**The Problem**

The goal of this assignment was to implement the background subtraction via codebooks algorithm. In order to accurately evaluate the results of this assignment later in this report, we were given roughly 3000 images. Each image is a frame from a video. There are obviously too many images to display the entire dataset here. To show a sample of the data, the following images are from the dataset:



**The Process**

There are a number of steps that we must complete before we may actually perform background subtraction. The first thing we must do is read in the images we intend to process, then we must build the codebook, then we must trim the codebook, then we may finally perform background subtraction. There is, however, an extra step for this assignment after performing background subtraction. The additional step is to perform morphological operations to clean the noise out of our foreground images.

First, to read in the images I iterated over the files in each of four directories containing images from the video we are processing. Since it would take about three to four hours to process all 3000 frames, I only read in every fifth frame and stored it into a cell of a cell array. By processing every fifth image I maximize the span of the 600 frames I used. I could process 600 frames consecutively, but would not cover as much of the video. Likewise, I stored the images into a cell array to optimize access times when building the codebook (as opposed to using imread all the time).

Now that we have all the images, we must build the codebook. To do this, I followed the pseudocode given to us in the *Background Modeling and Subtraction by Codebook Construction* paper, written by Kyungnam Kim, et al. To follow the pseudocode, I iterate over each of the frames store in the cell array. For each of these images, I iterate over each pixel. For each pixel, we fetch its RGB values and store that in a vector, while summing the RGB values and storing the sum in another variable. Since each pixel has its own codebook, we fetch the pixel’s corresponding codebook next. In my implementation, a codebook is a cell array storing cell arrays, each consisting of structs storing RGB and auxiliary values. Since each pixel has its own codebook, each codebook is store in a cell matrix. The cell matrix is the same size as the frames in the video. Therefore, when we retrieve a codebook we iterate over each cell array in it and extract the RGB and auxiliary values from the structs. Once we have these values, I put them into vectors for simpler processing. Once the values are in their corresponding vectors, we can check the color distortion as well as brightness to see if there is a match between any of the existing codewords and the current pixel (we stop searching on the first match). If there is a match, then we update that codeword with new values. If there is no match then we create a new codeword and enter it into the codebook. To check if there is a match, two conditions must be met. The first condition is that the color distortion must be less than some threshold we choose. To calculate the color distortion, we take the square root of the values we get from subtracting the squared dot product of the current RBG vector from the codeword’s RCB vector divided by the magnitude of the codeword’s RGB vector from the squared magnitude of the pixel’s RGB vector. Likewise, if brightness is calculated by checking if the current pixel’s RGB vector sum is in the range of low and high brightness values. The low brightness values is computed by multiplying the codeword’s maximum brightness by some factor alpha. We choose the value of alpha, but it’s typically between 0.4 and 0.7 (but always less than 1). Likewise, the high brightness is computed by choosing the minimum values of either beta multiplied by the codeword’s maximum brightness value (where beta is greater than 1) or the codeword’s minimum brightness is divided by alpha. If the pixel’s brightness falls into this range then we return true, otherwise false. Even though we finished constructing the codebook, we still need to do one more thing. What’s left to do before moving on to trimming the codebooks is to adjust the MNRL of each codeword (discussed in the next paragraph). Therefore we need to calculate the MNRL to the end of the video and from the beginning to the first time the codeword was created. Once we have these two values, we sum them and replace the current MNRL value with the newly computed one.

Now that the majority of the hard and time-consuming work is done, we need to trim the codebooks since some codebooks may have an unnecessary number of codewords. This is where the idea of Maximum Negative Run Length comes in. Maximum Negative Run Length is the longest interval in which the specific codeword has not recurred. We can trim the MNRL by checking to see if it is greater than or equal to the number of frames by two and removing the corresponding codeword if this condition is true. If we use half the number of frames as the threshold then we’re removing codewords that only appear at most, half the video. Thus, if a pixel doesn’t appear too frequently it must be a foreground pixel. Once we’ve removed the unnecessary codewords, we can may finally begin the background subtraction process. This process is much simpler than the codebook construction.

To perform background subtraction, we iterate over each pixel in each frame. For each pixel, just like before, we check to see if there is a match in the pixel’s corresponding ceodebook. We check if there is a match by using color distortion and the brightness measurement. If there is a match then that means that the pixel belongs to a foreground object. If there is not a match, then that means that we trimmed this from the codebook (because it doesn’t appear often enough). Therefore this pixel belongs to a background object. For each frame, we can construct a binary foreground image where foreground pixels are white and background pixels are black.

Now that we have the foreground images, there is likely to be some noise in the output. By noise, I mean some pixels that should not be considered foreground pixels were classified foreground pixels. To clean the foreground images, we need to perform some morphological operations. The morphological operation I chose to use was closing of the opening with discs of radius 1 for opening and radius 5 for closing for the structuring elements. The reason I am using this operation is because opening will get rid of bridges between pixels while closing will fill them in if they are small enough. By using these two together I can get rid of pixels by using opening, but will lose some pixels foreground pixels on foreground objects. In an attempt to recover the lost data, I use closing to fill in foreground objects. None of the more elementary operations (dilation, erosion, etc.) alone will work since they either only expand an objects range or reduce it.

**The Results**

Since my output was in AVI format, I will only discuss my results here. Please look in the zip file for the AVI files. The videos are labeled with their corresponding input e1, e2, alpha, and beta inputs.

While testing the outputs of my code, I found that empirically the thresholds 20 for e1 and 42 for e2 work best. The other values I used were 30 and 40 for e1 and 50 for e2. I would have tried a greater range of values, but to get a good a good result I needed to use a greater number of frames. However, using a greater number of frames greatly increases run time. Therefore I only tested these values.

The reason I tested [20, 30, 40] for e1 and [42, 50] for e2 is because for e1 I wanted the restriction to be looser than the restriction for e2. This way we could build a greater number of codewords to match to a pixel. The greater the number of codewords, the more certain we can be of our classification of foreground and background pixels. The reason I chose e2 to be higher than e1 is to not let in as much in BGS as we did in codebook construction, but not to exclude everything. This way we can balance between being strict and being lenient.

I found that all the values I tested gave relatively consistent results – {e1:20, e2:40} may have performed slightly better. This may be because the color distortion values were either much greater or much less than my chosen thresholds. The reason the color distortion values must have either been much greater or much smaller is because the range for e1 was 20.

There were still two more parameters that I had to pass to the code. These two variables were alpha and beta (when computing brightness for codeword matching). The paper suggested that when choosing a value for alpha, we stay between 0.4 and 0.7 and when choosing a value for beta we stay between 1.1 and 1.5. Like my results from choosing e1 and e2, my findings were relatively consistent across the different values of alpha and beta within these regions. To make sure that the low brightness threshold wasn’t too low, I typically chose 0.5 and to ensure that the high brightness threshold wasn’t too high, I typically chose 1.3. These values of alpha and beta may have given slightly better results than the other values, however the delta between the outputs are not too noticeable. I believe the reason that wasn’t too much difference in the outputs of these different alphas and betas is because there wasn’t too much variance in the brightness values of the pixels. They were relatively consistent throughout the frames (this makes sense since most frames were just the background).