Two Approaches to Background Separation

Nathaniel Guy and Ashwin T.A.N

Abstract— **This paper does a comparative evaluation of two different background separation/subtraction algorithms: the Mixture of Gaussians algorithm and an RPCA-based algorithm. We begin by examining the problem of background separation and its common applications, and then explain the theoretical foundations behind each of the two algorithms. We then implement each of the algorithms and run them on a standard video dataset to assess the results. Comparing them to ground truth data, we find a slight increase in separation accuracy for the RPCA-based algorithm, at the expense of runtime performance. We finish by giving recommendations for which algorithm to use, depending on needs and performance constraints.**

I. BACKGROUND SEPARATION: AN INTRODUCTION

Given a video sequence, the aim of background separation is to separate out the moving objects (called the foreground) from the static scene (background). Consider, for example, traffic surveillance footage from a busy intersection. We might be interested in finding out how many moving objects of each type (bicycles, trucks, cars, and pedestrians) are present at a given time, when the intersection is busiest, in what direction most of the vehicles are traveling, etc. The first step in answering all of these questions is to produce a video sequence containing only the foreground information. The general strategy is to construct a model for the background from the video sequence and then recover the foreground by thresholding the difference between each frame and the background model. See Fig. 1.

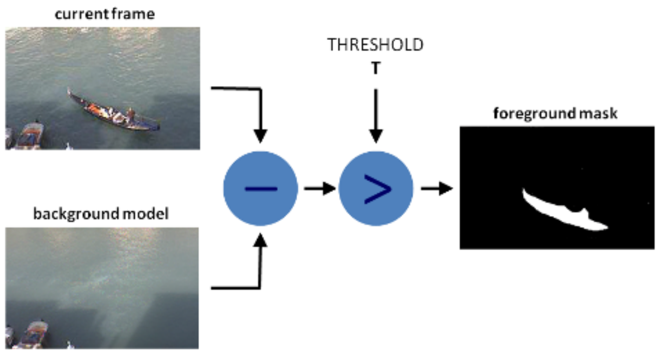

Fig. 1: An example flow for the process of background subtraction. Source: http://docs.opencv.org/3.1.0/Background_Subtraction_Tutorial_Scheme.png

In most real world applications, it is not possible to assume that the background is static. In our traffic surveillance example, there will be changes in illumination over the course of a day. A good background separation algorithm should be able to recognize waves on the surface of a water body, rain, trees swaying in the wind, fluttering flags and other such regular variations as part of the background. In this project, we considered and compared two algorithms to solve this problem: the first models each pixel as a mixture of Gaussians and the second is based on Robust Principal Component Analysis (RPCA).

II. APPROACH 1: GAUSSIAN MIXTURE MODEL

Suppose we are working with a grayscale video. Each pixel on the screen assumes varying scalar values over time. For example, if the color of each pixel is coded by 8 bits, the values are between 0-255, with 0 corresponding to completely black and 255 corresponding to completely white. Suppose we have a probability density function (PDF) that describes the values of a pixel over time. The idea behind the Gaussian mixture model approach is that foreground is transient, while background is persistent. Therefore, the pixel values which are more probable must be part of the background, while the less probable pixel values constitute the foreground. For videos with a red-green-blue color scheme, we have separate PDFs for each of the red, green and blue channels, with the more probable combinations corresponding to the background.

This raises the question of what kind of probability distribution function should we use to model the background. As we mentioned before, most backgrounds in real applications are dynamic, and therefore, unimodal distributions are not good at modeling them. Let's consider the following example of the pixel values corresponding to a region on the water surface. We'll look at an image like that shown in Fig. 2. This image's red-green scatterplot is shown in Fig. 3.

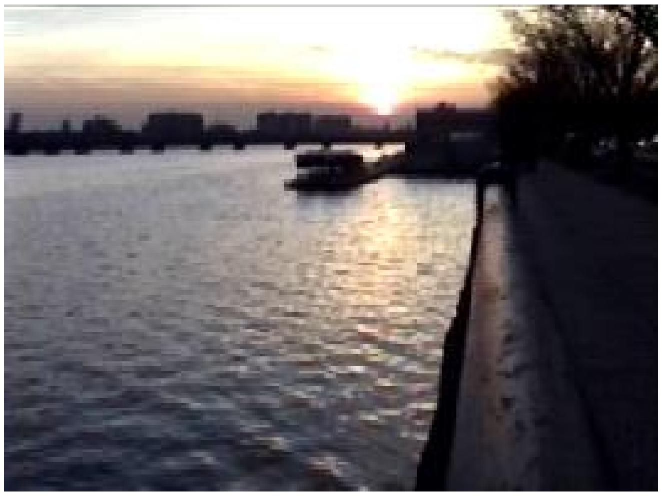

Fig. 2: A sample screenshot from a video of the surface of a body of water.

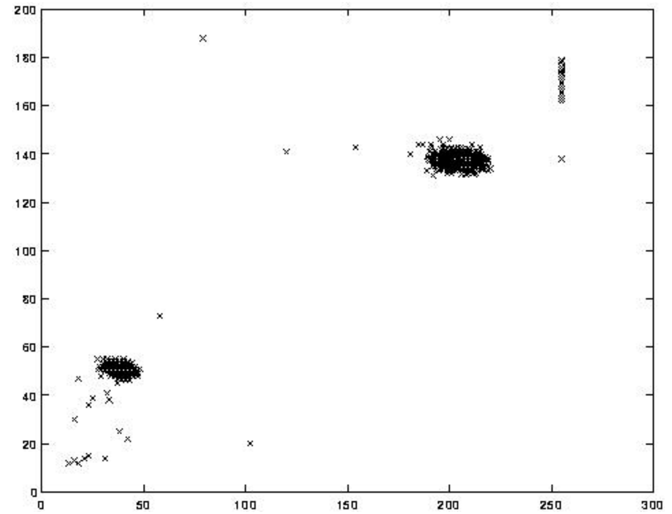

Fig. 3: The red-green scatterplot for a single pixel corresponding to a region on the water's surface. Source: http://www.ai.mit.edu/projects/vsam/Tracking/

In this example, we'll assume that the PDF is a mixture of a fixed number k of Gaussians. Let X_t denote the value of a particular pixel X at time t. Its PDF at time t is given by a mixture of k Gaussians $N(\mu_{1,t}, \sigma_{1,t}), \dots N(\mu_{k,t}, \sigma_{k,t})$ with means $\mu_{i,t}$ and variances $\sigma^2_{i,t}$. The weights of these Gaussians are denoted by $w_{i,t}$. The means, variances and relative weights are updated at each time step based on the value of X_t. In order to make the algorithm adaptive, the recent values of X_t are weighted more heavily.

A Gaussian is said to describe an observation *well* if the observation is within 2.5 standard deviations of the mean. At time t, we find the Gaussian with maximum weight that describes X_t well. If no such Gaussian exists, we replace the least likely Gaussian with a new Gaussian with X_t as its mean and an initially high variance. The weights of the Gaussians are readjusted as follows:

$$w_{i,t} = (1-\alpha) w_{i,t-1} + \alpha M_{i,t}$$

where α is a parameter called the learning rate and $M_{i,t} = 1$ for the model i that matched and 0 for all others. Note that if $N(\mu_{i,t}, \sigma_{i,t})$ is the new Gaussian introduced in this step because all the pre-existing Gaussians failed to match, we have $w_{i,t-1} = 0$ in the above formula. The weights are then renormalized to make their sum equal to 1. The means and variances of unmatched Gaussians remain the same. For the Gaussian $N(\mu_{i,t}, \sigma_{i,t})$ that matched, the parameters are updated as follows:

$$\begin{aligned} \mu_{i,t} &= (1-\rho)\mu_{i,t-1} + \rho X_t, \\ \sigma^2_{i,t} &= (1-\rho)\sigma^2_{i,t-1} + \rho (X_t - \mu_{i,t})^2, \end{aligned}$$

where

$$\rho = \alpha \eta (X_t | \mu_{i,t}, \sigma_{i,t}).$$

Here $\eta(\cdot|\mu, \sigma)$ is the Gaussian PDF with mean μ and variance σ^2.

Next, we declare each Gaussian to be background or foreground by requiring that a certain minimum portion of the data must be accounted for by the background Gaussians. The Gaussians are arranged in descending order of w. We declare the first B Gaussians to be part of the background where

$$B = \arg\ \min_b \left\{ \sum_i^b w_i > T \right\}.$$

Here T is a parameter between 0 and 1, representing the minimum portion of the data that should be accounted for by the background. If the pixel value X_t is described by any of these B distributions, it is declared to be part of the background.

III. Approach 2: Robust PCA

The second approach to the background subtraction problem is via Principal Component Analysis. Let us arrange the pixel values in a video into a matrix as follows: arrange the pixels on the screen in some order, and let X be the matrix whose entry in the ith row and jth column is the value of the pixel i at the jth time step. Thus, X has as many rows as the number of pixels on the screen and as many columns as the number of frames in the video. Each row represents the changing values of a single pixel over time while each column represents the values of all pixels in a given frame. We may imagine that the video can be decomposed as $X = L + S$ where L is the background video and S is the foreground video. Since the background is relatively persistent, it is reasonable to assume that most of the columns of L are linear combinations of a small number of column vectors representing the most probable background images. In other words, L must be of low rank. The matrix S must be small (in some sense) since the foreground is transient. Therefore, L is a *low-rank approximation* to X.

The standard Principal Component Analysis algorithm can be used to compute a low-rank approximation L to X. However, this technique is sensitive to large errors. In our case, the error matrix S representing the foreground need not have small entries. However, since the foreground is transient, it is reasonable to assume that it is sparse. This leads us to a formulation of our problem as an optimization problem:

$$\arg\ \min_{L,S} \{\text{rank } L + \lambda \|S\|_0 |\}, L + S = X.$$

Here $\|S\|_0$ is the number of non-zero entries of S and λ is a parameter, measuring the relative importance of making S sparse and reducing the rank of L. However, neither the rank function nor the zero-norm $\|\cdot\|_0$ is a convex function and consequently this problem is NP-hard. So we may try to find a convex approximation to the objective function.

Given that the entries of X are bounded, the rank of X is well-approximated by its nuclear norm $\|X\|_*$ (that is, the sum of the singular values of X). Similarly, the $\|S\|_0$ is well-approximated by $\|S\|_1$. Therefore, we may instead try to solve the following convex optimization problem:

$$\arg\ \min_{L,S} \{\|L\|_* + \lambda \|S\|_1 |\}, L + S = X.$$

Since the entries of X are discrete, the equality constraint $L + S = X$ might be hard to satisfy. Therefore, we may relax the constraint to an inequality $\|L + S - X\|_\infty < \alpha$ to obtain another formulation of the problem:

$$\arg \min_{L,S} \{\|L\|_* + \lambda \|S\|_1 |\}, \|L + S - X\|_\infty < \alpha.$$

Both of these optimization problems are easy to implement using TFOCS (Templates for First Order Conic Solvers), a Matlab template written by Stephen Becker, Emmanuel J. Candès and Michael Grant.

IV. RESULTS

In this section, we will present the results of implementing both of these algorithms and applying them to a typical video dataset for background subtraction.

A. SAMPLE TEST DATA

Before implementing the two algorithms examined in this paper, we needed a video dataset on which to run our algorithms for test purposes. We discovered the video dataset of the Institute for Infocomm Research (I2R), which is used as a standard video corpus for background separation algorithm evaluation in much of the literature we encountered. This video corpus includes many "surveillance"-type videos, with steady backgrounds against which lots of (mostly human) actors are moving rapidly and unpredictably. These human "foreground objects" exhibit large differences in color and trajectory, and there are many instances of them stopping to wait or interact with the environment, which can specifically test an adaptive algorithm's behavior; adaptive algorithms such as Mixture of Gaussians may make the determination that a stopped foreground object has become part of the background, and as such, these objects will be inaccurately classified as part of the background while they're stopped.

From the above-mentioned data set, we narrowed down our dataset to a single specific video, due to time constraints. This video, a surveillance camera feed from a shopping mall, exhibited the above-mentioned characteristics that we sought for testing. We reduced the resolution of the source video due to runtime constraints with the RPCA TFOCS Solver algorithm; the source video on which we operated had a resolution of 160 x 120 pixels, and consisted of 1,286 RGB frames.

A sample screenshot from this video can be seen in Fig. 4.

B. APPLYING MIXTURE OF GAUSSIANS

In order to apply the Mixture of Gaussians, we utilized the Python implementation of the OpenCV library (version 2) implementation. This well-tested implementation made it simple to get started processing video data for background/-foreground separation.

Using OpenCV, we loaded the shopping mall video data, and as a first step (in order to reduce complexity), converted it to grayscale using standard color channel weights. We then used OpenCV to build a Gaussian Mixture Model from the grayscale intensity values in each frame, creating a binary "foreground mask" for each frame–essentially a matrix $M \in \mathbb{R}^{x \times y}$ where $M_{ij} = 1$ when the pixel at coordinate (i, j) is a foreground pixel, and $M_{ij} = 0$ when the pixel at coordinate (i, j) is a background pixel.

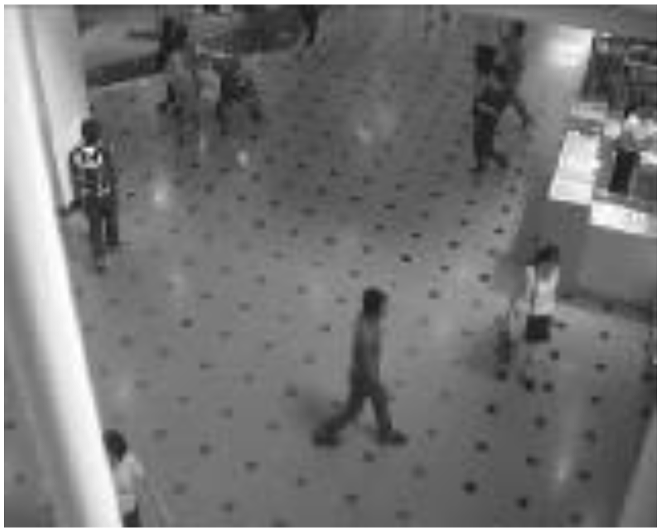
Fig. 4: A shot from the original "shopping mall" video dataset, with foreground and background intact.

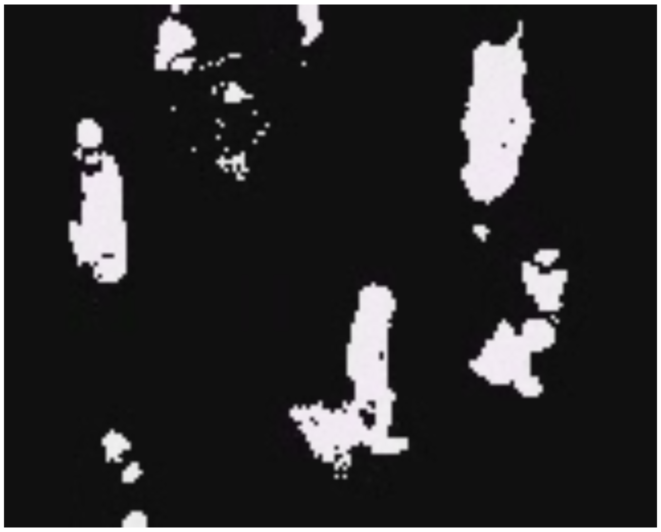
Fig. 5: A shot after the application of Mixtures of Gaussians background subtraction, showing only the foreground. This corresponds to the same shot shown in Fig. 4.

Applying this mask using a bitwise AND operator allowed us to produce an image of the foreground pixels only, which were set to solid white in order to stand out. This produced a video composed of images like that shown in Fig. 5. We were then able to recombine these masked frames in order to form a video consisting of the foreground only. We elected not to construct a background-only reconstruction due to time constraints; however, it should be noted that this is possible to do by the following algorithm:

1) For each frame in the video, read in the frame as a

matrix of pixels $M \in \mathbb{R}^{x \times y}$.

2) For every row i and column j, evaluate the pixel color (or intensity, in a grayscale image) at M_{ij}, and determine via the Mixture of Gaussians algorithm to which Gaussian distribution this pixel belongs.
3) After an associated distribution has been determined, find the distribution's mean color (or intensity) at this pixel μ_{ij} and set the value in a new matrix $P \in \mathbb{R}^{x \times y} = \mu_{ij}$.
4) Reconstruct the video the various mean color matrices P.

After reconstructing the foreground, we used the ground truth data provided by I2R to test the accuracy of the Mixture of Gaussians algorithm. We did a pixel-wise comparison on the masked frames we had generated with I2R's ground truth frames, and recorded the percentage which matched. Doing so, we found that Mixture of Gaussians had a 90.7% accuracy for this data.

C. *APPLYING RPCA USING TFOCS SOLVERS*

We next moved on to apply the RPCA formulation using the TFOCS solver template in MATLAB. In order to do so, we first loaded the I2R shopping mall video data, and reduced each frame to grayscale, similar to the preprocessing performed for the application of the Mixture of Gaussians algorithm. Each frame's pixels were vectorized and then concatenated into a matrix $M \in \mathbb{R}^{xy \times n}$ (where n is frame count).

After this, we were able to model the foreground/background separation as a convex optimization problem, and solved iteratively for a low-rank background matrix L and a sparse foreground matrix S such that $\min_{L,S} \|L\|_* + \lambda\|S\|_1$ and $X = L + S$ or $\|L + S - X\|_\infty \leq 0.5$ (we performed the optimization using both the equality constraint and the inequality constraint, but the results were very similar, so we will present only the results from the inequality constraint optimization here). λ was hand-tuned for optimal separation.

The inequality-constrained optimization converged in 56 iterations using the shopping mall video data. It produced a low-rank background matrix L with a rank of 140 (of a possible 1,286), and a sparse foreground matrix with 4.7% non-zero elements. (Note that this required slight thresholding, since the l_1 constraint on S only optimizes towards a pseudo-sparse matrix with lots of very small elements, as opposed to the non-convex l_0 alternative, which would produce actual sparsity after optimization. However, after reducing all elements below a very low threshold of 0.1 to 0, S retained only 4.7% non-zero elements.)

After the optimization was complete, we converted both L and S back into videos and observed them. L showed an eerily accurate reconstruction of the scene's background, with some minor ghosting of pixels from humans who hadn't been 100% removed from the feed. S showed the complement of L, but was more different to view, due to having many negative intensity values (due to pixel intensities that must be added to background intensities to equal certain low intensities in the source video). A screenshot from L is shown in 6.

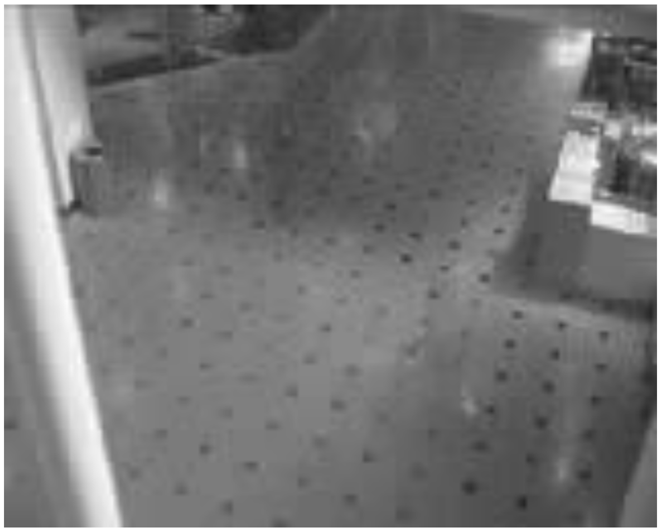
Fig. 6: A shot after the application of RPCA TFOCS solver background subtraction, showing only the background. This corresponds to the same shot shown in Fig. 4.

To better view the foreground determination by the RPCA algorithm, and to easily compare it to the ground-truth and the results from the MoG algorithm, we converted the RPCA foreground signal for each frame to a binary mask $M \in \mathbb{R}$, where $M_{ij} = 1$ if $S_{ij} \geq \epsilon$ and $M_{ij} = 0$ if $S_{ij} < \epsilon$, where ϵ represents a small threshold. This allowed us to reconstruct a masked foreground video similar to what we did with the Mixture of Gaussians results. A screenshot from this video is shown in Fig. 7.

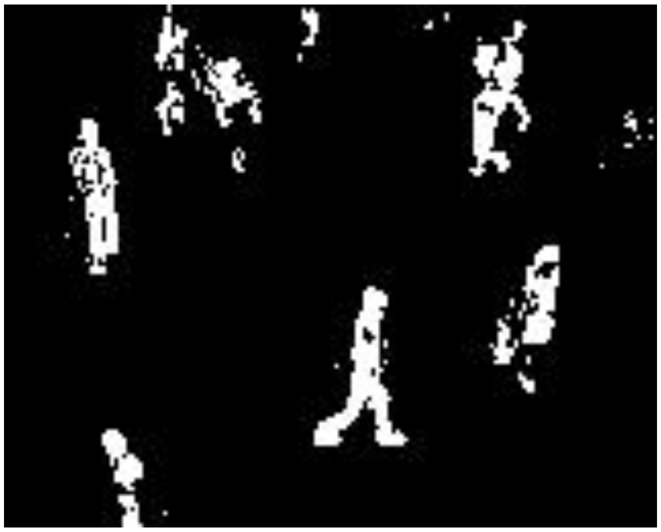
Fig. 7: A shot after the application of RPCA TFOCS solver background subtraction, showing only the foreground. This corresponds to the same shot shown in Fig. 4.

After reconstructing a masked foreground video, the next step was to do a full comparison with the ground truth pixels, via the same technique that we used with the Mixture of Gaussians results. In doing so, we found an accuracy of 93.0%, a little over 2% better than the Mixture of Gaussians results. However, qualitatively, the results from RPCA seemed even better than this. The reason seems to be related to the specific pixels of the foreground which are captured; RPCA seems to have occasional problems with accurately capturing internal pixels within the outlines of the people walking through the scene, but captured the outlines themselves quite cleanly, which the Mixture of Gaussians algorithm mostly failed to do. Either way, RPCA's results were visually satisfying, and the ground truth analysis bore this out.

D. COMPARISONS

It's useful to do a brief comparison of the two algorithms that we examined in this paper. In our ground truth tests on the shopping mall dataset, the Mixture of Gaussians algorithm gave an accuracy of 90.7%, while the RPCA algorithm gave a 93% accuracy. There were differences in how they handled noise, and the quality of their separations, as well; the Mixture of Gaussians algorithm had more obvious noise around the silhouettes of people walking, while the RPCA algorithm had less noise on the outside of silhouettes, with more noise on the insides of their outlines. To our eyes, the RPCA algorithm seems to make it easier to pick out human figures in the foreground, but both algorithms are effective for this purpose.

The two algorithms also have different approaches for how the background can be reconstructed after the fact. With the Mixture of Gaussians algorithm (though we did not employ this method), the background could be reconstructed by using the mean, for each pixel on each frame, of the most probable Gaussian for that pixel. In contrast, as we discussed earlier, the background reconstruction falls naturally out of the optimization, as the low-rank signal L.

It is worth noting that the runtimes between the two algorithms we studied are *drastically* different. There is an order of 10^2 difference; on an Intel i5 with 8 GB of RAM, the Mixture of Gaussians algorithm took 5 seconds to process 1,286 grayscale frames at 160 x 120 pixels; the RPCA with TFOCS solver took 1,987 seconds to process the same data. The bottleneck for RPCA seems to be the Singular Value Decomposition which must be performed in $\|L\|_*$, leading to a worst-case runtime of $O(3xyn^2)$, where xy is the number of pixels and n is the number of frames. In addition, RPCA requires very large amounts of memory to do this calculation, resulting in a full usage of 8 GB of RAM for larger image resolutions that we tested.

For these reasons, we conclude that Mixture of Gaussians is far-and-away the superior choice for real-time background subtraction, due to its speed and memory efficiency. Mixture of Gaussians would also be preferred for any sort of environment in which computational resources are constrained. For analyses after-the-fact, however, where computing resources are in excess, RPCA produces more visually pleasing results which score slightly better on ground truth comparisons, and we recommend its use.

V. CONCLUSION

In this paper, we introduced the problem of background separation for video data, including its applications, its challenges, and how the idea of "foreground" and "background" is usually modeled theoretically. We looked into the details of the Mixture of Gaussians background subtraction algorithm as well as the RPCA optimization formulation using the TFOCS solver. After discussing the theory behind both of these techniques, we introduced a sample video dataset, and applied both of these techniques as specified in order to separate the background and foreground in the video data. Finally, we compared our results by looking at quantitative metrics (percentage accuracy compared to ground-truth data) as well as qualitative ones. We showed that, in our tests, RPCA was able to get a 2% improvement in ground truth accuracy, and presented a qualitatively more pleasing separation of foreground and background. It was also able to successfully classify as background certain foreground objects that stopped moving for extended periods of time. However, as we showed, the run-time characteristics of the convex RPCA formulation are such that it takes several orders of magnitude longer to run than the Mixture of Gaussians implementation; thus, only Mixture of Gaussians is appropriate for real-time applications.

VI. ACKNOWLEDGMENTS

We'd like to thank Dr. Sasha Aravkin for his help with this course. We'd also like to thank the creators of the TFOCS package for their excellent writeup about convex optimization for background subtraction.

REFERENCES

[1] Charles Guyon, Thierry Bouwmans and El-hadi Zahzah (2012) Robust Principal Component Analysis for Background Subtraction: Systematic Evaluation and Comparative Analysis *Principal Component Analysis*, ISBN: 978-953-51-0195-6.

[2] Chris Stauffer and W.E.L Grimson (1999) Adaptive background mixture models for real-time tracking *Proceedings IEEE Conf. on Computer Vision and Pattern Recognition*, pp. 246-252, 1999.

[3] Jonathon Shlens (2005) A Tutorial on Principal Component Analysis `www.cs.cmu.edu/~elaw/papers/pca.pdf`.

[4] Becker, Stephen R and Candès, Emmanuel J and Grant, Michael C (2011) Templates for convex cone problems with applications to sparse signal recovery *Mathematical programming computation 3.3*, pp. 165-218, 2011.

[5] Zivkovic, Zoran (2004) Improved adaptive Gaussian mixture model for background subtraction *Pattern Recognition, 2004. ICPR 2004. Proceedings of the 17th International Conference on.*, Vol. 2. IEEE, pp. 28-31, 2004.

[6] Zivkovic, Zoran and van der Heijden, Ferdinand (2006) Efficient adaptive density estimation per image pixel for the task of background subtraction *Pattern recognition letters*, 27(7), pp.773-780, 2006.

VII. CODE

A. Mixture of Gaussians Background Subtraction (Python)

```
import cv2

# Get a frame from the current video source
def getFrame(cap):
    _, frame = cap.read()
    return frame

FPS = 25.0

videoFilename = 'reassembled_all_frames.avi'

# Get a camera input source
cap = cv2.VideoCapture(videoFilename)

# Set up background subtractor
subtractor = cv2.createBackgroundSubtractorMOG2(detectShadows = False)

# Get video output sinks
fourcc1 = cv2.VideoWriter_fourcc(*'DIVX')
out = cv2.VideoWriter('ShoppingMallThresholdedMoG.avi', fourcc1, FPS, (160,128))

while(cap.isOpened()):
    frame = getFrame(cap)
    if frame is None:
        break

    # Convert to grayscale
    gray_image = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    # Blur frame for processing
    blurred = cv2.blur(gray_image, (4,4))

    # Apply background subtraction to get a mask
    fgmask = subtractor.apply(blurred)

    masked_frame = cv2.bitwise_and(frame, frame, mask = fgmask)

    # Convert foreground to RGB for proper video renderin
    masked_rgb = cv2.cvtColor(fgmask, cv2.COLOR_GRAY2BGR)

    out.write(masked_rgb)

# Release everything if job is finished
cap.release()
out.release()
cv2.destroyAllWindows()
```

B. RPCA with TFOCS Solver Template Background Separation (MATLAB)

```
% Initial cleanup
clear; close all; clc;

% Load video
input_video = 'reassembled_all_frames.avi';
video_object = VideoReader(input_video);
nFrames = video_object.NumberOfFrames;

nPixels = video_object.Width * video_object.Height;

% Make an (nPixels*3 x nFrames)-sized matrix to store all of the frame data
% (We'll flatten color channels as well to simplify things)
video_frames = zeros(nPixels, nFrames);

% Dump each frame's content into the pre-allocated matrix
for i = 1:nFrames,
    frame = read(video_object, i);

    video_frames(:, i) = reshape(rgb2gray(frame), nPixels, 1);
end

% Convert image data into a matrix of doubles
X = double(video_frames);

% (Optimization template code below is copied from RPCA TFOCS demo)

% addpath('.\TFOCS-1.4');    % add TFOCS to mypath

nFrames      = size(X,2);

lambda  = 5e-4; % Tried: 1e-2 (too little sparse contribution), 1e-3 (quite good
    actually), 1e-4 (even better)

opts = [];
opts.stopCrit       = 4;
opts.printEvery     = 1;
opts.tol            = 1e-5;

opts.maxIts         = 100;

opts.errFcn{1}      = @(f,d,p) norm(p{1}+p{2}-X,'fro')/norm(X,'fro');

largescale      = false;

for inequality_constraints = 0:1

    if inequality_constraints
        % if we already have equality constraint solution,
        %   it would make sense to "warm-start":
        %         x0      = { LL_0, SS_0 };
        % but it's more fair to start all over:
        x0      = { X, zeros(size(X))   };
        z0      = [];
    else
        x0      = { X, zeros(size(X))   };
        z0      = [];
    end


    obj    = { prox_nuclear(1,largescale), prox_l1(lambda) };
    affine = { 1, 1, -X };

    mu = 1e-4;
    if inequality_constraints
        epsilon  = 0.5;
        dualProx = prox_l1(epsilon);
    else
        dualProx = proj_Rn;
    end

    tic
    % call the TFOCS solver:
    [x,out,optsOut] = tfocs_SCD( obj, affine, dualProx, mu, x0, z0, opts);
    toc

    % save the variables
    LL =x{1};
    SS =x{2};
    if ~inequality_constraints
        z0      = out.dual;
        LL_0    = LL;
        SS_0    = SS;
    end

end % end loop over "inequality_constraints" variable

width = 160;
height = 128;
mat  = @(x) reshape(x, [height, width]);
figure();
colormap('Gray');

% Scale matrices to [0.0, 1.0] range for display
X = X / 255.0;
SS = SS / 255.0;
SS_0 = SS_0 / 255.0;
LL = LL / 255.0;
LL_0 = LL_0 / 255.0;

% Also, clamp to the above range, just in case there were weird errors...
X = max(min(X, 1), 0);
SS = max(min(SS, 1), 0);
SS_0 = max(min(SS_0, 1), 0);
LL = max(min(LL, 1), 0);
LL_0 = max(min(LL_0, 1), 0);

% Output videos
writer = VideoWriter('original_bw.avi');
open(writer);
for k = 1:nFrames
    writeVideo(writer, mat(X(:,k)));
end
close(writer);
```

```
writer = VideoWriter('low_rank_unconstrained_bw.avi');
open(writer);
for k = 1:nFrames
    writeVideo(writer, mat(LL_0(:,k)));
end
close(writer);

writer = VideoWriter('sparse_unconstrained_bw.avi');
open(writer);
for k = 1:nFrames
    writeVideo(writer, mat(SS_0(:,k)));
end
close(writer);

writer = VideoWriter('low_rank_constrained_bw.avi');
open(writer);
for k = 1:nFrames
    writeVideo(writer, mat(LL(:,k)));
end
close(writer);

writer = VideoWriter('sparse_constrained_bw.avi');
open(writer);
for k = 1:nFrames
    writeVideo(writer, mat(SS(:,k)));
end
close(writer);

writer = VideoWriter('reconstructed_constrained_bw.avi');
open(writer);
for k = 1:nFrames
    writeVideo(writer, mat(SS(:,k)) + mat(LL(:,k)));
end
close(writer);

writer = VideoWriter('reconstructed_unconstrained_bw.avi');
open(writer);
for k = 1:nFrames
    writeVideo(writer, mat(SS_0(:,k)) + mat(LL_0(:,k)));
end
close(writer);

% Make copies for zeroing out
SS_to_white = SS;
SS_0_to_white = SS_0;

% Convert to solid white for non-zero entries
for i = 1:size(SS_to_white, 1)
    for j = 1:size(SS_to_white, 2)
        if ((SS_to_white(i, j) < -0.09) || (SS_to_white(i, j) > 0.09))
           SS_to_white(i, j) = 1.0;
        else
            SS_to_white(i, j) = 0.0;
        end
    end
end

% Convert to solid white for non-zero entries
```

```
for i = 1:size(SS_0_to_white, 1)
    for j = 1:size(SS_0_to_white, 2)
        if ((SS_0_to_white(i, j) < -0.09) || (SS_0_to_white(i, j) > 0.09))
            SS_0_to_white(i, j) = 1.0;
        else
            SS_0_to_white(i, j) = 0.0;
        end
    end
end

% Dump these to videos, too
writer = VideoWriter('sparse_constrained_bw_thresholded.avi');
open(writer);
for k = 1:nFrames
    writeVideo(writer, mat(SS_to_white(:,k)));
end
close(writer);

writer = VideoWriter('sparse_unconstrained_bw_thresholded.avi');
open(writer);
for k = 1:nFrames
    writeVideo(writer, mat(SS_0_to_white(:,k)));
end
close(writer);

% (Code below is copied from RPCA TFOCS demo)

%% Is there a difference between the two versions?
% Compare the equality constrained version and the inequality
%   constrained version.  To do this, we can check whether
%   the "L" components are really low rank
%   and whether the "S" components are really sparse.
% Even if the two versions appear visually similar, the variables
%   may behave quite differently with respect to low-rankness
%   and sparsity.

fprintf('"S" from equality constrained version has  \t%.1f%% nonzero entries\n',...
    100*nnz(SS_0)/numel(SS_0) );
fprintf('"S" from inequality constrained version has  \t%.1f%% nonzero entries\n'
    ,...
    100*nnz(SS)/numel(SS) );

s  = svd(LL);    % inequality constraints
s0 = svd(LL_0);  % equality constraints

fprintf('"L" from equality constrained version has numerical rank\t %d (of %d
   possible)\n',...
    sum( s0>1e-6), min( width*height, nFrames)  );
fprintf('"L" from inequality constrained version has numerical rank\t %d (of %d
   possible)\n',...
    sum( s > 1e-6), min( width*height, nFrames)  );
```

C. Code for Calculating Comparison Scores to Ground Truth (Python)

```python
import cv2
import numpy as np
import time

def getImageDiff(img1, img2):
    diff = np.absolute(img1 - img2) / 255.0
    return (img1.size - diff.sum()) / img1.size

# Get a frame from the current video source
def getFrame(cap):
    _, frame = cap.read()
    return frame

FPS = 25.0

original_video_filename = 'reassembled_all_frames.avi'
# foreground_video_filename = 'ShoppingMallThresholdedMoG.avi'
foreground_video_filename = 'sparse_constrained_bw_thresholded.avi'
# foreground_video_filename = 'sparse_unconstrained_bw_thresholded.avi'

gt_original_prefix = 'ground_truth/ShoppingMall'
gt_original_suffix = '.bmp'

gt_prefix = 'ground_truth/gt_new_ShoppingMall'
gt_suffix = '.bmp'

frames_to_compare = [1433, 1535, 1553, 1581, 1606, 1649, 1672, 1740, 1750, 1761,
    1780, 1827, 1862, 1892, 1899, 1920, 1980, 2018, 2055, 2123]

# Get a camera input source
orig_cap = cv2.VideoCapture(original_video_filename)
fg_cap = cv2.VideoCapture(foreground_video_filename)

frameNo = 1000

scores = []

while(orig_cap.isOpened()):
    orig_frame = getFrame(orig_cap)
    fg_frame = getFrame(fg_cap)

    if orig_frame is None or fg_frame is None:
        break

    if frameNo in frames_to_compare:
        # Load gt frame
        gt_frame = cv2.imread(gt_prefix + str(frameNo) + gt_suffix, 0)

        fg_frame = cv2.cvtColor(fg_frame, cv2.COLOR_BGR2GRAY)

        # Compare to thresholding bgsub frame
        diff = getImageDiff(gt_frame, fg_frame)
        print float(diff)
        scores.append(float(diff))

    frameNo += 1
```

```
print "Average score: %s" % (100.0 * sum(scores) / len(scores))
# Release everything if job is finished
orig_cap.release()
fg_cap.release()
cv2.destroyAllWindows()
```