

Numerical Methods, project A, Number 31

Krzysztof Rudnicki

Student number: 307585

Advisor: dr Adam Krzemieniowski

November 12, 2021

Contents

1	Problem 1 - Finding machine epsilon	4
1.1	Problem	4
1.2	Theoretical Introduction	4
1.2.1	Definition of machine epsilon	4
1.2.2	Practical applications of machine epsilon	4
1.3	Solution	6
1.3.1	Matlab code	6
1.4	Discussion of the result	6
2	Problem 2 - Solving a system of n linear equations - indicated method	8
2.1	Problem	8
2.2	Theoretical Introduction	8
2.2.1	Transform matrix into upper-triangular matrix	8
2.2.2	Backward substitution	10
2.2.3	Partial Pivoting	10
2.3	Results	11
2.4	Discussion of results	14
2.4.1	Errors in b)	17

3	Problem 3 - Solving a system of n linear equations - iterative algorithm	18
3.1	Problem	18
3.2	Theoretical introduction	19
3.2.1	Procedure	19
3.3	Results	23
3.3.1	Jacobi method result	23
3.3.2	Gauss-Seidel method result	27
3.4	Discussion of results	30
3.4.1	Comparison based on table	30
3.4.2	Convergence	30
4	Problem 4 - QR method of finding eigenvalues	34
4.1	Problem	34
4.2	Theoretical introduction	34
4.2.1	Eigenvalues	34
4.2.2	QR method for finding eigenvalues	34
4.3	Results	35
4.3.1	Starting matrix	35
4.3.2	QR method with no shifts	36
4.3.3	QR method with shifts	36
4.4	Discussion of the result	37
4.4.1	Plot	38
4.4.2	Shift method superiority	38
5	Code appendix	40
5.1	Task 2 Code	40
5.1.1	Main function	40
5.1.2	checkIfMatrixIsSquareMatrix	41
5.1.3	gaussianEliminationWithPartialPivoting	42
5.1.4	partialPivoting	42
5.1.5	partialPivotingSwapOneRow	42
5.1.6	swapRowMatrix	43
5.1.7	swapValueVector	43
5.1.8	gaussianElimination	43
5.1.9	subtractRows	44
5.1.10	backSubstitutionPhase	45
5.1.11	iterativeResidualCorrection	45

5.1.12	improveSolution	45
5.2	Task 3 code	46
5.2.1	initializeValues	46
5.2.2	decomposeMatrix	47
5.2.3	jacobiLoop	47
5.2.4	jacobiInsideLoop	48
5.2.5	jacobiEquation	48
5.2.6	gaussSeidelLoop	48
5.2.7	gaussiInsideLoop	49
5.2.8	gaussSeidelEquation	49
5.2.9	checkError	49
5.2.10	endOfLoop	50
5.2.11	dispFinalResults	50
5.2.12	plotIterations	51
5.2.13	plotIterations	52
5.3	Task 4 Code	53
5.3.1	Gram-Schmid algorithm	53
5.3.2	QRNoShifts	54
5.3.3	QRShifts	56

Chapter 1

Problem 1 - Finding machine epsilon

1.1 Problem

Write a program finding macheps in the MATLAB environment

1.2 Theoretical Introduction

1.2.1 Definition of machine epsilon

Machine epsilon is the maximal possible relative error of the floating-point representation. (Tatjewski, p.14) Machine epsilon is equal to 2^{-t} where t is number of bits in the mantissa. In our case when We use IEEE Standard 754, mantissa is 53 bits long with first bit omitted as it is always equal to '1', so We technically work with 52 bits mantissa which makes the machine epsilon equal to: $2^{-52} = 2.220446\text{e-}16$

1.2.2 Practical applications of machine epsilon

Since macheps is connected to IEEE754 standard it is always equal to the same number, which means that we can safely compare results from different machines without worrying about their individual errors. Before standards, everyone could use any way of representing floats.

Macheps is also essential when We calculate cumulation of errors of given

mathematical operation.

1.3 Solution

1.3.1 Matlab code

```
1 macheps = 1;
2 while 1.0 + macheps / 2 > 1.0
3     macheps = macheps/2;
4 end
```

Code above shifts macheps one bit to the right each iteration (by dividing by 2), it ends when we run out of mantissa bits which renders us unable to save smaller number. Due to underflow the value of macheps becomes 0 and therefore $1.0 > (\text{macheps} / 2) > 1.0$ will become false.

1.4 Discussion of the result

```
1 format long
2 disp(Display calculated macheps:)
3 disp(macheps);
4 disp(Display actual eps:)
5 disp(eps);
6 disp(Display 2^-52)
7 disp(2^-52)
8 disp(Display difference between calculated macheps and actual
   eps:)
9 disp(macheps - eps)
10 disp(Display difference between 2^-52 and actual eps:)
11 disp(2^-52 - eps) \
12 disp(Display difference between calculated macheps and 2^-52:)
13 disp(macheps - 2^-52)
```

Display calculated macheps:

2.220446049250313e-16

Display actual eps:

2.220446049250313e-16

Display 2^{-52} :

2.220446049250313e-16

Display difference between calculated macheps and actual eps:

0

Display difference between 2^{-52} and actual eps:

0

Display difference between calculated macheps and 2^{-52} :

0

As expected they are all equal to eachother. It means that our method of calculating macheps was correct.

Chapter 2

Problem 2 - Solving a system of n linear equations - indicated method

2.1 Problem

Write a program solving a system of n linear equations $Ax = b$ using the indicated method (Gaussian elimination with partial pivoting).

2.2 Theoretical Introduction

Gaussian elimination with partial pivoting consists of 3 main steps:

2.2.1 Transform matrix into upper-triangular matrix

Starting conditions

We start with the system of linear equations looking like this:

$$\begin{array}{cccccc} a_{11}x_1 & + & a_{12}x_2 & + & \dots & + & a_{1n}x_n & = & b_1, \\ a_{21}x_1 & + & a_{22}x_2 & + & \dots & + & a_{2n}x_n & = & b_2, \\ \vdots & & \vdots & & & & \vdots & & \vdots \\ a_{n1}x_1 & + & a_{n2}x_2 & + & \dots & + & a_{nn}x_n & = & b_n. \end{array}$$

In order for this method to work all the elements of **diagonal** line:

$$a_{11}, a_{22}, \dots, a_{nn}$$

Must be different from zero since We will be dividing by them.

We will denote rows as ' w_i ' where 'i' is number of the row.

Zeroing first column

We start transforming the system by **zeroing** elements in first column excluding first row element. We do it by multiplying first row by l_{i1} , where:

$$l_{i1} = \frac{a_{i1}^{(1)}}{a_{11}^{(1)}}$$

And then subtracting what We got ($l_{i1}w_1$), from i row.

Doing so We obtain a system of linear equations:

$$\begin{array}{ccccccc} a_{11}x_1 & + & a_{12}x_2 & + & \dots & + & a_{1n}x_n & = & b_1, \\ 0 & + & (a_{22} - a_{12}l_{21})x_2 & + & \dots & + & (a_{2n} - a_{1n}l_{21})x_n & = & b_2 - b_1l_{21}, \\ \vdots & & \vdots & & & & \vdots & & \vdots \\ 0 & + & (a_{n2} - a_{12}l_{n1})x_2 & + & \dots & + & (a_{nn} - a_{1n}l_{n1})x_n & = & b_n - b_1l_{n1}. \end{array}$$

Zeroing second column

We continue onto the second column, this time We will zero all elements except first and second rows. Row multiplier becomes:

$$l_{i2} = \frac{a_{i2}^{(2)}}{a_{22}^{(2)}}$$

Where:

$$a_{22}^{(2)} = (a_{22} - a_{12}l_{21})$$

And:

$$a_{i2}^{(2)} = (a_{i2} - a_{12}l_{i1})$$

They are modified values obtained from previous step. We continue as in the first step and We end up with:

$$\begin{array}{ccccccccc}
a_{11}x_1 & + & a_{12}x_2 & + & \dots & + & a_{1n}x_n & = & b_1, \\
0 & + & a_{22}^{(2)}x_2 & + & \dots & + & a_{2n}^{(2)}x_n & = & b_2^{(2)}, \\
\vdots & & \vdots & & & & \vdots & & \vdots \\
0 & + & 0 & + & \dots & + & a_{nn}^{(3)}x_n & = & b_2^{(3)},
\end{array}$$

Zeroing next columns

We repeat this process $n - 1$ times and We end up with upper triangular matrix:

$$\begin{array}{ccccccccc}
a_{11}x_1 & + & a_{12}x_2 & + & \dots & + & a_{1n}x_n & = & b_1, \\
0 & + & a_{22}^{(2)}x_2 & + & \dots & + & a_{i2}^{(2)}x_n & = & b_2^{(2)}, \\
\vdots & & \vdots & & & & \vdots & & \vdots \\
0 & + & 0 & + & \dots & + & a_{nn}^{(n)}x_n & = & b_2^{(n)},
\end{array}$$

2.2.2 Backward substitution

After transforming the system We solve the system from last to first. First We calculate value of last element:

$$x_n = \frac{b_n}{a_{nn}}$$

Then one **above**:

$$x_{n-1} = \frac{b_{n-1} - a_{n-1,n}x_n}{a_{n-1,n-1}}$$

And so on, for x_k :

$$x_k = \frac{b_k - \sum_{j=k+1}^n a_{kj}x_j}{a_{kk}}$$

2.2.3 Partial Pivoting

Gaussian elimination method has one flaw, where it can come into halt if:

$$a_{kk}^{(k)} = 0$$

To avoid it We use method of pivoting, in our case We will use partial pivoting method. We do it before each Gaussian elimination step since this will lead

to smaller error.

We first find a row i such that:

$$|a_{ik}^k| = \max_j \{|a_{kk}^k|, |a_{k+1,k}^k|, \dots, |a_{nk}^k|\}$$

Then We swap this row with k -th row. Since the matrix We use is assumed to be nonsingular then $|a_{ik}^k| \neq 0$ will be always true. After that We continue with the Gaussian elimination method.

Let's compare full pivoting to partial pivoting. Full pivoting carries more computational load. It comes from the fact that: In full pivoting we need to compare absolute values of the matrix elements. (Which gives us $k^2 - 1$ comparisons every step as opposed to $k - 1$ for partial pivoting). We also have column interchanges and corresponding interchanges in the order of elements of \mathbf{x} . Therefore when it comes to speed, partial pivoting is faster than full pivoting.

2.3 Results

Solutions vectors for matrix A and vector A and $n = 10$ Without residual correction:

$$x_{algorithm} = \begin{pmatrix} -0.930024655110760 \\ -1.223407298665613 \\ -1.273530574219411 \\ -1.230517757325955 \\ -1.151356031091789 \\ -1.056883669282743 \\ -0.952628310089775 \\ -0.834334594319914 \\ -0.683708806203301 \\ -0.450125157623323 \end{pmatrix}$$

With residual correction:

$$x_{algorithm} = \begin{pmatrix} -0.930024655110760 \\ -1.223407298665613 \\ -1.273530574219411 \\ -1.230517757325955 \\ -1.151356031091788 \\ -1.056883669282743 \\ -0.952628310089775 \\ -0.834334594319914 \\ -0.683708806203301 \\ -0.450125157623323 \end{pmatrix}$$

Matlab method:

$$x_{MatlabMethod} = \begin{pmatrix} -0.930024655110760 \\ -1.223407298665612 \\ -1.273530574219411 \\ -1.230517757325956 \\ -1.151356031091789 \\ -1.056883669282743 \\ -0.952628310089775 \\ -0.834334594319914 \\ -0.683708806203301 \\ -0.450125157623323 \end{pmatrix}$$

Error for 'A' system of equations for algorithm before residual correction:

$$1.986027322597818e - 15$$

Error for 'A' system of equations for algorithm after residual correction:

$$1.986027322597818e - 15$$

Error for 'A' system of equations for matlab method:

$$3.383918772654241$$

Solutions vectors for matrix B and vector B and n = 10 Without residual correction:

$$x_{algorithm} = 10^{14} * \begin{pmatrix} -0.000050600471710 \\ 0.001764889142984 \\ -0.022358533990003 \\ 0.142964542843099 \\ -0.526425616773059 \\ 1.184552423606169 \\ -1.655429692441864 \\ 1.402099618098568 \\ -0.659000958954395 \\ 0.131884594651908 \end{pmatrix}$$

With residual correction:

$$x_{algorithm} = 10^{14} * \begin{pmatrix} -0.000050600463347 \\ 0.001764888849422 \\ -0.022358530253741 \\ 0.142964518868662 \\ -0.526425528251739 \\ 1.184552223980917 \\ -1.655429412965713 \\ 1.402099381042299 \\ -0.659000847398248 \\ 0.131884572303072 \end{pmatrix}$$

Matlab method:

$$x_{MatlabMethod} = 10^{14} * \begin{pmatrix} -0.000050613652388 \\ 0.001765340333555 \\ -0.022364164880105 \\ 0.143000107689097 \\ -0.526555232536911 \\ 1.184841538613606 \\ -1.655830695338199 \\ 1.402437027506887 \\ -0.659158629093553 \\ 0.131915987214953 \end{pmatrix} =$$

Error for 'B' system of equations for algorithm before residual correction:

$$3.775702543583306e - 04$$

Error for 'B' system of equations for algorithm after residual correction:

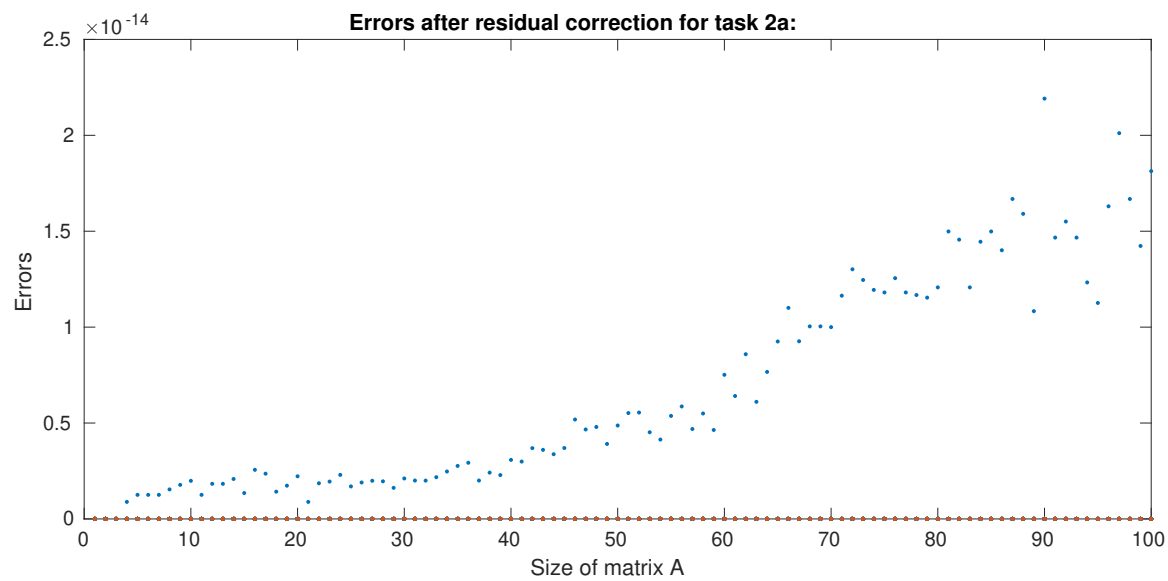
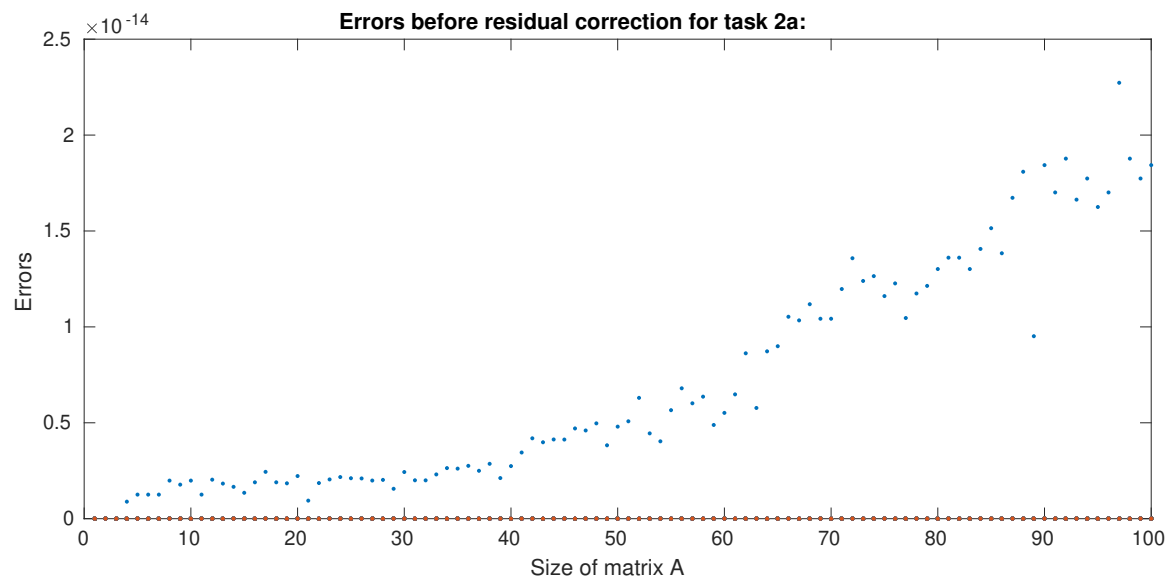
$$7.395459186003887e - 04$$

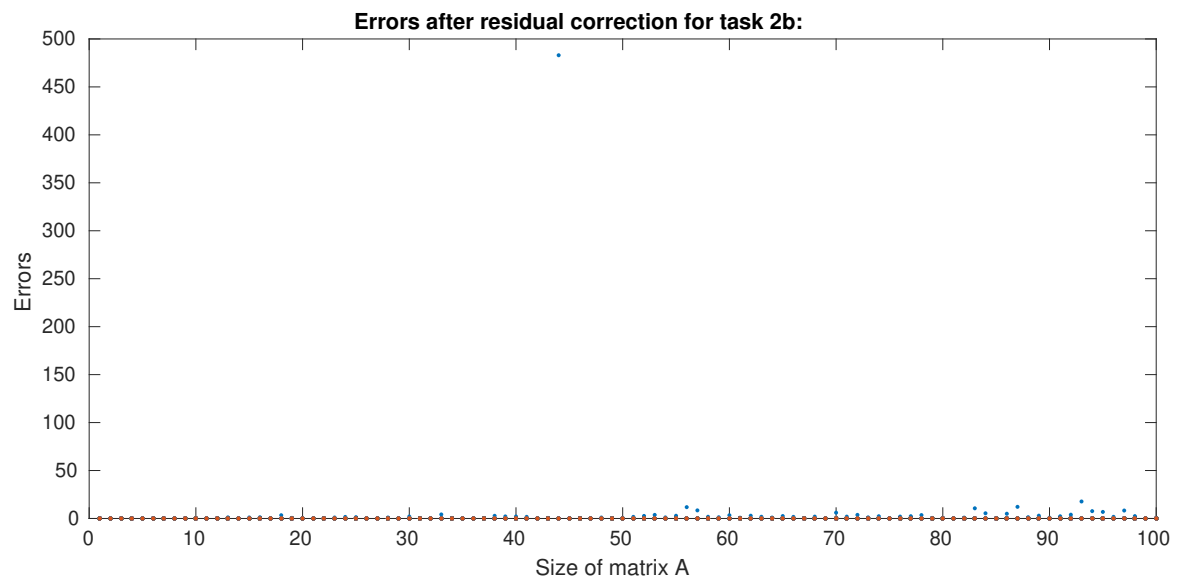
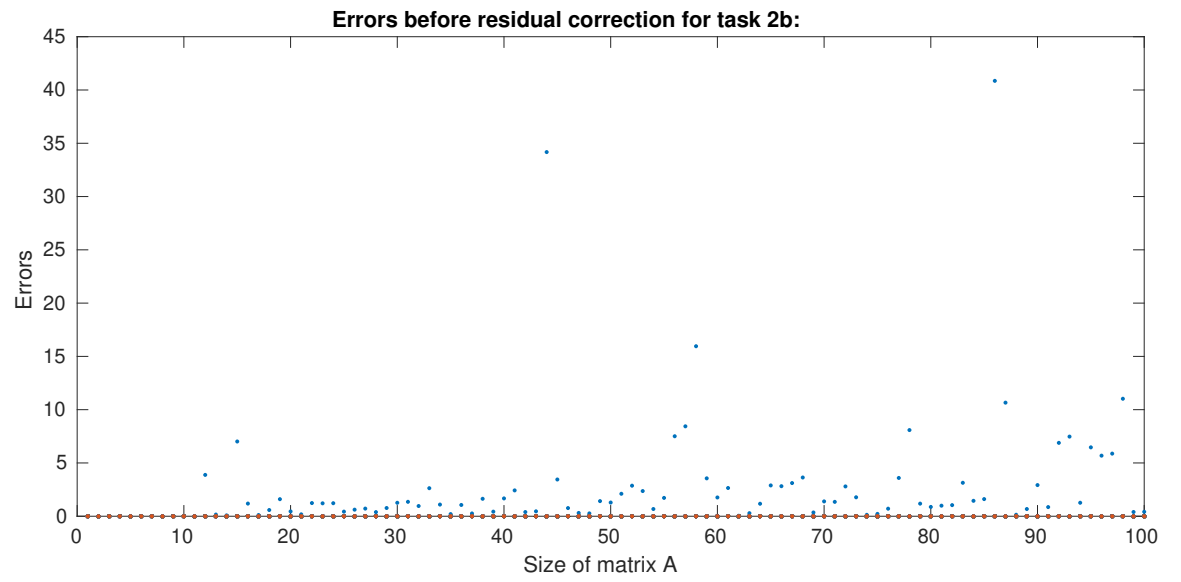
Error for 'B' system of equations for matlab method:

$$2.611906929269057e - 04$$

2.4 Discussion of results

As we can see error for 'B' system of equations increased after our residual correction. But that's just one matrix and vector pair, after testing more pairs we get following graphs (Tested on systems of equations from a) and b) respectively, with maximum size of matrices and vectors equal to 100):





As we can see our residual correction method **does** work for Matrix B and Vector B and decreases the error drastically.

2.4.1 Errors in b)

Matrix B suffers from relatively huge errors. Where do they come from? Matrix B is a variant of Hilbert matrix. Equation for generating elements of this matrix is as follows:

$$b_{ij} = \frac{5}{8(i+j+1)}$$

Equation for generating elements of the Hilbert matrix is as follows:

$$b_{ij} = \frac{1}{(i+j-1)}$$

The only thing that changes are the constants, we have 5 instead of 1 in the numerator and 8 in front of the summation in the denominator and +1 instead of -1. For sufficiently huge 'i' and 'j' those differences hardly matter. Hilbert matrix is badly conditioned. It's condition number for n size hilbert matrix is equal to:

$$\mathcal{O}\left(\frac{e^{3.5255n}}{\sqrt{n}}\right)$$

Let's calculate condition number of matrix B in matlab:

```
1 cond(matrixB(10))
2 ans = 2.428027097978043e+14
```

And compare it with condition number of matrix A:

```
1 cond(matrixA(10))
2 ans = 4.550344127923193
```

As we can see the difference is huge, condition number of matrix B for 10 elements is 10^{14} times bigger than for matrix A. This explains big errors for matrix B.

Chapter 3

Problem 3 - Solving a system of n linear equations - iterative algorithm

3.1 Problem

Write a general program for solving the system of n linear equations $Ax = b$ using the Gauss-Seidel **and** Jacobi iterative algorithms.

We are given following system:

$$\begin{cases} 10x_1 - 4x_2 + x_3 + 2x_4 = -8 \\ 2x_1 - 6x_2 + 3x_3 - x_4 = -12 \\ x_1 + 4x_2 - 12x_3 + x_4 = 4 \\ 2x_1 + 3x_2 - 3x_3 - 10x_4 = 1 \end{cases}$$

Then We need to compare the results of iterations plotting norm of the solution error versus the iteration number **k**, untill We get accuracy better than 10^{-10} .

We should also try to solve the equations from problem 2a) and 2b) for $n = 10$ using iterative method of our choice.

3.2 Theoretical introduction

We should also answer the question what happens if the sufficient condition is not fulfilled.

Iterative methods differ from the Gauss elimination method since they are iterative, which means that our solution will improve with each iteration. Building on that we can conclude that the number of iterations will depend on what accuracy we want to achieve. Since we are using iterative method we don't have the guarantee of how many iterations will be needed before we reach the solution,

In general: We start with: $x^{(0)}$ - being the best known approximation of the solution point

And we generate next vectors x^{i+1} in such way:

$$\mathbf{x}^{i+1} = \mathbf{M}\mathbf{x}^{(i)} + \mathbf{w}$$

Where \mathbf{M} is some matrix.

3.2.1 Procedure

Decomposing matrix

For both Jacobi and Gauss-Seidel method we first decompose starting matrix \mathbf{A} to:

$$\mathbf{A} = \mathbf{L} + \mathbf{D} + \mathbf{U}$$

where: \mathbf{L} - Subdiagonal matrix \mathbf{D} - Diagonal matrix \mathbf{U} - Matrix with entries over the diagonal.

For example: For:

$$\mathbf{A} = \begin{bmatrix} 2 & 3 & 3 \\ 1 & 2 & 3 \\ 1 & 1 & 2 \end{bmatrix}$$

We can get:

$$\mathbf{L} = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \end{bmatrix}$$

$$\mathbf{D} = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{bmatrix}$$

$$\mathbf{U} = \begin{bmatrix} 0 & 3 & 3 \\ 0 & 0 & 3 \\ 0 & 0 & 0 \end{bmatrix}$$

so:

$$\mathbf{A} = \mathbf{L} + \mathbf{D} + \mathbf{U}$$

$$\begin{bmatrix} 2 & 3 & 3 \\ 1 & 2 & 3 \\ 1 & 1 & 2 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \end{bmatrix} + \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{bmatrix} + \begin{bmatrix} 0 & 3 & 3 \\ 0 & 0 & 3 \\ 0 & 0 & 0 \end{bmatrix}$$

Jacobi's method

After decomposing matrix We can write the system of equations

$$\mathbf{Ax} = \mathbf{b}$$

in the form:

$$\mathbf{Dx} = -(\mathbf{L} + \mathbf{U})\mathbf{x} + \mathbf{b}$$

If We assume that diagonal entries of matrix \mathbf{A} are nonzero, then matrix \mathbf{D} is nonsingular therefore We can propose such an iterative method:

$$\mathbf{x}^{i+1} = -\mathbf{D}^{-1}(\mathbf{L} + \mathbf{U})\mathbf{x}^{(i)} + \mathbf{D}^{-1}\mathbf{b}$$

This is the Jacobi's method. We can rewrite this equation in the form of n independent scalar equations:

$$x_j^{i+1} = -\frac{1}{d_{jj}} \left(\sum_{k=1}^n (l_{jk} + u_{jk}) x_k^{(i)} + b_j \right)$$

Where d_{jj} ; l_{jk} ; u_{jk} are the elements of the respective matrixes \mathbf{D} , \mathbf{L} , \mathbf{U}

Thanks to this We can do those computations in parallel, totally or partially if We are using a computer that enables a parallelization of the computations.

Converging Jacobi's method is convergent if We have strong diagonal dominance of the matrix \mathbf{A}

Gauss-Seidel method

After decomposing matrix \mathbf{A} We can write the system of equations

$$\mathbf{Ax} = \mathbf{b}$$

in the form:

$$(\mathbf{L} + \mathbf{D})\mathbf{x} = -\mathbf{Ux} + \mathbf{b}$$

Again We assume that \mathbf{D} is nonsingular, in doing so We propose following iterative method:

$$\mathbf{Dx}^{(i+1)} = -\mathbf{Lx}^{(i+1)} - \mathbf{Ux}^{(i)} + \mathbf{b}$$

Since matrix \mathbf{L} is subdiagonal, provided that We organise the calculation of elements of the vector $x^{(i_1)}$ in a proper way, it does not hurt that $x^{(i_1)}$ is on the right side of the equation. In order to organise the calculation in the correct way We: First take into account the structure of matrixes \mathbf{D} and \mathbf{L} :

$$\begin{bmatrix} d_{11}x_1^{(i_1)} \\ d_{22}x_2^{(i_1)} \\ d_{33}x_3^{(i_1)} \\ \vdots \\ d_{nn}x_n^{(i_1)} \end{bmatrix} = - \begin{bmatrix} 0 & 0 & 0 & \cdots & 0 \\ l_{21} & 0 & 0 & \cdots & 0 \\ l_{32} & l_{32} & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ l_{n1} & l_{n2} & l_{n3} & \cdots & 0 \end{bmatrix} \begin{bmatrix} x_1^{(i_1)} \\ x_2^{(i_1)} \\ x_3^{(i_1)} \\ \vdots \\ x_n^{(i_1)} \end{bmatrix} - \mathbf{w}^{(i)}$$

Where

$$\mathbf{w}^{(i)} = \mathbf{Ux}^{(i)} - \mathbf{b}$$

So the order of calculations is as follows:

$$x_1^{(i+1)} = -\frac{w_1^{(i)}}{d_{11}}$$

$$x_2^{(i+1)} = -\frac{-l_{21}x_1^{(i+1)} - w_2^{(i)}}{d_{22}}$$

$$x_3^{(i+1)} = -\frac{-l_{31}x_1^{(i+1)} - l_{32}x_2^{(i+1)} - w_3^{(i)}}{d_{33}}$$

And so on

As opposed to Jacobi's method, Gauss-Seidel method computations must be performed sequentially. Every subsequent scalar equations uses results from the computation of the previous equations.

Converging Gauss-Seidel method is convergent if the matrix \mathbf{A} is strongly row or column diagonally dominant. If the matrix is symmetric, the method is also convergent if the matrix \mathbf{A} is positive definite. This method is also usually faster convergent compared to Jacobi's method.

Stop tests

There are two ways to check when to terminate iterations of the methods We just discussed:

1. Check differences between two subsequent iteration points

$$\|\mathbf{x}^{(i+1)} - \mathbf{x}^{(i)}\| \leq \delta$$

Where δ is an assumed tolerance, (in our case 10^{-10}). What We are really interested in though is whether the solution of the system of equation has required accuracy. If We want to check that We can additionally check (higher level, more computationally demanding test):

2. Check differences between two subsequent iteration points

$$\|\mathbf{Ax}^{(i+1)} - \mathbf{b}\| \leq \delta_2$$

Where δ_2 is an assumed tolerance. If this test is not passed then We can diminish the value of δ_2 and continue with the iterations. Value of δ_2 can not be too small since We are limited by the numerical errors.

A and b

We have been given the following system:

$$\begin{cases} 10x_1 - 4x_2 + x_3 + 2x_4 = -8 \\ 2x_1 - 6x_2 + 3x_3 - x_4 = -12 \\ x_1 + 4x_2 - 12x_3 + x_4 = 4 \\ 2x_1 + 3x_2 - 3x_3 - 10x_4 = 1 \end{cases}$$

Therefore our matrices will look like this:

$$\mathbf{A} = \begin{bmatrix} 10 & -4 & 1 & 2 \\ 2 & -6 & 3 & -1 \\ 1 & 4 & -12 & 1 \\ 2 & 3 & -3 & -10 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} -8 \\ -12 \\ 4 \\ 11 \end{bmatrix}$$

So:

$$\mathbf{L} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 \\ 1 & 4 & 0 & 0 \\ 2 & 3 & -3 & 0 \end{bmatrix} \quad \mathbf{D} = \begin{bmatrix} 10 & 0 & 0 & 0 \\ 0 & -6 & 0 & 0 \\ 0 & 0 & -12 & 0 \\ 0 & 0 & 0 & -10 \end{bmatrix} \quad \mathbf{U} = \begin{bmatrix} 0 & -4 & 1 & 2 \\ 0 & 0 & 3 & -1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\mathbf{D}^{-1} = \begin{bmatrix} \frac{1}{10} & 0 & 0 & 0 \\ 0 & -\frac{1}{6} & 0 & 0 \\ 0 & 0 & -\frac{1}{12} & 0 \\ 0 & 0 & 0 & -\frac{1}{10} \end{bmatrix}$$

3.3 Results

3.3.1 Jacobi method result

For system of equations We got in this task We got following results:
Without the change in demanded tolerance:

$$x = \begin{pmatrix} -0.076776098668341 \\ 2.105784262642568 \\ 0.395344797635474 \\ 0.397776619764909 \end{pmatrix}$$

Error:

$$r = \|\mathbf{Ax} - \mathbf{b}\| = 1.154375287358407e - 10$$

We managed to do this in **38** iterations of our loop, and the demanded tolerance did not change. (This required small change in code where We ommited the part of code responsible for changing demandedTolerance if $\|\mathbf{Ax} - b\| > \delta_2$))

With the change in demanded tolerance:

$$x = \begin{pmatrix} -0.076776098668341 \\ 2.105784262642568 \\ 0.395344797635474 \\ 0.397776619764909 \end{pmatrix}$$

Error:

$$r = \|\mathbf{Ax} - \mathbf{b}\| = 5.770361548895147e - 11$$

We got this result in **37** iterations and demanded tolerance was equal to $2 * 10^{-10}$

Compared to matlab function

$$x_{matlab} = \begin{pmatrix} -0.076776098662498 \\ 2.105784262636790 \\ 0.395344797637659 \\ 0.397776619767240 \end{pmatrix}$$

Matlab error:

$$r = \|\mathbf{Ax} - \mathbf{b}\| = 4.070144838902081e - 15$$

For data from task 2a We got:

Without change in demanded tolerance:

$$x_a = \begin{pmatrix} -0.930024655108186 \\ -1.223407298660663 \\ -1.273530574212508 \\ -1.230517757317628 \\ -1.151356031082747 \\ -1.056883669273682 \\ -0.952628310081466 \\ -0.834334594312996 \\ -0.683708806198363 \\ -0.450125157620744 \end{pmatrix}$$

Error:

$$r = \|\mathbf{Ax} - \mathbf{b}\| = 6.955194519943778e - 11$$

We managed to do this in **59** iterations of our loop, and the demanded tolerance did not change.

With change in demanded tolerance:

$$x_a = \begin{pmatrix} -0.930024655104470 \\ -1.223407298653515 \\ -1.273530574202540 \\ -1.230517757305602 \\ -1.151356031069692 \\ -1.056883669260597 \\ -0.952628310069469 \\ -0.834334594303006 \\ -0.683708806191233 \\ -0.450125157617020 \end{pmatrix}$$

Error:

$$r = \|\mathbf{Ax} - \mathbf{b}\| = 1.699812218689508e - 10$$

We managed to do this in **57** iterations of our loop, and the demanded tolerance changed to $4 * 10^{-10}$

Compared to matlab $A \ b$ function

$$x_{matlab} = \begin{pmatrix} -0.930024655110760 \\ -1.223407298665612 \\ -1.273530574219411 \\ -1.230517757325956 \\ -1.151356031091789 \\ -1.056883669282743 \\ -0.952628310089775 \\ -0.834334594319914 \\ -0.683708806203301 \\ -0.450125157623323 \end{pmatrix}$$

Matlab error:

$$r = \|\mathbf{Ax} - \mathbf{b}\| = 3.662053438817790e - 15$$

For Matrix and Vector from task 2b) error of

$$\|x^{(i+1)} - x^{(i)}\|$$

grew to infinity, therefore We could never achieve demanded tolerance, therefore the program executed infinite loop.

Minimizing the demanded error

We tried to minimize the demanded error using this steps:

1. We copied error from matlab function and pasted it into demanded tolerance.
2. If We did not get infinite loop We copied the newly acquired error and pasted it into demanded tolerance.
3. If We got infinite loop We used the previous error as **minimal** demanded error.

For original system of equations: We managed to get results with error as low as $1.776356839400250e-15$ with demanded tolerance = $3.202372833989376e-15$ for lower values program went into infinite loop. Results for demanded tolerance = $3.202372833989376e-15$ For given matrix:

$$x = \begin{pmatrix} -0.076776098662498 \\ 2.105784262636790 \\ 0.395344797637659 \\ 0.397776619767240 \end{pmatrix}$$

Error:

$$r = \|\mathbf{Ax} - \mathbf{b}\| = 3.108624468950438e-15$$

We got this result in **53** iterations and demanded tolerance did not change.

For task 2a) system of equations: We managed to get results with error as low as

$$3.108624468950438e-15$$

with demanded tolerance:

$$3.202372833989376e-15$$

for lower values program went into infinite loop.

For demanded tolerance = $3.202372833989376e - 15$: Results for 2a) system of equation

$$x_a = \begin{pmatrix} -0.930024655110760 \\ -1.223407298665613 \\ -1.273530574219411 \\ -1.230517757325955 \\ -1.151356031091788 \\ -1.056883669282743 \\ -0.952628310089775 \\ -0.834334594319914 \\ -0.683708806203301 \\ -0.450125157623323 \end{pmatrix}$$

Error:

$$r = \|\mathbf{Ax} - \mathbf{b}\| = 3.108624468950438e - 15$$

We managed to do this in **84** iterations of our loop, and the demanded tolerance did not change. We managed to achieve slightly better (as in, the error was smaller) results than Matlab custom function.

3.3.2 Gauss-Seidel method result

For system of equations We got in this task We got following results:

$$x = \begin{pmatrix} -0.076776098752996 \\ 2.105784262542701 \\ 0.395344797589652 \\ 0.397776619735316 \end{pmatrix}$$

Error:

$$r = \|\mathbf{Ax} - \mathbf{b}\| = 6.999102842719934e - 10$$

We managed to do this in **19** iterations of our loop, and the demanded tolerance did not change. Error:

$$r = \|\mathbf{Ax} - \mathbf{b}\| = 6.999102842719934e - 10$$

We got this result in **19** iterations and demanded tolerance was equal to $2 * 10^{-10}$

For data from task 2a We got:

$$x_a = \begin{pmatrix} -0.930024655049330 \\ -1.223407298590161 \\ -1.273530574151901 \\ -1.230517757273947 \\ -1.151356031055568 \\ -1.056883669259564 \\ -0.952628310076145 \\ -0.834334594312669 \\ -0.683708806199986 \\ -0.450125157622217 \end{pmatrix}$$

Error:

$$r = \|\mathbf{Ax} - \mathbf{b}\| = 5.311669113699570e - 10$$

We managed to do this in **28** iterations of our loop, and the demanded tolerance did not change.

For Matrix and Vector from task 2b) error of

$$\|x^{(i+1)} - x^{(i)}\|$$

grew to infinity, therefore We could never achieve demanded tolerance, therefore the program executed infinite loop.

Minimizing the demanded error

I tried to minimize the demanded error using this steps:

1. I copied error from matlab function and pasted it into demanded tolerance.
2. If I did not get infinite loop I copied the newly acquired error and pasted it into demanded tolerance.
3. If I got infinite loop I used the previous error as **minimal** demanded error.

For original system of equations: We managed to get results with error as low as $1.776356839400250e-15$ with demanded tolerance = $1.986027322597818e-$

15 for lower values program went into infinite loop. Results for demanded tolerance = $1.986027322597818e - 15$ For given matrix:

$$x = \begin{pmatrix} -0.076776098662498 \\ 2.105784262636790 \\ 0.395344797637659 \\ 0.397776619767240 \end{pmatrix}$$

Error:

$$r = \|\mathbf{Ax} - \mathbf{b}\| = 1.776356839400250e - 15$$

We got this result in **30** iterations and demanded tolerance did not change.

For task 2a) system of equations: We managed to get results with error as low as

$$1.538370149106851e - 15$$

with demanded tolerance:

$$1.538370149106851e - 15$$

for lower values program went into infinite loop.

For demanded tolerance = $1.538370149106851e - 15$: Results for 2a) system of equation

$$x_a = \begin{pmatrix} -0.930024655110760 \\ -1.223407298665613 \\ -1.273530574219411 \\ -1.230517757325955 \\ -1.151356031091788 \\ -1.056883669282743 \\ -0.952628310089775 \\ -0.834334594319914 \\ -0.683708806203301 \\ -0.450125157623323 \end{pmatrix}$$

Error:

$$r = \|\mathbf{Ax} - \mathbf{b}\| = 1.538370149106851e - 15$$

We managed to do this in **42** iterations of our loop, and the demanded tolerance did not change. We managed to achieve slightly better (as in, the error was smaller) results than Matlab custom function.

Table

system of equations	method	demanded tolerance	error	iterations
task 3 system	Jacobi method	10e-10	1.154375287358407e-10	38
task 3 system	Jacobi method	10e-10	5.770361548895147e-11	37
task 3 system	Jacobi method	3.202372833989376e-15	3.108624468950438e-15	53
task 3 system	Gaussian-Seidel method	10e-10	6.999102842719934e-10	19
task 3 system	Gaussian-Seidel method	1.776356839400250e-15	1.776356839400250e-15	30
task 3 system	Matlab function	?	4.070144838902081e-15	?
task 2a) system	Jacobi method	10e-10	6.955194519943778e-11	59
task 2a) system	Jacobi method	10e-10	1.699812218689508e-10	57
task 2a) system	Jacob method	3.202372833989376e-15	3.108624468950438e-15	84
task 2a) system	Gaussian-Seidel method	10e-10	5.311669113699570e-10	28
task 2a) system	Gaussian-Seidel method	1.538370149106851e-15	1.538370149106851e-15	42
task 2a) system	Matlab function	?	3.662053438817790e-15	?

3.4 Discussion of results

3.4.1 Comparison based on table

Gaussian-Seidel method takes less iterations, can achieve smaller error and works under smaller demanded tolerance, that being said Jacobi method has one crucial advantage - it can be run in parallel. That means that even when Jacobi method takes twice or thrice amount of iteration us Gaussian-Seidel one if we just run it in paraller we can diminish this difference. In the examples we worked in, just two jacobi method running in paraller provide us with similar amount of iterations per one process as in Gaussian-Seidel one, and three or more processes give the edge to Jacobi method. With modern CPU's with at least 5 cores and correct implementation of algorithm, jacobi method reigns supreme.

3.4.2 Convergence

The biggest problem is of course the fact that those iterative methods do not work on every system of equation. Like the one from task 2b). For Jacobi's method the matrix is convergent if it has strong diagonal dominance. \mathbf{A} , i.e:

1.

$$\|a_{ii}\| > \sum_{j=1, j \neq i}^n \|a_{ij}\|$$

2.

$$\|a_{jj}\| > \sum_{i=1, i \neq j}^n \|a_{ij}\|$$

For Gaussian-Seidel method the matrix is convergent if it has strong diagonal dominance **or** if the matrix **A** is symmetric and positive definite.

2b) task convergence

Let's check strong diagonal dominance from matrix **A** from task 2b) in matlab:

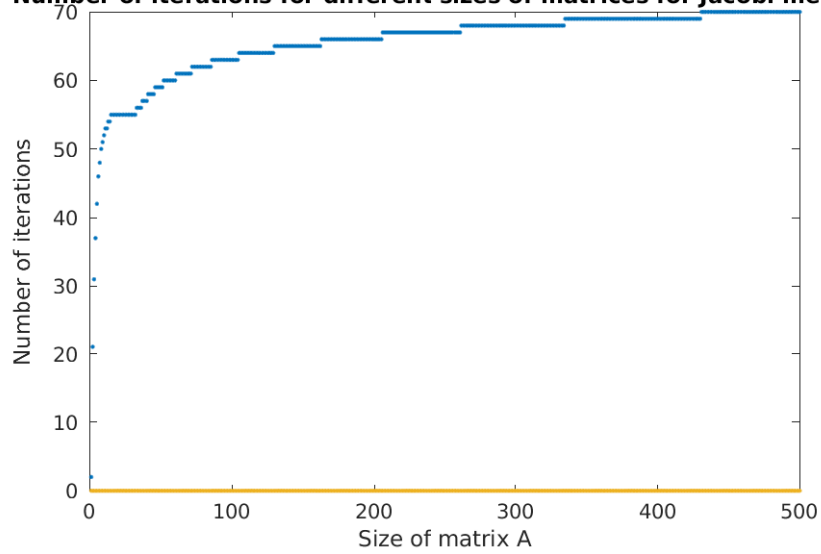
```
1 function d = checkIfDiagonallyDominant(Matrix)
2     d = 1;
3     [Rows, ~] = size(Matrix);
4     for i = 1 : Rows
5         rowsSum = 0;
6         for j = 1 : Rows
7             rowsSum = rowsSum + abs(Matrix(i, j));
8         end
9         rowsSum = rowsSum - Matrix(i, i);
10        if Matrix(i, i) <= rowsSum
11            d = 0;
12            break
13        end
14    end
15 end
```

checkIfDiagonallyDominant(matrixB(10)) returns '0', therefore matrix from task 2b) will **not** converge. And as expected it does not work when put in my jacobiLoop function.

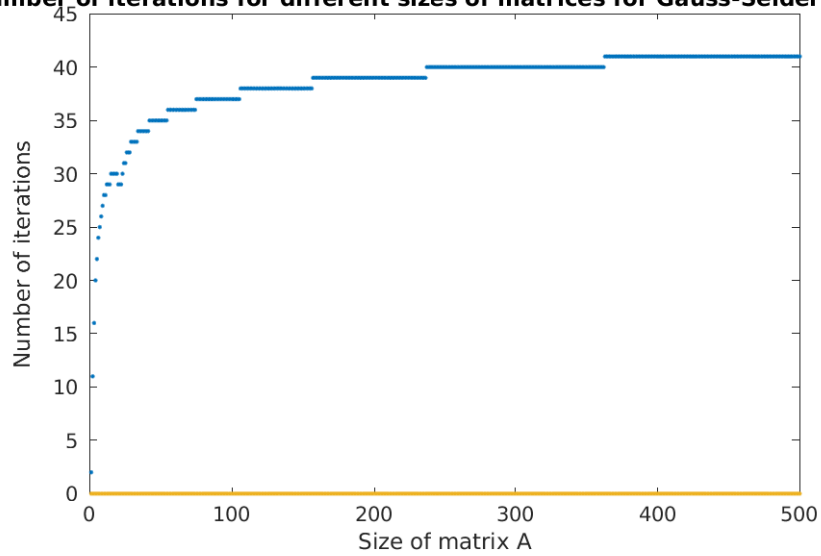
Iterations as function of size of Matrix

I have used matrix from task 2a), demanded tolerance equal to 10e-10, and max size of matrix equal to 500. for both methods

Number of iterations for different sizes of matrices for Jacobi method



Number of iterations for different sizes of matrices for Gauss-Seidel meth



Gauss Seidel method again requires fewer iterations, around half the amount for jacobi method. For both methods number of iterations acts like logarithmic function. Therefore we can assume that for big matrices required number of iterations will not change so drastically.

Chapter 4

Problem 4 - QR method of finding eigenvalues

4.1 Problem

We must find eigenvalues of 5x5 matrices. Both without shifts and with shifts.

4.2 Theoretical introduction

4.2.1 Eigenvalues

Eigenvalues of matrix \mathbf{A} are defined as a pair of number λ and vector \mathbf{v} such that:

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$$

This can be rewritten as:

$$(\mathbf{A} - \lambda\mathbf{I})\mathbf{v} = 0$$

And further we get characteristic equation:

$$\det(\mathbf{A} - \lambda\mathbf{I}) = 0$$

4.2.2 QR method for finding eigenvalues

First we start by transforming matrix \mathbf{A} into tridiagonal form. It increases the effectiveness of calculations.

Single step of QR method consists of:

$$\mathbf{A}^{(k)} = \mathbf{Q}^{(k)} \mathbf{R}^{(k)}$$

$$\mathbf{A}^{(k+1)} = \mathbf{R}^{(k)} \mathbf{Q}^{(k)}$$

$\mathbf{Q}^{(k)}$ being orthogonal, therefore:

$$\mathbf{R}^{(k)} = (\mathbf{Q}^{(k)})^{-1} \mathbf{A}^{(k)} = \mathbf{Q}^{(k)T} \mathbf{A}^{(k)} \mathbf{Q}^{(k)}$$

therefore:

$$\mathbf{A}^{(k+1)} = \mathbf{Q}^{(k)T} \mathbf{A}^{(k)} \mathbf{Q}^{(k)}$$

If we have symmetric matrix \mathbf{A} it converges to the diagonal matrix $diag(\lambda_i)$

Since this method can be slowly convergent we use shifts. Single iteration of QR method with shifts looks like this:

$$\begin{aligned} \mathbf{A}^{(k)} - p_k \mathbf{I} &= \mathbf{Q}^{(k)} \mathbf{R}^{(k)} \\ \mathbf{A}^{(k+1)} &= \mathbf{R}^{(k)} \mathbf{Q}^{(k)} + p_k \mathbf{I} \\ &= \mathbf{Q}^{(k)T} (\mathbf{A}^{(k)} - p_k \mathbf{I}) \mathbf{Q}^{(k)} + p_k \mathbf{I} \\ &= \mathbf{Q}^{(k)T} \mathbf{A}^{(k)} \mathbf{Q}^{(k)} \end{aligned} \tag{4.1}$$

Where p_k should be chosen as a best estimate of λ_{i+1}

4.3 Results

4.3.1 Starting matrix

I decided to generate random symmetric matrix using following code:

```
1 function A = matrix4()
2     A = 10 * rand(5); % rand generates 5x5 matrix filled with
   random numbers
3     % we multiply by 10 to get at least one digit in front of
   the dot
4     A = floor(A); % we floor the matrix we got to get nice
   natural numbers matrix
5     A = A * A'; % we get symmetric matrix
6     disp(issymmetric(A)); % we check if matrix is symmetric
7 end
```

I got the following matrix:

$$\begin{bmatrix} 171 & 48 & 133 & 81 & 93 \\ 48 & 108 & 63 & 35 & 30 \\ 133 & 63 & 131 & 64 & 91 \\ 81 & 35 & 64 & 41 & 37 \\ 93 & 30 & 91 & 37 & 106 \end{bmatrix}$$

Putting it through Matlab eig function I got eigen values equal to:

$$10^2 * \begin{bmatrix} 0.000050598692964 \\ 0.105696602175690 \\ 0.454589544003707 \\ 0.870708987227002 \\ 4.138954267900641 \end{bmatrix}$$

4.3.2 QR method with no shifts

Putting this matrix to my functions QRNoShifts and QRShifts I got: for qr method with no shifts: Final matrix:

$$\begin{bmatrix} 413.8954 & 0.0000 & -0.0000 & 0.0000 & 0.0000 \\ 0.0000 & 87.0709 & -0.0000 & -0.0000 & 0.0000 \\ 0.0000 & -0.0000 & 45.4590 & -0.0000 & -0.0000 \\ -0.0000 & 0.0000 & -0.0000 & 10.5697 & -0.0000 \\ 0.0000 & -0.0000 & 0.0000 & 0 & 0.0051 \end{bmatrix}$$

Eigen values:

$$10^2 * \begin{bmatrix} 4.138954267900639 \\ 0.870708987227001 \\ 0.454589544003706 \\ 0.105696602175689 \\ 0.000050598692964 \end{bmatrix}$$

It took **26** iterations to get those values.

4.3.3 QR method with shifts

For qr method with shiftts: Final matrix:

$$[413.8954]$$

Eigen values:

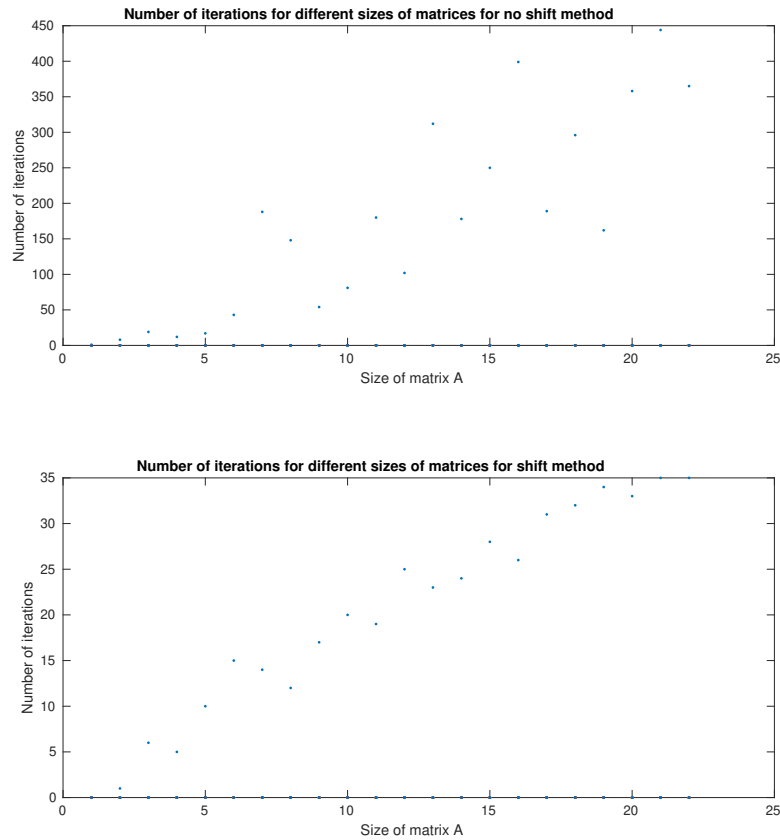
$$10^2 * \begin{bmatrix} 0.454589544003706 \\ 0.870708987227001 \\ 0.105696602175689 \\ 0.000050598692964 \\ 4.138954267900639 \end{bmatrix}$$

It took **12** iterations to get those values.

4.4 Discussion of the result

Following plot was generated using task4Plot function, using maxMatrixSize = 25 and generating random symmetrical matrix of size i x i in each loop. Both QR methods were given exactly the same matrices.

4.4.1 Plot



4.4.2 Shift method superiority

As we can see: QR method with shifts is much more efficient and reliable.

1. For big enough matrices it requires ten times less iterations compared to QR method without shifts.
2. It is more efficient in our chosen matrix.
3. It can work with non-symmetric matrices
4. Has no problem with generating complex eigen values.

One thing that does not change much is the precision, with both algorithms outputting similar results.

Chapter 5

Code appendix

5.1 Task 2 Code

5.1.1 Main function

```
1 function x = indicatedMethod(Matrix, Vector) % Name of the
    method as in the textbook
2 % x stands for obtained result
3     [~,Columns] = size(Matrix); % We need to know how big the
    matrix is in next steps
4     % notice the '~', since We assume We use square matrix, We
    do not need
5     % to have another variable for number of rows since it is
    the same as
6     % number of columns
7     checkIfMatrixIsSquareMatrix(Matrix);
8     [Matrix, Vector] = gaussianEliminationWithPartialPivoting(
        Columns, Matrix, Vector);
9     % Change matrix to upper triangular matrix
10    [Matrix, Vector, x] = backSubstitutionPhase(Columns, Matrix
        , Vector);
11    % Get the solution
12    x = iterativeResidualCorrection(Matrix, x, Vector); %
        Improve on the solution
13 end % end function
```

5.1.2 checkIfMatrixIsSquareMatrix

```
1 function checkIfMatrixIsSquareMatrix(Matrix)
2     [Rows,Columns] = size(Matrix);
3     if Rows ~= Columns
4         error ('Matrix is not square matrix!');
5     end % end if
6 end % end function
```

5.1.3 gaussianEliminationWithPartialPivoting

```
1 function [Matrix, Vector] =  
    gaussianEliminationWithPartialPivoting(Columns, Matrix,  
    Vector)  
2     for j = 1 : Columns  
3         centralElement = max(Matrix(j:Columns,j));  
4         % We stay in the same row (j) but We change columns  
         % , as in the  
5         % textbook  
6         [Matrix, Vector] = partialPivoting(Matrix, Vector,  
            j, centralElement, Columns);  
7         % ensures that a_kk != 0 and reduces errors  
8         [Matrix, Vector] = gaussianElimination(j, Columns,  
            Matrix, Vector);  
9         % change matrix into upper triangular matrix  
10    end % end for  
11 end % end function
```

5.1.4 partialPivoting

```
1 function [Matrix, Vector] = partialPivoting(Matrix, Vector, j,  
    centralElement, Columns)  
2     for k = j : Columns  
3         partialPivotingSwapOneRow(Matrix, Vector, j, k,  
            centralElement);  
4     end % end for  
5 end % end function
```

5.1.5 partialPivotingSwapOneRow

```
1 function [Matrix, Vector] = partialPivotingSwapOneRow(Matrix,  
    Vector, j, k, centralElement)  
2     if Matrix(k,j) == centralElement  
3         swapRowMatrix(Matrix, j, k); % swap jth row with kth row
```

```

4     swapValueVector(Vector, j, k); % swap jth value with kth
      value
5     end % end if
6 end % end function

```

5.1.6 swapRowMatrix

```

1 function Matrix = swapRowMatrix(Matrix, j, k)
2     temp = Matrix(j , :); % ' : ' denote all elements in jth
      row
3     Matrix(j , :) = Matrix(k, :);
4     Matrix(k, :) = temp; % temp equal to previous value of jth
      row
5 end

```

5.1.7 swapValueVector

```

1 function Vector = swapValueVector(Vector, j, k)
2     temp = Vector(j);
3     Vector(j) = Vector(k);
4     Vector(k) = temp; % temp equal to previous value of k
      element of vector
5 end % end function

```

5.1.8 gaussianElimination

```

1 function [Matrix, Vector] = gaussianElimination(j, Columns,
      Matrix, Vector)
2     for i = j + 1 : Columns
3         rowMultiplier = Matrix(i,j) / Matrix(j,j);
4         [Matrix, Vector] = subtractRows(Matrix, Vector, i,
            rowMultiplier, j, Columns);
5     end % end for
6 end % end function

```

5.1.9 subtractRows

```
1 function [Matrix, Vector] = subtractRows(Matrix, Vector, i,  
    rowMultiplier, j, Columns)  
2     Vector(i) = Vector(i) - rowMultiplier * Vector(j);  
3     for curentColumn = 1 : Columns  
4         Matrix(i,curentColumn) = Matrix(i,curentColumn) -  
            rowMultiplier * Matrix(j, curentColumn);  
5     end % end for  
6 end % end function
```

5.1.10 backSubstitutionPhase

```
1 function [Matrix, Vector, x] = backSubstitutionPhase(Columns,  
    Matrix, Vector)  
2     for k = Columns : -1 : 1  
3         % Start at final column and move by -1 each iteration until  
         % We reach 1  
4         equation = 0;  
5         for j = k+1 : Columns  
6             equation = equation + Matrix(k,j) * x(j, 1);  
7             % even though x is a vector We still need to put  
             % '1' to ensure  
8             % that number of columns in the first matrix  
             % matches number of  
9             % rows in second matrix  
10        end % end for  
11  
12        x(k, 1) = (Vector(k,1) - equation) / Matrix(k,k);  
13        % even though x is a vector We still need to put '1' to  
            % ensure  
14        % that We do not exceed array bounds  
15    end % end for  
16 end % end function
```

5.1.11 iterativeResidualCorrection

```
1 function x = iterativeResidualCorrection(Matrix, x, Vector)  
2     residuum = Matrix*x - Vector; % as in the book  
3     newResiduum = residuum;  
4     x = improveSolution(x, newResiduum, residuum, Matrix,  
        Vector);  
5 end % end function
```

5.1.12 improveSolution

```

1 function x = iterativeResidualCorrection(A, x, b)
2     r = A*x - b;
3     for i = 1 : 100
4         [~, ~, deltaX] = solveSystem(A, r);
5         newX = x - deltaX;
6         r = A*newX - b;
7         x = newX;
8     end
9
10 end % end function

```

5.2 Task 3 code

```

1 function [x_j, x_g] = iterative(Matrix, Vector)
2     [L, D, U, initial_x, whichIterationAreWeOnJ,
3         whichIterationAreWeOnG, demandedToleranceJ,
4         demandedToleranceG, flag, Rows] = initializeValues(
5             Matrix);
6     [x_j, whichIterationAreWeOnJ, demandedToleranceJ] =
7         jacobiLoop(Matrix, L, D, U, initial_x,
8             whichIterationAreWeOnJ, demandedToleranceJ, Vector, flag
9         );
10    [x_g, whichIterationAreWeOnG, demandedToleranceG] =
11        gaussSeidelLoop(Matrix, L, D, U, initial_x,
12            whichIterationAreWeOnG, demandedToleranceG, Vector, flag
13            , Rows);
14    dispFinalResults(x_j, x_g, demandedToleranceJ,
15        demandedToleranceG, whichIterationAreWeOnJ,
16        whichIterationAreWeOnG, Matrix, Vector);
17 end

```

5.2.1 initializeValues

```

1 function [L, D, U, initial_x, whichIterationAreWeOnJ,
2     whichIterationAreWeOnG, demandedToleranceJ,

```

```

1      demandedToleranceG, flag, Rows] = initializeValues(Matrix)
2      [Rows, ~] = size(Matrix);
3      [L, D, U] = decomposeMatrix(Matrix);
4      initial_x = zeros(Rows, 1);
5      whichIterationAreWeOnJ = 0;
6      whichIterationAreWeOnG = 0;
7      demandedToleranceJ = 10e-10; % as per task description
8      demandedToleranceG = 10e-10; % as per task description
9      flag = 0;
10 end

```

5.2.2 decomposeMatrix

```

1 function [L, D, U] = decomposeMatrix(Matrix)
2     D = diag(diag(Matrix));
3     U = triu(Matrix, 1); % Generates upper triangular part of
4         matrix
5     % where the second variable denotes on which diagonal of
6         matrix should we
7         % start
8     L = tril(Matrix, -1); % Generates lower triangular part of
9         matrix
10    % where the second variable denotes on which diagonal of
11        matrix should we
12        % start
13 end

```

5.2.3 jacobiLoop

```

1 function [x_j, whichIterationAreWeOn, demandedTolerance] =
2     jacobiLoop(Matrix, L, D, U, initial_x, whichIterationAreWeOn,
3         demandedTolerance, Vector, flag)
4     while flag ~= 1 % flag denotes whether norm(Matrix*x_g-
5         Vector) <= demandedTolerance
6         [x_j, whichIterationAreWeOn, demandedTolerance, flag,
7             initial_x] = jacobiInsideLoop(Matrix, L, D, U,
8                 initial_x, whichIterationAreWeOn, demandedTolerance,
9                 Vector, flag);
10    end

```



```

        initial_x, whichIterationAreWeOn, demandedTolerance,
        Vector);
4     end
5 end

```

5.2.4 jacobiInsideLoop

```

1 function [x_j, whichIterationAreWeOn, demandedTolerance, flag,
   initial_x] = jacobiInsideLoop(Matrix, L, D, U, initial_x,
   whichIterationAreWeOn, demandedTolerance, Vector)
2     x_j = jacobiEquation(D, L, U, initial_x, Vector);
3     [flag, demandedTolerance] = checkError(x_j, initial_x,
        demandedTolerance, Matrix, Vector);
4     [initial_x, whichIterationAreWeOn] = endOfLoop(x_j,
        whichIterationAreWeOn);
5 end

```

5.2.5 jacobiEquation

```

1 function x = jacobiEquation(D, L, U, initial_x, Vector)
2     x = - D \ ( L + U ) * initial_x + D \ Vector; % As per
        formula
3     % We will be using D \ Vector and D \ ( ) instead of
        inverseD since
4     % this is faster according to matlab
5 end

```

5.2.6 gaussSeidelLoop

```

1 function [x_g, whichIterationAreWeOn, demandedTolerance] =
   gaussSeidelLoop(Matrix, L, D, U, initial_x,
   whichIterationAreWeOn, demandedTolerance, Vector, flag, Rows
   )
2     while flag ~= 1 % flag denotes whether norm(Matrix*x_g-
        Vector) <= demandedTolerance

```

```

3         [x_g, whichIterationAreWeOn, demandedTolerance, flag,
           initial_x] = gaussiInsideLoop(Matrix, L, D, U,
           initial_x, whichIterationAreWeOn, demandedTolerance,
           Vector, Rows);
4     end
5 end

```

5.2.7 gaussiInsideLoop

```

1 function [x_j, whichIterationAreWeOn, demandedTolerance, flag,
           initial_x] = gaussiInsideLoop(Matrix, L, D, U, initial_x,
           whichIterationAreWeOn, demandedTolerance, Vector, Rows)
2     x_j = gaussSeidelEquation(D, L, U, initial_x, Vector, Rows)
           ;
3     [flag, demandedTolerance] = checkError(x_j, initial_x,
           demandedTolerance, Matrix, Vector);
4     [initial_x, whichIterationAreWeOn] = endOfLoop(x_j,
           whichIterationAreWeOn);
5 end

```

5.2.8 gaussSeidelEquation

```

1 function x_g = gaussSeidelEquation(D, L, U, initial_x, Vector,
           Rows)
2     W = U*initial_x - Vector;
3     x_g(1, 1) = -W(1, 1) / D(1,1);
4     for i = 2 : Rows
5         x_g(i, 1) = calculateNominator(i, L, x_g, W) / D(i, i);
6     end
7 end

```

5.2.9 checkError

```

1 function [flag, demandedTolerance] = checkError(x, initial_x,
           demandedTolerance, Matrix, Vector)

```

```

2     flag = 0;
3     currentError = norm(x - initial_x);
4     if currentError <= demandedTolerance
5         currentError = norm(Matrix*x-Vector);
6         if currentError <= demandedTolerance % if sequence as
           per textbook
7             flag = 1;
8         else
9             demandedTolerance = demandedTolerance * 2; %
              arbitrary value
10        end
11    end
12 end

```

5.2.10 endOfLoop

```

1 function [initial_x, whichIterationAreWeOn, flag] = endOfLoop(x
   , whichIterationAreWeOn)
2     initial_x = x;
3     whichIterationAreWeOn = whichIterationAreWeOn + 1;
4     flag = 0;
5 end

```

5.2.11 dispFinalResults

```

1 function dispFinalResults(x_j, x_g, demandedToleranceJ,
   demandedToleranceG, whichIterationAreWeOnJ,
   whichIterationAreWeOnG, Matrix, Vector)
2     disp(Final demandedTolerance for Jacobi method);
3     disp(demandedToleranceJ);
4     disp(Final demandedTolerance for Gaussian-Seidel method:);
5     disp(demandedToleranceG);
6     disp(Final Iteration for Jacobi method: );
7     disp(whichIterationAreWeOnJ);
8     disp(Final Iteration for Gaussian-Seidel method: );
9     disp(whichIterationAreWeOnG);

```

```

10     disp(Error for Jacobi method:);
11     disp(norm(Matrix*x_j - Vector));
12     disp(Error for Gaussian-Seidel method:);
13     disp(norm(Matrix*x_g - Vector));
14     disp(Aerror:);
15     disp(norm(Matrix * (Matrix\Vector) - Vector));
16     disp(Answer for Jacobi method: );
17     disp(x_j);
18     disp(Answer for Gaussian-Seidel method: );
19     disp(x_g);
20 end

```

5.2.12 plotIterations

```

1 function plotIterations()
2     maxMatrixSize = 500;
3     iterationsJ = zeros(maxMatrixSize);
4     iterationsG = zeros(maxMatrixSize);
5     for i = 1 : maxMatrixSize
6         [~, ~, whichIterationAreWeOnJ, whichIterationAreWeOnG]
           = iterative(matrixA(i), vectorA(i));
7         iterationsJ(i) = whichIterationAreWeOnJ;
8         iterationsG(i) = whichIterationAreWeOnG;
9     end
10    nexttile
11    plot(iterationsJ, '.');
12    title('Number of iterations for different sizes of matrices
           for Jacobi method');
13    xlabel('Size of matrix A');
14    ylabel('Number of iterations');
15    nexttile
16    plot(iterationsG, '.');
17    title('Number of iterations for different sizes of matrices
           for Gauss-Seidel method');
18    xlabel('Size of matrix A');
19    ylabel('Number of iterations');
20

```

```
21 end
```

5.2.13 plotIterations

```
1 function plotErrorsGaussian(maxMatrixSize)
2
3     errorsA = zeros(maxMatrixSize);
4     errorsB = zeros(maxMatrixSize);
5     errorsAR = zeros(maxMatrixSize);
6     errorsBR = zeros(maxMatrixSize);
7     for i = 1 : maxMatrixSize
8         [~, errorBeforeResidualCorrection,
9             errorAfterResidualCorrection] = indicatedMethod(
10             matrixA(i), vectorA(i));
11         errorsA(i) = errorBeforeResidualCorrection;
12         errorsAR(i) = errorAfterResidualCorrection;
13         [~, errorBeforeResidualCorrection,
14             errorAfterResidualCorrection] = indicatedMethod(
15             matrixB(i), vectorB(i));
16         errorsB(i) = errorBeforeResidualCorrection;
17         errorsBR(i) = errorAfterResidualCorrection;
18     end
19     nexttile
20     plot(errorsA, '.');
21     title('Errors before residual correction for task 2a:');
22     xlabel('Size of matrix A');
23     ylabel('Errors');
24     nexttile
25     plot(errorsAR, '.');
26     title('Errors after residual correction for task 2a:');
27     xlabel('Size of matrix A');
28     ylabel('Errors');
29     nexttile
30     plot(errorsB, '.');
31     title('Errors before residual correction for task 2b:');
32     xlabel('Size of matrix A');
33     ylabel('Errors');
```

```

30     nexttile
31     plot(errorsBR, '.');
32     title('Errors after residual correction for task 2b:');
33     xlabel('Size of matrix A');
34     ylabel('Errors');
35 end

```

5.3 Task 4 Code

5.3.1 Gram-Schmid algorithm

```

1 % performs QR or QRdash decomposition of a matrix
2 function [Q, R] = gramSchmidtAlgorithm(Matrix)
3     [columns, Q, R, d] = initializeGramSchmid(Matrix);
4     [Q, R] = factorizeColumnsOfQ(columns, Q, Matrix, R, d);
5     [Q, R] = normalizeColumns(columns, Q, R);
6 end

```

initializeGramSchmid

```

1 function [columns, Q, R, d] = initializeGramSchmid(Matrix)
2     % We start with empty matrices
3     [rows, columns] = size(Matrix);
4     Q = zeros(rows, columns);
5     R = zeros(columns, columns);
6     d = zeros(1, columns);
7 end

```

initializeGramSchmid

```

1 function [Q, R] = factorizeColumnsOfQ(columns, Q, Matrix, R, d)
2     for i = 1 : columns
3         Q(:, i) = Matrix(:, i);
4         R(i, i) = 1;
5         d(i) = Q(:, i)' * Q(:, i);

```

```

6         for i2 = i + 1 : columns
7             R(i, i2) = (Q(:, i)' * Matrix(:, i2)) / d(i);
8             Matrix(:, i2) = Matrix(:, i2) - R(i, i2) * Q(:, i);
9         end
10    end
11 end

```

initializeGramSchmid

```

1 function [Q, R] = normalizeColumns(columns, Q, R)
2     for i = 1 : columns
3         dd = norm(Q(:, i));
4         Q(:, i) = Q(:, i) / dd;
5         R(i, i:columns) = R(i, i : columns) * dd;
6     end
7 end

```

```

1 function [eigenValuesNoShifts, iterationsNoShifts,
    finalMatrixNoShifts, eigenValuesShifts, iterationsShifts,
    finalMatrixShifts] = task4(Matrix)
2     [eigenValuesNoShifts, iterationsNoShifts,
    finalMatrixNoShifts] = QRNoShifts(Matrix);
3     [eigenValuesShifts, iterationsShifts, finalMatrixShifts] =
    QRShifts(Matrix);
4 end

```

5.3.2 QRNoShifts

```

1 function [eigenValues, whichIterationAreWeOn, Matrix] =
    QRNoShifts(Matrix)
2
3     [whichIterationAreWeOn, threshold, startingMatrix,
    matlabEigen] = initializeValues(Matrix);
4     [Matrix, whichIterationAreWeOn] = QRNoShiftsLoop(threshold,
    Matrix, whichIterationAreWeOn);
5     eigenValues = diag(Matrix)';

```

```

6     displayResults(eigenValues, whichIterationAreWeOn, Matrix,
    startingMatrix, matlabEigen);
7 end

```

QRNoShiftsLoop

```

1 function [Matrix, whichIterationAreWeOn] = QRNoShiftsLoop(
    threshold, Matrix, whichIterationAreWeOn)
2     while threshold > 1e-6
3         [Matrix, whichIterationAreWeOn, threshold] =
            QRNoShiftsInsideLoop(Matrix, whichIterationAreWeOn);
4     end
5 end

```

QRNoShiftsInsideLoop

```

1 function [Matrix, whichIterationAreWeOn, threshold] =
    QRNoShiftsInsideLoop(Matrix, whichIterationAreWeOn)
2     [Q, R] = gramSchmidtAlgorithm(Matrix);
3     Matrix = R * Q;
4     whichIterationAreWeOn = whichIterationAreWeOn + 1;
5
6     % iterate until all non-diagonal elements are below the
        threshold
7     matrixWithoutDiagonal = Matrix - diag(diag(Matrix)); %
        first diag converts Matrix
8     % into vector consisting of values on the diagonal of
        matrix,
9     % second diag converts this vector into matrix with zeros
        on
10    % everything except diagonal
11    % If we subtract it from Matrix we get original Matrix
        with zeros
12    % on a diagonal
13    threshold = max(max(abs(matrixWithoutDiagonal)));
14    % first max returns vector of elements

```



```

15     % second max returns max element from this vector
16 end

```

displayResults

```

1 function displayResults(eigenValues, whichIterationsAreWeOn,
    Matrix, startingMatrix, matlabEigen)
2     disp(How many iterations it took:)
3     disp(whichIterationsAreWeOn)
4     disp(Starting Matrix:)
5     disp(startingMatrix)
6     disp(Final Matrix:)
7     disp(Matrix)
8     disp(eig(Matrix) eigen values:)
9     disp(matlabEigen)
10    disp(Our eigen values:)
11    disp(eigenValues);
12 end

```

initializeValues

```

1 function [whichIterationAreWeOn, threshold, startingMatrix,
    matlabEigen] = initializeValues(Matrix)
2     whichIterationAreWeOn = 0;
3     threshold = inf;
4     startingMatrix = Matrix;
5     matlabEigen = eig(Matrix);
6 end

```

5.3.3 QRShifts

```

1 function [eigenValues, whatIterationAreWeOn, Matrix] = QRShifts
    (Matrix)
2     eigenFromMatlab = eig(Matrix);
3     initialMatrix = Matrix;

```

```

4      [eigenValues, whatIterationAreWeOn, matrixSize,
        minThreshold] = initiateValues(Matrix);
5      [Matrix, whatIterationAreWeOn, eigenValues] = QRShiftLoop(
        matrixSize, Matrix, eigenValues, minThreshold,
        whatIterationAreWeOn);
6      dispResults(eigenValues, Matrix, whatIterationAreWeOn,
        eigenFromMatlab, initialMatrix);
7  end

```

initiateValues

```

1  function [eigenValues, whatIterationAreWeOn, matrixSize,
        minThreshold] = initiateValues(Matrix)
2      eigenValues = double.empty(1, 0);
3      whatIterationAreWeOn = 0;
4      matrixSize = size(Matrix, 1);
5      minThreshold = 1e-6;
6  end

```

QRShiftLoop

```

1  function [Matrix, whatIterationAreWeOn, eigenValues] =
        QRShiftLoop(matrixSize, Matrix, eigenValues, minThreshold,
        whatIterationAreWeOn)
2      while matrixSize >= 2
3          flag = 0;
4          [Matrix, matrixSize, whatIterationAreWeOn, eigenValues]
                = findEigenValue(Matrix, matrixSize,
                whatIterationAreWeOn, eigenValues, minThreshold,
                flag);
5          [matrixSize, Matrix] = deflateMatrix(Matrix, matrixSize
                );
6      end
7      eigenValues(size(eigenValues, 2) + 1) = Matrix(1, 1);
8  end

```

findEigenValue

```
1 function [Matrix, matrixSize, whatIterationAreWeOn, eigenValues
   ] = findEigenValue(Matrix, matrixSize, whatIterationAreWeOn,
   eigenValues, minThreshold, flag)
2 while flag == 0
3     eigenValueCorner = getEigenValueFromCorner(Matrix,
         matrixSize);
4     [Matrix, whatIterationAreWeOn] = shiftAndIterate(
         matrixSize, Matrix, eigenValueCorner,
         whatIterationAreWeOn);
5     [flag, eigenValues] = thresholdBreached(Matrix,
         matrixSize, minThreshold, eigenValues);
6 end
7 end
```

getEigenValueFromCorner

```
1 function eigenValueCorner = getEigenValueFromCorner(Matrix,
   matrixSize)
2     corner = Matrix((matrixSize - 1) : matrixSize, (matrixSize
   - 1) : matrixSize);
3     % get 2 x 2 corner of the matrix
4     eigenValueCorner = solveCharacteristicEquation(corner);
5 end
```

shiftAndIterate

```
1 function [Matrix, whatIterationAreWeOn] = shiftAndIterate(
   matrixSize, Matrix, eigenValueCorner, whatIterationAreWeOn)
2     % shift and iterate algorithm
3     identityMatrix = eye(matrixSize);
4     Matrix = Matrix - identityMatrix * eigenValueCorner;
5     [Q, R] = gramSchmidtAlgorithm(Matrix);
6     Matrix = R * Q + identityMatrix * eigenValueCorner;
7     whatIterationAreWeOn = whatIterationAreWeOn + 1;
```

```
8 end
```

deflateMatrix

```
1 function [matrixSize, Matrix] = deflateMatrix(Matrix,  
2     matrixSize)  
3     matrixSize = matrixSize - 1;  
4     Matrix = Matrix(1:matrixSize, 1:matrixSize);  
5 end
```

thresholdBreached

```
1 function [flag, eigenValues] = thresholdBreached(Matrix,  
2     matrixSize, minThreshold, eigenValues)  
3     flag = 0;  
4     threshold = max(abs(Matrix(matrixSize, 1:(matrixSize - 1)))  
5         );  
6     % once we zero the row (or rather get close enough to zero  
7     % ) exit loop  
8     if (threshold <= minThreshold)  
9         eigenValues(size(eigenValues, 2) + 1) = Matrix(  
10             matrixSize, matrixSize);  
11         flag = 1;  
12     end  
13 end
```

solveCharacteristicEquation

```
1 % finds the eigenvalue of a 2x2 matrix that is closer to the  
2 % lower right corner  
3 function eigen = solveCharacteristicEquation(Matrix)  
4     [eigenOne, eigenTwo] = calculateZeros(Matrix);  
5     eigen = valueCloserToLowerRightCorner(eigenOne, eigenTwo,  
6         Matrix);
```

```

5 end
6
7 function eigen = valueCloserToLowerRightCorner(eigenOne,
    eigenTwo, Matrix)
8     if abs(Matrix(4) - eigenOne) < abs(Matrix(4) - eigenTwo)
9         eigen = eigenOne;
10    else
11        eigen = eigenTwo;
12    end
13 end

```

calculateZeros

```

1 function [zeroOne, zeroTwo] = calculateZeros(Matrix)
2     b = Matrix(1) + Matrix(4);
3     ac = Matrix(1) * Matrix(4) - Matrix(2) * Matrix(3);
4     delta = (b) ^ 2 - 4 * ac;
5     % get delta of quadratic equation
6     squareRootDelta = sqrt(delta);
7     % square delta
8     zeroOne = (b - squareRootDelta) / 2;
9     zeroTwo = (b + squareRootDelta) / 2;
10 end

```

dispResults

```

1 function dispResults(eigenValues, Matrix, whatIterationAreWeOn,
    eigenFromMatlab, initialMatrix)
2     disp(Initial matrix: );
3     disp(initialMatrix);
4     disp(Final matrix: );
5     disp(Matrix);
6     disp(Number of iterations: );
7     disp(whatIterationAreWeOn);
8     disp(Eigen values from eig(Matrix:));
9     disp(eigenFromMatlab);

```

```
10     disp(Our eigen values: );  
11     disp(eigenValues);  
12 end
```

Bibliography

- [1] Piotr Tatjewski (2014) *Numerical Methods*, Oficyna Wydawnicza Politechniki Warszawskiej