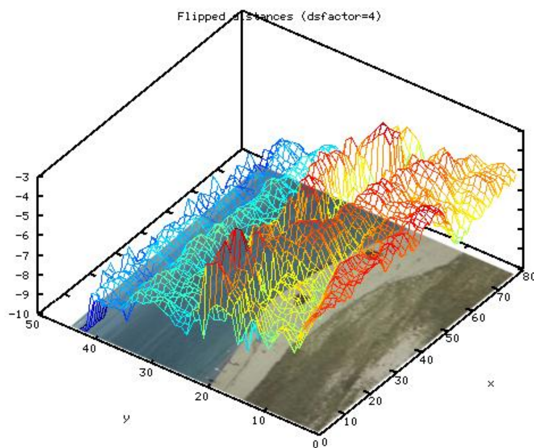


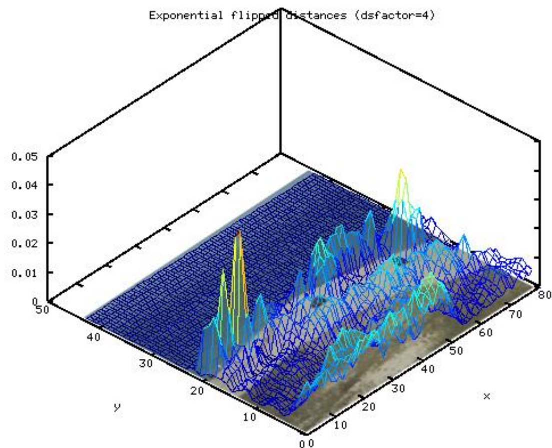


threshold=3.6; 35 detections

threshold=3.8; 163 detections

GP-2010-05-01_people003.png

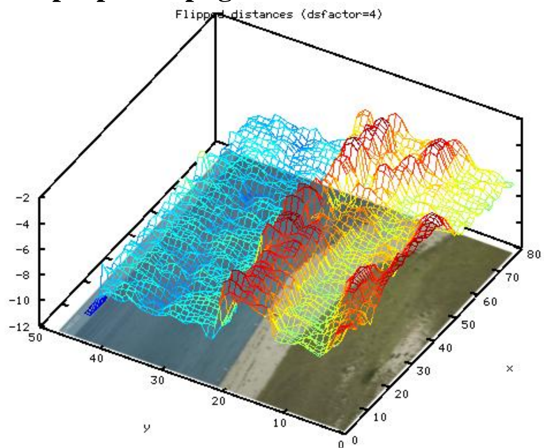


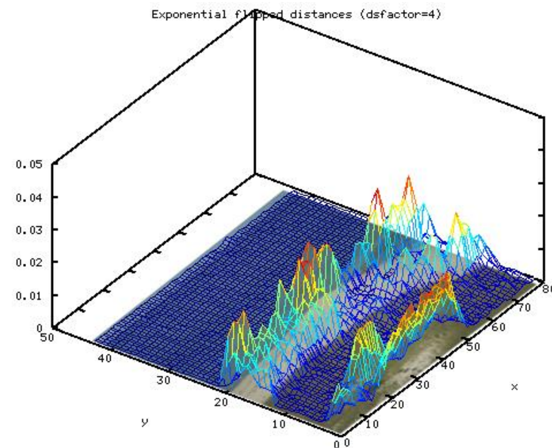


threshold=3.4, 45 detections (threshold=3.6, 67 detections looks similar)

threshold=3.8, 181 detections

GP-2010-08-02_people001.png





threshold=3.6, 2 detections

threshold=3.8, 98 detections

D AVSS2012 Draft Paper

This is a preliminary draft submitted on 2012-03-21 to the *9th IEEE Advanced Audio and Signal-Based Surveillance* conference.

000 001 002 003 004 005 006 007 008 009 010 011 012 013 014 015 016 017 018 019 020 021 022 023 024 025 026 027 028 029 030 031 032 033 034 035 036 037 038 039 040 041 042 043 044 045 046 047 048 049 050 051 052 053

Detection and Summarization of Salient Events in Coastal Environments

Anonymous AVSS submission for Double Blind Review

Paper ID 183

Abstract

The monitoring of coastal environments is of great interest to biologists and environmental protection organizations with video cameras being the dominant sensing modality. However, it is recognized that video analysis of maritime scenes is very challenging on account of background animation (water reflections, waves) and very large field of view. We propose a practical approach to the detection of three salient events, namely boats, motor vehicles and people appearing close to the shoreline, and their subsequent summarization. Our approach consists of three fundamental steps: region-of-interest (ROI) localization by means of behavior subtraction, ROI validation by means of feature-covariance-based object recognition, and event summarization by means of video condensation. The goal is to distill hours of video data down to a few short segments containing only salient events, thus allowing human operators to expeditiously study a coastal scene. We demonstrate the effectiveness of our approach on long videos taken at Great Point, Nantucket, Massachusetts.

1. Introduction

Technological improvements of the last decade have made digital cameras ubiquitous. They have become physically smaller, more power-efficient, wirelessly-networked, and, very importantly, less expensive. For these reasons, cameras are finding increasing use in new surveillance applications, outside of the traditional public safety domain, such as in monitoring coastal environments [5]. At the same time, many organizations are interested in leveraging video surveillance to learn about the wildlife, land erosion, impact of humans on the environment, etc. For example, biologists interested in marine mammal protection would like to know whether humans have come too close to seals on a beach. US Fish and Wildlife Service would like to know how many people and cars have been on the beach each day, and whether they have disturbed the fragile sand dunes. These are just two of the many uses of coastal video data.

However, with 100+ hours of video recorded by each camera per week, a search for salient events by human operators is not sustainable. Therefore, the goal of our research is to develop an integrated approach to analyze the video data and distill hours of video down to a few short segments of only the salient events. In particular, we are interested in identifying the presence of boats, motor vehicles and people in close proximity to the shoreline. This choice of objects of interest is dictated by our specific application but the approach we propose is general and can be applied in other scenarios as well.

Figure 1. Example of a detected boat and truck (top row) and a summary frame (bottom row) that shows both objects together.

Although to date many algorithms have been proposed for moving object localization and recognition, very few approaches deal explicitly with the marine environment [13, 10]. Among the many approaches to video summarization we note key-frame extraction, montage and synopsis. While key-frame extraction [7], by design, loses the dynamics of the original video, video montage [3] can result in loss of context as objects are displaced both in time *and* space. Video synopsis [8] does not suffer from the loss of context but is conceptually-complex and computationally-intensive.

There are three challenges to the detection and summarization of events in coastal videos. First, video analysis of maritime scenes poses difficulties because of water animation in the background, dune grasses blowing in the wind in the foreground, and the vast field of view. Secondly, a reliable recognition of objects from large distances (small scale) and especially of boats at sea is highly non-

054 055 056 057 058 059 060 061 062 063 064 065 066 067 068 069 070 071 072 073 074 075 076 077 078 079 080 081 082 083 084 085 086 087 088 089 090 091 092 093 094 095 096 097 098 099 100 101 102 103 104 105 106 107

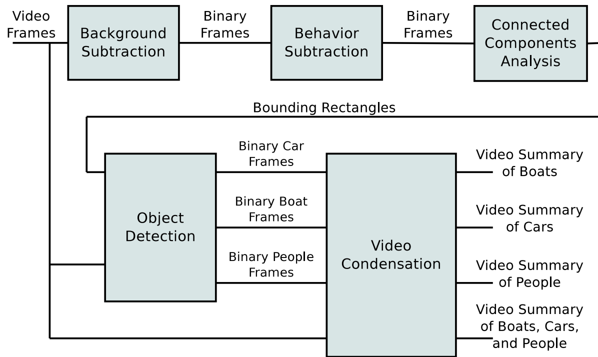


Figure 2. Block diagram of the entire system.

trivial. Thirdly, it is not obvious how to effectively summarize salient events for viewing by human operators.

We address these three challenges as follows. We rely on the movement of boats, motor vehicles and people for their detection. While difficult at small scales, motion detection is particularly challenging for boats at sea due to waves. We attack this problem by recognizing the fact that over longer time scales water movement can be considered stationary while boat movement is locally non-stationary both spatially and in time. In this context, we leverage the so-called *behavior subtraction* that has been shown to effectively deal with animated water while accurately detecting boat movement [9]. The obtained moving-object masks serve as regions of interest (ROIs) for subsequent boat, motor vehicle and people detection. We apply the *feature-covariance framework* [11, 12] to reliably distinguish between different objects even at small scales. Having detected and classified salient events, the last step is their summarization. We adapt the so-called *video condensation* [4] to our needs by devising a suitable cost needed by the algorithm based on the outcome of behavior subtraction and object detection.

Our paper focuses on one particular coastal environment, namely Great Point, Nantucket, MA. We have collected hundreds of hours of video data from networked cameras located close to the beach (Figure 1). We have demonstrated the effectiveness of our algorithms on this data set and obtained condensation ratios of up to almost 20:1 thus dramatically reducing a human operator involvement in search for events of interest.

2. Proposed Approach

The goal of our system is to automatically distill hours of video down to a few short segments containing only the types of objects sought. Our system must be simple enough to be used by operators with no expertise in image processing or video surveillance. Furthermore, we want a system that can be easily extended to detect new types of objects. The final summarization must be natural and useful to biologists, environmental protection agents, etc.

Our system, whose block diagram is shown in Figure 2, meets these requirements. Each of the processing blocks requires only a few parameters that are easy to tune; a fairly wide range of values will produce reasonable results. To extend the system to detect new types of objects, the user needs only to provide new images similar to the target class of objects of interest. Finally, any combination of classes of objects can be selected for summarization, which gives the operator a lot of control over the output.

We assume that input video is acquired by a camera with fixed position, orientation, and zoom, that no camera shake or vibration takes place, that there is no moisture on the lens (from rain or fog), and that the lighting is constant (i.e., no sudden changes in cloud cover or in camera gain). We plan to address some of these limitations in the near future.

Our system first runs background subtraction to detect the ROI (moving) areas in the video, followed by behavior subtraction to reduce the amount of uninteresting activity such as ocean waves. Next, the system runs a connected-component analysis on the behavior-subtracted (binary) video to label the regions where moving objects are likely. Then, the corresponding regions in video frames are tested for the presence of objects of interest using a feature-covariance classification. The regions classified as either boats, cars or people are then used as high-cost areas that cannot be removed in the subsequent video condensation.

The following sections describe each step of the overall system. However, for the sake of brevity, many details have been omitted, so we refer the reader to the literature.

2.1. Background Subtraction

We detect moving objects in the camera field of view by means of background subtraction. Many background subtraction algorithms have been developed to date [2] with more advanced ones requiring significant computational resources. Since our system needs to process many hours of video, it is critical that background subtraction be as efficient as possible. Therefore, we use a very simple and computationally-efficient background model in combination with a Markov model for the detected labels L [6]. First, an exponentially-smoothed moving average is used to estimate background B^k by means of recursive filtering:

$$B^k[\boldsymbol{x}] = (1-\alpha) * B^{k-1}[\boldsymbol{x}] + \alpha * I^k[\boldsymbol{x}] \quad (1)$$

where I^k is the input video frame, k is the frame number, $\boldsymbol{x} = [x\ y]^T$ are pixel coordinates, and α is a smoothing parameter. The computed background B^k is then used in the following hypothesis test on the current video frame:

$$|I^k[\boldsymbol{x}] - B^{k-1}[\boldsymbol{x}]| \underset{\mathcal{S}}{\overset{\mathcal{M}}{\gtrless}} \theta \exp\left((Q_{\mathcal{S}}[\boldsymbol{x}] - Q_{\mathcal{M}}[\boldsymbol{x}])/\gamma\right) \quad (2)$$

where $Q_{\mathcal{S}}$ and $Q_{\mathcal{M}}$ denote the number of static ($\mathcal{S}$) and moving ($\mathcal{M}$) neighbors of $\boldsymbol{x}$, respectively, θ is a thresh-

old and γ is a tuning parameter that adjusts the impact of Markov model. Note that the overall threshold in the above hypothesis test depends on the static/moving neighbors. If there are more moving than static neighbors, then the overall threshold is reduced thus encouraging an $\mathcal{M}$ label at $\boldsymbol{x}$. The above test produces moving/static labels L^k for each video frame I^k. In our experiments, we have used 2nd order Markov neighborhood (8 nearest pixels). The second row in Figure 4 shows typical results for the above algorithm.

2.2. Behavior Subtraction

In our coastline videos, there is a great deal of repetitive background motion, such as ocean waves and dune grass, that results in spurious detections after background subtraction. However, this motion can be considered stationary (in a stochastic sense), when observed over a longer period of time, and suitably characterized. One method to accomplish this is the so-called behavior subtraction [9] that we leverage here due to its simplicity and computational efficiency. The algorithm operates on binary frames of labels L produced by background subtraction (Figure 2) and has two phases: training and testing. In the training phase, a segment of N frames from an M-frame training sequence ($M \geq N$) is being examined. The training sequence is assumed to exhibit the stationary dynamics we want to remove, but no “interesting” moving objects. First, labels L at each pixel are accumulated across all frames of the N-length segment to form a 2D array. Then, all such 2D arrays (for all N-frame segments within M-frame training sequence) are compared to select the maximum at each pixel. The resulting 2D training array constitutes the description of stationary scene dynamics. In the testing step, a sliding N-frame segment is used to accumulate labels L, from the background-subtracted sequence being processed, in a new 2D test array. If a pixel in the test array exceeds the value of the same pixel in the training array (maximum across many segments) by a threshold of Θ, then a detection is declared (white pixel). The rationale behind this is that an “interesting” object will occupy those pixels for more frames than the maximum number of frames occupied by stationary behavior, thereby allowing us to remove regions with “uninteresting” motion, such as waves. Typical results of behavior subtraction are shown in the second row of Figure 4.

2.3. Region of Interest (ROI) Extraction

We use a simple connected-components algorithm to label regions in the behavior-subtracted video. Each such region potentially includes a moving object. Since the feature-covariance classification operates on rectangular regions of pixels for efficiency, we circumscribe an axis-aligned bounding box around each of the connected components. We discard any bounding boxes smaller than a given threshold (5×5 in our experiments) as too small to contain any of the target objects. We also increase the size of each bounding box by a constant scale factor (20% in our experiments) to ensure that it completely captures the target object. This margin is important because the connected components may be misaligned with the underlying video objects and we risk losing part of the object causing subsequent mis-detection. The resulting bounding boxes are passed to the object detection step.

2.4. Object Detection Using Feature Covariance

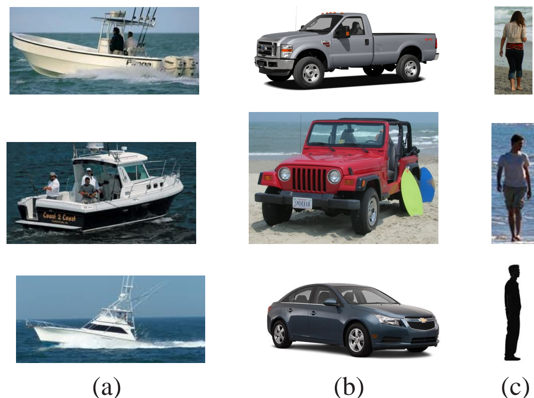
The bounding boxes identify video frame areas that likely contain moving objects of interest. The goal now is to verify whether each box contains either a boat, a motor vehicle or a person, or, alternatively, is a false alarm, e.g., due to an ocean wave. We employ a method developed by Tuzel *et al.* [11, 12] that compares covariance matrices of features. This approach entails computing a d-dimensional feature vector for every pixel in a region, and then generating a $d \times d$ covariance matrix from all such feature vectors. For computational efficiency, rectangular regions of pixels are often used. The degree of similarity between two regions is computed by applying a distance metric to their respective covariance matrices.

In order to detect objects in an image, Tuzel *et al.* apply exhaustive search where each object from a dictionary is compared with query rectangles of different sizes at all locations in the searched image. If the dictionary object and the query rectangle are sufficiently similar, the object is detected. However, even with the use of integral images [11], this approach is computationally expensive, especially when each video frame must be searched for many different sizes, locations and classes of objects. Instead, we use ROIs identified by the bounding boxes at the output of connected-component analysis (Figure 2) as the query rectangles and test each against three dictionaries: boats, motor vehicles, and people (described below). This greatly reduces the number of queries, thus increasing the search speed. Another advantage of our approach is the automatic selection of rectangle size (from the connected-component analysis). In contrast, Tuzel *et al.* use several fixed-size query rectangles, thus making an implicit assumption about the size of target objects. In our experiments, we used the following feature vector:

$$\vec{\xi}(\boldsymbol{x}) = \left[x, y, \left|\frac{\partial I[\boldsymbol{x}]}{\partial x}\right|, \left|\frac{\partial I[\boldsymbol{x}]}{\partial y}\right|, \left|\frac{\partial I^2[\boldsymbol{x}]}{\partial x^2}\right|, \left|\frac{\partial I^2[\boldsymbol{x}]}{\partial y^2}\right|\right], \tag{3}$$

that contains location, and first- and second-order derivatives of intensity I. Note that our feature vector is void of color attributes since we believe shape of objects of interest is more informative than their color.

Similarly to Tuzel *et al.*, we use the nearest-neighbor classifier to detect the objects. For each bounding box detected in a video frame (ROI), we compute the distances



(a) (b) (c)

Figure 3. Three samples from each of the dictionaries: (a) boats, (b) cars, and (c) people used in feature-covariance detection. Images have been obtained from a search on Google Images.

between its covariance matrix and the covariance matrix of each object in the three dictionaries. If the minimum distance is below a threshold ψ, then the class of the object from the dictionary corresponding to this minimum distance is assigned to the ROI. We repeat this procedure for all the detected bounding boxes.

Since covariance matrices do not lie in a Euclidean space, as the distance metric between two covariance matrices we use the Frobenius norm of the difference between covariance matrix logarithms proposed by Arsigny *et al.* [1], that is often referred to as the log-Euclidean metric.

For each class of objects we wish to detect, we created a dictionary composed of many representative images (samples are shown in Figure 3). In order to minimize bias in our results, we downloaded the images from Google Images rather than extracting them from our video data. The images are of assorted sizes and have been cropped to remove extraneous background clutter. We tried to diversify the selection to assure that our dictionaries are representative of real-life scenarios. For example, our motor vehicles dictionary consists of sedans, pick-up trucks, Jeeps, etc., at various orientations. Since our feature vector does not contain color, it does not matter what color these objects are; only the shape matters. Our people dictionary contains primarily silhouettes of people standing in different poses, and this works quite well because only the silhouettes of the people are discernible from far away. The dictionaries we have used in our tests are composed of 30-100 images.

2.5. Video Condensation

After each ROI has been identified as either a boat, motor vehicle or person, or, alternatively, ignored, a compact summary of all objects moving in front of the camera needs to be produced. To this effect, we employ the so-called video condensation [4], a method that is capable of shortening a long video with sparse activity to a short digest that compactly represents *all* of the activity while removing inactive space-time regions. Since the method is quite involved, we refer the reader to the original paper [4] and describe here only the main concepts.

Video condensation takes a video sequence and an associated cost sequence as inputs. The cost sequence is used to decide whether to remove specific pixels from the video sequence or not. Although, any cost can be used, we are interested in preserving the detected boats, cars or people while removing areas void of activity and thus we use the originally-proposed cost [4], namely moving/static labels at the output of behavior subtraction. More specifically, we use only those labels that are within bounding boxes of the detected boats, cars or people; the pixels outside are assigned 0 (no activity) and thus can be removed.

Consider a boat at sea traveling to the right in the field of view of a camera whose scan lines are aligned with the horizon (Figure 4). After the behavior subtraction step we obtain, ideally, a silhouette of the moving boat in each video frame. Jointly, all these silhouettes form a silhouette tunnel that is at some angle to the time axis t in the $x-y-t$ space of the video sequence (video cube). This tunnel starts at the left edge of the video cube and ends at the right edge (converse is true for a boat moving to the left). If another boat follows after the first boat disappears, another tunnel will be formed with a "dead" space, where is no activity, between them. In the first stage, video condensation eliminates this "dead" space by removing all frames void of silhouettes. In consequence, as soon as one tunnel finishes (boat exits on right), another tunnel starts (boat enters on left). In the next stage, ribbons rather than frames are removed. Ribbons are connected surfaces that, for the above case of horizontal motion, are rigid vertically and can flex horizontally between consecutive time instants (see [4] for details). This ability to flex in the $x-t$ plane (while remaining rigid vertically) allows ribbons to "fit" between two silhouette tunnels. If a ribbon cuts through "dead" space of no activity and is removed, the two tunnels get closer to each other. After recursively repeating this removal, the two tunnels almost touch each other which is manifested in the condensed video by the two corresponding boats appearing in the same video frame despite being far apart in time in the original video. Ribbons with increasing degree of flexibility can be used to remove topologically more complex "dead" space. Here, we first use rigid ribbons, or frames (flex-0), and follow them with progressively more and more flexible ribbons (flex-1, flex-2, and flex-3) [4].

2.6. Implementation Details

All code was written in C++ for computational efficiency and portability so that it may be readily deployed on embedded sensing platforms in the near future. For the image and video input and output, we used the open-source FFMPEG video libraries, and for connected-component analysis we

used the simple OpenCV library function *cvFindContours*. All other code was written "from scratch".

3. Experimental Results

We have collected hundreds of hours of coastal videos at 640×360 resolution and 5fps from cameras mounted at Great Point, Nantucket Island, MA. Figure 4 shows two examples from one of the videos we processed. We used the following ranges for parameters: background subtraction – $\alpha = 0.005$, $\theta = 20 - 25$, $\gamma = 1.0$, 2nd-order Markov model with 2 iterations; behavior subtraction – $M = 300 - 2500$, $N = 50 - 100$, $\Theta = 0.0 - 1.0$; connected-component analysis – 5×5 bounding box threshold, 20% box enlargement; object detection – dictionaries for boats, cars and people consisting of 30, 80 and 62 objects, respectively, with corresponding thresholds ψ between 2.5 and 4.0. The large range for M in behavior subtraction (training set size) is needed to adapt to weather conditions (see below).

Note the relative resilience of background subtraction to the presence of waves (second row in Figure 4). Although behavior subtraction provides only a slight additional wave suppression in this case, for videos with choppy water surface behavior subtraction offers a significant improvement for larger values of M. Also, note the detection of the boat on left despite a long wake behind it. The condensed frame on left (bottom row) shows two boats together that were never visible in the field of view of the camera at the same time. Similarly, on right, four boats have been combined into one condensed frame thus shortening the overall video.

The effectiveness of our joint detection and summarization system can be measured by the condensation ratio (CR) achieved for each class of objects (or combination thereof). Table 1 shows detailed results with cumulative condensation ratios (after flex-3) of over 18:1 for boats, 9:1 for people, but only about 5:1 for boats or people. Clearly, when one wants to capture a larger selection of objects, the condensation ratio suffers. Condensation ratios for another video with boats, cars and people are shown in Table 2. Note the last row in both tables labeled "behavior subtr." with very low condensation ratios. These are results for the whole-frame behavior subtraction output being used as the cost in video condensation instead of being limited to the bounding boxes of detected objects. Clearly, the spurious detections at the output of behavior subtraction reduce the condensation efficiency.

Table 3 provides the average execution time for each stage of processing in a single-threaded C++ implementation on an Intel Core i5 CPU with 4GB of RAM running Ubuntu 11.04 Linux. Note that the execution times for background subtraction and behavior subtraction depend only on the video resolution (in this case, 640×360). On the contrary, the execution times of object detection and video condensation vary depending upon the data (e.g., more activity means that more regions must be checked for the presence of objects and also that fewer frames can be dropped in the flex-0 stage of condensation). The benchmarks reported in the table were obtained for detections of cars in the video from Table 2 and are representative of typical execution times. Since video condensation operates on blocks

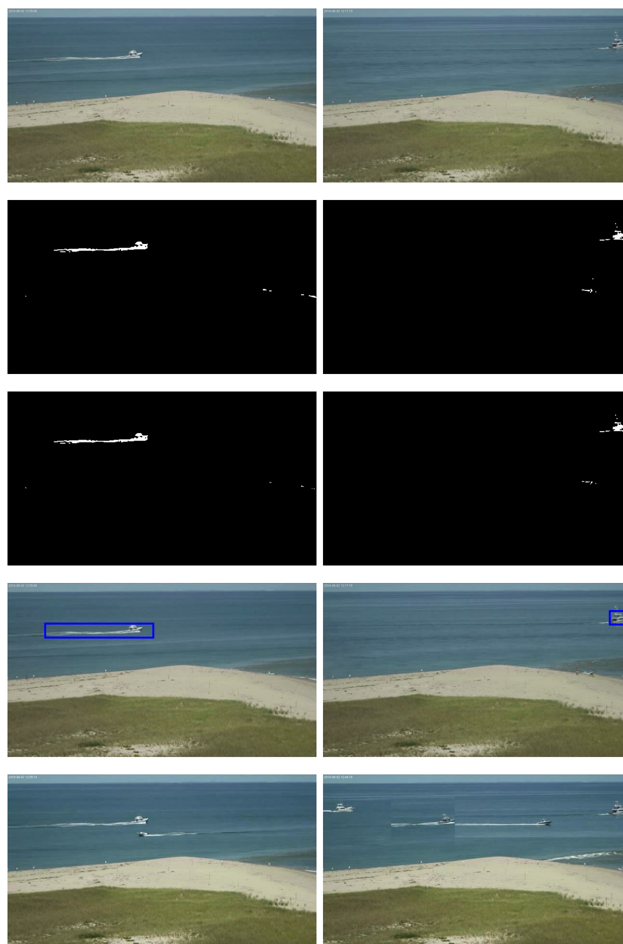


Figure 4. Samples of typical input video frames (top row) and outputs from the processing blocks in Figure 2: (row 2) background subtraction, (row 3) behavior subtraction, (row 4) object detection, (row 5) video condensation.

Table 1. Number of frames after each flex-step and cumulative condensation ratios (CR) for 38-minute, 5fps video with boats and people (11,379 frames after behavior subtraction).

Cost	# of frames after each step				CR
	flex-0	flex-1	flex-2	flex-3	
boats	1346	743	662	614	18.53:1
people	3544	2411	1772	1265	9.00:1
boats or people	4666	3225	2887	2414	4.71:1
behavior subtr.	11001	8609	8147	7734	1.47:1

Table 2. Number of frames after each flex-step and cumulative condensation ratios (CR) for 22-minute, 5fps video with boats, cars and people (6,500 frames after behavior subtraction).

Cost	# of frames after each step				CR
	flex-0	flex-1	flex-2	flex-3	
cars	768	598	513	439	14.81:1
boats	835	741	692	589	11.04:1
people	1729	1528	1527	1519	4.28:1
boats or people	2125	2045	2013	1969	3.30:1
behavior subtr.	6437	5843	5619	5460	1.19:1

of frames, we computed the average processing time of each "flex" pass by dividing the execution time for that pass by the number of input frames to that pass. For the background subtraction, we used second-order Markov neighborhood and two update iterations for each frame. Disabling the MRF model reduces the execution time to 0.156 sec/frame but significantly lowers the quality of the output. The object detection benchmark in the table includes the time for the connected components computation, the bounding box extraction, and the tests of each candidate region against three dictionaries (cars, people, and boats).

Clearly, our single-threaded implementation can process only 0.2fps. Even for a 5fps input video this is far from real time. One possibility to close this gap is by leveraging parallelism. For example, we have experimented with a pipelined, multithreaded implementation of video condensation for different flex parameters. We found that on a quad-core CPU this reduced the processing time by up to a factor of three for typical hour-long beach videos, compared to a traditional single-core approach. Similar parallelism can be applied to the other steps.

3.1. Conclusions

In this paper, we combined multiple video processing algorithms to create a flexible, robust coastal surveillance system that should be useful for marine biologists, environmental agents, etc. We tested our approach extensively using real coastal video sequences and showed that our system can reduce the length of typical videos up to about 20 times without losing any salient events. Our system can dramatically reduce human operator involvement in search for events of interest. Currently, our system does not operate in real time but with a careful implementation on a multicore architecture real-time performance is within reach. In the course of our research, we have observed that the detection step works better if objects with significantly different shape are treated as a separate class. For example, the sails of sailboats have large triangular shape, which is quite different from the hull and cabin of most motorboats; the detection works best in this case if we use a separate dictionary for each type of boat.

Table 3. Average execution time for each stage of processing.

Processing Step	Average Execution Time
Background Subtraction	0.292 sec/frame
Behavior Subtraction	0.068 sec/frame
Object Detection	0.258 sec/frame
Video Condensation	
flex 0	0.034 sec/frame
flex 1	2.183 sec/frame
flex 2	1.229 sec/frame
flex 3	0.994 sec/frame
Total	5.058 sec/frame

References

[1] V. Arsigny, P. Pennec, and X. Ayache. Log-euclidean metrics for fast and simple calculus on diffusion tensors. *Magnetic resonance in medicine*, 56(2):411–421, 2006. 4

[2] A. Bovik, editor. *The Essential Guide to Video Processing*. Academic Press, 2009. 2

[3] H.-W. Kang, Y. Matsuhita, X. Tang, and X.-Q. Chen. Space-time video montage. In *Proc. IEEE Conf. Computer Vision Pattern Recognition*, pages 1331–1338, June 2006. 1

[4] Z. Li, P. Ishwar, and J. Konrad. Video condensation by ribbon carving. *IEEE Trans. Image Process.*, 18(11):2572–2583, Nov. 2009. 2, 4

[5] T. Little, P. Ishwar, and J. Konrad. A wireless video sensor network for autonomous coastal sensing. In *Proc. Conf. on Coastal Environmental Sensing Networks (CESN)*, Apr. 2007. 1

[6] J. McHugh, J. Konrad, V. Saligrama, and P.-M. Jodoin. Foreground-adaptive background subtraction. *IEEE Signal Process. Lett.*, 16(5):390–393, May 2009. 2

[7] J. Oh, Q. Wen, J. Lee, and S. Hwang. Video abstraction. In S. Deb, editor, *Video Data Mangement and Information Retrieval*, chapter 3, pages 321–346. Idea Group Inc. and IRM Press, 2004. 1

[8] Y. Pritch, A. Rav-Acha, and S. Peleg. Non-chronological video synopsis and indexing. *IEEE Trans. Pattern Anal. Machine Intell.*, 30(11):1971–1984, Nov. 2008. 1

[9] V. Saligrama, J. Konrad, and P.-M. Jodoin. Video anomaly identification: A statistical approach. *IEEE Signal Process. Mag.*, 27(5):18–33, Sept. 2010. 2, 3

[10] A. Samama. Innovative video analytics for maritime surveillance. In *Int. Waterside Security Conference*, Nov. 2010. 1

[11] O. Tuzel, F. Porikli, and P. Meer. Region covariance: A fast descriptor for detection and classification. In *Proc. European Conf. Computer Vision*, May 2006. 2, 3

[12] O. Tuzel, F. Porikli, and P. Meer. Pedestrian detection via classification on Riemannian manifolds. *IEEE Trans. Pattern Anal. Machine Intell.*, 30(10):1713–1727, Oct. 2008. 2, 3

[13] Q. Wu, H. Cui, X. Du, M. Wang, and T. Jin. Real-time moving maritime objects segmentation and tracking for video communication. In *Int. Conf. on Communcation Technology*, Nov. 2006. 1

E Draft of System Specification

The following is an early draft of my system specification. It is a little out of date now, but I have included it because it still has some good information about the parameters that the system takes and how to tune them. The working title of this system was SEAL (Salient Extracted visuALS).

E.1 Introduction

The SEAL (Salient Extracted visuALs) program analyzes videos of the beach and distills them, creating statistics and summaries of the most interesting features. This document specifies the input, output, and behavior of the SEAL system.

E.2 System Block Diagram

The system block diagram can be found in Figure 19. A detailed diagram of the object detection block can be found in Figure 20. The block diagram shows all of the possible processing paths through which data in the system can flow. All of the modules shown in the block diagram are connected together in one top-level executable file.

E.3 Usage

The SEAL program has the following syntax:

```
./seal MODE CONFIG_FILE
```

The configuration file `CONFIG_FILE` is a human-readable text file containing all of the input arguments and configuration parameters for running the seal program.
There are three modes of operation that can be specified using the `MODE` option:

- **configure:** Generates a default configuration file with the recommended default parameters.

- **tune:** Processes and plays back the background-subtracted and/or behavior-subtracted and/or masked videos in (nearly) real-time. Use this to test the background subtraction parameters and/or behavior subtraction parameters and/or the mask file; iteratively change the configuration file and re-run the playback to assess the performance. Tuning is the step that requires the greatest amount of human intervention, which is why it has its own mode. The default parameters will generally work pretty well, but nevertheless, better performance can be

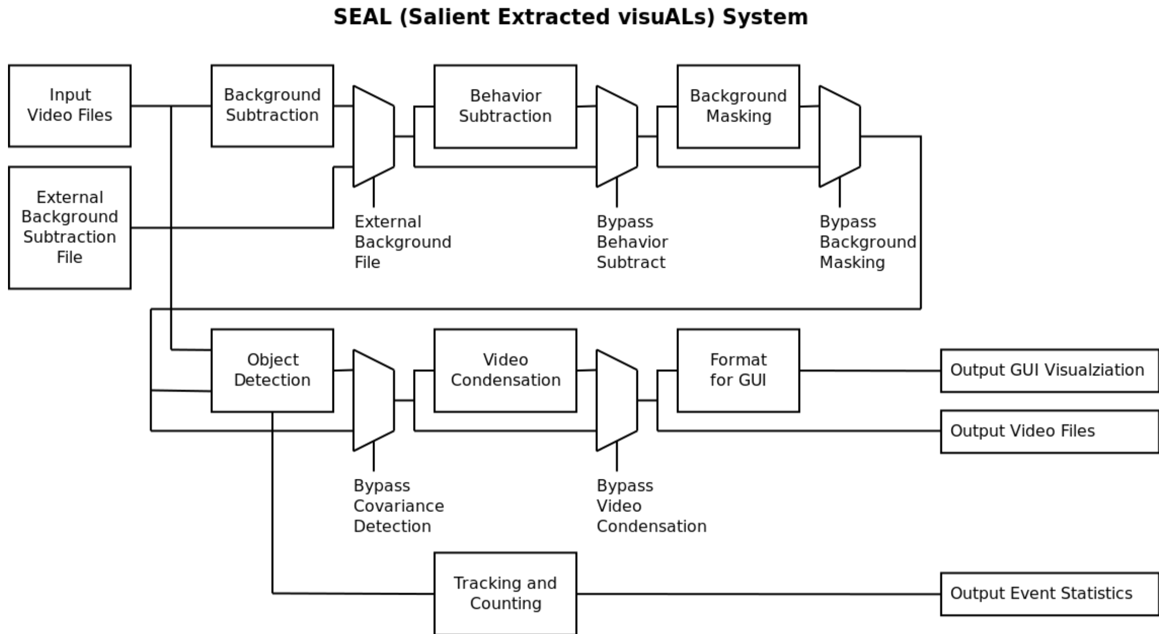


Figure 19: SEAL System Block Diagram

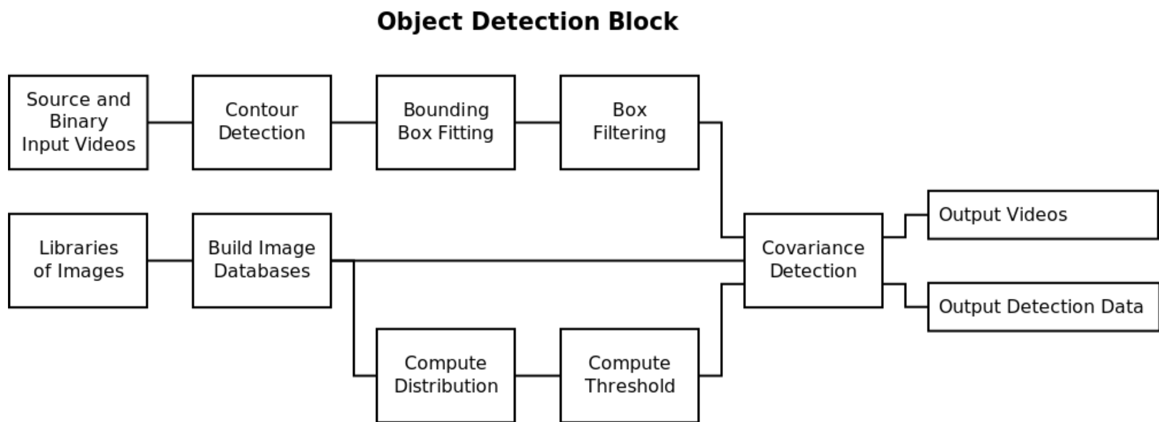


Figure 20: Object Detection Block Diagram

achieved if the behavior subtraction and background subtraction are tuned to minimize the numbers of false positives and false negatives. It makes sense that the `tune` mode only tests the background subtraction, behavior subtraction, and masking processing stages (i.e., it doesn't test video condensation or object detection) because the other stages don't have any parameters that need to be iteratively tuned like this. Luckily, this mode is easy to implement because these processing steps can be computed very quickly (i.e., in real time, or nearly so) and these steps all occur near the beginning of the processing pipeline. (For tips on selecting suitable parameters, please see report.tex.)

- **process:** Performs batch processing of all the input files specified in `CONFIG_FILE`. Note that the same parameters are applied for each of the specified input files; if instead you want different parameters for each file, you must use a separate configuration file for each input file and run the `seal` program separately for each configuration file.

E.4 Configuration File

There are two categories of entries in the configuration file: required fields and optional fields. Each type is described below. Note that the defaults for these fields (i.e., the settings generated by using the `CONFIGURE` mode) are given after each option.

Required Fields

Required fields are fields that must always be present in the configuration file.

- `input_file = filename.avi`
  - You must have at least one `input_file` entry. You can have as many as you want, one for each file file that you want to process.
  - Input video files can be in any video format supported for decoding by FFMPEG. (Input files may also be in my proprietary ".dan" video file format.)
- `output_dir = ./output`
  - Specifies the directory in which to save all of the output files.
  - Note that this program will clobber any existing files with the same names in the output directory without warning.

- `output_videos = final_videos`
  - These are flags that indicate which intermediate video files should be created (primarily for debugging purposes).
  - Simply list each type of video that you want to output, separated by spaces, for example, to output all videos, use the following: `output_flags = condensed0, condensed1, condensed2, condensed3 background-subtracted behavior-subtracted masked`. The 0, 1, 2, or 3 following the word "condensed" indicates to which pass of the the video condensation algorithm these videos correspond (flex0, flex1, flex2, or flex3). Flex0 means that the frames containing no activity have been dropped, but no further condensation has been performed. Flex3 videos will have the greatest amount of condensation. Flex1 and flex2 will fall somewhere in between.

Optional Fields

Although these fields are generally optional, many are still required

- `backgroundsubtract_external_file = (not included)`
  - This option allows you to specify an existing file to use as the output of the background subtraction, rather than computing.
  - This option is primarily used for debugging purposes.
  - Note that there is no corresponding `behaviorsubtract_external_file` option. This is because you could accomplish the same thing by just specifying an external behavior-subtracted file as the `backgroundsubtract_external_file` and bypassing the behavior subtraction block.
- `backgroundsubtract_type = MovingAverage`
  - *This option is required if* `backgroundsubtract_external_file` *is not used.*
  - Supported types: `MovingAverage`, `KDE`
  - The simple `MovingAverage` mode is recommended, since it is fast and has pretty good accuracy. `KDE` works better but is much slower.
- `backgroundsubtract_num_init = 100`
  - *This option is required if* `backgroundsubtract_external_file` *is not used.*

  - Number of frames to use to initialize the background for background subtraction.
  - Note that the frames used to initialize the background subtraction are discarded after the background subtraction module uses them; in other words, the frames that are outputted by the background subtraction module while it is initializing are NOT passed along to the next pipeline stage.

- `backgroundsubtract_alpha = 0.001`
  - *This option is required if `backgroundsubtract_external_file` is not used.*
  - Moving average parameter. (Controls the effective buffer length.)

- `backgroundsubtract_theta = 10.0`
  - *This option is required if `backgroundsubtract_external_file` is not used.*
  - Background subtraction threshold.

- `backgroundsubtract_gamma = 5.0`
  - Include this option if you wish to enable the Markov random field (MRF) model when performing background subtraction. Using the MRF model is somewhat computationally intensive but improves the background subtraction by reducing stray false positives and filling in some of the missing pixels inside objects. If you do not include this parameter, the MRF model with not be used.
  - This parameter adjusts the argument of the exponential when using Markov random fields to influence the thresholds.

- `behaviorsubtract_num_averaging = 200`
  - *This option is required unless the behavior subtraction module is being bypassed.*
  - Number of frames in the sliding window over which to perform the averaging.

- `behaviorsubtract_num_training = 1000`
  - *This option is required unless the behavior subtraction module is being bypassed.*

  - Number of frames to use to train the behavior subtraction.
  - Note that the frames used to train the behavior subtraction are discarded after the behavior subtraction module uses them; in other words, the frames that are outputted by the behavior subtraction module while it is training are NOT passed along to the next pipeline stage.

- `behaviorsubtract_threshold = 10`
  - *This option is required unless the behavior subtraction module is being bypassed.*
  - This is the distance between the window sum and the max sum when performing the comparisons that determine whether or not a pixel is "interesting."

- `mask_file = (not included)`
  - If this option is provided, a binary mask file will be applied to the output of the behavior subtraction stage.
  - If this option is not provided, no masking will be applied.

- `videocondensation_epsilon = 0`
  - If you want to use video condensation; you must include this option; otherwise, if you wish to bypass video condensation, simply omit this option from the configuration file.
  - This is the stopping criterion. Generally you will leave this at zero (which is the default), but if you want higher condensation ratios and you don't care about losing pieces of your objects, you can increase it.

- `objectdetect_library = (not included)`
  - This is a folder containing a database of images of the objects that you want to detect.
  - If you want to detect multiple classes of objects, give a separate `objectedetect_library` entry for each. For example, if you want to detect both cars and boats, you would have one entry that points to a directory containing images of cars and another entry that points to a directory containing images of boats.

- `objectdetect_confidencelevel = (not included)`
  - *This parameter must be specified when at least one* ***`objectdetect_library`*** *entry is included. An error will be given if this parameter is specified when no* ***`objectdetect_library`*** *entries are included, or if it is omitted when* ***`objectdetect_library`*** *entries are present.*
  - Allows you to specify the confidence level for detecting the objects.
  - It should be specified as a fraction, not a percentage (e.g., 0.90 rather than 90%).
  - For simplicity, this confidence level is applied to all classes of objects. For example, you can't detect cars with confidence of 95% and boats with confidence of 90%; you must use the same confidence level for each class of objects.

Syntax Notes

Here are a few notes on the syntax of this configuration file.

- The order of entries in the configuration file does not matter.
- The configuration file is case-sensitive.
- Comments may be included, with the following restrictions (since our parsing is so primitive):
  - A comment must appear on its own line; it cannot follow at the end of another line of text.
  - A **`'#'`** indicates the start of a comment. This MUST be the first character on the line; it cannot be preceded by whitespace.
- A minimal amount of checking will be performed to ensure that the parameters are all understood and that they all make sense. This should help to catch some errors, but the user should still take care to avoid typos.

Example

Here is an example configuration file.

```
input_file = beachvideoA.avi
input_file = beachvideoB.avi
input_file = beachvideoC.avi
output_dir = ./output
output_videos = final_videos
backgroundsubtract_type = MovingAverage
backgroundsubtract_num_init = 100
backgroundsubtract_alpha = 0.001
backgroundsubtract_theta = 10.0
backgroundsubtract_gamma = 5.0
behaviorsubtract_num_averaging = 200
behaviorsubtract_num_training = 1000
behaviorsubtract_threshold = 10
videocondensation_epsilon = 0
covobjectdetect_library = ./boat_library
covobjectdetect_library = ./car_library
covobjectdetect_confidenceleve = 0.90
```

E.5 Output

This program generates the following output in the specified output directory:

- Video files are specified by the option `output_videos`. The output videos have the same filenames as the corresponding input videos, except that their names are prefixed with the characters "condensed0_", "condensed1_", "condensed2_", "condensed3_", "backgroundsubtracted_", "behaviorsubtracted_", or "masked_", depending on which videos they are.
- A simple HTML file `index.html` is also generated, which gives hyperlinks to the generated video files for easy viewing, summaries of the parameters used, and statistics[2] related to the detected objects.

E.6 Caveats

This section describes some of the assumptions we have made when designing this system, as well as some of the limitations of our current system.

[2]Not implemented quite yet. Statistics include counts of numbers of boats, numbers of seals, times in the video at which interesting events occurred, etc.

- Input video files should be selected to meet certain assumptions for good data, such as no camera shake, no water on lens, constant lighting (i.e., no change in cloud cover and no sudden changes in camera gain), no zoom, etc.

- Maximum input video resolution: Generally about 320x240 due to high memory requirements of video condensation.

- "src_ref_mode" from my old "beachanalyzer" program will not be supported in the new "seal" program.

- Video condensation always runs flex0 through flex3; you cannot control the number of flex passes.

- We do not attempt to detect boats that are very far away on the horizon because they are very tiny and we probably do not have enough pixels to detect them accurately, anyway. Luckily, boats that are on the

- It is interesting to note that boats do not tend to go out onto the water when the ocean is very choppy. As a result, it is fairly rare that we will have to detect a boat in the middle of heavy waves.

- If we limit our system to process data on days in which the weather is good (not enough wind to make the camera shake), chances are that there will be fairly few waves to interfere with the boats.

E.7 Other Observations

- Since waves in the ocean and along the shoreline cause many false-positives, I have selected a mask that only includes the sandy beach region. I have also masked on the grassy dune, just because I don't care about this region, and I don't want any possibility of dune grass blowing in the wind to create false positives.

- Video condensation cuts out any frames without moving objects (i.e. people)

- Notice that the background subtraction algorithm detects something that wouldn't be visible by the human eye: a seal is moving around near the shoreline, in the left half of the video, near the car. This is like the person walking across the highway overpass in my older test videos; the human eye wouldn't have noticed

it, but marking it with white on black using the background subtraction makes it more visible.

- Notice that some notion of the time scale is important, as this affects what gets kept and what gets cut out. Depending size of the "window" or "buffer" of the background subtraction recursive filter, we may or may not detect the car as background, depending on how long it sits before driving away.

- When people stand near the shore, their legs are in front of the beach portion but their upper bodies are in front of the waves and ocean. Thus, we risk chopping people in half and other distortions in the condensed video because the mask that we are using to ignore the ocean passes through their bodies.

- There are a few false detections when clouds pass overhead and cast shadows on the beach. I could try to minimize these by adjusting my detection threshold.

E.8 Miscellaneous

Interesting side note: Sometimes it is useful to watch the cost function video because it can draw your attention to things that you might not otherwise notice. For example, the human eye tends to focus on people and cars in the foreground on the beach, and easily overlooks tiny (a few pixels wide) objects that blend into the background and move around near the periphery of the screen (such as seals). On the black and white cost function, these small objects show up as white detected pixels and are much more apparent to the viewer, which signals the viewer to go back and check the original video in these locations for something interesting. I have another video of the highway overpass near St. Mary's street, which also shows this: I would never have noticed a tiny pedestrian walking over the bridge had I not watched the cost function video.

Observations about the condensed videos:

- In the people and cars video, sometimes the people disappear and reappear as they walk quickly across the beach. This is because the networked camera dropped frames when it was capturing the original video. (It is NOT a problem with the background subtraction algorithm or video condensation algorithm.) The dropped frames cause the person to "teleport" from position to position,

rather than to move smoothly between them. It is simply a problem with the way the original video was recorded. (Analogies: strobe light. picket fence.)

- In the boats video, you'll notice that rectangular regions of the ocean near the boats don't match up. This is because the pixels in the region of one boat are from a different moment in time than the pixels in the region of another boat. The difference in coloration of the ocean pixels between these two regions is due to changing ambient light levels over time (due to clouds, movement of sun, etc.) or the camera's automatic gain control. Also, the textures of these two regions are different because the waves are different at different moments in time. These regions are rectangular due to the fact that our video condensation algorithm "carves" either horizontal or vertical "ribbons". The fact that we see these discontinuities indicates that our algorithm is really condensing the video by showing non-overlapping segments of video from different moments in time on the screen simultaneously!

- Notice that the condensation algorithm works better when the objects tend to move across the screen in the same the direction. For example, during certain times in the boats video, we see that several boats traveling in the same direction can be sent across the screen one right after another, in rapid succession. This is similar to the results of condensing video of a highway, in which cars are all traveling in the same direction (see http://vip.bu.edu/projects/video/video-condensation/) – under these circumstances, higher compression ratios can be achieved.

- Another example of a scenario that yields good condensation results is one in which activity occurs at opposite sides of the screen at different moments in time, but the objects involve stay on their respective sides of the screen; after condensation, the two activities can be played back at the same time, side by side.

F Preliminary C++ Video Condensation Benchmarks

This section provides some of our early benchmark results for our multithreaded, pipelined C++ implementation of video condensation that we implemented during

the summer of 2011. Please understand that these benchmarks are out of date and no longer very relevant because the source code has changed tremendously since then. We have included these results only to give an idea of the work that was done much earlier in the project, for historical reasons.

Summer 2011 Research Summary

Summary of software developed and data gathered for the multi-threaded C++ implementation of video condensation as applied to processing beach videos from Great Point, Nantucket.

Daniel J. Cullen
dcullen@bu.edu

2011-09-19

Prof. Little & Prof. Konrad

Electrical and Computer Engineering
College of Engineering
Boston University

1 Introduction

The purpose of this document is to document the work I did over the summer. It explains some of the code I wrote and some of the results we obtained. It is important to document this work so that we do not forget what I accomplished, and so that this work can be more useful to others in the future.

By no means is this document an exhaustive summary; it does not go into detail about the workings of all of the code, nor does it contain all of my research notes. For these things, please see the source code comments, Doxygen documentation, and my research notes. There are also many video files containing insightful results.

1.1 Benchmarks and Discussion

Preliminary Benchmark Results

For the benchmarks of the C++ code, we use the Linux `time` command. This command reports the "real" time, "user" time, and "system" time. The CPU time actually used by the process is "user+system", but we do not want to take into account the amount of time used on each of the cores. Rather, we want the "wall-clock" time, so we take the benchmark recorded by the "real" time statistic. Besides, taking the "real" time better agrees with the way that we take the MATLAB benchmark.

MATLAB benchmarks are recorded using the MATLAB `tic` and `toc` benchmarking functions. We sum up the total time it takes to process flex0 through flex3. This gives us the total wall-clock time of running the MATLAB code, which we can use to compare against our C++ code benchmarks.

(Sanity check: I verified that user+sys for multiple cores is approximately equal to real time for single core.)

For the multi-core benchmark, I left all 48 processor cores on iss9 enabled. (Of course, the algorithm is designed to use only up to four cores at once, but we wanted to allow the program easy access to all resources.)

Discussion of Preliminary Benchmark Results

Note that the local disk storage vs. the network drive storage doesn't make a huge difference in performance for any of these benchmarks.

Note that we get a speedup of about a factor of 3 of the multi-core approach over the single-core approach. This is as we would expect. It is not a factor of 4 because the first thread spends relatively time processing compared to the other two threads; it only handles loading and saving the data. When the background subtraction is enabled, we get a slightly higher (about 3.1 vs 2.9) speedup because the first thread gets to do more processing work. Another thing to note is that the

Table 1: Benchmark results

Benchmark Results			
Type of test	**Cores**	**Storage used**	**Wall-clock time**
C++, marsh plaza src and pre-computed cost (9800 frames)	Multi-core	/home (network disk)	74m10.021s
C++, marsh plaza src and pre-computed cost (9800 frames)	Multi-core	/tmp (local disk)	75m14.484s
C++, marsh plaza src and pre-computed cost (9800 frames)	Single-core	/tmp (local disk)	156m37.196s
MATLAB, marsh plaza src and pre-computed cost (9800 frames)	Single-core	/tmp (local disk)	61m9.1s
MATLAB, marsh plaza src and pre-computed cost (9800 frames)	Single-core	/home (network disk)	60m53.2s
C++, marsh plaza src and pre-computed cost (9800 frames)	Single-core	/tmp (network disk)	145m44.425s
C++, marsh plaza src and pre-computed cost (9800 frames)	Single-core	/home (local disk)	155m15.793s
C++, marsh plaza src, on-the-fly cost, long video (26488 frames)	Single-core	/tmp (local disk)	143m46.598s
C++, marsh plaza src, on-the-fly cost, long video (26488 frames)	Multi-core	/tmp (local disk)	49m17.085s
C++, marsh plaza src, on-the-fly cost, long video (52976 frames)	Single-core	/tmp (local disk)	282m47.967s
C++, marsh plaza src, on-the-fly cost, very long video (52976 frames)	Multi-core	/tmp (local disk)	90m14.248s

speedup is greater for long video sequences because the time required to fill the pipeline becomes less significant for longer sequences.

Also note the inconsistencies between the C++ marsh plaza single-core pre-computed cost and on-the-fly cost! A video with more than twice as many frames takes about one third the computation time! The culprit: probably disk access time. Also, disk access time would also explain why the MATLAB code runs twice as fast; the MATLAB code is using a Motion JPEG codec.

One thing that I might do to try to get a slightly better comparison between my C++ code is to create a ramdisk on iSS9, copy the input files to this ramdisk, and then run the code from the ramdisk; this will mitigate disk accses time as being an issue. Unfortunately, I'm not sure I really have enough ram on iss9 to make a sufficiently large ramdisk; running the command free -m only reported about 3.8GB free.)

Another thing that I might try to do is, when running the code, change the flags so that we don't write all the intermediate result videos to disk; only write the end result videos. I decided to do this test first. I ran single-core and multi-core instances of this, without outputting any output video files. I'm still using .dan input files (for source and cost videos), and these are being read from the network drive (/home). These results are shown in Table 2.

More Benchmark Results

Discussion

There are several other inconsistencies that must be addressed:

- My MATLAB benchmark is inaccurate because the video files are unreliable (due to the lossy MJPEG compression, even when quality setting is 100%.
- Also, the MATLAB benchmarks are unreliable because of how I use the MATLAB `tic/toc`

Table 2: More benchmark results

Benchmark Results			
Type of test	**Cores**	**Storage used**	**Wall-clock time**
MATLAB, marsh plaza src and pre-computed cost (9800 frames) (all output files)	Single-core	/tmp (local disk)	61m9.1s
MATLAB, marsh plaza src and pre-computed cost (9800 frames) (all output files)	Single-core	/home (network disk)	60m53.2s
C++, marsh plaza src and pre-computed cost (9800 frames) (all output files)	Single-core	/tmp (network disk)	145m44.425s
C++, marsh plaza src and pre-computed cost (9800 frames) (all output files)	Single-core	/home (local disk)	155m15.793s
C++, marsh plaza, pre-computed cost (9800 frames) (no output files)	Single-core	/home (network disk)	89m59.561s
C++, marsh plaza, pre-computed cost (9800 frames) (no output files)	Multi-core	/home (network disk)	61m38.005s

commands; to be consistent, I should do a wall-clock time, rather than separate times and summing them, just to make sure I haven't forgotten to capture any time. The wall-clock `tic` should be the first line of the MATLAB script and the corresponding `toc` should be the last line of the MATLAB script.

- I should really get the uncompressed AVI working in MATLAB and in my C++ code so that the codec is the same. Also, using the AVI uncompressed codec in MATLAB will allow us to have reliable results, unlike the unreliable lossy MJPEG files.
- The server iss9.bu.edu might need a reboot because most of its RAM is used up but it's idle. This might explain why my recent C++ benchmarks have been performing poorly.
- Another thing to consider is that my C++ code, when forced to run only on a single core, may perform worse than the algorithm coded in C++ to run several independent passes; this is because there is overhead to perform the context switches. (I'm not sure how the overhead for writing/reading the intermediate flex videos for the multi-pass approach factors into this, though.)
- Another issue I encounter that other people might be running intensive applications on iss9.bu.edu at the same time that I'm trying to run my applications. The machine has 48 cores and a bunch of RAM, but we might still be competing over some resources, especially RAM and access to harddisk or network drive.

Even more benchmarks

I addressed some of these issues. For example, I implemented the lossless MJPEG2000 support in MATLAB, which works correctly (unlike the lossless MJPEG support, which still gives compression artifacts even with 100% quality selected). The new benchmarks are given in Table 3.

Commentary on these benchmarks:

- Keep in mind that these latest benchmarks were run only using 9800 frames.

Table 3: Even more benchmark results

Benchmark Results			
Type of test	**Cores**	**Storage used**	**Wall-clock time**
MATLAB, lossless MJPEG marsh plaza (9800 frames, 270x162 resolution)	all cores enabled	/tmp (local disk)	78m40.125s
MATLAB, lossless MJPEG2000 marsh plaza (9800 frames, 270x162 resolution)	all cores enabled	/tmp (local disk)	92m43.758s
MATLAB, uncompressed AVI marsh plaza (9800 frames, 320x240 resolution)	all cores enabled	/tmp (local disk)	365m51.947s
C++, marsh plaza (.dan format) (9800 frames, 320x240 resolution)	single-core	/tmp (local disk)	420m35.318s
C++, marsh plaza (.dan format) (9800 frames, 320x240 resolution)	multi-core	/tmp (local disk)	225m18.521s

- I think other people were running jobs on the machine at this time too.
- Comparing the first two benchmarks in the table above, we see that the file format and file size has a significant impact on the execution time.
- Keep in mind that the uncompressed AVI benchmark can't be readily compared to the other benchmarks because it is a bigger resolution than the all others (320×240 rather than 270×162), because I had to pad it to get rid of artifacts.
- The ratio of C++ single-core to C++ multi-core from the 2011-09-17 results is about 1.867. The ratio of C++ single-core to C++ multi-core from the 2011-08-25 results is about 2.082. I forget why I thought this was important to note. Maybe to try to check for reproducibility and consistency on the machine on which I'm running the tests? Maybe the worse ratio on 2011-09-17 suggests that more people were using Processor Core 0 on 2011-09-17. Either that, or the 320x240 videos don't perform as well as the 270x162 videos? I'm not sure.

Here is another test that I thought about doing: I should Implement my C++ VideoReader to use mjpeg (270x162 res) input files or uncompressed RGB24 AVI (320x240 res) input files instead of .dan files. These would be great tests, but they won't work on iss9, since I haven't gotten the FFMPEG support for OpenCV working yet, which is why I keep avoiding this test.

Summary and Reflections:

Ideally, I would have used the exact same video file formats for the MATLAB and the C++ code. This wasn't possible because first I was struggling with formats that MATLAB could understand, and then I was struggling with getting the OpenCV and FFMPEG codecs working on the server iss9.bu.edu. (It would have been great if iss9 had ubuntu, because it was a breeze installing the packages on my Linux laptop. Or it would have been great if I could use my Linux laptop to do all the benchmarks, but I don't have MATLAB installed on my Linux laptop, and I needed a good common Linux platform to run the tests).

Also, I would have ideally had two different versions of my C++ code: one that was structured like the MATLAB code, that wrote to and read from separate files for each flex stage, which I could use for the single-core benchmarks, and another version that uses my pipeliend algorithm for doing the multi-core benchmarks. Thus, I could really get a better comparison between MATLAB and C++ and then between C++ single-thread and C++ multi-thread. Unfortunately, I didn't have the time to fully code or debug a version of the C++ code optimized for a single thread, so I tried to run the same multi-threaded code for both C++ tests, and just disable all cores except for one core when running the single-thread code. I had hoped that this would be a fairly good approximation of a C++ single-threaded implementation. Unfortunately, there are a lot of unknown factors that confound my measurements: overheads of multiple threads make the code less efficient when run on a single core (i.e. context switching overheads), time to write multiple video files, core management in the OS (and maybe other users on iss9 running processes on the chosen core), etc., so I'm not sure how reliable my approximation of single-threaded processing is.

Rather than just one benchmark, I've had to run so many benchmarks just to get a feel of things because I haven't been able to compare apples to apples. It's very hard when the best that you can do is compare apples to oranges. My hope, however, is that by comparing apples to oranges, peaches, bananas, and lots of different kinds of fruit, I can get a good qualitative, intuitive feel about the relative performance.

That said, I think the best results to compare are:

- The MATLAB lossless MJPEG2000 results (from 2011-09-16): **92m43.758s**
- The MATLAB lossless MJPEG2000 results (from 2011-09-19, test 1): **72m42.656s**
- The MATLAB lossless MJPEG2000 results (from 2011-09-19, test 2): **TBD**
- C++ single-core results (from 2011-08-30, all output files): **145m44.425s**
- C++ single-core results (from 2011-09-15, no output files): **89m59.561s**
- C++ multi-core results (From 2011-09-15, no output files): **61m38.005s**

You'll notice that there are several MATLAB lossless MJPEG2000 results above. They were all run more or less the same way, but for some reason they came out somewhat inconsistently. I ran this benchmark several times because I was curious if the results are consistent, and the answer turns out to be: not really.

Another thing you'll notice is that since I've discovered that file format/file size makes a difference, I thought that for the C++ benchmark, a huge input file with no output files might be roughly good comparison against the MATLAB with the losslsess MJPEG2000 which has smaller

file sizes, which is why I ran the benchmarks on 2011-09-15 with no output files. I also thought I might try to level the playing field by comparing tests involving uncompressed AVI in MATLAB against C++ test using .dan format (granted, there is some lossless zlib compression involved in these .dan format tests, but I think the results will still be fairly comparable). These comparisons are given below:

- MATLAB uncompressed AVI results (from 2011-09-17): **365m51.947s**
- C++ .dan format single-core (from 2011-09-17): **420m35.318s**
- C++ .dan format multi-core (from 2011-09-17): **225m18.521s**

Some bottom-line results are given in the next section.

1.2 Bottom Line on time performance benchmarks

The performance between the MATLAB code and the single-core C++ code is approximately the same; the single-core C++ tests might be slightly (less than 10%) slower because the threaded algorithm is optimized for four cores, not one core.

The C++ multi-core code runs approximately 2x faster than the C++ single-core code for shorter (i.e., approx. 10,000 frames) sequences, and as much as 3x or more faster for longer (i.e., approx 25,000 frames or more) sequences. The first stage of the pipeline (background subtraction and flex0) is currently under-utilized, since the flex1, flex2, and flex3 stages of the pipeline require considerably more computation time. With more processing in the first stage (e.g., behavior subtraction?), the multi-core vs. single-core performance increase factor will approach closer to the 4x upper limit.

Aside from implementing the multi-threaded algorithm, I have not spent much time trying to optimize my C++ code. With a little more time optimizing the code (e.g., fixing some memory management things in the ribbon carving code), I should be able to increase the speed of my C++ code by 20% for each of the flex processing stages.

Other observations:

- The file format and file size has a significant impact on execution time.
- The storage medium (local disk versus network drive) doesn't make much of a difference on execution time.

So, the bottom line is that even though the single-core C++ algorithm doesn't really improve over MATLAB, the multi-core algorithm does, and will be especially useful when we make the cost function processing in the first pipeline stage more computationally intensive. Furthermore, there is still plenty of room for optimization improvement with the C++ code.

1.3 Next steps

I really need to finish optimizing the C++ code. Earlier this summer, I spent a lot of time just trying to get my C++ code to produce the exact same results as the MATLAB code. More recently, I've been spending just a lot of time trying to compare the C++ vs. MATLAB performance. I've spent so much time working on these other things that I haven't had time to sit down and refine my C++ code at all. For example, I think I can probably get my C++ ribbon carving code to run about 20% faster than Huan-Yu's C version that I'm currently using, just because I can manage the memory more efficiently. (See log.odt on 2011-09-14 for my benchmark of this. Granted, there are still some problems with it, but I still believe there's room for improvement.) There are also probably many other things I can do, like make sure the compiler is properly inlining the functions for accessing pixel data.

2 Conclusion

This concludes the *Summer 2011 Results* report.