Robust PCA for Background Subtraction

MATH 123 Fall 2022
Maliheh Teimouri, Nirmiti Naik, Joseph Egan

Introduction

Since the 1990s, background subtraction has been attracting great attention in the fields of image processing and computer vision. As data becomes more complex, it is increasingly important that we can analyze it with greater accuracy to make meaningful observations. Data is not always free of outliers or corrupt observations, so it is imperative that we are able to analyze data with methods that are insensitive to such anomalous information. Proposed by Candes et al. in 2009 [2], Robust Principal Component Analysis (RPCA) is a method of low-rank approximation which is resistant to outliers that separates data into two components: a low-rank matrix and a sparse matrix. RPCA is an extension of Principal Component Analysis, a method of low-rank approximation which is susceptible to both outliers and corruption. Due to its resistance to noisy data, RPCA is commonly used for background or feature subtraction in various media like photographs or videos. It is useful in this domain since it can remove specific features (objects, foreground, background) from the media and leave behind objects of interest. In this project, RPCA was used to decompose a video and four images with various degrees of noise.

Applications

RPCA is especially useful for video surveillance since it allows for the backgrounds of videos to be removed so that the moving objects are isolated. In this case, the background would be captured by the low-rank matrix and the foreground would be captured by the sparse matrix. It is also a great approach for performing facial recognition; faces would be captured by the low-rank matrix and the noisy features (shadows, occlusions, etc.) would be captured by the sparse matrix. In a slightly different domain, RPCA can also be applied to Netflix rankings to reveal user ratings for unseen movies based on the movies that the user has rated. As video surveillance is used more and more often in different contexts (security and police investigations to name a few), it is ideal that it can be analyzed meaningfully so that meaningful conclusions can be drawn from it.

Modeling of RPCA

As mentioned, RPCA was initially developed by Candes et al. in their paper “Robust Principal Component Analysis?” [1]. In the paper, they proposed the decomposition of a matrix “X” consisting of data containing either outliers or corruption into two matrices: a structured low-rank matrix “L” and a sparse matrix “S”. The idea being that the L matrix would capture the variation in the uncorrupted data, on which we could perform PCA or other analyses on, and the S matrix would capture the outliers or corruptions.

Finding this decomposition is equivalent to this optimization problem:

$$\min_{L,S} rank(L) + \|S\|_0 \; subject\ to\ L + S = X$$

As we discussed in class with the compressive sensing problem, finding the 0-norm of a matrix is not computationally efficient. In fact, finding the rank and 0-norm of a matrix are non-convex problems. Fortunately, we can find an optimal solution with high probability by relaxing the optimization to:

$$\min_{L,S} \|L\|_* + \lambda\|S\|_1 \; subject\ to\ L + S = X$$

So instead, we are minimizing the nuclear norm of L and the product of the 1-norm of S times parameter λ. Since the 1-norm promotes sparsity, the second equation has been proven to converge to the solution to the original optimization with high probability if:

1. $\lambda = \frac{1}{sqrt(\max(n,m))}$ where n and m are the dimensions of X. It is a regularization parameter.
2. L is not sparse
3. S is not low-rank – the entries are distributed in such a way that they do not have low-dimensional column space

This optimization can be solved via augmented Lagrangian multipliers (ALM) in the form of:

$$\mathcal{L}(L,S,Y) = \|L\|_* + \lambda\|S\|_1 + \langle Y, X - L - S\rangle + \frac{\mu}{2}\|X - L - S\|_F^2$$

Where Y represents the Lagrangian multiplier matrix. Lagrange multiplier algorithm would solve PCP by repeatedly setting $(L_k, S_k) = \arg min_{L,S}\, l(L,S,Y_k)$, and then updating the Lagrange multiplier matrix via $Y_{k+1} = Y_k + \mu(M - L_k - S_k)$. For our low-rank and sparse decomposition problem, we can avoid having to solve a sequence of convex programs by recognizing that $min_L\, l(L,S,Y)$ and $min_S\, l(L,S,Y)$ both have very simple and efficient solutions. Now, we denote the shrinkage operator $S_\tau\,[x] = sgn(x)\max(|x| - \tau, 0)$, and extend it to matrices by applying it to each element [1]. It is easy to show that

$$arg\min_S l(L,S,Y) = S_{\lambda/\mu}(M - L + \mu^{-1}Y)$$

Similarly, for matrices X, denote the singular value thresholding operator given by

$$D_\tau(X) = US_\tau(\Sigma)V^*$$

where $X = U\Sigma V^*$, singular value decomposition. And low rank matrix:

$$arg\min_L l(L,S,Y) = D_{\lambda/\mu}(M - S + \mu^{-1}Y)$$

Thus, a more practical strategy is to first minimize l with respect to L (fixing S), then minimize l with respect to S (fixing L), and then finally update the Lagrange multiplier matrix Y based on the residual M−L−S, a strategy that is summarized as

1. Initializes L, S and Y
2. Initializes λ per the convergence requirement mentioned above, along with a parameter for shrinking, μ which is equal to $nm/4\|M\|_1$ and n and m are the size of image.
3. Iteratively shrinks L and S back and forth using a "shrink" function
    a. For S, this promotes sparsity in S itself
    b. For L, it is used to promote sparsity in the singular value matrix after using the SVD to decompose the resulting matrix (L)
4. This is repeated until either the number of iterations reaches a max count, or the Frobenius norm of the matrix representing the difference between our original matrix, X, and L + S falls below a threshold.

A full description of our implementation is available in code A.1 (Appendix). We terminate the algorithm when $\|M - L - S\|_F \leq \delta\|M\|_F$ with $\delta = 10^{-7}$ [1].

When complete, L will represent the background or 'regular' data, which can be analyzed without bias from the outliers and corruption.

Data

To demonstrate the efficacy of RPCA, we chose two datasets that had not been evaluated yet using this process: (a) a random video (we downloaded a mute video from YouTube as students walking across campus (submitted separately)) and (b) a face image with various types of noise, and applying. The purpose of using dataset (a) is to show how this method can separate videos into a low-rank L matrix representing the background and a sparse S matrix representing the foreground, in this case the campus and students, respectively. The purpose of using dataset (b) is to show that RPCA can separate a variety of types of noise and mask from the relevant content of an image. To introduce noise into the original image, the Imnoise function was used, and three types of noise were applied: salt and pepper, Gaussian and speckle. For the mask, we used texr.png from Matlab library.

Results

Video Data with Movement

For the video data, we had to convert a sequence of RGB matrices representing a single frame into one matrix, X. To do so, we first converted the colors to simply gray, and then reformatted the resulting matrix for each frame into an array. The gray colored arrays for each frame were then stacked into a single matrix, which we could then perform the RPCA algorithm on to separate out L and S.

As seen in the application of RPCA to dataset (a), we are left with two new videos: the low-rank L matrix video in the center and the sparse S matrix video on the right. The L matrix video has isolated the background and shows faint shadows of the students. The S matrix video has, for the most part, isolated the students and captures their movement throughout the video. These two matrices demonstrate RPCA's viability as a method for video surveillance and editing. Combining the L and S videos back together yields a nearly perfect recreation of the original video (Code A.4 (Appendix)).

Image Data with Noise

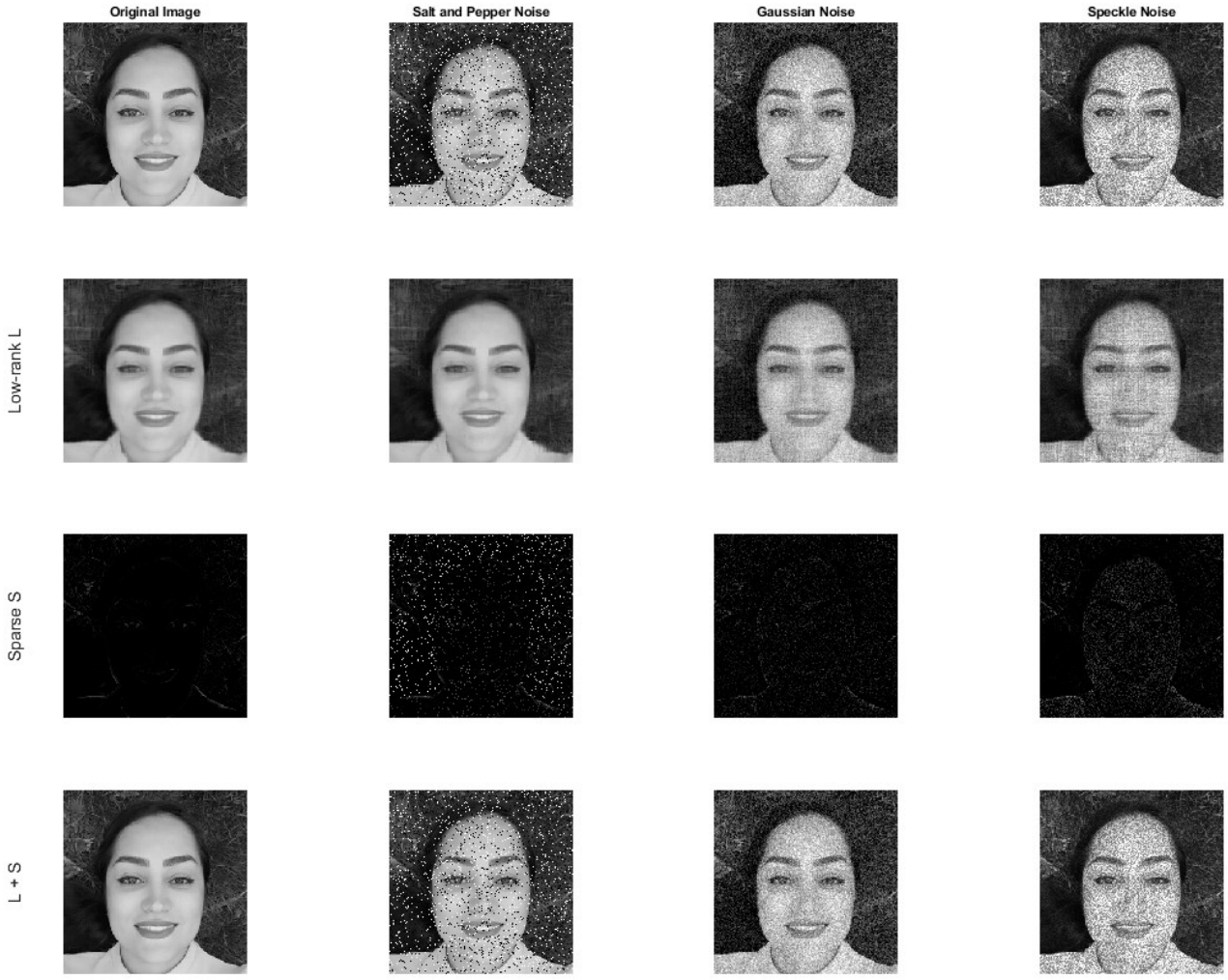

Figure 1 Results of RPCA on various applications of noise (Code A.2)

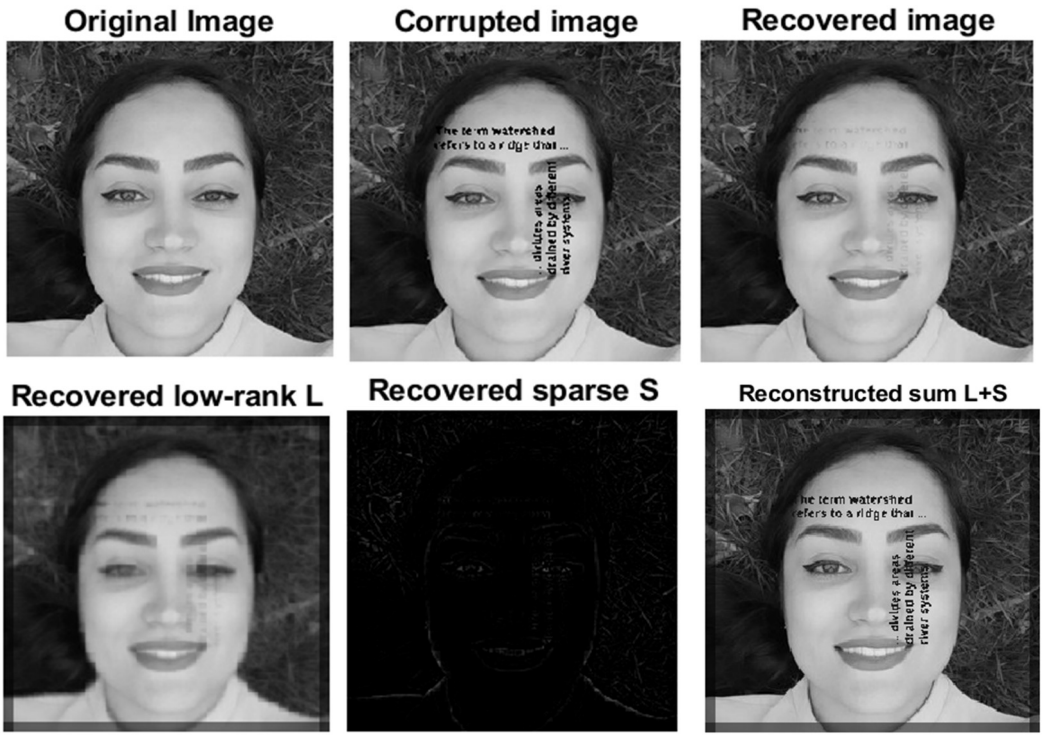

Figure 2 Results of RPCA in removing text (Code A.3)

As seen in the first set of images above (Figure 1), RPCA has varying levels of success in removing different types of noise from image data. Each of the four images in the top row (original, salt and pepper, Gaussian, speckle) are decomposed into an L matrix containing the face of the individual and an S matrix containing the different kinds of noise. It is interesting that the salt and pepper noise application produces the best low-rank L eigenface out of the corrupt images even though it visually looks like it has the most noise. This can be explained by the fact that the noise is mainly lighter pixels which occur on distinct areas of the face while the Gaussian and speckle noise are caused by mid-grayscale pixels which are present more uniformly in the image. Therefore, it is easier to isolate the more distinct salt and pepper outliers. By comparing RPCA's performance on different kinds of noise, one can see the limitations that RPCA might have in certain applications, and it can be helpful when determining if it is the appropriate algorithm to use. However, our results show that while the quality of success varies, overall, the RPCA is "robust". This is an important takeaway since a good model/algorithm should be able to do its job reasonably well regardless of how many outliers exist.

In Figure 2, we can see that RPCA could remove the text (mask) and show background and foreground image by sparse and low rank matrix. Both images, recovered image and recovered low-rank L, are the result of RPCA. However, because real images have NaN values which are non-physical values and in the function of RPCA we define NaN equal to zero, there is a discrepancy in the ascribed value to NaN and its true meaning. To prevent the effect of this non-physical value on the result, we copy the number of index equal to NaN from original image and

paste to the low rank matrix and then plot the final image. The result of reconstructed image after replacing NaN values is shown in the top left image in Figure 2 (Recovered Image). (Code A.2,3 (Appendix)).

Image Reconstruction vs Regular PCA

In this part we computed RPCA and PCA on both a color image and grey image as well. We compared how RPCA performs in image reconstruction compared to regular PCA. Regular PCA attempts to capture the variance of a dataset into an ‘r’ number of components. For image reconstruction, the value of r required to capture sufficient information important to the image can be idiosyncratic, and thus require some exploration to find an optimal level.

However, RPCA requires no such tuning, as the optimization already finds the corruption or outlier free data in a low rank matrix. We ran a comparison on our image between RPCA and PCA with 5, 10, and 15 components (the number of components is an input and user can enter the number of first PC’s). In the below images, the top row contains the original image, with fully colorized on the left, and grayscale in right and PCA and RPCA is computed for color and gray scale. A full description of the code is available in Code A.5 (Appendix).

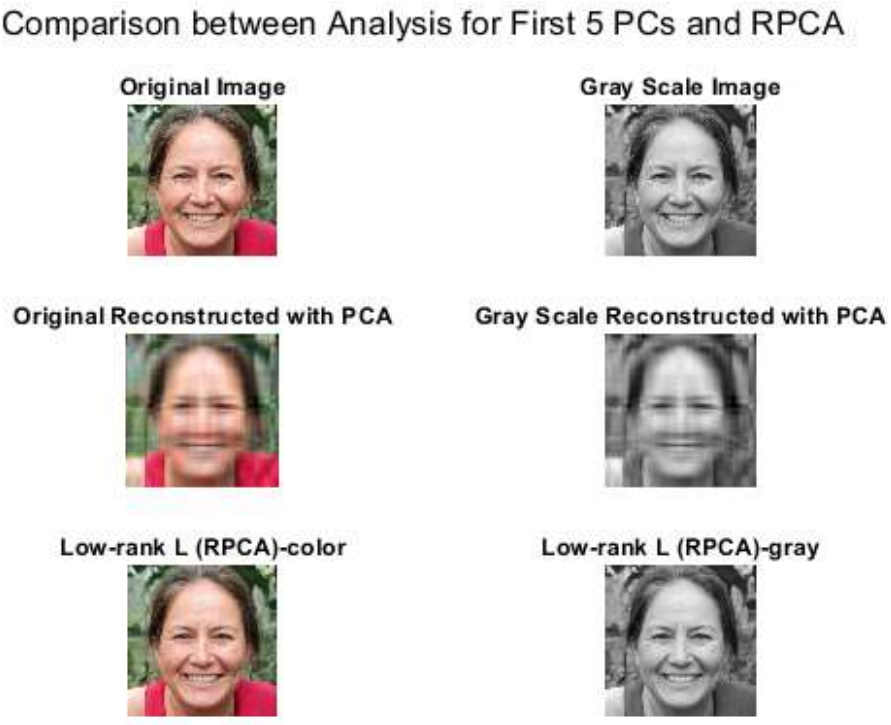

Figure 3 Comparison between RPCA and PCA with 5 principal components

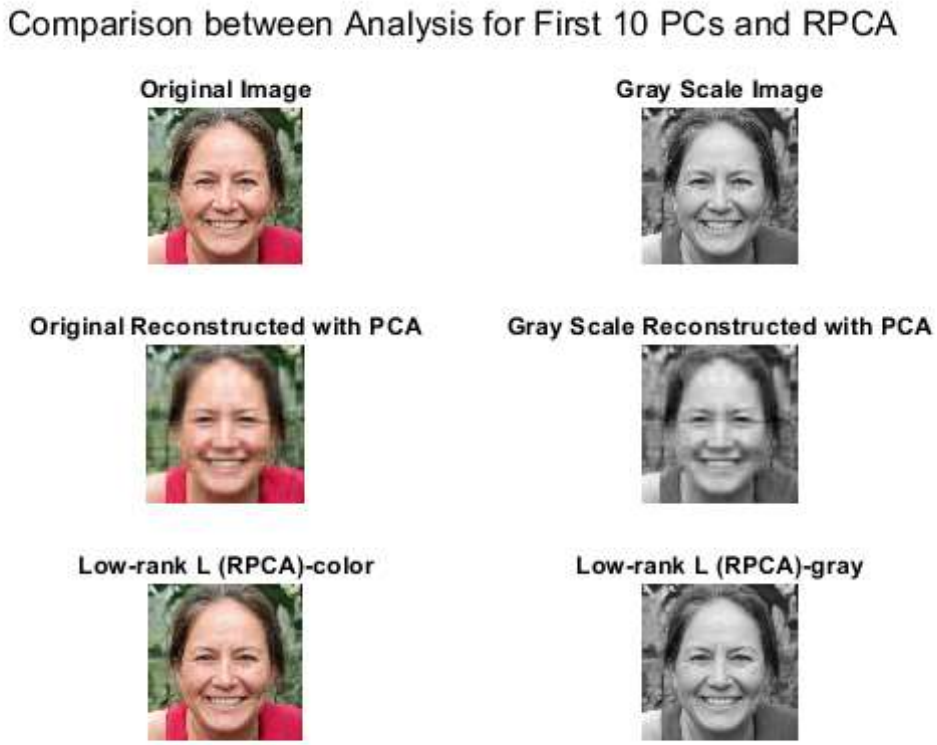

Figure 5 Comparison between RPCA and PCA with 10 principal components

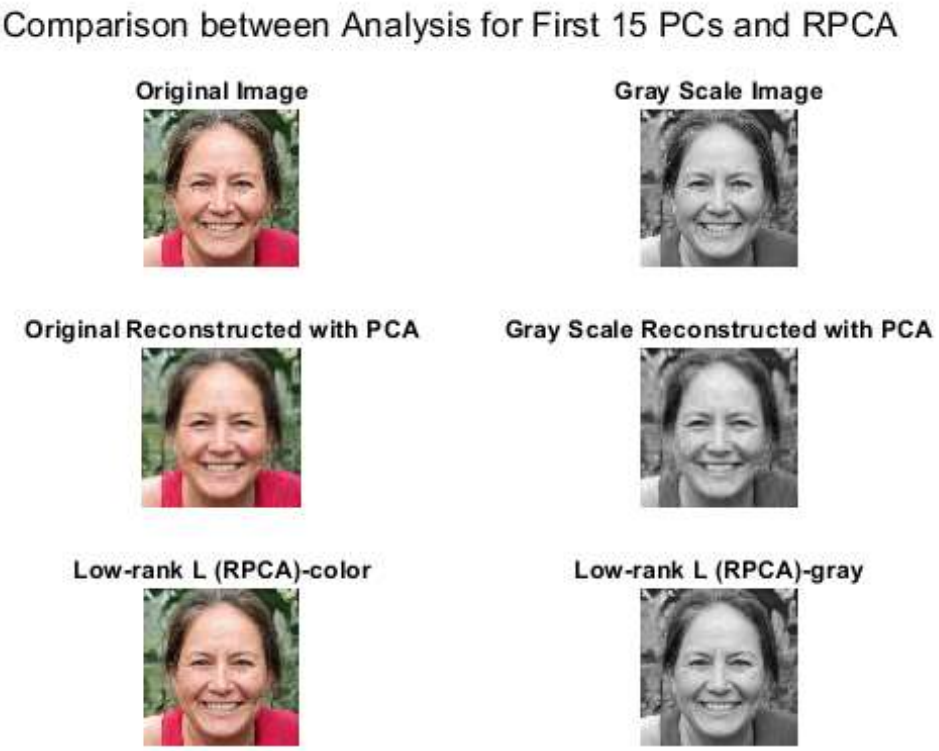

Figure 4 Comparison between RPCA and PCA with 15 principal components

As you can see, the low-rank L matrix presents a clearer representation of the original image (color and gray scale) than even 15 principal components from regular PCA.

Conclusion

As we have shown, RPCA is a powerful algorithm for removing corruptions and outliers from data. The mathematical derivation of its formulation makes it a convenient option for image reconstruction as well. However, RPCA does have some limitations which can result in it being less effective. For example, depending on the context, there can sometimes be more efficient methods for dealing with outliers such as fissurization. Additionally, the repeated calculation of the SVD for very large matrices can be computationally intense and therefore results in the algorithm taking some time to complete. Despite these drawbacks, RPCA's wide variety of applications in modern data science are a testament to its power, but it is important to consider its limitations and choose the appropriate method for specific datasets or research goals.

References

[1] Candès, E. J., Li, X., Ma, Y., & Wright, J. (2011). *Robust principal component analysis?*. Journal of the ACM (JACM), 58(3), 1-37.

[2] Guyon, C., Bouwmans, T., & Zahzah, E. H. (2012*). Foreground detection via robust low rank matrix decomposition including spatio-temporal constraint*. In Asian conference on computer vision (pp. 315-320). Springer, Berlin, Heidelberg.

[3] Brunton, S.L. and Kutz, J.N. (2022) "Sparsity and Compressed Sensing," in *Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control*. Cambridge: Cambridge University Press, pp. 85–114.

Appendix

In this part, we explained the Matlab code for every parts of the project:

Code A.1: Function RPCA

```
function [L, S] = RPCA(X)
   % - X is a data matrix (of the size N x M) to be decomposed
   % - X can also contain NaN's for unobserved values
   % - lambda - regularization parameter, default = 1/sqrt(max(N,M))
   % - mu - the augmented lagrangian parameter, default = 10*lambda
   % - tol - reconstruction error tolerance, default = 1e-6
   % - max_iter - maximum number of iterations, default = 1000
   [M, N] = size(X);
   unobserved = isnan(X);
```

```
    X(unobserved) = 0;

    lambda = 1 / sqrt(max(M,N));    %regularization parameter
    mu=N*M/(4*sum(abs(X(:))));      %lagrangian parameter
    tol = 1e-7;                     %error tolerance
    max_iter = 1000;                %maximum number of iterations

    % initial solution
    L = zeros(M, N);
    S = zeros(M, N);
    Y = zeros(M, N);

    for iter = (1:max_iter)
        % ADMM step: update L and S for every iteration to decrease the error
        L = SVT(X - S + (1/mu)*Y,1/mu);
        S = shrink(X - L + (1/mu)*Y,lambda/mu);
        % and augmented lagrangian multiplier
        Z = X - L - S;
        Z(unobserved) = 0; % skip missing NaN values
        Y = Y + mu*Z;
        %error that mentioned in the paper
        normX = norm(X, 'fro');
        err = norm(Z, 'fro') / normX;
        if (err < tol) break; end
    end
end
function r = shrink(X,tau)
    % shrinkage operator
    r = sign(X) .* max(abs(X) - tau, 0);
end
function r = SVT(X,tau)
    % shrinkage operator for singular values
    [U, S, V] = svd(X, 'econ');
    r = U*shrink(S,tau)*V';
end
```

Code A.2: Plot an image and apply different noises and compute RPCA and plot output of RPCA for every images

```
clc; clear all; close all;
addpath('E:\VE\Malih\PHD\math\final\example');
image=imread('Malih256.jpg');
subplot(4,4,1)
% Convert 3D Array of Image to 2D Grayscale Image
img2d = rgb2gray(image);
imshow(img2d)
[t,s] = title('Original Image');
t.FontSize = 8;
img = double(img2d);
```

```
img = double(img2d)/256;
[L1,S1]=RPCA(img);
subplot(4,4,5)
imshow(L1)
ylabel('Low-rank L')
subplot(4,4,9)
imshow(S1)
ylabel('Sparse S')
subplot(4,4,13)
imshow(L1+S1)
ylabel('L + S')

%% add salt and pepper noise
X = imnoise(img, 'salt & pepper', 0.1);
[L,S]=RPCA(X);

subplot(4,4,2)
imshow(X)
[t,s] =title('Salt and Pepper Noise');
t.FontSize = 8;
subplot(4,4,6)
imshow(L)
subplot(4,4,10)
imshow(S)
subplot(4,4,14)
imshow(L+S)
%% add gaussian noise
X2 = imnoise(img, 'gaussian');
[L2,S2]=RPCA(X2);
subplot(4,4,3)
imshow(X2)
[t,s] =title('Gaussian Noise');
t.FontSize = 8;
subplot(4,4,7)
imshow(L2)
subplot(4,4,11)
imshow(S2)
subplot(4,4,15)
imshow(L2+S2)
%% add speckle noise
X3 = imnoise(img, 'speckle');
[L3,S3]=RPCA(X3);
subplot(4,4,4)
imshow(X3)
[t,s] =title('Speckle Noise');
t.FontSize = 8;
subplot(4,4,8)
imshow(L3)
subplot(4,4,12)
```

```
imshow(S3)
subplot(4,4,16)
imshow(L3+S3)
```

Code A.3: Plot an image and apply text.png as mask and compute RPCA and plot recovered image

```
clc; close all; clear all;
addpath('E:\VE\Malih\PHD\math\final\example');
image=imread('Malih256.jpg');
% Convert 3D Array of Image to 2D Grayscale Image
img2d = rgb2gray(image);
img = double(img2d);
% read image and add the mask
img = double(img2d)/256;
msk = zeros(size(img));
msk(65:192,65:192) = imresize(imread('text.png'), 0.5);
img_corrupted = img;
img_corrupted(msk > 0) = nan;
% create a matrix X from overlapping patches
ws = 16; % window size
no_patches = size(img, 1) / ws;
X = zeros(no_patches^2, ws^2);
k = 1;
for i = (1:no_patches*2-1)
   for j = (1:no_patches*2-1)
       r1 = 1+(i-1)*ws/2:(i+1)*ws/2;
       r2 = 1+(j-1)*ws/2:(j+1)*ws/2;
       patch = img_corrupted(r1, r2);
       X(k,:) = patch(:);
       k = k + 1;
   end
end
% apply Robust PCA
tic
[L, S] = RobustPCA(X);
toc
% reconstruct the image from the overlapping patches in matrix L
img_reconstructed = zeros(size(img));
img_noise = zeros(size(img));
k = 1;
for i = (1:no_patches*2-1)
   for j = (1:no_patches*2-1)
       % average patches to get the image back from L and S
       % todo: in the borders less than 4 patches are averaged
       patch = reshape(L(k,:), ws, ws);
       r1 = 1+(i-1)*ws/2:(i+1)*ws/2;
       r2 = 1+(j-1)*ws/2:(j+1)*ws/2;
       img_reconstructed(r1, r2) = img_reconstructed(r1, r2) + 0.25*patch;
       patch = reshape(S(k,:), ws, ws);
```

```
        img_noise(r1, r2) = img_noise(r1, r2) + 0.25*patch;
        k = k + 1;
    end
end
img_final = img_reconstructed;
img_final(~isnan(img_corrupted)) = img_corrupted(~isnan(img_corrupted));
% In this line we copy the NaN value of original image and paste to final
image to preventing the non physical values that affect computation results

%show the results
figure;
subplot(2,2,1), imshow(img_corrupted), title('Corrupted image')
subplot(2,2,2), imshow(img_final), title('Recovered image')
subplot(2,2,3), imshow(img_reconstructed), title('Recovered low-rank L')
subplot(2,2,4), imshow(img_noise), title('Recovered sparse S')
```

Code A.4: Plot a video from Internet/Youtube (mute video is better) and compute RPCA for every frame and make a new video (it submitted separately)

```
clear all; close all; clc;
addpath('E:\VE\Malih\PHD\math\final\example');
% ! the movie will be downloaded from the internet !
movieFile='college-students-walking-on-campus.mp4';
% open the movie
n_frames = 180;  % you can change the number of frame to have longer video
movie = VideoReader(movieFile);
frate = movie.FrameRate;
height = movie.Height;
width = movie.Width;
% vectorize every frame to form matrix X
X = zeros(n_frames, height*width);
for i = (1:n_frames)
    frame = read(movie, i);
    frame = rgb2gray(frame);
    X(i,:) = reshape(frame,[],1);
end
% apply Robust PCA for every frame
lambda = 1/sqrt(max(size(X)));
tic
[L,S] = RobustPCA(X, lambda/3, 10*lambda/3, 1e-5);
toc
% prepare the new movie file
vidObj = VideoWriter('college-students-walking-on-campus_output.avi');
vidObj.FrameRate = frate;
open(vidObj);
range = 255;
map = repmat((0:range)'./range, 1, 3);
S = medfilt2(S, [5,1]); % median filter in time
for i = (1:size(X, 1))
    frame1 = reshape(X(i,:),height,[]);
```

```
    frame2 = reshape(L(i,:),height,[]);
    frame3 = reshape(abs(S(i,:)),height,[]);
    % median filter in space; threshold
    frame3 = (medfilt2(abs(frame3), [5,5]) > 5).*frame1;
    % stack X, L and S together
    frame = mat2gray([frame1, frame2, frame3]);
    frame = gray2ind(frame,range);
    frame = im2frame(frame,map);
    writeVideo(vidObj,frame);
end
close(vidObj);
```

Code A.5: For comparison of PCA and RPCA, we can open every image and compute both PCA and RPCA. The number of components for PCA (r) is an input integer.

```
clear all; close all; clc;
[filename,filepath]=uigetfile({'*.png;*.jpg;*.JPG; *.jpeg; *.JPEG; *.img; *.IMG;
*.tif; *.TIF; *.tiff, *.TIFF'},'Select and image');
% axes(handles.axes2);
subplot(3,2,1)
imshow((filename));
title('Original Image')
imread(filename);
whos
imwrite(ans,'myGray.png')
c_img=imread('myGray.png');
g_image=rgb2gray(c_img);
subplot(3,2,2)
imshow(g_image)
title('Gray Scale Image')
prompt = "The number of PC's you want to find?";
nComp = input(prompt);
%
sizePC=size(c_img);
MaxPC= sizePC(1);
if nComp>MaxPC
%
  errordlg('Entered no. is greater than the number of PCs','Error');
   return
end

g_image_d=double(g_image);
g_image_d_m=mean(g_image_d);
g_image_d_m_adjusted=(g_image_d-g_image_d_m);
[coeff,score,latent,~,explained] = pca(g_image_d_m_adjusted);
reconstruction = score(:,1:nComp)*coeff(:,1:nComp)';
final_reconstruction=(reconstruction+g_image_d_m);
final_reconstruction=uint8(final_reconstruction);
imwrite(final_reconstruction,'gray_reconstructed.png')

subplot(3,2,4)
imshow(final_reconstruction)
title('Gray Scale Reconstructed with PCA')
```

```
d_c_imgr=double(c_img(:,:,1));
d_c_img_mr=mean(d_c_imgr);
d_c_img_m_adjustedr=(d_c_imgr-d_c_img_mr);
[coeffr,scorer,latentr,~,explainedr] = pca(d_c_img_m_adjustedr);
r_reconstruction = scorer(:,1:nComp)*coeffr(:,1:nComp)';
r_final_reconstruction=(r_reconstruction+d_c_img_mr);
r_final_reconstruction=uint8(r_final_reconstruction);

d_c_imgg=double(c_img(:,:,2));
d_c_img_mg=mean(d_c_imgg);
d_c_img_m_adjustedg=(d_c_imgg-d_c_img_mg);
[coeffg,scoreg,latentg,~,explainedg] = pca(d_c_img_m_adjustedg);
g_reconstruction = scoreg(:,1:nComp)*coeffg(:,1:nComp)';
g_final_reconstruction=(g_reconstruction+d_c_img_mg);
g_final_reconstruction=uint8(g_final_reconstruction);

d_c_imgb=double(c_img(:,:,3));
d_c_img_mb=mean(d_c_imgb);
d_c_img_m_adjustedb=(d_c_imgb-d_c_img_mb);
[coeffb,scoreb,latentb,~,explainedb] = pca(d_c_img_m_adjustedb);
b_reconstruction = scoreb(:,1:nComp)*coeffb(:,1:nComp)';
b_final_reconstruction=(b_reconstruction+d_c_img_mb);
b_final_reconstruction=uint8(b_final_reconstruction);
im=cat(3,r_final_reconstruction,g_final_reconstruction,b_final_reconstruction);

imwrite(im,'c_reconstructed.png')
subplot(3,2,3)

imshow(im)
title('Original Reconstructed with PCA')
txt =['Comparison between Analysis for First ',num2str(nComp), ' PCs and RPCA'];
sgtitle(txt)

%% RPCA

imgg=double(g_image)/256;
[L1,S1]=RPCA(imgg);
subplot(3,2,6)
imshow(L1)
title('Low-rank L (RPCA)-gray')
d_c_imgr=double(c_img(:,:,1))/256;
d_c_imgg=double(c_img(:,:,2))/256;
d_c_imgb=double(c_img(:,:,3))/256;
[L2,S2]=RPCA(d_c_imgr);
[L3,S3]=RPCA(d_c_imgg);
[L4,S4]=RPCA(d_c_imgb);
L = zeros([size(L2) 3]);
L(:,:,1) = L2;
L(:,:,2) = L3;
L(:,:,3) = L4;
subplot(3,2,5)
imshow(L)
title('Low-rank L (RPCA)-color')
```