Numerical Methods

Project A no. 23

Bartłomiej Jacak 303859

October, 2020

1 Finding machine epsilon in MATLAB environment

1.1 Theoretical Introduction

Machine epsilon, also called macheps, determines the boundaries of relative error obtained due to floating-point arithmetics. In simple words, it can tell us the maximal difference between the true floating-point number and the one calculated on a computer. The reason behind the inaccuracy is a finite number of bits used to store numbers and therefore roundings that need to take place.

We can represent any floating-point number in a following way:

$$x_{t,r} = m_t \cdot 2^{e_r} \tag{1}$$

where m_t is t-bit long mantissa (fractional part of the number) and e_r is r-bits long exponent. We need to remember that the range of numbers represented by mantissa should be normalized. One of the most commonly used standards for representing floating-point numbers is IEEE 754 [1], for which, the following mantissa normalization is applied:

$$1 \leqslant |m_t| < 2 \tag{2}$$

According to that assumption, the first bit will always be equal to 1, so it can be ommited in mantissa binary representation. The rest t-bits represent a summation of numbers with base 2 and negative exponents from -1 to -t. Moreover, it can be stated that for any given values of t and r, the set of floating-point numbers $M \subset R$ is finite and:

- the bigger t, the more dense and precise is the set M.
- the bigger r, the more broad is the interval of numbers covered by the set M.

Therefore, we can deduce that machine epsilon is not dependent on the sign nor the exponent — only on the mantissa. More accurately, we can say that macheps is always equal to 2^{-t}.

[1] Technical standard for floating-point arithmetic established in 1985 by the Institute of Electrical and Electronics Engineers (IEEE).

1.2 Algorithm

Dividing number by 2 is an equivalent of shifting mantissa one bit to the right. We can use this property, along with the fact that another way to describe machine epsilon *eps* is to say that it is the smallest positive number such that $(1 + eps) > 1$.

So, let's choose a number 1 as the initial value of *eps* – as it satisfies $(1 + 1) > 1$. Then start dividing it by 2 until $(1 + eps) > 1$ stops being true.

1.3 Obtained results

As could be read in documentation[2], MATLAB defaultly constructs floating-point numbers with a double-precision, which is an equivalent of using 64 bits in a following way:

1 bit	**11 bits**	**52 bits**
sign	exponent	mantissa

The result we obtained is equal to $2.220446049250313e-16$, which is exactly the result of 2^{-52}. We can also verify that our result is correct by using MATLAB's internal `eps` funtion, which returns the machine epsilon.

2 Solving system of n linear equations using Gaussian elimination with partial pivoting

2.1 Theoretical Introduction

Gaussian elimination with partial pivoting is a finite methods which can be used to solve systems of n linear equations expressed in a matrix form as $Ax = b$. We need to make an assumption that our A matrix is nonsingular so that partial pivoting method can work.

We use partial pivoting to prevent divisions by zero in situations when element from matrix diagonal (x_{kk} for k-row) is equal to 0. It also reduces the round-off errors as it makes the process less likely to contain addition or substraction with a very small or very large numbers.

Gaussian elimination with partial pivoting consists of two phases:

2.1.1 Elimination with partial pivoting

This phase is responsible for the conversion of A matrix to the upper triangular matrix U so that $Ux = c$, where c is an altered vector b. It consists of two steps:

Partial pivoting

Partial pivoting is called *partial* because we perform it only on the rows. Let's denote current row as k and the diagonal element as x_{kk}. Then perform the following steps:

1. Search all rows greater than k for the one with the first occurance of an element with the highest magnitude.

[2] https://www.mathworks.com/help/matlab/matlab_prog/floating-point-numbers.html

2. Swap found fow with the current one (the *pivot row*). We made an assumption that A matrix is nonsingular, so we know for sure that we will come across at least one non-zero element.

Elimination

Now, in order to obtain upper triangular matrix, we need to reduce all elements below the diagonal to 0. We do this by manipulating rows below the pivot row. We are going to substract the pivot row elements times some factor from each of them. The beforementioned factor can be defined as the ratio of the element in the current row divided by the diagonal element. Suppose that we denote current row as k, then we can define factor l as:

$$l_{i,k} \stackrel{\text{def}}{=} \frac{x_{ik}}{x_{kk}}$$

for all $i > k$.

This way, we can easily obtain following equations, which will be used for *back-substitution* phase:

$$u_{11}x_1 + u_{12}x_2 + ...u_{1(n-1)}x_3 + u_{1n}x_4 = c_1$$

$$u_{22}x_2 + u_{23}x_3 + u_{24}x_4 = c_2$$

$$...$$

$$u_{(n-1)(n-1)}x_{(n-1)} + u_{(n-1)n}x_n = c_{(n-1)}$$

$$u_{nn}x_n = c_n$$

2.1.2 Back-substitution phase

In this phase, we proceed to solve all the linear algebraic equations. Let's start from the last, n-th equation:

$$u_{nn}x_n = c_n \iff x_n = \frac{c_n}{u_{nn}}$$

Both c_n and u_{nn} are constant values so solving the equation above will give us the x_n value. By knowing x_n we can easily get the $x_{(n-1)}$ value:

$$u_{(n-1)(n-1)}x_{(n-1)} + u_{(n-1)n}x_n = c_{(n-1)}$$

$$x_{(n-1)} = \frac{c_{(n-1)} - u_{(n-1)n}x_n}{u_{(n-1)(n-1)}}$$

As well as the $x_{(n-2)}$ value:

$$u_{(n-2)(n-2)}x_{(n-2)} + u_{(n-2)(n-1)}x_{(n-1)} + u_{(n-2)n}x_n = c_{(n-2)}$$

$$x_{(n-2)} = \frac{c_{(n-2)} - u_{(n-2)(n-1)}x_{(n-1)} - u_{(n-2)n}x_n}{u_{(n-2)(n-2)}}$$

Therefore, we can observe a patter that for any n-size system, supposing that i takes values from 1 to $n-1$, we can write a general formulas for all x variables:

$$x_n = \frac{c_n}{u_{nn}} \tag{3}$$

$$x_i = \left(c_i - \sum_{j=i+1}^{n} u_{ij} x_j \right) \left(\frac{1}{u_{ii}} \right) \tag{4}$$

2.1.3 Residual correction

We can define solution error, often called *residuum* as:

$$r^{(n)} \stackrel{\text{def}}{=} Ax^{(n)} - b$$

where n is the number of iteration.

After obtaining the results for the first iteration we might not be satisfied with our $r^{(1)}$ which absolute value might be too big. To solve this problem we can make use of residual correction. Denoting the exact solution by $\hat{x}$ we have:

$$x^{(1)} = \hat{x} + \delta x$$

$$A(x^{(1)} - \delta x) = b$$

$$A\delta x = Ax^{(1)} - b$$

$$A\delta x = r^{(1)}$$

So we can calculate δx and second iteration of x can be denoted as $x^{(2)} = x^{(1)} - \delta x$. Then the second iteration of the residuum can be written this way:

$$r^{(2)} = Ax^{(2)} - b$$

If our error is still too large we can repeat whole procedure until we obtain the one which satisfies our conditions.

2.2 Algorithm

We perform the following steps for k iterations where $n = 10 \cdot 2^k$ is the size of the equation system.

In order to reduce performed operations, for each iteration, we can create an augmented matrix W so that the vector b is attached to the right side of the A. We can do that by simply concatenating them: $W = [Ab]$.

Next, we move along the diagonal elements of old A matrix on W and perform partial pivoting and elimination on each one according to steps described in theoretical introduction. Then, after

splitting our matrix back we obtain $Ux = c$ form, where U is the upper triangular matrix.

Now we can perform back-substitution in accordance to equations (3) and (4).

Next we should calculate solution error. To do that, we are going to substitute obtained solution x to $r = Ax - b$ and calculate Euclidean norm of the obtained vector r.

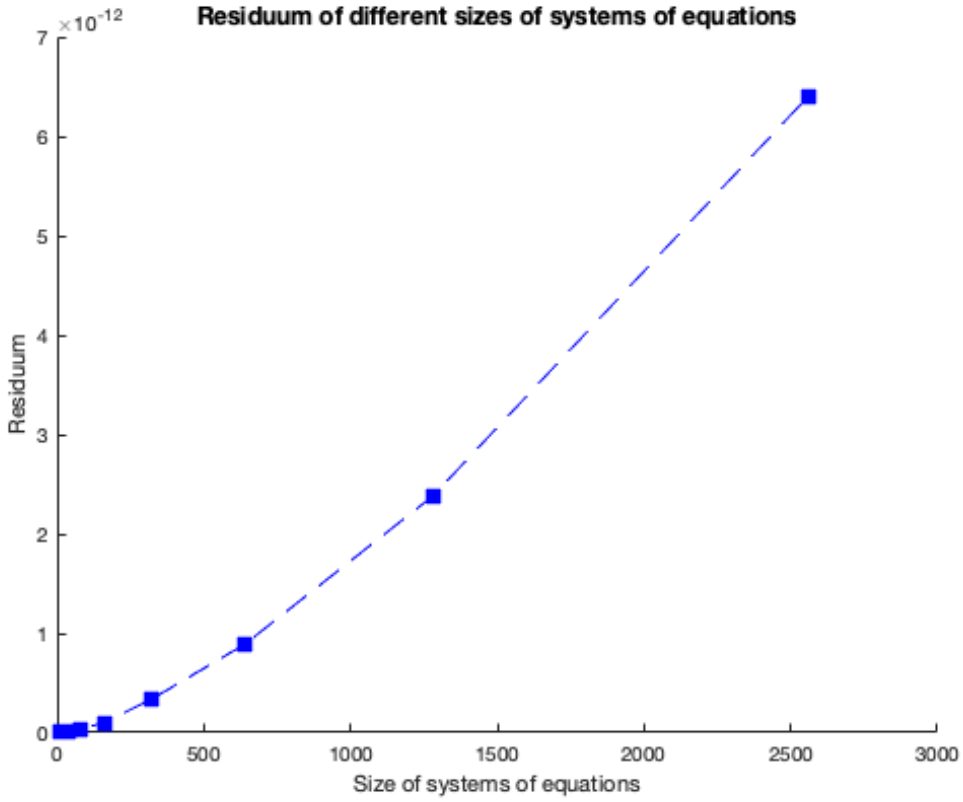
At the end we should also print solutions and solutions errors for the equation system of size 10 and check whether one iteration of residual correction will improve our result.

2.3 Obtained results

For both cases a) and b) I performed *9 iterations* as more of them resulted in excessively high solution time. Obtained results:

a)

Residuum plot:



As can be observed the error is relatively small, however there is a clear expotential growth so it might get pretty big for huge n values. Here are results and errors for $n = 10$:

$$x = \begin{bmatrix} -0.5195 \\ -0.5682 \\ -0.4692 \\ -0.3239 \\ -0.1644 \\ -0.0016 \\ 0.1589 \\ 0.3077 \\ 0.4179 \\ 0.4051 \end{bmatrix} \qquad r = 1.0e-15 \begin{bmatrix} -0.4441 \\ -0.8882 \\ 0.4441 \\ -0.2220 \\ 0.2220 \\ 0 \\ 0.1110 \\ 0.2220 \\ 0 \\ 0 \end{bmatrix} \qquad e = \|r\|_2 = 1.1591e-15$$

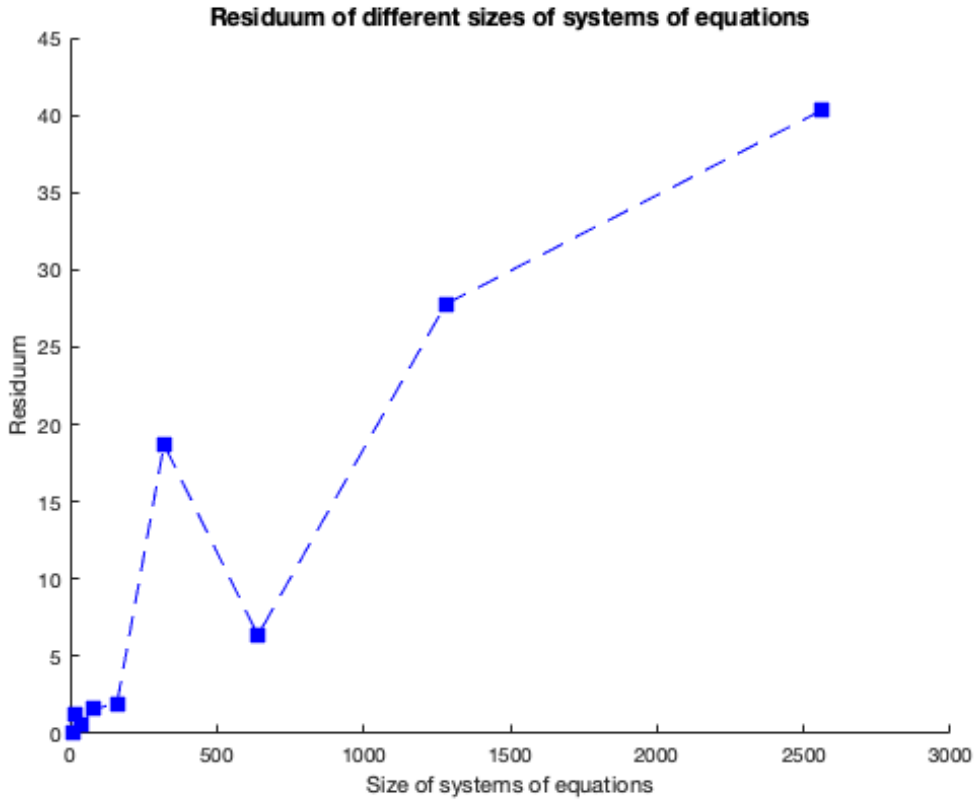
We can also try to perform residual correction on system of size $n = 10$ and check if our error improves the solutions:

$$x^{(2)} = \begin{bmatrix} -0.5195 \\ -0.5682 \\ -0.4692 \\ -0.3239 \\ -0.1644 \\ -0.0016 \\ 0.1589 \\ 0.3077 \\ 0.4179 \\ 0.4051 \end{bmatrix} \qquad e^{(2)} = 4.9651e-16$$

So we can easily conclude that one iteratin of correction indeed improved our solution, even though there is very small difference between first and corrected solution (even though I used short formating in MATLAB, for the better readibility of results).

b)

Residuum plot:



Case b) is a bit more interesting as obtained error gets very big very soon. Huge errors might come from the fact that matrix A in this case was some variation of Hilbert's matrix – known as an example of ill-conditioned matrices. We can also observe a clear increase in error value for even values of k, where b was a vector construced from zeros.

Here are results and errors for $n = 10$. In my opinion the only tested size of the system where errors are close to being acceptable:

$$x = 1.0e+13 \begin{bmatrix} 0.0002 \\ -0.0073 \\ 0.0929 \\ -0.5940 \\ 2.1871 \\ -4.9212 \\ 6.8774 \\ -5.8249 \\ 2.7378 \\ -0.5479 \end{bmatrix} \qquad r = 1.0e-03 \begin{bmatrix} -0.1003 \\ 0.3052 \\ -0.1351 \\ 0 \\ 0.0288 \\ 0.2747 \\ -0.2628 \\ -0.1526 \\ -0.2485 \\ 0.1068 \end{bmatrix} \qquad e = \|r\|_2 = 6.0269e-04$$

We can also try to perform residual correction on system of size $n = 10$ and check if our error improves the solutions:

$$x^{(2)} = 1.0e+13 \begin{bmatrix} 0.0002 \\ -0.0073 \\ 0.0930 \\ -0.5947 \\ 2.1899 \\ -4.9277 \\ 6.8865 \\ -5.8326 \\ 2.7414 \\ -0.5486 \end{bmatrix} \qquad e^{(2)} = 8.7702e-04$$

Here we can observe another interesting thing. Second iteration, in fact, made our results only worse.

3 Solving system of n linear equations using Gauss-Seidel and Jacobi iterative algorithms

3.1 Theoretical Introduction

Both Gauss-Seidel and Jacobi algorithms can be used to solve systems of n linear equations. They differ from the Gauss elimination method by the fact that they are iterative methods which means that the solution improves in subsequent iterations. Therefore, it can be stated that the number of iterations depends on an assumed accuracy of the solution.

In iterative methods we start from an initial solution vector $x^{(0)}$ which is ususally the best known approximation of the solution and we try to improve it using iterative algorithm. For $i = 0, 1, 2, ...$ and a given $x^{(0)}$ we can state the following:

$$\{x^{(i)}\}: \qquad x^{(i+1)} = Mx^{(i)} + w \tag{5}$$

For both Gauss-Seidel and Jacobi methods we firstly need to decompose A matrix into lower triangular subdiagonal matrix L, diagonal matrix D and upper triangular superdiagonal matrix U, so that:

$$A = L + D + U$$

3.1.1 Jacobi Iterative Method

Method based on the notion that it is very easy to invert diagonal matrix, as we just need to invert each of the matrix elements. Let us obtain equation consisting of inverse diagonal matrix D:

$$Ax = b \iff (L + D + U)x = b \iff Dx = -(L + U)x + b$$

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

$$x^{(i+1)} = -D^{-1}\left[(L+U)x^{(i)} - b\right] \qquad i = 0, 1, 2, ...$$

$$x^{(i+1)} = -D^{-1}(L+U)x^{(i)} + D^{-1}b \qquad i = 0, 1, 2, ... \tag{6}$$

So according to (3) we can say that for Jacobi method matrix M is equal to $-D^{-1}(L+U)$ and vector w to $D^{-1}b$. Matrix equation (4) can be written in a form of n independent scalar equations, which can be computed in a parallel:

$$x_j^{(i+1)} = -\frac{1}{d_{jj}}\left(\sum_{k=1}^{n}(l_{jk} + u_{jk})x_k^{(i)} + b_j\right), \qquad j = 1, 2, ..., n$$

Sufficient condition for convergence of the Jacobi's method:

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

3.1.2 Gauss-Seidel Iterative Method

For this method we make use the fact that L is a subdiagonal matrix.

$$Ax = b \iff (L + D + U)x = b \iff (D + L)x = -Ux + b$$

$$(D + L)x^{(i+1)} = -Ux^{(i)} + b$$

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

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

$$\begin{bmatrix} d_{11}x_1^{(i+1)} \\ d_{22}x_2^{(i+1)} \\ d_{33}x_3^{(i+1)} \\ \vdots \\ d_{nn}x_n^{(i+1)} \end{bmatrix} = - \begin{bmatrix} 0 & 0 & 0 & \dots & 0 \\ l_{21} & 0 & 0 & \dots & 0 \\ l_{31} & l_{32} & 0 & \dots & 0 \\ \vdots & \vdots & & & \vdots \\ l_{n1} & l_{n2} & l_{n3} & \dots & 0 \end{bmatrix} \begin{bmatrix} x_1^{(i+1)} \\ x_2^{(i+1)} \\ x_3^{(i+1)} \\ \vdots \\ x_n^{(i+1)} \end{bmatrix} - w^{(i)}$$

Denoting $w^{(i)} = Ux^{(i)} - b$ we can obtain:

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

$$x_2^{(i+1)} = \left(-l_{21} \cdot x_1^{(i+1)} - w_2^{(i)}\right)/d_{22}$$

$$\dots$$

$$x_j^{(i+1)} = \left(-\sum_{k=1}^{j-1}(l_{jk} \cdot x_k^{i+1}) - w_j^{(i)}\right)/d_{jj} \qquad j = 1, 2, ..., n$$

Those computations cannot be done in a parallel and must be performed sequentially in the proper order. Gauss-Seidel method has the same condition for convergence as Jacobi's method. Additionally, for symmetric matrices it is convergent if the matrix A is positive definite.

3.1.3 Stop condition

We need to specify some criterias to check whether it is the time to terminate iterations of an iterative method. For our algorithms we will be checking an Euclidean norm of the solution error vector:

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

3.2 Algorithm

Firstly, before applying any method we must check the convergence conditions appropriate to the method. Then we can start iterating solutions until stop condition occurs. Eventually, we can implement maximal number of iterations, so that we protect ourselves from unwanted infinity loops.

3.3 Obtained results

We assumed Euclidean norm of $Ax - b$ vector as an error and 10^{-10} as a stop condition – maximal error that satisfy us. For the given system of equations:

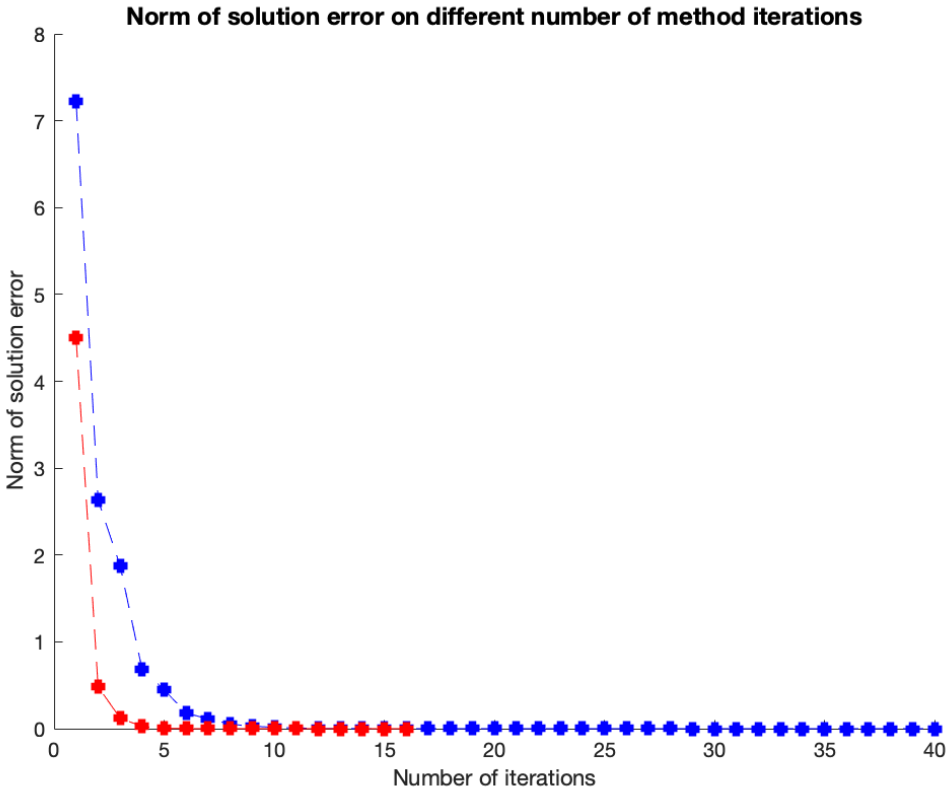
$$14x_1 - x_2 - 6x_3 + 5x_4 = 10$$

$$x_1 - 8x_2 - 4x_3 - x_4 = 0$$

$$x_1 - 4x_2 - 12x_3 - x_4 = -10$$

$$x_1 - x_2 - 8x_3 - 16x_4 = -20$$

we needed to perform **41 iterations of Jacobi's method** and only **17 iterations of Gauss-Seidel method**. Comparison of both methods can be seen on the following plot, where Jacobi's method iterations are plotted with blue color and Gauss-Seidel method iteration are plotted with red color:



Let us test Gauss-Seidel method on $n = 10$ size system of equations from 2a) and 2b).

2a)

Using 10^{-10} as stop condition we were able to obtain satisfying result after **24 iterations**:

$$x = \begin{bmatrix} -0.5195 \\ -0.5682 \\ -0.4692 \\ -0.3239 \\ -0.1644 \\ -0.0016 \\ 0.1589 \\ 0.3077 \\ 0.4179 \\ 0.4051 \end{bmatrix} \qquad r = 1.0e-10 \begin{bmatrix} -0.2714 \\ -0.2096 \\ -0.1394 \\ -0.0837 \\ -0.0461 \\ -0.0233 \\ -0.0107 \\ -0.0042 \\ -0.0012 \\ -0.0000 \end{bmatrix} \qquad e = \|r\|_2 = 3.8318e-11$$

2b)

Finding a proper stop condition was a bit tricky in this case. After a few tries I finally set it to 0.4 and a satisfying result was obtained after **68946 iterations** which is an impressive number:

$$x = 1.0e+05 \begin{bmatrix} 0.1818 \\ -1.6135 \\ 4.7114 \\ -5.8336 \\ 4.4117 \\ -4.9236 \\ 4.6910 \\ -4.1022 \\ 5.4491 \\ -2.9803 \end{bmatrix} \qquad r = \begin{bmatrix} -0.0293 \\ 0.1433 \\ -0.2440 \\ 0.1688 \\ -0.1145 \\ 0.1287 \\ -0.0665 \\ 0.0958 \\ -0.0856 \\ -0.0000 \end{bmatrix} \qquad e = \|r\|_2 = 0.4000$$

4 Appendix

4.1 Task 1 Code

```
macheps = 1;
while 1.0 + (macheps / 2.0) > 1.0
    macheps = macheps / 2.0;
end

disp("Machine epsilon is equal to:");
disp(macheps);
```

```
>> task_1
Machine epsilon is equal to:
     2.220446049250313e-16
```

4.2 Task 2 Code

```
function [solution] = ge_partial_pivoting(A, b)
    W = [A b]; % create the augmented matrix
    % get size of A matrix
    matrix_size = size(A);
    n = matrix_size(1);

    % Elimination with partial pivoting
    for k = 1 : n - 1
        [~, max_row_index] = max(abs(W(k : n, k))); % find the maximal magnitude
        max_row_index = max_row_index + k - 1; % add offset so index is correct

        % swap current pivot
        temp = W(k, :);
        W(k, :) = W(max_row_index, :);
        W(max_row_index, :) = temp;

        % gaussian elimination
```

```
        for i = k + 1 : n
            ratio = W(i, k) / W(k, k);
            W(i, :) = W(i, :) - ratio * W(k, :);
        end
    end

    % split matrix back into Ux = c
    U = W(:, 1 : n);
    c = W(:, n + 1);

    % initialize solutions vector
    solution = zeros(n, 1);

    % calculate solution to last equation
    solution(n) = c(n) / U(n, n);

    % calculate x(i)
    for i = (n - 1): -1 : 1
        row_sum = 0;
        for j = i + 1 : n
          row_sum = row_sum + U(i, j) * solution(j);
        end

        solution(i) = (c(i) - row_sum) / U(i, i);
    end
end

% Problem 2, task a)

format short;

n = 10;
number_of_iterations = 9;

n_values = zeros(1, number_of_iterations);
norms = zeros(1, number_of_iterations);

for iteration = 1 : number_of_iterations
    n_values(iteration) = n;

    % Build A matrix
    A = zeros(n, n);
    for i = 1 : n
       for j = 1 : n
            if i == j
```

```
                A(i,j) = 7;
            elseif i == j - 1 || i == j + 1
                A(i,j) = -2;
            end
        end
    end

    % Build b vector
    b = zeros(n, 1);
    for i = 1 : n
        b(i, 1) = -3 + 0.5 * i;
    end

    x = ge_partial_pivoting(A, b);
    r = A*x - b;
    normalized_residuum = norm(r);
    norms(iteration) = normalized_residuum;

    if n == 10
        disp("For n = 10");
        disp("Solution:");
        disp(x);
        disp("Residuum (solutions' errors):");
        disp(r);

        % Residual correction
        x_sigma = ge_partial_pivoting(A, r);
        x_2 = x - x_sigma;
        r_2 = A*x_2 - b;

        disp("After 1 iteration of residual correction:");
        disp("Improved solutions:");
        disp(x_2);
        disp("Error of x_1:")
        disp(normalized_residuum);
        disp("Error of x_2:")
        disp(norm(r_2));
    end

    n = n * 2;
end

hold on
plot(n_values, norms, 'b--');
plot(n_values, norms, 'b +', 'LineWidth', 10);
```

```
xlabel('Size of systems of equations');
ylabel('Residuum');
title('Residuum of different sizes of systems of equations','FontSize',12);
hold off

% Problem 2, task b)

n = 10;
number_of_iterations = 9;

n_values = zeros(1, number_of_iterations);
norms = zeros(1, number_of_iterations);

for iteration = 1 : number_of_iterations
    n_values(iteration) = n;

    % Build A matrix
    A = zeros(n, n);
    for i = 1 : n
       for j = 1 : n
            A(i, j) = 3 / (7 * (i + j + 1));
       end
    end

    % Build b vector
    b = zeros(n, 1);
    for i = 1 : n
        if mod(i,2) == 1
            b(i, 1) = 9 / (7 * i);
        end
    end

    x = ge_partial_pivoting(A, b);
    r = A*x - b;
    normalized_residuum = norm(r);
    norms(iteration) = normalized_residuum;

    if n == 10
        disp("For n = 10");
        disp("Solution:");
        disp(x);
        disp("Residuum (solutions' errors):");
        disp(r);

        % Residual correction
```

```
        x_sigma = ge_partial_pivoting(A, r);
        x_2 = x - x_sigma;
        r_2 = A*x_2 - b;

        disp("After 1 iteration of residual correction:");
        disp("Improved solutions:");
        disp(x_2);
        disp("Error of x_1:")
        disp(normalized_residuum);
        disp("Error of x_2:")
        disp(norm(r_2))
    end

    n = n * 2;
end

hold on
plot(n_values, norms, 'b--');
plot(n_values, norms, 'b +', 'LineWidth', 10);
xlabel('Size of systems of equations');
ylabel('Residuum');
title('Residuum of different sizes of systems of equations','FontSize',12);
hold off
```

```
>> task_2a
For n = 10
Solution:
   -0.5195
   -0.5682
   -0.4692
   -0.3239
   -0.1644
   -0.0016
    0.1589
    0.3077
    0.4179
    0.4051

Residuum (solutions' errors):
   1.0e-15 *

   -0.4441
   -0.8882
    0.4441
   -0.2220
```

```
    0.2220
         0
    0.1110
    0.2220
         0
         0

After 1 iteration of residual correction:
Improved solutions:
   -0.5195
   -0.5682
   -0.4692
   -0.3239
   -0.1644
   -0.0016
    0.1589
    0.3077
    0.4179
    0.4051

Error of x_1:
   1.1591e-15

Error of x_2:
   4.9651e-16

>> task_2b
For n = 10
Solution:
   1.0e+13 *

    0.0002
   -0.0073
    0.0929
   -0.5940
    2.1871
   -4.9212
    6.8774
   -5.8249
    2.7378
   -0.5479

Residuum (solutions' errors):
   1.0e-03 *
```

```
   -0.1003
    0.3052
   -0.1351
         0
    0.0288
    0.2747
   -0.2628
   -0.1526
   -0.2485
    0.1068

After 1 iteration of residual correction:
Improved solutions:
   1.0e+13 *

    0.0002
   -0.0073
    0.0930
   -0.5947
    2.1899
   -4.9277
    6.8865
   -5.8326
    2.7414
   -0.5486

Error of x_1:
   6.0269e-04

Error of x_2:
   8.7702e-04
```

4.3 Task 3 Code

```
A = [14 -1 -6 5; 1 -8 -4 -1; 1 -4 -12 -1; 1 -1 -8 -16];
b = [10; 0; -10; -20];
minimal_normalized_error = 10^-10;

% get size of A matrix
matrix_size = size(A);
n = matrix_size(1);

U = triu(A, 1);
L = tril(A, -1);
D = diag(diag(A));
```

```
x = zeros(n, 1);
normalized_error = Inf;

iteration = 1;

iterations = zeros(1, 1);
norms = zeros(1, 1);

% Jacobi method
while normalized_error > minimal_normalized_error
    prev_x = x;

    for j = 1 : n
        sum = 0;
        for k = 1 : n
            sum = sum + ((L(j, k) + U(j,k)) * prev_x(k));
        end

        x(j) = (1 / D(j,j)) * (b(j) - sum);
    end

    error = (A * x) - b;
    normalized_error = norm(error);
    norms(iteration) = normalized_error;
    iterations(iteration) = iteration;

    iteration = iteration + 1;
end

fprintf('Jacobi method finished after %i iterations\n', iteration);

hold on
plot(iterations, norms, 'b--');
plot(iterations, norms, 'b +', 'LineWidth', 10);
xlabel('Number of iterations');
ylabel('Norm of solution error');
title('Norm of solution error on different number of method iterations','FontSize',12);
hold off

x = zeros(n, 1);
normalized_error = Inf;

iteration = 1;
```

```
iterations = zeros(1, 1);
norms = zeros(1, 1);

% Gauss-Seidel method
while normalized_error > minimal_normalized_error
    prev_x = x;
    w = (U * prev_x) - b;

    for j = 1 : n
        sum = 0;
        for k = 1 : j - 1
            sum = sum + (L(j, k) * x(k));
        end

        x(j) = (1 / D(j,j)) * (-sum - w(j));
    end

    error = (A * x) - b;
    normalized_error = norm(error);
    norms(iteration) = normalized_error;
    iterations(iteration) = iteration;

    iteration = iteration + 1;
end

fprintf('Gauss-Seidel method finished after %i iterations\n', iteration);

hold on
plot(iterations, norms, 'r--');
plot(iterations, norms, 'r +', 'LineWidth', 10);
hold off
```

```
>> task_3
Jacobi method finished after 41 iterations
Gauss-Seidel method finished after 17 iterations
```

```
% Build A matrix
A = zeros(10, 10);
for i = 1 : 10
   for j = 1 : 10
        if i == j
            A(i,j) = 7;
        elseif i == j - 1 || i == j + 1
            A(i,j) = -2;
        end
   end
```

```
end

% Build b vector
b = zeros(10, 1);
for i = 1 : 10
   b(i, 1) = -3 + 0.5 * i;
end

minimal_normalized_error = 10^10;

U = triu(A, 1);
L = tril(A, -1);
D = diag(diag(A));

x = zeros(10, 1);
error = Inf;
normalized_error = Inf;

iteration = 1;

% Gauss-Seidel method
while normalized_error > minimal_normalized_error
    prev_x = x;
    w = (U * prev_x) - b;

    for j = 1 : 10
        sum = 0;
        for k = 1 : j - 1
            sum = sum + (L(j, k) * x(k));
        end

        x(j) = (1 / D(j,j)) * (-sum - w(j));
    end

    error = (A * x) - b;
    normalized_error = norm(error);

    iteration = iteration + 1;
end

disp("2a) using Gauss-Seidel method");
fprintf("Took %i iterations to compute\n", iteration);
disp("Solution:");
disp(x);
disp("Error:");
```

```
disp(error);
disp("Normalized error:");
disp(normalized_error);
```

```
>> task_3_2a
2a) using Gauss-Seidel method
Took 24 iterations to compute
Solution:
   -0.5195
   -0.5682
   -0.4692
   -0.3239
   -0.1644
   -0.0016
    0.1589
    0.3077
    0.4179
    0.4051

Error:
   1.0e-10 *

   -0.2714
   -0.2096
   -0.1394
   -0.0837
   -0.0461
   -0.0233
   -0.0107
   -0.0042
   -0.0012
   -0.0000

Normalized error:
   3.8318e-11
```

```
% Build A matrix
A = zeros(10, 10);
for i = 1 : 10
   for j = 1 : 10
        A(i, j) = 3 / (7 * (i + j + 1));
   end
end

% Build b vector
b = zeros(10, 1);
```

```
for i = 1 : 10
    if mod(i,2) == 1
        b(i, 1) = 9 / (7 * i);
    end
end

minimal_normalized_error = 0.4;

U = triu(A, 1);
L = tril(A, -1);
D = diag(diag(A));

x = zeros(10, 1);
error = Inf;
normalized_error = Inf;

iteration = 1;

% Gauss-Seidel method
while normalized_error > minimal_normalized_error
    prev_x = x;
    w = (U * prev_x) - b;

    for j = 1 : 10
        sum = 0;
        for k = 1 : j - 1
            sum = sum + (L(j, k) * x(k));
        end

        x(j) = (1 / D(j,j)) * (-sum - w(j));
    end

    error = (A * x) - b;
    normalized_error = norm(error);

    iteration = iteration + 1;
end

disp("2b) using Gauss-Seidel method");
fprintf("Took %i iterations to compute\n", iteration);
disp("Solution:");
disp(x);
disp("Error:");
disp(error);
disp("Normalized error:");
```

```
disp(normalized_error);
```

```
>> task_3_2b
2b) using Gauss-Seidel method
Took 68946 iterations to compute
Solution:
   1.0e+05 *

    0.1818
   -1.6135
    4.7114
   -5.8336
    4.4117
   -4.9236
    4.6910
   -4.1022
    5.4491
   -2.9803

Error:
   -0.0293
    0.1433
   -0.2440
    0.1688
   -0.1145
    0.1287
   -0.0665
    0.0958
   -0.0856
   -0.0000

Normalized error:
    0.4000
```