Truncated SVD for Image Compression

Mehmet Ozgur Turkoglu
Student ID: s1814389 , E-mail: moturkoglu@gmail.com
Educational Program: MSc. in Electrical Engineering
Sunday 4th June, 2017

I. Introduction

Image compression is very important for digital environment, because images contain vast amount of information and without any compression, required memory for image is huge. Before any compression algorithm is applied, the required memory for 2 hours movie with a resolution of 2048x1080, with a color depth of 3 bytes at 25 frame/s is around 1200 Gb. Therefore, images have to be compressed to make them feasible to store and transfer.

Nowadays there are many successful image and video compression methods such as JPG and MPG. These methods already compress the original images or videos with very high compression rate and without loss. However, especially in computer vision and robotics application, these compressed images should be compressed even more. For some applications, many images has to be stored (e.g. face recognition applications) and these images should be compressed without least information loss, in this case truncated SVD image compression method can be very useful. In this work, truncated SVD method for image compression is studied.

II. Method

Any matrix $A \in \mathbb{R}^{mxn},\ m \geq n$ can be represented as following.

$$A = U\Sigma V^T, \tag{1}$$

$$U \in \mathbb{R}^{mxn},\ \Sigma = diag(\sigma_1, \sigma_2, ..., \sigma_n),\ V \in \mathbb{R}^{nxn}$$

where the matrices U and V have orthonormal columns and $\sigma_1 \geq \sigma_2 \geq \ ... \geq \sigma_n \geq 0$, this representation is called thin SVD of the matrix A. This equation can be also written in the following form.

$$A = [\sigma_1 U_1, \sigma_2 U_2, ..., \sigma_n U_n] \begin{bmatrix} V_1^T \\ V_2^T \\ ... \\ V_n^T \end{bmatrix} \tag{2}$$

$$A = \sigma_1 U_1 V_1^T + \sigma_2 U_2 V_2^T + ... + \sigma_n U_n V_n^T \tag{3}$$

If the sequence σ_1, σ_2, . . . , σ_n is fast decaying, we may approximate the matrix A by using only first r dominant terms.

$$A \approx \sigma_1 U_1 V_1^T + \sigma_2 U_2 V_2^T + ... + \sigma_r U_r V_r^T\ , r < n \tag{4}$$

This is called the truncated SVD approximation of matrix A. If the matrix A represents gray-scale image (An image is assumed to be gray-scale just for simplicity, it can be easily extended for RGB image.), we can use this property for image compression by storing u_1,...,u_r, v_1,...,v_n and the scalars σ_1,...,σ_n instead of storing image (matrix A of size m by n) itself. So in this way, we need to store $(m + n + 1)r$ pixels instead of mn pixels.

The relation (3) is always true when $m \geq n$, so we should generalize the relation for all the cases ($m \geq n$ and $m < n$) because image size could be anything. Let define a new variable $p = min(m, n)$, the generalized version of relation (3) is following.

$$A = \sigma_1 U_1 V_1^T + \sigma_2 U_2 V_2^T + ... + \sigma_p U_p V_p^T \tag{5}$$

Then, relation (4) turns into a following form.

$$A \approx \sigma_1 U_1 V_1^T + \sigma_2 U_2 V_2^T + ... + \sigma_r U_r V_r^T\ , r < p \tag{6}$$

1) Implementation 1-a: When we implement this method, we should consider two situation. First situation is that image size (m and n) is not too big and we can compute the thin SVD directly by using MATLAB built-in function "svd". In order to reduce the computation cost, we should use function

"svd" with the option "econ". In that way, if $m \geq n$, MATLAB computes n by n matrix V and n by n diagonal matrix Σ; then, it computes matrix U by using relation (8). If $m < n$, MATLAB computes m by m matrix U and m by m diagonal matrix Σ; then, it computes matrix V by using relation (11). Therefore, MATLAB avoids computations with a unitary matrix of size $max\{m,n\}$ by $max\{m,n\}$.

If $m \geq n$, V is orthonormal square matrix, so $V^TV = I_{n,n}$, then we can find matrix U by matrix multiplication.

$$AV = U\Sigma \tag{7}$$

$$U_i = \frac{AV_i}{\sigma_i},\ i = 1, ..., n \tag{8}$$

If $m < n$, U is orthonormal square matrix, so $U^TU = I_{m,m}$, then we can find matrix V as following.

$$A^T = V\Sigma U^T \tag{9}$$

$$A^TU = V\Sigma \tag{10}$$

$$V_i = \frac{A^TU_i}{\sigma_i},\ i = 1, ..., m \tag{11}$$

A. Implementation 1-b

We can also avoid computation with matrix of size $max\{m,n\}$ by $max\{m,n\}$ when $m \neq n$ by first computing thin QR factorization of matrix A or A^T. Let assume $m > n$, A can be written in following form.

$$A = QR, \tag{12}$$

$$Q \in \mathbb{R}^{mxn}, R \in \mathbb{R}^{nxn}$$

Q is matrix with orthonormal columns ($Q^TQ = I_{n,n}$) and R is upper triangle matrix. Now, we can compute the SVD of R which is n by n matrix.

$$R = U\Sigma V^T \tag{13}$$

If we rewrite the relation (12), it is going to be as following.

$$A = QU\Sigma V^T \tag{14}$$

$$A = U'\Sigma V^T \tag{15}$$

where $U' = QU$. U' can be easily proved to be matrix with orthonormal columns.

$$U' = [QU_1, QU_2, ..., QU_n] \tag{16}$$

$$U'^TU' = \begin{bmatrix} U_1^TQ^T \\ U_2^TQ^T \\ ... \\ U_n^TQ^T \end{bmatrix} [QU_1, QU_2, ..., QU_n] \tag{17}$$

Because $Q^TQ = I$, Q's can be eliminated.

$$U'^TU' = \begin{bmatrix} U_1^T \\ U_2^T \\ ... \\ U_n^T \end{bmatrix} [U_1, U_2, ..., U_n] = I \tag{18}$$

As a result, we compute SVD of matrix A by computing SVD of matrix R whose size is less than of A. In the other case, $m < n$, we apply same procedure to A^T instead of A.

$$A^T = QR, \tag{19}$$

$$Q \in \mathbb{R}^{nxm}, R \in \mathbb{R}^{mxm}$$

$$A^T = QU\Sigma V^T \tag{20}$$

$$A = V\Sigma U^TQ^T \tag{21}$$

$$A = U'\Sigma V'^T \tag{22}$$

where $U' = V$ and $V' = QU$.

B. Implementation 2

Regarding to implementation, the second situation is that image size (m and n) is large and it is too expensive to compute the thin SVD or thin QR of matrix A. In this case, we approach the problem differently; we solve eigenproblem for AA^T (or A^TA) using iterative eigensolvers and then, find truncated SVD of matrix A as following relations.

Let assume $m \geq n$ (for simplicity) and calculate A^TA by using relation (3).

$$A^T = \sigma_1V_1U_1^T + \sigma_2V_2U_2^T + ... + \sigma_nV_nU_n^T \tag{23}$$

$$A^TA = \sigma_1^2V_1V_1^T + \sigma_2^2V_2V_2^T + ... + \sigma_n^2V_nV_n^T \tag{24}$$

If we multiply matrix A^TA with any of vectors, V_1,... ,V_n, we obtain the following equation.

$$A^TAV_i = \sigma_i^2V_i\ ,\ i = 1, 2, ..., n \tag{25}$$

So (V_i, σ_i^2)'s are the eigenpairs of matrix A^TA. I previously showed that it is possible to find U_i when we know V_i and σ_i (see Relation (8)). Thus, we can compute SVD of A by finding eigenpairs of A^TA.

It is important to select appropriate one among A^TA and AA^T. In order to reduce computational cost, we should select the one with smaller size. Therefore, if $m > n$ we should select A^TA whose size is n by n; whereas if $m < n$ we should select AA^T whose size is m by m. If A is square matrix, we can use either of them. If we use AA^T, eigenpair of this matrix is (U_i, σ_i^2), so after finding eigenpairs, we can find V_i's by using relation (11).

We do not need to compute all the eigenpairs of A^TA (or AA^T) since we only need most dominant r eigenvalues and the associated eigenvectors. In this project, we use Jacobi-Davidson eigensolver (MATLAB implementation is available).

In order to make Jacobi-Davidson method more efficient, we use preconditioner. Because the matrix in question (A^TA or AA^T) is not sparse so we should use other preconditioner instead of ILU or SSOR preconditioners. The reasonable preconditioner is Cholesky factorization of the matrix $A^TA+\alpha I$ where I is the identity matrix and α is small positive scale parameter. We add αI term in order to make the matrix non-singular. Because when A is an image, A^TA (or AA^T) is quite prone to be singular because in general image pixels are not random and images are smooth (adjacent column and row vectors are almost the same) unless there are an edge or corner. Therefore, matrix A is close to be non-singular itself for instance the condition number of image in Figure 2 is $1.0594x10^4$. A^TA or AA^T are more non-singular than A because the size of matrix is large and values of matrix element are limited by certain numbers (assume from 0 to 1 or 0 to 255) and inner product of column vectors produce similar values as seen in Figure 1. For instance, the condition number of A^TA is $2.7704x10^{22}$ I take α value as a small fraction of norm-2 of A. Because matrix second norm is a spectral norm which equals to square root of maximum eigenvalue of A^TA in that way we increase the eigenvalues with some fraction of maximum eigenvalue and matrix becomes non-singular. I empirically determine the value in a way that minimum value which makes the matrix non-singular and $\alpha = ||A||_2/10$.

In order to increase computational efficiency of Jacobi-Davidson method, we do not calculate A^TA (or AA^T) explicitly, instead we only do matrix-vector multiplication because the matrix in question is not sparse, matrix-matrix multiplication is expensive (n times more expensive than matrix-vector multiplication). It is important to note that MATLAB does the calculation from left to right, if we do not use parenthesis in the function we pass to the Jacobi-Davidson function (jdqr), it first multiply A^T and A so we should use parenthesis like $A^T * (A * x)$ instead of $A^T * A * x$. Also in order to avoid matrix-matrix multiplication for precondioner to be M_1M_2 , $M_2 = M_1^T$, with M_1 being the diagonal and lower triangular parts of A^T with skipped zero columns plus a αI.

Fig. 1: Matrix A^TA, condition number= $2.7704x10^{22}$ (A is given in Figure 2.)

Fig. 2: Matrix A, condition number = $1.0594x10^4$

III. Experiment & Results

All the experiments are conducted with the real images instead of randomly generated matrices. It is important to note that images are gray-scale (MATLAB 'rgb2gray' function converts RGB image into gray-scale image.) but all the implementation in this work can be easily applied to RGB images. Images are converted to double precision by using MATLAB function 'im2double' before processing.

Method	1a	1b
Cat	0.0106	0.0119
Amsterdam	0.0112	0.0117
Wave	0.1470	0.1510

TABLE I: CPU time for Method 1a and 1b (See Figure 3, 4, and 5).

In this work memory requirement (MR) for compressed images are given in the ratio of the memory needed for a compressed image to the memory needed for an original image.

$$MR = \frac{(m+n+1)r}{mn}$$

There are several metrics to evaluate the quality of image compression algorithm. In this work, Mean Square Error (MSE) which is one of the most popular performance metric is used. The equation for calculating MSE, denoting the mean square error between the two images (original and reconstructed (or compressed) images) is given as

$$MSE = \frac{1}{mn}\sum_{i=1}^{m}\sum_{j=1}^{n}(r_{i,j} - o_{i,j})^2$$

where $o_{i,j}$ and $r_{i,j}$ are a pixel of original and reconstructed images respectively.

A. Is the implementation 1a or 1b more efficient?

The implementation 1a and 1b are for small-sized images. These two implementation are tested on several small-sized images (see Figure 3,4, 5). Both method were run 100 times for 'Cat', 'Amsterdam', and 'Wave' images which are small (medium)-sized images. The average CPU times are given in Table I. According to results, method 1a is slightly more efficient, it is probably because MATLAB 'svd' function is pre-compiled or more optimized somehow. Some example of reconstructed images (RI) are given in Figure 3, 4, and 5.

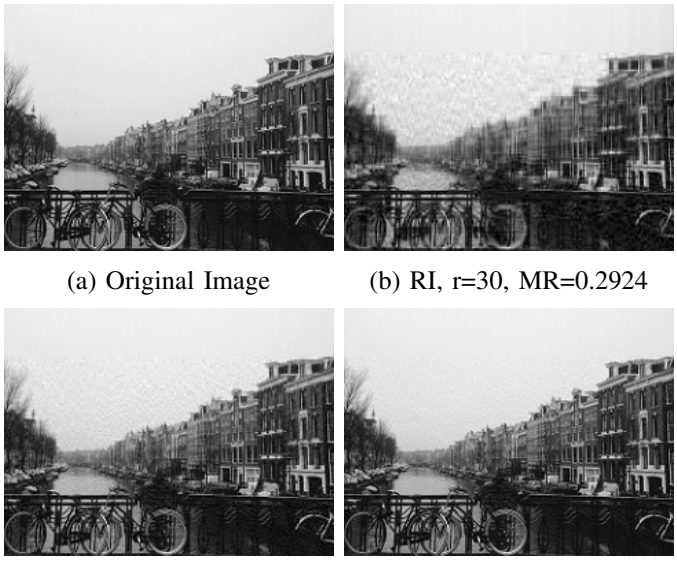
B. Is the code more efficient if A^TA or AA^T is not calculated?

In order to see whether not computing A^TA (or AA^T) explicitly increases the performance, the implementation-2 is tested on 'Madrid' ($m < n$) and 'Chair' ($m > n$) images which are large-sized images. The code is run 5 times and average CPU times are listed in Table II. According to the results, the implementation is more efficient if A^TA (or AA^T) is not computed explicitly.

(a) Original Image (b) RI, r=30, MR=0.2924

(c) RI, r=60, MR=0.5847 (d) RI, r=90, MR=0.8771

Fig. 3: 'Cat' image. Original image size: 240x180.



(a) Original Image (b) RI, r=30, MR=0.2924

(c) RI, r=60, MR=0.5847 (d) RI, r=90, MR=0.8771

Fig. 4: 'Amsterdam' image. Original image size: 180x240.

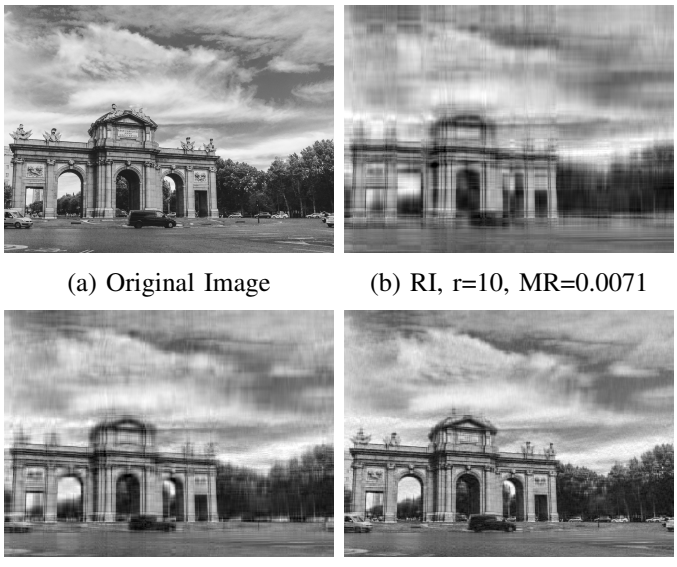
(a) Original Image (b) RI, r=30, MR=0.1001

(c) RI, r=60, MR=0.2002 (d) RI, r=90, MR=0.3003

Fig. 5: 'Wave' image. Original image size: 525x700.

r	10	20	50	100
Madrid-1	15.1007	15.5689	17.2867	18.9045
Madrid-2	17.5421	17.8494	18.4628	19.2378
Chair-1	11.1754	11.8750	11.9998	16.4475
Chair-2	12.5845	12.8448	13.2292	17.3668

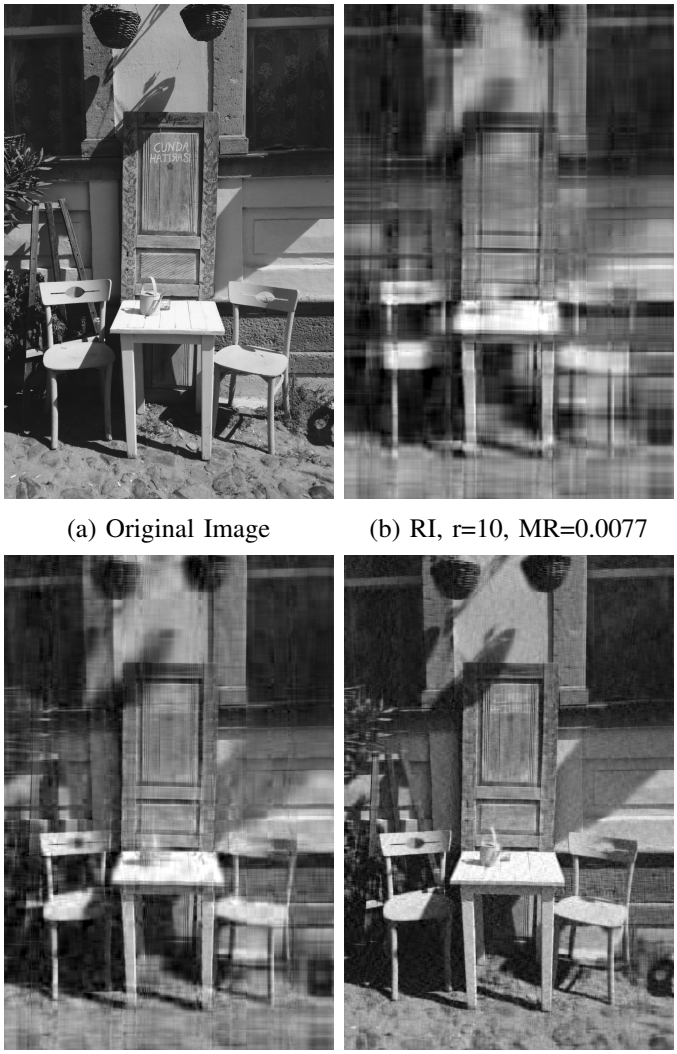
TABLE II: CPU time for 'Madrid' and 'Chair' images (See Figure 6, 7). First and third columns show CPU time when A^TA or AA^T is not computed explicitly; second and fourth columns show CPU time when A^TA or AA^T is computed.



(a) Original Image (b) RI, r=10, MR=0.0071

(c) RI, r=20, MR=0.0143 (d) RI, r=50, MR=0.0358

Fig. 6: 'Madrid' image. Original image size: 2448x3264.



(a) Original Image (b) RI, r=10, MR=0.0077

(c) RI, r=20, MR= 0.0153 (d) RI, r=50, MR=0.0383

Fig. 7: 'Chair' image. Original image size: 3264x2176.

C. Which r should be chosen?

For all the images previously used, MSE curves are given in Figure 8, 9, 10, 11, and 12. According to these curves, how much information is lost after compression can be inferred. So one can choose suitable r value for specific application by examining these MSE curves. For instance, let examine MSE curve for 'Wave' image (see Figure 10), if the compression has to be almost lossless then suitable r should be larger than 50; whereas, if the application concerns only the main structure in the image then r can be chosen as around 20.

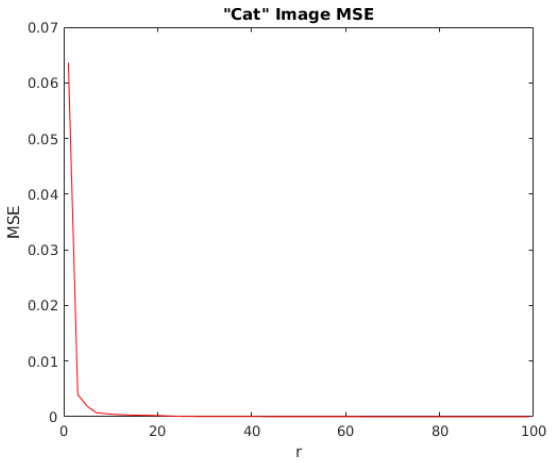


Fig. 8: MSE Curve of 'Cat' image.

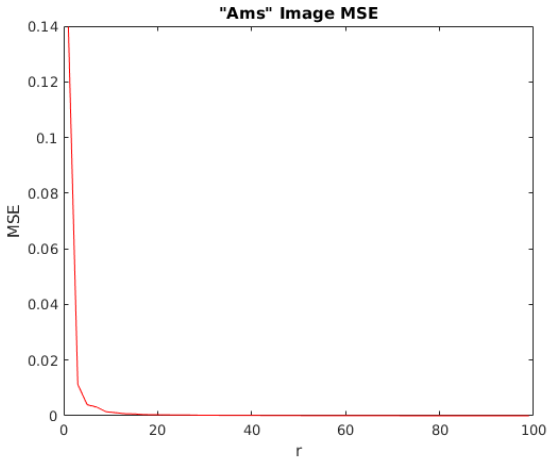


Fig. 9: MSE Curve of 'Amsterdam' image.

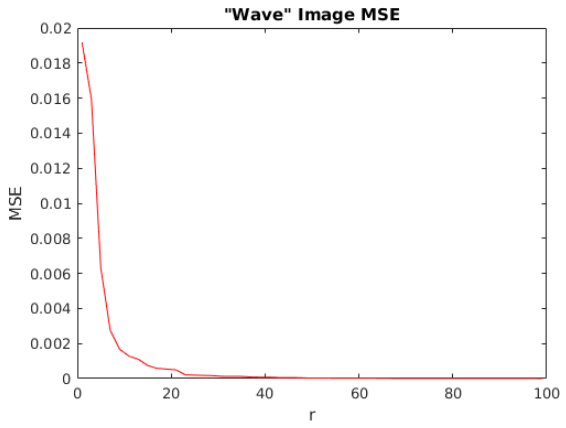


Fig. 10: MSE Curve of 'Wave' image.

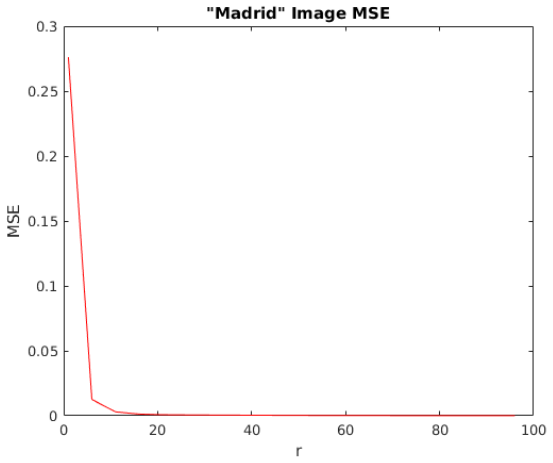


Fig. 11: MSE Curve of 'Madrid' image.

D. Can we increase computational efficiency by reshaping image?

Computational cost of this compression algorithm depends on the size of the matrix in question so we may reduce the computational work and CPU time by reshaping the images before the compression algorithm is applied. For instance, if our image is of size 1080 by 1920 and r is chosen as 100 then we can reshape the image in a way that image size is 108 by 19200. In this way the algorithm deals with 108 by 108 matrix instead of 1080 by 1080 matrix. Notice that after resizing image r has to be lower than minimum of new m and n.

This phenomena is tested on 'XX' and 'Oasis' images (see Figure 13 and 14) whose size are 1080 by 1920 and 1500 by 1500 respectively. New variable s as a scaling factor is defined so the size of reshaped image is m/s by sn. For different s values, CPU times are listed in Table III and reconstructed images are given in Figure 13 and 14. ($r = 50$ for both images for all cases.)

It is important to note that when the image is reshaped before compression algorithm is applied, memory requirement changes for same r value. Memory requirement (MR) for reshaped image is following.

$$MR = \frac{(m/s + sn + 1)r}{mn}$$

According to results obtained, this method (reshaping before compression) definitely increases the computational efficiency. Normally compression with higher MR needs more computation because more eigenpairs have to be computed; but by reshaping before compression algorithm applied, less CPU time might be needed for same r value and also MR can increase (According to selection of s value, this situation might be reverse). There is an

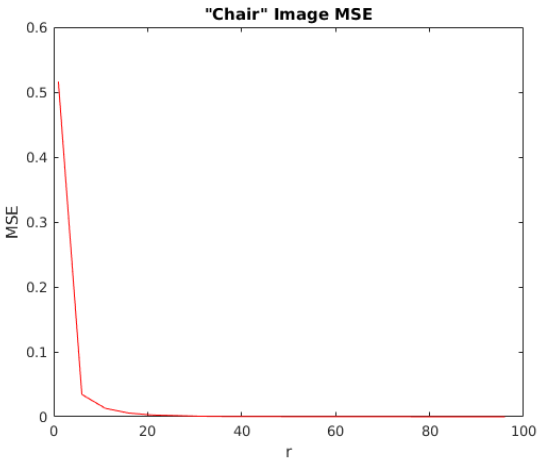


Fig. 12: MSE Curve of 'Chair' image.

s	1	5	10	20
XX	1.6484	0.9908	0.6778	0.4638
Oasis	4.5302	1.8764	1.5198	12.4086

TABLE III

exception when $s = 20$ for 'Oasis' image, CPU time increases because the matrix in question becomes nearly non-singular and Jacobi-Davidson method gets slower.

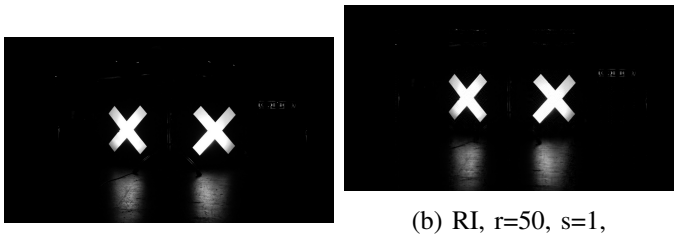
APPENDIX

MATLAB CODE

```
function [U,Sigma,V] = trunc_svd1a(A,r)
%Method 1a, thin SVD
%This function computes the truncated SVD of matrix A
%r is desired number of the most dominant eigenvalues
% r has to be less than or equal to min(m,n)
% U is m by r with orthonormal columns
%V is n by r with orthonormal columns
%Sigma is r by r diagonal matrix with dominant eigenvalues

[m,n] = size(A);

%Check if r is valid
if r>=min(m,n)
```



(a) Original Image

(b) RI, r=50, s=1, MR=0.0724, MSE=$2.28x10^{-4}$

(c) RI, r=50, s=10, MR=0.4656, MSE=$2.02x10^{-11}$

(d) RI, r=50, s=20, MR=0.5232, MSE=$1.38x10^{-14}$

Fig. 13: 'XX' image. Original image size: 1080x1920.

(a) Original Image

(b) RI, r=50, s=1, MR=0.0667, MSE=$1.67x10^{-5}$

(c) RI, r=50, s=10, MR=0.3367, MSE=$4.38x10^{-9}$

(d) RI, r=50, s=20, MR=0.6684, MSE=$1.54x10^{-10}$

Fig. 14: 'Oasis' image. Original image size: 1500x1500.

```
    display('r has to be less than or eqaul to min(m,n), r is set to min(m,n)');
    r = min(m,n);
end

[U,Sigma,V] = svd(A,'econ');
U = U(:,1:r);
V = V(:,1:r);
Sigma = Sigma(1:r,1:r);

end

%-------------------------------------
function [U,Sigma,V] = trunc_svd1b(A,r)
%Method 1b, thin QR, then SVD

[m,n] = size(A);

%Check if r is valid
if r>=min(m,n)
    display('r has to be less than or eqaul to min(m,n), r is set to min(m,n)');
    r = min(m,n);
end

if m>=n
    %QR factorization of input matrix A
    %Option '0' enables thin QR factorization
    [Q,R] = qr(A,0);

    %SVD of square matrix R
    [U,Sigma,V] = svd(R);

    U = Q*U;


else % in the case of m<n, then we should work with A^T
    %QR factorization of transpose of input matrix A
    [Q,R] = qr(A',0);

    %SVD of square matrix R
    [U1,Sigma,V1] = svd(R);

    U = V1;
    V = Q*U1;

end

U = U(:,1:r);
V = V(:,1:r);
Sigma = Sigma(1:r,1:r);

end

%-------------------------------------

function [U,Sigma,V] = trunc_svd2(A,r)
%Method 2, large-scale SVD

global N_dim;
[m,n] = size(A);

%Check if r is valid
if r>=min(m,n)
    display('r has to be less than or eqaul to min(m,n), r is set to min(m,n)');
    r = min(m,n);
end

%Define alpha
alpha = norm(A)/10;

%if m>=n, use A'A otherwise AA'
if m>=n
    M1 = tril(A(1:n,1:n)')+alpha*eye(n); M2 = M1';
    %Precond matrix
    M = [M1,M2];

    N_dim = n;
    [V,D] = jdqr('ATA','K',r,struct('Precond',M));
    Sigma = sqrt(D);
    temp = diag(sqrt(D));
    Sigma_inv = diag(1./temp);
    U = A*V*Sigma_inv;

else
```

```
    M1 = tril(A(1:m,1:m))+alpha*eye(m); M2 = M1';
    %Precond matrix
    M = [M1,M2];

    N_dim = m;

    [U,D] = jdqr('AAT','K',r,struct('Precond',M));
    Sigma = sqrt(D);

    temp = diag(sqrt(D));
    Sigma_inv = diag(1./temp);
    V = Sigma_inv*(U')*A;
    V = V';

end


end

%-------------------------------

function y = ATA(x,flag)
global A;
global N_dim;

if nargin <2
y = A'*(A*x);

elseif strcmp(flag,'dimension')
    y = N_dim;

else
   y = [];
end

return

%-------------------------------

function y = AAT(x,flag)
global A;
global N_dim;

if nargin <2
y = A*(A'*x);

elseif strcmp(flag,'dimension')
    y = N_dim;

else
   y = [];
end

return

%-------------------------------
clear;close all;
%% Truncated SVD Image Compression
global A;

img = imread('images/1.jpg');
if size(size(img),2) == 3
   %if the image is RGB, convert to the gray-scale
   img = rgb2gray(img);
end


%Convert image into double precision
A = im2double(img);

%Image size
[m,n]=size(A);

%Display  the input image
figure();imshow(A,[]);

%Reshape the matrix (image)
s=1;
A = reshape(A,[m/s,n*s]);

%Define the number of dominant terms
r = 100;

%Truncated SVD for small-sized images
repeat = 100;
t1=zeros(repeat,1);
t2=zeros(repeat,1);
for i=1:repeat
    tic,
    [U,S,V] = trunc_svd1a(A,r);
    t1(i)=toc;
    tic,
```

```
190       [U1,S1,V1] = trunc_svd1b(A,r);
191       t2(i)=toc;
192   end
193
194   %Mean CPU times
195   t1 = mean(t1) %CPU time for
          trunc_svd1a
196   t2 = mean(t2) %CPU time for
          trunc_svd1b
197
198
199   %Truncated SVD for large-sized
          images
200   repeat = 5;
201   t1=zeros(repeat,1);
202   t2=zeros(repeat,1);
203   for i=1:repeat
204       tic,
205       %A^TA (or AA^T) is not used
206       [U2,S2,V2] = trunc_svd2(A,r);
207       t1(i)=toc;
208
209       tic,
210       %A^TA (or AA^T) is used
211       [U3,S3,V3] = trunc_svd2b(A,r);
212       t2(i)=toc;
213   end
214
215   %Mean CPU times
216   t1 = mean(t1)
217   t2 = mean(t2)
218
219   %Reconstructed image
220   A_reconst = U2*S2*V2';
221
222   %Calculate MSE (Mean square error)
223   MSE = sum(sum((A_reconst-A)).^2)/(
          m*n)
224
225   %Reshape reconstructed image
226   A_reconst = reshape(A_reconst,[m,n
          ]);
227
228   %Display reconstructed image
229   figure();imshow(A_reconst,[]);
230
231   %Calculate memory requirement (MR)
232   MR = (m/s+n*s+1)*r/(m*n)
```