Assignment 5

CPSC 302 - 2022W1

Devam Sisodraker
69899771

Problem 1

Problem 1
Consider the 2×2 matrix

$$A = \begin{pmatrix} 2 & -1 \\ -1 & 2 \end{pmatrix},$$

and suppose we are required to solve $A\vec{x} = \vec{b}$, where $\vec{b}$ is an arbitrary right-hand side vector. Clearly, solving a 2×2 linear system using a stationary scheme is an utterly ridiculous idea, but the following is helpful in understanding more about convergence of stationary methods.

(a) Find the spectral radius of the Jacobi and Gauss-Seidel iteration matrices and the asymptotic rate of convergence for these two schemes, namely $-\log_{10} \rho(T)$, where $\rho(T)$ denotes the spectral radius of the corresponding iteration matrix, T.

(b) How much faster does Gauss-Seidel converge compared to Jacobi for a fixed reduction in the relative residual norm, $\frac{\|\vec{r}_k\|_2}{\|\vec{b}\|_2}$, in terms of iteration counts?

(c) Write down the SOR iteration matrix as a function of the relaxation parameter, ω.

(d) Find the optimal SOR parameter, ω_{opt}, and the spectral radius of the corresponding iteration matrix.

(e) Approximately how much faster does SOR with ω_{opt} converge compared to Jacobi?

Solution (a).

Given A let us first calculate the Jacobi iteration matrix of A, denoted D. Since D all zeros along the non-diagonal entries and shares the same diagonal entries as A then we conclude that D is the following. We also indicate its inverse below.

$$D = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix} \qquad\qquad D^{-1} = \begin{pmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{pmatrix}$$

Since we have a computed value for D^{-1} we can now calculate the iteration matrix T_{Jacobi}.

$$\begin{aligned} T_{\text{Jacobi}} &= I - D^{-1}A \\ &= \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} - \begin{pmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{pmatrix} \begin{pmatrix} 2 & -1 \\ -1 & 2 \end{pmatrix} \\ &= \begin{pmatrix} 0 & \frac{1}{2} \\ \frac{1}{2} & 0 \end{pmatrix} \end{aligned}$$

continued on next page...

Now let us calculate the eigenvalues of T_{Jacobi} to obtain the spectral radius.

$$\begin{aligned}
\det\left(T_{\text{Jacobi}} - I\lambda\right) &= 0 \\
\begin{vmatrix} -\lambda & \frac{1}{2} \\ \frac{1}{2} & -\lambda \end{vmatrix} &= 0 \\
\lambda^2 - \frac{1}{4} &= 0 \\
\lambda &= \pm\frac{1}{2}
\end{aligned}$$

We know that the largest absolute eigenvalue is $\frac{1}{2}$.

$$\rho\left(D^{-1}\right) = \lambda_1 = \frac{1}{2}$$

Now that we have a value for $\rho\left(D^{-1}\right)$, we can calculate the asymptotic rate of convergence for the Jacobi iteration matrix of A to be the following. *Note: The following results were calculated using Matlab.*

$$-\log_{10}\rho\left(D^{-1}\right) = -\log_{10}\left(\frac{1}{2}\right) \approx \mathbf{0.3010}$$

We have now calculated the asymptotic rate of convergence for the Jacobi iteration matrix of A. Now let us calculate the asymptotic rate of convergence and the spectral radius for the Gauss-Seidel iteration matrix of A.

$$E = \begin{pmatrix} 2 & 0 \\ -1 & 2 \end{pmatrix} \qquad E^{-1} = \frac{1}{2 \cdot 2 - 0 \cdot (-1)} \begin{pmatrix} 2 & 0 \\ 1 & 2 \end{pmatrix} = \begin{pmatrix} 1/2 & 0 \\ 1/4 & 1/2 \end{pmatrix}$$

Since we now have a value for E^{-1} we can now calculate the iteration matrix.

$$\begin{aligned}
T_{\text{GS}} &= I - E^{-1}A \\
&= \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} - \begin{pmatrix} 1/2 & 0 \\ 1/4 & 1/2 \end{pmatrix} \begin{pmatrix} 2 & -1 \\ -1 & 2 \end{pmatrix} \\
&= \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} - \begin{pmatrix} 1 & -1/2 \\ 0 & 3/4 \end{pmatrix} \\
&= \begin{pmatrix} 0 & 1/2 \\ 0 & 1/4 \end{pmatrix}
\end{aligned}$$

continued on next page...

Now that we have a value for T_{GS} let us calculate its eigenvalues.

$$\begin{aligned} \det\left(T_{\text{GS}} - I\lambda\right) &= 0 \\ \begin{vmatrix} -\lambda & \frac{1}{2} \\ 0 & 1/4 - \lambda \end{vmatrix} &= 0 \\ -\lambda \cdot \left(\frac{1}{4} - \lambda\right) - \frac{1}{2} \cdot 0 &= 0 \\ \lambda &= 0, \frac{1}{4} \end{aligned}$$

We know that the largest absolute eigenvalue is $\frac{1}{4}$.

$$\rho\left(E^{-1}\right) = \lambda_1 = \frac{1}{4}$$

Now let us calculate the asymptotic rate of convergence.

$$-\log_{10}\rho\left(E^{-1}\right) = -\log_{10}\left(\frac{1}{4}\right) \approx \mathbf{0.6021}$$

Solution (b).

Jacobi will take twice as many iterations as Gauss-Seidel since the asymptotic rate of Jacobi is roughly double that of Gauss-Seidel, thus we can say that Gauss-Seidel will converge at faster than Jacobi will converge.

Solution (c).

We know that the following given an SOR iteration.

$$M = \frac{1-\omega}{\omega} D + E$$

Let us substitute M into the function T_{SOR} where $T_{\text{SOR}} : \mathbb{R}^{n \times n} \to \mathbb{R}^{n \times n}$.

$$\begin{aligned} T_{\text{SOR}}(\omega) &= I - M^{-1}A \\ &= I - \left(\frac{1-\omega}{\omega} D + E \right)^{-1} A \end{aligned}$$

Thus we have the following function.

$$T_{\text{SOR}} = I - \left(\frac{1-\omega}{\omega} D + E \right)^{-1} A$$

Solution (d).

Because we already have a value for ρ_{Jacobi}, let us substitute it into the ω_{opt} equation from the textbook.

$$\begin{aligned}\omega_{opt} &= \frac{2}{1+\sqrt{1-(\rho_J)^2}} \\ &= \frac{2}{1+\sqrt{1-\left(\frac{1}{2}\right)^2}} \\ &\approx \mathbf{1.0718}\end{aligned}$$

Thus we obtain $\omega_{opt} \approx \mathbf{1.0718}$.

Solution (e).

From the textbook we know that $\rho_{\text{SOR}} = \omega_{opt} - 1$. Thus we know that $\rho_{\text{SOR}} = 0.0718$. Let us now calculate an asymptotic rate of convergence.

$$-\log_{10}(\rho_{\text{SOR}}) \approx \mathbf{1.1439}$$

Let us note that the asymptotic rate of convergence for SOR is around 4 times that of Jacobi's. Thus we can say that SOR uses 4 times fewer iterations than Jacobi when $\omega = \omega_{opt}$ is used for SOR. We can conclude that SOR converges 4 times faster than Jacobi.

Problem 2

Problem 2

(a) Download from the assignment webpage the file `J_GS.m`. In your MATLAB command line, run the command: `J_GS(100);`
Include in your assignment solution the convergence graph that you are seeing. This is a linear system with the two-dimensional Laplacian of size $10,000 \times 10,000$ (we have $n = 100, n^2 = 10,000$), and we are plotting the norm of the relative residual after 10,000 iterations for Jacobi and Gauss-Seidel. There is no reason to be impressed with this graph; convergence here is slow. But we are going to see the effect of using SOR.

(b) Given the eigenvalues of the Laplacian in the slides, show that the optimal SOR parameter can be expressed as

$$\omega_{\text{opt}} = \frac{2}{1 + \sin\left(\frac{\pi}{n+1}\right)}.$$

(c) Modify the MATLAB function so that in addition to the Jacobi and Gauss-Seidel graphs (for the same matrix) a convergence plot for the SOR method is included as well. Use the same initial guess (the zero vector) and the same stopping criterion: $\frac{\|\vec{r}_k\|_2}{\|\vec{b}\|_2} < 10^{-6}$.

Tip: You should be seeing an *extremely dramatic* improvement in convergence. If you are not seeing such an improvement, then you must have done something wrong.

(d) Explain your results.

Solution (a).

Note: The line

```
saveas(gcf, "DevamSisodraker_2a.jpg", "jpg");
```

was appended to the end of `J_GS.m` to ease the exporting process.

File: `J_GS.m`

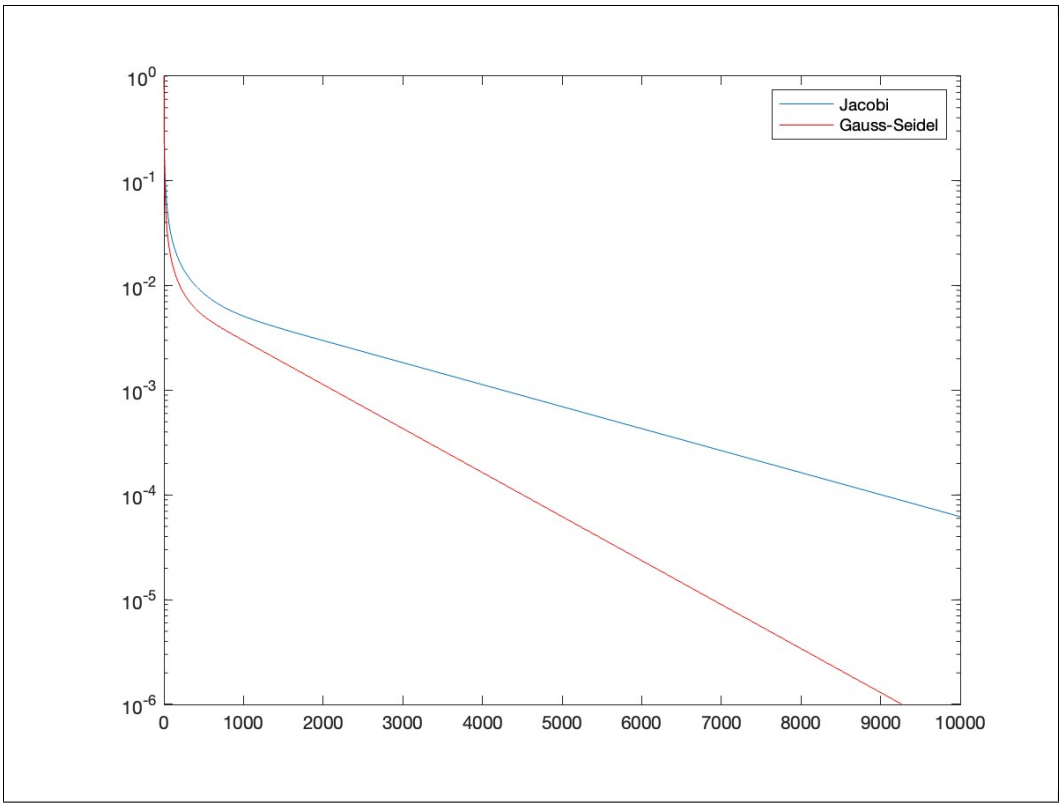
```
function J_GS(n)
close all
gcf
% Apply a stationary method to a linear system involving the n^2-by-n^2
% Laplacian

A=delsq(numgrid('S',n+2));
D=diag(diag(A));    % M=D for Jacovi
E=tril(A);          % M=E for Gauss-Seidel
itermax=10000;     % maximum number of iterations
resvecJ=zeros(itermax,1);  % allocate initial space for Jacobi residual norm vector
resvecGS=zeros(itermax,1); % allocate initial space for Gauss-Seidel residual vector

% Jacobi
xJ=zeros(n^2,1);
b=A*ones(n^2,1);    % generate a solution of all 1s and a right-hand side
nb=norm(b);
r=b;
for i=1:10000
    resvecJ(i)=norm(r)/nb;  % relative residual norm
    if resvecJ(i)<1e-6, break,end  % terminate loop if stopping criterion is satisfied
    xJ=xJ+D\r;              % next iterate
    r=b-A*xJ;               % update residual
end
semilogy(resvecJ);          % plot relative residual norm for Jacobi

% Gauss-Seidel
xGS=zeros(n^2,1);
r=b;
for i=1:10000
    resvecGS(i)=norm(r)/nb; % relative residual norm
    if resvecGS(i)<1e-6, break,end  % terminate loop if stopping criterion is satisfied
    xGS=xGS+E\r;            % next iterate
    r=b-A*xGS;              % update residual
end
hold on
semilogy(resvecGS,'r');     % plot relative residual norm for Gauss-Seidel
legend('Jacobi','Gauss-Seidel')
saveas(gcf, "DevamSisodraker_2a.jpg", "jpg");
```

continued on next page...

File: DevamSisodraker_2a_out.txt

```
>> J_GS(100)

ans =

  Figure (1) with properties:

      Number: 1
        Name: ''
       Color: [0.9400 0.9400 0.9400]
    Position: [584 595 560 420]
       Units: 'pixels'

  Show all properties

>>
```

continued on next page...


Jacobi
Gauss-Seidel
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000

Solution (b).

The following is given in the lecture slides.

$$\lambda_{i,j} = 4 - 2\left(\cos\frac{i\pi}{n+1} + \cos\frac{j\pi}{n+1}\right)$$

Before we continue, let us first recall that the spectral radius of T is given by $\mu_{1,1}$. Thus it follows that $\rho(T) = \mu_{i,i} = \cos\frac{\pi}{n+1}$. Then let us substitute this $\rho(T)$ into the equation which defines ω_{opt}.

$$\begin{aligned}
\omega_{opt} &= \frac{2}{1+\sqrt{1-\big(\rho(T)\big)^2}} \\
&= \frac{2}{1+\sqrt{1-\left(\cos\frac{\pi}{n+1}\right)^2}} && \text{substitution of } \rho(T) \\
&= \frac{2}{1+\sqrt{\left(\sin\frac{\pi}{n+1}\right)^2}} && \text{by Pythagorean trigonometric identity} \\
&= \frac{2}{1+\sin\frac{\pi}{n+1}}
\end{aligned}$$

Solution (c).

File: DevamSisodraker_2c.m

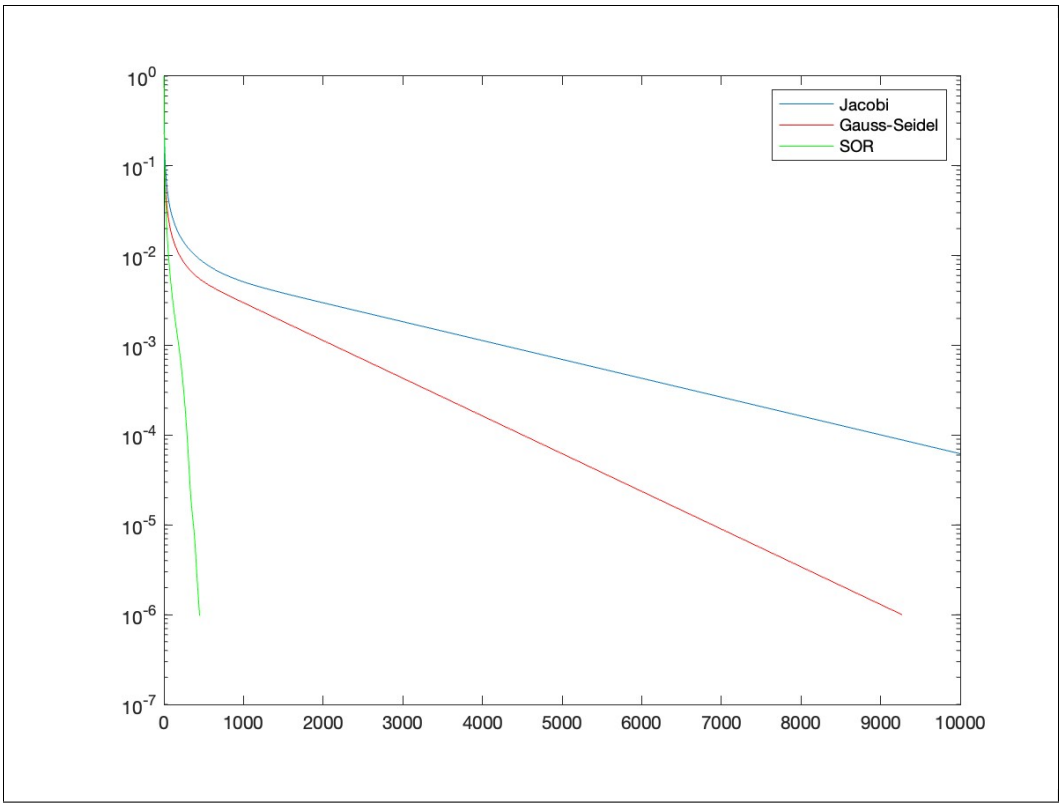
```
function J_GS(n)
close all
gcf
% Apply a stationary method to a linear system involving the n^2-by-n^2
% Laplacian

A=delsq(numgrid('S',n+2));
D=diag(diag(A));   % M=D for Jacovi
E=tril(A);         % M=E for Gauss-Seidel
itermax=10000;    % maximum number of iterations
resvecJ=zeros(itermax,1);  % allocate initial space for Jacobi residual norm vector
resvecGS=zeros(itermax,1); % allocate initial space for Gauss-Seidel residual vector
resvecSOR=zeros(itermax,1); % allocate initial space for SOR residual vector

% Jacobi
xJ=zeros(n^2,1);
b=A*ones(n^2,1);   % generate a solution of all 1s and a right-hand side
nb=norm(b);
r=b;
for i=1:10000
    resvecJ(i)=norm(r)/nb;  % relative residual norm
    if resvecJ(i)<1e-6, break,end  % terminate loop if stopping criterion is satisfied
    xJ=xJ+D\r;              % next iterate
    r=b-A*xJ;               % update residual
end
semilogy(resvecJ);          % plot relative residual norm for Jacobi

% Gauss-Seidel
xGS=zeros(n^2,1);
r=b;
for i=1:10000
    resvecGS(i)=norm(r)/nb; % relative residual norm
    if resvecGS(i)<1e-6, break,end  % terminate loop if stopping criterion is satisfied
    xGS=xGS+E\r;            % next iterate
    r=b-A*xGS;              % update residual
end
hold on
semilogy(resvecGS,'r');     % plot relative residual norm for Gauss-Seidel

% SOR
xSOR=zeros(n^2,1);
r=b;
for i=1:10000
    wOpt = 2 / (1 + sin(pi / (n + 1)));
    resvecSOR(i)=norm(r)/nb; % relative residual norm
    if resvecSOR(i)<1e-6, break,end  % terminate loop if stopping criterion is satisfied
    xSOR=xSOR+((1 - wOpt)*D + wOpt*E)\r;            % next iterate
    r=b-A*xSOR;              % update residual
end
hold on
semilogy(resvecSOR,'g');     % plot relative residual norm for Gauss-Seidel


legend('Jacobi','Gauss-Seidel','SOR')
saveas(gcf, "DevamSisodraker_2c.jpg", "jpg");
```

continued on next page...

File: DevamSisodraker_2c_out.txt

```
>> DevamSisodraker_2c(100)

ans =

  Figure (1) with properties:

      Number: 1
        Name: ''
       Color: [0.9400 0.9400 0.9400]
    Position: [584 595 560 420]
       Units: 'pixels'

  Show all properties

>>
```

continued on next page...


Jacobi
Gauss-Seidel
SOR
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000

Solution (d).

We can see in our graphs that SOR using the $\omega = \omega_{opt}$ converges much faster than the Jacobi or Gauss-Seidel iterative methods. We know that SOR using the optimal parameter requires only $\mathcal{O}(n)$ to iterate and $\mathcal{O}(n^2)$ calculations per iteration, thus resulting in a $\mathcal{O}(n^3)$ efficient algorithm whereas Jacobi and Gauss-Seidel require $\mathcal{O}(n^4)$ in total. This is the significant difference in time complexity shown in the graphs above.

Problem 3

Problem 3

(a) Suppose A is an n-by-n orthogonal matrix. Show that all its singular values are equal to 1.

(b) Recall that an orthogonal projector is a symmetric matrix P for which $P^2 = P$. What are the eigenvalues of an orthogonal projector?

Solution (a).

Let us suppose that A is an $n \times n$ orthogonal matrix. This means that $A^T A = I$. Because the singular values of A are simply the square roots of the eigenvalues of $A^T A$ then the singular values of A are equal to the eigenvalues of I. Since all eigenvalues of I are 1 and since $\sqrt{1} = 1$ then it holds that all $n \times n$ orthogonal matrices have all singular values equal to 1.

Solution (b).

The eigenvalues of P are represented by λ in the equation $P\vec{v} = \lambda\vec{v}$. Since $P = P^2$ then it holds that $\lambda^2\vec{v} = P^2\vec{v} = P\vec{v} = \lambda\vec{v}$. We can manipulate it to obtain the following.

$$\begin{aligned} P^2\vec{v} &= P\vec{v} \\ \lambda^2\vec{v} &= \lambda\vec{v} \\ \lambda^2 &= \lambda \\ \lambda^2 - \lambda &= 0 \\ \lambda(1-\lambda) &= 0 \\ \lambda &= 0, 1 \end{aligned}$$

Thus we conclude that orthogonal projectors can only have values that are either 0 or 1. However, since orthogonal projectors are also symmetric $n \times n$ matrices, then it holds that the only eigenvalue that an orthogonal projector can have is $\lambda = 1$.

Problem 4

Problem 4
Load the following .mat matrix that appears on the assignment page:

```
load powerMatrix;
```

If you hit `whos` you should be seeing a matrix called A, of size 100×100. To validate any of your results below, you may run the MATLAB command `eig`, as long as you understand that in typical eigenvalue computations (in a potentially more challenging computational environment) we generally do not have the luxury of running `eig` to check ourselves.

(a) Apply the power method. Terminate the iteration once the iterates satisfy

$$\lambda_1^{(k)} - \lambda_1^{(k-1)}| < 10^{-4}.$$

Your program should print out the value of the final iterate and a graph of the absolute errors: $|\lambda_1^{(k)} - \lambda_{\max}|$. For better visualization, use `semilogy` for your graphs when necessary. As an initial guess for the eigenvector use a vector produced by the MATLAB command `randn`. (When you repeat your experiments, the number of iterations may slightly vary due to the random initial guess.)

(b) Repeat your computations with the *inverse* power iteration, with a shift $\alpha = 4$, and produce the same graph as you did for the power method.

(c) Discuss the differences between the performance of the power and the inverse power methods in terms of the cost of single iterations and t overall computational cost.

(d) Suppose now that we know that A has an eigenvalue close to 3 and we are interested to compute it to six correct decimal digits. Suggest an efficient procedure for doing so. Implement your suggested algorithm and compute the eigenvalue.

Solution (a).

File: DevamSisodraker_4a.m

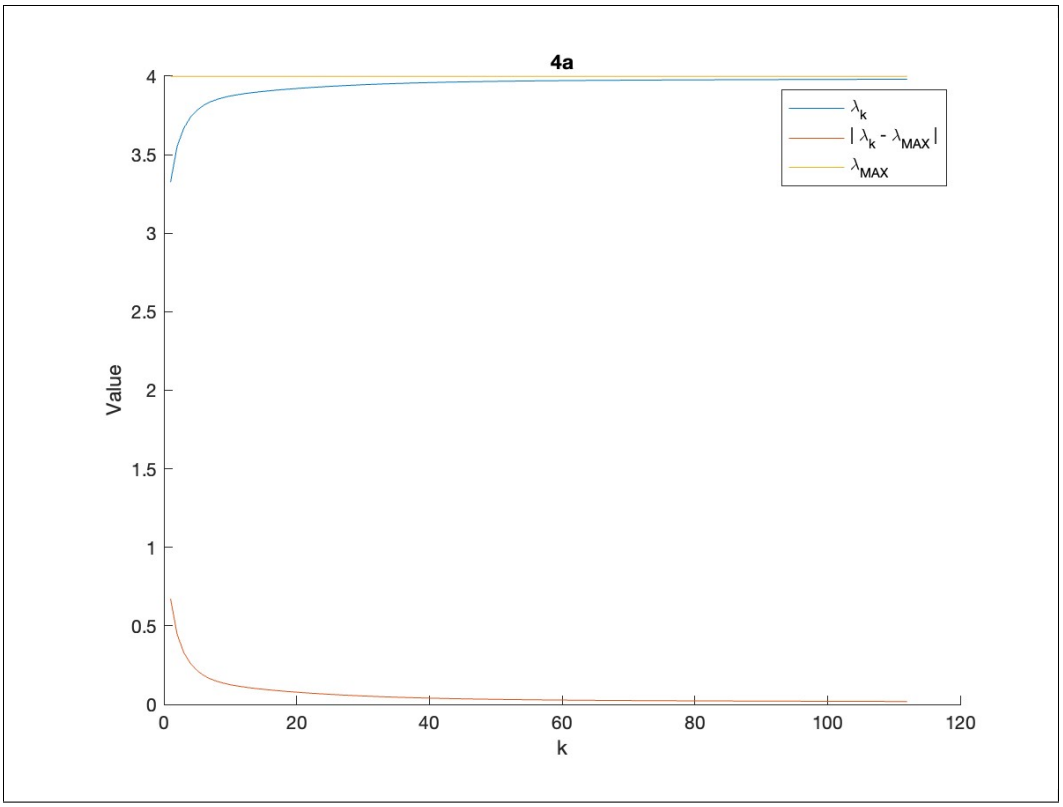
```
gcf
hold on;

clear;
load('powerMatrix.mat')

k = 0;
vector_k = randn(100, 1);
lambda_k = transpose(vector_k)*A*vector_k;
lambda_k_1 = transpose(vector_k)*A*vector_k;
lambda_audit = [];
lambda_delta_audit = [];
eigval = max(eig(A));

while true
    % statements here
    % if ~WhileCondition, break ; end
    lambda_k_1 = lambda_k;
    vector_k = A*vector_k;
    vector_k = vector_k/norm(vector_k);
    lambda_k = transpose(vector_k)*A*vector_k;
    lambda_audit = [lambda_audit, lambda_k];
    lambda_delta_audit = [lambda_delta_audit, abs(lambda_k - eigval)];
    k = k + 1;
    if abs(lambda_k_1 - lambda_k) < 10^-4
        break;
    end
end

plot(1:1:k, lambda_audit);

plot(1:1:k, lambda_delta_audit);

plot([1, k], [eigval, eigval]);

title("4a");
legend({ ...
    '\lambda_k', ...
    '| \lambda_k - \lambda_{MAX} |', ...
    '\lambda_{MAX}', ...
});
xlabel("k");
ylabel("Value");

hold off;
saveas(gcf, "DevamSisodraker_4a.jpg", "jpg");
```

continued on next page...

File: DevamSisodraker_4a_out.txt

```
>> DevamSisodraker_q4a

ans =

  Figure (1) with properties:

      Number: 1
        Name: ''
       Color: [0.9400 0.9400 0.9400]
    Position: [584 595 560 420]
       Units: 'pixels'

  Show all properties


lambda_k =

    3.9804

>>
```

continued on next page...


4a
Value
k
λ_k
$|\lambda_k - \lambda_{MAX}|$
λ_{MAX}
0
0.5
1
1.5
2
2.5
3
3.5
4
0
20
40
60
80
100
120

Solution (b).

File: DevamSisodraker_4b.m

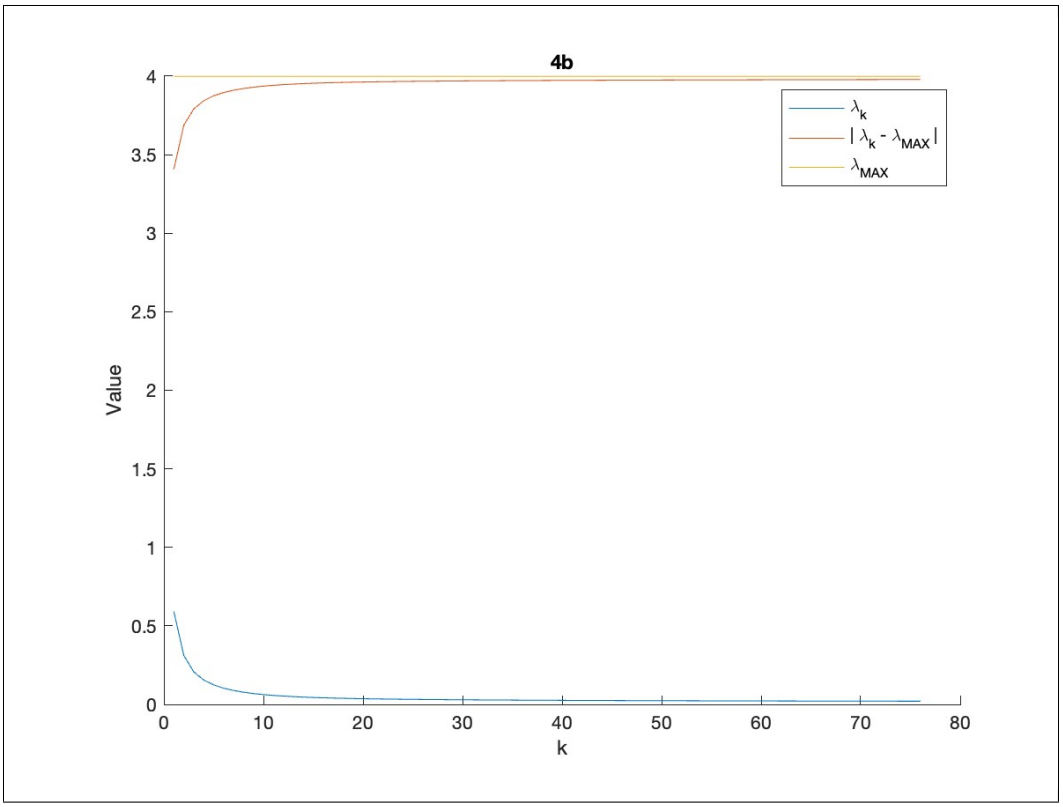
```
gcf
hold on;

clear;
load('powerMatrix.mat');

k = 0;
vector_k = randn(100, 1);
vector_k_1 = vector_k;
lambda_k = transpose(vector_k)*A*vector_k;
lambda_k_1 = transpose(vector_k)*A*vector_k;
lambda_audit = [];
lambda_delta_audit = [];
eigval = max(eig(A));
alpha = 4;

while true
    % statements here
    % if ~WhileCondition, break ; end
    lambda_k_1 = lambda_k;
    vector_k_1 = vector_k;

    vector_k = (A - alpha * eye(size(A)))/transpose(vector_k_1);
    vector_k = vector_k/norm(vector_k);
    lambda_k = transpose(vector_k)*A*vector_k;

    lambda_audit = [lambda_audit, lambda_k];
    lambda_delta_audit = [lambda_delta_audit, abs(lambda_k - eigval)];
    k = k + 1;

    if abs(lambda_k_1 - lambda_k) < 10^-4
        break;
    end
end

plot(1:1:k, lambda_audit);

plot(1:1:k, lambda_delta_audit);

plot([1, k], [eigval, eigval]);

title("4b");
legend({ ...
    '\lambda_k', ...
    '| \lambda_k - \lambda_{MAX} |', ...
    '\lambda_{MAX}', ...
});
xlabel("k");
ylabel("Value");

hold off;
saveas(gcf, "DevamSisodraker_4b.jpg", "jpg");
lambda_k
```

continued on next page...

File: DevamSisodraker_4b_out.txt

```
>> DevamSisodraker_4b

ans =

  Figure (1) with properties:

      Number: 1
        Name: ''
       Color: [0.9400 0.9400 0.9400]
    Position: [584 595 560 420]
       Units: 'pixels'

  Show all properties


lambda_k =

    0.0202

>>
```

continued on next page...


4b
Value
k
λ_k
$|\lambda_k - \lambda_{MAX}|$
λ_{MAX}
0
0.5
1
1.5
2
2.5
3
3.5
4
0
10
20
30
40
50
60
70
80

Solution (c).

Upon first glance, we notice that the inverse power method converges much faster than the power method sometimes taking around 100 times less. As with all iterative methods, the computational cost is balanced out between the number of iterations and cost per iteration. Thus, let us also note that while the inverse power method converges much faster than the power method, the inverse power method requires us to solve a linear system for each iteration while the power method only requires us to multiply a matrix by a vector, thus resulting in much heavier per iteration computational cost.

Solution (d).

We will use the inverse power method with $\alpha = 3$ to find the eigenvalue close to 3.

continued on next page...

File: DevamSisodraker_4d.m

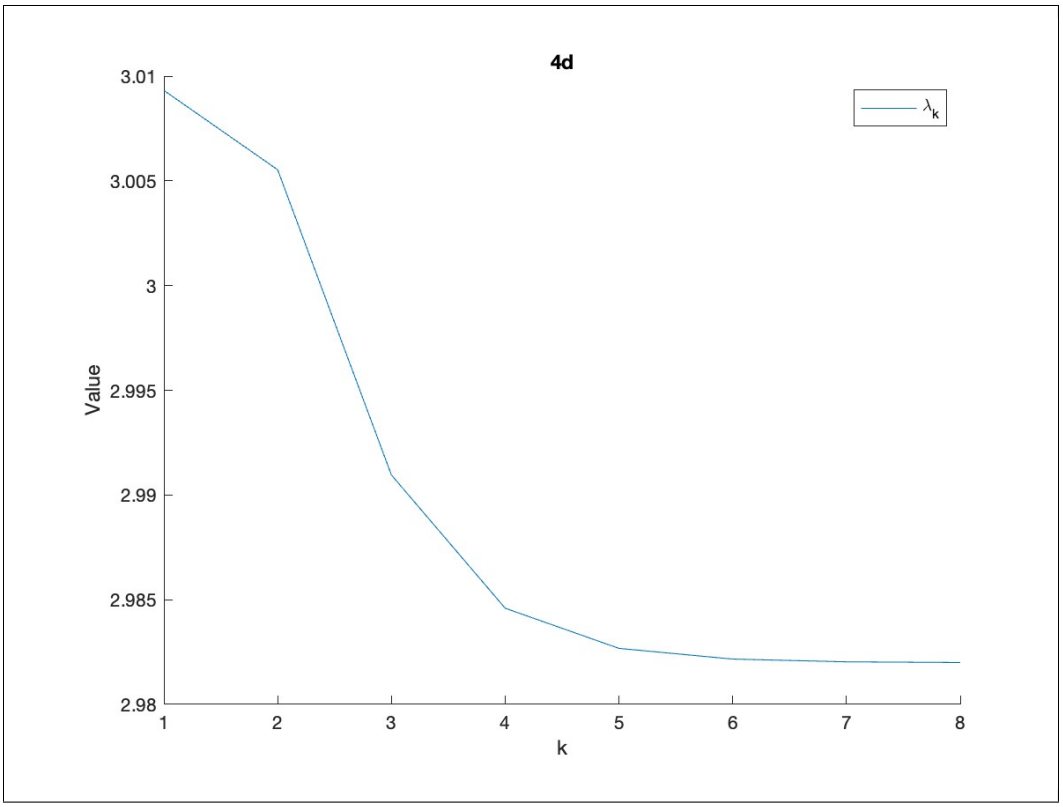
```
gcf
hold on;

clear;
load('powerMatrix.mat');

k = 0;
vector_k = randn(100, 1);
vector_k_1 = vector_k;
lambda_k = transpose(vector_k)*A*vector_k;
lambda_k_1 = transpose(vector_k)*A*vector_k;
lambda_audit = [];
alpha = 3;

while true
    % statements here
    % if ~WhileCondition, break ; end
    lambda_k_1 = lambda_k;
    vector_k_1 = vector_k;

    vector_k = (A - alpha * eye(size(A)))\vector_k_1;
    vector_k = vector_k/norm(vector_k);
    lambda_k = transpose(vector_k)*A*vector_k;

    lambda_audit = [lambda_audit, lambda_k];
    k = k + 1;

    if abs(lambda_k_1 - lambda_k) < 10^-4
        break;
    end
end

plot(1:1:k, lambda_audit);

title("4d");
legend({ ...
    '\lambda_k', ...
});
xlabel("k");
ylabel("Value");

hold off;
saveas(gcf, "DevamSisodraker_4d.jpg", "jpg");
lambda_k
```

continued on next page...

File: DevamSisodraker_4d_out.txt

```
>> DevamSisodraker_4d

ans =

  Figure (1) with properties:

      Number: 1
        Name: ''
       Color: [0.9400 0.9400 0.9400]
    Position: [584 595 560 420]
       Units: 'pixels'

  Show all properties


lambda_k =

    2.9820

>>
```

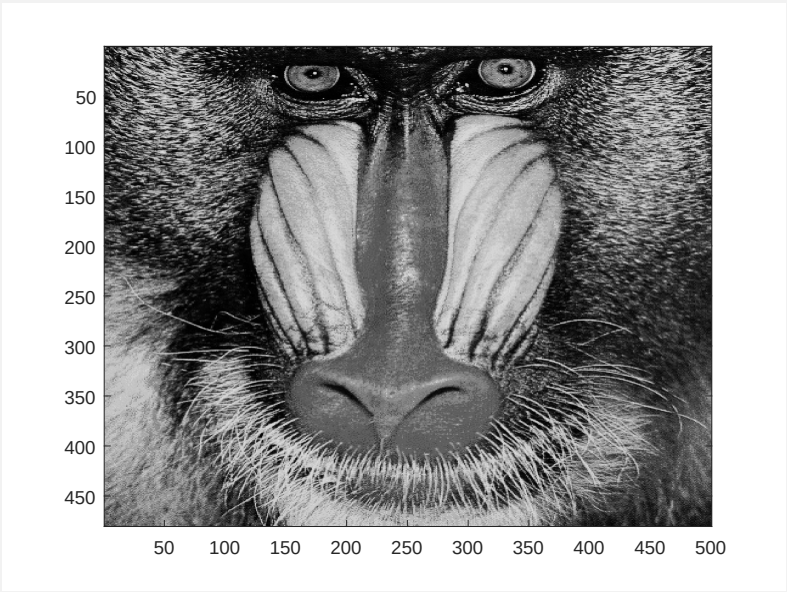
continued on next page...


4d
λ_k
3.01
3.005
3
2.995
2.99
2.985
2.98
Value
1
2
3
4
5
6
7
8
k

Problem 5

Problem 5
The following picture appears in MATLAB's repository of images, and can be retrieved by entering

```
load mandrill;
colormap('gray');
image(X);
```



(a) Print out the images generated by the truncated SVD. Start with $r = 2$ and go up by powers of 2, to $r = 2^6 = 64$ (six plots in total). For a compact presentation of your figures, you may use the command `subplot(3,2)`. (Check out `help subplot`.)

(b) Comment briefly on the quality of the images as a function of r. For what value of r would you say that the quality of the image is acceptable, in that we can be rather confident of what we are seeing? (We are not looking for a specific "correct answer" here - just make your subjective observation.)

(c) For the value of r you stated in part (b), how much storage is required? Compare it to the storage required for the original image.

Solution (a).

File: DevamSisodraker_5a.m

```
clear;
load mandrill;
colormap("gray");
[U,S,V] = svd(X);
image(U*S*V');

close all;
for i = 1:1:6
    r = 2^i;

    dims = size(X);
    S_trunc = diag(S);
    S_trunc((r + 1):min(dims(1), dims(2))) = 0;
    S_trunc = diag(S_trunc);
    S_trunc(dims(1), dims(2)) = 0;

    gcf
    hold on;
    Xout = U * S_trunc * V';
    colormap("gray");
    image(flipud(Xout));
    title(strcat("r=", string(r)));
    saveas(gcf, strcat("DevamSisodraker_5a_", string(r), ".jpg"), "jpg");
    hold off;
end
```

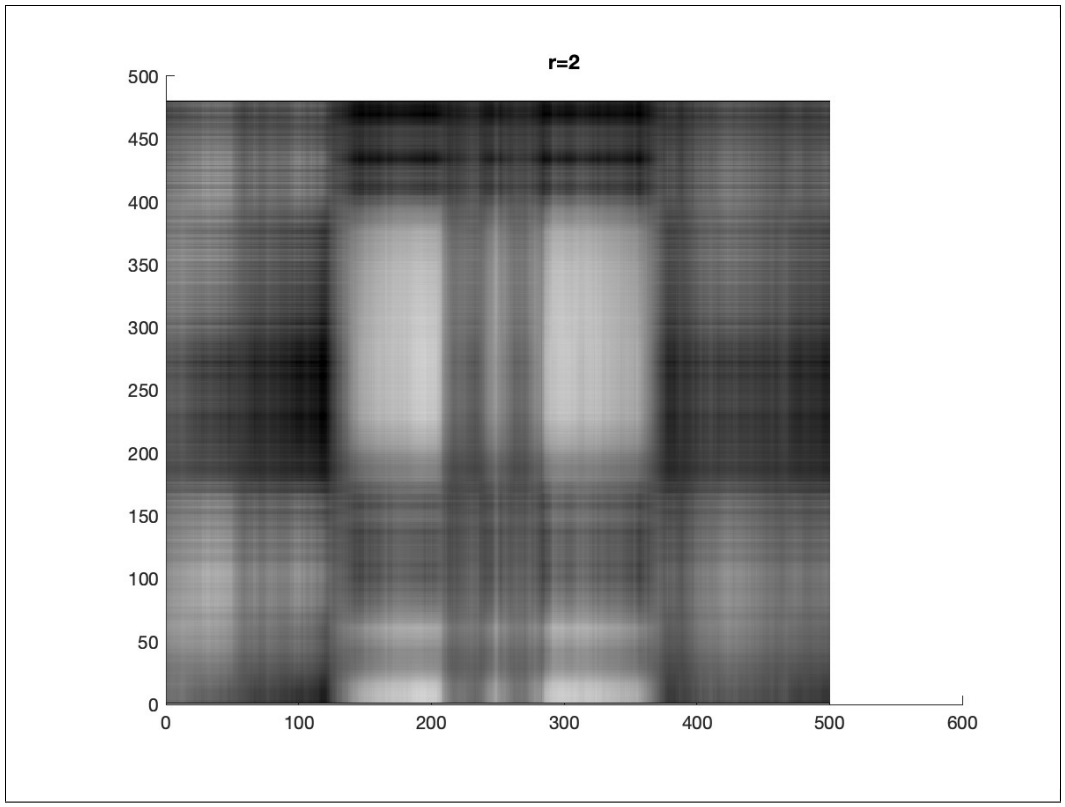
continued on next page...

File: DevamSisodraker_5a_out.txt

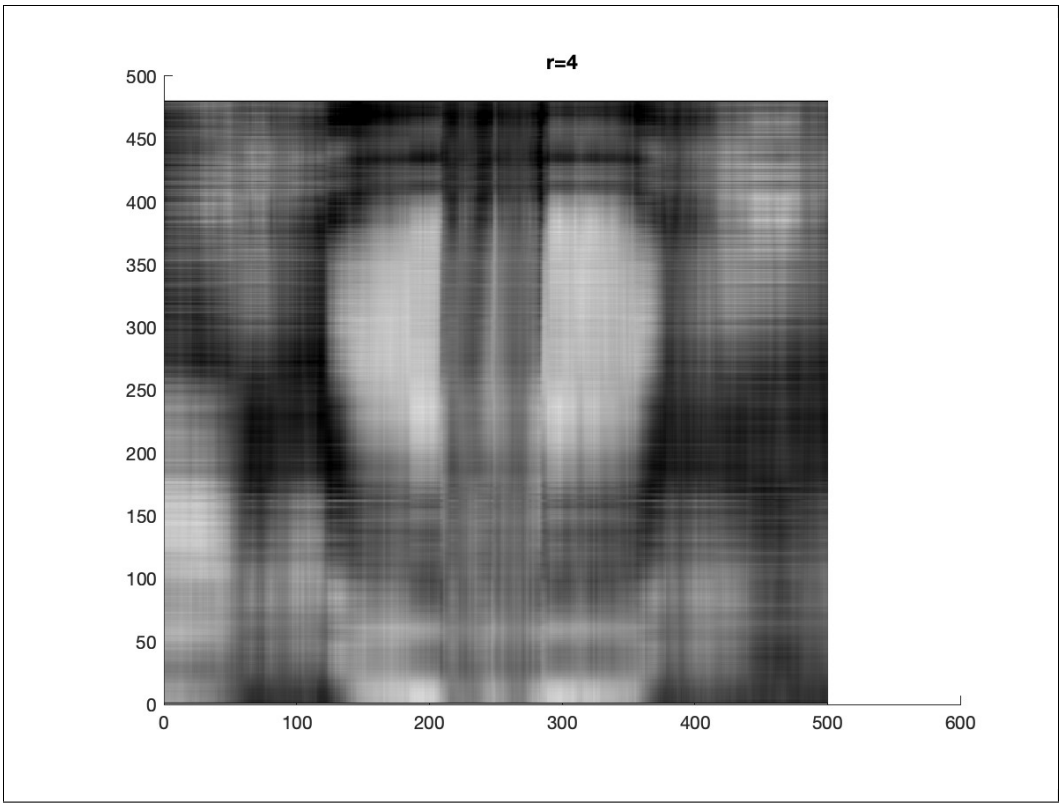
```
>> DevamSisodraker_5a

ans =

  Figure (1) with properties:

      Number: 1
        Name: ''
       Color: [0.9400 0.9400 0.9400]
    Position: [584 595 560 420]
       Units: 'pixels'

  Show all properties


ans =

  Figure (1) with properties:

      Number: 1
        Name: ''
       Color: [0.9400 0.9400 0.9400]
    Position: [584 587 560 420]
       Units: 'pixels'

  Show all properties


ans =

  Figure (1) with properties:

      Number: 1
        Name: ''
       Color: [0.9400 0.9400 0.9400]
    Position: [584 587 560 420]
       Units: 'pixels'

  Show all properties


ans =

  Figure (1) with properties:

      Number: 1
        Name: ''
       Color: [0.9400 0.9400 0.9400]
    Position: [584 587 560 420]
       Units: 'pixels'

  Show all properties


ans =

  Figure (1) with properties:
```

```
      Number: 1
        Name: ''
       Color: [0.9400 0.9400 0.9400]
    Position: [584 587 560 420]
       Units: 'pixels'

  Show all properties


ans =

  Figure (1) with properties:

      Number: 1
        Name: ''
       Color: [0.9400 0.9400 0.9400]
    Position: [584 587 560 420]
       Units: 'pixels'

  Show all properties

>> >
```

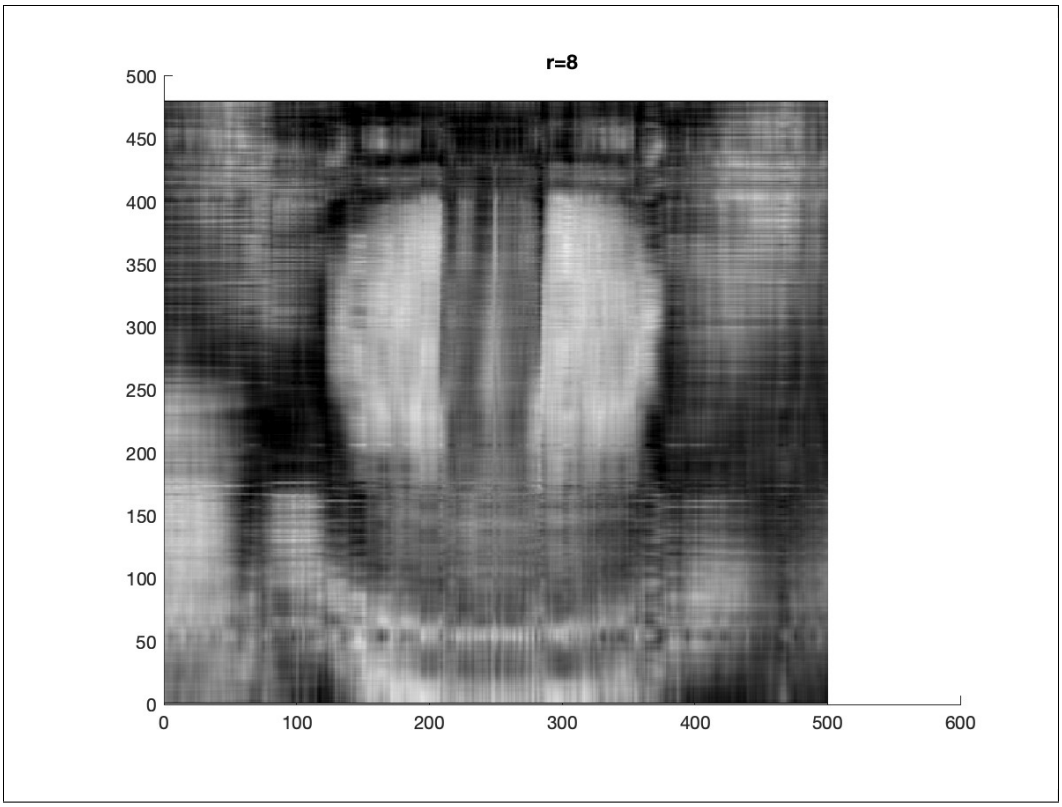
continued on next page...



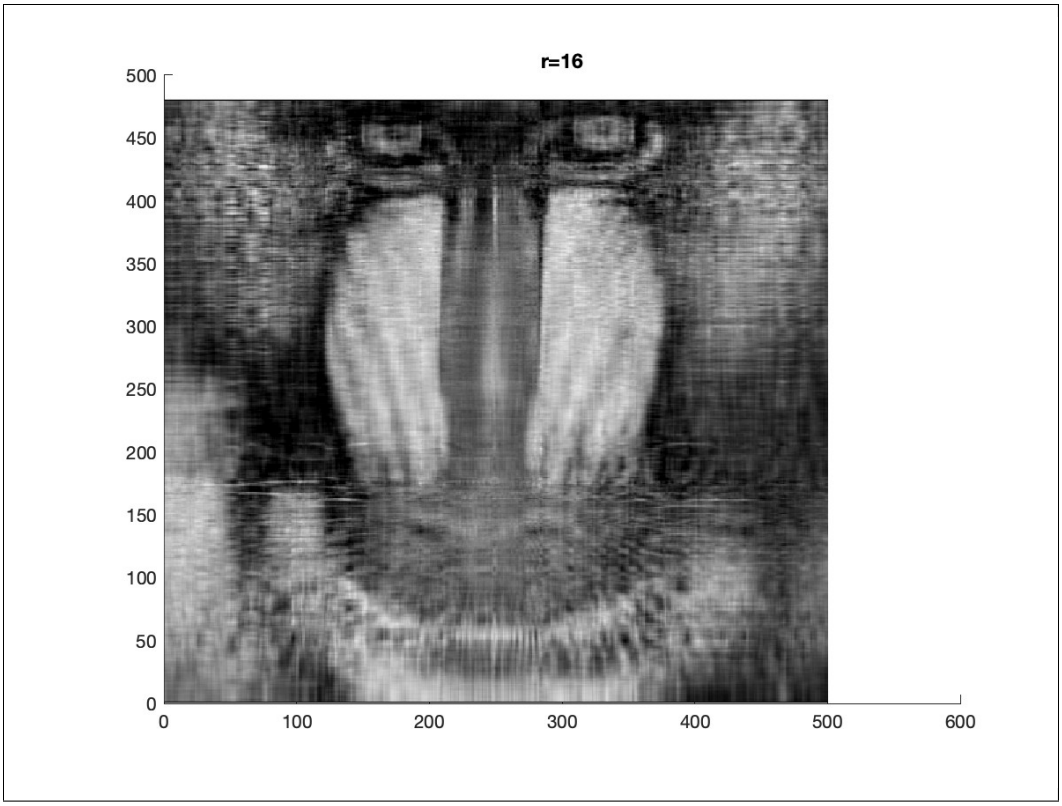
continued on next page...



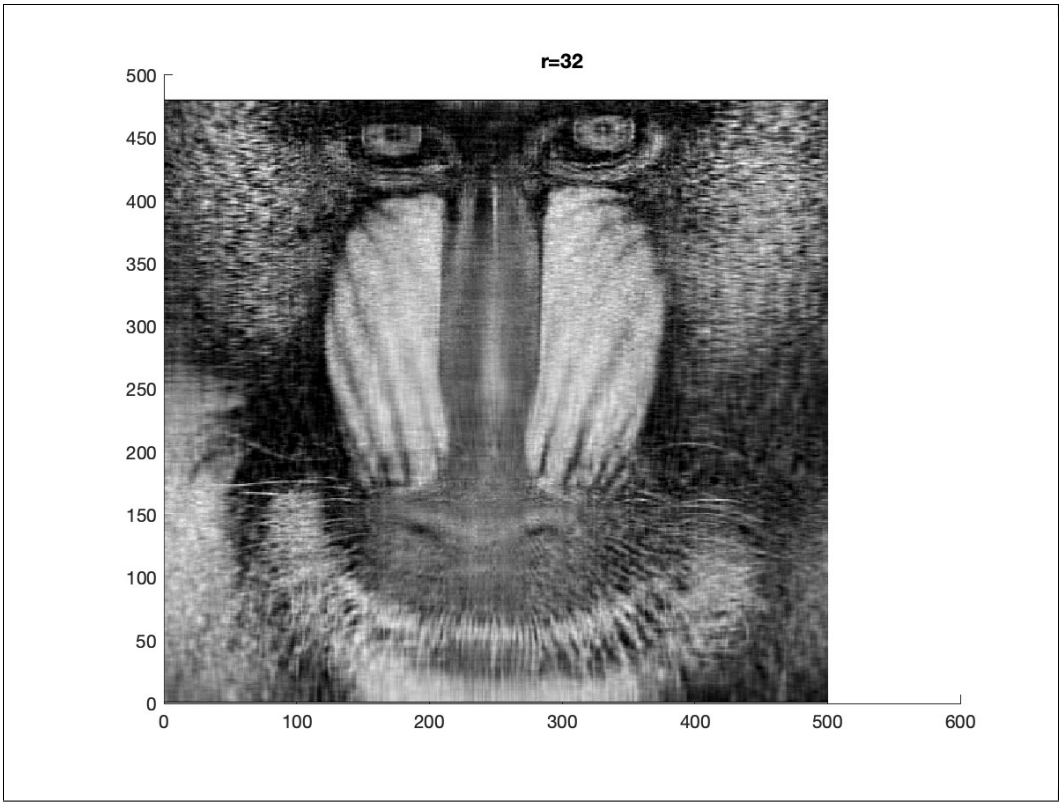
continued on next page...



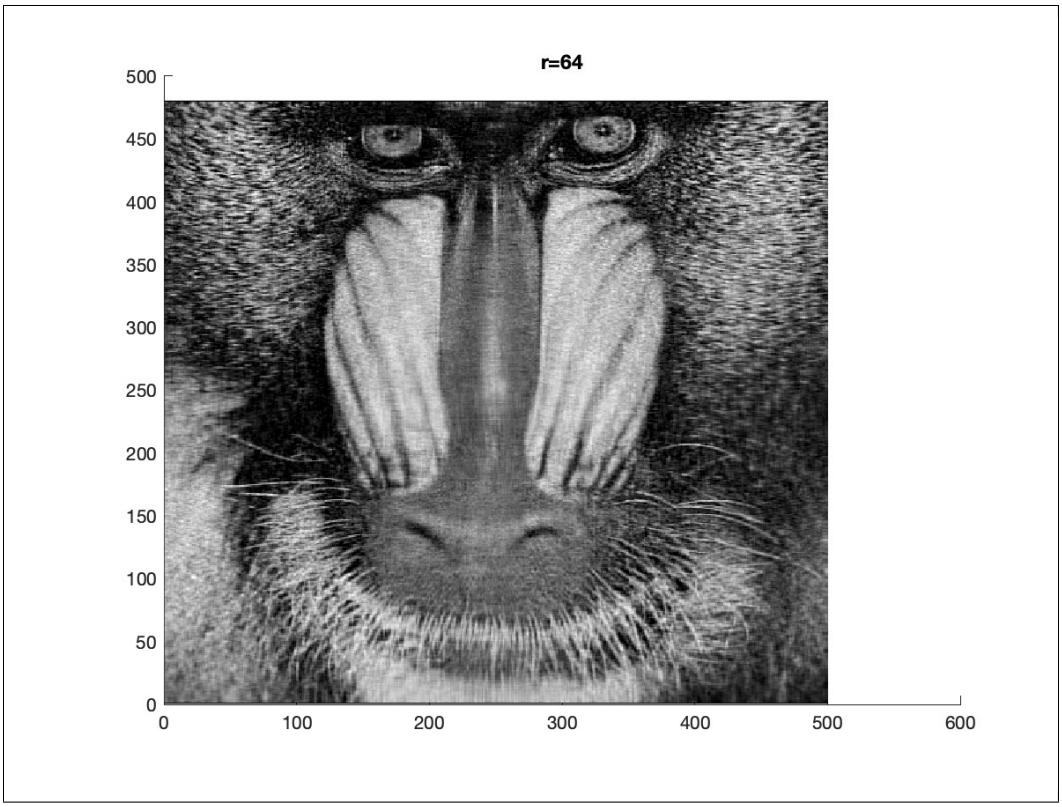
continued on next page...



continued on next page...



continued on next page...


r=64
500
450
400
350
300
250
200
150
100
50
0
0
100
200
300
400
500
600

Solution (b).

While we can start to discern the original image when $r = 16$, it is still possible to notice various obvious abnormalities in the image produced (including various lines and inconsistencies around the bottom of the image). Therefore I choose $r = 32$ to be (in my subjective opinion) to be "acceptable" since it clearly shows us a very close depiction of the original image.

Solution (c).

Since the amount of significant singular values of a matrix effectively determines what its "rank" is, then we only require space to store the 32 column vectors of U and V (32 vectors each) that are associated with the 32 singular values (as dictated by the value of r). Thus it costs $32 \cdot (480 + 500 + 1) = 31392$ "units" of storage versus the original image which costs $480 \cdot 500 = 240000$ "units" of storage. When $r = 32$ we use around $\frac{240000}{31392} \approx 8$ times less storage.