Numerical Methods, project A No. 54

Student: Piotr Skibiński

ID: 303840

Task 1

Write a program finding macheps in the MATLAB environment on a lab computer or your computer.

Definition:

$$eps = \min\{g \epsilon M : fl(1+g) > 1, g > 0\}$$

$Where$:

$fl - result\ of\ addition.$

Algorithm description:

At page 15 of book Numerical Methods by Piotr Tatjewski we can read that: "The machine epsilon eps can also be defined as a minimal positive machine floating-point number g satisfying the relation fl(1 + g) > 1, i.e". In other words to find eps we do need to obtain the smallest value for which macheps/2 + 1 is bigger than 1. Following that formula I the above code.

Important question:

Why would macheps/2 + 1 be equal to one? It all comes to the standards on which software and hardware operate to provide a solution of our problems in a real time. In MATLAB values operate on IEEE 754 standard, furthermore, each value has 64 bits to operate on. It is called double precision. With this facts the answer is rather simple. We do know that each value has limited space to work and that is why at some point program do need to round the solution. And this is why an expression x/2 + 1 = 1 can be valid in computer world.

Comments on the result:

At first glance the value looks quite strange by if we take a closer look we can tell that we have found exactly what we were looking for. As we know MATLAB operates on 64bit numbers. Those 64bits are divided into 1 bit of a sign, 11 bits of exponent and finally 53 bits of mantissa (first bit is omitted). So what is the *smallest possible value in this number? We do have 52 bits of space for mantissa so *smallest possible is 2^{-52} and this is equal to around 2.220446049250313e-16 – our output.

*Smallest – smallest positive number.

Code:

```
function macheps = Task1()
    macheps = 1;
    while 1 + (macheps/2)>1
        macheps = macheps/2;
    end
end
```

Output:

2.220446049250313e-16

Task 2

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

Code usage:

a) and b)

To run the subpoint a) or b) with a given N we do need to create a TASK2 object:

>> x = TASK2(N);

And then use the following method:

>> x = TaskA(x); <- Subpoint A

>> x = TaskB(x); <- Subpoint B

And finally display the solution and the error:

>> DispSolutionAndError(x);

For N = $10 * 2^i$ where $1 \leq i < \infty$ I do recommend to use following script:

```
function Task2bForN(n, subpoint, resCor)
    TableOfErrors = zeros(n, 1);
    TableOfNValues = zeros(n, 1);
    for i = 1:n
        TableOfNValues(i) = 10*2^(i-1);
        disp(i);
        x = TASK2(10*2^(i-1));
        if subpoint == 'A'
            x = TaskA(x);
        elseif subpoint == 'B'
            x = TaskB(x);
        end
        if resCor == 'T'
            x = ResidualCorrection(x);
            x= GetErrors(x);
        end

        TableOfErrors(i) = norm(x.errors);
    end
    plot(TableOfNValues, TableOfErrors, '-o');
    ylabel('Maximal absolute value of an error')
    xlabel('N')
end
```

For subpoint a) F- without residual correction, T – with.

>> TaskABPlot(i, 'A', ‘F or T’);

And b)

>> TaskABPlot(i, 'B', ‘F or T’);

It is going to plot the graph of a max(abs(error)) for a given N.

Output:

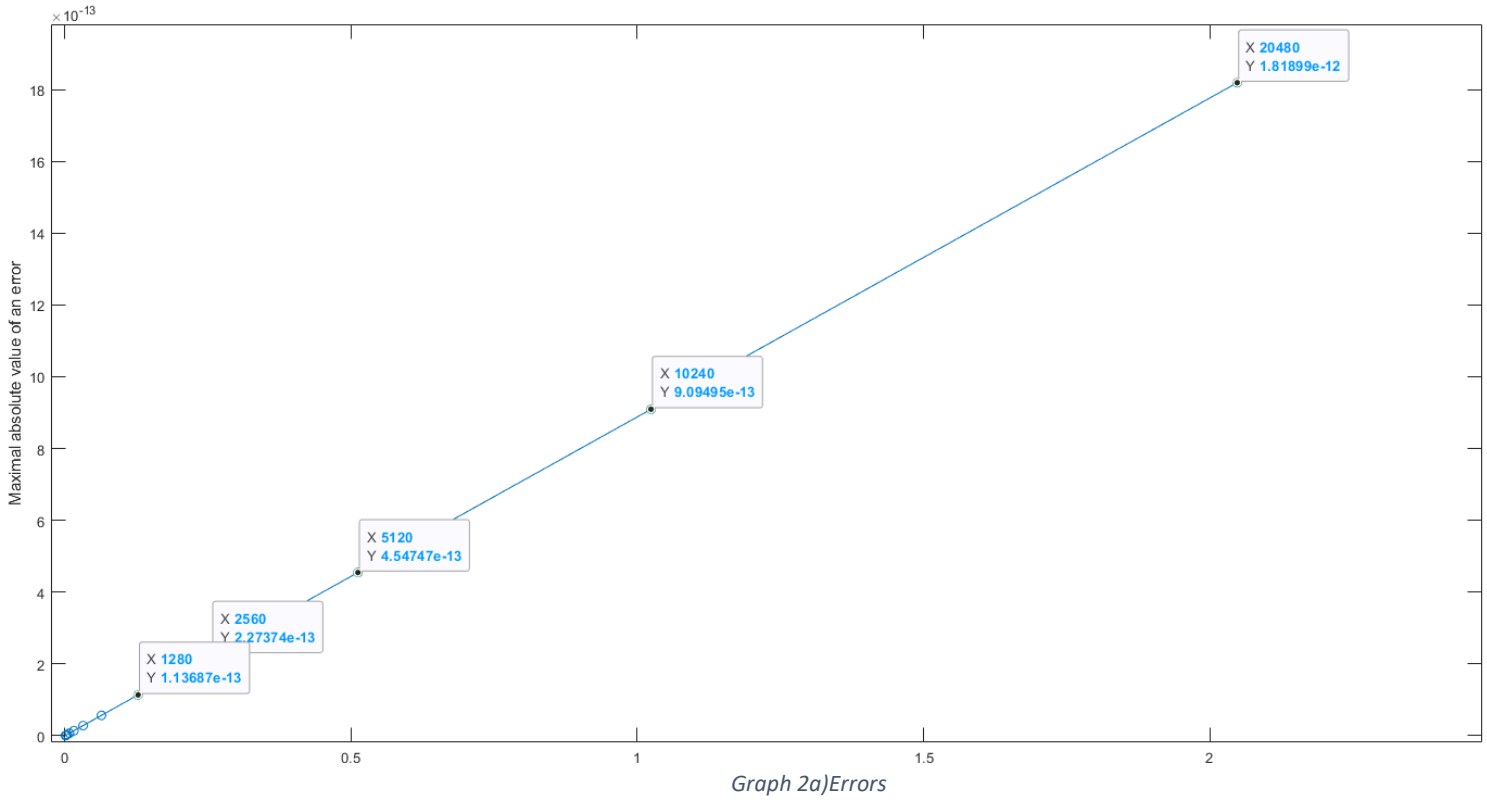
a)

For N = 10:

X	**Errors:**
0.182177616817955	0
0.157922745341646	0
0.204573460821696	0
0.227213506987843	0.444089209850063 * 1.0e-15
0.256982641467815	0
0.287592908241757	0
0.308340406746473	0
0.360300769832206	0
0.320682091298858	0
0.547482433446505	0

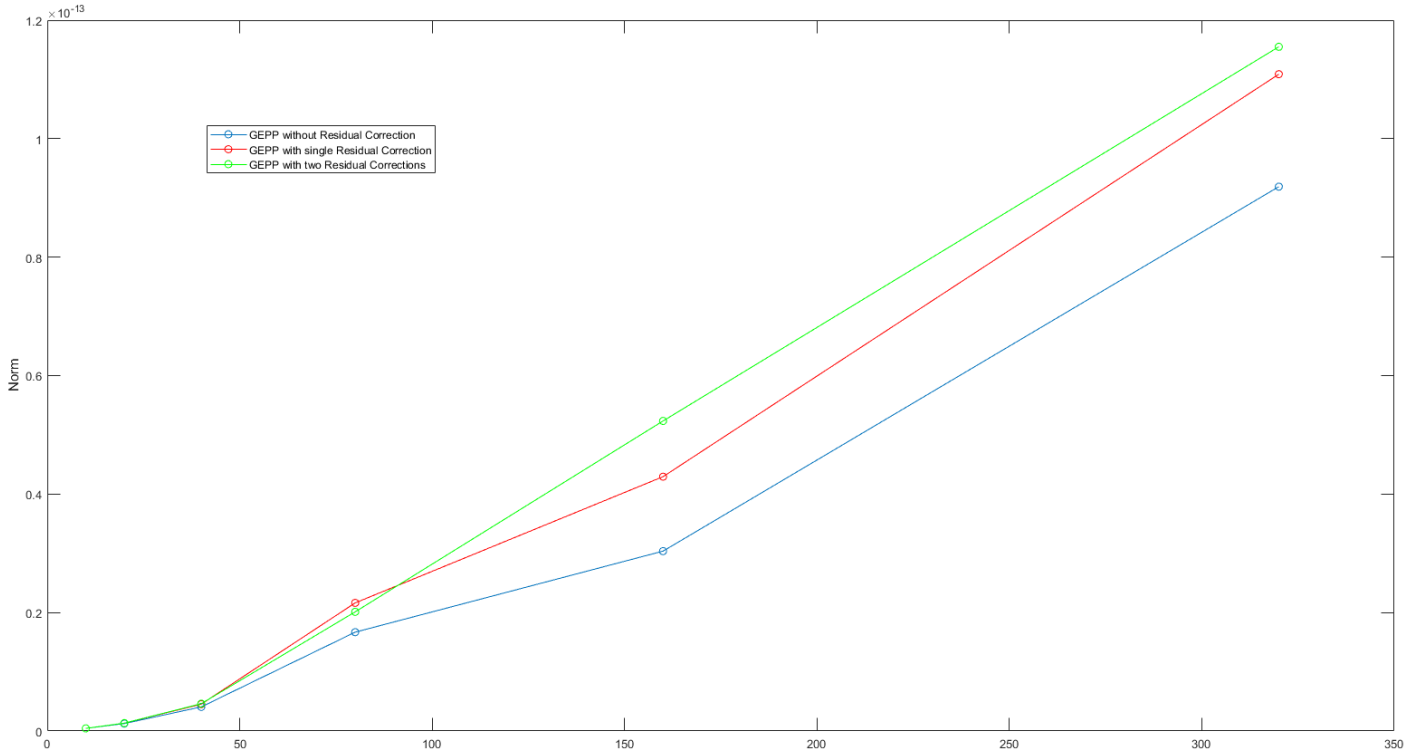
For N = 10 * 2^i (my PC could handle $1 \leq i \leq 12$) I managed to obtain following plot:

Where X is a matrix size and Y is a maximal absolute error that occurred.



Graph 2a)Errors

Errors norm with residual correction



Graph 2a)Correction

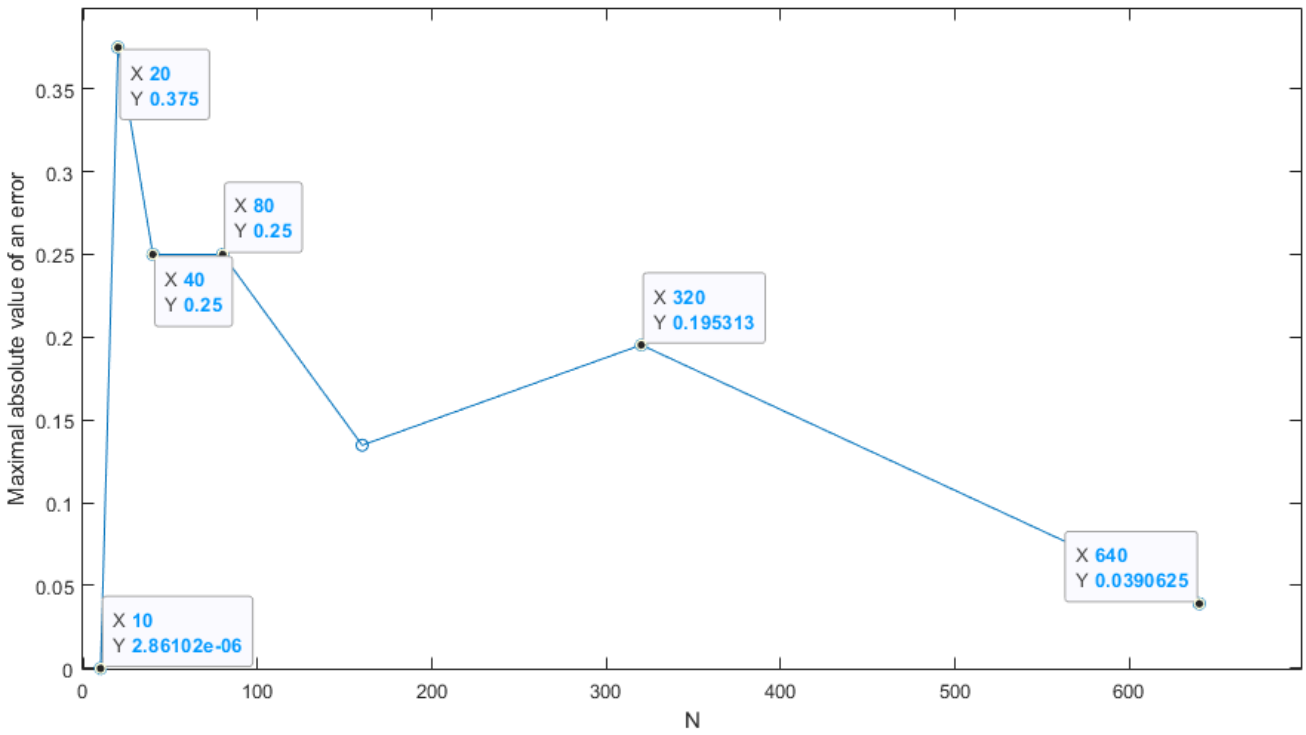
b)

For N = 10:

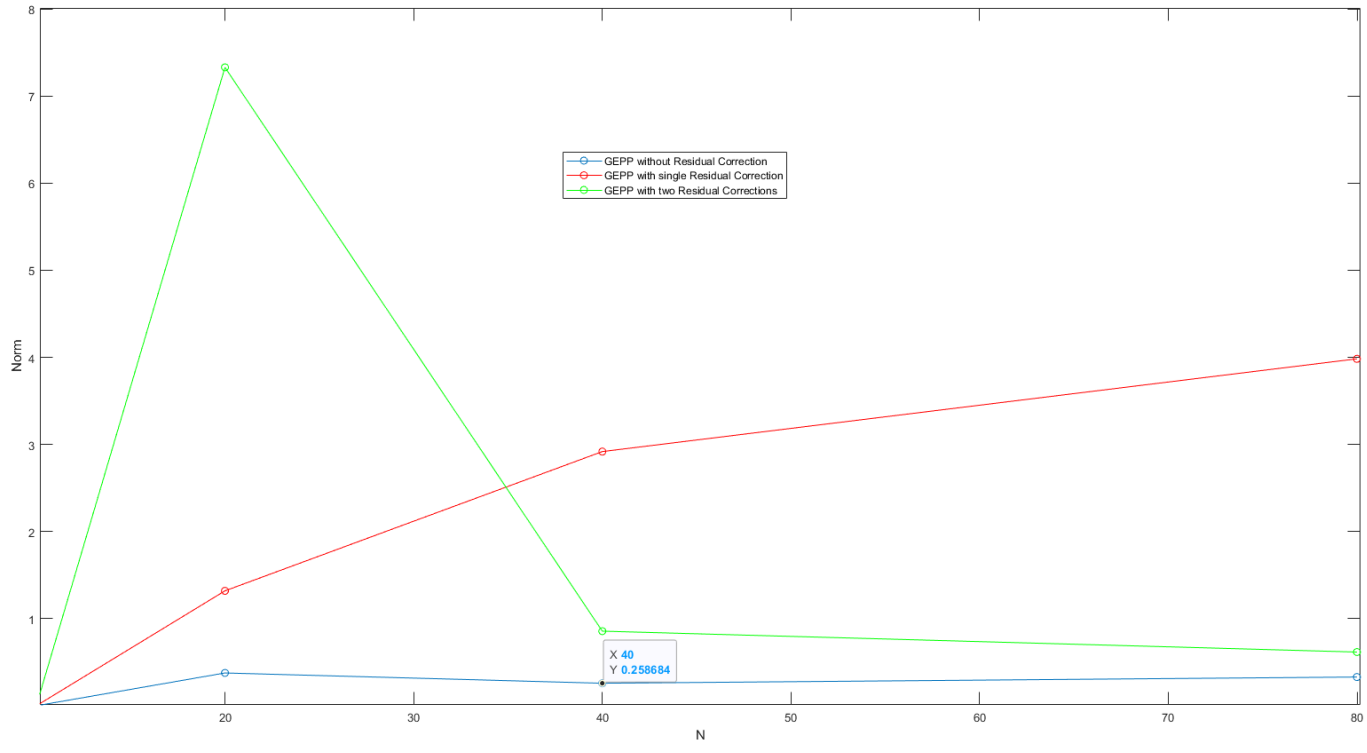
X	**Errors:**
-0.000014936619286 * 1.0e+11	0.286102294921875 * 1.0e-05
0.001286334938920 * 1.0e+11	-0.047683715820313 * 1.0e-05
-0.027348597237423 * 1.0e+11	-0.023841857910156 * 1.0e-05
0.248355787783307 * 1.0e+11	-0.000425747492816 * 1.0e-05
-1.183790878256755 * 1.0e+11	0.000757222337611 * 1.0e-05
3.252735933558811 * 1.0e+11	-0.000029335708168 * 1.0e-05
-5.335041250630115 * 1.0e+11	0.000000176963999 * 1.0e-05
5.154518080519383 * 1.0e+11	0.000000002303713 * 1.0e-05
-2.705621208265633 * 1.0e+11	0.000000000001388 * 1.0e-05
0.594931100626308 * 1.0e+11	0

For N = 10 * 2^i (my PC could handle $1 \leq i \leq 7$) I managed to obtain following plot:

Where X is a matrix size and Y is a maximal absolute error that occurred.



Graph 2b)Errors



Graph 2b) Correction

The norm function:

To plot the graph I have used a norm method. In this context it calculates the Euclidean norm which is defined as

$$[\![x]\!]_2 = \sqrt{\sum_{i=1}^{n} |x_i|^2}$$

Algorithm description:

In this part I am going to be focused mainly on a Gauss Elimination method because I do thing that generation of the table is rather obvious and I don't need to explain it. But we can have a look on 2 of the public method inside of the TASK2 class.

```
function obj = TaskA(obj)
    obj = SetSize(obj, obj.n);
    obj = TaskAArray(obj);
    obj = gaussWithPartialPiv(obj);
    obj = GetErrors(obj);
end
function obj = TaskB(obj)
    obj = SetSize(obj, obj.n);
    obj = TaskBArray(obj);
    obj = gaussWithPartialPiv(obj);
    obj = GetErrors(obj);
end
```

Those are the key method that are running the code. As I said ill skip the SetSize and TaskAArray/TaskBArray. Let's jump straight to the gaussWithPartialPiv method with is responsible for key calculations.

Firstly let's take a step back and talk about Gauss Elimination. As we know with the use of it we are not really capable of finding all of the solutions of the systems of linear equations Ax = b this is mainly because of the biggest disadvantage that Gauss Elimination have. Let's assume that we have a matrix A and currently we are executing k-th step of the elimination and we get $a_{kk}^{(k-1)} = 0$ in the next step we do need to find our $r = \frac{a_{ik}^{(k-1)}}{a_{kk}^{(k-1)}}$ with is not possible because we are going to divide by 0. So this is a part where partial pivoting comes. In Numerical Methods by Piotr Tatjewski we can read that "Every (k-th) step of the Gaussian elimination algorithm starts with a selection of the central element, but now it is the one with maximal absolute value *chosen from all elements of the k×k lower right corner submatrix of the matrix* $A^{(k)}$". In gaussWithPartialPiv It look like that:

```
%finding the maximal value of a column
index = j;
locMax = abs(obj.A(j, j));
for i = j+1:obj.n
    if abs(obj.A(i, j)) > locMax
        locMax = abs(obj.A(i, j));
        index = i;
    end
end
%checking neccessary condition for the further execution
if locMax == 0
    disp(locMax);
    disp('Wrong Matrix');
    return
```

```
            end
            if index ~= j
                %swaping the rows in neccessary (highest value found
                %not in j row)
                obj.A([index, j],:) = obj.A([j, index],:);
                obj.B([index, j]) = obj.B([j, index]);
            end
```

The only tricky part in here is that we do need to keep in mind that our local maximum cannot be equal to zero. After the pivoting part is done we can simple execute the classical Gauss elimination for k-th step. As we know the $a_{ij}^{(k)} = a_{ij}^{(k)} - \frac{a_{ik}^{(k-1)}}{a_{kk}^{(k-1)}} a_{kj}^{(k-1)}$ where j = 1, 2, …, n. i= k+1, k+2….n. and n – size of the matrix. I the case of following code I did not connect array A and array B so I do need to execute separate calculation for B. In classical algorithm we proceed n + 1 times because of that.

```
for i = j+1:obj.n
      r = obj.A(i, j)/obj.A(j,j);
      %if r == 0 we dont need to continue
      if r ~= 0
            %changing the main array A
            for locj = j+1:obj.n
                  obj.A(i, locj) = obj.A(i, locj) - r*obj.A(j, locj);
            end
            %and the B array aswell
            obj.B(i) = obj.B(i) - r*obj.B(j);
      end
  end
```

It is good to notice that in case of $a_{ik}^{(k-1)} = 0$ we do not need to calculate anything because $a_{ij}^{(k)} = a_{ij}^{(k)} - 0$. Finally after loop finishes all of the steps we can get the results thanks to the back-substitution phase.

```
        obj.x(obj.n) = obj.B(obj.n)/obj.A(obj.n, obj.n);
        for i = obj.n-1:-1:1
            buffor = 0;
            for j = i+1:obj.n
                buffor = buffor + obj.A(i, j)*obj.x(j);
            end
            obj.x(i) = (obj.B(i) - buffor)/obj.A(i,i);
        end
```

About the residual correction:

When the solution of the system is not satisfactory the following formula is satisfied:

$$Ax^{(1)} - b \neq 0$$

We should calculate the residuum $r^{(1)} = Ax^{(1)} - b$. When the absolute value of it is too large we can commit the residual correction. After the calculation of $r^{(1)}$, we do need to solve the following set $A\delta x = r^{(1)}$. Using the previously design method of factorization. After finding the δx we can calculate the $x^{(2)} = x^{(1)} - \delta x$. This correction is going to be an extremely important clue for the future error analysis. In the code method ResidualCorrection(); is responsible for RC.

Comments on the result:

a)

Graph shows how linear increase of maximum absolute error norm that occurred with a matrix of N size. 20480 x 20480 matrix generated maximal error of 1.81899e-12. This linear increase is rather natural because for higher N we do need to execute more arithmetic operations and this leads to higher round off errors. The second very important evidence is picture 'Figure 1Graph 2a)Correction' it shows the relation between an error and number of Residual Correction operations. Blue line is an entry error and in fact it has a lowest values. It turned out that for this example Residual Correction increased the errors. All of that leads to a simple conclusion errors in subpoint a) are caused by the round off error.

b)

Subpoint b) generates an interesting results. The errors are significantly higher, furthermore, they achieve a pick at around N = 20. The interpretation of the results could be different.

1. We are operating on large numbers. We do need to keep in mind that in subpoint a the results for N = 320 were around <0.1, 13> with the error of 2.8e-14 . When in the subpoint b we operate on <-2.1 * 1.0e+16, 2.1 * 1.0e+16> with the maximal absolute error of 0.195. This leads to a conclusion that b) could also be a result of round off error.

2. Unfortunately, picture 'Graph 2b) Correction' shows that proceeding an algorithm of Residual correction generated strange and unsatisfactory results. First execution generates lowest errors. Second highest average and third highest pick. In conclusion, in this example partial pivoting fails. The solution could be an application of full pivoting.

Task 3

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

Code usage:

To generate a plot of the norm of solution error just generate a TASK3:

```
>> z = TASK3;
```

And type in command:

```
>> x = Task3a(x, 'E')  or  >> x = Task3a(x, 'N')
```

Where second parameter is defined as type of the plot

'E' – plot of a maximal absolute value of x^k against the number of iterations k.

'N' – plot of a norm of the solution errors = Ax – b against the number of iterations.

For generation of the results from task 2 a) b) following method is needed:

```
>> z = Task3With2a(z, input, type); or  >> z = Task3With2b(z, input, type);
```

Where input is an algorithm which is going to be use in the calculations:

'G' – Gauss-Seidel algorithm.

‘J’– Jacobi algorithm.

‘B’ – Both methods.

Type Is a type of the plot (same as previously) ‘E’ and ‘N’

Definitions:

The Jacobi method

When it comes to Jacobi Method two assumptions are made:

1. We need a system of linear equations

$$a_{11}x_1 + a_{12}x_2 + \cdots a_{1n}x_n = b_1$$
$$a_{21}x_1 + a_{22}x_2 + \cdots a_{2n}x_n = b_2$$
$$\dots$$
$$a_{n1}x_1 + a_{n2}x_2 + \cdots a_{nn}x_n = b_n$$

2. Following matrix has no zeros on it main diagonal, $a_{11}, a_{12}, \dots, a_{nn}$

The algorithm.

Firstly we do need to rewrite the equation in a following way:

$$x_1 = (b_1 - a_{11}x_2 - a_{12}x_3 - \cdots a_{1n}x_n)/a_{11}$$

$$x_2 = (b_2 - a_{21}x_1 - a_{12}x_3 - \cdots a_{1n}x_n)/a_{22}$$

$$\dots$$

$$x_n = (b_n - a_{n1}x_1 - a_{n2}x_2 - \cdots a_{nn-1}x_{n-1})/a_{nn}$$

Secondly, we do need to make initial guess of the solution of out equation by rewriting the solutions:

$$x^0 = (x_1, x_2, x_3, \dots, x_n) \rightarrow x^1 = x^0$$

Those are the all steps of a single iteration.

So **Jacobi** method can be defined as:

For each k ≥ 1, we generate the component x^k

$$x_i^k = \frac{1}{a_{ii}}\left[\sum_{j=1}^{n} (-a_{ij} * x_j^{(k-1)}) + b_i\right], where\ i = 1, 2, 3, \dots, n-1, n$$

The Gauss-Seidel method

The difference between the Gauss-Seidel and Jacobi method is rather straightforward. With the Jacobi method the values of x_i^k are unchanged until the entire iteration has been calculated. When with the Gauss-Seidel method we don’t use the x_i^{k+1}as soon as it is possible, but when it is possible obviously when they are calculated.

So the definition of Gauss-Seidel method will look like that:

$$x_i^k = \frac{1}{a_{ii}}\left[-\sum_{j=1}^{i-1} (a_{ij} * x_j^{(k)}) - \sum_{j=i+1}^{n} (-a_{ij} * x_j^{(k-1)}) + b_i\right], where\ i = 1, 2, 3, \dots, n-1, n$$

Comments of the code:

Using aforementioned formulas I was able to create a properly working core but I feel like I do need to highlight some things.

1. **Stop tests**
   Those are the criteria that needs to be check in every iteration of the code. I am using two of them. The first one is checking the differences between two subsequent iteration points: Defined as: $[\![x^{(i+1)} - x^{(i)}]\!] \leq \delta$ where δ is a accuracy defined by the user under obj.acc element and $[\![x^{(i+1)} - x^{(i)}]\!]$ is a variable err

```
while  k <= m && err > obj.acc
```

   the error is afterwards defined by

```
if err > localMax
      err = localMax
```

   where localMax keeps the track of each local $[\![x^{(i+1)} - x^{(i)}]\!]$ for possible plots.

   The second criteria checks the number of iterations. The bound is set to 10000 under the m variable and can be changed by the user. With this protection code is not going to run into an infinite loop.

2. **Used build in methods.**
   The 63 line of the code uses the build in MATLAB function norm(), In this context it calculates the Euclidean norm which is defined as

$$[\![x]\!]_2 = \sqrt{\sum_{i=1}^{n} |x_i|^2}$$

Output:

For the given linear equation the results are as follows:

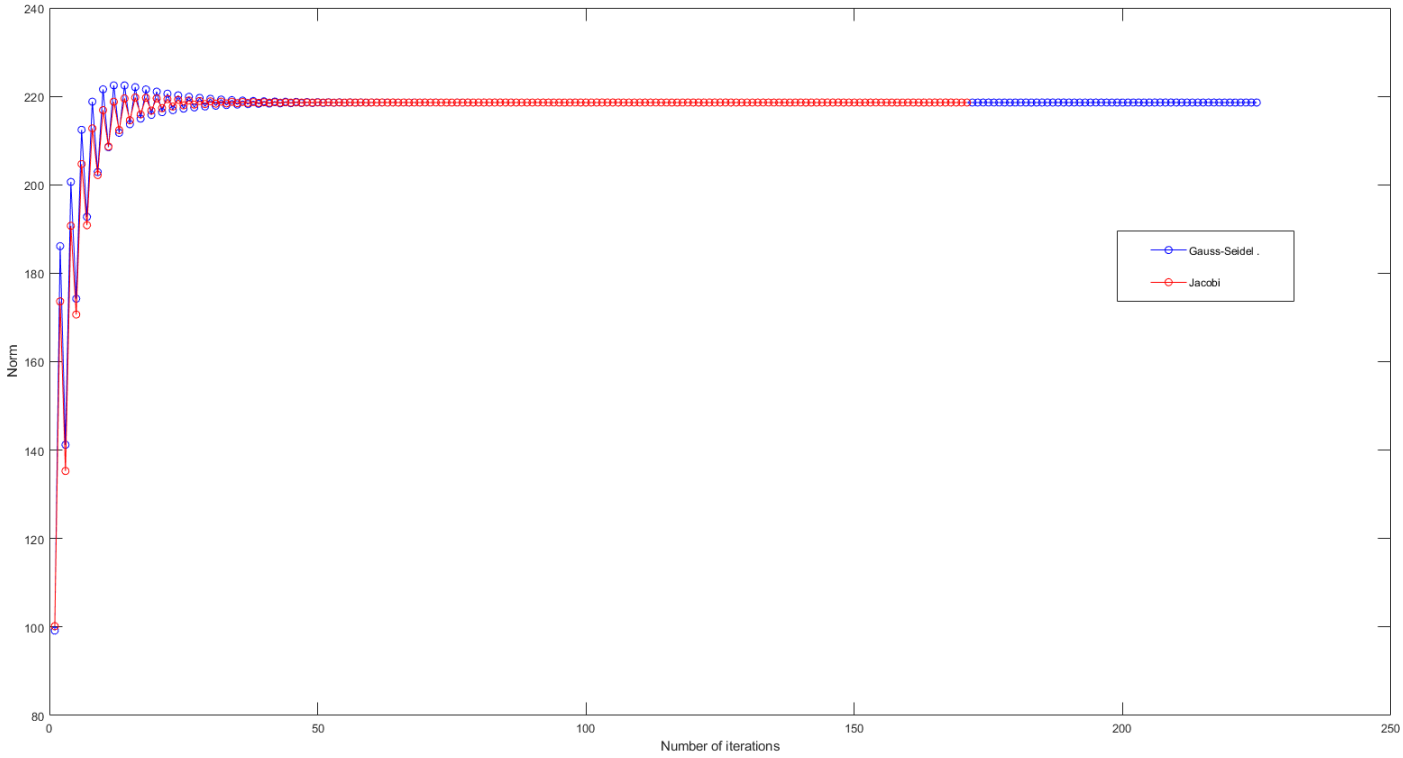
With the use of Gauss-Seidel

x_1	-9.485462555046253
x_2	9.464904552100087
x_3	-16.327165932482728
x_4	-10.236123347972185
iterations: 225	

With the use of Jacobi

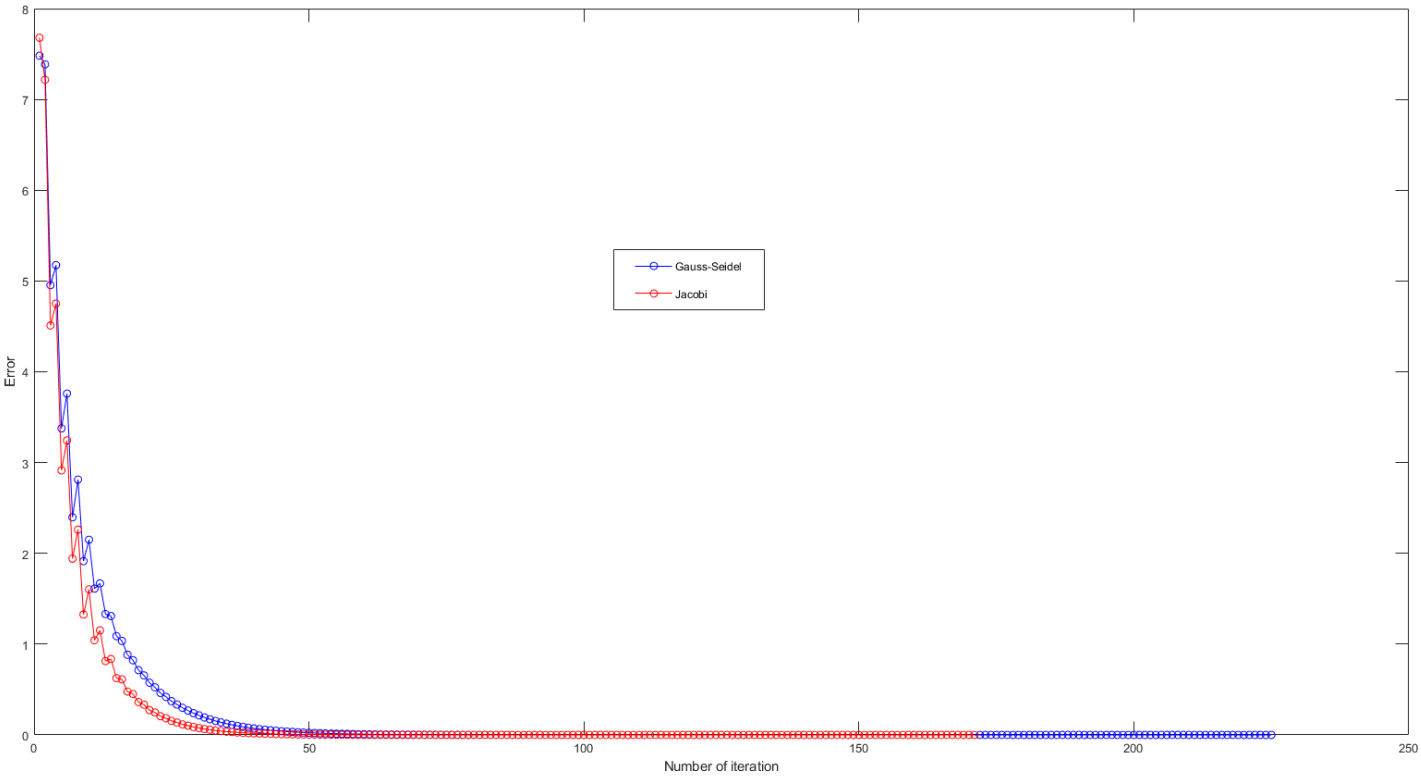
x_1	-9.485462555055488
x_2	9.464904552104757
x_3	-16.327165932479499
x_4	-10.236123347974612
iterations: 171	

Following example generated a norm plot



Graph 3 norm.

I have also decided to include a graph of local maximum error max ($[\![x^{(i+1)} - x^{(i)}]\!]$)

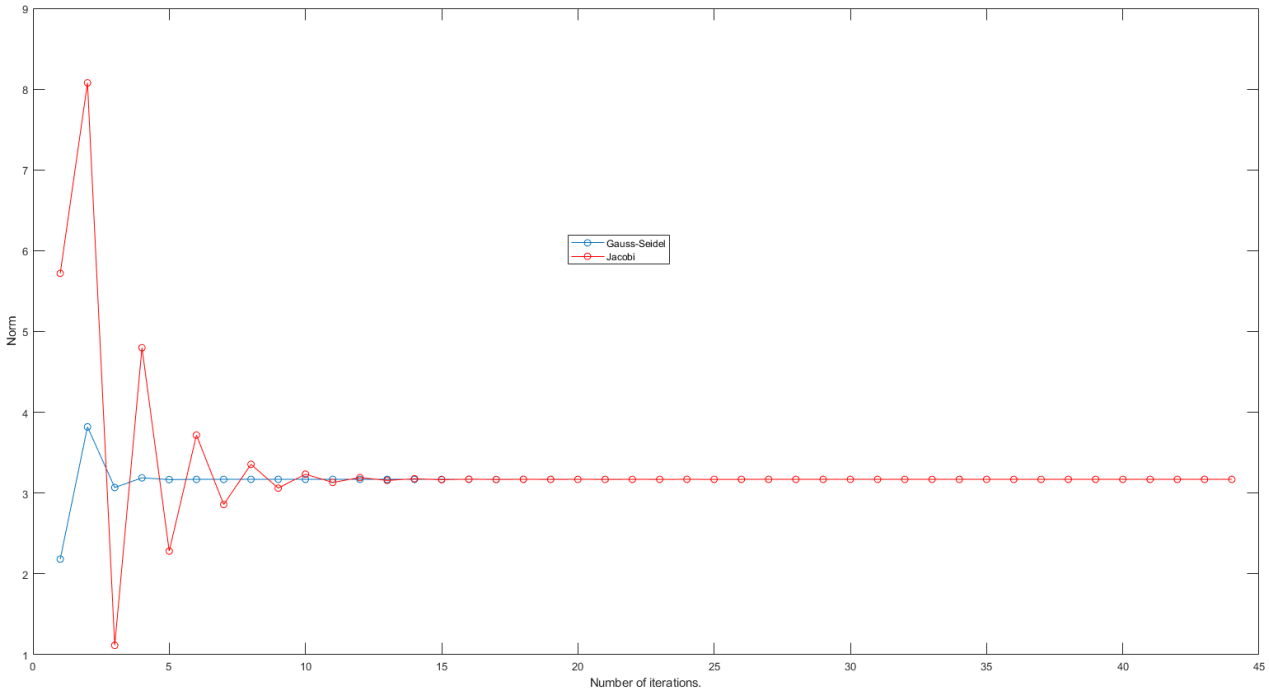


Graph 3 maximal local error.

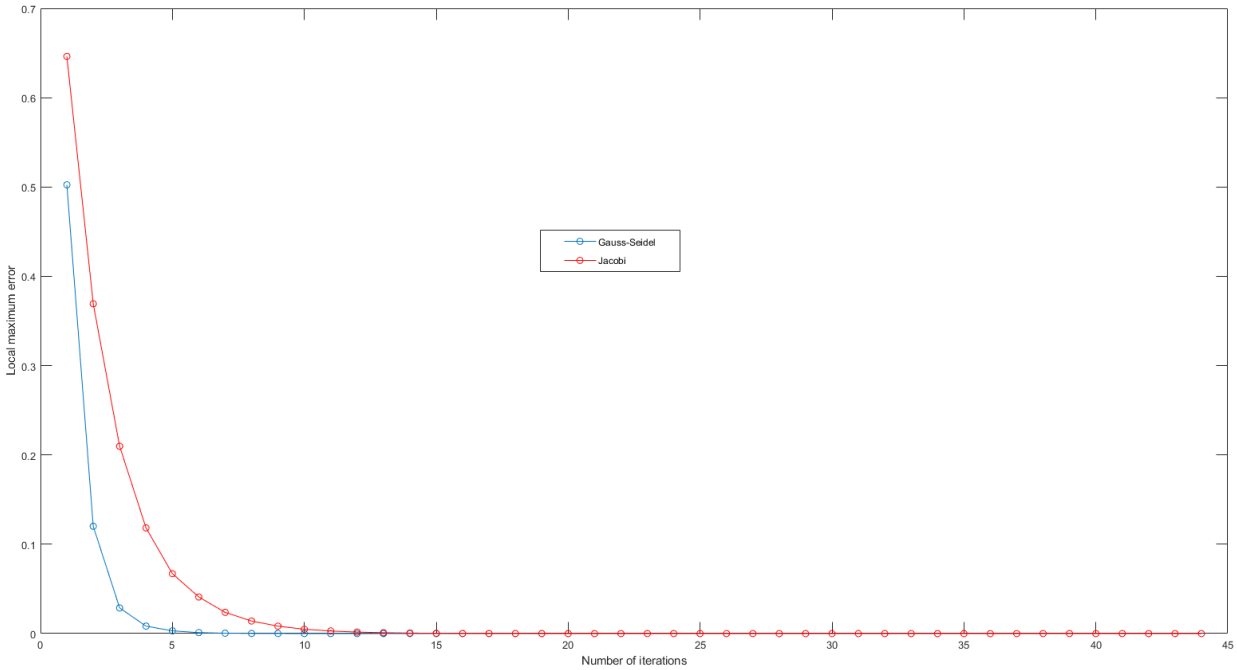
When it comes to solving the equations from task 2 I have got:

In a)

Gauss-Seidel and Jacobi



Graph 3 with 2a) norm.

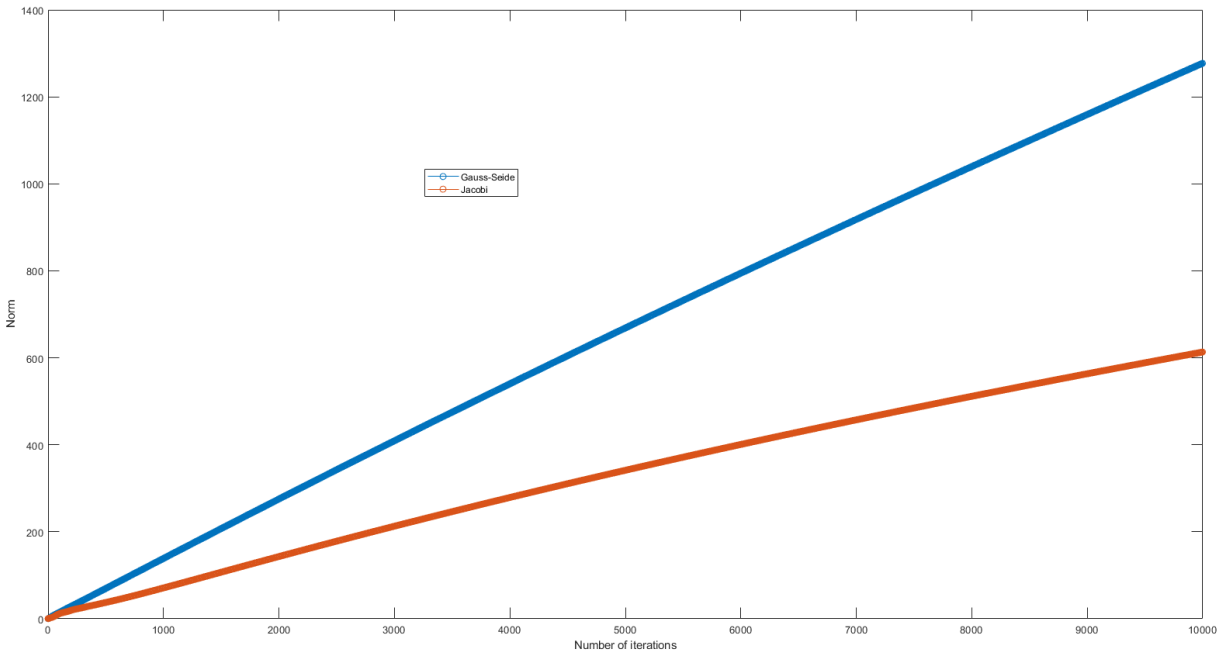


Graph 3 with 2a) maximal local error.

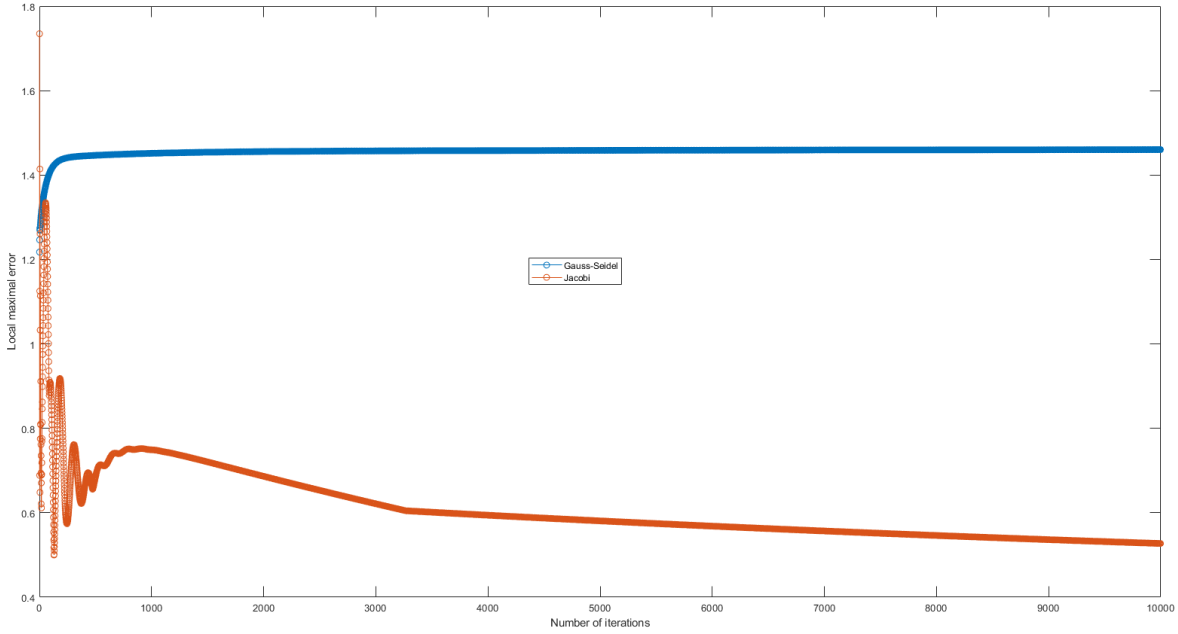
With the 23 iterations for Gauss Seidel and 44 for the Jacobi.

In a subpoint b)

Both methods fails by triggering the 10000 iterations lock.



Graph 3 with 2b) norm.



Graph 3 with 2b) local maximal error.

Comments of the results & a bit more about Gauss-Seidel and Jacobi convergence:

For a give linear equation Jacobi method tuned out to be on top with the 171 iterations vs the Gauss-Seidel 225. In general Gauss-Seidel method is more efficient than the Jacobi but this example shows a bit different result. To understand the phenomenon here we need to take a closer look on how and when those methods converge:

The *sufficient convergence condition* for the Jacobi method is the strong diagonal dominance of a matrix.

$$|a_{ii}| > \sum_{j=1, j\neq i}^{n} |a_{ij}|, i = 1, 2, \dots, n - row\ strong\ dominance.$$

$$|a_{jj}| > \sum_{i=1, i\neq j}^{n} |a_{ij}|, i = 1, 2, \dots, n - column\ strong\ dominance.$$

When it comes to Gauss-Seidel method it is convergent when a *sufficient converge condition* or the matrix A is a *positive definite* when:

$$A_x > 0\ for\ all\ x \in R \setminus 0.$$

At page 59 of Numerical Methods by Piotr Tatjewski we can read that: “The Gauss-Seidel method is usually faster convergent than the Jacobi’s method (if both are convergent, making a comparison possible).”

Theory vs the results:

So I have made the comparison and in the case of **task 3** linear equation Jacobi method is faster. It could be because of the *sufficient converge condition* which is satisfied for most od the rows and columns, furthermore, matrix is not positive definite. This leads to the better efficiency of a Jacobi algorithm.

When is comes to **task 3 with 2 a) matrix** the Gauss-Seidel executes with the significantly lower amount od iterations. It is good to mention that this matrix is not only a *strong diagonal dominant* but it is also a *positive definite.* In my opinion this is some sort od a sweet point for Gauss-Seidel method and it dominated the Jacobi method in this matrix.

In the **task3 with b) matrix** both methods failed. It is a result of a main A matrix not being a diagonally dominant. As I previously mentioned the Jacobi and Gauss-Seidel in some cases is divergent I do believe that this case that problem have occurred.

Task 4

Write a program of the QR method for finding eigenvalues of 5×5 matrices:

a) without shifts.

b) with shifts calculated on the basis of an eigenvalue of the 2×2 right-lower-corner submatrix.

Definitions:

The orthogonal-triangular (QR) factorization: can be executed in every matrix

$A_{mxn} where\ A \in R^{mxn}, m \geq n$ with all columns linearly independent. An algorithm make a decomposition of a $A\ into\ a\ QR$ where Q is a orthogonal matrix Q and R is a upper triangular matrix. The orthogonal matrix is a real square matrix (a matrix with the same amount of rows and columns) whose rows and columns are orthonormal vectors. A the case of a square matrix Q is called an orthogonal matrix if $Q^T Q = I$ where T is the transpose of a matrix and I - is the identity matrix. R is an upper-triangular matrix with unity elements on the diagonal.

Modified Gram-Schmidt and classical Gram-Schmidt algorithm.

In algorithm to obtain the A = QR the columns Q are obtained by successively orthogonalizing the columns of A. The classical Gram-Schmidt as well as the modified Gram-Schmidt do actually share the property that the calculated matrices Q and R a specific bounds defined as:

$$\left||A - QR|\right|_2 \leq \gamma \left||A|\right|_2$$

Where γ is a constant that depends on a matrix size. The difference between those two algorithms lies is an ability of orthogonalizing the individual columns of A. The only very important thing to mention is that Gram-Schmidt procedure can suffer from numerical instability. Round-off errors can accumulate at some point and destroy the further orthogonality of the resulting vectors. That's is why I am going to use modified Gram-Schmidt algorithm. The code will be simplified be the lack of transformation to the upper Hessenberg form. It is traditional for the symmetric matrices and those are given into the task.

About the eigenvalues

Those are a extremely important in the computer science and we are going to use them to provide the solution. An eigenvalue and a corresponding eigenvector A_n are defined as a pair of the following numbers $\lambda \in \mathbb{C}$ and a vector $v \in \mathbb{C}^n$

$$Av = \lambda v$$

Where λ – is an eigenvalue, v – is a corresponding eigenvector. A square n-dimensional matrix has n eigenvalues.

Here come the shifts

In is a very important step to use shifts. The convergence rate of an algorithm without them is linear with a ratio of $\left|\frac{\lambda_{i+1}}{\lambda_i}\right|$. The method can be really slow if certain eigenvalues have similar values.

An algorithm of a QR factorization with shifts:

Firstly, the eigenvalue λ_n need to be found as close to the $d_n^{(k)}$ $eigenvalue\ as\ it\ is\ possible$.

Secondly, the last column of the actual matrix $A^{(k)}$are deleted and then only the submatrix $A_{n-1}^{(k)}$is taken into account.

$$A_{n-1}^{(k)} = \begin{pmatrix} d_1^{(k)} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \begin{bmatrix} d_{n-2}^{(k)} & e_{n-2}^{(k)} \\ e_{n-2}^{(k)} & d_{n-1}^{(k)} \end{bmatrix} \end{pmatrix}$$

Thirdly, the next eigenvalue $\lambda_{n-1}$is being found using the following formula. Matrix $A_{n-1}^{(k)}$ is regenerated using the QR procedure until $e_{n-2}^{(k)} = 0$. But when it comes to full triadiagonal matrix transformations, an algorithm is performed until all of the elements expect for the diagonal ones are nonzero.

Code usage:

a) and b)

To run the subpoint a) or b) with a given N we do need to create a TASK4 object:

```
>> x = TASK4;
```

And then use the following method:

```
>> x = SetExample(x); <- Subpoint A
```

```
>> x = SetExampleShifts(x); <- Subpoint B
```

Output:

For the following matrix:

2	**33**	**8**	**-3**	**4**
33	**1**	**-6**	**5**	**-3**
8	**-6**	**-5**	**-6**	**8**
-3	**5**	**-6**	**3**	**2**
4	**-3**	**8**	**2**	**45**

Method without shift generated 821 iterations while method with shifts only 18.

eigenvalues

Without shifts	**With shifts**
46.870193049792448	-8.475509850986441
-35.205728779511318	-35.205728779511759
34.511143445015094	8.299902135690195
-8.475509850986404	34.511143445015563
8.299902135690152	46.870193049792455

QR factorization without shifts in the last iteration gave a following output:

46.8702	**0**	**0**	**0**	**0**
0	**-35.2057**	**0**	**0**	**0**
0	**0**	**34.5111**	**0**	**0**
0	**0**	**0**	**-8.4755**	**0**
0	**0**	**0**	**0**	**8.2999**

A QR factorization with shift we obtained the following matrix

…

↓

-35.2057	**-0.0154**	**0***
-0.0154	**-8.4755**	**0***
0*	**0***	**8.2999**

↓

-8.4755	**0***
0*	**-35.2057**

*zero here is a value which is smaller than a given precision (in our task 10^{-6})

Comments on the results:

Aforementioned results show how important using shifts can be in a eigenvalues calculations. An algorithm with shifts managed to output the results satisfying the given precision with only 18 iterations while method without them had to proceed 821 of iterations.

The end.

Sources: lectures and 'Numerical Methods' by Piotr Tatjewski.