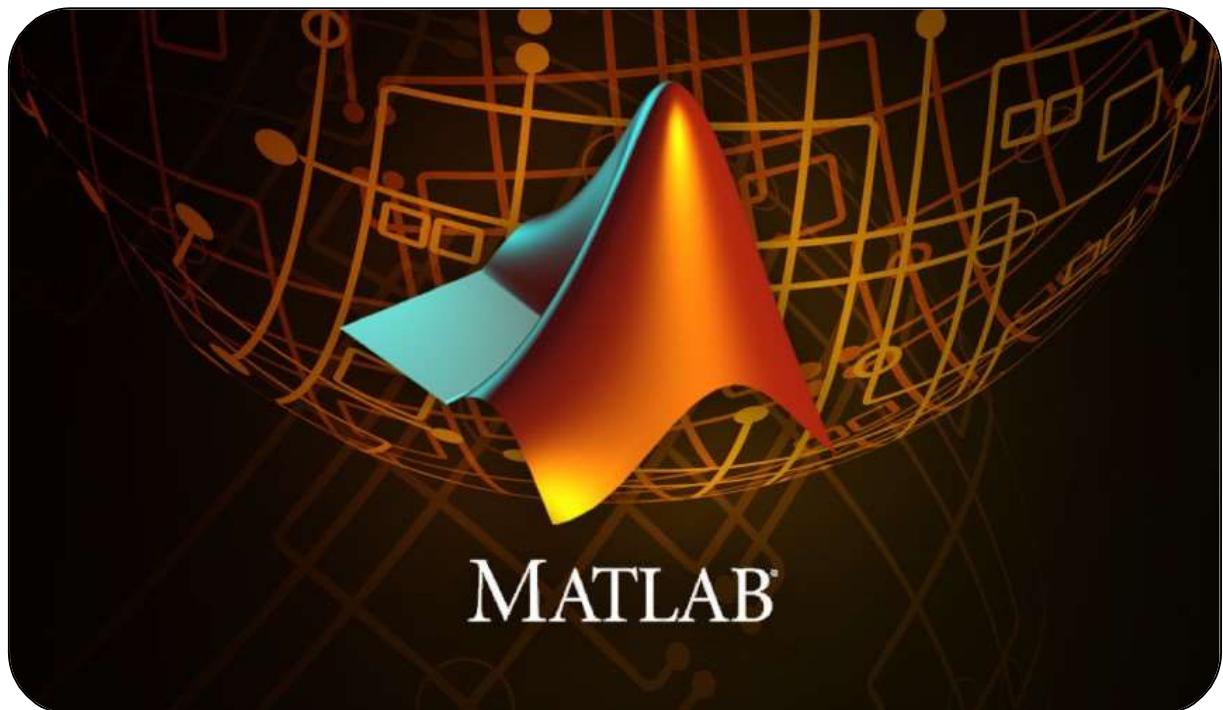


ŁUKASZ STANISZEWSKI, NR INDEKSU: 304098

METODY NUMERYCZNE

PROJEKT 1



1 ZADANIE 1 – DOKŁADNOŚĆ MASZYNOWA KOMPUTERA

Celem tego zadania było napisanie programu wyznaczającego dokładność maszynową komputera i uruchomienie go na swoim komputerze.

W arytmetyce zmiennopozycyjnej występują dane wejściowe obarczone błędami reprezentacji maszynowej jak i elementarne operacje oraz funkcje, które generują błędy numeryczne. Standard IEEE754 narzucił komputerom wymóg zorganizowania jednostki arytmetycznej tak, aby wartość bezwzględna z błędu zaokrąglenia (ϵ) wyniku działań elementarnych była nie większa od dokładności maszynowej komputera (eps), co można zapisać (wzór 1.5):

$$|\epsilon| \leq eps.$$

W ogólnym ujęciu dokładność maszynową eps można określić jako taką najmniejszą dodatnią liczbę maszynową g , że wynik działania zmiennopozycyjnego fl polegającego na dodaniu tej liczby do 1 będzie wciąż liczbą większą od 1 , co można zapisać (wzór 1.6):

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

gdzie M oznacza zbiór liczb maszynowych oraz $M \subset \mathbb{R}$.

Przy założeniu, że podstawą w reprezentacji zmiennoprzecinkowej liczb jest 2, zaczynając od $g = 2^0 = 1$, w pętli dzieliłem liczbę g przez 2 do momentu, aż kolejny taki iloraz zsumowany z liczbą 1 przestanie dawać liczbę większą od 1, co jest realizowane w funkcji *precision.m*, której kod wygląda następująco:

```
function [eps] = precision()
% REPREZENTACJA Oblicza dokładność maszynową (eps) komputera.
% deklaracja + inicjalizacja zmiennych
eps = 1; % eps reprezentuje g przed podzieleniem
% (najbardziej aktualne g, ale spełniające warunek)
g = 1; % g jest w kolejnych iteracjach pętli dzielone przez 2
fl = 1.0 + g; % wynik zsumowania aktualnej wersji g z 1
while fl > 1.0
    % wynik powinien zwracać takie g, które spełnia dalej warunek dlatego
    % konieczne jest aby eps zapamiętywało g przed podzieleniem
    eps = g;
    g = g / 2.0;
    fl = 1.0 + g;
end
end
```

A wynikiem działania programu jest:

```
>> eps = precision()
eps =
    2.2204e-16
```

Wynik ten oznacza, że dokładność maszynowa eps mojego komputera wynosi $2.2204 \cdot 10^{-16} = 2^{-52}$, co jest zgodne ze standardem **IEEE754** dla liczb w podwójnej precyzji (gdzie mantysa zapisywana jest właśnie na **52 bitach**).

2 ZADANIE 2 – ELIMINACJA GAUSSA Z CZĘŚCIOWYM WYBOREM ELEMENTU PODSTAWOWEGO

Metoda eliminacji Gaussa jest skończoną metodą rozwiązywania układów równań liniowych postaci $Ax = b$. Metoda ta realizowana jest w dwóch etapach:

1. Eliminacji zmiennych wykonywanym w n krokach, gdzie w wyniku odpowiednich przekształceń zarówno na macierzy A jak i wektorze b , otrzymamy równoważny układ równań z macierzą trójkątną górną w miejscu macierzy A .
2. Zastosowania algorytmu rozwiązywania układu równań z macierzą trójkątną górną.

Jej wersja realizowana w tym zadaniu – z częściowym wyborem elementu podstawowego – wprowadza do etapu eliminacji zmiennych w każdym kroku k konieczność wybrania ze zbioru elementów macierzy A z kolumny k z wierszy o indeksie niemniejszym niż k , element o największym module (wzór 2.28):

$$|a_{ik}^{(k)}| = \max_j \{|a_{jk}^{(k)}|, j = k, k+1, \dots, n\}.$$

i zamianę w macierzy rozszerzonej miejscami wiersza i wraz z wierszem k .

Działanie te nie dość, że prowadzi do mniejszych błędów numerycznych, to również eliminuje sytuację, gdy w k -tym kroku $a_{kk}^{(k)} = 0$. Tak więc realizację całego algorytmu można opisać w punktach:

1. Połączenie macierzy A oraz wektora b w macierz rozszerzoną Ab :

$$\left[\begin{array}{ccc|c} a_{11} & \cdots & a_{1n} & b_1 \\ \vdots & \ddots & \vdots & \vdots \\ a_{n1} & \cdots & a_{nn} & b_n \end{array} \right]$$

2. W $n - 1$ krokach wykonanie etapu eliminacji zmiennych, gdzie k -ty krok tego działania to zamiana **k -tego** wiersza macierzy rozszerzonej z wierszem i , w którym znajduje się wcześniej opisany element o największym module (wybierany częściowo), a następnie wykonanie działania:

$$w_i = w_i - \frac{a_{ik}^{(k)}}{a_{kk}^{(k)}} * w_k, \text{ gdzie } i = k+1, \dots, n \text{ oraz } w_i \text{ to } i\text{-ty wiersz macierzy rozszerzonej.}$$

3. W wyniku powyższego etapu, po $n - 1$ krokach, uzyskany układ równań to $A^{(n)}x = b^{(n)}$, gdzie $A^{(n)}$ to macierz trójkątna górna.
4. Na końcu, w celu zdobycia wektora x , koniecznym jest rozwiązanie układu $A^{(n)}x = b^{(n)}$, stosując klasyczny algorytm rozwiązywania układu równań z macierzą trójkątną polegający na przejściu od wiersza ostatniego w górę i wykonaniu działania (wzór 2.17):

$$x_k = \frac{(b_k - \sum_{j=k+1}^n a_{kj} x_j)}{a_{kk}} \text{ dla } k = n, n-1, n-2, \dots, 1.$$

Uwaga: w swojej realizacji algorytmu postanowiłem działania z punktów 2 oraz 4 wykonać bezpośrednio na macierzy rozszerzonej, a nie na macierzy A i wektorze b oddzielnie, aby uprościć kod.

Powyżej przedstawiony schemat działań w celu obliczenia rozwiązania układu równań metodą eliminacji Gaussa z wyborem elementu podstawowego został zrealizowany w funkcji `solveGaussPartial.m`.

```
function [x] = solveGaussPartial(A, b)
% SOLVEGAUSSPARTIAL Wyznacza rozwiązanie układu równań liniowych Ax=b.
n = size(A, 1); % liczba równań
Ab = [A b]; % dla ułatwienia łączymy poziomo A z b (macierz rozszerzona)
for k=1:n % k to krok eliminacji gaussa
    % wybór elementu głównego
    main_el = abs(Ab(k,k));
    main_el_ind = k;
    for j = k+1:n
        if abs(Ab(j, k)) > main_el
            main_el = abs(Ab(j, k));
            main_el_ind = j;
        end
    end
    % zamiana wierszy
    temp_row = Ab(main_el_ind,:);
    Ab(main_el_ind,:) = Ab(k, :);
    Ab(k, :) = temp_row;

    % eliminacja Gaussa - liczymy l_ik i odejmujemy pomnożone przez
    % wiersz k od wiersza i macierzy rozszerzonej
    for i=k+1:n
        l = Ab(i,k) / Ab(k,k);
        Ab(i,:)=Ab(i,:) - l * Ab(k, :);
    end
end
% teraz posiadamy już macierz trojkatna oraz przekształcony wektor
% prawych stron b i rozwiązujemy równanie od dołu
x = zeros(n,1);
for k = n:-1:1
    sum_x = 0;
    for j = k+1:n
        sum_x = sum_x + x(j)*Ab(k,j);
    end
    x(k) = (Ab(k,n+1) - sum_x)/Ab(k,k);
end
end
```

Na początku koniecznym było przetestowanie funkcji dla zadania wymiaru $n = 5$ z gęstą macierzą $A = GG^T$ i wektora b wygenerowanych losowo:

```
>> [A, b] = prepareTest();
>> x = solveGaussPartial(A, b)
```

x =

```
0.0060
0.8827
-0.5848
-0.2053
0.3134
```

```
>> norm(A*x-b,2)
```

ans =

```
1.9860e-14
```

Wynik ten oznacza, że dla zadania o małym wymiarze i gęstej macierzy, zaimplementowany przeze mnie algorytm sprawdza się dobrze.

Następnie, koniecznym było zastosowanie algorytmu do rozwiązywania układów równań o rozmiarze n dla $n = 5, 10, 50, 100, 200$.

W pierwszym kroku, dla macierzy A oraz wektora b z podpunktu (a) obliczyłem błąd rozwiązania $\varepsilon_1 = \|Ax - b\|$ dla każdego n przy użyciu skryptu:

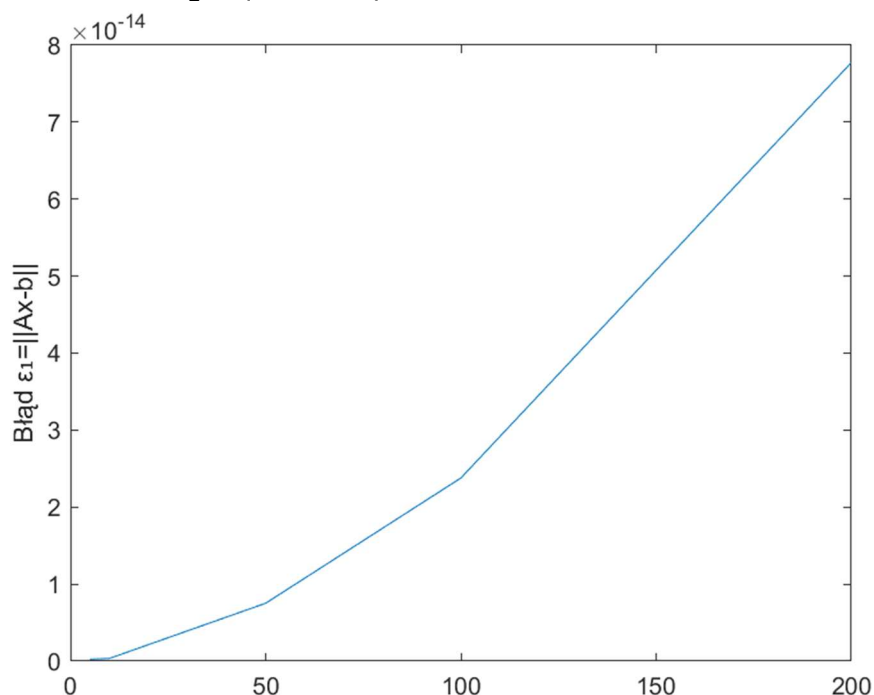
```
n_of_equations = [5, 10, 50, 100, 200]; % możliwe n
errors_1 = zeros(size(n_of_equations,2), 1); % wektor błędów
% wyznaczanie błędu dla każdego n
for n_idx = 1:size(n_of_equations,2)
    n = n_of_equations(n_idx);
    [A, b] = prepareMatricesA(n); % zdobycie macierzy A i wektora b
    x = solveGaussPartial(A, b);
    errors_1(n_idx,1) = norm(A*x-b,2);
end
% wyświetlenie wektora błędów
disp(errors_1);
% zobrazowanie zależności w postaci wykresu
plot(n_of_equations, errors_1);
xlabel("Liczba równań");
ylabel("Błąd  $\varepsilon_1 = \|Ax-b\|$ ");
```

W wyniku czego otrzymałem zestawienie liczby układów równań do błędu rozwiązania ε_1 :

n	5	10	50	100	200
$\varepsilon_1 = \ Ax - b\ $	$0.0025 * 10^{-13}$	$0.0035 * 10^{-13}$	$0.0752 * 10^{-13}$	$0.2378 * 10^{-13}$	$0.7774 * 10^{-13}$

Tabela 1 – zestawienie liczby układów równań n do błędu rozwiązania ε_1 dla macierzy i wektora z podpunktu (a) dla metody eliminacji Gaussa z częściowym wyborem el. podstawowego

oraz rysunek zależności $\varepsilon_1 = \|Ax - b\|$ do liczby równań n :



Wykres 1 – zależność błędu rozwiązania ε_1 do liczby układów równań n dla macierzy i wektora z podpunktu (a) dla metody eliminacji Gaussa z częściowym wyborem el. podstawowego

Następnie, w sposób analogiczny do podpunktu (a), dla macierzy A oraz wektora b z podpunktu (b) przeprowadziłem obliczenia, zamieniając w skrypcie `exercise2.m` linię kodu nr 6:

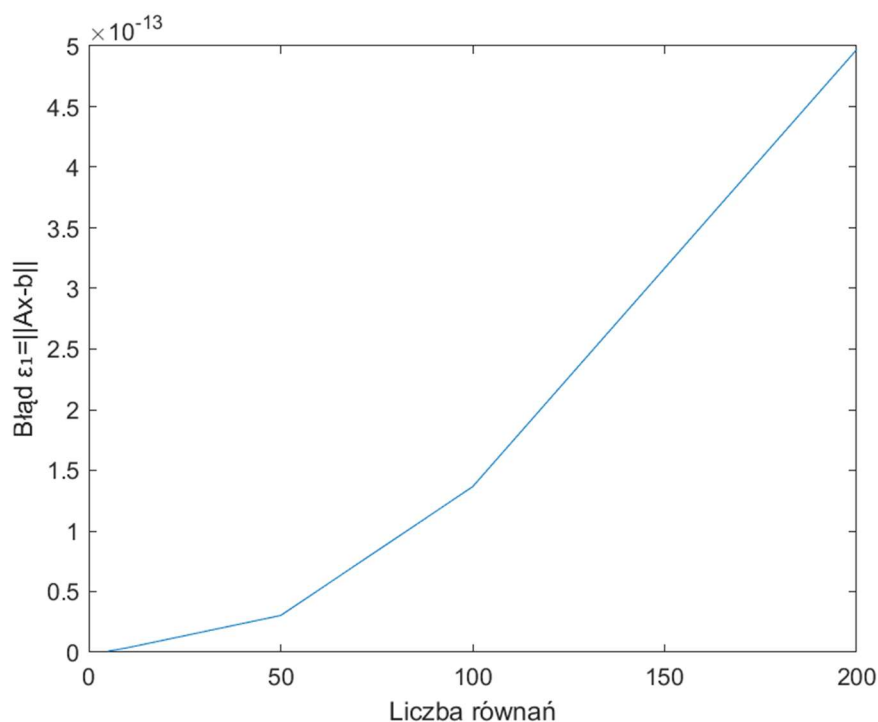
```
...
n = n_of_equations(n_idx);
[A, b] = prepareMatricesB(n); % zdobycie macierzy A i wektora b z (b)
x = solveGaussPartial(A, b);
...
```

W wyniku uruchomienia skryptu otrzymałem zestawienie tabelaryczne liczby układów równań do błędu rozwiązania ε_1 :

n	5	10	50	100	200
$\varepsilon_1 = \ Ax - b\ $	$0.0009 * 10^{-12}$	$0.0036 * 10^{-12}$	$0.0303 * 10^{-12}$	$0.1365 * 10^{-12}$	$0.4969 * 10^{-12}$

Tabela 2 – zestawienie liczby układów równań n do błędu rozwiązania ε_1 dla macierzy i wektora z podpunktu (b) dla metody eliminacji Gaussa z częściowym wyborem el. podstawowego

a także rysunek zależności $\varepsilon_1 = \|Ax - b\|$ do liczby równań n :



Wykres 2 – zależność błędu rozwiązania ε_1 do liczby układów równań n dla macierzy i wektora z podpunktu (b) dla metody eliminacji Gaussa z częściowym wyborem el. podstawowego

Na podstawie wyżej przeprowadzonych symulacji można stwierdzić, że błąd rozwiązania układu równań metodą eliminacji Gaussa z częściowym wyborem elementu podstawowego rośnie wielomianowo wraz ze zwiększaniem się liczby równań, które za jego pomocą trzeba rozwiązać. Dla punktu (a) i (b) największym uzyskanym błędem rozwiązania jest wynik $\varepsilon_1 \cong 7.7 * 10^{-14}$, co oznacza, że metoda ta z całkiem dużą dokładnością daje poprawne rozwiązanie.

3 ZADANIE 3 – METODA ITERACYJNA JACOBIEGO

Przy rozwiązywaniu układów równań liniowych można spotkać się z sytuacją, gdy problem będzie wielkowymiarowy, a macierze – rzadkie. W takich przypadkach przydatne okazać się może skorzystanie z iteracyjnych metod rozwiązywania układów równań liniowych, gdzie zaczynając z przybliżenia początkowego rozwiązania, w kolejnych krokach poprawiamy je, aby zbiegało do rozwiązania.

Jedną z takich metod jest metoda Jacobiego, polegająca na:

1. Zdekomponowaniu macierzy A na sumę macierzy poddiagonalnej L , macierzy diagonalnej D oraz macierzy naddiagonalnej U : $A = L + D + U$.
2. Dopóki nie będzie spełniony warunek stopu (w tym przypadku dopóki norma euklidesowa różnicy między kolejnymi przybliżeniami rozwiązania nie będzie mniejsza niż podane ε_2), wykonywaniu przybliżenia wektora x (wzór 2.59):

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

Dla metody Jacobiego istotne jest zwrócenie uwagi na to, że żeby ona zadziałała, macierz A musi spełniać dwa istotne założenia:

1. Macierz diagonalna D , na którą m.in. dekomponuje się macierz A , jest macierzą nieosobliwą (wartości na diagonalu A są niezerowe).
2. Aby metoda Jacobiego była zbieżna, macierz A musi charakteryzować się silną diagonalną dominacją wierszową lub kolumnową, tzn. musi spełniać jeden z poniższych warunków:
 - a. $|a_{ii}| > \sum_{j=1, j \neq i}^n |a_{ij}|$, dla każdego $i = 1, 2, \dots, n$ – **dominacja wierszowa**,
 - b. $|a_{jj}| > \sum_{i=1, i \neq j}^n |a_{ij}|$, dla każdego $j = 1, 2, \dots, n$ – **dominacja kolumnowa**.

Warunki te sprawdzane są z użyciem funkcji `checkDominance.m`:

```
function [is_dominant] = checkDominance(A)
%CHECKDOMINANCE Sprawdza czy macierz A jest silnie diagonalnie dominująca.
is_dominant_row = true;
is_dominant_col = true;
for i=1:size(A, 1)
    sum_row = 0;
    sum_col = 0;
    for j=1:size(A, 1)
        if i ~= j
            sum_row = sum_row + abs(A(i,j));
            sum_col = sum_col + abs(A(j,i));
        end
    end
    if abs(A(i,i)) <= sum_row
        is_dominant_row = false;
    end
    if abs(A(i, i)) <= sum_col
        is_dominant_col = false;
    end
end
if is_dominant_row || is_dominant_col
    is_dominant = true;
else
    is_dominant = false;
end
end
```

W wyniku sprawdzenia dominacji macierzy A z punktu (a):

```
>> b = checkDominance(A)
```

```
b =
```

```
logical
```

```
1
```

co oznacza, że macierz A posiada silną diagonalną dominację wierszową lub kolumnową, natomiast dla macierzy A z punktu (b):

```
>> b = checkDominance(A)
```

```
b =
```

```
logical
```

```
0
```

co oznacza, że macierz A nie posiada tu zarówno silnej diagonalnej dominacji wierszowej jak i kolumnowej i wtedy iteracyjnie wyliczany wektor x nie zbiega do rozwiązania.

Funkcja, rozwiązująca układ n równań liniowych metodą Jacobiego z podanym błędem granicznym ε_2 liczonym jako norma euklidesowa różnicy kolejnych przybliżeń x realizowana jest w `solveJacobi.m`:

```
function [x] = solveJacobi(A, b, e2)
    %SOLVEJACOBI Oblicza układ równań metodą Jacobiego
    % sprawdzenie dominacji diagonalnej
    if(checkDominance(A) == false)
        disp("Macierz bez silnej dominacji diagonalnej!");
        return;
    end
    % Stworzenie macierzy L, D i U
    n = size(A, 1); % n - liczba równań
    L = zeros(n);
    D = zeros(n);
    U = zeros(n);
    x = zeros(n, 1); % x to wektor wynikowy
    for i=1:n
        D(i,i) = A(i, i);
        for j=1:n
            if i>j
                L(i,j) = A(i,j);
            elseif i<j
                U(i, j) = A(i, j);
            end
        end
    end
    if(det(D)==0)
        disp("Macierz D nieosobliwa!");
        return;
    end
```



```

lim_error = e2;
% dopóki nie jest przekroczony błąd graniczny
while lim_error >= e2
    x_before = x;
    % tworzymy wektor x dla kolejnego kroku przybliżenia
    for j=1:n
        sum_k = 0;
        for k=1:n
            sum_k = sum_k + (L(j,k) + U(j,k))*x_before(k);
        end
        x(j) = -1 / D(j,j) * (sum_k - b(j));
    end
    % na koncu różnica w postaci normy z różnicy kolejnych przybliżeń
    lim_error = norm(x - x_before, 2);
end
end

```

Z powodu, że macierz A z podpunktu (b) nie posiada silnej dominacji, nie liczyłem błędu ε_1 dla macierzy A i wektora b z (b), tylko skupiłem się na wyliczaniu dokładności rozwiązania przykładu (a) w zależności od liczby układu równań n dla kilku możliwych wartości błędu granicznego ε_2 . Skrypt realizujący to znajduje się w pliku `exercise3.m`.

a) Dla błędu granicznego $\varepsilon_2 = 0.00000001$:

n	$\varepsilon_1 = \ Ax - b\ $
5	$0.7167 * 10^{-7}$
10	$0.7687 * 10^{-7}$
50	$0.8318 * 10^{-7}$
100	$0.8292 * 10^{-7}$
200	$0.8215 * 10^{-7}$

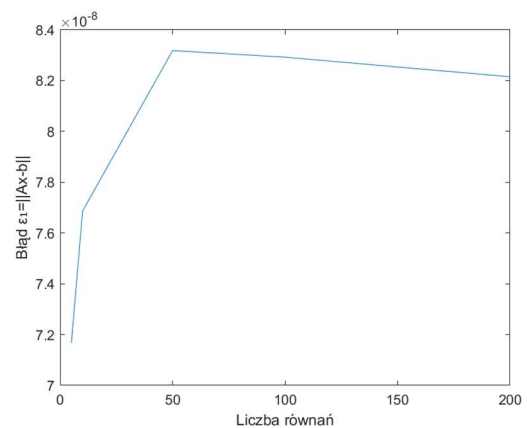


Tabela 3 i Wykres 3 – reprezentują zależność błęd rozwiązania ε_1 do liczby równań liniowych n do rozwiązania układu równań liniowych (a) dla metody Jacobiego dla błędu granicznego $\varepsilon_2 = 0.00000001$.

b) Dla błędu granicznego $\varepsilon_2 = 0.0001$:

n	$\varepsilon_1 = \ Ax - b\ $
5	$0.5564 * 10^{-3}$
10	$0.7585 * 10^{-3}$
50	$0.7950 * 10^{-3}$
100	$0.8926 * 10^{-3}$
200	$0.8580 * 10^{-3}$

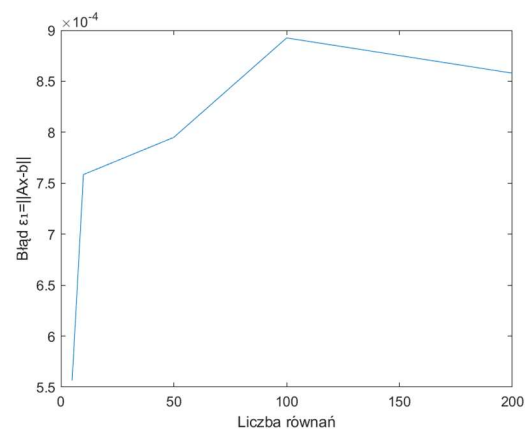


Tabela 4 i Wykres 4 – reprezentują zależność błęd rozwiązania ε_1 do liczby równań liniowych n do rozwiązania układu równań liniowych (a) dla metody Jacobiego i błędu granicznego $\varepsilon_2 = 0.0001$.

c) Dla błędu granicznego $\varepsilon_2 = 0.01$:

n	$\varepsilon_1 = \ Ax - b\ $
5	0.0692
10	0.0783
50	0.0875
100	0.0761
200	0.0879

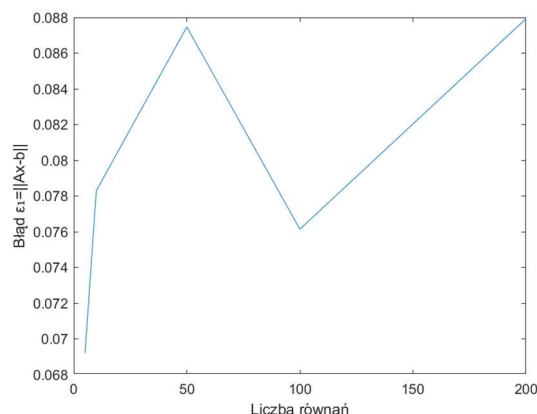


Tabela 5 i Wykres 5 – reprezentują zależność błędu rozwiązania ε_1 do liczby równań liniowych n do rozwiązania układu równań liniowych (a) dla metody Jacobiego i błędu granicznego $\varepsilon_2 = 0.01$.

W tym przypadku nie widać zależności wartości błędu rozwiązania ε_1 od liczby równań liniowych do rozwiązania n , co oznacza, że błąd wyniku będzie w przypadku metody Jacobiego nieprzewidywalny, jednak dla nisko dobranego parametru błędu granicznego (nie większego niż **0.001**), metoda daje poprawne wyniki z, w miarę, dużą dokładnością – można zauważyć, że otrzymana dokładność błędu rozwiązania ε_1 zależy tu od dokładności samego błędu granicznego ε_2 i w zależności od tego, jaki błąd rozwiązania jest dla nas akceptowalny, tak możemy sterować parametrem błędu granicznego.