Baylor University

Department

Electrical and Computer Engineering

ELC 5396

Advanced Scientific Computation Laboratory

Class Report 3

Paniz Karbasi

12/12/2016

Contents

Lab 1

Norms

Evaluate the inequalities in exercise 3.3 of the text by plotting them for 100 random x and A. Choose your x and A to be about $x \in \Re^4$ and $A \in \Re^{5\times 3}$. Rewrite the inequalities so they state some function as greater than or equal zero. For instance $\|x\|_{infty} \leq \|x\|_2$ would be rewritten as $0 \leq \|x\|_2 - \|x\|_\infty$. For each of the 1000 uniformly distributed random x and A evaluate the expressions and store the result in a 1000×4 matrix (each column is for one of the inequalities, and each row for a random x or A). Plot the matrix of these calculated results using 'plot2d'. If a line is always non-negative then it could be true. Which ones look legitimate?

Do exercise 3.3. Were your predictions correct?

As an example of a proof consider the following proof for inequality 3.3a.

$$\begin{aligned}
\|x\|_\infty &= \max_{i\in\{1,2,\ldots,m\}} |x_i| && (1.1)\\
&= \max_{i\in\{1,2,\ldots,m\}} \sqrt{x_i^2} && (1.2)\\
&\leq \sqrt{\sum_{i=1}^{m} x_i^2} && (1.3)\\
&\leq \|x\|_2 && (1.4)
\end{aligned}$$

1.1 Solutions to Lab 1

1.

$$\begin{aligned}\|x\|_\infty &= \max_{i\in\{1,2,\dots,m\}} |x_i| && (1.5)\\ &= \max_{i\in\{1,2,\dots,m\}} \sqrt{x_i^2} && (1.6)\\ &\leq \sqrt{\sum_{i=1}^{m} x_i^2} && (1.7)\\ &\leq \|x\|_2 && (1.8)\end{aligned}$$

2.

$$\begin{aligned}\sqrt{m}\|x\|_\infty &= \sqrt{m}\max_{i\in\{1,2,\dots,m\}} |x_i| && (1.9)\\ &= \sqrt{m}\alpha && (1.10)\\ &= \sqrt{m\alpha^2} && (1.11)\\ &= \sqrt{\sum_{i=1}^{m} \alpha^2} && (1.12)\\ &\geq \sqrt{\sum_{i=1}^{m} x_i^2} && (1.13)\\ &= \|x\|_2 && (1.14)\end{aligned}$$

3. To prove $\|A\|_\infty \leq \sqrt{n}\|A\|_2$ we use $\|x\|_\infty \leq \|x\|_2$ which we have already proved in section 1 of this exercise, therefore for $A \in R^{m\times n}$ with $x \in R^n$

$$\|A\|_\infty = \max_{x\neq 0} \frac{\|Ax\|_\infty}{\|x\|_\infty} \qquad (1.15)$$

But we have proved that for $x \in R^n$, $\frac{\sqrt{n}}{\|x\|_2} \geq \frac{1}{\|x\|_\infty}$, thus

$$\frac{\|Ax\|_\infty}{\|x\|_\infty} \leq \sqrt{n}\frac{\|Ax\|_\infty}{\|x\|_2} \leq \sqrt{n}\|A\|_2 \qquad (1.16)$$

$$\|A\|_\infty \leq \sqrt{n}\|A\|_2 \qquad (1.17)$$

4. We have shown that $\|x\|_2 \leq \sqrt{m}\|x\|_\infty$, therefore for $A \in R^{m\times n}$ with $x \in R^n$

$$\|Ax\|_2 \leq \sqrt{m}\|Ax\|_\infty \quad \rightarrow \tfrac{\|Ax\|_2}{\|x\|_2} \leq \tfrac{\sqrt{m}\|Ax\|_\infty}{\|x\|_2}, x \neq 0 \tag{1.18}$$

We have also proved that $\frac{1}{\|x\|_\infty} \geq \frac{1}{\|x\|_2}$, thus

$$\frac{\|Ax\|_2}{\|x\|_2} \leq \frac{\sqrt{m}\|Ax\|_\infty}{\|x\|_2} \leq \frac{\sqrt{m}\|Ax\|_\infty}{\|x\|_\infty} \leq \sqrt{m}\|A\|_\infty \tag{1.19}$$

Equation 1.19 must hold for all $x \neq 0$, therefore

$$\max_{x\neq 0} \frac{\|Ax\|_2}{\|x\|_2} = \|A\|_2 \tag{1.20}$$
$$\leq \sqrt{m}\|A\|_\infty \tag{1.21}$$

1.1.1 Visualization of Norms with Matlab

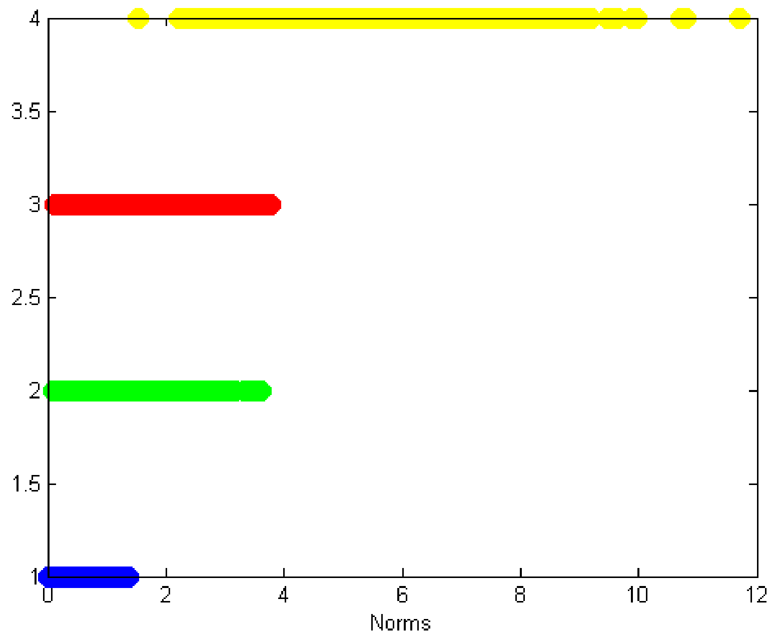


Figure 1.1: Visualization of the four norm inequalities stated in the Lab 1. Blue line represents the first inequality, green line corresponds to the second inequality, red line is for the third inequality and finally yellow line corresponds to the last inequality. As we have expected from the proofs, all of the lines are greater than zero.

Matlab Code

```
x = zeros(4,1);
A = zeros(5,3);
R = zeros(1000,4);

for i=1:1000
    x = randn(size(x));
    R(i,1) = norm(x,2) - norm(x,Inf);
end

for i=1:1000
    x = randn(size(x));
    R(i,2) = sqrt(size(x,1)) * norm(x,Inf) - norm(x,2);
end

for i=1:1000
    A = randn(size(A));
    R(i,3) = sqrt(size(A,2)) * norm(A,2) - norm(A,Inf);
end

for i=1:1000
    A = randn(size(A));
    R(i,4) = sqrt(size(A,1)) * norm(A,Inf) - norm(A,2);
end

plot(R(:,1),1,'-bo','LineWidth',5);
hold on
plot(R(:,2),2,'-go','LineWidth',5);
plot(R(:,3),3,'-ro','LineWidth',5);
plot(R(:,4),4,'-yo','LineWidth',5);
```

Lab 2

SVD and 4 Fundamental Spaces

2.1 Develop SVD

An procedure for finding the SVD by hand (don't do this on a computer):

1. Form $A^H A$
2. Calculate eigenvalues of $A^H A$, denote as λ_i
3. Singular values, $\sigma_i = \sqrt{\lambda_i}$
4. Calculate right singular values, V, by finding $\mathcal{N}(A^H A - \lambda_i I)$ for each of the eigenvalues.
5. Form AA^H, which has the same non-zero eigenvalues.
6. Calculate left singular values, U, by finding $\mathcal{N}(AA^H - \lambda_i I)$ for each of the eigenvalues.

Calculate the SVD by hand for the 3×2 matrices below.

1. $\begin{bmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$
2. $\begin{bmatrix} 1 & 1 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}$
3. $\begin{bmatrix} 1 & 1 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$
4. $\begin{bmatrix} 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$

Show the spaces (dimensions and a basis set) and mappings (from, to, and scale for all four spaces) for the matrices.

Calculate the SVD in MatLab using `[u,s,v]=svd(A);` and compare to what you obtained. Verify that per Theorem 4.1 that the SVD is unique to within unit magnitude scalars.

2.2 Visualize SVD

Do the following:

1. Generate a random 2×2 matrix and takes its SVD.
2. Plot the basis of the domain (V) in subplot 1 and the scaled range basis ($U\Sigma$) in subplot 2, and hold both.
3. Generate N evenly spaced 2-vectors around the unit circle, denoted them by x_i for $i = 1 : N$ (yes we will be in MatLab).
4. Plot all x_i in subplot 1, and all Ax_i in subplot 2.

Try it a few times. Describe what you learned.

2.3 Solutions to Lab 2

2.3.1 Developing SVD

1. $A^TA = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$

   Finding the eigenvalues of A^TA by
   $\begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \lambda \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$
   which represents the system of equations

$$2x_1 + x_2 = \lambda x_1 \tag{2.1}$$
$$x_1 + 2x_2 = \lambda x_2 \tag{2.2}$$

   which rewrite as

$$(2-\lambda)x_1 + x_2 = 0 \tag{2.3}$$
$$x_1 + (2-\lambda)x_2 = 0 \tag{2.4}$$

   which are solved by setting

$\begin{vmatrix} 2-\lambda & 1 \\ 1 & 2-\lambda \end{vmatrix} = 0$

Which works out as:

$(2-\lambda)^2 - 1 = 0$

$\lambda_1 = 3$, $\lambda_2 = 1$ and $\sigma_1 = \sqrt{3}$, $\sigma_2 = 1$.

Substituting λ back into the original equations to find corresponding eigenvectors yields for λ_1 and λ_2

$$2x_1 + x_2 = 3x_1 \tag{2.5}$$
$$x_1 + 2x_2 = 3x_2 \tag{2.6}$$

$x_1 = x_2$.

$$2x_1 + x_2 = x_1 \tag{2.7}$$
$$x_1 + 2x_2 = x_2 \tag{2.8}$$

$x_1 = -x_2$.

In the matrix below the eigenvector for $\lambda = 3$ is the first column and the eigenvector for $\lambda = 1$ is the second column:

$\begin{bmatrix} -1 & -1 \\ -1 & 1 \end{bmatrix}$

Finally, we have to convert this matrix into an orthogonal matrix which we do by applying the Gram-Schmidt orthonormalization process to the column vectors and we get:

$V = \begin{bmatrix} -\frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$

We do the same procedure for calculating the U matrix.

$AA^T = \begin{bmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$

For $\lambda_1 = 3$

$$-x_1 + x_2 + x_3 = 0 \tag{2.9}$$
$$x_1 - 2x_2 = 0 \tag{2.10}$$
$$x_1 - 2x_3 = 0 \tag{2.11}$$

thus $x_1 = -2$, $x_2 = -1$, $x_3 = -1$. For $\lambda_2 = 1$

$$x_1 + x_2 + x_3 = 0 \tag{2.12}$$
$$x_1 = 0 \tag{2.13}$$
$$x_1 = 0 \tag{2.14}$$

thus, $x_1 = 0$, $x_2 = -1$, $x_3 = 1$, and for $\lambda_3 = 0$

$$2x_1 + x_2 + x_3 = 0 \tag{2.15}$$
$$x_1 + x_2 = 0 \tag{2.16}$$
$$x_1 + x_3 = 0 \tag{2.17}$$

thus, $x_1 = -1$, $x_2 = 1$, $x_3 = 1$.

Therefore we get

$\begin{bmatrix} -2 & 0 & -1 \\ -1 & -1 & 1 \\ -1 & 1 & 1 \end{bmatrix}$

and after normalization we get

$U = \begin{bmatrix} -\frac{2}{\sqrt{6}} & 0 & -\frac{1}{\sqrt{3}} \\ -\frac{1}{\sqrt{6}} & -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{3}} \\ -\frac{1}{\sqrt{6}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{3}} \end{bmatrix}$

Thus, $A = \begin{bmatrix} -\frac{2}{\sqrt{6}} & 0 & -\frac{1}{\sqrt{3}} \\ -\frac{1}{\sqrt{6}} & -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{3}} \\ -\frac{1}{\sqrt{6}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{3}} \end{bmatrix} \begin{bmatrix} \sqrt{3} & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} -\frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$

Spaces and mappings of A:

(a) Domain of A or orthonormal basis for the row space: $v_1 = \begin{bmatrix} -\frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{bmatrix}$, $v_2 = \begin{bmatrix} -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$

(b) Range of A or orthonormal basis for the column space: $u_1 = \begin{bmatrix} -\frac{2}{\sqrt{6}} \\ -\frac{1}{\sqrt{6}} \\ -\frac{1}{\sqrt{6}} \end{bmatrix}$, $u_2 = \begin{bmatrix} 0 \\ -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$

(c) rank of A: 2

(d) Mapping: $\sigma_1 = \sqrt{3}$ scales $v_1 \to u_1$, and $\sigma_2 = 1$ scales $v_2 \to u_2$

(e) $\mathcal{N}(A)$: Zero

(f) $\mathcal{N}(A^T)$ or left Null space: $u_1 = \begin{bmatrix} -\frac{1}{\sqrt{3}} \\ \frac{1}{\sqrt{3}} \\ \frac{1}{\sqrt{3}} \end{bmatrix}$

SVD in Matlab using the `[u,s,v]=svd(A);` verifies the uniqueness of singular values or s matrix:

```
A = [1 1
     1 0
     0 1];

sigma = [sqrt(3) 0
         0 1
         0 0]; % calculated by hand

[u,s,v] = svd(A);

s
norm(sigma - s)

s =

    1.7321         0
         0    1.0000
         0         0


ans =

     0
```

2. $A^TA = \begin{bmatrix}1 & 0 & 0\\1 & 1 & 0\end{bmatrix}\begin{bmatrix}1 & 1\\0 & 1\\0 & 0\end{bmatrix} = \begin{bmatrix}1 & 1\\1 & 2\end{bmatrix}$

$\begin{vmatrix}1-\lambda & 1\\1 & 2-\lambda\end{vmatrix} = 0$

Which works out as:

$(1-\lambda)(2-\lambda)-1=0$

$\lambda^2-3\lambda+1=0$

$\lambda_1 = \frac{3+\sqrt{5}}{2}$, $\lambda_2 = \frac{3-\sqrt{5}}{2}$ and $\sigma_1 = \sqrt{\frac{3+\sqrt{5}}{2}}$, $\sigma_2 = \sqrt{\frac{3-\sqrt{5}}{2}}$.

Calculating eigenvectors for $\lambda_1 = \frac{3+\sqrt{5}}{2}$:

$$(1-\lambda_1)x_1 + x_2 = 0 \tag{2.18}$$
$$x_1 + (2-\lambda_1)x_2 = 0 \tag{2.19}$$
$$x_1^2 + x_2^2 = 1 \tag{2.20}$$

Based on the above equations, we know $\frac{x_1}{x_2} = \frac{1}{\lambda_1 - 1}$, thus we set $x_2 = (\lambda_1 - 1)$ into equation (2.20) and finally we get $x_1 = 0.5280$ and $x_2 = 0.8507$. Calculating eigenvectors for $\lambda_2 = \frac{3-\sqrt{5}}{2}$ is like the above case. Thus, for the second eigenvalue, we get $x_1 = -0.8507$ and $x_2 = 0.5280$.

$V = \begin{bmatrix} 0.5257 & -0.8507 \\ 0.8507 & 0.5257 \end{bmatrix}$

$AA^T = \begin{bmatrix} 1 & 1 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 2 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$

Calculating eigenvectors:

$$(2-\lambda)x_1 + x_2 = 0 \tag{2.21}$$
$$x_1 + (1-\lambda)x_2 = 0 \tag{2.22}$$
$$x_1^2 + x_2^2 + x_3^2 = 1 \tag{2.23}$$

We set $\lambda = \frac{3+\sqrt{5}}{2}$, $\lambda = \frac{3-\sqrt{5}}{2}$ and $\lambda = 0$ to the above equations to get the three corresponding eigenvector that form the U matrix.

$U = \begin{bmatrix} 0.8507 & -0.5257 & 0 \\ 0.5257 & 0.8507 & 0 \\ 0 & 0 & 1 \end{bmatrix}$

Therefore, $A = \begin{bmatrix} 0.8507 & -0.5257 & 0 \\ 0.5257 & 0.8507 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1.6180 & 0 \\ 0 & 0.6180 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 0.5257 & 0.8507 \\ -0.8507 & 0.5257 \end{bmatrix}$

Spaces and mappings of A:

(a) Domain of A or orthonormal basis for the row space: $v_1 = \begin{bmatrix} 0.5257 \\ 0.8507 \end{bmatrix}$, $v_2 = \begin{bmatrix} -0.8507 \\ 0.5257 \end{bmatrix}$

(b) Range of A or orthonormal basis for the column space: $u_1 = \begin{bmatrix} 0.8507 \\ 0.5257 \\ 0 \end{bmatrix}$, $u_2 = \begin{bmatrix} -0.5257 \\ 0.8507 \\ 0 \end{bmatrix}$

(c) rank of A: 2

(d) Mapping: $\sigma_1 = 1.6180$ scales $v_1 \to u_1$, and $\sigma_2 = 0.6180$ scales $v_2 \to u_2$

(e) $\mathcal{N}(A)$: Zero

(f) $\mathcal{N}(A^T)$ or left Null space: $u_1 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$

SVD in Matlab using the `[u,s,v]=svd(A);` verifies the uniqueness of singular values or s matrix.

```
A = [1  1
     0  1
     0  0];

sigma = [sqrt((3+sqrt(5))/2)  0
         0  sqrt((3-sqrt(5))/2)
         0  0]; % calculated by hand

[u,s,v] = svd(A);

s
norm(sigma - s)

s =

    1.6180         0
         0    0.6180
         0         0


ans =

     0
```

3. $A^TA = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$

   $\begin{vmatrix} 1-\lambda & 1 \\ 1 & 1-\lambda \end{vmatrix} = 0$

   Which works out as:

   $(1-\lambda)(1-\lambda) - 1 = 0$

   $\lambda^2 - 2\lambda = 0$, Therefore $\lambda = 2$, and $\sigma = \sqrt{2}$.

   $\mathcal{N}(A^TA - \lambda_i I) = \begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix}$

   Thus, $V = \begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$

   $AA^T = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$

   $\mathcal{N}(AA^T - \lambda_i I) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$

Thus, $A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \sqrt{2} & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$

Spaces and mappings of A:

(a) Domain of A or orthonormal basis for the row space: $v_1 = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$

(b) Range of A or orthonormal basis for the column space: $u_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$

(c) rank of A: 1

(d) Mapping: $\sigma_1 = \sqrt{2}$ scales $v_1 \to u_1$

(e) $\mathcal{N}(A)$: $v_2 = \begin{bmatrix} -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$

(f) $\mathcal{N}(A^T)$ or left Null space: $u_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$, $u_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$

SVD in Matlab using the `[u,s,v]=svd(A);` verifies the uniqueness of singular values or s matrix.

```
A = [1  1
     0  0
     0  0];

sigma = [sqrt(2)  0
         0  0
         0  0]; % calculated by hand

[u,s,v] = svd(A);

s
norm(sigma - s)

s =

    1.4142         0
         0         0
         0         0


ans =

     0
```

4. $A^TA = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$

   $\begin{vmatrix} -\lambda & 0 \\ 0 & 1-\lambda \end{vmatrix} = 0$

   Which works out as:

   $\lambda^2 - \lambda = 0$, Therefore $\lambda = 1$, and $\sigma = 1$.

   $\mathcal{N}(A^TA - \lambda_i I) = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$

   Thus, $V = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$

   $AA^T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$

   $\mathcal{N}(AA^T - \lambda_i I) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$

   Thus, $A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$

   Spaces and mappings of A:

   (a) Domain of A or orthonormal basis for the row space: $v_1 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$

   (b) Range of A or orthonormal basis for the column space: $u_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$

   (c) rank of A: 1

   (d) Mapping: $\sigma_1 = 1$ scales $v_1 \rightarrow u_1$

   (e) $\mathcal{N}(A)$: $v_2 = \begin{bmatrix} -1 \\ 0 \end{bmatrix}$

   (f) $\mathcal{N}(A^T)$ or left Null space: $u_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$, $u_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$

   SVD in Matlab using the `[u,s,v]=svd(A);` verifies the uniqueness of singular values or s matrix.

```
A = [0 1
     0 0
     0 0];
```

```
    sigma = [1  0
              0 0
              0  0]; % calculated by hand

    [u,s,v] = svd(A);

    s
    norm(sigma - s)

    s =

         1        0
         0        0
         0        0


    ans =

         0
```

2.3.2 Visualize SVD with Matlab

Based on the results in the top plot of Figure 2.1, we see that v_1 and v_2 have unit length because they are limited to the unit length circle. Results in the bottom plot of Figure 2.1, illustrates that v_1 and v_2 vectors have been scaled and rotated by some amount and this scaling and rotation demonstrates the mapping from the $v_i \to u_i$.

Matlab Code

```
A = randn(2,2);
[u,s,v] = svd(A);
u_s = u * s;
angle = atan2(u(1,1), u(2,1));
R = [ cos(angle) sin(angle); -sin(angle) cos(angle) ];

% plot SVD
subplot(2,1,1);
plot([0 v(1,1)], [0 v(2,1)],'-b','LineWidth',2);
hold on;
plot([0 v(1,2)], [0 v(2,2)],'-b','LineWidth',2);
theta=0:0.01:2*pi;
x=cos(theta);
y=sin(theta);
for i = 1:size(x,2)
    plot(x(i),y(i),'-r');
end
```

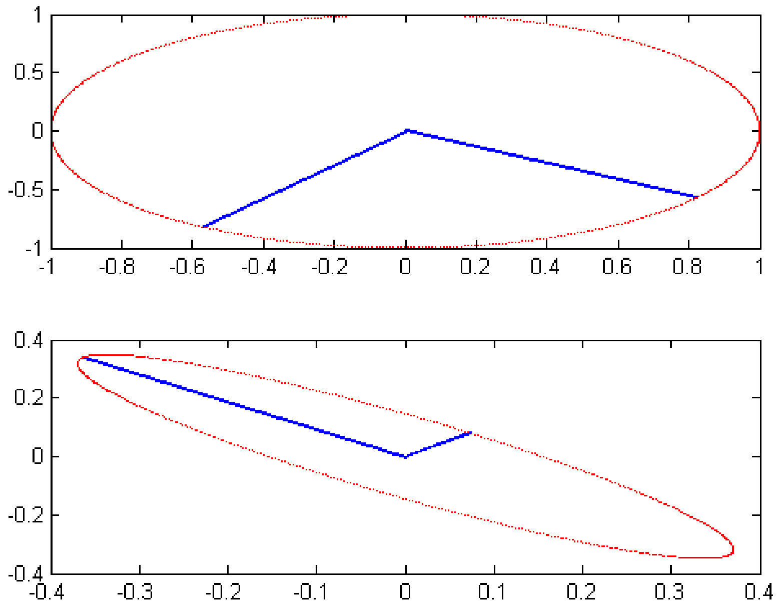


Figure 2.1: Illustration of the SVD of a 2 $\times$ 2 randomly generated matrix

```
subplot(2,1,2);
plot([0  u_s(1,1)], [0  u_s(2,1)],'-b','LineWidth',2);
hold on;
plot([0  u_s(1,2)], [0  u_s(2,2)],'-b','LineWidth',2);
for i = 1:size(x,2)
    data = A*[x(i) y(i)]';
    plot(data(1),data(2),'-r');
end
```

Lab 3

SVD and Moore Penrose Pseudo-inverse

When a matrix is rank deficient or not square then it does not have an inverse. If there is no inverse we want to find the closest equivalent to an inverse, which we term the pseudo-inverse. The Moore-Penrose conditions specify what the pseudo-inverse of a matrix is. We denote the pseudo-inverse of a matrix, A, as $A^\dagger$. The Moore-Penrose conditions are

1. $AA^\dagger A = A$

2. $A^\dagger AA^\dagger = A^\dagger$

3. $AA^\dagger$ is symmetric

4. $A^\dagger A$ is symmetric

All four must be met, as it is possible for a matrix to violate only one condition.

The easiest way to calculate the pseudo-inverse is to use the singular value decomposition (SVD). In particular let the SVD of a $m \times n$ matrix A be given by

$$U\Sigma V^T \tag{3.1}$$

with U and V unitary matrices and Σ a diagonal matrix with elements $\sigma_1 \geq \sigma_2 \geq \ldots \sigma_n \geq 0$. We refer to the σ_i as the singular values. The number of non-zero singular values are called the rank.

Let the rank of A be r[1], then the pseudo-inverse is given by

$$\begin{aligned}
A^\dagger &= V\Sigma^\dagger U^T && (3.2)\\
&= V \begin{matrix} \frac{1}{\sigma_1} & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & \frac{1}{\sigma_1} & \ddots & \vdots & \vdots & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 & 0 & \cdots & 0 \\ 0 & \cdots & 0 & \frac{1}{\sigma_r} & 0 & \cdots & 0 \\ 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \end{matrix} U^T && (3.3)\\
&= V[:,1:r] \begin{matrix} \frac{1}{\sigma_1} & 0 & \cdots & 0 \\ 0 & \frac{1}{\sigma_1} & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & \frac{1}{\sigma_r} \end{matrix} U[:,1:r]^T && (3.4)\\
&= V\text{diag}(\frac{1}{\sigma_1}, \frac{1}{\sigma_2}, \ldots, \frac{1}{\sigma_r})U^T && (3.5)
\end{aligned}$$

Generate 5 random 10x20 matrices and confirm that the SVD gives you the Moore-Penrose pseudo-inverse.

Prove the SVD algorithm we outlined above must work for all matrices.

Is it unique?[2] [3]

3.1 Solutions to Lab 3

3.1.1 Proofs

1. $AA^\dagger A = A$

$$\begin{aligned}
AA^\dagger A &= U\Sigma V^T V\Sigma^{-T}U^T U\Sigma V^T && (3.6)\\
&= U\Sigma\Sigma^{-T}\Sigma V^T && (3.7)\\
&= U\Sigma V^T && (3.8)\\
&= A && (3.9)
\end{aligned}$$

2. $A^\dagger AA^\dagger = A^\dagger$

[1]The equation shows $r < \min(m,n)$ but we could have the matrix be full rank, i.e. $r = \min(m,n)$ and the matrix would just not have the extra rows and/or columns of the $\Sigma^\dagger$ matrix.

[2]The singular values are unique, but the singular vector matrices are not.

[3]The pseudoinverse exists and is unique.

$$\begin{aligned} A^\dagger A A^\dagger &= V\Sigma^{-T}U^TU\Sigma V^TV\Sigma^{-T}U^T && (3.10)\\ &= V\Sigma^{-T}\Sigma\Sigma^{-T}U^T && (3.11)\\ &= V\Sigma^{-T}U^T && (3.12)\\ &= A^\dagger && (3.13)\end{aligned}$$

3. $AA^\dagger$ is symmetric, or $AA^\dagger = (AA^\dagger)^T$

$$AA^\dagger = U\Sigma V^TV\Sigma^{-T}U^T \quad (3.14)$$

$$\begin{aligned} (AA^\dagger)^T &= (U\Sigma V^TV\Sigma^{-T}U^T)^T && (3.15)\\ &= U(\Sigma^{-T})^TV^TV\Sigma^TU^T && (3.16)\\ &= U(\Sigma^{-T})^T\Sigma^TU^T && (3.17)\\ &= U\Sigma\Sigma^{-T}U^T && (3.18)\\ &= U\Sigma V^TV\Sigma^{-T}U^T && (3.19)\end{aligned}$$

Therefore $AA^\dagger$ is symmetric.

4. $A^\dagger A$ is symmetric, or $A^\dagger A = (A^\dagger A)^T$

$$A^\dagger A = V\Sigma^{-T}U^TU\Sigma V^T \quad (3.20)$$

$$\begin{aligned} (A^\dagger A)^T &= (V\Sigma^{-T}U^TU\Sigma V^T)^T && (3.21)\\ &= V\Sigma^TU^TU(\Sigma^{-T})^TV^T && (3.22)\\ &= V\Sigma^T(\Sigma^{-T})^TV^T && (3.23)\\ &= V\Sigma^{-T}\Sigma V^T && (3.24)\\ &= V\Sigma^{-T}U^TU\Sigma V^T && (3.25)\end{aligned}$$

Therefore $A^\dagger A$ is symmetric.

3.1.2 Testing with Matlab

1. $AA^\dagger A = A$

$$norms = \begin{bmatrix} 8.11917080062955e-15\\ 7.35738675657055e-15\\ 1.00290429425367e-14\\ 3.89791339243802e-14\\ 2.75093215041579e-14\end{bmatrix}$$

2. $A^\dagger A A^\dagger = A^\dagger$

$$norms = \begin{bmatrix} 8.37048147749967e-16 \\ 6.70493641093655e-16 \\ 6.16040612895675e-16 \\ 8.93965711013902e-16 \\ 7.73410888369713e-16 \end{bmatrix}$$

3. $A^\dagger A$ is symmetric

$$norms = \begin{bmatrix} 2.68400072912619e-15 \\ 2.13419513451013e-15 \\ 1.90084366992159e-15 \\ 2.01084639481567e-15 \\ 1.44589046790825e-15 \end{bmatrix}$$

4. $AA^\dagger$ is symmetric

$$norms = \begin{bmatrix} 1.08770018059034e-14 \\ 8.10341820176379e-15 \\ 2.24717784981763e-15 \\ 1.55101035935962e-15 \\ 3.30769179521066e-15 \end{bmatrix}$$

Matlab Code

1.
```
A = zeros(10,20);
norms = zeros(5,1);

for i = 1:5
     A = randn(10,20);
     [u,s,v] = svd(A);
     sig_inv = s;
     sig_inv(sig_inv>0) = 1./sig_inv(sig_inv>0);
     A_pseudo = v * sig_inv' * u';
     norms(i) = norm((A * A_pseudo * A) - A,2);
end
```

2.
```
A = zeros(10,20);
norms = zeros(5,1);

for i = 1:5
     A = randn(10,20);
     [u,s,v] = svd(A);
     sig_inv = s;
     sig_inv(sig_inv>0) = 1./sig_inv(sig_inv>0);
     A_pseudo = v * sig_inv' * u';
     norms(i) = norm((A_pseudo * A * A_pseudo) - A_pseudo,2);
end
```

3.
```
A = zeros(10,20);
norms = zeros(5,1);

for i = 1:5
    A = randn(10,20);
    [u,s,v] = svd(A);
    sig_inv = s;
    sig_inv(sig_inv>0) = 1./sig_inv(sig_inv>0);
    A_pseudo = v * sig_inv' * u';
    norms(i) = norm((A_pseudo * A) - (A_pseudo * A)',2);
end
```

4.
```
A = zeros(10,20);
norms = zeros(5,1);

for i = 1:5
    A = randn(10,20);
    [u,s,v] = svd(A);
    sig_inv = s;
    sig_inv(sig_inv>0) = 1./sig_inv(sig_inv>0);
    A_pseudo = v * sig_inv' * u';
    norms(i) = norm((A * A_pseudo) - (A * A_pseudo)',2);
end
```

Lab 4

Projectors

Do 6.1 and 6.2

4.1 Solutions to Lab 4

1. Problem 6.1 from the text

   In order to prove that $(I - 2P)$ is unitary, we have to show that $(I - 2P)(I - 2P)^T = (I - 2P)^T(I - 2P) = I$.

$$\begin{aligned}
(I - 2P)(I - 2P)^T &= (I - 2P)(I^T - 2P^T) && (4.1)\\
&= (I - 2P)(I - 2P) && (4.2)\\
&= I - 2P - 2P + 4P^2 && (4.3)\\
&= I && (4.4)
\end{aligned}$$

   Also,

$$\begin{aligned}
(I - 2P)^T(I - 2P) &= (I^T - 2P^T)(I - 2P) && (4.5)\\
&= (I - 2P)(I - 2P) && (4.6)\\
&= I - 2P - 2P + 4P^2 && (4.7)\\
&= I && (4.8)
\end{aligned}$$

   Thus, $(I - 2P)$ is unitary.

   The geometric interpretation is that if we consider the residual (r) of projecting a vector (b) onto a plane (Pb) as $r = (I - P)b$, thus, $I - 2P$ is like reflecting the vector b.

2. Problem 6.2 form the text

   In order to determine whether E is a projector or not, we have show that $E = E^2$. Based on the definition, $Ex = \frac{x+Fx}{2}$ where F is the **R**everse **O**rder **I**dentity matrix. Thus, we could write E as

$$E = \frac{I+F}{2} \tag{4.9}$$

   Thus, we show that $E = E^2$

$$\begin{aligned}
E^2 &= (\frac{I+F}{2})(\frac{I+F}{2})^T && (4.10)\\
&= \frac{I^2 + IF^T + FI + F^2}{4} && (4.11)\\
&= \frac{I + F^T + F + F^2}{4} && (4.12)\\
&= \frac{I + 2F + I}{4} && (4.13)\\
&= \frac{2I + 2F}{4} && (4.14)\\
&= \frac{I+F}{2} && (4.15)\\
&= E && (4.16)
\end{aligned}$$

   Therefore E is a projector. Also for showing that if E is an orthogonal projector or not, we have to show that $E = E^T$.

$$\begin{aligned}
E^T &= (\frac{I+F}{2})^T && (4.17)\\
&= \frac{I^T + F^T}{2} && (4.18)\\
&= \frac{I+F}{2} && (4.19)
\end{aligned}$$

   Thus, E is an orthogonal projector. As an example, the entries of E for a 3×3 matrix are:

$$\frac{\begin{bmatrix}1&0&0\\0&1&0\\0&0&1\end{bmatrix} + \begin{bmatrix}0&0&1\\0&1&0\\1&0&0\end{bmatrix}}{2} = \frac{\begin{bmatrix}1&0&1\\0&2&0\\1&0&1\end{bmatrix}}{2}$$

   and the above matrix extracts the even part of an 3-vector like the following:

$$\frac{\begin{bmatrix}1&0&1\\0&2&0\\1&0&1\end{bmatrix}\begin{bmatrix}x\\y\\z\end{bmatrix}}{2} = \frac{\begin{bmatrix}x+z\\2y\\x+z\end{bmatrix}}{2}$$

Lab 5

Classical QR

Classical QR uses Gram-Schmidt orthogonalization and is unfortunately unstable. That of course does not mean that it never works, in fact it will work well for a large number of 'nice' problems. Right now we just want to develop an intuition of QR.

Write your own function that implements algorithm 7.1 in MatLab.

Use your QR function in Experiment 1 of chapter 9[1].

Do problem 7.2 the following way. Generate two random matrices A and B. Use your QR to find Q_A and Q_B. Interleave the columns of Q_A and Q_B, as described in 7.2. Now use your QR again and look at the resulting R. What do you notice? Can you prove it?

5.1 Solutions to Lab 5

5.1.1 Matlab Implementation of Classical QR

```
function [Q,R] = qr_classic(A)
% [Q,R] = qr_classic(A)
% Q: m*m orthogonal matrix
% R: n*n upper triangular matirx

[m,n] = size(A);
Q = zeros(m,m);
R = zeros(m,n);

for j = 1:n
    v = A(:,j);
    for i=1:j-1
          R(i,j) = Q(:,i)'*A(:,j);
          v = v - R(i,j)*Q(:,i);
    end
    R(j,j) = norm(v);
```

[1]Chapter 9 is the MatLab tests for the QR chapters.

```
    Q(:,j) = v/R(j,j);
end

end
```

5.1.2 Discrete Legendre Polynomials

```
x = (-128:128)'/128;
A = [x.^0 x.^1 x.^2 x.^3];
[Q,R] = qr_classic(A);
scale = Q(257,:);
Q = Q*diag(1 ./ scale);
plot(Q)
```

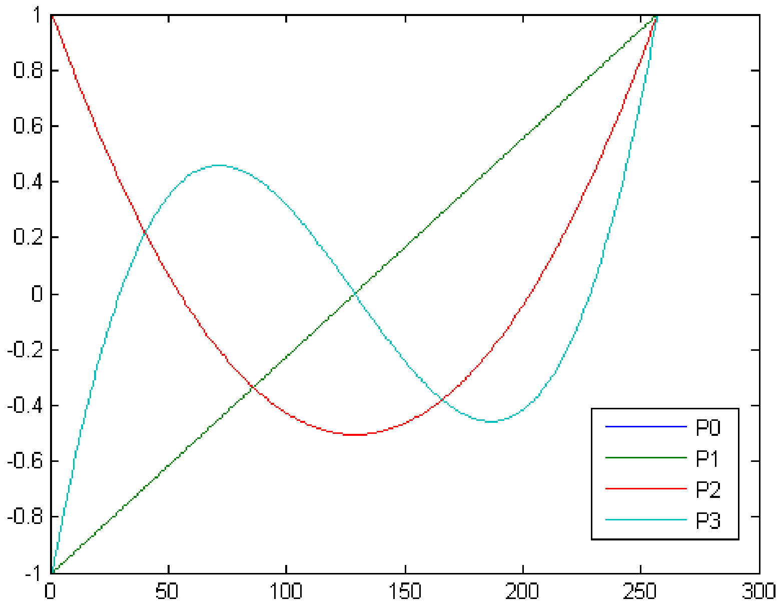


Figure 5.1: Illustration of the Discrete Legendre Polynomials

5.1.3 Problem 7.2 from Text

R is an upper triangular matrix with $r_{ij} = 0$ when $i + j$ is odd. The proof comes from the fact that A has an special structure such that an odd column of A is a linear combination of odd columns of Q and an even column of A is a linear combination of even columns of Q . Therefore, if $i + j$ is odd, it means that $R(i, j) = Q(:, i)' \times A(:, j)$ is zero.

Lab 6

Modified QR

Write a MatLab function to implement Algorithm 8.1.

Do Experiment 2 from Lecture 9 in the text. Use your own functions to do classical and modified Gram-Schmidt.

6.1 Solutions To Lab 6

6.1.1 Matlab Implementation of Modified QR Algorithm

```
function [Q,R] = qr_modified(A)
% [Q,R] = qr_modified(A)
% Q: m*m orthogonal matrix
% R: m*n upper triangular matirx

[m,n] = size(A);
Q = zeros(m,m);
R = zeros(m,n);
v = A;

for i = 1:n
    R(i,i) = norm(v(:,i));
    Q(:,i) = v(:,i)/R(i,i);
    for j = (i+1):n
        R(i,j) = Q(:,i)'*v(:,j);
        v(:,j) = v(:,j) - R(i,j)*Q(:,i);
    end
end

end
```

6.1.2 Classical vs. Modified Gram-Schmidt

```
[U,X] = qr(randn(80));
[V,X] = qr(randn(80));
S = diag(2.^(-1:-1:-80));

A = U*S*V;

[QC,RC] = qr_classic(A);
[QM,RM] = qr_modified(A);

d1 = diag(RC); semilogy(d1,'*'); hold
d2 = diag(RM); semilogy(d2,'or');
```

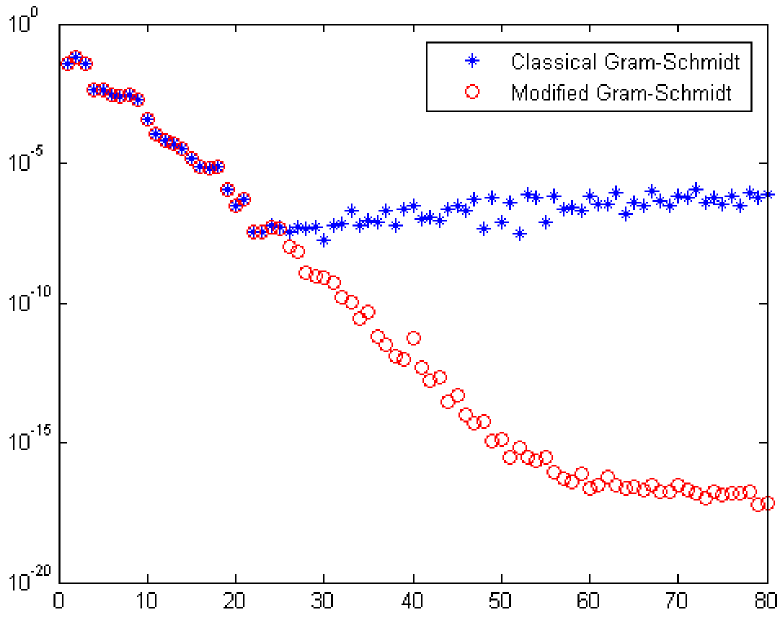


Figure 6.1: Classical vs. Modified Gram-Schmidt

Lab 7

QR Solving

7.1 Using QR to Solve

Now calculate $y = A1 \quad 2 \quad 3 \quad 4^T$. Use the qr decomposition of A and y to solve $Az = y$ for z. As we discussed in lab you have

$$\begin{aligned} QRz &= y && (7.1) \\ Rz &= Q^T y && (7.2) \end{aligned}$$

Since R is upper triangular you can use backward substitution to solve for z. Write the elements of the matrices

$$\begin{array}{cccc} r_{11} & r_{12} & r_{13} & r_{14} z_1 \\ 0 & r_{22} & r_{23} & r_{24} z_2 \\ 0 & 0 & r_{33} & r_{34} z_3 \\ 0 & 0 & 0 & r_{44} z_4 \end{array} = \begin{array}{c} [Q^T y]_1 \\ [Q^T y]_2 \\ [Q^T y]_3 \\ [Q^T y]_4 \end{array} \qquad (7.3)$$

where $[Q^T y]_i$ is the i^{th} element of the vector $Q^T y$. Since R has four columns, start with row four and solve for the one unknown, z_4, as this will only require one division. Continue by working up the rows of R till all of z is known. Can errors build by doing this? Write a MatLab function "BackSubstitute" (see Algorithm 17.1) that takes an R, x, and b and calculates $x = R^{-1} b$ by back substitution.

Verify that your value is $1 \quad 2 \quad 3 \quad 4^T$. Usually we do not know the "true" value, so we cannot verify we found it with our method. In this case, since we started with the true value we can verify. We do not use the true value in the solution (only A and y are used) since we would not know it in real problems.

7.2 Practical Use of QR

In reality you don't need Q. Make a new version of your modified QR algorithm that just returns R. To this new QR pass the augmented matrix $[Ay]$, and you will get $[RQ^T y]$. Make a random $A \in \mathbb{R}^{n \times n}$ and $b \in \mathbb{R}^n$ for $n = \{10, 50, 100, 500, 1000\}$, then time how long it takes to solve both ways (not you don't need a separate $A^T y$ step. Timing can be done by the commands "tic" and "toc" in MatLab. How do the methods compare?

7.3 Solutions to Lab 7

7.3.1 Using QR to Solve

1. Matlab Implementation of the Backsubstitute Algorithm

```
function x = backSubstitute(R,b)
% Solving an upper triangular system by back-substitution
% R: n*n upper triangular matrix
% b: n*1 vector
% x: The solution to the linear system R x = b

n = size(b,1);
x=zeros(n,1);
for j=n:-1:1
     if (R(j,j)==0) error('Error!'); end;
     x(j)=b(j)/R(j,j);
     b(1:j-1)=b(1:j-1)-R(1:j-1,j)*x(j);
end
```

2. Verifying the back substitution algorithm

```
A = randn(4,4);
x_true = [1 2 3 4]';
b = A*x_true;
[Q,R] = qr_modified(A);
x = backSubstitute(R,Q'*b)
norm(x-x_true)

% Results
x =

     1.0000
     2.0000
     3.0000
     4.0000


ans =

   9.5383e-013
```

7.3.2 Practical Use of QR

1. Matlab Implementation of the practical use of QR

```
function R = qr_modified_R(A)
% R = qr_modified_R(A)
% A: m*n (augmented) matrix
% R: n*n upper triangular matirx

[m,n] = size(A);
Q = zeros(m,n);
R = zeros(n-1,n);
v = A(:,1:n);

for i = 1:n-1
    R(i,i) = norm(v(:,i));
    Q(:,i) = v(:,i)/R(i,i);
    for j = (i+1):n
        R(i,j) = Q(:,i)'*v(:,j);
        v(:,j) = v(:,j) - R(i,j)*Q(:,i);
    end
end

end
```

2. Timing and comparison of QR and practical QR for solving the linear system of equations $Ax = b$

```
clc;
s = [10 50 100 500 1000];

for i=1:size(s,2)
    n = s(i);
    A = randn(n,n);
    b = randn(s(i),1);

    disp(['****** Timing for n = ' num2str(s(i)) ' ******'])

    disp('---Practical QR---')
    tic
    A_b = [A b];
    [R_Q_b] = qr_modified_R(A_b);
    x_1 = backSubstitute(R_Q_b(:,1:n),R_Q_b(:,n+1));
    toc

    disp('---------QR---------')
    tic
    [Q,R] = qr_modified(A);
    Q_b = Q'*b;
    x_2 = backSubstitute(R,Q_b);
    toc
    disp('*******************************')
    disp(' ')
    disp(' ')

end
```

```
****** Timing for n = 10 ******
---Practical QR---
Elapsed time is 0.001507 seconds.
---------QR---------
Elapsed time is 0.001771 seconds.
*******************************


****** Timing for n = 50 ******
---Practical QR---
Elapsed time is 0.005196 seconds.
---------QR---------
Elapsed time is 0.006419 seconds.
*******************************
```

```
****** Timing for n = 100 ******
---Practical QR---
Elapsed time is 0.018133 seconds.
----------QR----------
Elapsed time is 0.013165 seconds.
*******************************


****** Timing for n = 500 ******
---Practical QR---
Elapsed time is 0.416801 seconds.
----------QR----------
Elapsed time is 0.423356 seconds.
*******************************


****** Timing for n = 1000 ******
---Practical QR---
Elapsed time is 2.318204 seconds.
----------QR----------
Elapsed time is 2.442693 seconds.
*******************************
```

Based on the timings, when we use the augmented form of $[Ab]$ to compute x where $Ax = b$, it is slightly faster.

Lab 8

Householder QR

Write a MatLab function to implement Algorithm 10.1.

Do Experiment 3 from Lecture 9 in the text. Use your own functions.

8.1 Solutions to Lab 8

8.1.1 Matlab Implementation of Householder QR

```
function [Q,R] = qr_hh(A)
% [Q,R] = qr_hh(A)
% Q: m*m orthogonal matrix
% R: m*n upper triangular matirx

[m, n] = size(A);
I = eye(m);
Qt = eye(m);
R = A;
for i = 1:n
   e = I(:,i);
   r = R(:,i);
   r(1:i-1) = zeros(i-1, 1);
   v = r + sign(r(i)) * norm(r) * e;
   H = I - 2 * (v*v') / (v'*v);
   R = H * R;
   Qt = H * Qt;
end
Q = Qt';
end
```

8.1.2 Numerical Loss of Orthogonality

```
A = [0.70000  0.70711;
     0.70001  0.70711];

[QH,RH] = qr_hh(A); %compute factor Q by Householder
norm(QH'*QH-eye(2))
     ans =  3.1117e-016

[QM,RM] = qr_modified(A); %compute facror Q by modified GS
norm(QM'*QM-eye(2))
     ans = 2.3014e-011
```

Lab 9

Least Squares

9.1 Surveying

In surveying, a common problem is to measure the height of a point. To avoid errors it is common to measure not only the height but the relative heights. For a particular system with four points to be measured (x_1, x_2, x_3, x_4) the following measurements were made:

$$\begin{aligned}
x_1 &= 2.95 \\
x_2 &= 1.74 \\
x_3 &= -1.45 \\
x_4 &= 1.32 \\
x_1 - x_2 &= 1.23 \\
x_1 - x_3 &= 4.45 \\
x_1 - x_4 &= 1.61 \\
x_2 - x_3 &= 3.21 \\
x_2 - x_4 &= 0.45 \\
x_3 - x_4 &= -2.75
\end{aligned}$$

Form the problem into matrix notation ($Ax \cong b$) and compute the values of the x_i using your three QR methods, MatLab's QR, and SVD. Compare to the measured values (first four elements above).

9.2 Mass Spectrometry

Mass spectrometry[1] is a widely used technique to analyze the structure, composition, and quantity of an unknown substance. Mass spectrometers convert a substance to gas and ionizes it. The ions are then sorted by charge to mass ratio and detected. The result is a series of peaks with heights h_j. Each molecule, i, makes a contribution to peak height, h_j, based on its concentration, p_i, and

[1]For more information see, "http://www.asms.org/whatisms/".

a coefficient of proportionality, c_{ij}, that is based on the molecules structure. "Applied Numerical Analysis" by Gerald and Wheatly states that the following values of c_{ij} were reported by Carnahan in 1964.

Peak (j)	$CH_4.(i = 1)$	$C_2H_4.(i = 2)$	$C_2H_6.(i = 3)$	$C_3H_6.(i = 4)$	$C_3H_8.(i = 5)$
1	0.165	0.202	0.317	0.234	0.182
2	27.7	0.862	0.062	0.073	0.131
3	.	22.35	13.05	4.420	6.001
4	.	.	11.28	0.000	1.110
5	.	.	.	9.850	1.684
6	.	.	.	.	15.94

The sample measured had peak heights of: $h_1 = 5.20$, $h_2 = 61.7$, $h_3 = 149.2$, $h_4 = 79.4$, $h_5 = 89.3$, and $h_6 = 69.3$.

9.3 Solutions to Lab 9

9.3.1 Surveying

```
clear
clc
x_true = [2.95 1.74 -1.45 1.32];
b = zeros(size(10,1));
b = [2.95,
     1.74,
     -1.45,
     1.32,
     1.23,
     4.45,
     1.61,
     3.21,
     0.45,
     -2.75];
A = eye(10,4);
A(5,1) = 1; A(5,2) = -1;
A(6,1) = 1; A(6,3) = -1;
A(7,1) = 1; A(7,4) = -1;
A(8,2) = 1; A(8,3) = -1;
A(9,2) = 1; A(9,4) = -1;
A(10,3)= 1; A(10,4) = -1;
[m ,n] = size(A);

[QC,RC] = qr_classic(A);
xc = RC\(QC'*b)
norm_qr_classic = norm(x_true'-xc)

[QM,RM] = qr_modified(A);
xm = RM\(QM'*b)
```

```
norm_qr_modified = norm(x_true'-xc)

[QH,RH] = qr_hh(A);
xh = RH\(QH'*b)
norm_qr_hh = norm(x_true'-xh)

[QMT,RMT] = qr(A);
xmt = RMT\(QMT'*b)
norm_qr_matlab = norm(x_true'-xmt)

[u,s,v] = svd(A);
for i=1:n
    s(i,i) = 1/s(i,i);
end
 xsvd = v*(s'*(u'*b))
 norm_svd_matlab = norm(x_true'-xsvd)



 xc =

    2.9600
    1.7460
   -1.4600
    1.3140


norm_qr_classic =

    0.0165


xm =

    2.9600
    1.7460
   -1.4600
    1.3140


norm_qr_modified =

    0.0165


xh =
     2.9600
     1.7460
    -1.4600
     1.3140


norm_qr_hh =

     0.0165


xmt =

     2.9600
     1.7460
    -1.4600
     1.3140


norm_qr_matlab =

     0.0165


xsvd =

     2.9600
     1.7460
    -1.4600
     1.3140


norm_svd_matlab =

     0.0165
```

9.3.2 Mass Spectrometry

```
clear
clc
h = [5.20,
      61.7,
      149.2,
      79.4,
      89.3,
      69.3];
```

```
A = zeros(6,5);
A(1,1) = 0.165;
A(1,2) = 0.202;
A(1,3) = 0.317;
A(1,4) = 0.234;
A(1,5) = 0.182;
A(2,1) = 27.7;
A(2,2) = 0.862;
A(2,3) = 0.062;
A(2,4) = 0.073;
A(2,5) = 0.131;
A(3,2) = 22.35;
A(3,3) = 13.05;
A(3,4) = 4.420;
A(3,5) = 6.001;
A(4,3) = 11.28;
A(4,5) = 1.110;
A(5,4) = 9.850;
A(5,5) = 1.684;
A(6,5) = 15.94;
[m ,n] = size(A);

[QC,RC] = qr_classic(A);
xc = RC\(QC'*h)

[QM,RM] = qr_modified(A);
xm = RM\(QM'*h)

[QH,RH] = qr_hh(A);
xh = RH\(QH'*h)


[QMT,RMT] = qr(A);
xmt = RMT\(QMT'*h)

[u,s,v] = svd(A);
for i=1:n
    s(i,i) = 1/s(i,i);
end
 xsvd = v*(s'*(u'*h))


xc =

    2.1701
    0.0021
    6.6112
    8.3227
    4.3476


xm =

    2.1701
    0.0021
    6.6112
    8.3227
    4.3476


xh =

    2.1701
    0.0021
    6.6112
    8.3227
    4.3476


xmt =

    2.1701
    0.0021
    6.6112
    8.3227
    4.3476


xsvd =

    2.1701
    0.0021
    6.6112
    8.3227
    4.3476
```

Lab 10

Conditioning

10.1 A Perturbing Problem ...

Consider the following problem:

$$\begin{bmatrix} 1.50 & 1.50 \\ 1.01 & 0.99 \end{bmatrix} x = \begin{bmatrix} 3.00 \\ 2.00 \end{bmatrix} \tag{10.1}$$

What is the result?

$$\begin{bmatrix} 1.00 \\ 1.00 \end{bmatrix} \tag{10.2}$$

Now let's slightly perturb (less than 1% to the lower left entry of the matrix) the system to:

$$\begin{bmatrix} 1.50 & 1.50 \\ 1.00 & 0.99 \end{bmatrix} x = \begin{bmatrix} 3.00 \\ 2.00 \end{bmatrix} \tag{10.3}$$

What is the result?

$$\begin{bmatrix} 2.00 \\ 0.00 \end{bmatrix} \tag{10.4}$$

Why the big swing? What is the condition number of the matrix? The product of the condition number and the perturbation value gives an upper bound on the perturbation of the system solution. What is the upper bound on the perturbation of the solution? How big is the actual solution perturbation?

- If we consider the geometric interpretation of the least squares solution, as we perturb A, line Ax_{ls} changes and thus line $Ax_{ls} - b$ which we are trying to minimize, changes.
- The condition number of the original matrix is 432.0044 and after perturbation is 216.6687.

- The upper bound on the perturbation of the solution is 0.01×216.6687 which is 2.1667, and the actual solution perturbation is 1.00.

10.2 Needs a Robust Solution

Rather than solving using standard pseudo-inverse/least squares,

$$\begin{aligned} x_{ls} &= A^{\dagger}b && (10.5)\\ &= (A^TA)^{-1}A^Tb && (10.6)\end{aligned}$$

Let's consider the basic for of the robust least squares solution,

$$x_{rls} = (A^TA+\psi I)^{-1}A^Tb \qquad (10.7)$$

Using the SVD we learned easier this becomes,

$$\begin{aligned} x_{rls} &= (A^TA+\psi I)^{-1}A^Tb && (10.8)\\ &= ((U\Sigma V^T)^TU\Sigma V^T+\psi I)^{-1}(U\Sigma V^T)^Tb && (10.9)\\ &= (V\Sigma U^TU\Sigma V^T+\psi I)^{-1}V\Sigma U^Tb && (10.10)\\ &= (V\Sigma\Sigma V^T+\psi VV^T)^{-1}V\Sigma U^Tb && (10.11)\\ &= V(\Sigma^2+\psi I)^{-1}V^TV\Sigma U^Tb && (10.12)\\ &= V(\Sigma^2+\psi I)^{-1}\Sigma U^Tb && (10.13)\end{aligned}$$

Note the middle term $(\Sigma^2+\psi I)^{-1}\Sigma$ is a diagonal matrix with entries $\frac{\sigma_i}{\sigma_i^2+\psi}$. This can be easily implemented in Matlab, so create a function to do it, which takes A, b, and ψ and returns x_{rls}. Let ψ be the same size as the perturbation 0.01 and show the solution.

```
function x = robust_least_square(A,b,psi)
    [m,n] = size(A);
    [u,s,v] = svd(A);
    u_b = u'*b;
    p = zeros(n,1);
    for i=1:n
        p(i) = (s(i,i)/(s(i,i)^2 + psi)) * u_b(i);
    end
    x = v*p;
end

>> x = robust_least_square(A,b,0.01)

x =

    1.0050
    0.9950
```

```
>> round(x)
    1
    1
```

Is this just a lucky guess? Plot $x_{rls}(1)$ and $x_{ls}(1)$ in one subplot and plot $x_{rls}(2)$ and $x_{ls}(2)$ in another subplot. Note the pattern. What happens to x_{rls} with the following?

1. $\lim_{\psi\to\pm\infty} x_{rls}$ As $\psi \to \pm\infty$ the diagonal entries of the diagonal matrix $(\Sigma^2 + \psi I)^{-1}\Sigma$ go to zero and thus the x_{rls} vector becomes zero.
2. $\lim_{\psi\to-\sigma_i^2} x_{rls}$ As $\psi \to \sigma_i^2$ the diagonal entries of the diagonal matrix $(\Sigma^2 + \psi I)^{-1}\Sigma$ become $\frac{1}{2\sigma_i}$, thus x_{rls} becomes half of the x_{ls}.

Will any value of ψ work?

```
psi=-1:0.01:1;
x_rls = zeros(2,size(psi,2));
x_ls = zeros(2,size(psi,2));
A = [1.5  1.5;  1.0  0.99];
b = [3;2];
x_true = A\b;
for  i=1:size(psi,2)
     x_rls(:,i) = robust_least_square(A,b,psi(i));
     x_ls(:,i) = x_true;
end

subplot(2,1,1)
plot(psi,round(x_rls(1,:)),  'b','LineWidth',2.2)
hold on
plot(psi,x_ls(1,:),  'r','LineWidth',5)
subplot(2,1,2)
plot(psi,round(x_rls(2,:)),'b','LineWidth',2.25)
hold on
plot(psi,x_ls(2,:),'r','LineWidth',3)
```

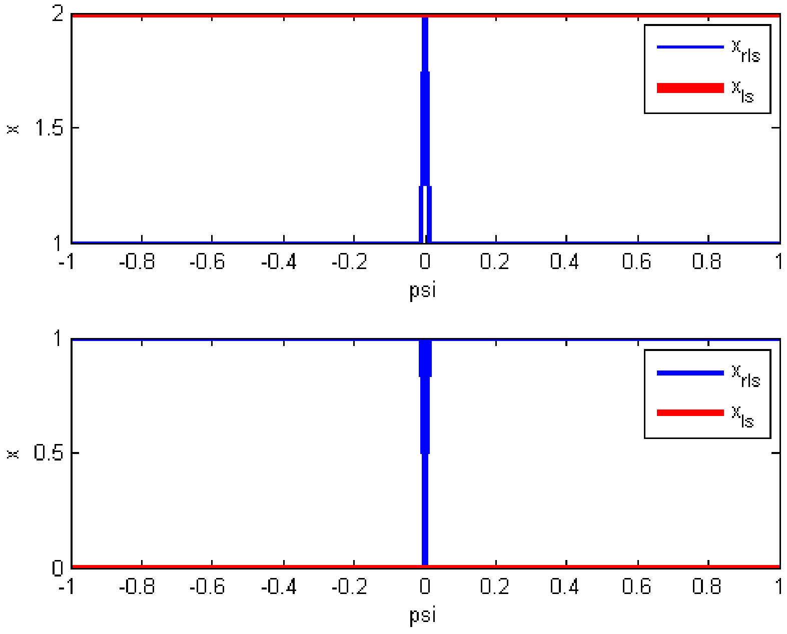


Figure 10.1: Illustration of robust least square vs. least square solution

Lab 11

Stability of Householder and Back Substitution

Do the experiment from lecture 16.

11.1 Solution to Lab 11

11.1.1 Matlab Code

```
R = triu(randn(50));
[Q,X] = qr(randn(50));
A = Q*R;
[Q2,R2] = qr(A);


norm(Q2-Q)
     0.0021

norm(R2-R)/norm(R)
     4.8824e-04

norm(A-Q2*R2)/norm(A)
     1.0627e-15

Q3 = Q+1e-4*randn(50);
R3 = R+1e-4*randn(50);

norm(A-Q3*R3)/norm(A)
     8.3945e-04
```

Results from the matlab experiment indicate that Householder QR only causes small error when we reconstruct the matrix such tha $A = QR$. This is happening while the difference between the true Q, R and the Householder Q_2, R_2 is very big. On the other hand, if we slightly perturb the original Q, R, still the error associated with the calculated A and the original A is very big.

Actually the reason is that the errors in Q_2 and R_2 are forward error that are due to an ill-conditioned problem. On the other hand, the error in Q_2R_2 is the backward error or residual. The smallness of this error suggests that Householder triangularization is backward stable.

Lab 12

Least Squares Conditioning

12.1 Conditioning

The book showed that condition numbers from b to $y = Ax$ and x were given by

$$\begin{aligned}\kappa_{b \to y} &= \frac{1}{\cos(\theta)} \\ \kappa_{b \to x} &= \frac{\kappa_A}{\eta \cos(\theta)}\end{aligned}$$

where θ is the angle between b and y, κ_A is the condition number of A, and

$$\eta = \frac{\|A\|\|x\|}{\|Ax\|}.$$

You are going to empirically verify this. You will generate five random least squares problems of dimension $m = 3$ and $n = 2$. Solve them to obtain x and y by MatLab's internal solver (right division). Using these values you can calculate the above quantities. Now generate three different perturbations of b for each problem, so that $\|\delta b\|_2 = 10^{-6}\|b\|$ for one, $\|\delta b\|_2 = 10^{-1}\|b\|$ for another, and $\|\delta b\|_2 = \|b\|$ for the third.

You know know A, b, x, y, δb, and can easily calculate $P_A(b + \delta b)$ and $A^\dagger(b + \delta b)$. Use the definition of condition number and compare to the theoretical numbers.

Recall that:

$$\kappa = \sup_{\delta z} \frac{\|\delta f(z)\|}{\|f(z)\|} \frac{\|z\|}{\|\delta z\|}$$

so for the mapping from b to y we have

$$\begin{aligned}\kappa &= \sup_{\delta b} \frac{\|P_A(b - \delta b) - P_A b\|}{\|P_A b\|} \frac{\|b\|}{\|\delta b\|} \\ &= \sup_{\delta b} \frac{\|P_A \delta b\|}{\|P_A b\|} \frac{\|b\|}{\|\delta b\|} \\ &\geq \frac{\|P_A \delta b\|}{\|P_A b\|} \frac{\|b\|}{\|\delta b\|}.\end{aligned}$$

You have to find the equation for b to x.

12.2 Covariance

The covariance matrix for an $m \times n$ least squares problem, $Ax \cong b$, is given by:

$$\begin{aligned} C &= \sigma^2 (A^T A)^{-1} \\ &\sigma = \|Ax_{ls} - b\|_2^2/(m-n) \end{aligned}$$

The entries specify goodness of fit and cross correlation information, and thus is very useful. If we have $A = QR$ then this can be rewritten as

$$\begin{aligned} C &= \sigma^2 (R^T R)^{-1} \\ &\sigma = \|Rx_{ls} - Q^T b\|_2^2/(m-n) \end{aligned}$$

Write a complete Least Squares solver, including your own QR algorithm that returns R, $Q^T b$, and C. Use this new routine to solve the following problem.

12.3 Solutions to Lab 12

12.3.1 Mapping from b to x

$$\begin{aligned} \kappa &= \sup_{\delta b} \frac{\|A^\dagger(b+\delta b) - A^\dagger b\|}{\|A^\dagger b\|} \frac{\|b\|}{\|\delta b\|} \\ &= \sup_{\delta b} \frac{\|A^\dagger \delta b\|}{\|A^\dagger b\|} \frac{\|b\|}{\|\delta b\|} \\ &\geq \frac{\|A^\dagger \delta b\|}{\|A^\dagger b\|} \frac{\|b\|}{\|\delta b\|}. \end{aligned}$$

12.3.2 Matlab Code for Conditioning

```
clear;
data = zeros(30,1);
for i=1:5
    m = 3; n = 2;
    A = randn(m,n);
    b = randn(m,1);
    x = A\b;
    y = A*x;
    r = rank(A);
    A_pinv = pinv(A);
    [u,s,v] = svd(A);
    P_A = A*A_pinv;
    norm_b = norm(b);
    norm_y = norm(y);

    % condition numbers
    cos_theta = norm_y/norm_b;
    eta = (norm(A) * norm(x)) / norm(A*x);
    k_b_y = 1/cos_theta;
    k_b_x = cond(A)/(eta * cos_theta);

    % generate perturbations of b
    p_1 = 10^(-6).*randn(m,1);
    p_2 = 10^(-1).*randn(m,1);
    p_3 = randn(m,1);

    % comparing condition number base on definition vs theoretical
    def_k_b_y_1 = (norm(P_A*p_1) / norm(P_A *b)) * (norm_b / norm(p_1));
    def_k_b_y_2 = (norm(P_A*p_2) / norm(P_A *b)) * (norm_b / norm(p_2));
    def_k_b_y_3 = (norm(P_A*p_3) / norm(P_A *b)) * (norm_b / norm(p_3));

    indx = (i-1) * 6;
    data(indx + 1) = k_b_y - def_k_b_y_1;
    data(indx + 2) = k_b_y - def_k_b_y_2;
    data(indx + 3) = k_b_y - def_k_b_y_3;

    % comparing condition number base on definition vs theoretical
    def_k_b_x_1 = (norm(A_pinv*p_1) / norm(A_pinv *b)) * (norm_b / norm(p_1));
    def_k_b_x_2 = (norm(A_pinv*p_2) / norm(A_pinv *b)) * (norm_b / norm(p_2));
    def_k_b_x_3 = (norm(A_pinv*p_3) / norm(A_pinv *b)) * (norm_b / norm(p_3));

    data(indx + 4) = k_b_x - def_k_b_x_1;
    data(indx + 5) = k_b_x - def_k_b_x_2;
    data(indx + 6) = k_b_x - def_k_b_x_3;
end


plot(data,ones(30,1),'-b');
```

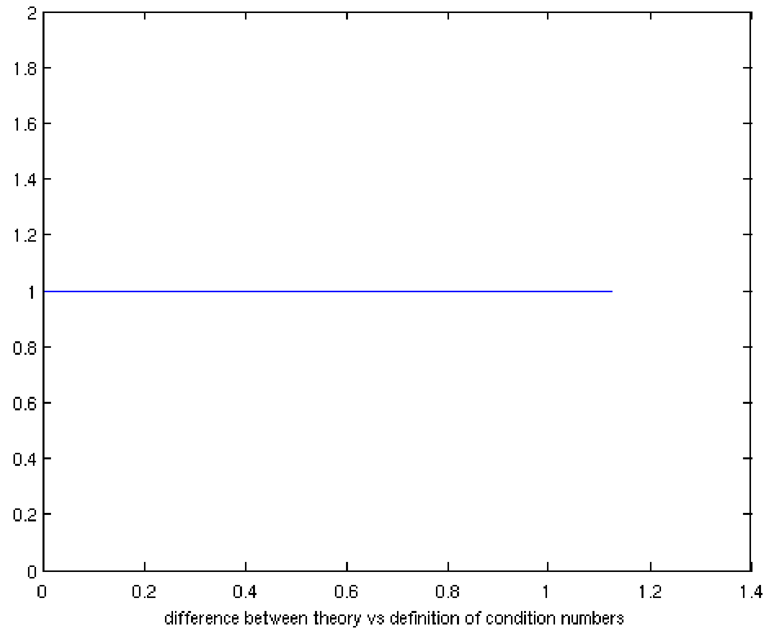


Figure 12.1: Visualization of the difference between the condition numbers calculated based on theory and definition As we expect, the difference is always positive which is compatible with the definition of condition numbers.

12.3.3 Matlab Code for Covariance

```
function [C, x] = Covariance(A,b)
     [m,n] = size(A);
     [Q_b,R] = qr_alg(A,b);
     x = backSubstitute(R,Q_b);
     sigma = norm(R * x - Q_b) /(m-n)
     C = sigma^2 .*inv(R'*R)

end

clear;
data = zeros(30,1);
for i=1:5
     m = 3; n = 2;
     A = randn(m,n);
     b = randn(m,1);
     [C,x] = Covariance(A,b);
     y = A*x;
     r = rank(A);
```

```
    A_pinv = pinv(A);
    [u,s,v] = svd(A);
    P_A = A*A_pinv;
    norm_b = norm(b);
    norm_y = norm(y);

    % condition numbers
    cos_theta = norm_y/norm_b;
    eta = (norm(A) * norm(x)) / norm(A*x);
    k_b_y = 1/cos_theta;
    k_b_x = cond(A)/(eta * cos_theta);

    % generate perturbations of b
    p_1 = 10^(-6).*randn(m,1);
    p_2 = 10^(-1).*randn(m,1);
    p_3 = randn(m,1);

    % comparing condition number base on definition vs theoretical
    def_k_b_y_1 = (norm(P_A*p_1) / norm(P_A *b)) * (norm_b / norm(p_1));
    def_k_b_y_2 = (norm(P_A*p_2) / norm(P_A *b)) * (norm_b / norm(p_2));
    def_k_b_y_3 = (norm(P_A*p_3) / norm(P_A *b)) * (norm_b / norm(p_3));

    indx = (i-1) * 6;
    data(indx + 1) = k_b_y - def_k_b_y_1;
    data(indx + 2) = k_b_y - def_k_b_y_2;
    data(indx + 3) = k_b_y - def_k_b_y_3;

    % comparing condition number base on definition vs theoretical
    def_k_b_x_1 = (norm(A_pinv*p_1) / norm(A_pinv *b)) * (norm_b / norm(p_1));
    def_k_b_x_2 = (norm(A_pinv*p_2) / norm(A_pinv *b)) * (norm_b / norm(p_2));
    def_k_b_x_3 = (norm(A_pinv*p_3) / norm(A_pinv *b)) * (norm_b / norm(p_3));

    data(indx + 4) = k_b_x - def_k_b_x_1;
    data(indx + 5) = k_b_x - def_k_b_x_2;
    data(indx + 6) = k_b_x - def_k_b_x_3;
end


plot(data,ones(30,1),'-b');
```

```
>> Lab12_part2

sigma =
    2.4825e-16


C =

    1.0e-31 *

      0.2597      0.0554
      0.0554      0.2930


sigma =

       0


C =

       0        0
       0        0


sigma =

    5.5511e-17


C =

    1.0e-32 *

      0.8759      0.8425
      0.8425      0.8682


sigma =

       0


C =

       0        0
       0        0
```

```
sigma =

     0


C =

     0     0
     0     0
```

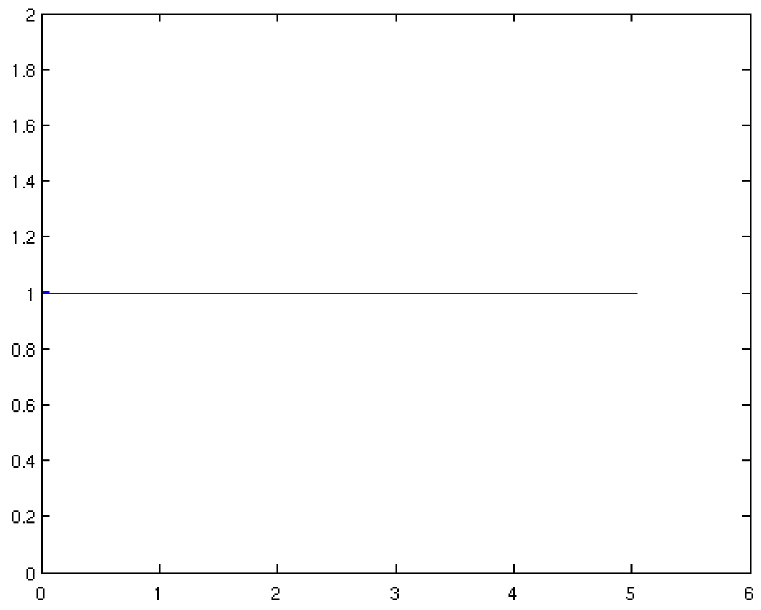


Figure 12.2: Visualization of the difference between the condition numbers calculated based on theory and definition As we expect, the difference is always positive which is compatible with the definition of condition numbers.

Lab 13

Eigenvalue Problems

Do exercise 24.1 and 24.2

13.1 Solutions to Lab 10

1. Exercise 24.1

   (a) True. Assuming that A is a non-defective matrix $A = X\Lambda X^{-1}$, we want to find the ew of $A - \mu I$.

   $$\begin{aligned} A - \mu I &= X\Lambda X^{-1} - \mu I & (13.1)\\ &= X\Lambda X^{-1} - \mu X X^{-1} & (13.2)\\ &= X(\Lambda X^{-1} - \mu X^{-1}) & (13.3)\\ &= X(\Lambda - \mu I)X^{-1} & (13.4)\\ & & (13.5) \end{aligned}$$

   Based on the above formula, the ev of $A - \mu I$ is the same as A, but the ew of $A - \mu I$ is $(\Lambda - \mu I)$.

   (b) False. If $A = I$, 1 is an ew, but -1 is not.

   (c) True. If λ is an ew of real matrix A, then λ is a root of the real coefficient polynomial $det(A - xI)$, in other words $det(A - \lambda I) = 0$. The complex conjugate root theorem states that if P is a polynomial in one variable with real coefficients, and $a + bi$ is a root of P with a and b real numbers, , then its complex conjugate $a - bi$ is also a root of P. Therefore $det(A - \overline{\lambda} I) = 0$, and thus $\overline{\lambda}$ is also an ew of A.

   (d) True. If A is non-singular, A^{-1} exists, thus

   $$\begin{aligned} I &= AA^{-1} & (13.6)\\ &= X\Lambda X^{-1} X \Lambda^{-1} X^{-1} & (13.7)\\ &= X\Lambda I \Lambda^{-1} X^{-1} & (13.8) \end{aligned}$$

In order to make the (13.8) be equal to I, $\Lambda\Lambda^{-1}$ should be equal to I as well. Therefore $\Lambda^{-1} = \text{diag}(\frac{1}{\lambda_1} \ldots \frac{1}{\lambda_n})$. Thus, we continue with from (13.8):

$$\begin{aligned} X\Lambda\Lambda^{-1}X^{-1} & & (13.9)\\ &= XIX^{-1} & (13.10)\\ &= XX^{-1} & (13.11)\\ &= I & (13.12)\end{aligned}$$

Therefore we conclude that if λ is an ew of A and A is nonsingular, then λ^{-1} is an ew of A^{-1}.

(e) False. Considering, $A = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$. It is clear that all the ew's of A are zero, but $A \neq 0$.

(f) True. A hermitian matrix has a complete set of orthogonal eigenvectors, and all of the eigenvalues are real. In other words, we can write A as

$$\begin{aligned} A &= Q\Lambda Q^* & (13.13)\\ &= Q|\Lambda| sign(\Lambda) Q^* & (13.14)\end{aligned}$$

Since $sign(\Lambda)Q^*$ is unitary whenever Q is unitary, (13.14) is an SVD of A with the singular values equal to the diagonal entries of $|\Lambda|$, $|\lambda_j|$.

(g) True. If A is diagonalizable,

$$\begin{aligned} A &= X\Lambda X^{-1} & (13.15)\\ &= X\lambda I X^{-1} & (13.16)\\ &= \lambda X I X^{-1} & (13.17)\\ &= \lambda X X^{-1} & (13.18)\\ &= \lambda I & (13.19)\end{aligned}$$

2. Exercise 24.2

(a) $Ax = \lambda x$ implies that:

$$\Sigma_{j\neq i} a_{ij} x_j = (\lambda - a_{ii}) x_i \quad (13.20)$$

Then, we take the norm of both side and divide by x_i:

$$\begin{aligned} |\Sigma_{j\neq i} a_{ij} x_j| &= |\lambda - a_{ii}| & (13.21)\\ \leq \Sigma_{j\neq i} |a_{ij} x_j| & & (13.22)\\ \leq \Sigma_{j\neq i} |a_{ij}| & & (13.23)\end{aligned}$$

(b) Based on the fact that an ew always has to be within a disc, and due to the continuity of the ew's path there is no way that an ew can move from one isolated group to another isolated group without being found in a region outside of any disc. Being outside of a disc violates Gerschgorin Theorem and therefore every disjoing group that has n discs in it must have n eigenvalues in it.

Lab 14

Householder Reduction

Write a MatLab function to implement Algorithm 26.1. Note it is very similar to your QR algorithm, so you should be able to implement it quickly.

Test your algorithm by generating five 10x10 symmetric matrices[1], and verifying your algorithm preserves the eigenvalues.

Now implement a MatLab function to implement algorithm 27.3 (Rayleigh Quotient Iteration). Use your implementation of Algorithm 26.1 as a preconditioner in your implementation of the Rayleigh Quotient Iteration. Test your implementation by redoing the last lab and comparing your results to the ones generated by spec.

14.1 Solutions to Lab 14

14.1.1 Matlab Implementation of Householder Reduction

```
function [R] = hh_reduction(A)
% Implementation of the Householder reduction Algorithm

[m, n] = size(A);
R = A;
for k=1:m-2
    x=R(k+1:m,k);
    e = zeros(length(x),1); e(1) = 1;
    v = sign(x(1))*norm(x)*e + x;
    v = v/norm(v);
    R(k+1:m,k:m) = R(k+1:m,k:m) - 2*v*(v'*R(k+1:m,k:m));
    R(1:m,k+1:m) = R(1:m,k+1:m) - 2*(R(1:m,k+1:m)*v)*v';
end
```

[1]Generate a random 10x10 matrix and copy the lower triangle to the upper triangle or vice versa.

14.1.2 Testing the Householder Reduction Code

```
clear
n = 10;

for i = 1:5
    A = randn(n,1);
    A_sym = toeplitz(A);
    [R] = hh_reduction(A_sym);
    r = norm(sort(eig(R)) - sort(eig(A_sym)))
end

>> Lab14

r =

   5.5269e-15


r =

   1.4156e-14


r =

   1.2351e-14


r =

   6.8803e-15


r =

   7.7286e-15
```

14.1.3 Matlab Implementation of Rayleigh Quotient Iteration

```
function [lambda,v0] = rayleigh_quotient(A,n,iter)
% A: input matrix of size n by n
% iter: Number of iterations
% lambda: Eigen value
% v0: Eigen vector

    v0 = randn(n,1);
    v0 = v0/norm(v0);
    lambda = (v0'*A*v0)/(v0'*v0);
    for k=1:iter
        M = A - lambda*eye(size(A));
        w = M\v0;
        v0 = w/norm(w);
        lambda = v0'*A*v0;
    end
end
```

Lab 15

Power and Inverse Iteration

Implement Algorithm 27.1 (power iteration) and Algorithm 27.2 (inverse iteration). Compare them to the Rayleigh quotient iteration you wrote in the last lab. How much does Householder reduction help each?

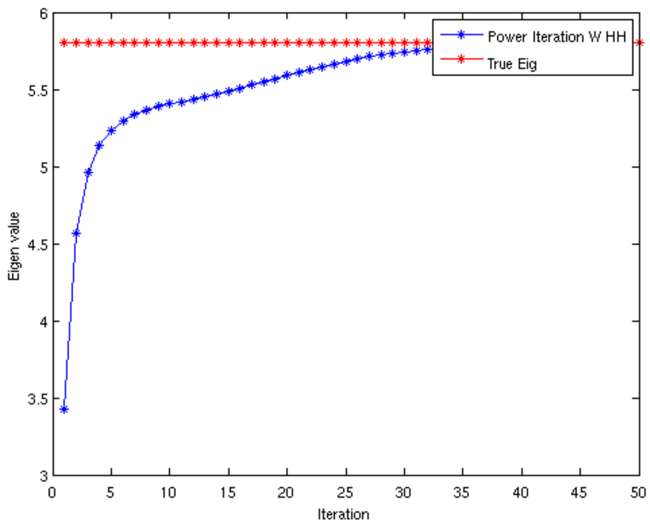
15.1 Solutions to Lab 15

In this lab, the convergence rate of three methods with and without householder reduction has been illustrated with random matrices.

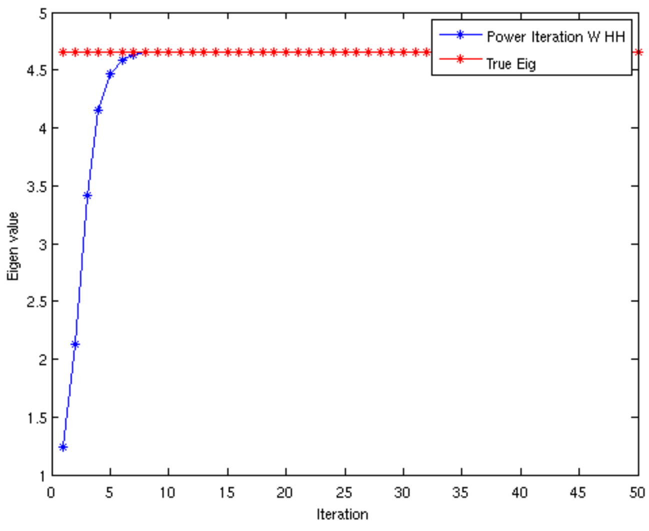
Figures 15.1, and 15.2 illustrate the convergence rate of the Power Iteration algorithm with and without Householder reduction. Based on the results in these two figures, Householder reduction helps the Power Iteration algorithm converges faster on average.

Figures 15.3, and 15.4 illustrate the convergence rate of the Inverse Iteration algorithm with and without Householder reduction. Based on the results in these two figures, Householder reduction helps the Inverse Iteration algorithm converges faster on average.

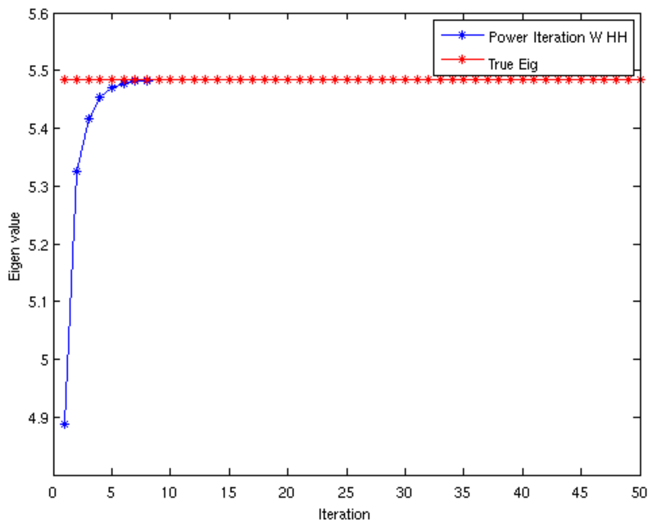
Figures 15.5, and 15.6 illustrate the convergence rate of the Rayleigh Quotient algorithm with and without Householder reduction. The convergence rate of Rayleigh Quotient is so fast because it is cubic convergence. It usually converges to the true eigenvalue after three iterations. It does not make a huge difference if we use householder or not, since its convergence rate is very fast anyway.



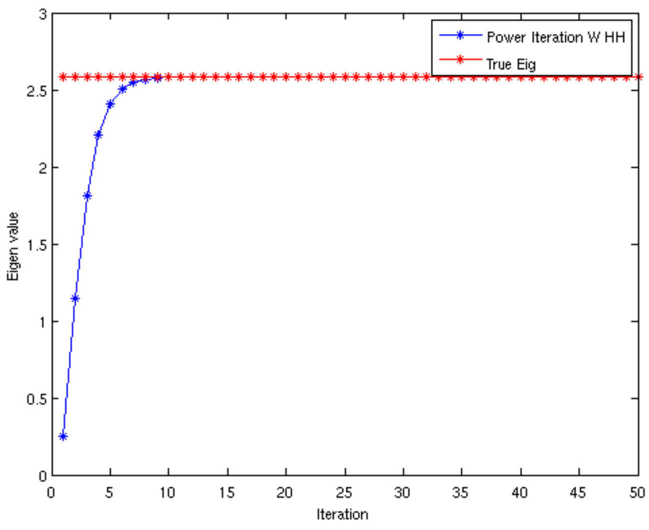
(a)



(b)



(c)



(d)

Figure 15.1: Convergence of Power Iteration with householder reduction.

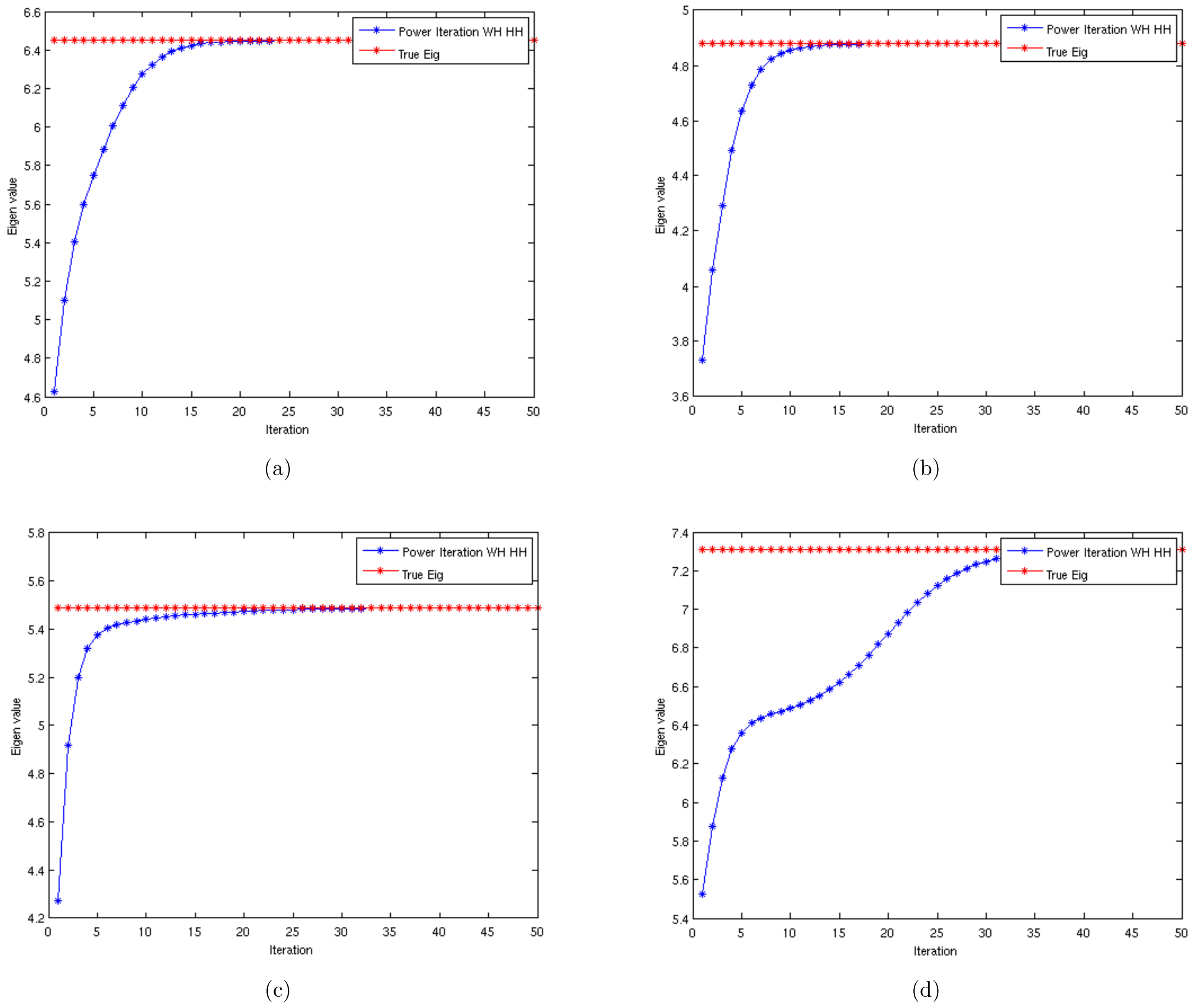
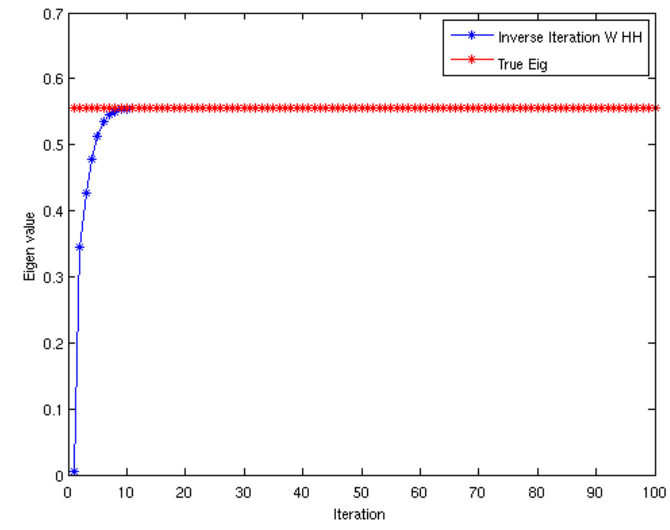
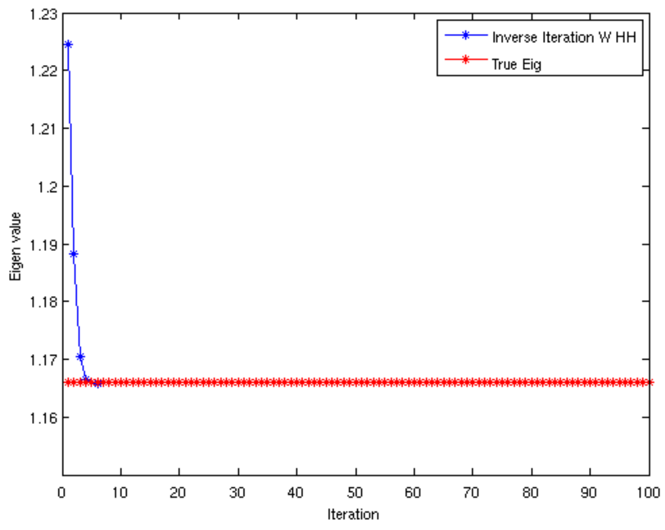


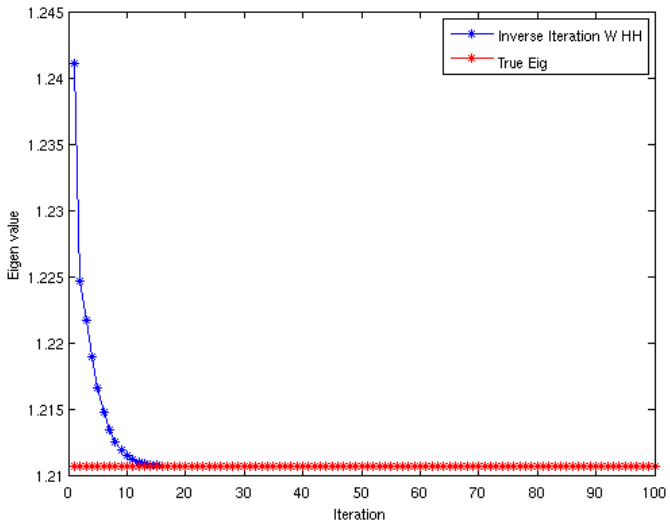
Figure 15.2: Convergence of Power Iteration without householder reduction.



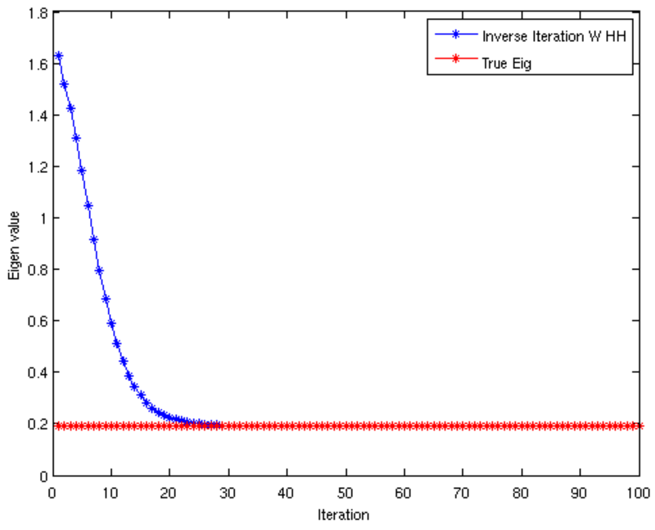
(a)



(b)

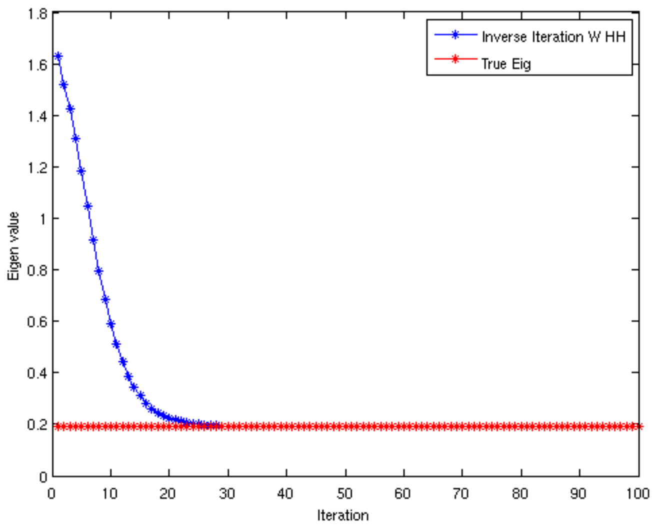


(c)

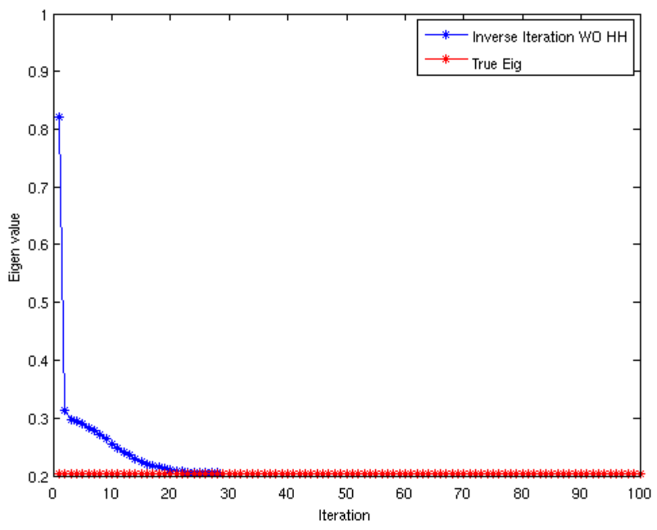


(d)

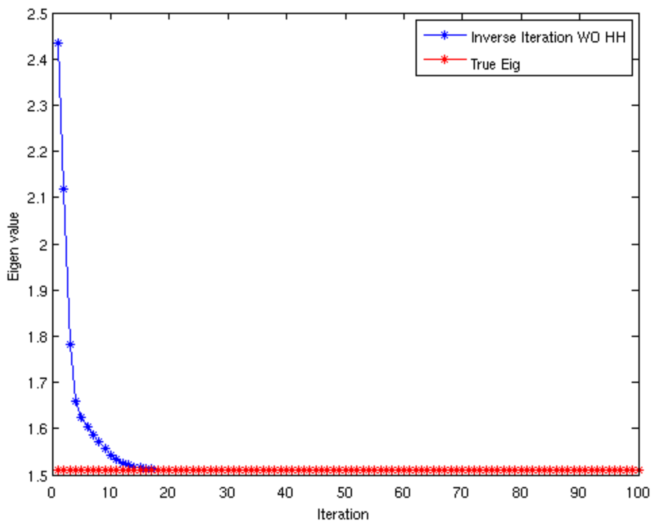
Figure 15.3: Convergence of Inverse Iteration with householder reduction.



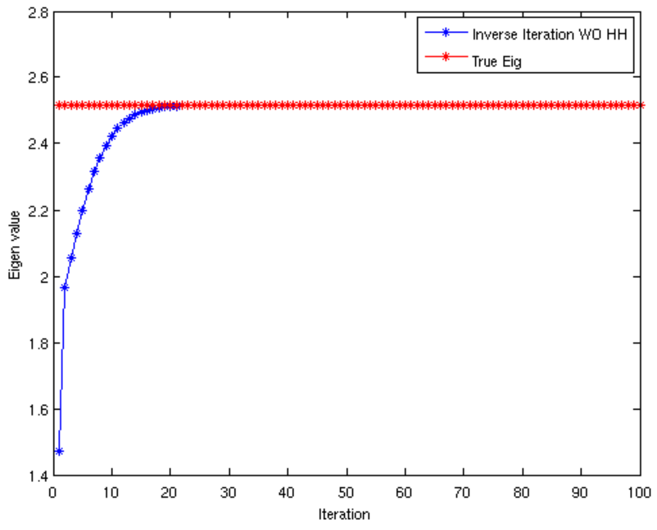
(a)



(b)

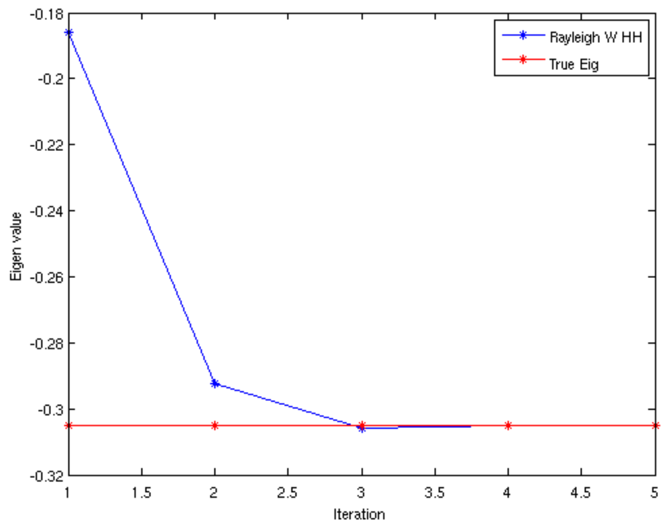


(c)

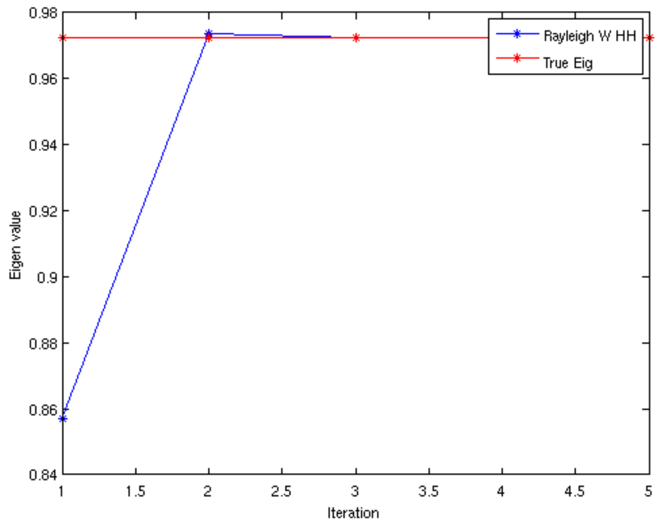


(d)

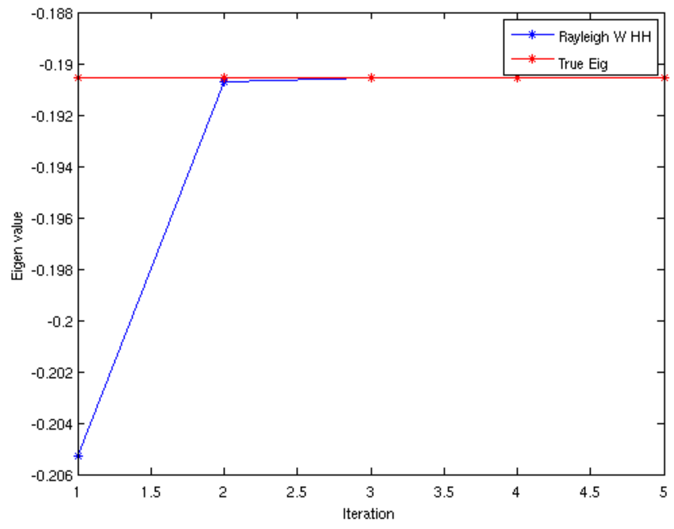
Figure 15.4: Convergence of Inverse Iteration without householder reduction.



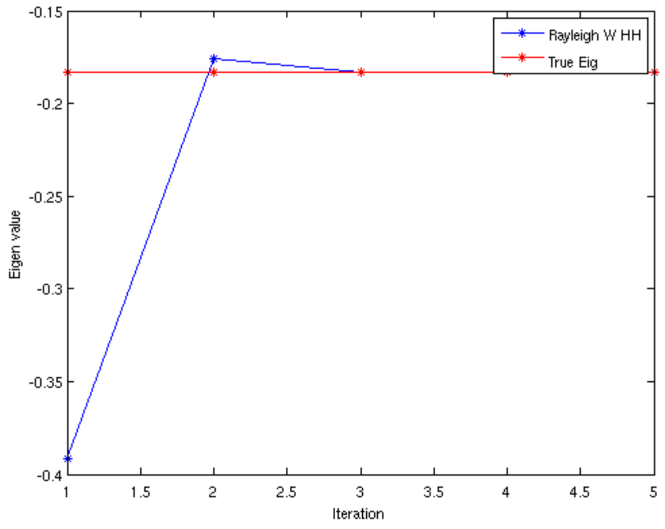
(a)



(b)

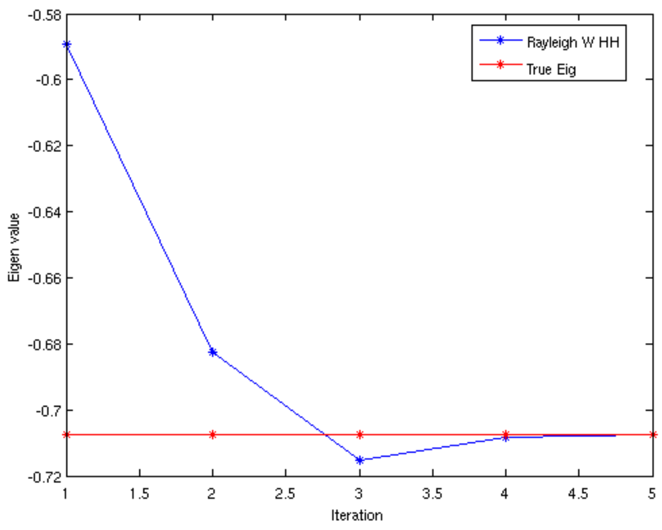


(c)

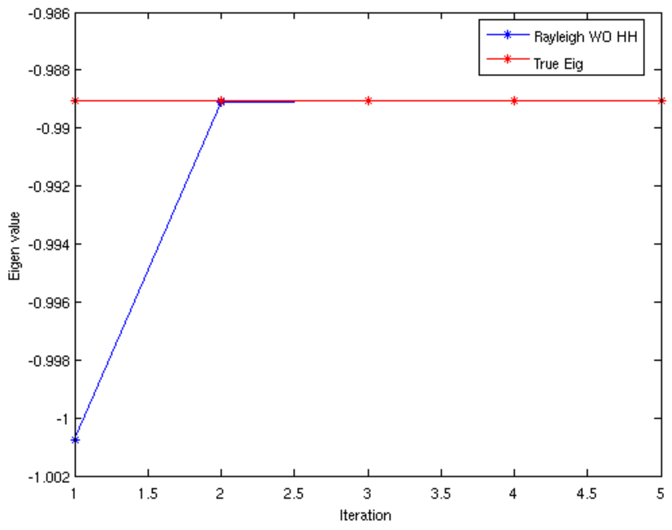


(d)

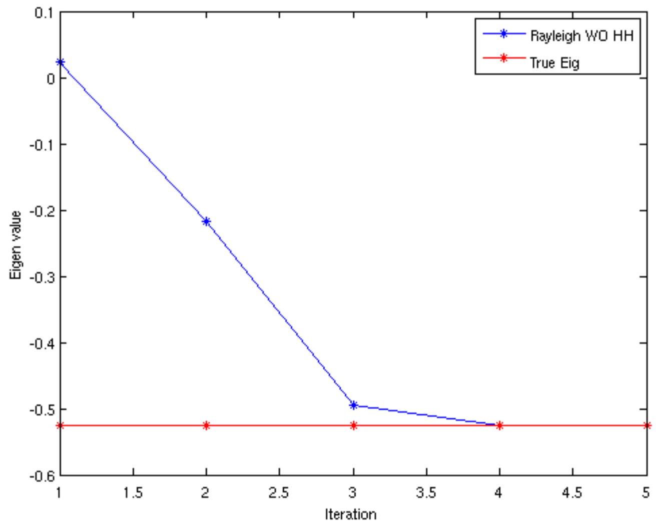
Figure 15.5: Convergence of Rayleigh Quotient with householder reduction.



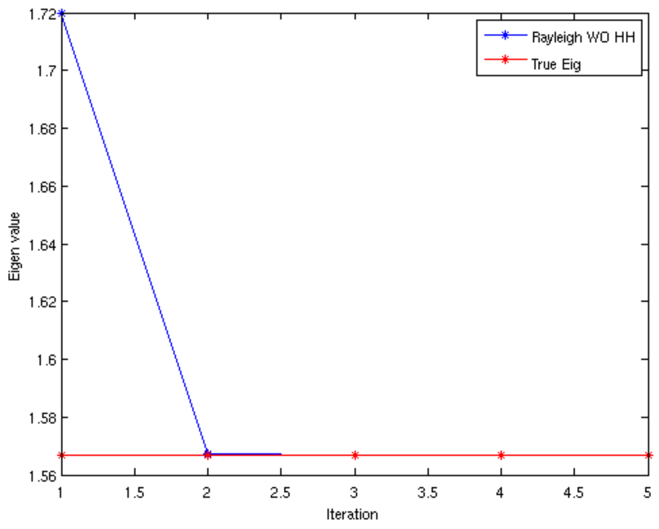
(a)



(b)



(c)



(d)

Figure 15.6: Convergence of Rayleigh Quotient without householder reduction.

15.2 Matlab Code

15.2.1 Power Iteration

```
function [lambda,lambda_v] = power_iteration(A,n,iter)
% A: input matrix of size n by n
% iter: Number of iterations
% lambda: Largest eigen value of A

    v0 = rand(n,1);
    v0 = v0/norm(v0);
    lambda_v = zeros(iter,1);
    for k =1:iter
        w = A*v0;
        v0 = w/norm(w);
        lambda = v0'*A*v0;
        lambda_v(k) = lambda;
    end
end
```

```
clear
n = 10;
iter = 50;

for i = 1:4
    A = randn(n,1);
    A_sym = toeplitz(A);
    [R] = hh_reduction(A_sym);
    eig_A = eig(A_sym);
    [lambda1,lambda_v1] = power_iteration(R,n,iter);
    [v,indx_1] = max(abs(eig_A));
    figure
    plot((1:iter),(abs(lambda_v1)),'-b*');
    hold on
    true_eig = v.*ones(1,iter);
    plot((1:iter), true_eig,'-r*');
    xlabel('Iteration');
    ylabel('Eigen value');
    legend('Power Iteration W HH','True Eig');
end
```

```
clear
n = 10;
epsilon = 10*10^(-5);
iter = 50;

for i = 1:10
    A = randn(n,1);
    A_sym = toeplitz(A);
    eig_A = eig(A_sym);
    [lambda1,lambda_v1] = power_iteration(A_sym,n,iter);
    [v,indx_1] = max(abs(eig_A));
    figure
    plot((1:iter),(abs(lambda_v1)),'-b*');
    hold on
    true_eig = v.*ones(1,iter);
    plot((1:iter), true_eig,'-r*');
    xlabel('Iteration');
    ylabel('Eigen value');
    legend('Power Iteration WH HH','True Eig');
end
```

15.2.2 Inverse Iteration

```
function [lambda,lambda_v,v0] = inverse_iteration(A,n,iter)
% A: input matrix of size n by n
% iter: Number of iterations
% lambda: Eigen value

    v0 = rand(n,1);
    v0 = v0/norm(v0);
    mu = randn(1,1);
    lambda_v = zeros(iter,1);
    for k=1:iter
        w = (A - mu*eye(size(A)))\v0;
        v0 = w/norm(w);
        lambda = v0'*A*v0;
        lambda_v(k) = lambda;
    end
end
```

```
clear
n = 10;
epsilon = 10*10^(-5);
iter = 100;

for i = 1:4
    A = randn(n,1);
    A_sym = toeplitz(A);
    [R] = hh_reduction(A_sym);
    eig_A = eig(A_sym);
    [lambda1,lambda_v1,v0] = inverse_iteration(R,n,iter);
    indx_1 = find(abs(lambda1 - eig_A) < epsilon);
    figure
    plot((1:iter),(abs(lambda_v1)),'-b*');
    hold on
    true_eig = eig_A(indx_1).*ones(1,iter);
    plot((1:iter), true_eig,'-r*');
    xlabel('Iteration');
    ylabel('Eigen value');
    legend('Inverse Iteration W HH','True Eig');
end
```

```
clear
n = 10;
epsilon = 10*10^(-10);
iter = 100;

for i = 1:10
    A = randn(n,1);
    A_sym = toeplitz(A);
    eig_A = eig(A_sym);
    [lambda1,lambda_v1,v0] = inverse_iteration(A_sym,n,iter);
    indx_1 = find(abs(lambda1 - eig_A) < epsilon);
    figure
    plot((1:iter),(abs(lambda_v1)),'-b*');
    hold on
    true_eig = eig_A(indx_1).*ones(1,iter);
    plot((1:iter), true_eig,'-r*');
    xlabel('Iteration');
    ylabel('Eigen value');
    legend('Inverse Iteration WO HH','True Eig');
end
```

15.2.3 Rayleigh Quotient

```
function [lambda,v0] = rayleigh_quotient(A,n,iter)
% A: input matrix of size n by n
% iter: Number of iterations
% lambda: Eigen value
% v0: Eigen vector

    v0 = randn(n,1);
    v0 = v0/norm(v0);
    lambda = (v0'*A*v0)/(v0'*v0);
    for k=1:iter
        M = A - lambda*eye(size(A));
        w = M\v0;
        v0 = w/norm(w);
        lambda = v0'*A*v0;
    end
end
```

```
clear
n = 10;
epsilon = 10*10^(-5);
iter = 5;

for i = 1:4
    A = randn(n,1);
    A_sym = toeplitz(A);
    [R] = hh_reduction(A_sym);
    eig_A = eig(A_sym);
    [lambda1,lambda_v1,v01] = rayleigh_quotient(R,n,iter);
    indx_1 = find(abs(lambda1 - eig_A) < epsilon);
    figure
    plot((1:iter),(lambda_v1),'-b*');
    hold on
    true_eig = eig_A(indx_1).*ones(1,iter);
    plot((1:iter), true_eig,'-r*');
    xlabel('Iteration');
    ylabel('Eigen value');
    legend('Rayleigh W HH','True Eig');
end
```

```
clear
n = 10;
epsilon = 10*10^(-5);
iter = 5;

for i = 1:4
    A = randn(n,1);
    A_sym = toeplitz(A);
    eig_A = eig(A_sym);
    [lambda1,lambda_v1,v01] = rayleigh_quotient(A_sym,n,iter);
    indx_1 = find(abs(lambda1 - eig_A) < epsilon);
    figure
    plot((1:iter),(lambda_v1),'-b*');
    hold on
    true_eig = eig_A(indx_1).*ones(1,iter);
    plot((1:iter), true_eig,'-r*');
    xlabel('Iteration');
    ylabel('Eigen value');
    legend('Rayleigh WH HH','True Eig');
end
```

```
function [R] = hh_reduction(A)
% Implementation of the Householder reduction Algorithm

[m, n] = size(A);
R = A;
for k=1:m-2
    x=R(k+1:m,k);
    e = zeros(length(x),1); e(1) = 1;
    v = sign(x(1))*norm(x)*e + x;
    v = v/norm(v);
    R(k+1:m,k:m) = R(k+1:m,k:m) - 2*v*(v'*R(k+1:m,k:m));
    R(1:m,k+1:m) = R(1:m,k+1:m) - 2*(R(1:m,k+1:m)*v)*v';
end
```

Lab 16

Eigenvalues by QR

Write MatLab functions to implement Algorithm 28.1 and Algorithm 28.2. In both algorithms the loop does not have a specified exit condition. Write your functions so that you can tell it how many iterations you want it to run, i.e.: make the iteration count a parameter you pass to the functions.

Generate a random symmetric matrix A with known eignvalues λ as follows:

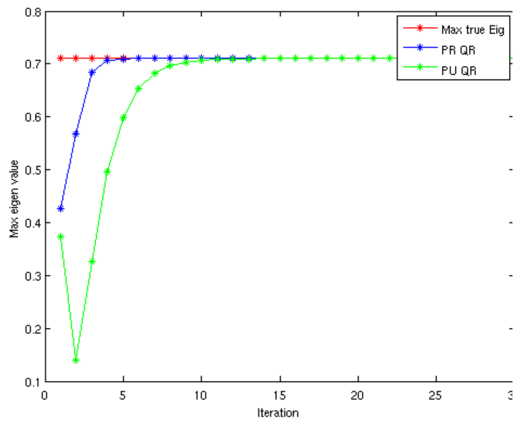
```
$T \leftarrow rand(n, n)$
$[Q, R] \leftarrow qr(T)$
$\lambda \leftarrow rand(n, 1)$
$A \leftarrow Q^H * diag(\lambda) * Q$
```

You are going to solve for the eigenvalues of A using both algorithms for iteration counts from 1 to 30, and plot the errors[1]. From this you can check the convergence of your algorithms. Which performs better and by how much? How many steps did it take to get 4 digits of accuracy?

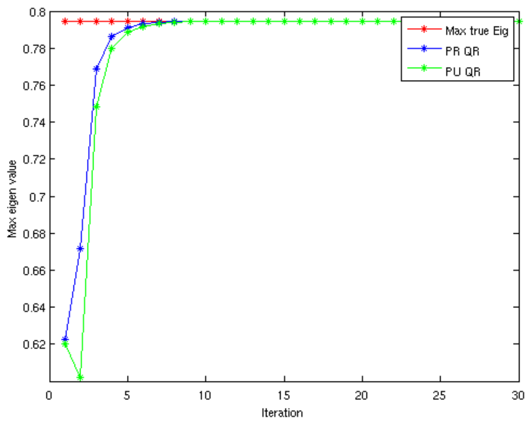
16.1 Solutions to Lab 16

The following figures illustrate the convergence rate of the pure QR (28.1) vs the practical QR (28.2) algorithms. Based on the results, practical QR algorithms performs better. In fact, the convergence rate of the pure vs practical QR is linear with constant $\max_k |\lambda_{(k+1)}/\lambda_k|$, where k is the number of iterations.[2] vs cubic convergence. Therefore, because of the cubic convergence rate of the practical QR algorithm, in most of the trials, it takes 3-4 steps to get the 4 digits of accuracy.

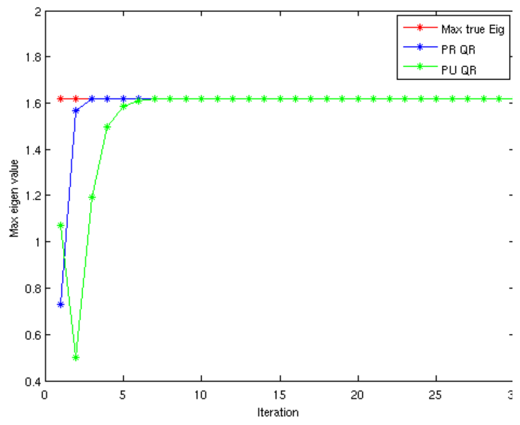
[1]You can do this since you know the true λ.
[2]Theorem 28.1 in text



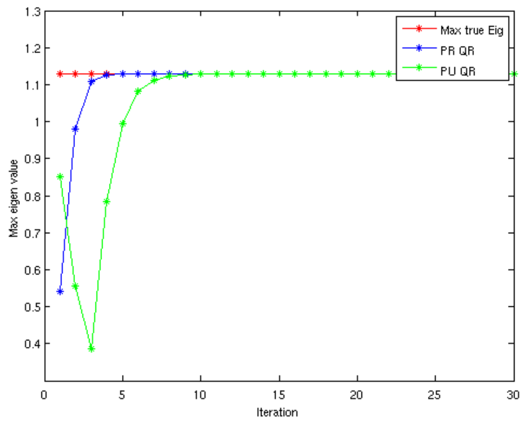
(a)



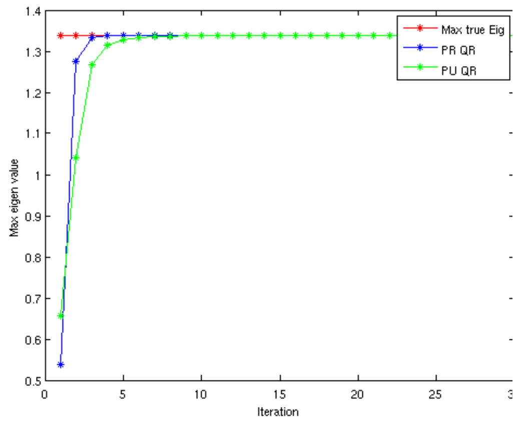
(b)



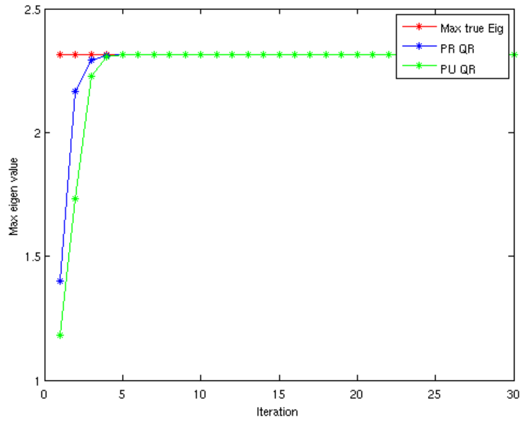
(c)



(d)



(e)



(f)

Figure 16.1: Convergence rate comparison of pure vs practical of 6 random symmetric matrices after 30 iterations. The horizontal axis is the number of QR iterations. The red plot related to the maximum true eigenvalue, the blue plot is for the practical QR and the green plot is for the Pure QR method.

16.2 Matlab Code

16.2.1 Pure QR

```
function [D] = qr_pure(A, k)
    T = A;
    D = zeros(size(A,1),k);
    for i=1:k
        [Q,R] = qr(T);
        T = R*Q;
        D(:,i) = diag(T);
    end
end
```

16.2.2 Practical QR

```
function [D] = qr_practical_main(T, k)
    m = size(T,1);
    %[T] = hh_reduction(A);
    D = zeros(m,k);
    epsilon = 10^(-3);
    for i=1:k
        mu = T(m,m);
        [Q,R] = qr(T - mu.*eye(size(T)));
        T = R*Q + mu.*eye(size(T));
        for j = 1:m-1
            if(abs(T(j,j+1)) < epsilon)
                T(j,j+1) = 0;
                T(j+1,j) = 0;
                if(size(T(1:j,1:j),1) > 1 && size(T(j+1:end,j+1:end),1) > 1 )
                    qr_practical(T(1:j,1:j),k);
                    qr_practical(T(j+1:end,j+1:end),k);
                end
            end
        end
        D(:,i) = diag(T);
        i
    end
end

function qr_practical(T, k)
    m = size(T,1);
    %[T] = hh_reduction(A);
    epsilon = 10^(-3);
    for i=1:k
        mu = T(m,m);
        [Q,R] = qr(T - mu.*eye(size(T)));
        T = R*Q + mu.*eye(size(T));
        for j = 1:m-1
            if(abs(T(j,j+1)) < epsilon)
                T(j,j+1) = 0;
                T(j+1,j) = 0;
                if(size(T(1:j,1:j),1) > 1 && size(T(j+1:end,j+1:end),1) > 1)
                    qr_practical(T(1:j,1:j),k);
                    qr_practical(T(j+1:end,j+1:end),k);
                end
            end
        end
    end
end
```

Lab 17

QR with Shifts

In this exercise you will be building a general solver for real symmetric matrices. This process is nicely outlined in Exercise 29.1.

17.1 Solutions to Lab 17

In this lab, the QR algorithm has implemented with two different shifts, the normal way of shift where $\mu = A(m,m)$, and the Wilkinson shift. Figure 17.1 illustrates the $|t_{m,m-1}|$ at every QR iteration when we have tested `A = hilb(4)` matrix[1]. Based on the results in this figure, after 6 QR iterations, the off-diagonal entries are $|t_{m,m-1}| < 10^{(-12)}$.

At this point, the norm of the Ritz values and the true eigenvalues of the input matrix is in order of $10^{(-12)}$.

Figure 17.2 illustrates the results of the QR algorithm with Wilkinson shifts. As the convergence rate of this method is cubic, we see after 3 iterations, it converges. At this point, the norm of the Ritz values and the true eigenvalues of the input matrix is in order of $10^{(-15)}$.

Figures 17.3, and 17.4 illustrate the results of the QR method with normal and Wilkinson shifts on the input matrix `A = diag(15:-1:1) + ones(15,15)`. Based on the results in Figure 17.3, after 25 iterations, $|t_{m,m-1}| < 10^{(-25)}$. the convergence is close to linear and $1-2$ iterations is enough for each eigenvalue, which we have 15 of them.

But in the case of Wilkinson shift, the convergence is still close cubic which is illustrated in Figure 17.4.

[1]Matrix is reduced to traditional form before being tested by the QR algorithm.

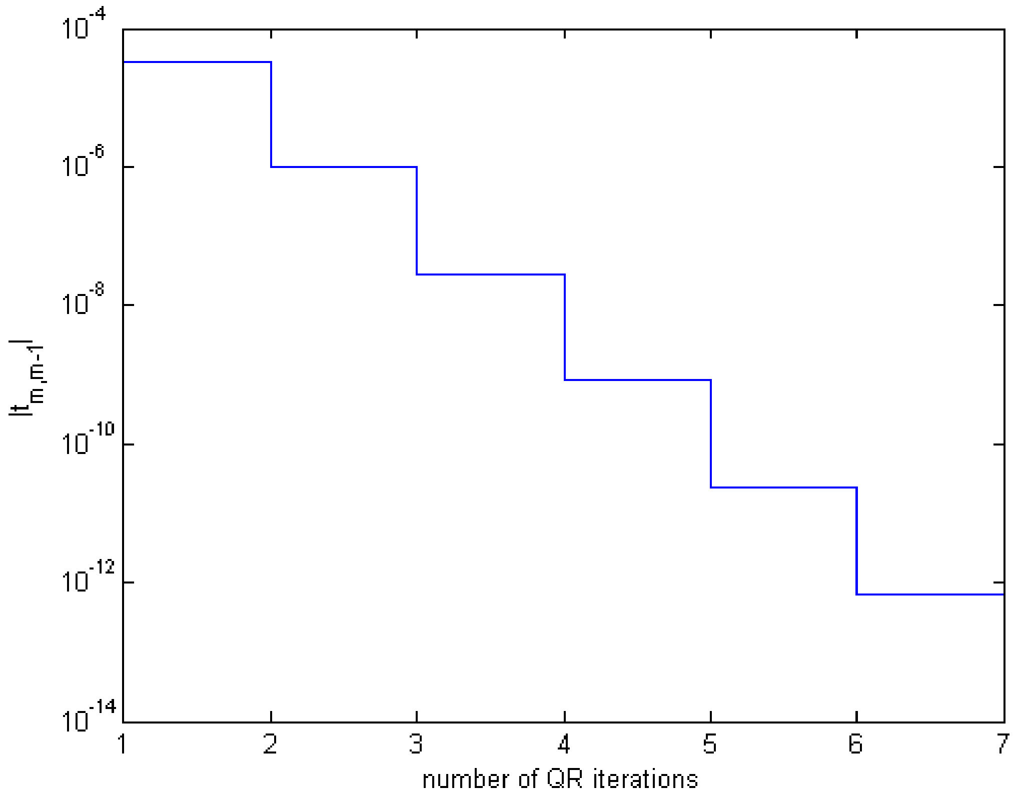


Figure 17.1: QR iterations with normal shifts vs tolerance.

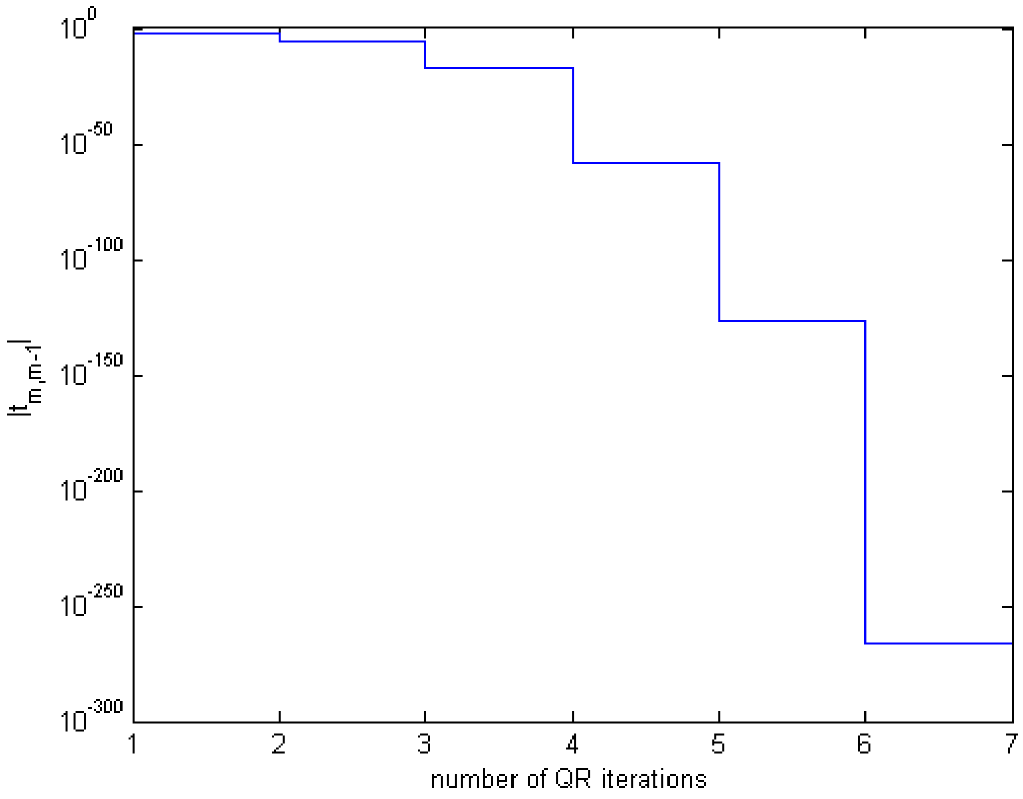


Figure 17.2: QR iterations with Wilkinson shifts vs tolerance.

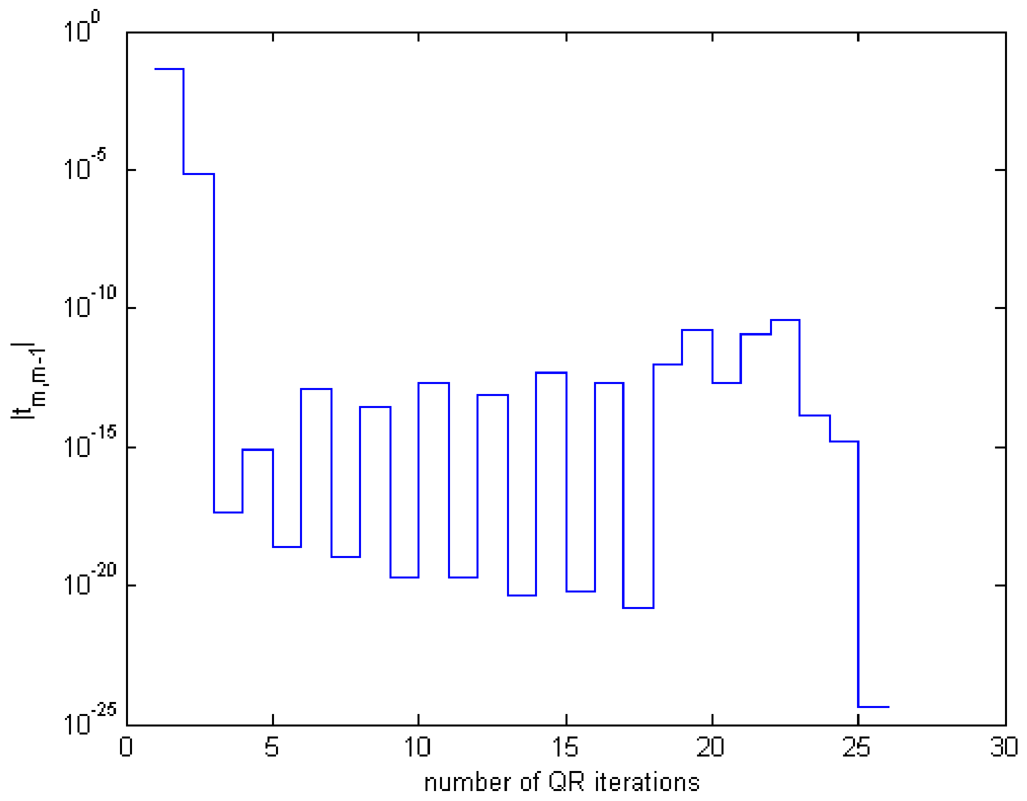


Figure 17.3: QR iterations with Wilkinson shifts vs tolerance.

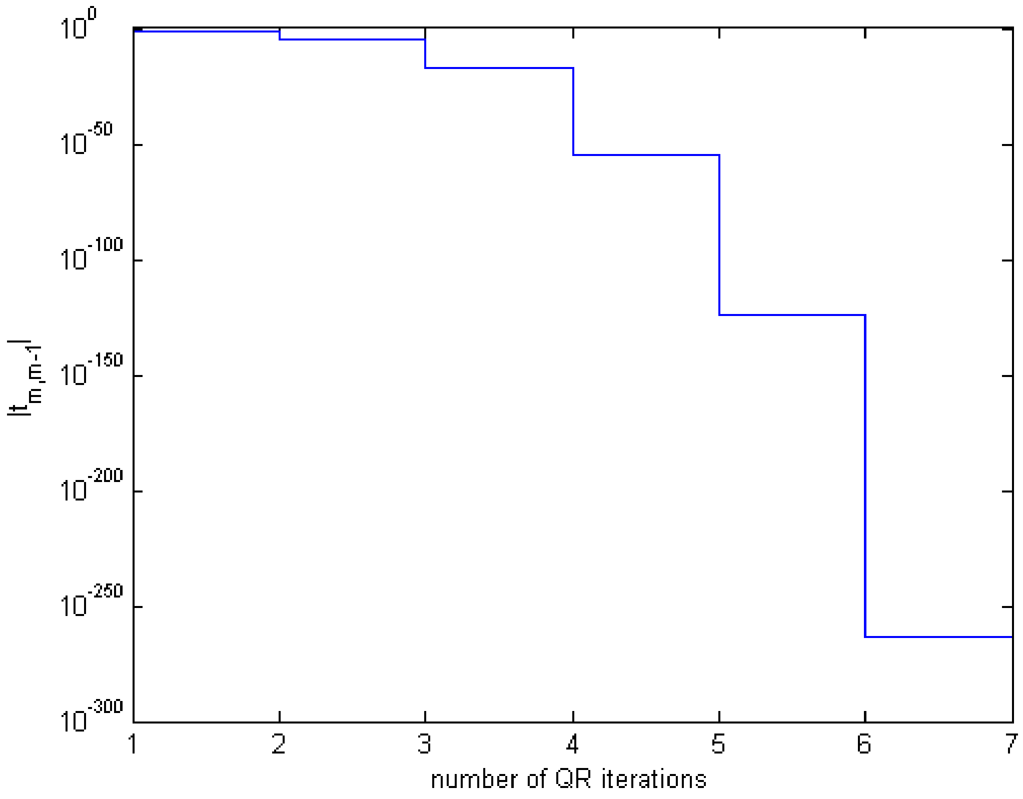


Figure 17.4: QR iterations with Wilkinson shifts vs tolerance.

17.2 Matlab Code

```
function [T_new, off_diag] = qr_shifted_main(T_old, tol)
% Implementing the QR algorithm with normal shift
% Syntax:  [T_new, off_diag] = qr_shifted_main(T_old, tol)
%
% Inputs:
%    T - Input triangular matrix
%    tol - tolerance of the off diagonals
%
% Outputs:
%    T_new - Diagonal of this matrix contains the Ritz valus
%    off_diag - The off diagonal values at each QR iteration
    n =  size(T_old,2);
    T_new = T_old;
    mu = T_new(n,n);
    [c, i] = min(abs(diag(T_new,-1)));
    j = 0;
    off_diag = 0;
    while( c > tol )
        [Q,R] = qr(T_new - mu .* eye(size(T_new)));
        T_new =  R*Q + mu .* eye(size(T_new));
        [c,i] = min(abs(diag(T_new,-1)));
        j=j+1;
        off_diag(j) = c;
    end
    if( (i-1) > 1)
        qr_shifted(T_new(1:i-1,1:i-1), tol, off_diag, j);
    end
    if( (n-i) > 1)
        qr_shifted(T_new(i:n,i:n), tol, off_diag, j);
        deflation = deflation + 1;
    end
    deflation
end
```

```
function qr_shifted(T_new, tol, off_diag, indx)
    n =  size(T_new,2);
    mu = T_new(n,n);
    [c, i] = min(abs(diag(T_new,-1)));
    while( c > tol )
        [Q,R] = qr(T_new - mu .* eye(size(T_new)));
        T_new =  R*Q + mu .* eye(size(T_new));
        [c,i] = min(abs(diag(T_new,-1)));
        indx = indx+1;
        off_diag(indx) = c;
    end
    if( (i-1) > 1)
        qr_shifted(T_new(1:i-1,1:i-1), tol, off_diag, indx);
    end
    if( (n-i) > 1)
        qr_shifted(T_new(i:n,i:n), tol, off_diag, indx);
    end
end
```

```
function [ritz,off_diag] = qr_wilkinson_shift(A,tol)
% Implementing the QR algorithm with Wilkinson shift
% Syntax: [ritz,off_diag] = qr_wilkinson_shift(A,tol)
%
% Inputs:
%    A - Input real symmetric of size m*m
%    tol - toleracen of the off diagonals
%
% Outputs:
%    ritz - Diagonal values of input matrix after some QR iterations
%    off_diag - The off diagonal values at each QR iteration

[n,n] = size(A);
k = 1;
m = n;
iter = 0;
while m > 1
  S = A(m-1:m,m-1:m);
  if abs(S(2,1)) < tol*(abs(S(1,1)) + abs(S(2,2)))
    A(m,m-1) = 0;
    A(m-1,m) = 0;
    m = m-1;
  else
    shift = wilkinson(A);
    [Q,R] = qr(A-shift*eye(n));
    A = R*Q + shift*eye(n);
    iter = iter+1;
    [c,indx] = min(abs(diag(A,-1)));
    off_diag(iter) = c;
  end
  k = k+1;
end
ritz = diag(A);
```

```
function mu = wilkinson(A)
% Wilkinson's shift (mu) of the symmetric matrix A.

[n, n] = size(A);
if (A == diag(diag(A)))
    mu = A(n,n);
    return
end
mu = A(n,n);
if (n > 1)
   d = (A(n-1,n-1)-mu)/2;
   if (d ~= 0)
       sn = sign(d);
   else
       sn = 1;
   end
   bn = A(n,n-1);
   mu = mu - sn*bn^2/(abs(d) + sqrt(d^2+bn^2));
end
```

```
% Exercise 29.1
clear
N = 4;
A = hilb(N);
T = hh_reduction(A);

% QR with shifts
[ritz,off_diag] = qr_shifted_main(T, 10^-12);

figure
step = size(off_diag,2);
for i=step:-1:1
    semilogy((i:i+1),[off_diag(i) off_diag(i)]);
    hold on
    if(i < step)
        semilogy([i+1 i+1],[off_diag(i) off_diag(i+1)]);
    end
end
xlabel('number of QR iterations');
ylabel('|t_{m,m-1}|');

norm(sort(eig(A)) - sort(diag(ritz)))

% QR with Wilkonson shift
[ritz2,off_diag2] = qr_wilkinson_shift(T,10^(-12));


figure
step = size(off_diag2,2);
for i=step:-1:1
    semilogy((i:i+1),[off_diag2(i) off_diag2(i)]);
    hold on
    if(i < step)
        semilogy([i+1 i+1],[off_diag2(i) off_diag2(i+1)]);
    end
end
xlabel('number of QR iterations');
ylabel('|t_{m,m-1}|');

norm(sort(eig(A)) - sort(ritz2))


% Part (e)
A = diag(15:-1:1) + ones(15,15);
T = hh_reduction(A);
```

```
[ritz,off_diag] = qr_shifted_main(A, 10^(-12),1);

figure
step = size(off_diag,2);
for i=step:-1:1
    semilogy((i:i+1),[off_diag(i) off_diag(i)]);
    hold on
    if(i < step)
        semilogy([i+1 i+1],[off_diag(i) off_diag(i+1)]);
    end
end
xlabel('number of QR iterations');
ylabel('|t_{m,m-1}|');

norm(sort(eig(A)) - sort(diag(ritz)))


[ritz2,off_diag2] = qr_wilkinson_shift(A,10^(-12));

figure
step = size(off_diag2,2);
for i=step:-1:1
    semilogy((i:i+1),[off_diag2(i) off_diag2(i)]);
    hold on
    if(i < step)
        semilogy([i+1 i+1],[off_diag2(i) off_diag2(i+1)]);
    end
end
xlabel('number of QR iterations');
ylabel('|t_{m,m-1}|');

norm(sort(eig(A)) - sort(ritz2))
```

Lab 18

Other Eigenvalue Algorithms

Do problem 30.5. (Jacobi)
Do problem 30.6 (Bisection)

18.1 Solutions to Lab 18

1. Problem 30.5 (Jacobi). The QR factorization using the Givens rotations has been implemented for finding the eigenvalues of an input symmetric matrix. Figure 18.1 illustrates the number of *sweeps* vs. sum of the squares of the off-diagonal entries of the matrix. The input matrices are traditional symmetric matrices of size 20×20, 40×40, and 80×80. Based on the results, after 2 *sweeps*, the sum of squares of off-diagonal entries is close to 10^{-3}. Also, we can see that the 20×20 matrix converges faster than the other two, but at the same time, the other two matrices achieve more accuracy in their off-diagonal entries.

2. In class it was mentioned instead of Bisection (30.5), we do Parallel Jacobi (30.4).

   Since the Jacobi method deals only with pairs of rows and columns at a time, it can be implemented in parallel. As each Jacobi update consists of a row rotation that affects only rows p and q, and a column rotation that affects only columns p and q, up to $m/2$ Jacobi updates can be performed in parallel. Therefore, a sweep can be efficiently implemented by performing $m-1$ series of $m/2$ parallel updates in which each row i is paired with a different row j, for $i \neq j$.

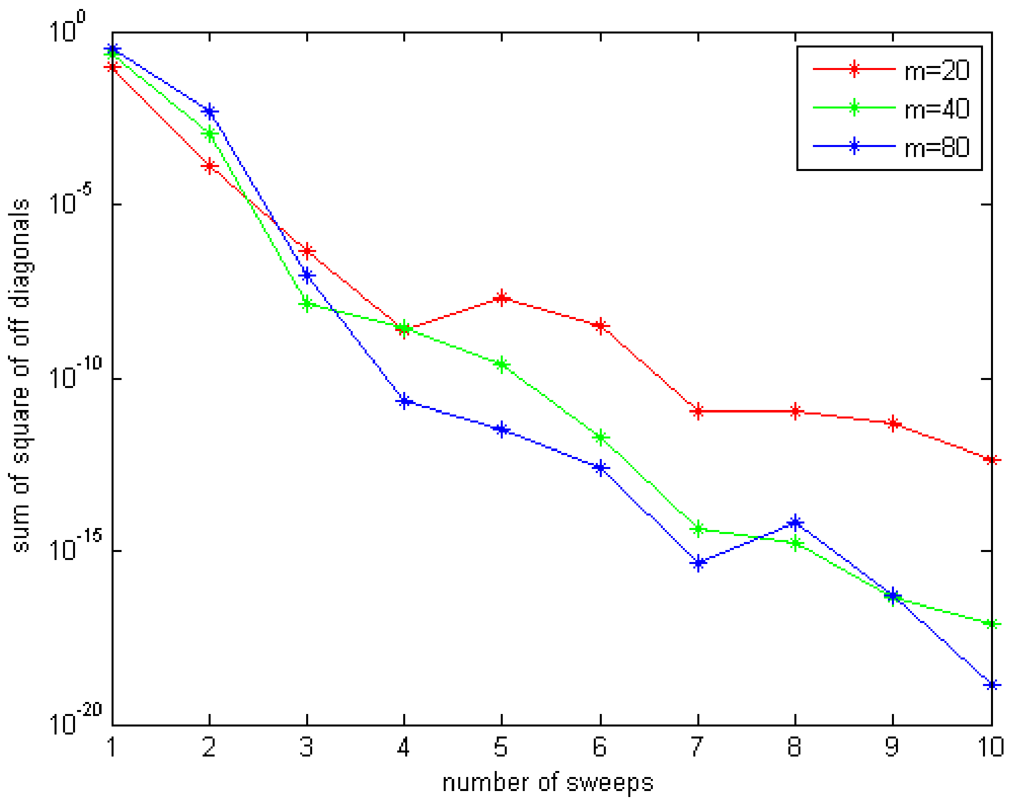


Figure 18.1: Eigenvalue Comparison between H and A matrices.

18.2 Matlab Code

```
function [D,sumsqrs] = jacobi_rotation(A,sweep)
% Implementing the QR algorithm Jacobi rotation
% Syntax:  [V,D,sumsqrs] = jacobi_rotation(A,sweep)
%
% Inputs:
%    A - Tridiagonal input matrix
%    sweep - Number of sweeps
%
% Outputs:
%    D - Diagonal of this matrix contains the Ritz valus
%    sumsqrs - Sum of square of off diagonal entries
D = A;
n = size(A,1);
V = eye(n);
progress = 0;

while (progress < sweep)
        t = sum(diag(D));
    for p = 1:(n-1),
      for q = (p+1):n,
          t = D(p,q)/(D(q,q) - D(p,p));
          c = 1/sqrt(t*t+1);
          s = c*t;
          R = [c s; -s c];
          D([p q],:) = R'*D([p q],:);
          D(:,[p q]) = D(:,[p q])*R;
      end
    end
    progress = progress + 1
    off_diag_1 = diag(D,1);
    off_diag_2 = diag(D,-1);
    off_diag = [off_diag_1 off_diag_2];
    sumsqrs(progress) = sumsqr(off_diag);

end
D = diag(diag(D));
end
```

```
c = ['-*r'; '-*g'; '-b*'];
sweep = 10;
for i=1:3
    m = 20*i;
    epsilon = 10^(-3);
    A = hilb(m); % create m by m sym matrix
    A_h = hess(A); % reduce A to triangular
    [D,sumsqrs] = jacobi_rotation(A_h,10);
    eigs = sort(diag(D));
    semilogy(1:sweep,sumsqrs,c(i,:));
    hold on
end

legend('m=20','m=40','m=80');
xlabel('number of sweeps');
ylabel('sum of square of off diagonals');
```

Lab 19

SVD Calculation

Do Exercise 31.4.

19.1 Solutions to Lab 19

In this lab, we are going to calculate the minimum singular values of $m \times m$ matrices in two ways:

1. using the standard matlab svd
2. forming A^*A and finding the square root of it's positive eigenvalues

Based on the results in Figure 19.1, the eigenvalues calculated by the standard matlab svd are much more accurate than the svd by the second method. The results conform the general discussion of these algorithms. As mentioned in the text, the algorithm that finds the eigenvalues by forming A^*A is unstable, since it reduces the svd problem to an eigenvalue problem that may be much more sensitive to perturbations.

On the other hand, as mentioned in the text, since we do the square rooting of eigenvalues, it affects the smallest singular values so bad, and we must expect a loss of accuracy of order $\kappa(A)$-a "squaring of the condition number".

We can verify this accuracy loss in Figure 19.1. This figure illustrates the condition number of the same $m \times m$ matrices. As we see, the loss of accuracy of the smallest singular values in Figure 19.1 is approximately of order square root of the condition number. Figure 19.2 illustrates the square root of the condition number. Based on this figure, as we get close to matrix dimensions 20×20 and beyond, the loss of accuracy is of order square root of condition numbers.

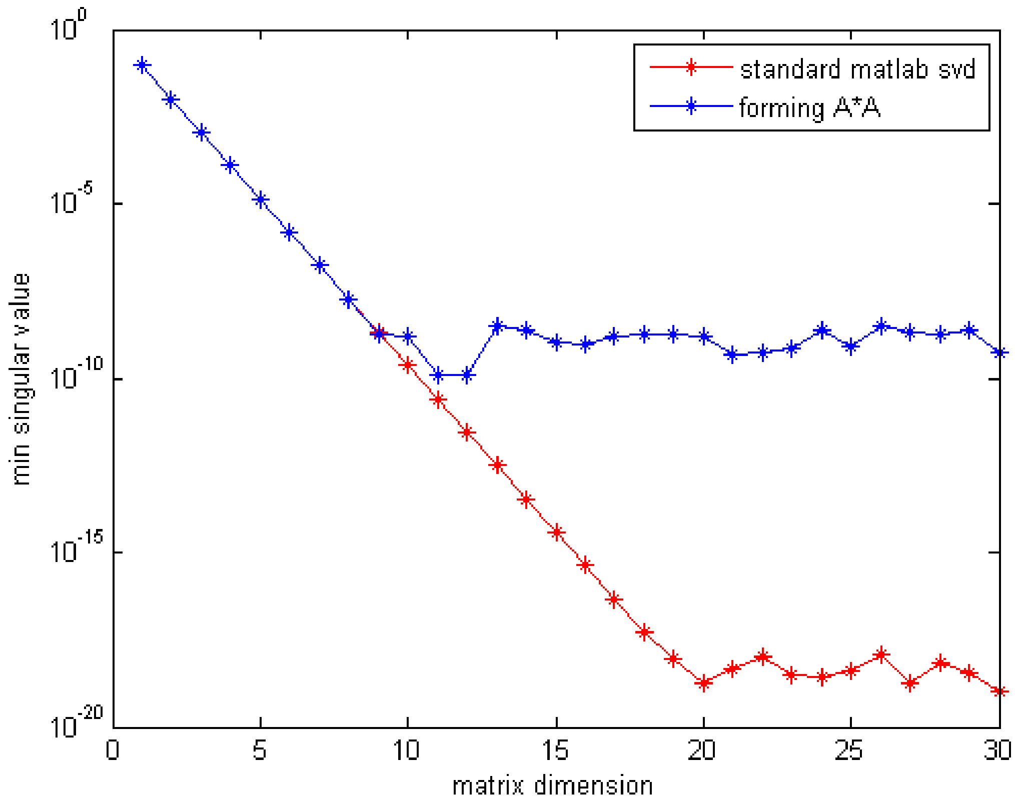


Figure 19.1: Comparing the smallest singular values of $m \times m$ matrices using two different methods: matlab svd, and eigenvalues of A^*A.

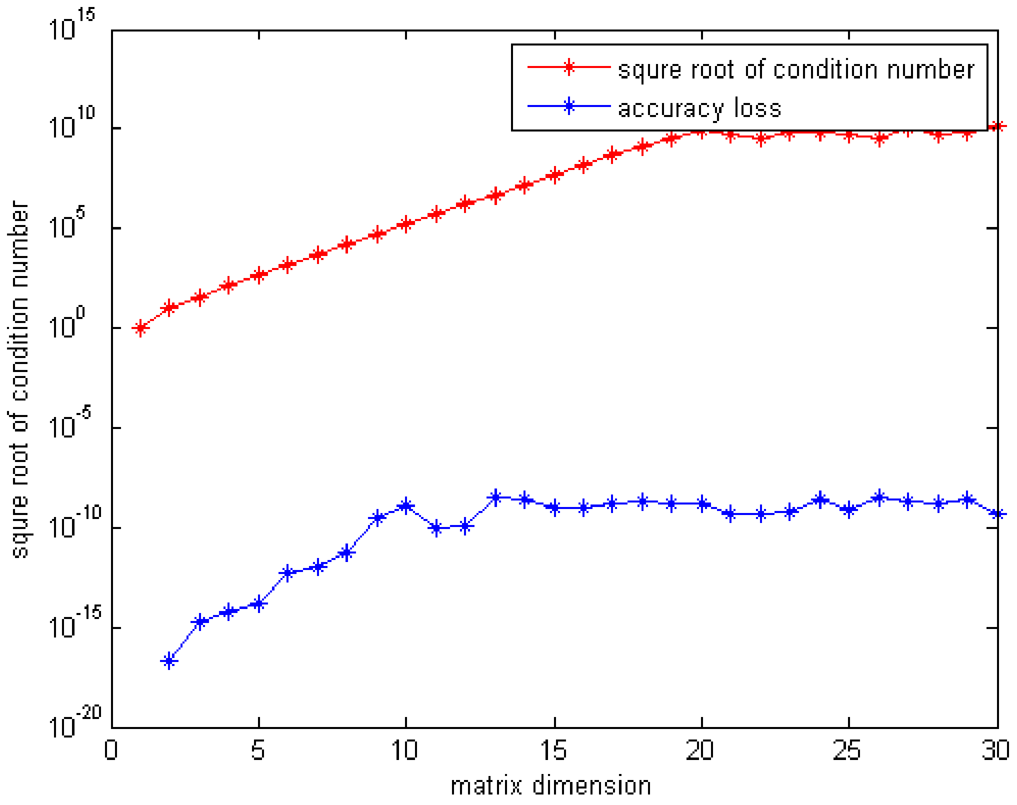


Figure 19.2: Square root of condition numbers of $m \times m$ matrices and the norm between the singular values calculated in two different ways mentioned earlier.

19.2 Matlab Code

```
k = 30;
min_s_A = zeros(1,k);
min_s_B = zeros(1,k);
cond_A = zeros(1,k);
norms = zeros(1,k);
for m=1:k
    A = ones(m);
    for i=1:m
        A(i,i) = 0.1;
    end
    A_t = triu(A);
    cond_A(m) = sqrt(cond(A_t));
    % standard SVD software
    s = svd(A_t);
    min_s_A(m) = min(s);
    % SVD by forming A*A
    B = A_t'*A_t;
    eig_B = eig(B);
    s_B = sqrt(abs(eig_B));
    min_s_B(m) = min(s_B);
    norms(m) = norm(min_s_A(m)-min_s_B(m));
end

figure
semilogy(1:k,min_s_A,'r*-',1:k,min_s_B,'b*-');
legend('standard matlab svd','forming A*A');
xlabel('matrix dimension');
ylabel('min singular value');

figure
semilogy(1:k,cond_A,'r*-',1:k,norms,'b*-');
legend('squre root of condition number','accuracy loss');
xlabel('matrix dimension');
ylabel('squre root of condition number');
```

Lab 20

Overview of Iterative and Direct

Do problem 32.2.

20.1 Solutions to Lab 20

(a) For $n \times n$ matrix multiplication, the obvious algorithm performs:

- $n^3 - n^2$ additions
- n^3 multiplications
- The regular complexity: $2n^3$

(b) For $n = 2$ matrix multiplication, the Strassen algorithm performs:

- 18 additions
- 7 multiplications
- If $n = 2^k$, the total complexity is $7n^{\log_2^7} - 6n^2$

(c) If we perform the matrix multiplication using a divide and conquer technique we do 8 multiplications for matrices of size $\frac{n}{2} \times \frac{n}{2}$ and 4 additions. Addition of two matrices takes $O(n^2)$ time, so the time complexity can be written as

$$T(n) = 8T(n/2) + O(n^2) \tag{20.1}$$

In the Strassen divide and conquer method, the main component for time complexity is 7 recursive calls. Strassens method is a divide and conquer method in the sense that this method also divide matrices to sub-matrices of size $\frac{n}{2} \times \frac{n}{2}$. Based on the fact that Strassen performs 7 recursive calls, the time complexity can be written as

$$T(n) = 7T(n/2) + O(n^2) \tag{20.2}$$

Based on the Master's theorem, the time complexity for matrices of dimension $n = 2^k$ as $n \to \infty$ is $O(n^{Log7})$.

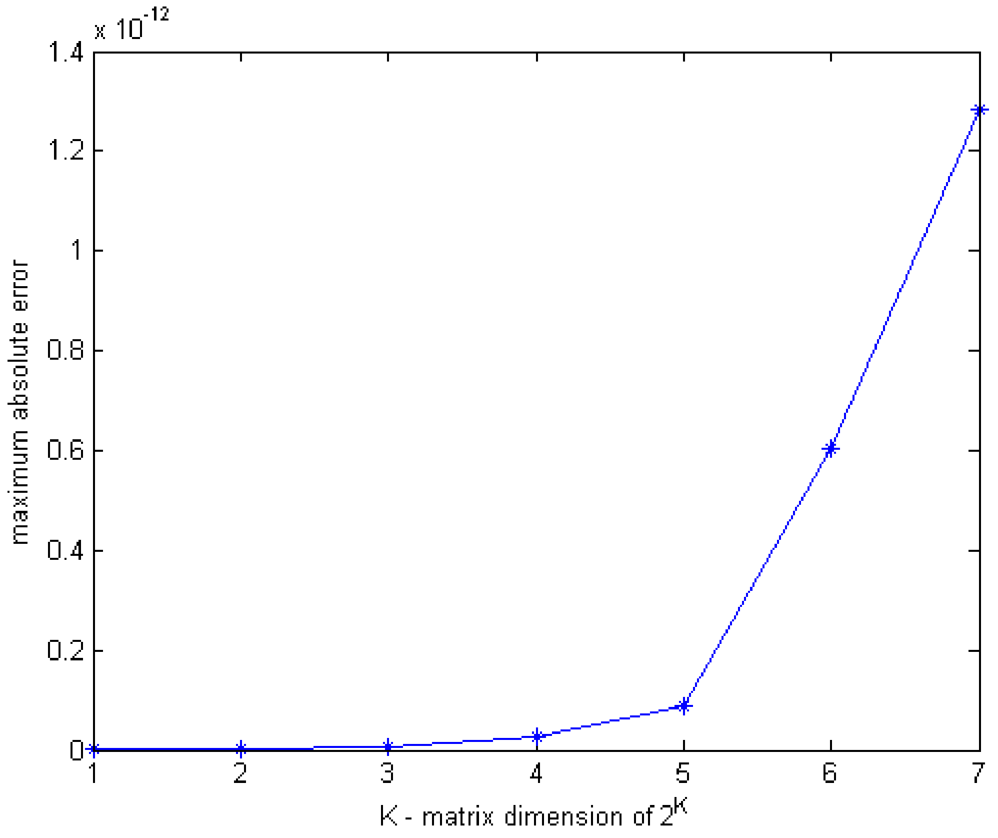


Figure 20.1: Maximum absolute difference value between the matlab multiplication and Strassen's method of two matrices with sizes $n = 2^k$, and $1 \leq k \leq 7$.

(d) Figure 20.1 illustrates the maximum absolute difference value between the matlab's multiplication result and Strassen's method result. The way we have compared the two results, is performing an absolute difference on the two multiplication results (element wise) and plotted the maximum of these differences.of two matrices with sizes $n = 2^k$, and $1 \leq k \leq 7$. Based on the results, the error is in the order of $10^{(-12)}$ in the worst case.

20.2 Matlab Code

```
function [C,num_add,num_mult] = strassen(A, B,num_add,num_mult)
% Impleents the Strassen multiplication
% Input:
%       A, B: input matrices of size 2^n
% Output:
%        C: Result of multiplication
%        num_add: Number of additions perfomed by Strassen
%        num_mult: Number of multiplications performed by Strassen

    n = length(A);
    if (n == 1)
        C = A*B;
    else
        num_mult = num_mult +7 ;
        num_add = num_add +18;
        i = 1:(n/2);
        j = ((n/2)+1):n;
        [P1,num_add,num_mult] = strassen( A(i,i)+A(j,j), B(i,i)+B(j,j),num_add,num_mult);
        [P2,num_add,num_mult] = strassen( A(j,i)+A(j,j), B(i,i), num_add,num_mult);
        [P3,num_add,num_mult] = strassen( A(i,i), B(i,j)-B(j,j), num_add,num_mult);
        [P4,num_add,num_mult] = strassen( A(j,j), B(j,i)-B(i,i), num_add,num_mult);
        [P5,num_add,num_mult] = strassen( A(i,i)+A(i,j), B(j,j), num_add,num_mult);
        [P6,num_add,num_mult] = strassen( A(j,i)-A(i,i), B(i,i)+B(i,j), num_add,num_mult);
        [P7,num_add,num_mult] = strassen( A(i,j)-A(j,j), B(j,i)+B(j,j), num_add,num_mult);
       W = P1+P4-P5+P7;
        X = P3+P5;
        Y = P2+P4;
        Z = P1+P3-P2+P6;
        C = [ W  X;
            Y  Z ];
    end
end
```

```
K = 7;
abs_diff = zeros(K,1);
num_add_init = 0;
num_mult_init = 0;
num_mults = zeros(K,1);
num_adds = zeros(K,1);

for k=1:K
    A = randn(2^k);
    B = randn(2^k);
    [C,num_add,num_mult] = strassen(A,B,num_add_init,num_mult_init);
    C_matlab = A*B;
    abs_diff(k) = max(max(abs(C-C_matlab)));
    num_mults(k) = num_mult;
    num_adds(k) = num_add;
end

figure
plot(1:K,abs_diff,'-*');
xlabel('K - matrix dimension of 2^K');
ylabel('maximum absolute error');


figure
plot(1:K,num_adds,'b*');
hold on
plot(1:K,num_mults,'g*');
legend('additions','multiplications');
xlabel('K - matrix dimension of 2^K');
ylabel('total number of operations');
```

Lab 21

Arnoldi

Implement Algorithm 33.1 (Arnoldi Iteration) as a MatLab function.

21.1 Solutions to Lab 21

In this lab, the Arnoldi algorithm has been implemented and then tested in two ways:

- comparing the eigenvalues of H and A matrices
- comparing the runtime of the Arnoldi algorithm for an sparse vs dense input A matrix

21.1.1 Eigenvalue Comparison of H and A

As we see in Figure 21.1, the norm of the eigenvalues between H and A is in the order of magnitude 10^{-15} which is expected.

21.1.2 Runtime Comparison

For 200×200 input matrix, and 6 iterations of the Arnoldi algorithm, the average runtime for the sparse matrix and dense matrices are 1.01 seconds and 1.9 seconds respectively.

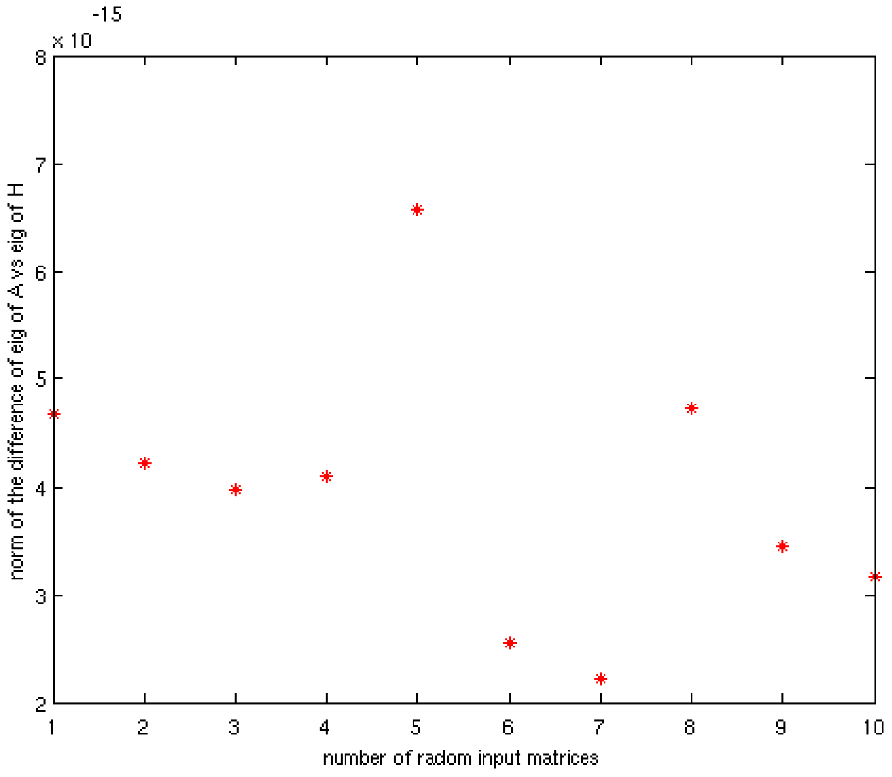


Figure 21.1: Eigenvalue Comparison between H and A matrices.

21.2 Matlab Code

```
function [Q,H] = arnoldi(A,b,N)
% Implementing the Arnoldi algorithm for finding the
% eigenvalues of a matrix (A ~= A*)
% Syntax:  [Q,H] = arnoldi(A,b,N)
%
% Inputs:
%    A - Input system matrix of size m*m
%    b - Random initial vector
%    N - Number of iterations
%
% Outputs:
%    Q - as in A = QHQ*
%    H - as in A = QHQ*

    Q = zeros(size(A,2),N+1);
    H = zeros(N+1,N);
    Q(:,1) =  b/norm(b);
    for n=1:N
        v = A*Q(:,n);
        for j=1:n
            H(j,n) = Q(:,j)'*v;
            v = v - H(j,n)*Q(:,j);
        end
        H(n+1,n)= norm(v);
        Q(:,n+1) = v/H(n+1,n);
    end
end
```

```
% Testing arnoldi with sparse matrix
n = 6;
t = 10;
for i=1:t
    A_sparse = sprandn(n,n,1);
    b = randn(n,1);
    eig_A = eigs(A_sparse);
    [Q,H] = arnoldi(A_sparse,b,n);
    eig_H = eig(H(1:end-1,:));
    plot(i,norm(sort(eig_A) - sort(eig_H)),'r*')
    hold on
end
 xlabel('number of radom input matrices');
 ylabel('norm of the difference of eig of A vs eig of H');

%% Testing arnoldi with dense matrix
% for i=1:t
%     A_dense = randn(n);
%     b = randn(n,1);
%     eig_A = eig(A_dense);
%     [Q,H] = arnoldi(A_dense,b,n);
%     eig_h = eig(H(1:n,:));
% end
```

Lab 22

Arnoldi finding Eigenvalues

Exercise 34.3

22.1 Solutions to Lab 22

(a) Figure 22.1 illustrates the singular values of the A matrix. Also, Figures 22.2, 22.3, 22.4, and 22.5 illustrate the spectrum and the boundaries of the pseudospectra of A.

(b) Figure 22.6 illustrates the convergence of the Arnoldi maximum eigenvalue estimate. After 30 iterations of Arnoldi, we achieve to the 15 digits of accuracy in the maximum eigenvalue estimate.

Another feature is apparent in Figure 22.6 is that after the initial few dozen steps, the convergence begin to accelerate, which is a phenomenon common in Krylov subspace iterations. What is happening here is that the iteration is beginning to resolve some of the other outer eigenvalues of A, near the unit circle.

(c) In this section, we have used the pseudospectra of A by $\widetilde{H_n}$. This idea has been tested by plotting the ϵ-pseudospectra of $\widetilde{H_n}$ for $n = 20$. Figures 22.7-22.10 illustrate the results. Based on the results, the plots are the same as the plots in part (a).

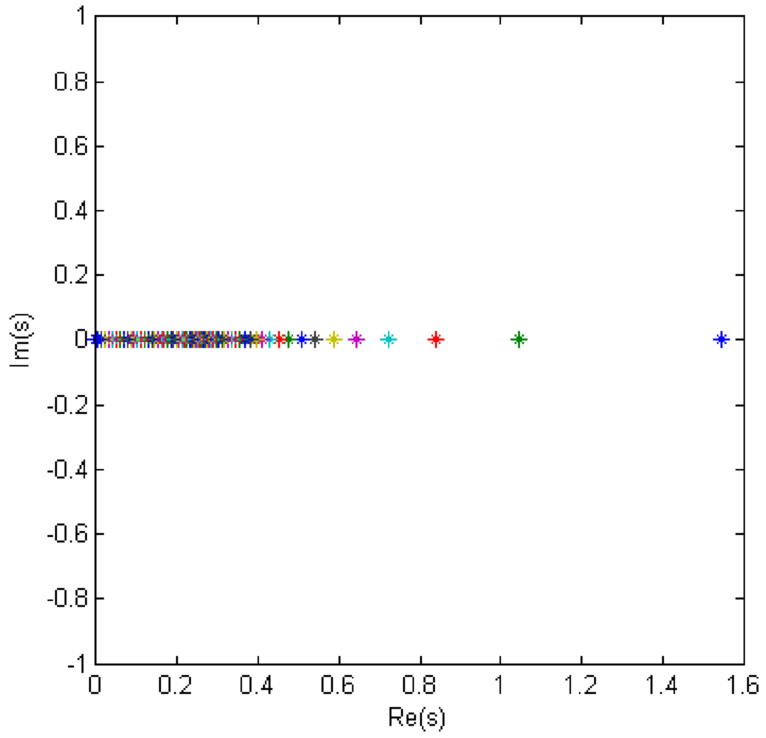


Figure 22.1: Singular values of the A matrix

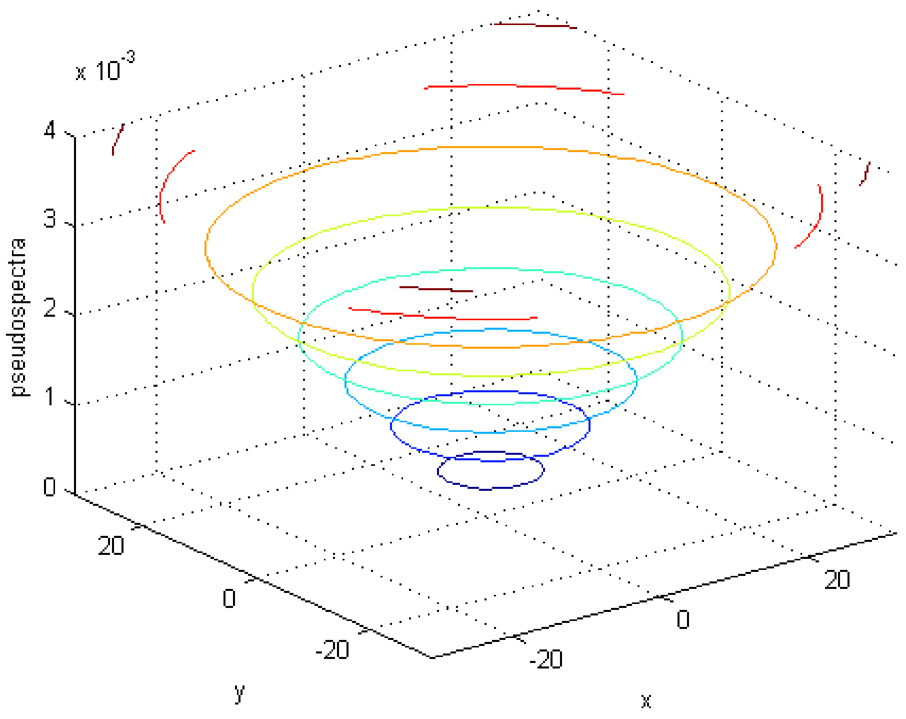


Figure 22.2: pseudospectra of A with $\epsilon = 10^{-4}$

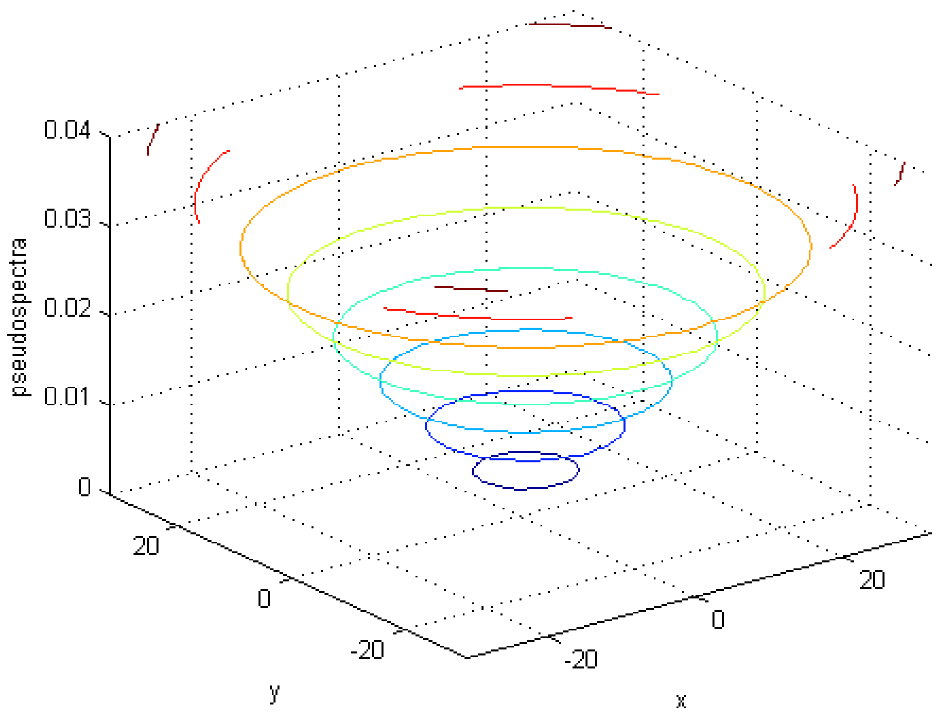


Figure 22.3: pseudospectra of A with $\epsilon = 10^{-3}$

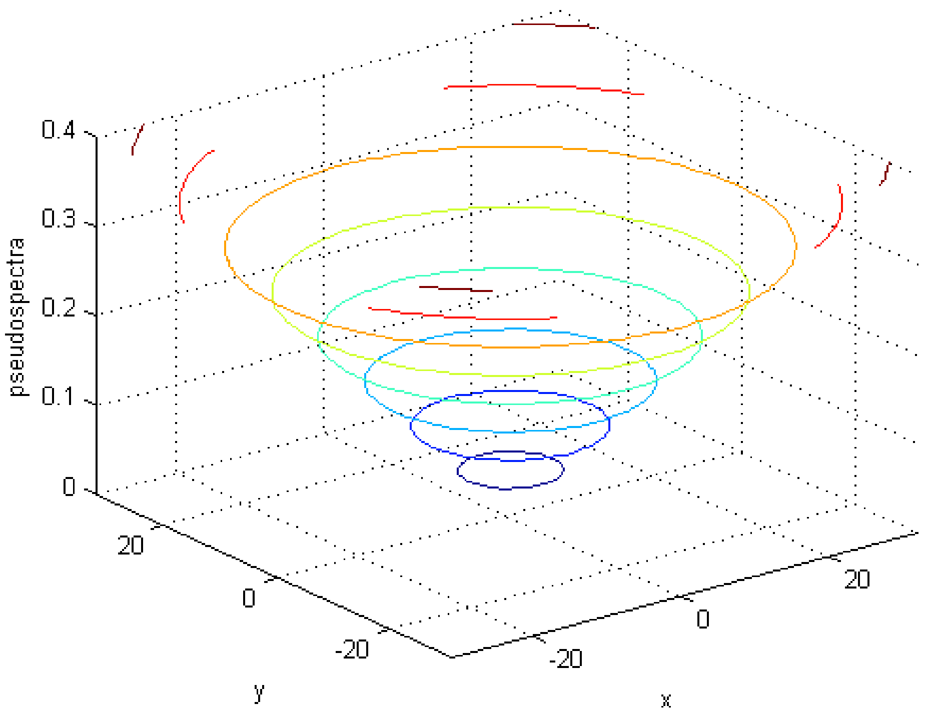


Figure 22.4: pseudospectra of A with $\epsilon = 10^{-2}$

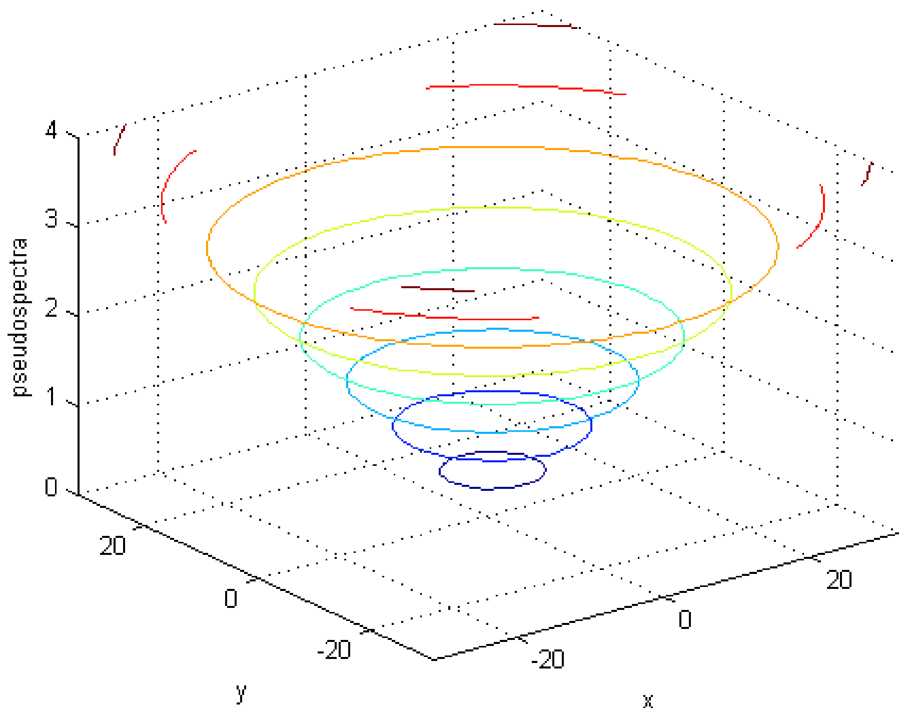


Figure 22.5: pseudospectra of A with $\epsilon = 10^{-1}$

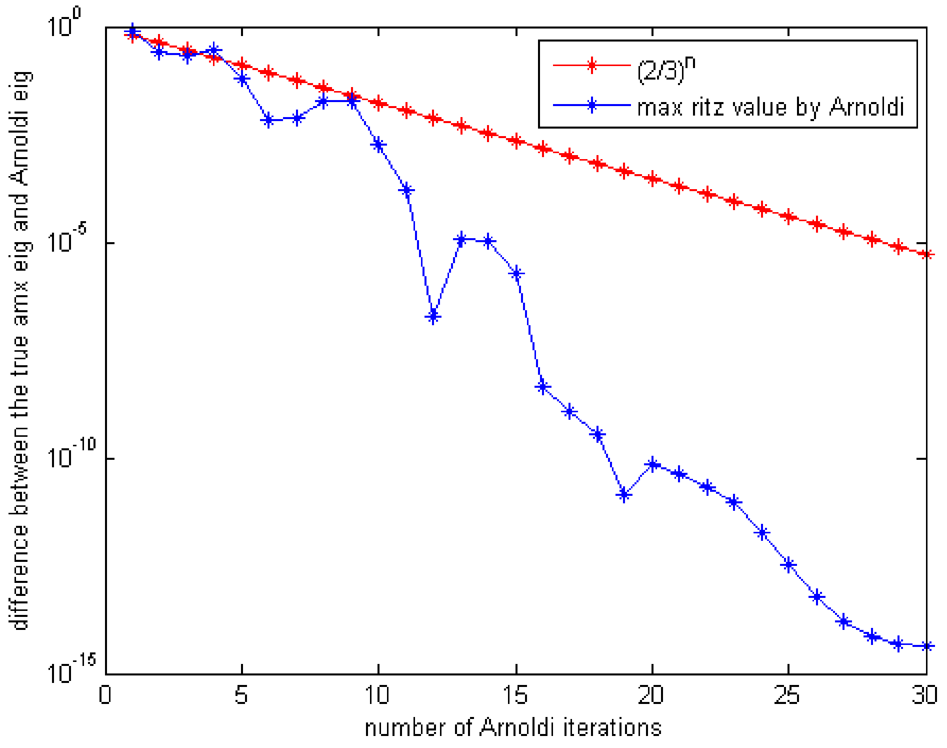


Figure 22.6: Convergence of the Arnoldi maximum eigenvalue estimate.

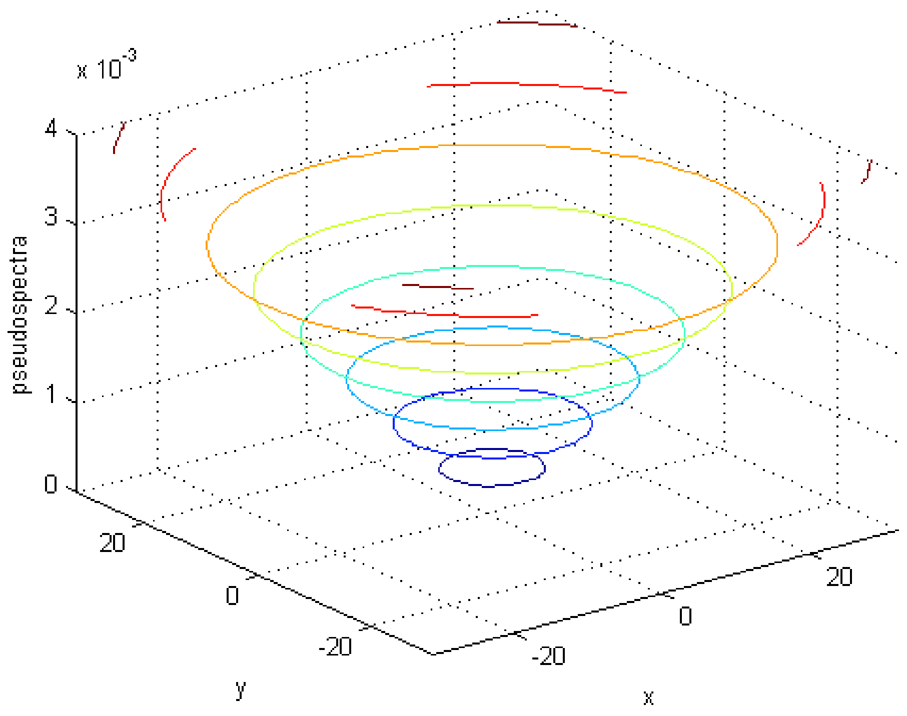


Figure 22.7: pseudospectra of $\widetilde{H_n}$ for $n = 20$ with $\epsilon = 10^{-4}$

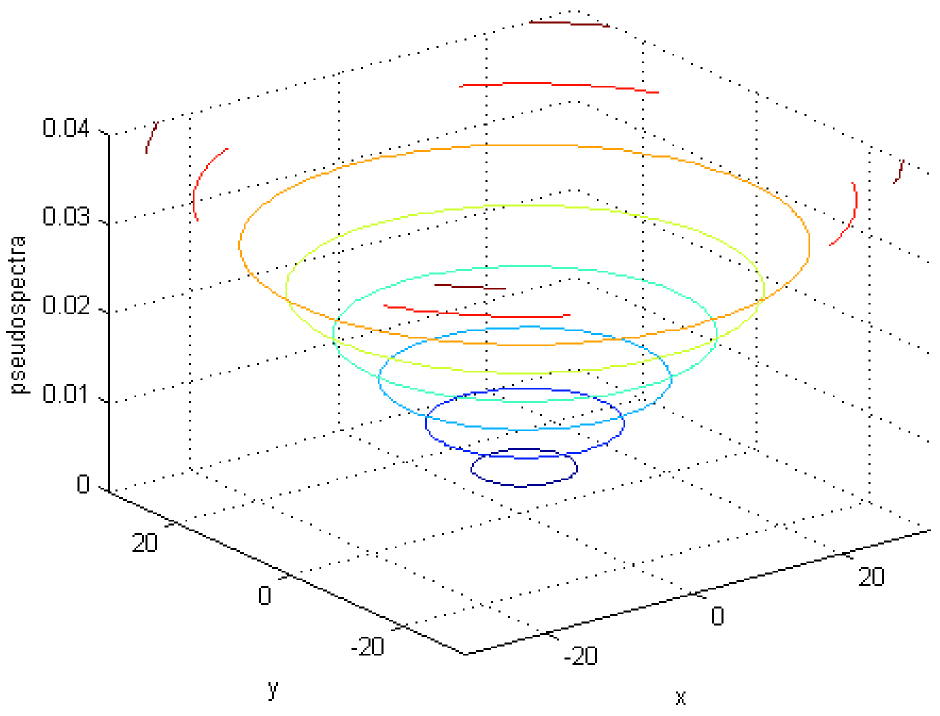


Figure 22.8: pseudospectra of $\widetilde{H_n}$ for $n = 20$ with $\epsilon = 10^{-3}$

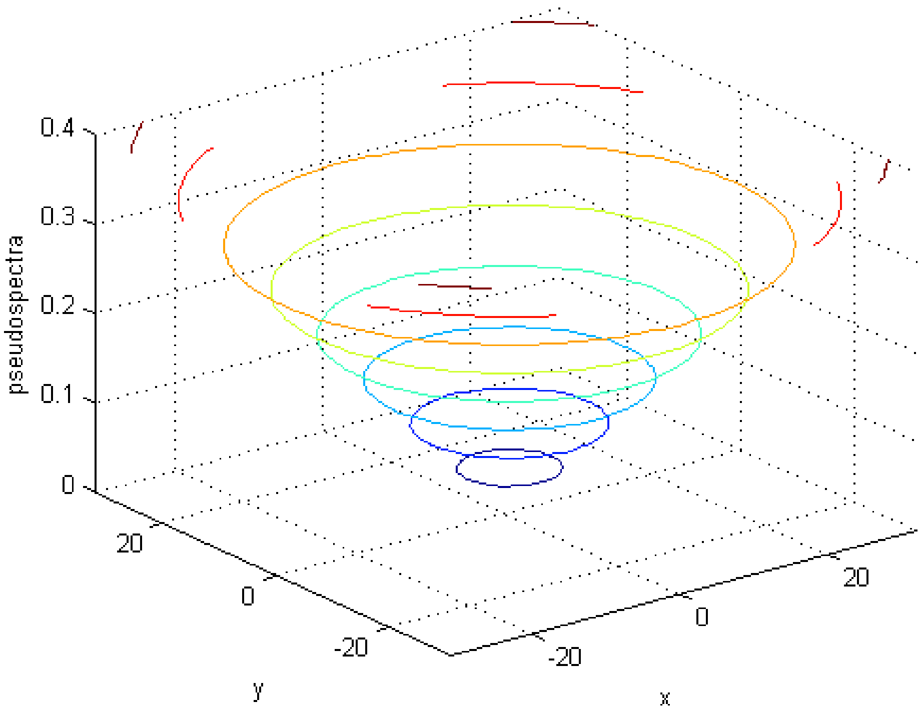


Figure 22.9: pseudospectra of $\widetilde{H_n}$ for $n = 20$ with $\epsilon = 10^{-2}$

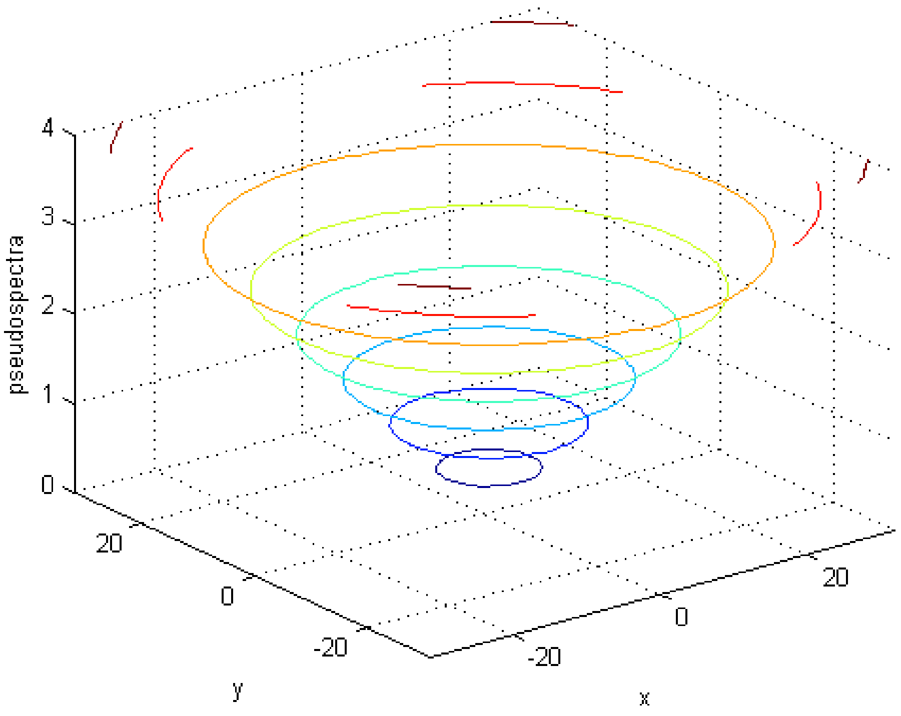


Figure 22.10: pseudospectra of $\widetilde{H_n}$ for $n = 20$ with $\epsilon = 10^{-1}$

22.2 Matlab Code

```
function [Q,H] = arnoldi(A,b,N)
% Implementing the Arnoldi algorithm for finding the
% eigenvalues of a matrix (A ~= A*)
% Syntax:  [Q,H] = arnoldi(A,b,N)
%
% Inputs:
%     A - Input system matrix of size m*m
%     b - Random initial vector
%     N - Number of iterations
%
% Outputs:
%     Q - as in A = QHQ*
%     H - as in A = QHQ*

    Q = zeros(size(A,2),N+1);
    H = zeros(N+1,N);
    Q(:,1) =  b/norm(b);
    for n=1:N
        v = A*Q(:,n);
        for j=1:n
            H(j,n) = Q(:,j)'*v;
            v = v - H(j,n)*Q(:,j);
        end
        H(n+1,n)= norm(v);
        Q(:,n+1) = v/H(n+1,n);
    end
end
```

```
% exercise 34.3

N = 64;
A = zeros(N,N);


% Construct the A matrix
for i=1:N-1
    val = i^(-1/2);
    A(i,i+1) = val;
    A(i,i) = val;
end
A(64,64) = 64^(-1/2);

% Plot for spectrum and pseudospectra of A
epsilon = zeros(4,1);
for i=1:4
    epsilon(i) = 10^(-i);
end

[u,s,v] = svd(A);
figure
plot(s,zeros(1,64),'*'); axis square


x = -31:1:32;
y = -31:1:32;
for i=1:4
    for k=1:N
        for j=1:N
            sigmin(j,k) = min(svd((x(k)+y(j)*1i)*eye(N)-A))*epsilon(i);
        end
    end
    figure
    contour3(x,y,sigmin)
    xlabel('x');
    ylabel('y');
    zlabel('pseudospectra');
end


% Convergence rate of Arnoldi
b = randn(N,1);
abs_diff_ev = zeros(30,1);
max_eig_diff = zeros(30,1);
true_eig = sort(eig(A),'descend');
```

```
for i=1:30
    [Q,H] = arnoldi(A,b,i);
    H_n = H(1:end-1,:);
    ritz = abs(eig(H_n));
    abs_diff_ev(i) = norm(true_eig(1:i) - ritz(1:i));
    max_eig_diff(i) = abs(max(true_eig) - max(ritz));
end

figure
semilogy((1:30),(2/3).^[1:30],'-*r');
hold on
semilogy(1:1:30,max_eig_diff,'*-b')
legend('(2/3)^n','max ritz value by Arnoldi');
xlabel('number of Arnoldi iterations');
ylabel('difference between the true amx eig and Arnoldi eig');


figure
plot(eig(A),'x') ; axis square
hold on
plot(ritz,'rx') ; axis square
legend('true eig(A)','ritz value by Arnoldi after 30 iterations');
xlabel('number of eig(A)');
ylabel('eig');

% Pseudospectra of A using the Arnoldi with H_n
for n=5:5:20
    [Q,H] = arnoldi(A,b,n);
    H_n = H(1:end-1,:);
    for i=1:4
        for k=1:N
            for j=1:N
                sigmin_H(j,k) = min(svd((x(k)+y(j)*1i)*eye(n)-H_n))*epsilon(i);
            end
        end
        figure
        contour3(x,y,sigmin_H);
        xlabel('x');
        ylabel('y');
        zlabel('pseudospectra');
    end
end
```

Lab 23

GMRES

Algorithm 35.1

23.1 Solutions to Lab 23

In this lab, GMRES algorithm has been implemented and then test by example 35.1. Figure 23.1 illustrates the eigenvalues of the matrix A, which is a circle with radius $\frac{1}{2}$, and center $z = 2$. Also, Figure 23.2 illustrates the GMRES convergence curve for the same matrix A. As we expect, this rapid convergence is illustrative of Krylov subspace iterations under ideal circumstances, when A is a well-behaved matrix.

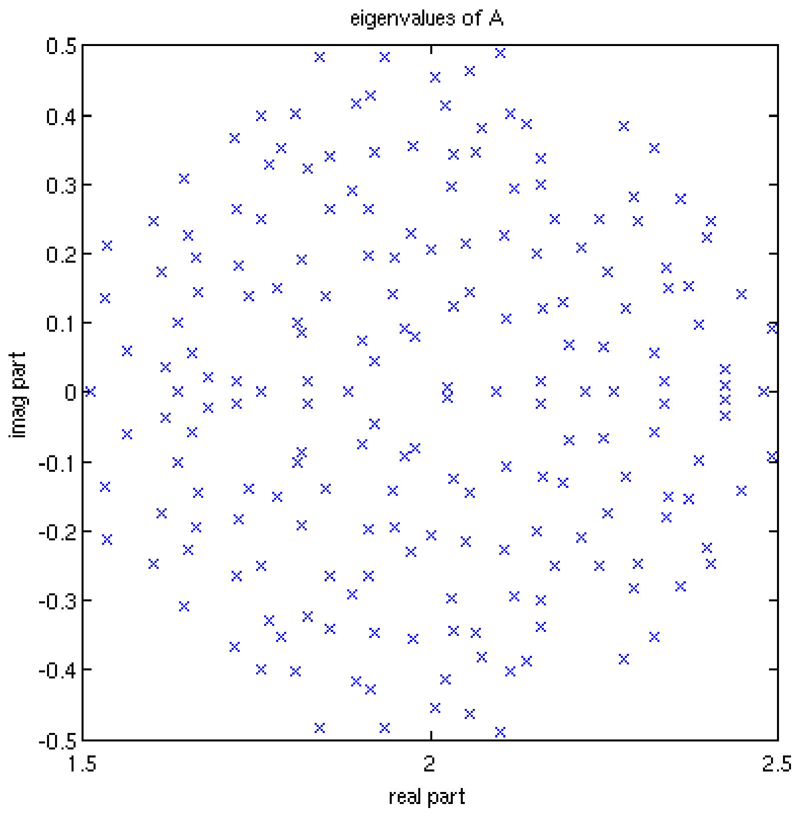


Figure 23.1: Eigenvalue Comparison between H and A matrices.

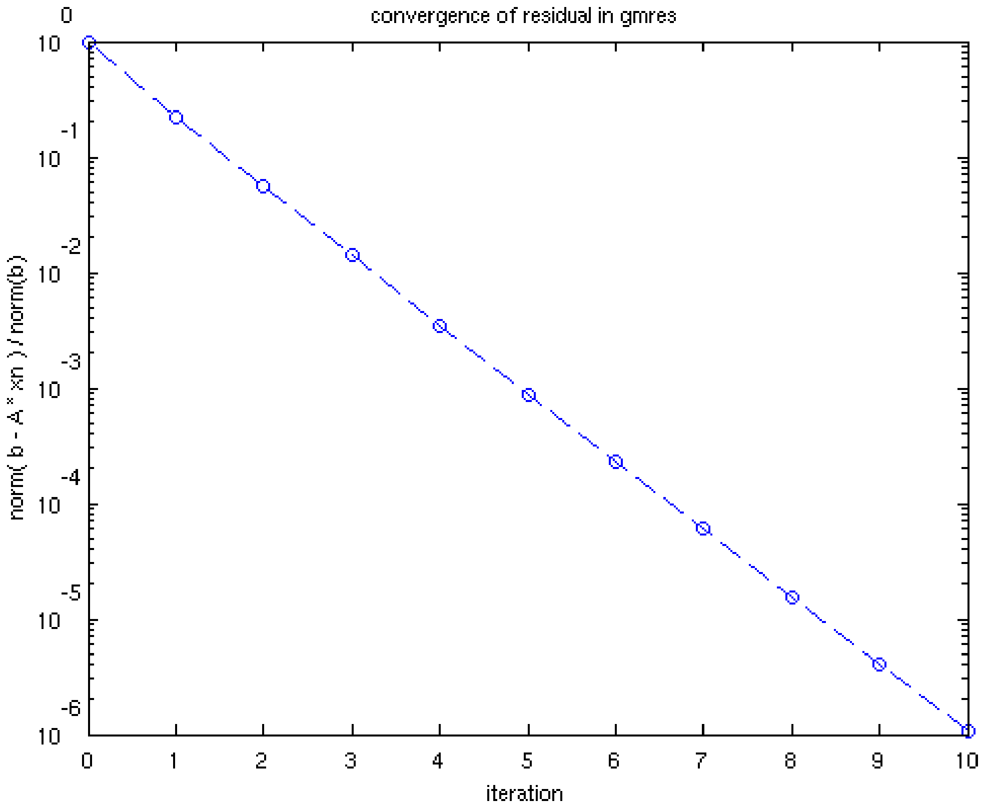


Figure 23.2: Eigenvalue Comparison between H and A matrices.

23.2 Matlab Code

```
function [x , norm_res] = gmres_alg(A,b,k)
% Implementing the GMRES algorithm for solving Ax = b
% Syntax:  [x , norm_res] = gmres_alg(A,b,k)
%
% Inputs:
%    A - Input system matrix of size m*m
%    b - Input vector such that Ax = b
%    k - Number of iterations
%
% Outputs:
%    x - Solution vector
%    norm_res - norm(b - A*x)/norm(b)

Q = [];
H = 0;
m = size(A,1);
normb = norm(b);
norm_res=normb;
Q(:,1) = b / normb;

for n = 1:k
    % Arnoldi
    v = A*Q(:,n);
    for j = 1:n
        H(j,n) = Q(:,j)'* v;
        v = v  - H(j,n)*Q(:,j);
    end
    Hn = H(1:n,1:n);
    H(n+1,n) = norm(v);
    if H(n+1,n) == 0
        break
    end
    Q(:,n+1) = v / H(n+1,n);
    e1 = [1;zeros(n,1)];
    y = H \ (normb*e1);
    norm_res = [norm_res,norm(H*y-normb*e1)];
end
x= Q(:,1:n)*y;
end
```

```
% example 35.1
m=200;
k = 10;
A=2*eye(m)+0.5*randn(m)/sqrt(m);
eig_A=eig(A);
figure
plot(eig_A,'x');
axis square
title('eigenvalues of A')
xlabel('real part')
ylabel('imag part')
xtrue=randn(m,1);
b = A*xtrue;
normb = norm(b);
A = sparse(A);
[x,norms] = gmres_alg(A,b,k);
figure
semilogy(0:k,norms/normb,'--o')
xlabel('iteration')
ylabel(' norm( b - A * xn ) / norm(b)')
title('convergence of residual in gmres')
```

Lab 24

Lanczos

Implement algorithm 36.1 (Lanczos Iteration) in MatLab.
ex 36.3 or 36.4 are good.

24.1 Solutions to Lab 24

In this lab, Lanczos algorithm has been implemented and tested by exercise 36.3.

Based on the results in Figure 24.1, after 30 iterations of Lanczos, the smallest eigenvalue reaches the 6 digits of accuracy.

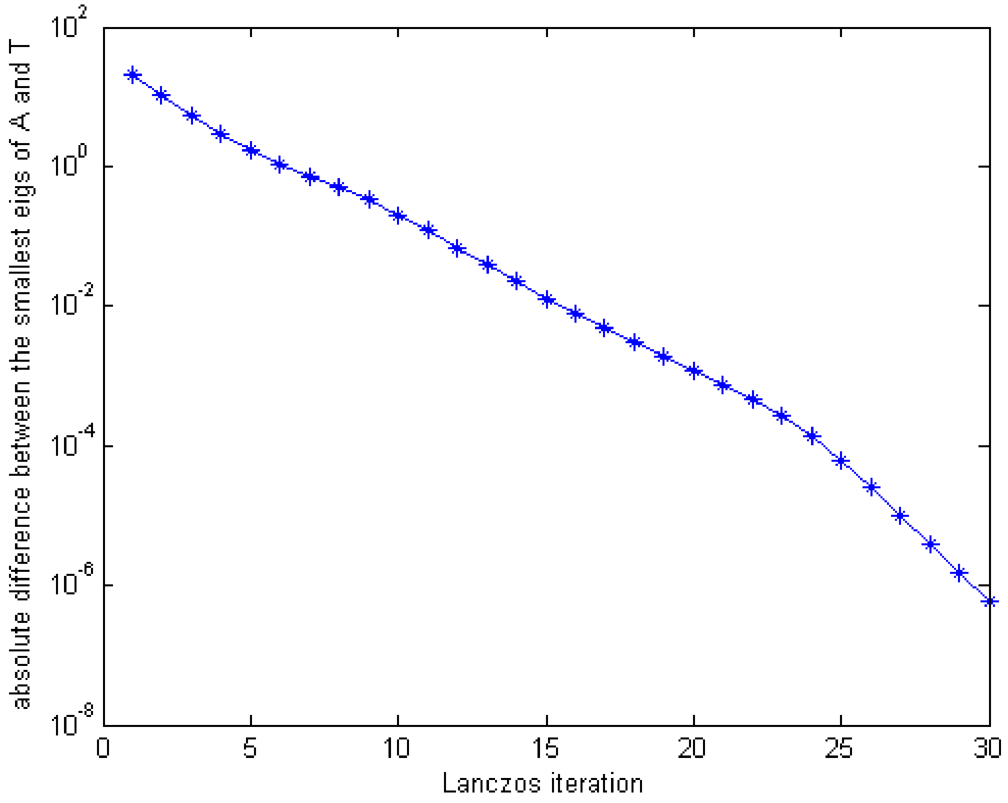


Figure 24.1: The difference between the smallest true eigenvalue of A and the smallest eigenvalue calculated by Lanczos algorithm.

24.2 Matlab Code

```
function [T,Q] = lanczos_alg(A,b,k)
% Implementation of Lanczos algorithm for findin the eigenvalues of
% a symmetric matrix
% Input
%       A - Input matrix such that A lambda = lambda x
%       b - random vector
%       k - Number of Lanczos iterations

    Q = zeros(size(A,1),k+2);
    Q(:,2) = b/norm(b);
    T = zeros(k+2,k+1);
    for n=2:k+1
        q_n = Q(:,n);
        v = A*q_n;
        alpha_n = q_n'*v;
        T(n,n) = alpha_n;
        v = v - T(n,n-1)*Q(:,n-1)-alpha_n*q_n;
        betha_n = norm(v);
        T(n+1,n) = betha_n;
        T(n,n+1) = betha_n;
        Q(:,n+1) = v/betha_n;
    end

    T = T(2:end,2:end-1);
    Q = Q(:,2:end);
end
```

```
m = 1000;
A = zeros(m,m);
for i=1:m
    for j=1:m
        if(i==j)
            A(i,j) = sqrt(i);
        elseif(abs(i-j) == 1)
            A(i,j) = 1;
        elseif(abs(i-j) == 100)
            A(i,j) = 1;
        else
            A(i,j) = 0;
        end
    end
end

A_p = sparse(A);
b = randn(size(A,1),1);
k = 30;
min_eig_A = min((eig(A))).*ones(k,1);
min_eig_T = zeros(k,1);

for n=1:k
    [T,Q] = lanczos_alg(A_p,b,n);
    T_n = T(1:n,:);
    min_eig_T(n) = min(eig(T_n));
end

figure
semilogy(1:k,abs(min_eig_T - min_eig_A),'-*b')
xlabel('Lanczos iteration');
ylabel('absolute difference between the smallest eigs of A and T');
```

Lab 25

Lanczos to Gauss Quadrature

exercise 37.4, possibly 37.1

25.1 Solutions to Lab 25

25.1.1 Exercise 37.4

Calculating the integral of $f(x) = e^x$ in the $[-1, 1]$ interval using the equally spaced or Newton-Cotes points, and where points, $n = 4$ gives the integral with three digits of accuracy which is 2.3558[1]. We expect the three digits of accuracy because if we use the Newton-Cotes points, the accuracy would be $n - 1$ which $n = 4$ in this case.

Figures 25.1 and 25.2 illustrate the results of the $I(e^x) - I_n(e^x)$ and $I(e^{|x|}) - I_n(e^{|x|})$ for $1 \leq n \leq 40$ respectively. Based on the results of Figure 25.1, the best accuracy we get is in the order of $10^{(-3)}$.

Also Figures 25.3 and 25.4 illustrate the polynomials for $n = 4$, and $n = 40$ respectively.

Figures 25.5 and 25.6 illustrate the results of the $I(e^x) - I_n(e^x)$ and $I(e^{|x|}) - I_n(e^{|x|})$ for $1 \leq n \leq 40$ respectively. In these results, the points have been spaced equally in the $[-1, 1]$ interval.

[1]The true integral is 2.35040239

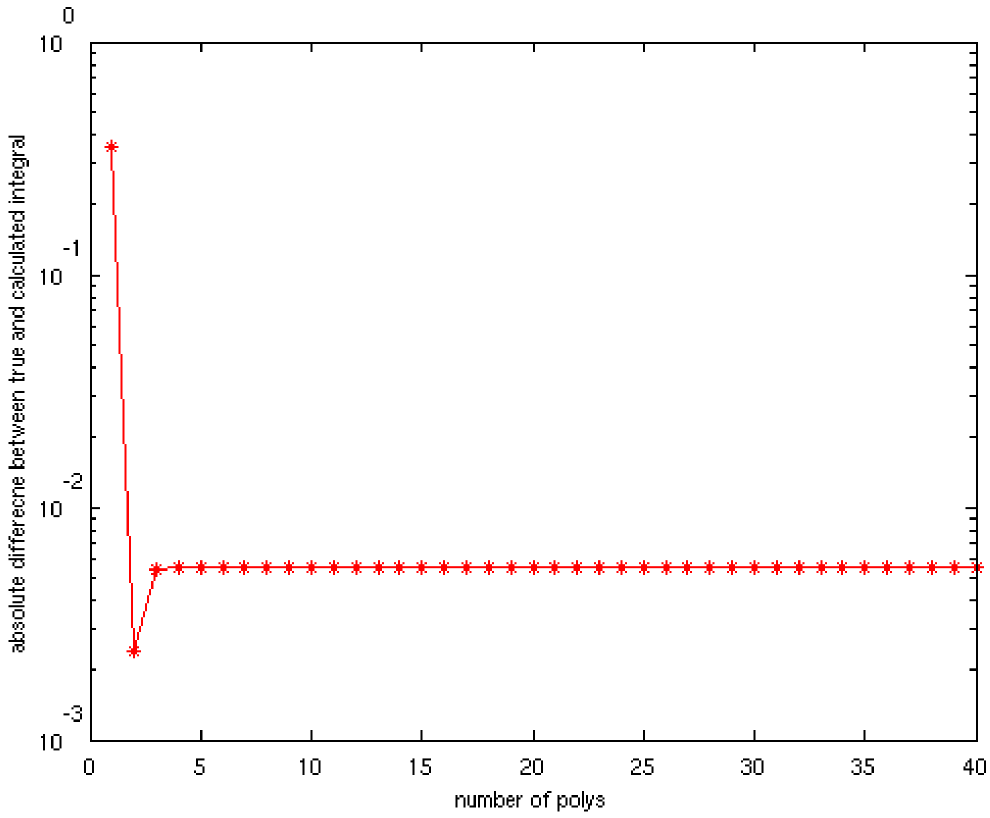


Figure 25.1: Absolute difference between the true and Gauss-Legendre integral or $I(e^x) - I_n(e^x)$ vs. $1 \leq n \leq 40$. Points are equally spaced in the $[-1, 1]$ interval.

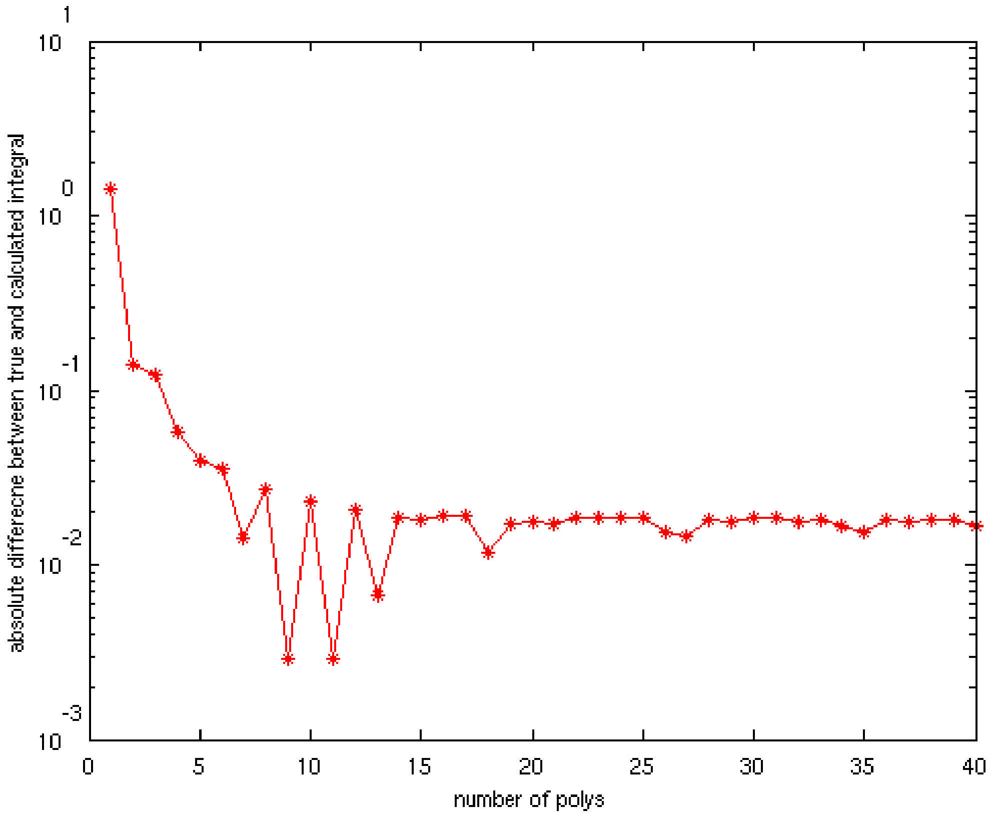


Figure 25.2: Absolute difference between the true and Gauss-Legendre integral or $I(e^{|x|}) - I_n(e^{|x|})$ vs. $1 \leq n \leq 40$. Points are equally spaced in the $[-1, 1]$ interval.

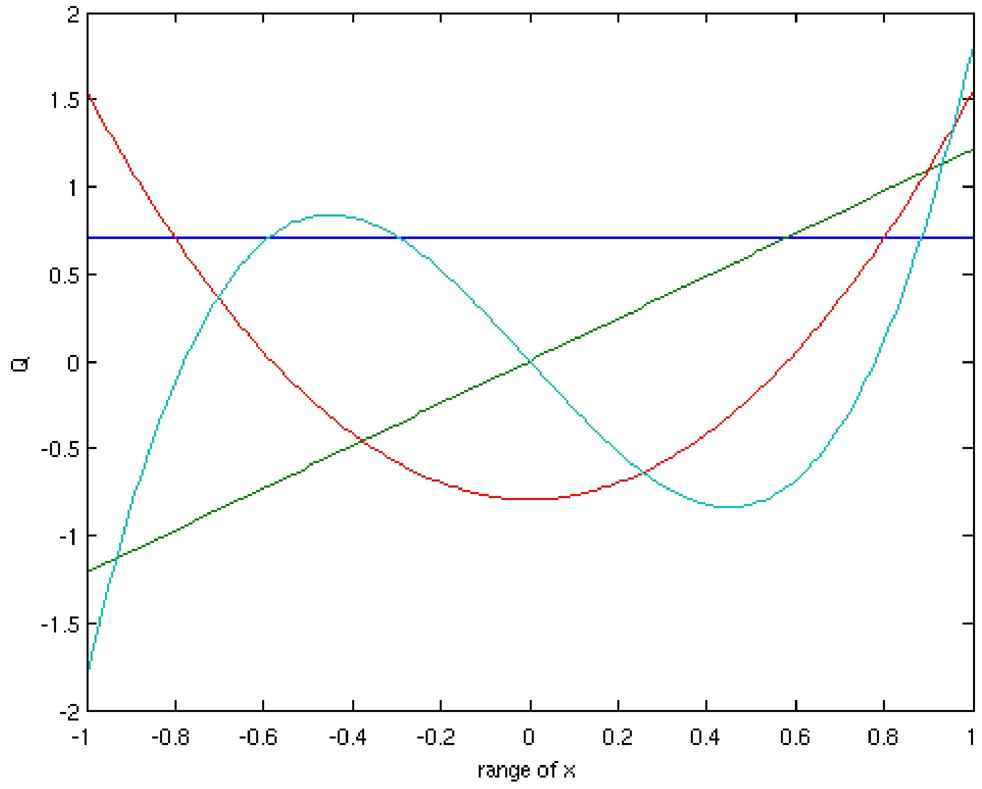


Figure 25.3: Polynomials for $n = 4$.

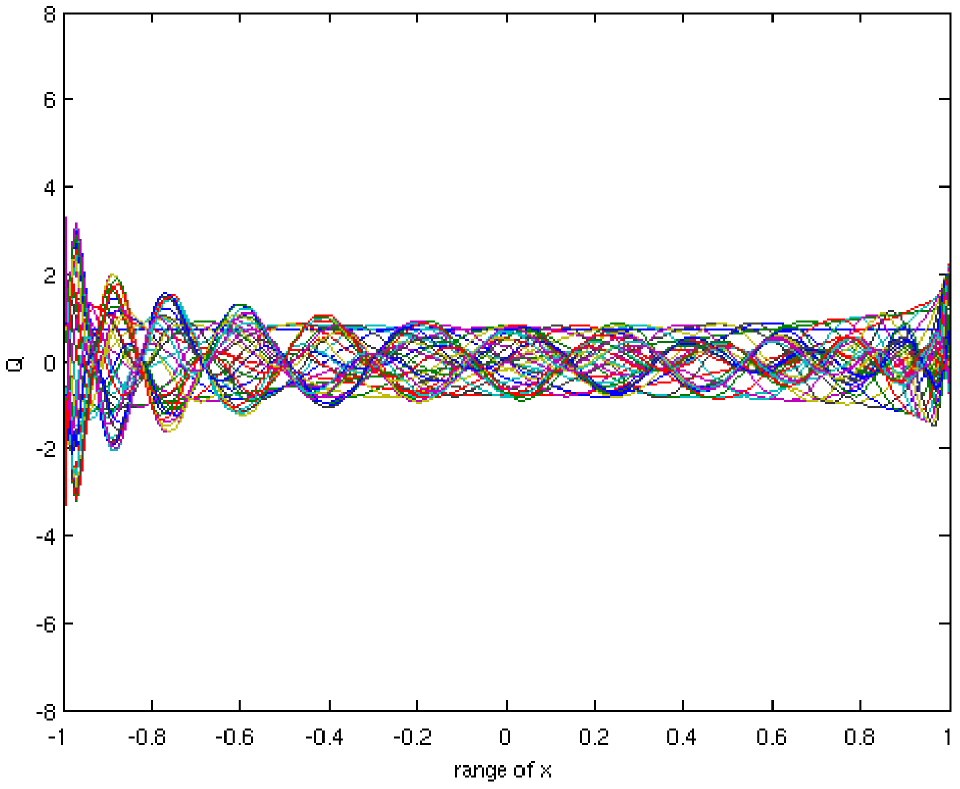


Figure 25.4: Polynomials for $n = 40$.

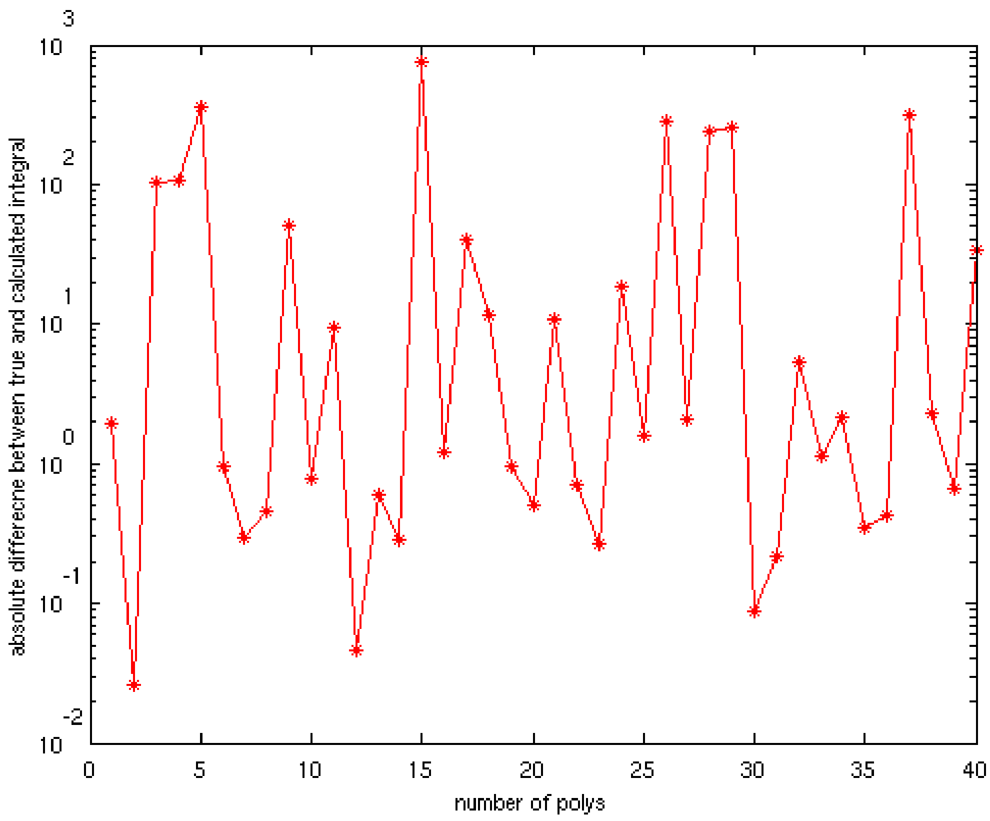


Figure 25.5: Absolute difference between the true and Gauss-Legendre integral or $I(e^x) - I_n(e^x)$ vs. $1 \leq n \leq 40$. Points are NOT equally spaced in the $[-1, 1]$ interval.

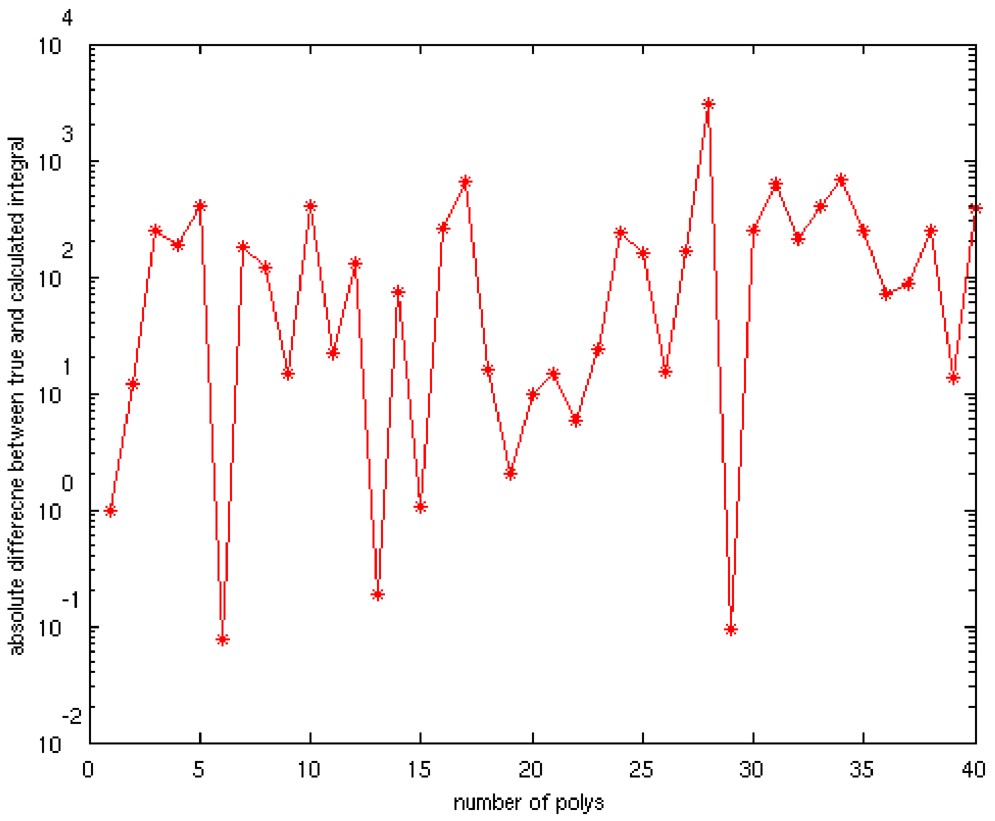


Figure 25.6: Absolute difference between the true and Gauss-Legendre integral or $I(e^{|x|}) - I_n(e^{|x|})$ vs. $1 \leq n \leq 40$. Points are NOT equally spaced in the $[-1, 1]$ interval.

25.2 Matlab Code

```
function integral_val = integral_lanczos(fun, a, b, points, polys, newton)
% Calculating the integral of a function in the [a b] range using the
% Lanczos algorithm
% Syntax:  integral_val = test_legendre_by_lanczos(fun, a, b, points)
%
% Inputs:
%    fun - Function we want to calculate its integral
%    a - Lower limit of integral
%    b - Uppert limit of integral
%    points - Number of points for the Legendre
%
% Outputs:
%    integral_val - Value of calculated integral



    [T,Q] = Legendre_by_Lanczos( points,polys, a, b, newton );

    h = b - a;
    c = (a + b)/2;
    stepsize=2/(points-1);
    if (newton == 1)
        x=(-1:stepsize:1)';
    else
        x = ((b-a).*rand(1,points) + a)';
        x(1) = -1;
        x(end) = 1;
        x = sort(x);
    end

    x_s = (h/2).*x + c;

    figure;
    plot(x_s,Q')
    [V,D]=eig(T);
    gauss_pts=diag(D)';
    gauss_wt=2*(V(1,:).^2);
    gauss_pts
    gauss_wt


    integral_val = 0;
    for i=1:polys
        integral_val = fun(gauss_pts(i)) * gauss_wt(i) + integral_val
    end

    integral_val = ((b-a)/2) * integral_val;

end
```

```
function [T,Q] = Legendre_by_Lanczos( points,polys,a, b, newton )
%[T,Q] = Legendre_by_Lanczos( points,polys )
%   Q is orthogonal columns,
%   T is coeffs for gauss quad

    h = b - a;
    c = (a + b)/2;

    beta=zeros(polys+1,1);
    alpha=zeros(polys+1,1);
    Q=zeros(points,polys+2);
    Q(:,2)=ones(points,1)./sqrt(2);
    stepsize=2/(points-1);

    if (newton == 1)
        x=(-1:stepsize:1)';
    else
        x = ((b-a).*rand(1,points) + a)';
        x(1) = -1;
        x(end) = 1;
        x = sort(x);
    end

    x_s = (h/2).*x + c;
    w=sqrt(stepsize);

    for i=2:polys+1
        v=x_s.*Q(:,i);
        alpha(i)=(Q(:,i)'*v)*w;
        v=v-beta(i-1)*Q(:,i-1)-alpha(i)*Q(:,i);
        beta(i)=norm(v)*w;
        Q(:,i+1)=v/beta(i);
    end
    Q=Q(:,2:polys+1);
    T=diag(alpha(2:polys+1))+diag(beta(2:polys),-1)+diag(beta(2:polys),1);
end
```

```
function y = myfun1(x)
    y = exp(x);
end

function y = myfun2(x)
    y = exp(abs(x));
end
```

```
% inrtegral of e^(x)
points1 = 205;
polys = 4;
a = -1;
b = 1;

integral_val_1 = integral_lanczos(@myfun1,a,b,205,polys,1);



% integral value by matlab
true_int = integral(@myfun1,a,b);
true_int2 = integral(@myfun2,a,b);

abs(q - integral_val_1)
abs(q - integral_val_2)

% plots
N = 40;
diff_1 = zeros(N,1);
for n=1:N
    int_val = integral_lanczos(@myfun1,-1,1,points1,n,0);
    diff_1(n) = abs(true_int - int_val);
end

semilogy(1:N, diff_1,'*-r')
xlabel('number of polys');
ylabel('absolute differecne between true and calculated integral');


% inrtegral of e^(|x|)

% plots
N = 40;
diff_1 = zeros(N,1);
for n=1:N
    int_val = integral_lanczos(@myfun2,-1,1,points1,n,0);
    diff_1(n) = abs(true_int2 - int_val);
end

semilogy(1:N, diff_1,'*-r')
xlabel('number of polys');
ylabel('absolute differecne between true and calculated integral');
```