

Course Project
Singular Value Decomposition
Group 111

Shi Wei / 120040085

Lu Yi / 121090386

Ma Haoyu / 120020168

Feng xinyang / 120090445

Tan Yifan / 120090416

The Chinese University of Hong Kong, Shenzhen



香港中文大學(深圳)
The Chinese University of Hong Kong, Shenzhen

Part1

Shi Wei

1. Overview

In the Singular Value Decomposition part, we devote to use two phases to find SVD.

In the Phase I, we repeat Householder transformation to get the final bidiagonal form matrix.

In Phase II-A and Phase II-B, we use two different methods to get corresponding singular value decomposition (SVD) value.

2. Main Process and Focus

- In Phase I, we apply distinct orthogonal transformation to bring A into bidiagonal form. By alternating left and right householder reflections, it can transform $m \times n$ matrix A to $B = U_1^T A V_1$. B has been reduced to bidiagonal form, where U_1 ($m \times m$) and V_1 ($n \times n$) are orthogonal matrices.
- In Phase II-A, we use QR iteration with Wilkinson shift to diagonalize the bidiagonal matrix B. From searching the internet, I got some equation for my coding:

Wilkinson shift μ function: $\mu = t_{n,n} + d - \text{sign}(d) \sqrt{d^2 + t_{n,n-1}^2}$, $d = \frac{t_{n-1,n-1} - t_{n,n}}{2}$

(T is tridiagonal matrix). Besides, by the Implicit Q Theorem, it accomplish the same effect as computing the QR factorization $QR = T - \mu I$ and then computing $T = RQ + \mu I$.

Calculating V by diagonalizing $B^T B$. i.e. find orthogonal matrix Q such that

$Q^T B^T B Q = D$, which D is diagonal entries in descending order. So $V = V_1 Q$.

Similarly, calculating U by diagonalizing BB^T . i.e. find orthogonal matrix P such

that $P^T B B^T P = d$, which d is diagonal entries in descending order. So $U = U_1 P$.

Finally, we need to adjust some columns of U so that $U^T A V = \Sigma$ (or directing setting Σ) have non-negative diagonal entries. This step can not be save since matrices P, Q (Q in code) are not completely determined in code.

- In Phase II-B, the algorithm use the Cholesky decomposition to obtain singular values and Σ . Then, we use the same algorithm in Phase II-A with zero shift to calculate the U and V .

3. Main results, Observations and Discussions

In this task, we use two-phase procedure to obtain the Singular Value Decomposition (SVD). The main result shows the success of the method for getting corresponding right SVD.

For testing the result, we can use inbuilt code to check the result.

Testing :

$$\begin{aligned} A &= randn(5,7) \\ [U, Sigma, V] &= mySVD1(A) \\ [u, s, v] &= svd(A) \end{aligned}$$

In this process, we observe that the singular value are unique. Also, for distinct positive singular values, $s_j > 0$, the j th columns of U and V are unique up to a sign change of both columns. It shows that the value of U and V is not unique.

However, the algorithm is not that perfect. In phase II-B, the call of Σ that calculated by Cholesky Decomposition is failed. But it actually have the same value with later Σ calculation.

In conclusion, this task helps us to better understand how to compute singular value decomposition and better to utilize it in more application areas.

ⁱ <https://web.stanford.edu/class/cme335/lecture5>

Part2

Haoyu Ma(120020168)

Yi Lu(121090386)

1. Overview and Division of Work

In this problem, we deal with the deblurring problem. Main method using in this problem are SVD decomposition and truncated reconstruction.

Note: our group upload two codes corresponding to this question. Both of them will return good results. The difference are only in input format, you can choose any of them which you think is more convenient.(report follows the input of part2Ma.m)

Yi Lu(121090386):

- (1). Transform data and plot original, blurry and reconstructed image.(2.2)
- (2). Judge the type of the picture and get the size of it.(2.3)
- (3). Get the blurry image(2.5)
- (4). Construct SVD decomposition and compute pseudoinverse of A_l , A_r and truncated reconstruction. (2.6)
- (5). Packaging all method as a function.(2.9)
- (6). Debug the program.

Haoyu Ma(120020168):

- (1). Construct input. (2.1)
- (2). Construct three different blurry kernel A_l , A_r .(2.4)
- (3). Compute PNSR and cpu-time.(2.8)
- (4). Measure the performance of different truncation number and blurry kernel and analyze the performance.(3)
- (5). Conclude and write report.
- (6). Debug the program.

2. Main Process

2.1 Input parameters.

In this step, let the user input truncation number and picture name and which blurry kernel and svd method he want to use.

```
%input picture name,(256_256_buildings.png,640_640_lion.png,512_512_town_02.png)
X_name=input('please input the picture name ','s');
n=get_size_of_X(X_name);
%input truncation number
l_trunc=input('please input left truncation number ');
r_trunc=input('please input right truncation number ');

%input the blurry matrix you want
%blurry="original"(method1) or "shift"(method2) or "geometric"(method3);
blurry=input('please input which blurry matrix you want to use ','s');

%input the svd method you want
%svdme="builtin"(matlab built in svd) or "mySVD1"(phase A) or "mySVD2"(phase B);
svdme=input('please input which svd method you want to use ','s');
```

2.2 Data Transformation

In this problem, the first step was transform the given figure into data. Here, we use “imread()” function to transform the figure “X_name” into matrix “X” as our original picture. Then use “double()” function to convert the type of X into double type.

2.3 Judge the type of the picture and get the size of it.

In this step, we judge whether the picture is colorful or not and get the size of it.

```
function judge=colorful(X_name)
    X=imread(X_name);
    num_of_dim=ndims(X);
    if num_of_dim==2
        judge=false;
    else
        judge=true;
    end
end

function value=get_size_of_X(X_name)
    X=imread(X_name);
    n=size(X);
    value=n(1);
end
```

2.4 Blurring Kernel Construction

In this part, we build A_l and A_r as blurring kernels using following different method. Method 1:

Using T_k in previous exercise ($A_l = A_r = T_k$) :

Observing that the power of $T_k = 2 \times$ the number of diagonals + 1, so if we want more diagonals, we can just input a larger k .

Then let $T_{kl} = T_{kr} = T_k$.

```
function [Tkl Tkr]=originalTk(n,kl,kr)
    a=linspace(1/4,1,1/4,1,n-1);
    d=linspace(2.1/4,1,2.1/4,1,n);
    T=zeros(n);
    for i = 1:(n-1)
        T(i,i)=d(i);
        T(i+1,i)=a(i);
        T(i,i+1)=a(i);
    end
    T(n,n)=d(n);
    Tkl=T;
    Tkr=T;

    for i=1:(kl-1)
        Tkl=Tkl*T;
    end
    for i=1:(kr-1)
        Tkr=Tkr*T;
    end
end
```

Method 2:

Construct different asymmetric A_l and A_r by create different diagonals.

Here, we construct different T_{kl} and T_{kr} by construct three different diagonals.

For T_{kl} , it was larger n_2 on left subdiagonal and smaller n_3 on right subdiagonal.

For T_{kr} , it was larger n_2 on right subdiagonal and smaller n_3 on left subdiagonal.

```
function [Tkl Tkr]=shiftTk(n,kl,kr,n1,n2,n3)
    a=linspace(n2,n2,n-1);
    a1=linspace(n3,n3,n-1);
    d=linspace(n1,n1,n);
    Tl=zeros(n);
    Tr=zeros(n);
    for i = 1:(n-1)
        Tl(i,i)=d(i);
        Tr(i,i)=d(i);
        Tl(i+1,i)=a(i);
        Tr(i+1,i)=a1(i);
        Tr(i,i+1)=a(i);
        Tl(i,i+1)=a1(i);
    end

    Tl(n,n)=d(n);
    Tr(n,n)=d(n);
    Tkl=Tl;
    Tkr=Tr;
    for i=1:(kl-1)
        Tkl=Tkl*Tl;
    end
    for i=1:(kr-1)
        Tkr=Tkr*Tr;
    end
end
```

Method 3:

In this method, observing that in method 1, the power of $T_k = 2 \times$ the number of diagonals + 1, so I use geometric sequence to build the blurry kernel. When I build geometric sequence, I assume the common ratio $q = 2$, you can also change it easily. Using the condition " $a_1 + a_2 + \dots + a_{2n} = 1$ " I got the two simultaneous equations and got the following method.

```
function [Tkl Tkr]=geometricTk(n,q)
    %Geometric sequence
    an=(1-1/q)/(2*(1-(1/q)^n)-(1-1/q));
    T=zeros(n);
    for i = 0:(n-1)
        for j=1:(n-i)
            T(j,j+i)=an/q^i;
            T(j+i,j)=an/q^i;
        end
    end
    Tkl=T;
    Tkr=T;
end
```

2.5 Get the blurry image

In this step, we compute the blurry image by using the blurry kernel we built in 2.3.

```
% get the blurry picture
B=[];
if colorful(X_name)
    for i=1:3
        B(:,i)=A_l*X(:,i)*A_r;
    end
else
    B=A_l*X*A_r;
end
% show the blurry image
subplot(1,3,2);imshow(B),title('blurry picture')
```

2.6 Compute the pseudoinverse of A_l , A_r and recover the blurry image

In this step, we use MATLAB built-in function “svd” to compute pseudoinverse of A_l , A_r . Then using the truncated SVDs with some truncation number.

```
% get SVD decomposition of A_l, A_r from built-in svd
if svdme=='builtin'
    [U_l,sigma_l,V_l]=svd(A_l);
    [U_r,sigma_r,V_r]=svd(A_r);
    fprintf("When we use the svd result in built-in function svd")
elseif svdme=='mySVD1'
    [U_l,sigma_l,V_l]=mySVD1(A_l);
    [U_r,sigma_r,V_r]=mySVD1(A_r);
    fprintf("When we use the svd result in phaseA")
else
    [U_l,sigma_l,V_l]=mySVD2(A_l);
    [U_r,sigma_r,V_r]=mySVD2(A_r);
    fprintf("When we use the svd result in phaseB")
end

% recover the original image
X_trunc=[];
if colorful(X_name)
    for i=1:3
        X_trunc(:,i)=A_l_pseudoinverse*B(:,i)*A_r_pseudoinverse;
    end
else
    X_trunc=A_l_pseudoinverse*B*A_r_pseudoinverse;
end

% determine the value of truncation number
l_trunc=trunc;
r_trunc=trunc;
%compute the pseudoinverse of A_l, A_r
A_l_pseudoinverse=zeros(n);
A_r_pseudoinverse=zeros(n);
for i=1:l_trunc
    A_l_pseudoinverse=A_l_pseudoinverse+V_l(:,i)*U_l(:,i)'/sigma_l(i,i);
end
for i=1:r_trunc
    A_r_pseudoinverse=A_r_pseudoinverse+V_r(:,i)*U_r(:,i)'/sigma_r(i,i);
end
```

2.7 Get the singular value

In this step, we store A_l and A_r 's singular value in sigma_l.mat and sigma_r.mat .

```
% save the singular values of A_l,A_r
save sigma_l
save sigma_r
```

2.8 Measure the quality of reconstruction and cpu-time

In this step, we compute PSNR to measure the quality of reconstruction and use MATLAB built-in tic and toc to measure our code's cpu-time.

```
tic
toc
time=toc;
% compute PSNR and cputime
XF=X_trunc-X;
if colorful(X_name)
    num1=trace(XF(:,1).'*XF(:,1));
    num2=trace(XF(:,2).'*XF(:,2));
    num3=trace(XF(:,3).'*XF(:,3));
    num=num1+num2+num3;
else
    num=trace(XF);
end
PSNR=10*log10(n^2/num);
```

2.9 Packaging all method as a function

In this step, we package all method used before to be a function to help us analyze different truncation number.

```
function [time,PSNR]=deblur_and_recover_picture(X_name,Tk,l,Tkr,l_trunc,r_trunc,svdme)
```

2. Main results, Observations and Discussions

3.1 256_256_buildings.png:

3.1.1 Using method 1 (i.e.original Tk)

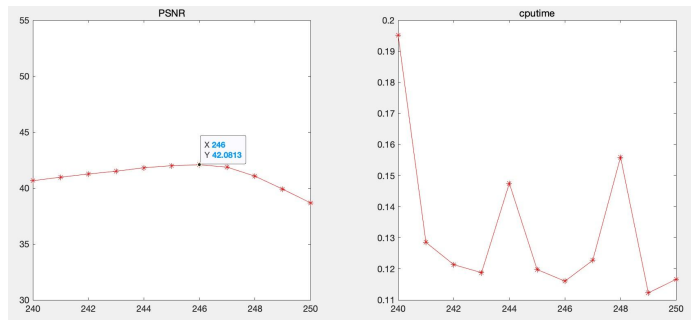
When I use truncation number = 210, Tk as blurry kernel and $k_l = k_r = 6$, the result id the following:



PSNR = 34.116726, cpu-time = 0.131111s.

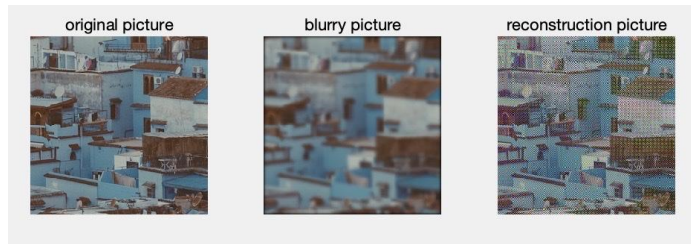
When $k_l = k_r = 1, 2, 3, 4, 5 \dots$ we also have the same PSNR and similar cpu-time.

For $k_l = k_r = 5$, we can get the largest PSNR when truncation number = 246. The cpu-time has little difference. You can also try other k 's and truncation numbers. (You can check this code in evaluating.m)



But when $k_l = k_r \geq 9$, the picture was too blurry to be reconstructed.

When $k_l = k_r = 9$, look at the following picture, figure 3 has some strange colorful points.



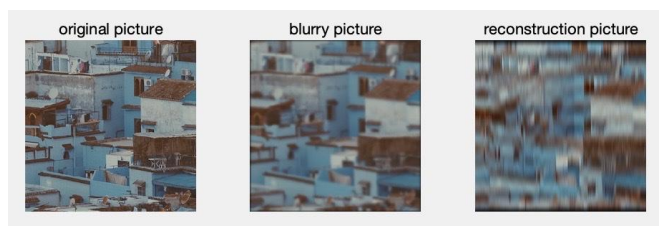
Discussion:

1. In this method, the larger the k is, the more blurry of the second picture, and the harder to recover it.
2. When k is small and truncation number is fixed at a enough large value (e.g.210), the PSNR are same.
3. When the difference between l_trunc and r_trunc is too large, we can't get a good reconstruction.

$l_trunc = 210, r_trunc=20$:



$l_trunc = 20, r_trunc=210$:

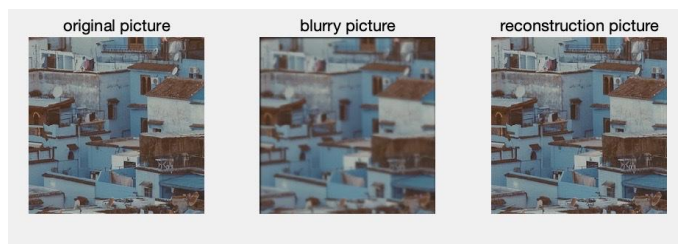


3.1.2 Using method 2 (i.e.shift Tk)

When I use shifted blurry kernel I built in method 2 and let left and right truncation number = 210, $n_1 = 0.6$, $n_2 = 0.3$, $n_3 = 0.1$,

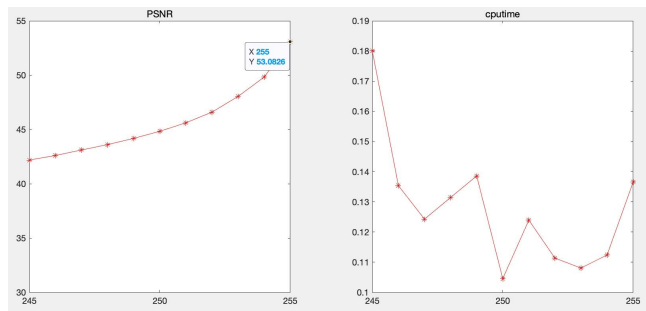
$k_l = k_r = 5$, I got the following result. You can input different parameters to build different blurry kernel.

And note that $n_1 + n_2 + n_3 = 1$, $n_2 \geq n_3$.



PSNR = 33.091477, cpu-time=0.172891s.

Using $n_1 = 0.6$, $n_2 = 0.3$, $n_3 = 0.1$, $k_l = k_r = 5$, I got the largest PSNR at truncation number = 255, which is 53.0826.



Discussion:

1. The larger the n_1 you input, it means the blurry matrix is closer to identity matrix, so the blurry degree is smaller.
2. The larger the n_2 and n_3 you input, it means the blurry matrix is more dispersive, so the blurry degree is larger.
3. If you input $n_3 = 0$, then T_{kl} only has diagonal and left subdiagonals.
4. If you input $n_2 = n_3$, then it was similar to method 1, which only change the parameter in the diagonals.
5. In this method, the blurry picture move to right and downside slightly.

And the larger k_l is, the more distance it moves down. When $k_l = 1$ and $k_r = 8$:



The larger k_r is, the more distance it moves to right. When $k_l = 8$ and $k_r = 1$:



And it is also true if we take $n_2 = n_3$: if $k_l > k_r$, then the picture will move down, if $k_l < k_r$, then the picture will move to right.

You can also try other n_1, n_2, n_3, k_l, k_r and the result is corresponding to the discussion.

6. Remember in method 1, we cannot get good reconstruction when truncation number = 210 and $k \geq 9$, but using this method2, I find that we can recover the picture successfully when $k=9$, and until $k \leq 15$, this method will always give a good reconstruction.



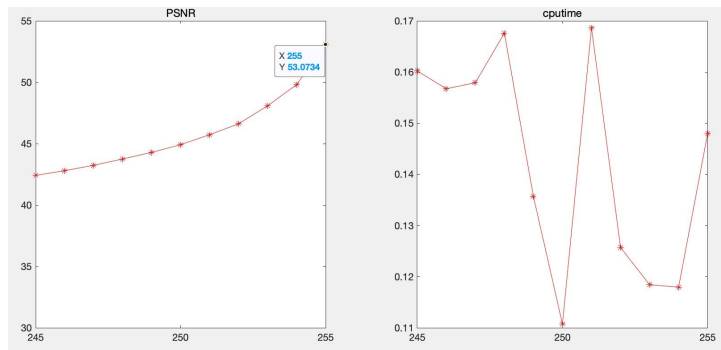
3.1.3 Using method 3 (i.e. geometric T_k)

When I use geometric sequence blurry kernel I built in method 3 and truncation number = 210, $q = 2$, I got the following result.



PSNR = 34.347611, cpu-time = 0.198642s.

Using this method, I get the biggest PSNR when truncation number = 255, which is 53.0734.



Discussion:

1. The reason that I use this geometric sequence is that I observe that in method 1, when I multiple T to Tk, the number of diagonals is plus 2, it's similar to geometric sequences.
2. All these three method of building blurry matrix have good performance and the PSNR and cpu-time of different method didn't have very large difference.
3. Using MATLAB built-in `svd()` function, we can deblur the picture rapidly.

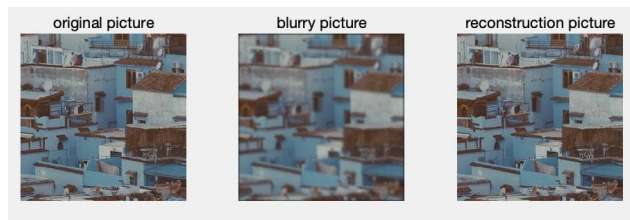
3.1.4 Using methods in problem 1

Using methods built in problem1, we can also get the result successfully, but the speed is much slower.(Using method1 as an example)

Phase A: When $l_trunc = r_trunc = 210$, $kl = kr = 5$, PSNR = 34.116726, cpu-time = 631.753492s.



Phase B: When $l_trunc = r_trunc = 210$, $kl = kr = 5$, PSNR = 34.116726, cpu-time = 750.135112s.



Using two svd methods in problem1, we can also recover the blurry picture successfully. But it was much slower.

And due to it's speed, we will not test this method in following examples.

3.2 640_640_lion.png

For this picture, I will show my result precisely because the theory is similar.

3.2.1 Using method 1 (i.e.original Tk)

When $k = 15$ and truncation number = 500:



PSNR = 27.993494, cpu-time = 0.739998s.

You can measure performance of different k and truncation number easily.

3.2.2 Using method 2 (i.e.shift Tk)

When I use method 2 and let left and right truncation number = 500, $n_1 = 0.6$, $n_2 = 0.3$, $n_3 = 0.1$, $k_l = k_r = 15$, I got the following result. You can input different parameters to build different blurry kernel.



PSNR = 27.780650, cpu-time = 0.774896s.

3.2.3 Using method 3 (i.e.geometric Tk)

When I use geometric sequence blurry kernel I built in method 3 and truncation number = 210, $q = 2$, I got the following result.



PSNR = 28.040590, cpu-time = 0.633339s.

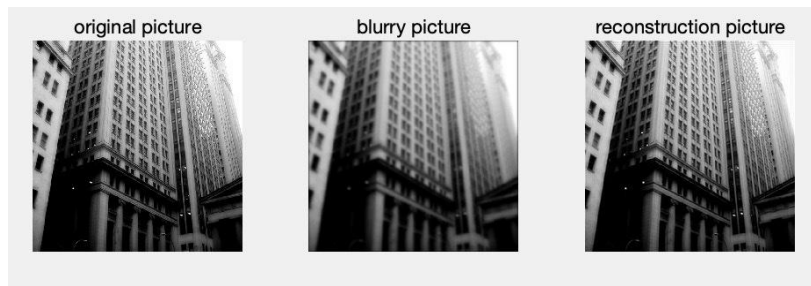
3.3 512_512_town_02.png

For this picture, I will show my result precisely because the theory is similar.

It is a picture without color, but our algorithm can also work.

3.2.1 Using method 1 (i.e.original Tk)

When $k_l = k_r = 6$ and truncation number = 450:

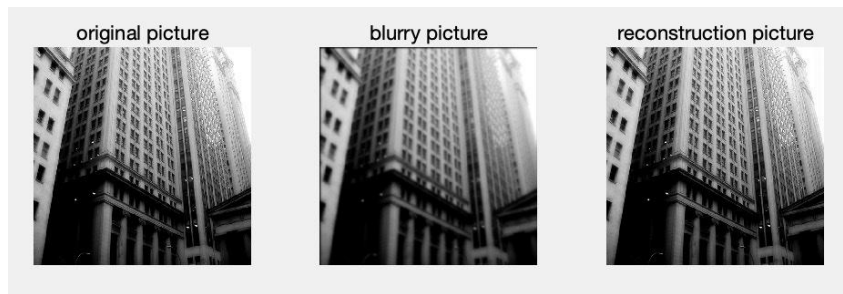


PSNR = 64.759155, cpu-time = 0.312406s.

You can measure performance of different k and truncation number easily.

3.3.2 Using method 2 (i.e.shift Tk)

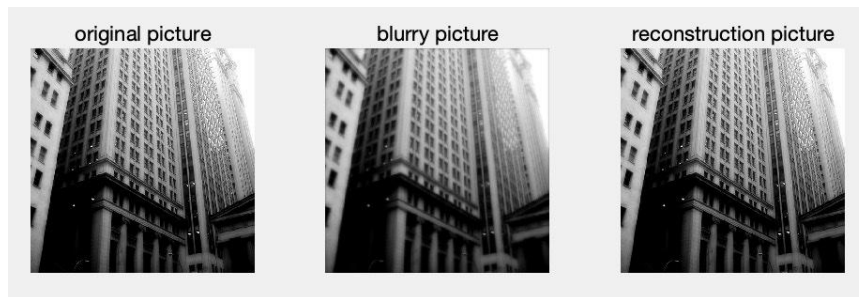
When I use method 2 and let left and right truncation number = 450, $n_1 = 0.6$, $n_2 = 0.3$, $n_3 = 0.1$, $k_l = k_r = 12$, I got the following result. You can input different parameters to build different blurry kernel.



PSNR = 61.708360, cpu-time = 0.317298s.

3.3.3 Using method 3 (i.e.geometric Tk)

When I use geometric sequence blurry kernel I built in method 3 and truncation number = 450, $q = 2$, I got the following result.



PSNR = 62.162049, cpu-time = 0.311370s.

Part3

Xinyang Feng
YiFan Tan

1 Overview

This is part of the project is about extracting the background information(image) from some video data.

We utilize power iteration method based on singular value decomposition(SVD) to get the main background information of the (converted) video data, after which we visualize the background information.

2 Main Process and Focus

The Major procedure for solving this part focuses on:

- 1. Data Transformation: We use MATLAB commands VideoReader and Read to transform information of each frame of the video $M_i \in \mathbf{R}^{m \times n \times 3}$ to one single matrix. Since this is a large scale problem, in order to save storage, we convert the video frames from RGB to gray(more discussion will be in Section 3). After that, we get matrix $A \in \mathbf{R}^{mn \times s}$, where m and n is the height and width, s is the number of frames, and each column of A here contains information of each frame.
- 2. Single Value Decomposition: We implement two approaches to apply singular value decomposition(SVD) to matrix A

Approach 1 Normalized power-iteration with Rayleigh quotient, which returns the largest eigenpair of matrix $A^T A$. By using the formula $\sigma^k = (x^k)^T A^T A x^k$, the Rayleigh quotient σ^k will converge to the largest eigenvalue of $A^T A$, and by using the formula $x^{k+1} = A^T A x^k$, iterate x^k will converge to the corresponding eigenvector. Since the height m is much larger than the width n , the largest eigenpair of matrix $A^T A$ is just the largest singular value of the matrix A

Approach 2 The standard inbuilt command $[U, S, V] = \text{svds}(A, 1)$ in MATLAB, which return the largest singular value and associated singular pair.

The largest singular value σ_1 and associated singular vector u_1, v_1 store the main background information. By using the formula:

$$\text{vec}(B) = \sigma_1 (v_1^T \mathbf{e}_1) \cdot u_1 \quad (1)$$

we get the extracted background information matrix B.

- 3. Visualization: By using the command imshow in MATLAB, we depict the background information and show several frame images to make comparsion.

3 Main results, Observations and Discussions

We have tested out three different videos in the video datasets using two approaches. The main result is that both approaches give success extraction, which shows that the idea of SVD and power iteartion can help to solve such problem.



(a) frame M_{50}



(b) frame M_{150}



(c) frame M_{300}



(d) frame M_{350}



(e) background(approach1)



(f) background(approach2)

Figure 1: sanfrancisco_01(360×640)

Model	Running time(s)		
	sanfrancisco_01(360×640)	rooster_01(360×640)	street(2560×1440)
Approach1	2.4324	2.2776	45.7813
Approach2	4.1330	4.7351	142.0451

Table 1: running time

Due to page limit, figure result of only one video is shown here. The other two are included the Appendix. As shown in Figure 1, (e)and(f) depict the extracted

background by using Approach1 and Approach2, and both approaches perform well in terms of extraction quality.

As shown in Table 1, Approach1 perform better in terms of the running time. And the difference between the two in running time increases as the scale of the video increases. This power-iteration based method runs faster than MATLAB inbuilt command, which shows that such inbuilt function is not always the most efficient.

However, when the objects in the video stay still for too long, the extraction may not work so well. The algorithm returns approximation of the most common(dominant) number in the information matrix, which means that it may be "fooled" if all frames doesn't have too much common information.

We used "RGB to gray" command to transform image data. This is helpful for the sake of saving storage. However, the cost is that we can not transform gray back to RGB, which means that we can only extract the grayscale background image. An alternative way can be done by applying the same procedure to each color of RGB, then we can get the background image in RGB.

In conclusion, this algorithm can return and calculate the approximated background image for general given videos. The idea of singular value decomposition and power iteration can be implemented to solve similiar problems.