Chmielewska Urszula

300167

Numerical Methods

Project A No. #15

Advisor: Dr Adam Krzemienowski

TASK 1.

The program which allows to find machine epsilon value.

Theoretical background

Machine epsilon is the maximal possible relative error of the floating-point representation. To obtain the value of it, we need to analyze the theory of floating point representation of real numbers.

To obtain machine epsilon it is necessary to provide with theoretical background of real number **x** representation as a floating point $\mathbf{x_{t,r}}$**.** It looks as follows:

$$x_{t,r} = m_t \cdot P^{Cr}$$

where:

m_t – mantissa

c_r – exponent

P – base

t – number of positions in the mantissa

r – number of positions in the exponent

Mantissa should be normalized to get a properly defined floating-point representation. The most common assumption for the normalization is that

$$0.5 \leq |m_t| < 1$$

Another convention of normalization is used in IEEE 754 standard and it assumes that

$$1 \leq |m_t| < 2$$

In any used convention, 1 is always the first position of any normalized mantissa. However it is necessary to understand the difference between these two assumptions. For a given example of $\mathbf{m_4=1011}$, in the IEEE 754 standard, the point separating the fractional part of a number is placed after the first element of the mantissa, so it corresponds to **1.011**. On the contrary, in the first normalization the m_4 corresponds to **0.1011**.

We want to obtain the most exact floating-point representation what is possible it the following condition is satisfied

$$|rd(x) - x| \leq min_{g\epsilon M}|g - x|$$

Where **rd(x)** is the floating-point representation of number **x**, for **P=2**.

As I stated at the beginning of the theoretical introduction, to assess the machine epsilon value, we need to define, universally true, equation for a relative error, which is

$$\left|\frac{rd(x) - x}{x}\right| = \frac{|m_t - m|}{|m|} \leq \frac{2^{-(t+1)}}{2^{-1}} = 2^{-t}$$

However there exists another method to define a machine epsilon. It can be assessed as a minimal positive machine floating-point number g satisfying the relation $\boldsymbol{fl(1 + g) > 1}$, i.e.,

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

It is necessary to explain why we are looking for $\boldsymbol{fl(1 + g)}$ in the area of value 1. Technically it means, that epsilon value is the smallest possible value which we can add to 1, using the calculator, and as a result obtain a value minimally greater than 1, instead of a rounding to exactly 1.

Algorithm

Principally, the program finds the smallest possible number for which adding machine epsilon to 1, will show a number with slightly higher value than 1. If the obtain sum will be equal to 1 it means that the machine epsilon value was not correct and the program rounded it to 0.

The above description, in Matlab's code, looks like that

```
macheps = 1.0;
while((1.0 + (macheps/2.0)) > 1.0)
    macheps = macheps/2.0;
end
```

Results

Calculated machine epsilon:

2.220446049250313e-16

Matlab's machine epsilon value:

2.220446049250313e-16

Conclusions

The built-in Matlab's function eps gives the same result as the program which I wrote. When we perform many operations, relatively small errors accumulate and may cause looses of some data. Machine epsilon is useful for making an explanations why in the process we did not obtain exact solutions and the solution errors are high.

TASK 2.

General program which solves a system of **n=10,20,40,80,160,…** linear equations **Ax=b** using Gaussian elimination with partial pivoting. Only elementary mathematical operations on numbers and vectors are allowed. The program is applied for a given matrix A and vector b, until the solution time becomes prohibitive, for:

a) $a_{ij} = \begin{cases} 9 & for\ i = j \\ 1 & for\ i = j = 1\ or\ i = j + 1, \\ 0 & other\ cases \end{cases} \qquad b_i = 1.4 + 0.6i, \qquad i, j = 1, \dots, n;$

b) $a_{ij} = \frac{3}{[a(i+j-1)]}, \qquad b_i = \frac{1}{i}, i - odd;\ b_i = 0, i - even, \qquad i, j = 1, \dots, n.$

For each case, calculate also the solution error defined as the Euclidean norm of the vector od residuum r = Ax-b, where z is the solution. Plot error vs. n. For n=10: print the solutions it's errors, make the residual correction and check if it improves the solutions.

Theoretical background

The Gaussian elimination consists of two steps:

1) The Gaussian elimination phase
2) The back-substitution phase

Step 1

The goal of this step is to convert the system of linear equations Ax = b to an equivalent system with an upper-triangular matrix

Firstly, we have to define row multipliers

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

With such definition, we zero the elements, below the diagonal, by subtracting from each of the rows w_i the *k-th* row w_k multiplied by the *k-th* row multiplier l_{ik}

$$w_i = w_i - l_{ik} w_k$$

After such subtraction we obtain:

$$a_{11}^{(1)} x_1 + a_{12}^{(1)} x_2 + \cdots + a_{1n}^{(1)} x_n = b_1^{(1)},$$

(equation to be finish)

Step 2

The goal of this step it to solve the upper-triangular system of equations. We can obtain this, by starting solving from the last row. Then solve the second to last using the obtained x value from the last equation and so on.

However, it may happen that algorithm cannot continue, for example if some $a_{kk}^{(k)} = 0$. This can be omitted by using partial or full pivoting.

In partial pivoting we firstly have to choose a central element $a_{jk}^{(k)}$, which is the biggest absolute value from a row

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

Secondly, we interchange the *i-th* and the *k-th* row and we have to perform again the matrix transformation as previously. Such approach in solving each step of the algorithm is preferable, because it leads us to a smaller numerical errors.

Residual correction

Can be obtained in three steps. It is used to obtain the most accurate solutions.

1) Calculate the residuum: $r^{(1)} = Ax^{(1)} - b$
2) Solve the set: $A\delta x = Ax^{(1)} - b$, to obtain a corrected solution: $x^{(2)} = x^{(1)} - \delta x$
3) Calculate the residuum: $r^{(2)} = Ax^{(2)} - b$

If the $r^{(2)}$ is still too large, then the procedure is repeated to obtain the closest to the real value, approximated solution.

a) Results

Solution and the solutions' error (Euclidean norm of the residuum vector)

```
x =

   0.1961

   0.2348

   0.2911

   0.3454

   0.4000

   0.4546

   0.5090

   0.5647

   0.6090

   0.7546


residuumEuclideanNorm = s1.2561e-15
```

Solution x after the correction of the residuum and the corrected Euclidean Norm

```
newx =

   0.1961

   0.2348

   0.2911

   0.3454

   0.4000

   0.4546

   0.5090

   0.5647

   0.6090

   0.7546


finalResiduumEuclideanNorm = 0
```

Plot of the error (Euclidean norm of the residuum) vs. number of equations n

Plot for the 8 iterations of n (where in following iterations n = n*2)

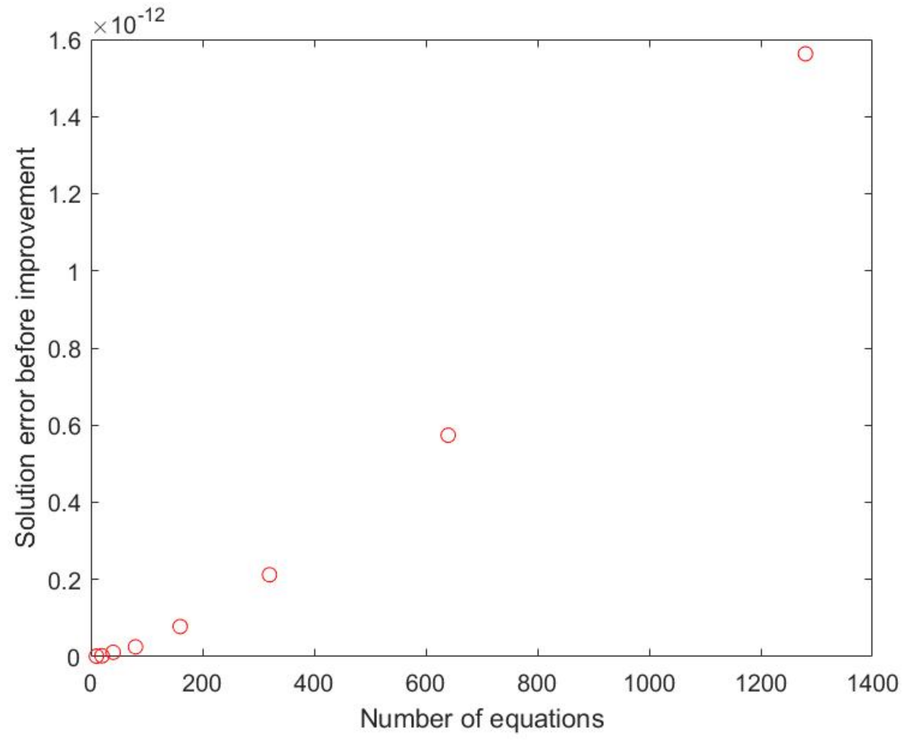

Fig. 1 Plot of the error vs. number of equations in a)

Plot of the Euclidean norm of the corrected residuum for 10x10 matrix

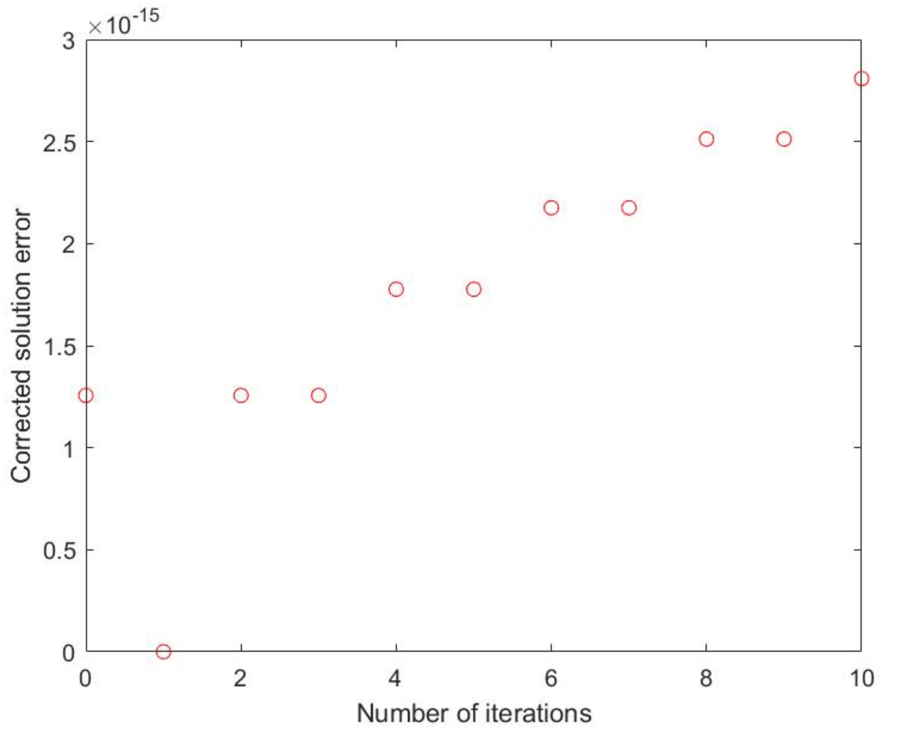

Fig. 2 Plot of the corrected error vs. number performed iterations in 10x10 matrix

b) Results

Solution and the solutions’ error (Euclidean norm of the residuum vector)

```
x =

 1.0e+12 *

   -0.0000

    0.0003

   -0.0058

    0.0530

   -0.2525

    0.6939

   -1.1381

    1.0996

   -0.5772

    0.1269


residuumEuclideanNorm = 2.2978e-05
```

Solution x after the correction of the residuum and the corrected Euclidean Norm

```
newx =

   1.0e+12 *

   -0.0000

    0.0003

   -0.0058

    0.0530

   -0.2525

    0.6939

   -1.1381

    1.0996

   -0.5772

    0.1269


finalResiduumEuclideanNorm = 1.2902e-05
```

Plot of the error (Euclidean norm of the residuum) vs. number of equations n

Plot for the 8 iterations of n (where in following iterations n = n*2)

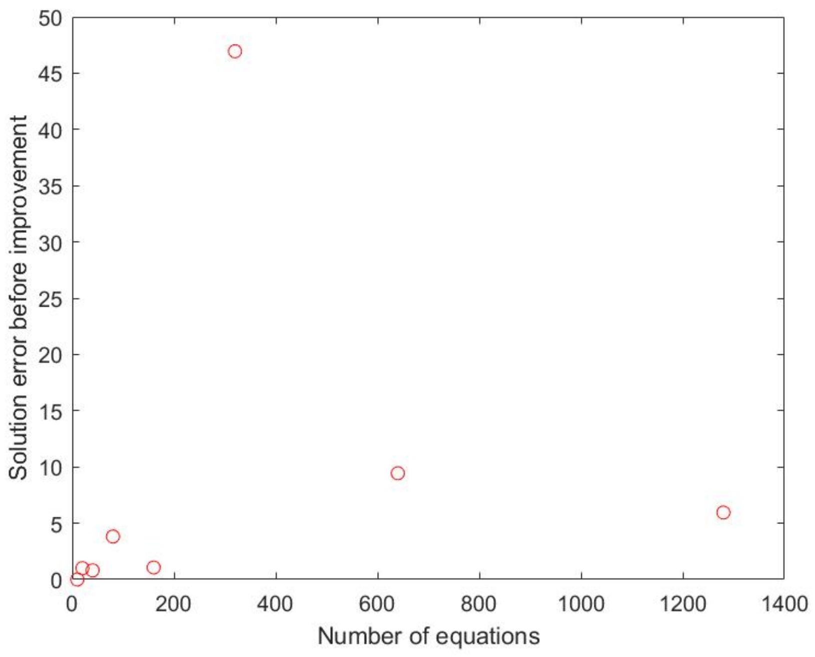

Fig. 3 Plot of the error vs. number of equations in b)

Plot of the Euclidean norm of the corrected residuum for 10x10 matrix

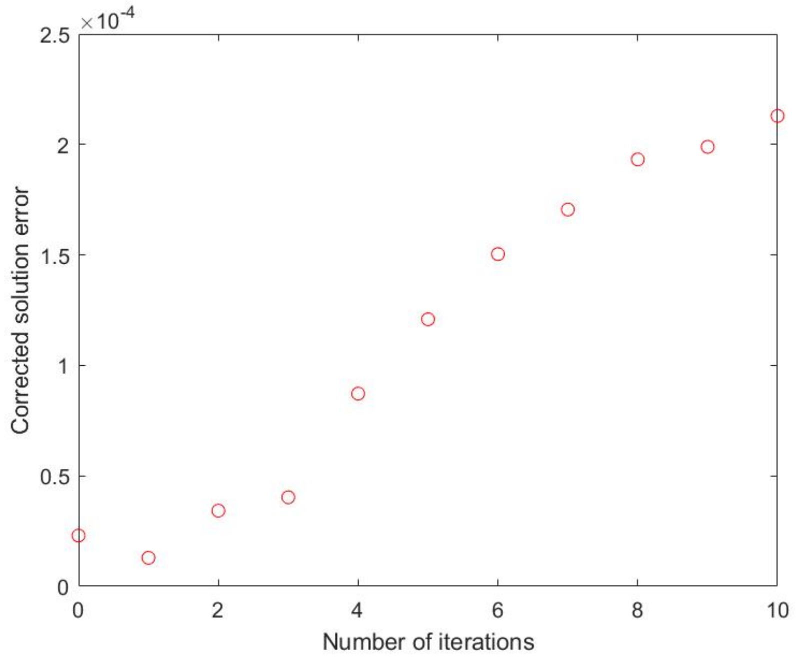

Fig. 4 Plot of the corrected error vs. number performed iterations in 10x10 matrix

Conclusions

Using the build in Matlab function linsolve(A,b) I obtained the same results of x's in both cases. Also in both subsections I did not obtain an improved solution of x's for n = 10. The results stayed the same. Results obtained in the task a) and b) act in a totally different way.

From the observations I can expect that the problem with task b) is that this problem is ill-conditioned. The matrix is said to be ill-conditioned when its condition number is very large.

As expected, I obtained such solutions for 10x10 matrixes:

a) cond(A) = 1.5420
b) cond(A) = 1.6025e+13

where the formula for calculation the condition number of a matrix is:

$$cond(A) = \|A^{-1}\|\|A\|$$

Ill-conditioned problem means that relatively small perturbations in the data values, results in relatively high changes in the value of the result. To check if the problem is ill-conditioned it is enough to check it's condition number, because this number characterizes the increase of the relative error of the result vs the relative error of the data.

I plotted and performed calculations for 8 iterations of the equation number n (so for 1280 linear equations) because the method failed for the higher number. This method is not efficient for large number equations. It already took a while to calculate 8 iterations. It is better to use iterative methods for solving systems with large number of equations.

On the Fig. 2 and Fig. 3 I presented the changes of the Euclidean norm of the corrected residuum for 10x10 matrixes. From the Fig. 2 we can conclude that there is a correction in the first iteration, however then the solution error is equal to 0. From such results we can conclude that the obtained error value was smaller than the Machine Epsilon, so Matlab rounded it to 0. In the next iterations, as it is presented on the graph, we did not obtain any better result of the corrected residuum, comparing to the 0'th obtained value. Due to this observations, we also should not expect any improvements in obtained values of x'es.

On the Fig. 3, we can observe that the first iteration gave us an improved result of the Euclidean Norm of the residuum. However this improvement did not have a significant impact on the results of x.

I checked also how the correction of the residuum acts in the case of matrixes with the higher amount of equations for the task 2a). In the 20x20 matrix, the best residuum was also this obtained in the 0'th iteration. However in the 40x40, 80x80 and bigger matrixes, in the first iteration, we can observe a real corrections of the Euclidean Norm of the corrected residuum.

In the task 2b), for the increasing number of equation in matrixes, I did not observe any significant changes, in the comparison to the observations discussed above.

TASK 3.

The program solves the system of n linear equations Ax = b using Gauss-Seidel and Jacobi iterative algorithm. It plots the norm of the solution error $||Ax_k - b||_2$ vs. the iteration number k=1, 2, 3 until the accuracy $||Ax_k - b||_2 < 10^{-10}$ is achieved.

Applied for the system:

$8x1 + 2x2 - 3x3 + x4 = 7$

$2x1 - 25x2 + 5x3 - 18\,x4 = 12$

$x1 + 3x2 + 15x3 - 8\,x4 = 24$

$x1 + x2 - 2x3 - 10\,x4 = 28$

Theoretical background

Jacobi's method

Firstly we have to decompose a given matrix A into:

$$A = L + D + U$$

L – subdiagonal matrix

D – diagonal matrix

U – matrix with entries over the diagonal

Given system of equations as Ax = b, we can rewritten as:

$$Dx = -(L + U)x + b$$

If we make an assumption that matrix D is nonsigular, we can write such iterative algorithm which is known as the Jacobi's method:

$$Dx^{(i+1)} = -(L + U)x^{(i)} + b, \qquad \text{for } i = 0, 1, 2, \ldots.$$

Jacobi's method is called a parallel computational scheme.

Gauss-Seidel method

As previously, we have to decompose matrixes at first and secondly write the system of given equations as:

$$(L + D)x = -Ux + b$$

We again use the assumption that D is nonsingular, and write the following iterative method which is known as the Gauss-Seidel method:

$$(D + L)x^{(i+1)} = -Ux^{(i)} + b, \qquad \text{for } i = 0, 1, 2, \ldots.$$

Gauss-Seidel method is called sequential, because it's computations cannot be performed in parallel (as in Jacobi's) but in a specified order.

Conditioning for the above methods

Both, Jacobi's and Gauss-Seidel method work only if to following condition is fulfilled:

$$sr(M) < 1$$

Where M is calculated differently for each method:

For Jacobi's method: M = (L+U)/(-D)

For Gauss-Seidel method: M = -U/(D+L)

When having M matrix, the spectral radius is calculated as follows:

$$sr(M) = \max\{|\lambda_1|, |\lambda_2|, \dots, |\lambda_n|\}$$

For the Jacobi's method, a **strong diagonal dominance** of the matrix A is the sufficient convergence condition. It can be either row or column strong dominance.

Stop test

In every iterative method it is necessary to have a criterium to terminate the iterations. I have chosen to check an Euclidean norm after each iteration:

$$\left\|Ax^{(i+1)} - b\right\| \le \delta$$

where δ means an assumed terminate criterion.

Results

Solutions for the system from the task 3

srJ = 0.5347

srGS =0.4531

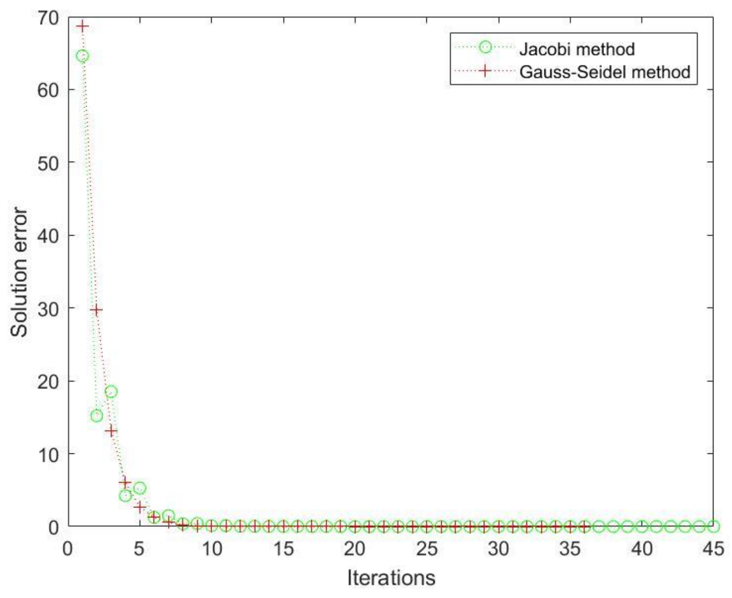

Fig. 5 Plot of the solution error norm vs. the iteration number for Jacobi and Gauss-Seidel method

Solutions for the system from the task 2a)

srJacobi =0.2132

srGauss-Seidel = 0.0455

Using Jacobi and Gauss-Seidel method to solve a matrix from the task 2a) I obtained the same results as previously:

```
x =

   0.1961

   0.2348

   0.2911

   0.3454

   0.4000

   0.4546

   0.5090

   0.5647

   0.6090

   0.7546
```

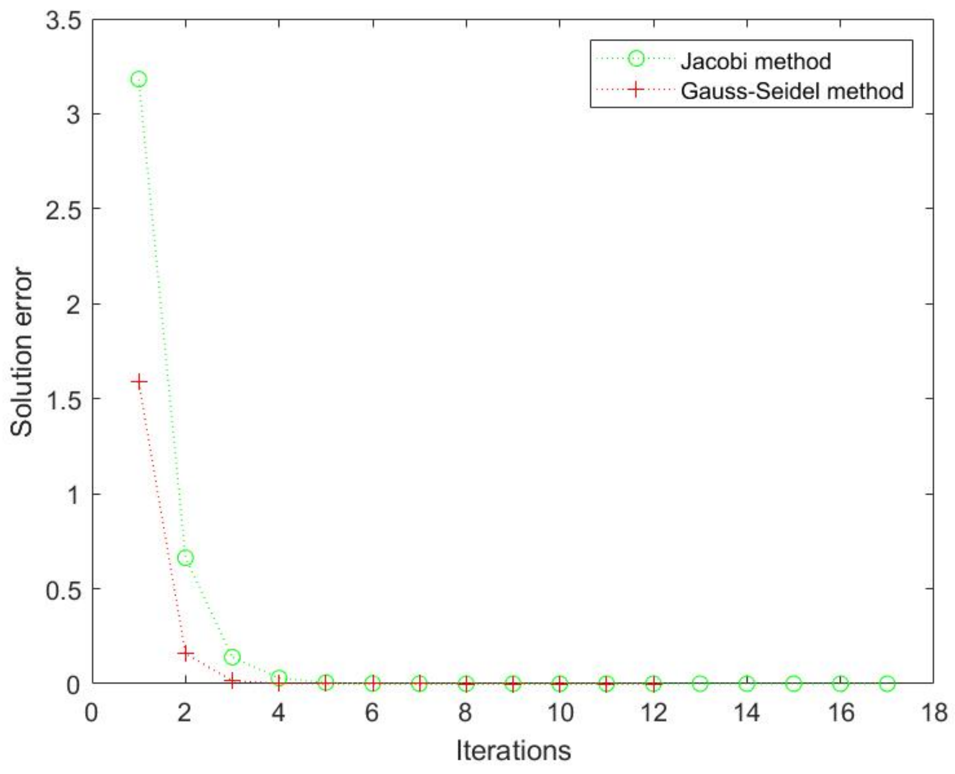

Fig. 6 Plot of the solution error norm vs. the iteration number for Jacobi and Gauss-Seidel method while solving task 2a)

Solutions for the system from the task 2b)

Jacobi and Gauss-Seidel method failed to solve matrixes from the task 3b), because the condition for the spectral radius (sr(M)<1) was not fulfilled.

srJ = 7.7798

srGS = 1.0000

Conclusions

From the Fig. 3 and Fig. 4 we can clearly notice that the solution was obtained much faster using Gauss-Seidel method than Jacobi's method. For the four equation matrix, we can notice that while using Jacobi's method the solution error did not decrease in each iteration. Error fluctuated few times, until it normalized and gradually decreased. In the 10x10 matrix solution, we can observe that the solution error in first iteration is more than two times bigger for the Jacobi's method. The difference in the error from the first iteration is not so high for the 4x4 matrix, however the solution error for Gauss-Seidel method is in this case also smaller.

The stop condition used in this task, was the Euclidean norm. It is not the fastest and the most optimal, because we have to perform operations on matrixes in each iteration, however due to the fact that we calculate the Euclidean norm, we can observe the error solution of the whole system of equation. This stop condition provides more information to be analyzed.

After the examination of my results, we can conclude, that Gauss-Seidel method is more optimal than the Jacobi's method.

TASK 4.

Program finds eigenvalues of 5x5 matrixes using QR method, with the following criterions:

a) Without shifts
b) With shifts calculated on the basis of an eigenvalue of the 2x2 right-lower-corner submatrix

There can be found a comparison of both approaches for a chosen symmetric matrix 5x5 in terms of iterations needed to force all off-diagonal elements below the prescribed absolute value threshold 10^{-6}.

Theoretical background

To use QR method for finding eigenvalues, firstly we have to use QR factorization. At the beginning, it is recommended to transform the matrix A to the tridiagonal form, because it increases the effectiveness of calculations. Secondly, the matrix A is decomposed into an orthogonal matrix Q and an upper triangular matrix R.

The best algorithm to perform QR factorization is modified Gram-Schmidt algorithm, because it has better numerical properties. In this algorithm, columns are orthogonalizes one after another.

In a square n-dimensional matrix, we always expect to obtain exactly n eigenvalues and n corresponding eigenvectors. If λ satisfies the characteristic equation:

$$\det(A - \lambda I) = 0$$

then it is called an eigenvalue of the matrix A.

The set of all eigenvalues of a matrix A is the spectrum of A, denoted as sp(A).

Eigenvalues using QR method can be found in two versions: without shifts and with shifts.

We can find eigenvalues using QR method in two versions: without shifts and with shifts. In this task, for the case b) with shifts, they are calculated on the basis of an eigenvalue of the 2x2 right-lower-corner submatrix. For QR method without shifts, the convergence ratio is:

$$\frac{|a_{i+1,i}^{(k+1)}|}{|a_{i+1,i}^{(k)}|} \cong \left|\frac{\lambda_{i+1}}{\lambda_i}\right|$$

So the method is slowly convergent if certain eigenvalues have similar values. On the other hand, for QR method with shifts, the convergence ratio is:

$$\frac{|a_{i+1,i}^{(k+1)}|}{|a_{i+1,i}^{(k)}|} \cong \left|\frac{\lambda_{i+1} - p_k}{\lambda_i - p_k}\right|$$

where pk is the shift and it is the best if it was chosen as an actual estimate of λi+1.

Usage of shifts, increases the speed of the convergence.

Results

Initial 5x5 matix

```
A = 1    0    2    4    1
    0    2    5    3    2
    2    5    4    3    5
    4    3    3    1    5
    1    2    5    5    2
```

Eigenvalues calculated with QR method without shifts:

```
  -5.5283
  -1.5653
  -0.3626
   2.5435
  14.9127
```

Iterations: 36

The convergence ratio: 0.17056

Eigenvalues calculated with QR method with shifts:

```
  -5.5283
  -1.5653
  -0.3626
   2.5435
  14.9127
```

Iterations: 1

The shift: 2.5435

The convergence ratio: 2.8772e-16

Eigenvalues calculated using Matlab's function 'eig':

```
  -5.5283
  -1.5653
  -0.3626
   2.5435
  14.9127
```

Final 5x5 matix

```
EigMatrix =

  14.9127    0.0000    0.0000   -0.0000   -0.0000
   0.0000   -5.5283    0.0000   -0.0000   -0.0000
        0    0.0000    2.5435    0.0000   -0.0000
        0         0    0.0000   -1.5653    0.0000
        0         0         0   -0.0000   -0.3626
```

Conclusions

The speed of convergence depends on the ratios of the successive eigenvalues. The convergence is faster if the ratio is close to zero. When it is close to 1, then the speed decreases.

From the obtained results we can notice that the method without shifts required many iterations, comparing to the method with shifts, where 1 iteration was enough to obtain the same results.

In the case without shifts, the convergence ratio is equal to 0.17056, so it is relatively high and relatively close to 1. In the case with shift, the chosen shift was equal to 2.5435, what is a good approach cause if we subtract this shift from the two consecutive eigenvalues, their convergence ratio was equal to 2.8772e-16 what is very close to 0.

The above deductions were performed on the basis of equations presented in the Theoretical Background section. The main conclusion from this task is that the QR method is much more efficient with shifts.

Appendix

```
close all
clear
clc

% to terminate the proper task, ungreen consecutive parts

%TASK1
%MachineEpsilon;

%TASK2
%plotResiduum(8);
%plotCorrectedResiduum;

%TASK3
% [A,b]=matrixTask3
% plotJacobiVSGauss(A, b)

%TASK4
%[A,newANoShifts]=ShiftsVsNoShifts()

%==============TASK 1==============
function MachineEpsilon
format long;

macheps = 1.0;
while((1.0 + (macheps/2.0)) > 1.0)
    macheps = macheps/2.0;
end

disp("Calculated machine epsilon: ")
disp(macheps)

disp("Matlab's machine epsilon value: ")
disp(eps)

end

%=============TASK 2 DATA==============

function [A,b] = matrixesInA(n)
format short;

for i=1:n
    for j=1:n
        if i==j
            A(i,j) = 9;
        elseif i==(j-1) || i==(j+1)
            A(i,j) = 1;
        else
            A(i,j) = 0;
        end
    end
end
```

```
for i=1:n
    b(i,1)=1.4+0.6*i;
end
end


function [A,b] = matrixesInB(n)
format short;

for i=1:n
    for j=1:n
        A(i,j) = 3/(4*(i+j-1));
    end
end

for i=1:n
    if mod(i,2) == 0;
        b(i,1)=1/i;
    else
        b(i,1)=0;
    end
end
end

%=========TASK 2 NEEDED FUNCTIONS==========

function [Aaug]=AugmentedMatrixWithPP(A,b);
Aaug = [A,b];
n = size(A,1);

for k = 1:n
   i = k;
   for j = i+1:n
       if (Aaug(j,i)> Aaug(i, j))
           i = j;
       end
   end
   if k~=i
       Aaug([k i],:) = Aaug([i k],:);
   end
end
end


function [Aech]=RowEchelonForm(Aaug);
n = size(Aaug,1);
Aech = Aaug;

for k = 1:n
    for j = k+1:n
        rowMultiple = Aech(j, k) / Aech(k, k);
        Aech(j, :) = Aech(j, :) - Aech(k, :)*rowMultiple;
    end
end
end
```

```matlab
function [x]=backSubstitution(Aech);
n = size(Aech,1);
x = zeros(1,n);
NewA = Aech(:, 1:end-1);
NewB = Aech(:, end);

for i = 1:n
    k = n-i+1;
    sum = 0;
    for j = k+1:n
        sum = sum + Aech(k, j)*x(j);
    end
    x(k) = (NewB(k) - sum) / NewA(k,k);
end
x = x';
end

function [residuum]=calculateResiduum(A,b,x);
residuum = A*x - b;
end

function [residuumEuclidean]=calculateEuclideanNorm(residuum);
norm = 0;
for i = 1:size(residuum)
    norm = norm +(residuum(i)*residuum(i));
end
residuumEuclidean = sqrt(norm);
end

function [finalResiduum, newx]=improvedResiduum(A,b,x,residuum);
residuum = residuum';
deltax = residuum/A;
deltax = deltax';
newx = x - deltax;
finalResiduum = A*newx - b;
end

function [x, residuumEuclideanNorm, finalResiduumEuclideanNorm] =
solveGaussianEPP(n);
[A,b]=matrixesInA(n);
%[A,b] = matrixesInB(n);
[Aaug]=AugmentedMatrixWithPP(A,b);
[Aech]=RowEchelonForm(Aaug);
[x]=backSubstitution(Aech);
[residuum]=calculateResiduum(A,b,x);
[residuumEuclideanNorm]=calculateEuclideanNorm(residuum);
[finalResiduum, newx]=improvedResiduum(A,b,x,residuum);
[finalResiduumEuclideanNorm]=calculateEuclideanNorm(finalResiduum)
end

function plotResiduum(iterations);
n = 10;

for i = 1 :  iterations
    [~, residuumEuclideanNorm, ~] = solveGaussianEPP(n);
    numberOfEquations(i) = n;
    errors(i) = residuumEuclideanNorm;
    n = n*2;
end

plot(numberOfEquations, errors, "or")
xlabel("Number of equations");
ylabel("Solution error before improvement");

end

function plotCorrectedResiduum;
%[A,b]=matrixesInA(10);
[A,b] = matrixesInB(40);
[Aaug]=AugmentedMatrixWithPP(A,b);
[Aech]=RowEchelonForm(Aaug);
[x]=backSubstitution(Aech);
[residuum]=calculateResiduum(A,b,x);
[residuumEuclideanNorm]=calculateEuclideanNorm(residuum);

first = 0;
plot(first,residuumEuclideanNorm,"or");
hold on;

%make the corrected residuum in a loop and transform it into
Euclidean norm
norm = 0;
for j = 1:1:10
    residuum = residuum';
    deltax = residuum/A;
    deltax = deltax';
    newx = x - deltax;
    residuum = A*newx - b;

    for i = 1:size(residuum)
        norm = norm +(residuum(i)*residuum(i));
    end
    residuumEuclidean(j) = sqrt(norm);
    iteration(j) = j;
    ;

end

plot(iteration, residuumEuclidean,"or");
hold off;
xlabel("Number of iterations");
ylabel("Corrected solution error");

end

%==============TASK 3==============

function [A,b]=matrixTask3;

A = [8 2 -3 1; 2 -25 5 -18; 1 3 15 -8; 1 1 -2 -10];
b = [7; 12; 24; 28];
```

```
end

function [L, D, U] = decompose(A);
%decomposition of the matrix A = L + D + U

L = tril(A);
for i = 1:size(A)
    L(i, i) = 0;
end

D = diag(diag(A));

U = triu(A);
for i = 1:size(A)
    U(i, i) = 0;
end
end

function [sr] = getSpectralRadiusJacobi(L, D, U);

M = -inv(D)*(L+U);
sr = max(abs(eig(M)));

end

function [sr, x, errors, iterations] = JacobiMethod(A, b);

[L, D, U] = decompose(A);
[sr] = getSpectralRadiusJacobi(L, D, U);

sizeA = size(A);
%initial solution matrix filled with zeros
x = zeros(sizeA(1), 1);

solutionError = inf;
lambda = 10^(-10);

i = 1;
imax = 100;
while solutionError > lambda && i < imax
    currentx = D\(b - A*x);
    x = x + currentx;

    solutionError = norm(A*x - b);

    errors(i) = solutionError;
    iterations(i) = i;
    i = i + 1;
end

if i >= imax
    disp("Maximal number of iterations reached");
end

errors = errors';
iterations = iterations';
```

```
end

function [srM] = getSpectralRadiusGaussSeidel(L, D, U);

M = -U/(D+L);
srM = max(abs(eig(M)));

end

function [sr, x, errors, iterations] = GaussSeidelMethod(A, b);

[L, D, U] = decompose(A);
[sr] = getSpectralRadiusGaussSeidel(L, D, U);

sizeA = size(A);
%initial solution matrix filled with zeros
x = zeros(sizeA(1), 1);

D = diag(A);

Error = inf;
lambda = 10^(-10);

i = 1;
imax = 100;

while Error > lambda && i < imax

    for k = 1:size(x)
        newx = x;

        for j = 1:size(x)
            newx(j) = newx(j) * A(k,j);
        end

        x(k) = (b(k) - (sum(newx) - newx(k)))/D(k);
    end

    Error = norm(A*x - b);
    errors(i) = Error;
    iterations(i) = i;
    i = i + 1;
end

errors = errors';
iterations = iterations';

end

function plotJacobiVSGauss(A, b);

[sr, x, errorsJ, iterationsJ] = JacobiMethod(A, b)
[sr, x, errorsGS, iterationsGS] = GaussSeidelMethod(A, b)

plot(iterationsJ, errorsJ, ':og')
```

```matlab
hold on
plot(iterationsGS, errorsGS, ':+r')
legend("Jacobi method", "Gauss-Seidel method");
xlabel("Iterations");
ylabel("Solution error");
hold off;

end

%==============TASK 4==============

function [A]=matrixTask4(m);

    A = randi(6, m, m) - 1;
    A = A - tril(A, -1) + triu(A, 1)';

end

function [Q, R, d] = getQR(A);

[m, n] = size(A);
Q = zeros(m, n);
R = zeros(n, n);
d = zeros(1, n);

for i=1:n
    Q(:, i) = A(:, i);
    R(i, i) = 1;
    d(i) = Q(:, i)' * Q(:, i);

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

    for i = 1:n
        dd = norm(Q(:, i));
        Q(:, i) = Q(:, i) / dd;
        R(i, i:n) = R(i, i:n) * dd;
    end

end

end

function [x1, x2] = getRoots(a, b, c)

delta = b*b - 4*a*c;

x1 = (-b + sqrt(delta))/(2*a);
x2 = (-b - sqrt(delta))/(2*a);

end

function [A, eigenvalues, iterations, convergenceRatio] =
GSNoShifts(A)
```

```
A = hess(A);
tol = 10^(-6);

imax = 100;
n = size(A, 1);
i = 1;

while i <= imax & max(max(A - diag(diag(A)))) > tol
    [Q1, R1] = getQR(A);
    A = R1 * Q1;
    i = i + 1;
end

iterations = i;
eigenvalues = diag(A);
eigenvalues = sort(eigenvalues);
convergenceRatio = abs(eigenvalues(n-1)/eigenvalues(n));
end

function [eigenvalues, iterations, shift,convergenceRatio] =
GSWithShifts(A)

A = hess(A);
tol = 10^(-6);

imax = 100;
n = size(A, 1);

eigenvalues = diag(ones(n));
initialsubA = A;

for k = n:-1:2
    DK = initialsubA;
    i = 0;
    while i <= imax & max(abs(DK(k,1:k-1))) > tol
        DD = DK(k-1:k,k-1:k);
        [ev1, ev2] = getRoots(1, -(DD(1,1)+DD(2,2)),
DD(2,2)*DD(1,1)-DD(2,1)*DD(1,2));
        if abs(ev1 - DD(2,2)) < abs(ev2 - DD(2,2))
            shift = ev1;
        else
            shift = ev2;
        end
        DP = DK - eye(k)*shift;
        [Q1,R1] = getQR(DP);
        DK = R1*Q1 + eye(k)*shift;
        i = i+1;
    end

    eigenvalues(k) = DK(k,k);
    convergenceRatio = abs((eigenvalues(k) - shift) /
(eigenvalues(k-1) - shift));

    if k > 2
        initialsubA = DK(1:k-1, 1:k-1);
    else
        eigenvalues(1) = DK(1,1);
    end
end

eigenvalues = sort(eigenvalues);
iterations = i;

end

function [A,newANoShifts]=ShiftsVsNoShifts()

[A] = matrixTask4(5);

[newANoShifts,eigenvaluesNoShift, iterationsNoShift,
convergenceRatioNoShift] = GSNoShifts(A);
[eigenvaluesWithShift, iterationsWithShift,
shift,convergenceRatioWithShift] = GSWithShifts(A);

 disp("Eigenvalues calculated with QR method without shifts:")
 disp(eigenvaluesNoShift)
 disp("Iterations: " + iterationsNoShift)
 disp("The convergence ratio: " + convergenceRatioNoShift)
 disp("Eigenvalues calculated with QR method with shifts:")
 disp(eigenvaluesWithShift)
 disp("Iterations: " + iterationsWithShift)
 disp("The shift: " + shift)
 disp("The convergence ratio: " + convergenceRatioWithShift)

 matlabEV = eig(A);
 disp("Eigenvalues calculated using Matlab's function 'eig':")
 disp(matlabEV)


end
```

References

Piotr Tatjewski “Numerical Methods”