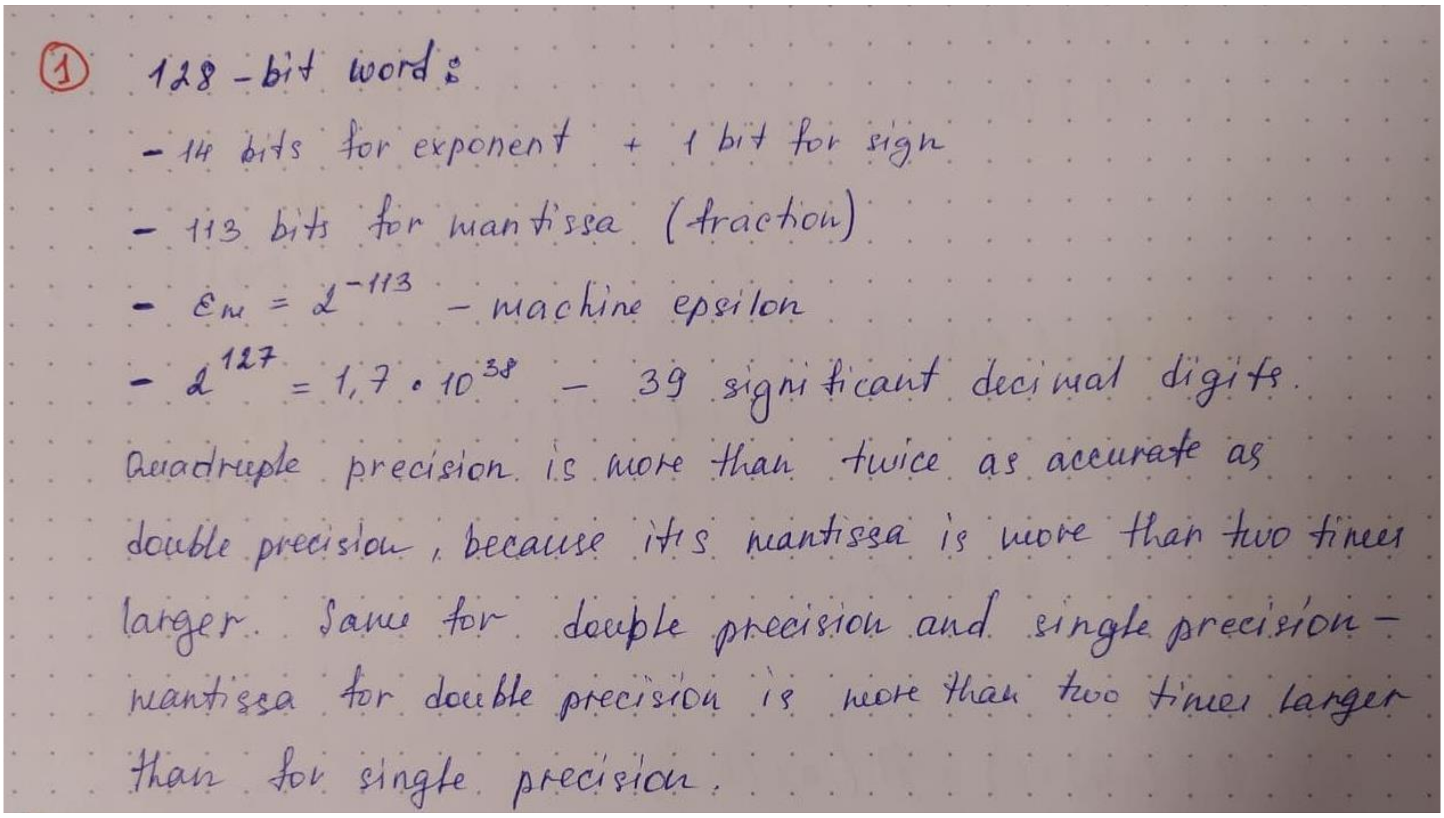
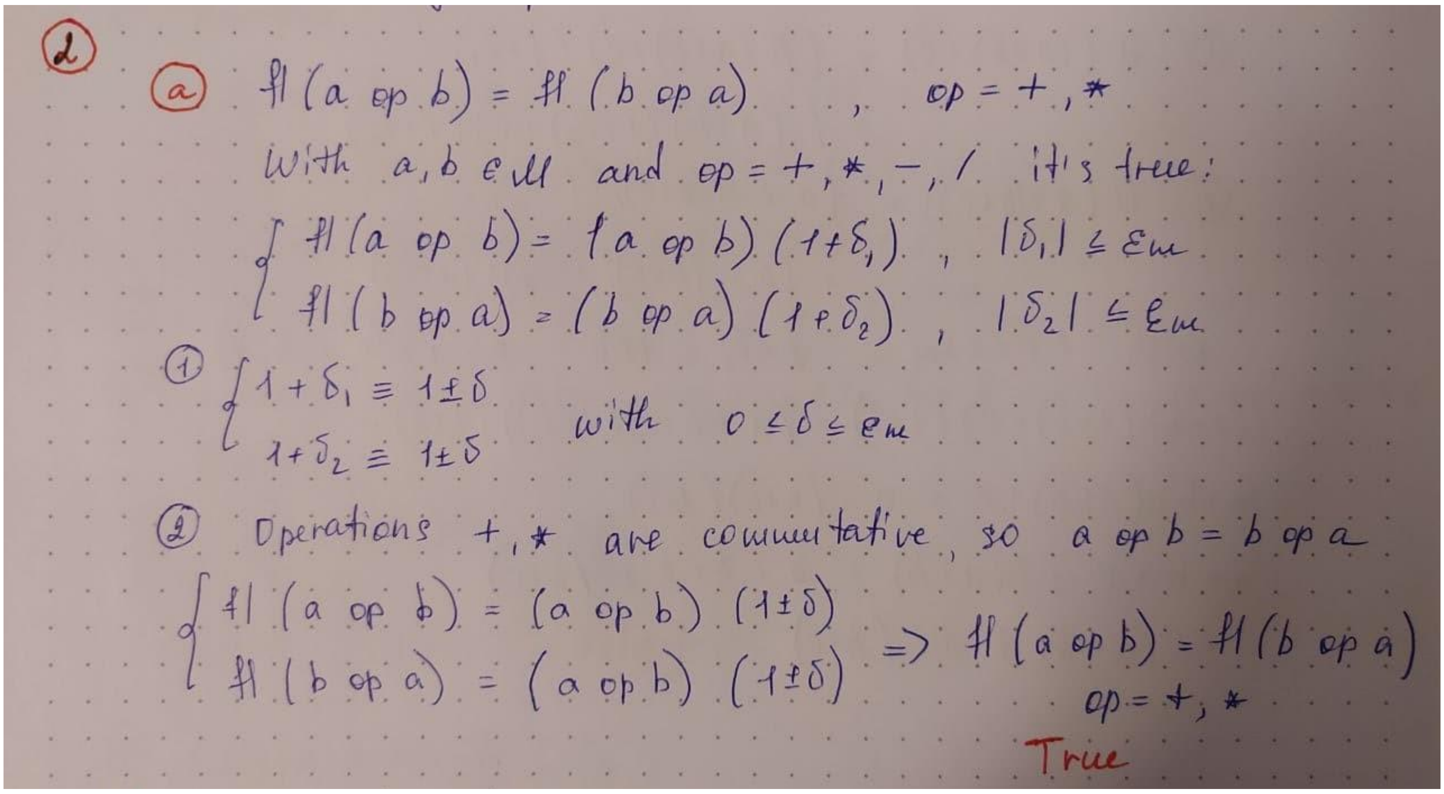
Numerical Algorithms. Assignment 1.
Krukovich Palina
a12134146

Paper-and-Pencil Exercises



1. 128-bit word:
- 14 bits for exponent + 1 bit for sign
- 113 bits for mantissa (fraction)
- $\varepsilon_m = 2^{-113}$ – machine epsilon
- $2^{127} = 1,7 \cdot 10^{38}$ – 39 significant decimal digits.

Quadruple precision is more than twice as accurate as double precision, because it's mantissa is more than two times larger. Same for double precision and single precision – mantissa for double precision is more than two times larger than for single precision.



2. a) $fl(a \text{ op } b) = fl(b \text{ op } a)$, $op = +, *$

With $a, b \in \mathcal{M}$ and $op = +, *, -, /$ it's true:

$$\begin{cases} fl(a \text{ op } b) = (a \text{ op } b)(1+\delta_1), & |\delta_1| \le \varepsilon_m \\ fl(b \text{ op } a) = (b \text{ op } a)(1+\delta_2), & |\delta_2| \le \varepsilon_m \end{cases}$$

1. $\begin{cases} 1+\delta_1 \equiv 1 \pm \delta \\ 1+\delta_2 \equiv 1 \pm \delta \end{cases}$ with $0 \le \delta \le \varepsilon_m$

2. Operations $+, *$ are commutative, so $a \text{ op } b = b \text{ op } a$

$$\begin{cases} fl(a \text{ op } b) = (a \text{ op } b)(1 \pm \delta) \\ fl(b \text{ op } a) = (a \text{ op } b)(1 \pm \delta) \end{cases} \Rightarrow fl(a \text{ op } b) = fl(b \text{ op } a)$$
$op = +, *$

True

(b) $fl(a+a) = fl(2*a)$

$(a+a)(1+\delta_1) = (2*a)(1+\delta_2) \leftarrow$ $\boxed{\begin{array}{l} \text{with } 0 \le \delta \le \varepsilon_m ; \\ 1+\delta_1 \equiv 1\pm\delta \\ 1+\delta_2 \equiv 1\pm\delta \end{array}}$

$(2*a)(1\pm\delta) = (2*a)(1\pm\delta)$

True

$\boxed{a+a = 2*a}$

(c) $fl((a+b)+c) = fl(a+(b+c))$

① $fl((a+b)+c) = (fl(a+b)+c)(1+\delta_1) =$

$= ((a+b)(1+\delta_2)+c)(1+\delta_1)$

② $fl(a+(b+c)) = (a+fl(b+c))(1+\delta_3) =$

$= (a+(b+c)(1+\delta_4))(1+\delta_4)$

with $0 \le \delta \le \varepsilon_m$: $1+\delta_i \equiv 1\pm\delta$ for $i \in \{1,2,3,4\}$

$((a+b)(1\pm\delta)+c)(\cancel{1\pm\delta}) = (a+(b+c)(1\pm\delta))(\cancel{1\pm\delta})$ -

$(a+b)(1\pm\delta)+c = a+(b+c)(1\pm\delta)$

$\cancel{a}+\cancel{b}+\cancel{c} \pm\delta(a+b) = \cancel{a}+\cancel{b}+\cancel{c} \pm\delta(b+c)$

$\pm\delta(a+b) \neq \pm\delta(b+c)$ False

③ $S_n = \sum_{i=1}^{n} x_i$ The order of evaluation is from left to right.

$x_i \in M$

① $S_i = x_1 + x_2 + \ldots + x_i$ - i-th partial sum

② It follows that:

$S_1 = x_1$

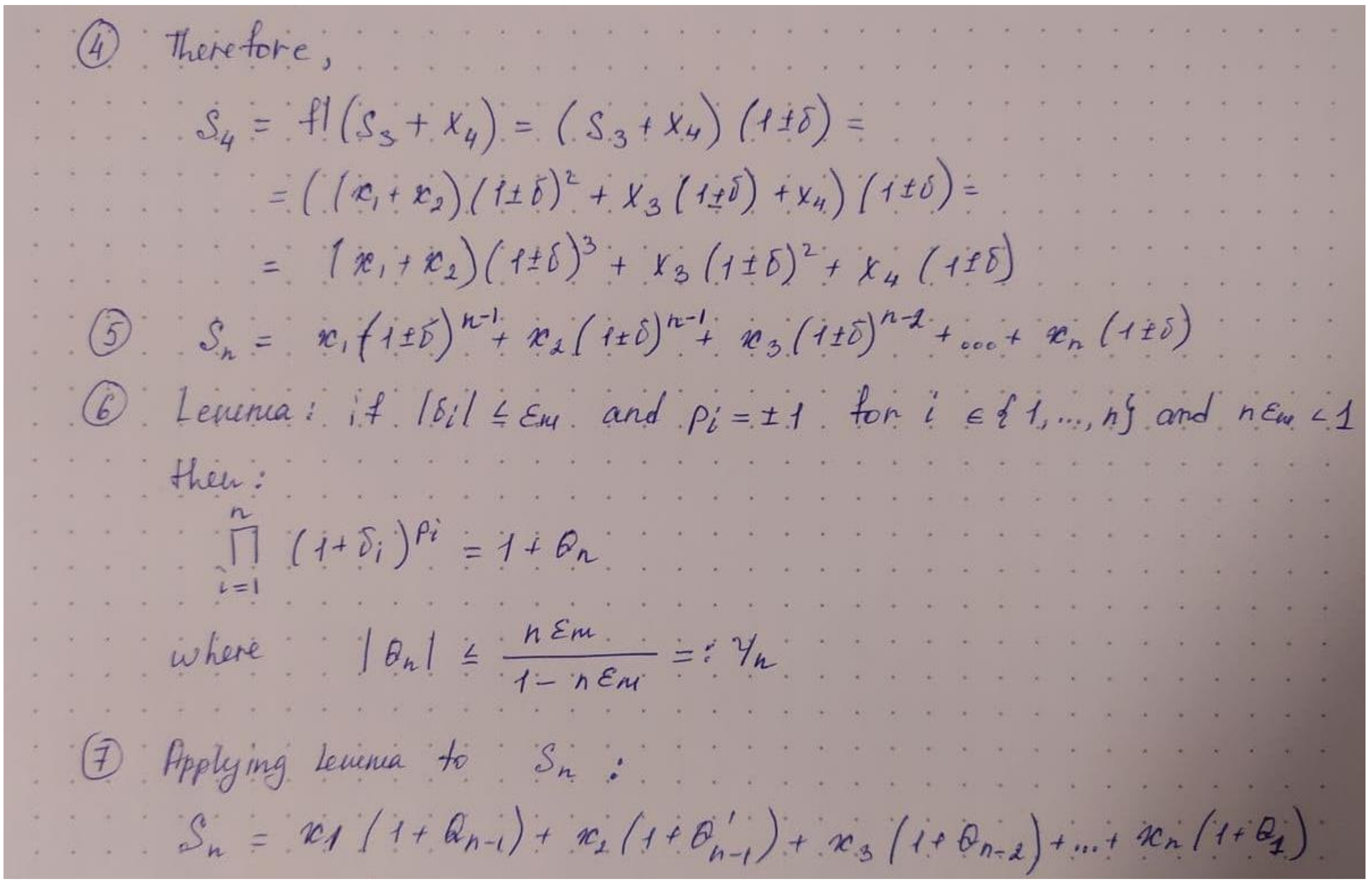
$S_2 = fl(S_1+x_2) = (S_1+x_2)(1+\delta_1) = (x_1+x_2)(1+\delta_1)$

$S_3 = fl(S_2+x_3) = (S_2+x_3)(1+\delta_2) = ((x_1+x_2)(1+\delta_1)+x_3)(1+\delta_2) =$

$= (x_1+x_2)(1+\delta_1)(1+\delta_2) + x_3(1+\delta_2)$

with $|\delta_i| \le \varepsilon_m$, $i \in \{1,2\}$

③ with $0 \le \delta \le \varepsilon_m \Rightarrow 1+\delta_i \equiv 1\pm\delta$



(4) Therefore,

$$S_4 = fl(S_3 + x_4) = (S_3 + x_4)(1 \pm \delta) =$$
$$= ((x_1 + x_2)(1 \pm \delta)^2 + x_3(1 \pm \delta) + x_4)(1 \pm \delta) =$$
$$= (x_1 + x_2)(1 \pm \delta)^3 + x_3(1 \pm \delta)^2 + x_4(1 \pm \delta)$$

(5) $S_n = x_1(1 \pm \delta)^{n-1} + x_2(1 \pm \delta)^{n-1} + x_3(1 \pm \delta)^{n-2} + \dots + x_n(1 \pm \delta)$

(6) Lemma: if $|\delta_i| \leq \varepsilon_m$ and $p_i = \pm 1$ for $i \in \{1, \dots, n\}$ and $n\varepsilon_m < 1$

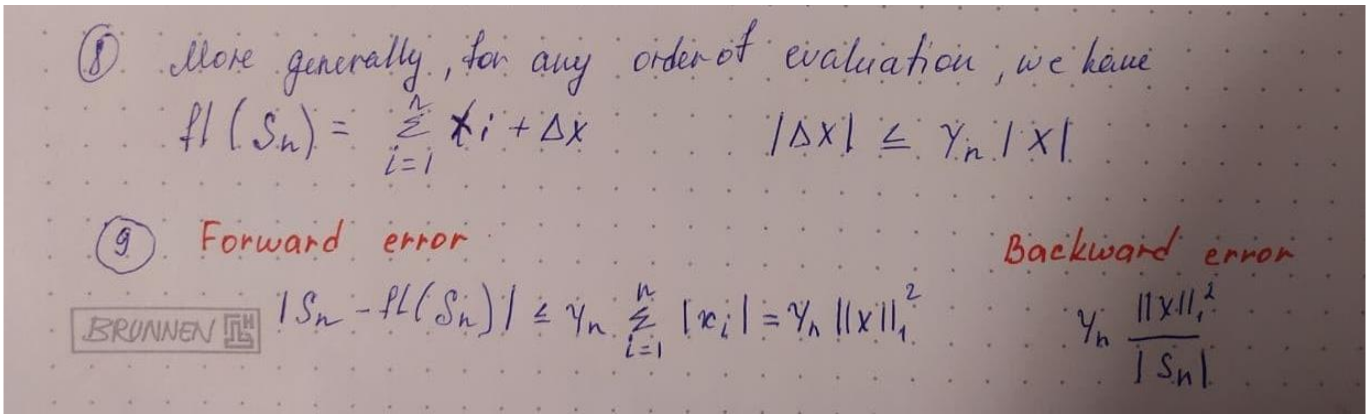
then:

$$\prod_{i=1}^{n} (1 + \delta_i)^{p_i} = 1 + \theta_n$$

where $$|\theta_n| \leq \frac{n\varepsilon_m}{1 - n\varepsilon_m} =: \gamma_n$$

(7) Applying Lemma to S_n:

$$S_n = x_1(1 + \theta_{n-1}) + x_2(1 + \theta'_{n-1}) + x_3(1 + \theta_{n-2}) + \dots + x_n(1 + \theta_1)$$



(8) More generally, for any order of evaluation, we have

$$fl(S_n) = \sum_{i=1}^{n} x_i + \Delta x \qquad |\Delta x| \leq \gamma_n |x|$$

(9) Forward error

$$|S_n - fl(S_n)| \leq \gamma_n \sum_{i=1}^{n} |x_i| = \gamma_n \|x\|_1^2$$

Backward error

$$\gamma_n \frac{\|x\|_1^2}{|S_n|}$$

Programming Exercise

Introduction

In the programming part of the homework the main focus is on the algorithm for numerically solving dense linear systems of equations: **Ax = b** – Gaussian Elimination and LU factorization.

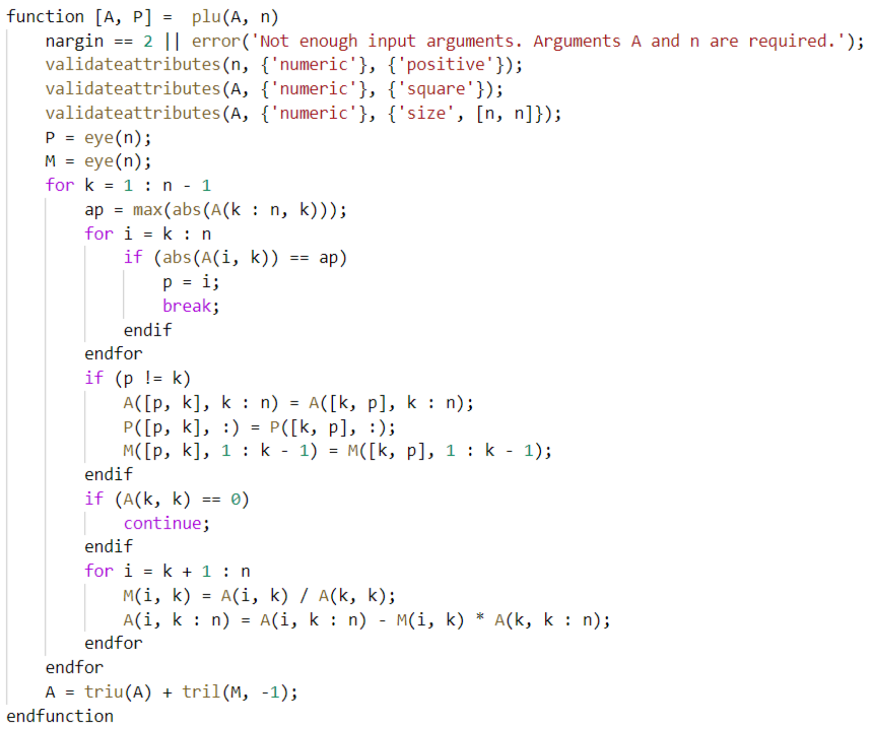
A = LU

LUx = b can be solved by solving **Ly = b** (forward substitution) and **Ux = y** (Backward substitution).

I used Octave 6.3.0 for the exercises.

Part 1

In the first part we need to implement the standard “scalar” (unblocked) algorithm (three nested loops) in two versions: **with partial pivoting** and **without partial pivoting.** On the picture below you can find my implementation of the algorithm with partial pivoting.



```
function [A, P] =  plu(A, n)
    nargin == 2 || error('Not enough input arguments. Arguments A and n are required.');
    validateattributes(n, {'numeric'}, {'positive'});
    validateattributes(A, {'numeric'}, {'square'});
    validateattributes(A, {'numeric'}, {'size', [n, n]});
    P = eye(n);
    M = eye(n);
    for k = 1 : n - 1
        ap = max(abs(A(k : n, k)));
        for i = k : n
            if (abs(A(i, k)) == ap)
                p = i;
                break;
            endif
        endfor
        if (p != k)
            A([p, k], k : n) = A([k, p], k : n);
            P([p, k], :) = P([k, p], :);
            M([p, k], 1 : k - 1) = M([k, p], 1 : k - 1);
        endif
        if (A(k, k) == 0)
            continue;
        endif
        for i = k + 1 : n
            M(i, k) = A(i, k) / A(k, k);
            A(i, k : n) = A(i, k : n) - M(i, k) * A(k, k : n);
        endfor
    endfor
    A = triu(A) + tril(M, -1);
endfunction
```

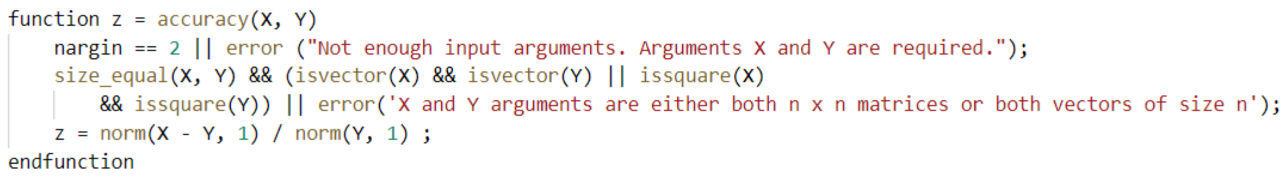
First, we validate the arguments in the function **plu(A, n)**. After validation the actual algorithms starts. We initialize matrices **P** (permutation matrix) and **M** (elimination matrix) and do Gaussian elimination. After that the **L** matrix is stored in **M** and **U** matrix is stored in **A**. The function returns result in the form: **A = L + U – I** and **P.**

The function **wplu(A, n)** performs LU factorization without partial pivoting. It works without rows interchanging.

The correctness of the LU factorization algorithms implementation was verified by evaluating the relative residual:

$$R = \frac{\|P^T LU - A\|_1}{\|A\|_1}$$

The implementation of accuracy is shown below:



```
function z = accuracy(X, Y)
    nargin == 2 || error ("Not enough input arguments. Arguments X and Y are required.");
    size_equal(X, Y) && (isvector(X) && isvector(Y) || issquare(X)
        && issquare(Y)) || error('X and Y arguments are either both n x n matrices or both vectors of size n');
    z = norm(X - Y, 1) / norm(Y, 1) ;
endfunction
```

Matrix **A** is randomly generated using the **rand(n)** Octave function. The problem sizes are as follows: **N = [2 : 10, 15 : 5 : 50, 60 : 10 : 150, 200 : 50 : 500, 600 : 100 : 1000]**

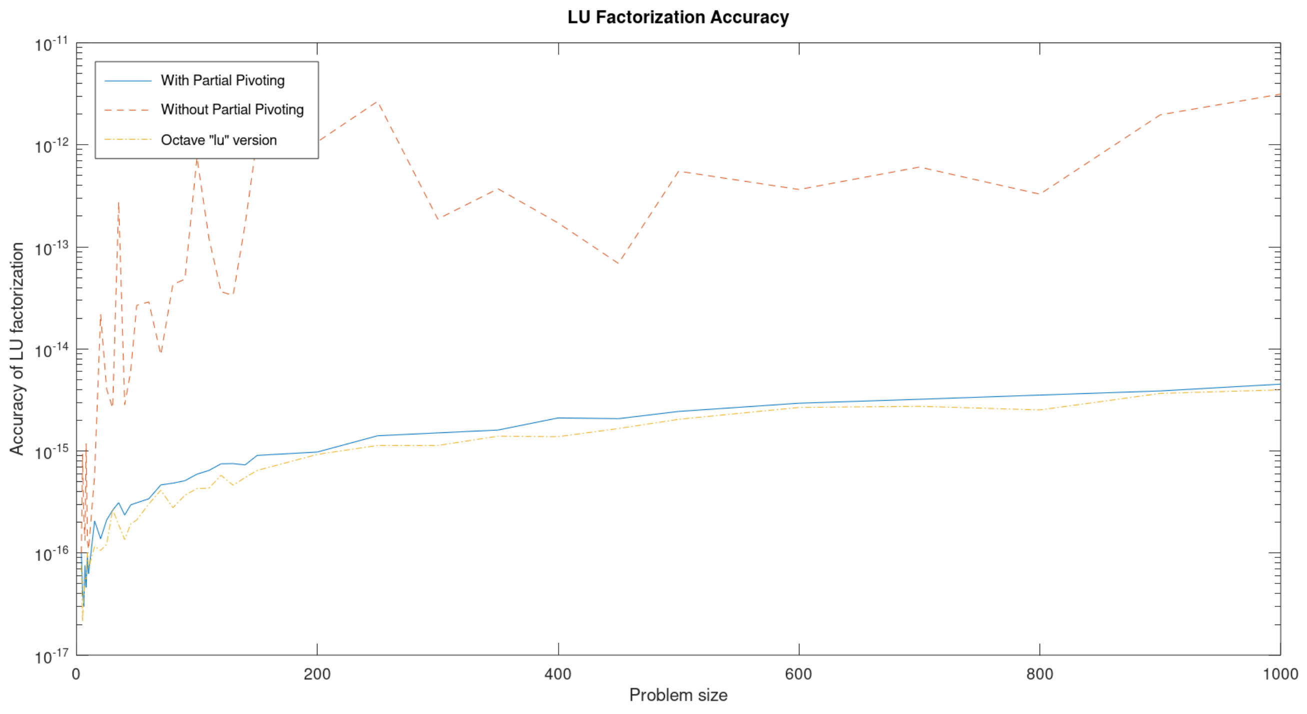
For each **n** random matrix **A** is generated. Then the LU factorization function is called to find **L**, **U** and **P** matrixes, after that the accuracy is calculated for original matrix and matrix **A** got by formula: **P' * L * U**. These steps are repeated for custom LU factorization with and without pivoting and the Octave **lu(A)** function.

```
[LU, P] = plu(A, n);
L = tril(LU, -1) + eye(n);
U = triu(LU);
Y_plu(1, i) = accuracy(P' * L * U, A);

LU = wplu(A, n);
L = tril(LU, -1) + eye(n);
U = triu(LU);
Y_wplu(1, i) = accuracy(L * U, A);

[L, U, P] = lu(A);
Y_lu(1, i) = accuracy(P' * L * U, A);
```

The results are presented in the chart below. We can see that custom LU factorization with partial pivoting performs almost as good as Octave version. But LU factorization without partial pivoting performs relatively worse.



Part 2

In part 2 of the programming exercises two extra routines were implemented: Forward and Backward Substitution - which solve **n x n** triangular systems. Using forward substitution, we can find **x** from the lower triangular system: **Lx = b**. Backward substitution allows us to find **x** in upper triangular linear system: **Ux = b**.

The formula for solving lower triangular linear systems: $x_1 = b_1 / L_{1,1}$, $x_2 = (b_2 - L_{2,1}x_1) / L_{2,2}$ ...

We can see here that on each iteration we divide the result by **L(i, i)**. But it our case the diagonal values in matrix **L** are always equal to 1, so the division can be skipped. Custom implementation of the **solveL(B, b, n)** function is presented below. The input argument **B** is a matrix that is equal to **L + U – I** where **I** is the Identity matrix, **L** and **U** are lower and upper triangular matrices.

```
function x = solveL(B, b, n)
    nargin == 3 || error('Not enough input arguments. Arguments B, b and n are required.');
    validateattributes(n, {'numeric'}, {'positive'});
    validateattributes(B, {'numeric'}, {'size', [n, n]});
    validateattributes(b, {'numeric'}, {'size', [n, 1]});
    L = tril(B, -1) + eye(n);
    x = zeros(n, 1);
    for i = 1 : n
        x(i) = b(i);
        b(i + 1 : n) = b(i + 1 : n) - L(i + 1 : n, i) * x(i);
    endfor
endfunction
```

Backward substitution works as forward substitution but from the backwards. Here we can't skip division by **U(i, i)** because upper triangular matrix has values on Its diagonal. Custom implementation of the Backward substitution is presented below.

```
function x = solveU(B, b, n)
    nargin == 3 || error('Not enough input arguments. Arguments B, b and n are required.');
    validateattributes(n, {'numeric'}, {'positive'});
    validateattributes(B, {'numeric'}, {'size', [n, n]});
    validateattributes(b, {'numeric'}, {'size', [n, 1]});
    U = triu(B);
    x = zeros(n, 1);
    for i = n : -1 : 1
        if (U(i, i) == 0)
            continue;
        endif
        x(i) = b(i) / U(i, i);
        b(1 : i - 1) = b(1 : i - 1) - U(1 : i - 1, i) * x(i);
    endfor
endfunction
```

The accuracy of the code was evaluated for increasing **n** in terms of relative residual and forward error (accuracy, see part 1). The following problem sizes were chosen: **N = [1 : 10, 15 : 5 : 50, 60 : 10 : 150, 200 : 50 : 500, 600 : 100 : 1000]**. Relative residual is calculated by the following formula:

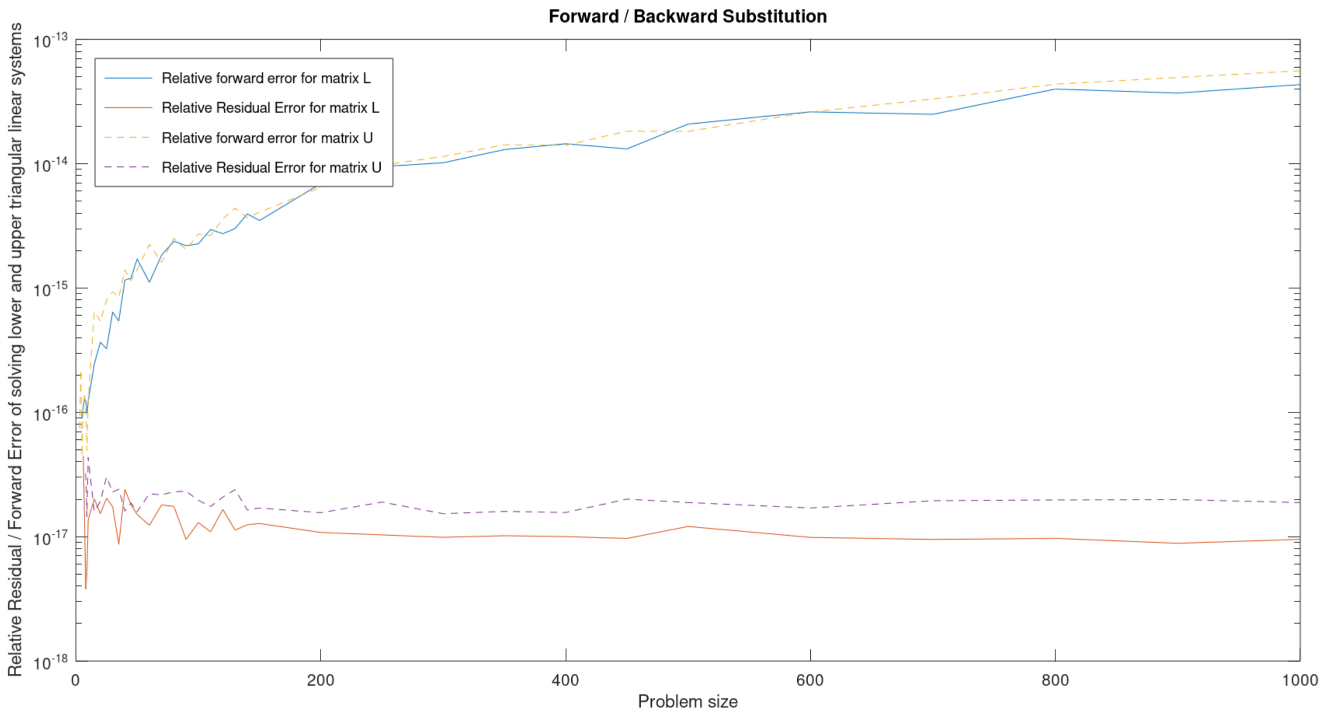
$$r := \frac{||A\hat{x} - b||_1}{||A||_1||\hat{x}||_1}.$$

The implementation of residual is presented below:

```
function r =  residual(A, x_hat, b)
    nargin == 3 || error('Not enough input arguments. Arguments A, x_hat and b are required.');
    validateattributes(A, {'numeric'}, {'square'});
    validateattributes(x_hat, {'numeric'}, {'vector'});
    validateattributes(b, {'numeric'}, {'vector'});
    isequal(rows(x_hat), rows(b), columns(A)) || error('A is an n x n matrix, x_hat, b are vectors of size n');
    r = norm((A * x_hat) - b, 1) / (norm(A, 1) * norm(x_hat, 1));
endfunction
```

For these experimental evaluations, the random (non-singular) **L** and **U** were generated using **rand(n)** Octave function. The singularity of the matrix is checked using rank. If rank of matrix **n x n** equals **n**, then the matrix is non-singular. Also, if we perform basic operations on the non-singular matrix, it stays non-singular. So, the LU factorization is performed on random matrix to get **L** and **U**.

The vector **b** is determined such that the exact solution **x** is a vector of all ones: **x = (1, 1, …, 1, 1)^T**. To get the solution as all ones, vector **b** should be equal to sum of all the rows values in matrix **A**. The vector **b** is calculated using Octave functions **sum(L, 2)** and **sum(U, 2)**.



The results of second part of the assignment are presented on the figure above. We can see that the relative residual is very small, but it cannot lead to the conclusion that the solution is accurate. Solving linear triangular system in terms of both upper and lower matrices is very accurate for small **n** and becomes worse with increasing **n**.

Part 3

The main purpose of the third part is to evaluate the numerical accuracy of the linear systems solver implemented in Parts 1 and 2 with and without partial pivoting for different test matrices and to compare it with the built-in solver from Octave (using the \ operator: **x = A \\ b**).

In order to create a LU-based linear solver we need to combine LU factorization routines implemented in part 1 with triangular linear system solvers implemented in part 2.

The LU-based linear solver with partial pivoting is implemented in the following way:

```
function x = linSolve(A, b, n)
    nargin >= 3 || error('Not enough input arguments. Arguments A, b and n are required.');
    validateattributes(n, {'numeric'}, {'positive'});
    validateattributes(A, {'numeric'}, {'size', [n, n]});
    validateattributes(b, {'numeric'}, {'size', [n, 1]});

    [LU, P] = plu(A, n);
    y = solveL(LU, P * b, n);
    x = solveU(LU, y, n);
endfunction
```

It takes **A**, **n** and **b** arguments and solves the system **Ax = b**. First, it uses LU factorization to find **L** and **U** matrices and then used **solveL()** and **solveU()** functions to solve lower and upper triangular linear systems in order to find the **x** vector.

The same is done for the LU factorization without partial pivoting:

```
function x = wlinSolve(A, b, n)
    nargin >= 3 || error('Not enough input arguments. Arguments A, b and n are required.');
    validateattributes(n, {'numeric'}, {'positive'});
    validateattributes(A, {'numeric'}, {'size', [n, n]});
    validateattributes(b, {'numeric'}, {'size', [n, 1]});

    LU = wplu(A, n);
    y = solveL(LU, b, n);
    x = solveU(LU, y, n);
endfunction
```

Part 3.1

The first test matrix is generated randomly with entries uniformly distributed in the interval [-1,1].

```
S = rand(n) * 2 - 1;
```

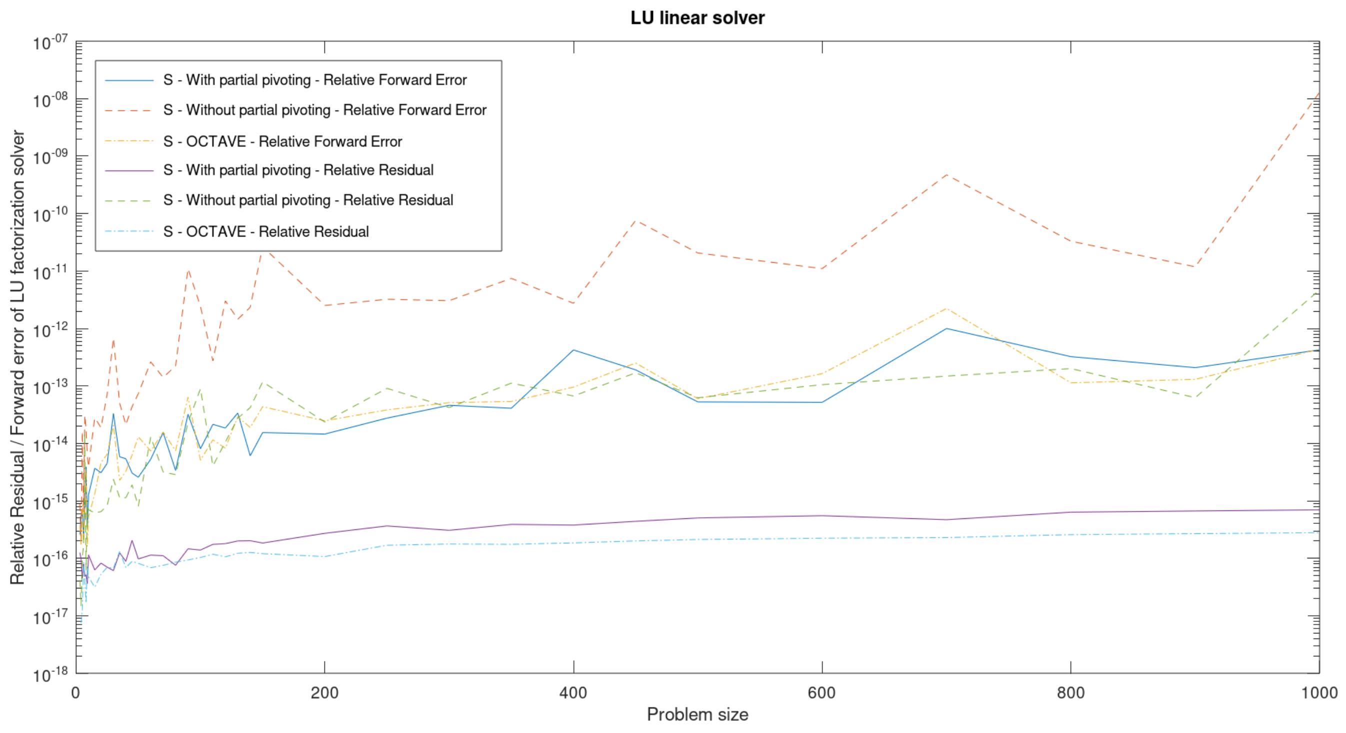
where **n** is in range: **N = [1 : 10, 15 : 5 : 50, 60 : 10 : 150, 200 : 50 : 500, 600 : 100 : 1000]**.

The vector **b** as in part 2 is calculated such the solution **x** is all ones.

The resulted accuracy of LU-based linear solver is presented below. As we can see on the figure the accuracy of solver with partial pivoting is high on matrix **S** and has a small tendency to decrease with increasing **n** (works as good as Octave version). The solver without partial pivoting performs worse in terms of accuracy.

The residual of Octave solver and custom solver with partial pivoting is very small. And for solver without partial pivoting, it is relatively larger.

We can conclude that LU-based solver with partial pivoting performs well on this type of matrix. Using pivoting we can avoid problems caused by singularity of the matrix (or at least avoid some of them), this why it performs better than the solver without pivoting.

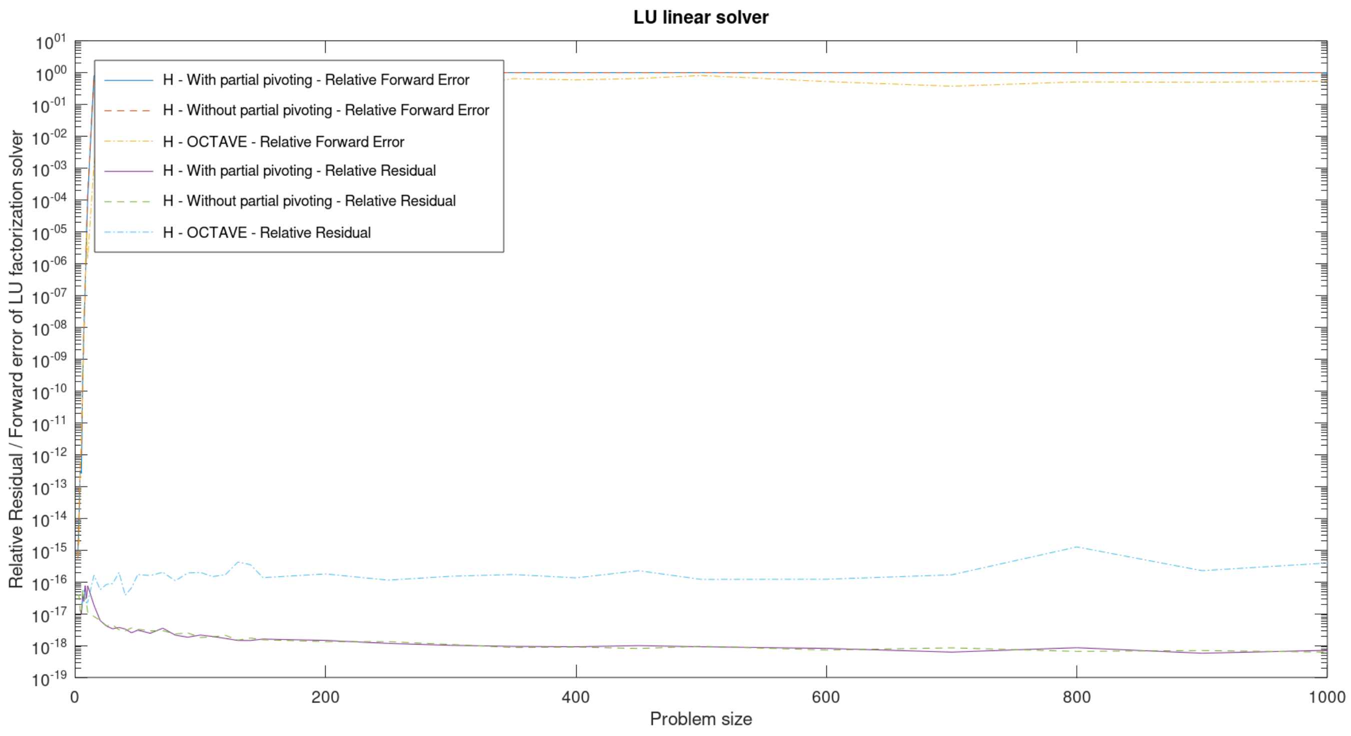


Part 3.2

The second matrix is generated using the following algorithm:

$$H_{ij} := \frac{1}{i+j-1} \quad for\ i = 1, \dots, n \text{ and } j = 1, \dots, n.$$

The resulted accuracy of LU-based linear solver is presented below. The residual of Octave solver and custom solver with and without partial pivoting is very small. This tells us that the solver probably performs very well. But as we can see on the figure the accuracy of all the solvers is very low on matrix **H**. This is happening because the algorithm produces a lot of singular matrixes with increasing **n**. And the structure of the matrix (how values are placed in there) prevents the LU factorization from performing pivoting. So, both solvers with and without pivoting perform bad on this type of matrix.


LU linear solver
H - With partial pivoting - Relative Forward Error
H - Without partial pivoting - Relative Forward Error
H - OCTAVE - Relative Forward Error
H - With partial pivoting - Relative Residual
H - Without partial pivoting - Relative Residual
H - OCTAVE - Relative Residual
Relative Residual / Forward error of LU factorization solver
Problem size