Sprawozdanie projekt nr 1. MNUM

Autor: Marcel Kaliński

# Wstęp

Projekt zawiera w sobie 3 zadania:

1. Wyznaczenie dokładności maszynowej komputera
2. Implementacja algorytmu do rozwiązywania układów równań za pomocą metody eliminacji Gaussa z częściowym wyborem elementu podstawowego.
3. Implementacja algorytmu do rozwiązywania układów równań za pomocą metody Jacobiego.
4. Wyznaczenie błędów rozwiązań i sprawdzenie ich poprawności.

# Zad 1.

Dokładność maszynowa, oznaczana jako „eps” jest maksymalnym błędem względnym reprezentacji zmiennoprzecinkowej. Zależy ona jedynie od długości mantysy (liczby bitów). Dlatego też, jeśli komputery używają tych samych standardów zapisu liczb zmiennopozycyjnych a co za tym idzie długość mantysy jest taka sama, to i eps będzie na różnych komputerach taki sam, dla tych samych typów. Dokładność maszynową można też interpretować jako maksymalną wartość, która po dodaniu do danej liczby nie zmieni jej zapisu w komputerze. Wyraża się to następującym wzorem:

Rd(x) – x = x \* ℮, gdzie |℮| <= 2-t = eps i ‘t’ to liczba bitów mantysy

, czyli Rd(x) = x(1+℮), |℮| <= eps

Zaprojektowany przeze mnie algorytm do obliczania eps wygląda następująco:

function eps = my\_epsilon()

x = 1.0;

y = 2.0;

while (y > 1)

eps = x;

x = x/2;

y = 1.0 + x;

end

end

Wynikiem w środowisku Matlab jest zawsze 2.2204e-16. Wynika to z tego, że domyślnie liczby zmiennopozycyjne są tam zapisywane w standardzie IEEE® 754 jako liczby z podwójną precyzją, czyli ich mantysa ma długość 52 bitów. Wynik algorytmu zgadza się z wartością obliczoną teoretycznie ze wzoru

Eps = 2-t

W języku C/C++ ten sam algorytm będzie dawał różne wyniki, z tego względu, że można tam definiować jakim typem danych komputer ma się posługiwać. Dla tego też dla typu double wynik będzie taki sam jak w Matlabie, dla typu float będzie to 1.1902e-07 a dla long double 1.0842e-19.

Tak wyglądał algorytm dla języka C++.

#include<iostream>

int main()

{

//float or long double

double eps, x, y;

eps = 0;

x = 1.0;

y = 2.0;

while(y > 1)

{

eps = x;

x = x/2;

y = 1.0 + x;

}

std::cout << "machine epsilon equals " << eps << std::endl;

}

# Zad 2.

Metoda eliminacji gausa z częściowym wyborem elementu podstawowego składa się z 2 kroków:

1. Eliminacja Gaussa, czyli doprowadzenie macierzy do macierzy trójkątnej
2. Postępowanie odwrotne (ang *back substitution*), czyli podstawianie pod zmienne od końca.

Wybór elementu głównego następuje przy każdej pętli eliminacji zmiennych w kroku 1.

Tak wygląda zaimplementowana przeze mnie funkcja do eliminacji Gausa.

function [mat\_L, mat\_U, mat\_B] = gaussian\_eliminate(input)

Mat = input;

A\_dims = size(Mat);

n\_rows = A\_dims(1);

n\_cols = A\_dims(2);

mat\_L = zeros(n\_rows, n\_cols-1);

for k=1 : n\_rows

%find the row with the highest value

row = k;

col = k;

%temporary vector to find the greatest element(absolute value)

vec\_k = Mat(:, col);

%disp(vec\_k);

index\_max = find(vec\_k == max(abs(vec\_k(row:n\_rows))));

if ~(index\_max == k)

%swap row 'i' with 'k'

tmp = Mat(row , :);

Mat(row , :) = Mat(index\_max , :);

Mat(index\_max , :) = tmp;

end %end if

%element g?ówny jest wybrany -> (index\_max)

for n=k+1 : n\_rows

L = Mat(n, k) / Mat(k, k);

Wn = Mat(n, col:n\_cols);

Wk = Mat(k, col:n\_cols);

Mat(n, col:n\_cols) = Wn - L\*Wk;

mat\_L(n, k) = L;

end

end %end main for

mat\_U = Mat(:, 1:n\_cols-1);

mat\_B = Mat(:, n\_cols);

end %end function

Następnie kolejna funkcja przechwytuje macierze mat\_U i mat\_B i oblicza kolejne zmienne od końca.

function result = backsub(mat\_U, B)

A\_dims = size(mat\_U);

n\_rows = A\_dims(1);

result = zeros(n\_rows, 1);

for k=n\_rows : -1 : 1

sum = 0;

%counting sum

for j=n\_rows-1 : -1 : k

sum = sum + (mat\_U(k, j+1) \* result(j+1));

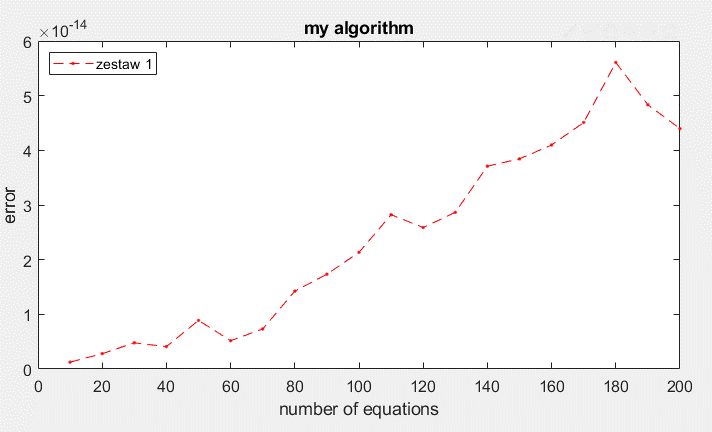
end

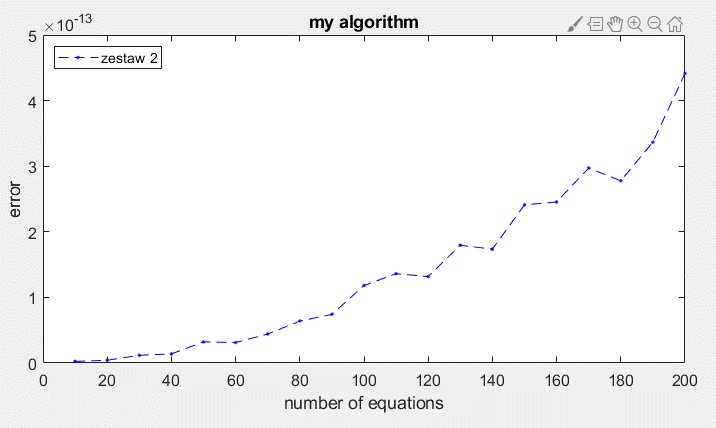
result(k) = (B(k) - sum) / mat\_U(k, k);

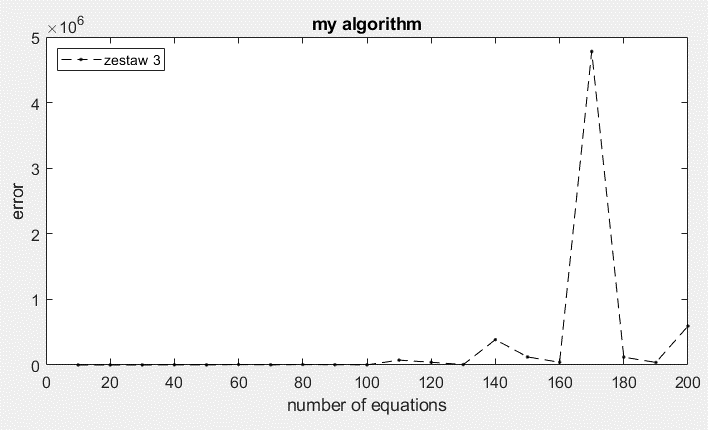
end

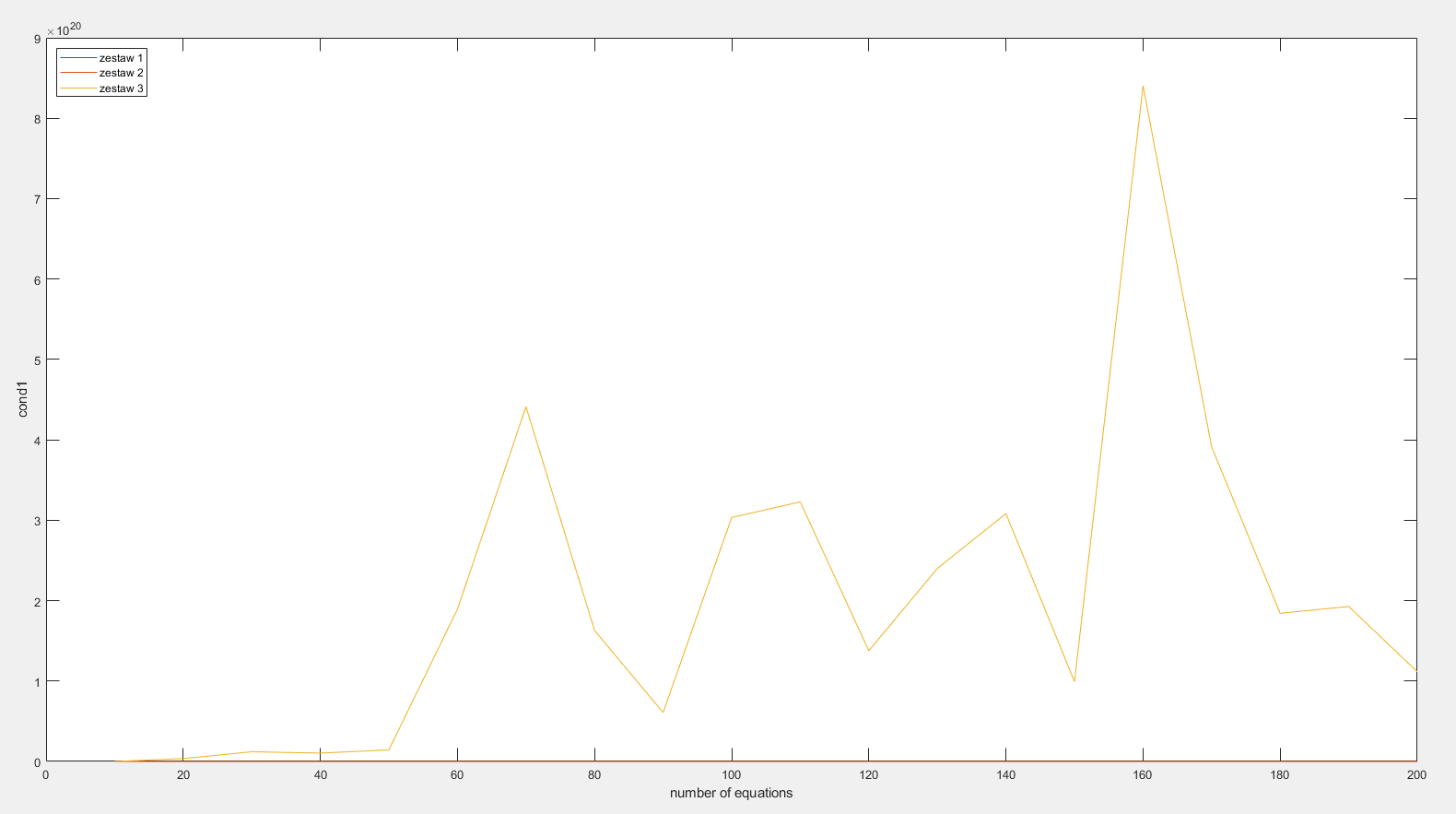
end

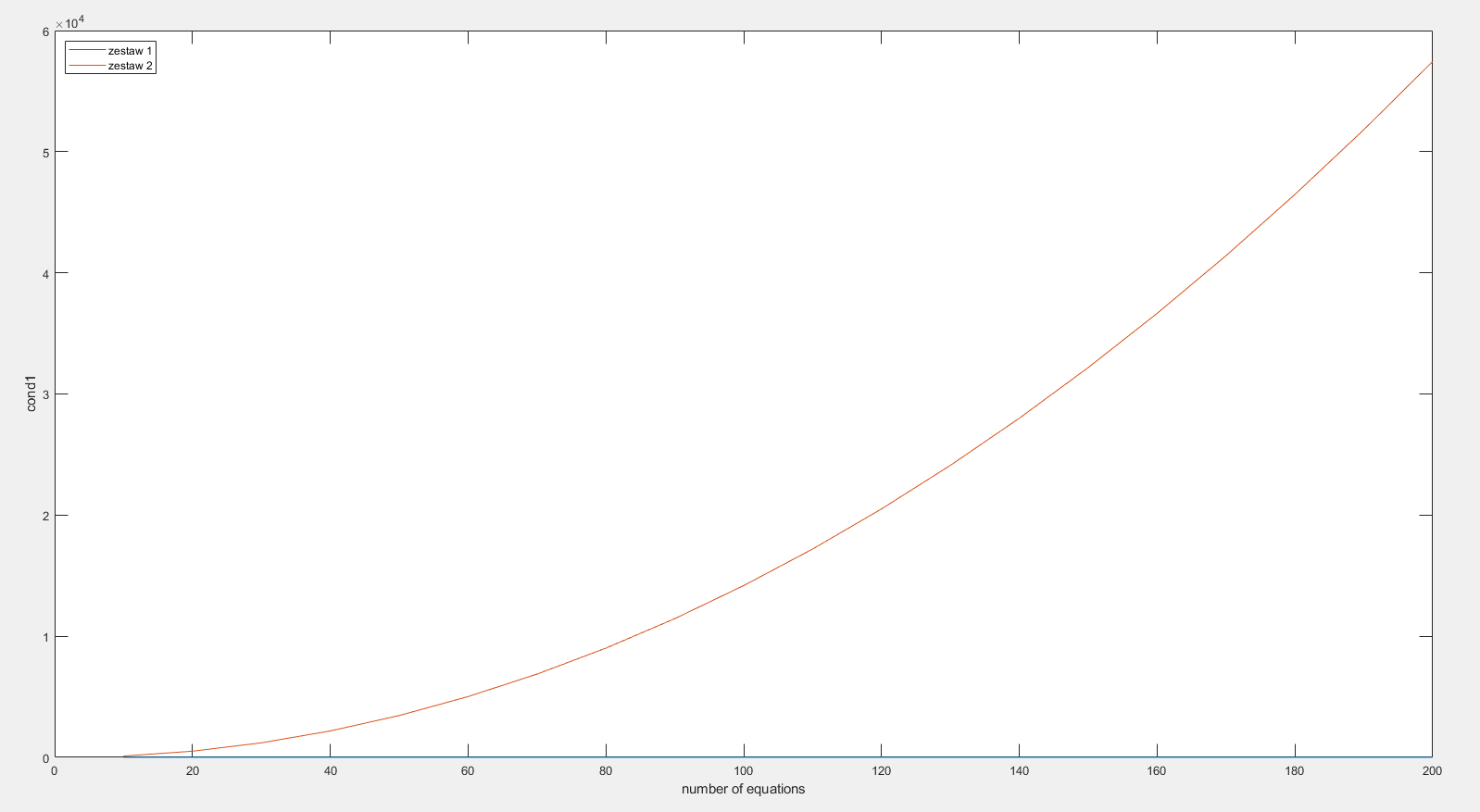
Poniżej przedstawię wykresy błędu w zależności od liczba równań w układzie dla 3 różnych zestawów danych. Pierwsze 2 zestawy wyglądają bardzo obiecująco, gdyż błędy są rzędu 10-14 , niestety nie można tego powiedzieć o zestawie 3, gdzie przy około 170 równaniach błąd rozwiązania wynosi prawie 5\*106.

Zestaw 1

Zestaw 2

Zestaw 3

Żeby dowiedzieć się skąd bierze się tak duży błąd w zestawie nr 3 policzyłem wskaźniki uwarunkowania dla paru przykładowych macierzy z każdego zestawu. Wyniki z tych pomiarów zaprezentuję na poniższym wykresie.

Widać dokładnie, że zestaw 3 ma bardzo duży wskaźnik uwarunkowania (rzędu 1020), w przeciwieństwie do zestawu 1 i 2, które w takiej skali mają wskaźnik bliski zeru. Dlatego też zamieszczam wykres tylko wskaźników dla zestawu 1 i 2.

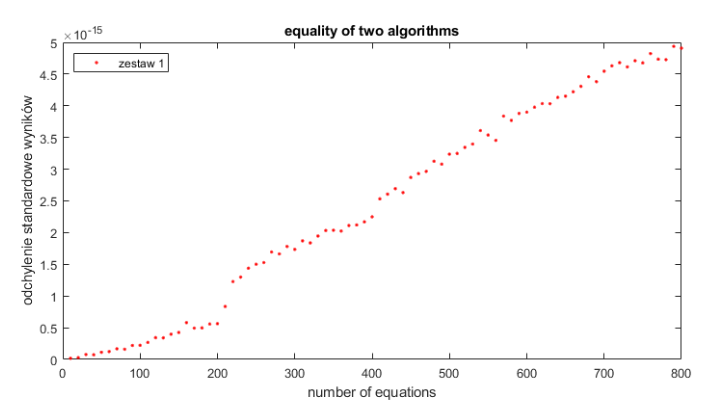
Tu znowu widać różnicę w wielkościach. Dla zestawu drugiego wskaźnik ten rośnie od 500 do wartości rzędu 104 , natomiast dla zestawu pierwszego znów jest on bliski zera. Żeby nie dodawać kolejnego wykresu wspomnę tylko, że wskaźnik uwarunkowania dla tego zestawu wynosił około 10, aby w granicy osiągnąć wartość 13.

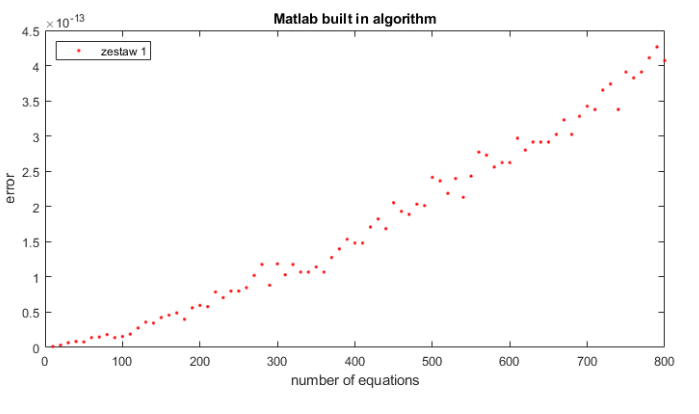
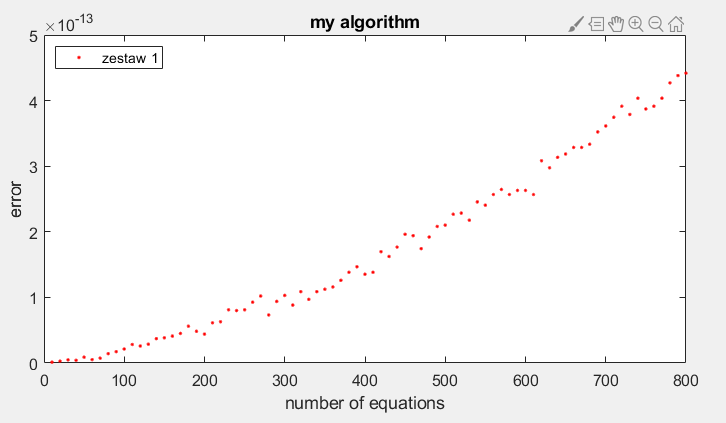
Do policzenia tych wskaźników posłużyłem się normą pierwszą używając następującej funkcji w Matlabie:

cond(mat, 1);

Znając wskaźniki uwarunkowania można stwierdzić, że zestaw 3 jest źle uwarunkowany, dlatego też nie powinno się rozwiązywać tego układu numerycznie.

Ostatnim zadaniem było sprawdzenie poprawności działania algorytmu. Zrobiłem to porównując wyniki do wyników zwróconych przez algorytm wbudowany w Matlaba (x = A\B). Rezultat przedstawię na kolejnych 3 wykresach, które będą dotyczyły danych z zestawu pierwszego.

Pierwszy wykres prezentuje różnicę między wynikami obu algorytmów. Jest ona rzędu 10-15, co pokazuje że mój algorytm daje poprawne wyniki.



# Zad 3.

Trzecie zadanie polegało na zaimplementowaniu programu rozwiązującego układy równań metodą Jacobiego.

W programie posłużyłem się następującym wzorem:

N = D-1

M = -N(L + U)

X(i+1) = Mxi + Nb , gdzie L, D, U to macierze: poddiagonalna, diagonalna i naddiagonalna.

Warunkiem dostatecznym zbieżności metody Jacobiego jest silna dominacja wierszowa lub kolumnowa, więc żeby niepotrzebnie nie liczyć macierzy, które i tak nie mogą zostać poprawnie policzone tą metodą stworzyłem krótką funkcję, która sprawdza, czy dana macierz spełnia jeden z danych warunków.

function [warunek\_spelniony] = check(mat)

%UNTITLED Summary of this function goes here

% Detailed explanation goes here

s = size(mat);

n = s(1);

A = mat(:, 1:n);

dominacja\_wierszowa = true;

dominacja\_kolumnowa = true;

for i=1 : n

val = A(i, i);

row\_i = A(i, :);

row\_i = sum(row\_i, 'all') - val;

if(abs(val) < abs(row\_i))

dominacja\_wierszowa = false;

break;

end

end

for i=1 : n

val = A(i, i);

col\_i = A(:, i);

col\_i = sum(col\_i, 'all') - val;

if(abs(val) < abs(col\_i))

dominacja\_kolumnowa = false;

break;

end

end

if(dominacja\_wierszowa==true || dominacja\_kolumnowa==true)

disp('warunek dostateczny spelniony');

warunek\_spelniony = true;

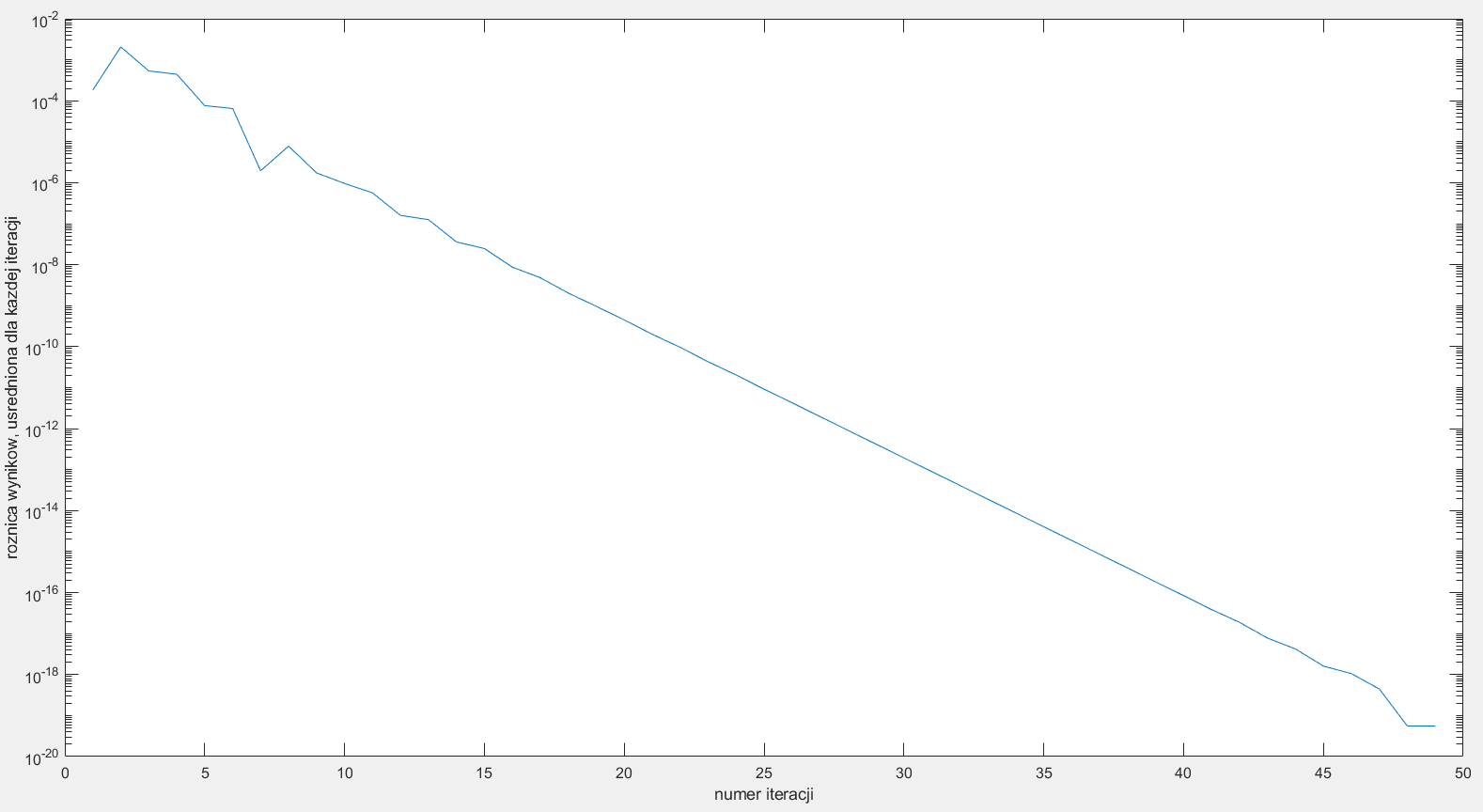
else

disp('warunek dostateczny NIE spelniony');

warunek\_spelniony = false;

end

end

Podstawiając macierz z zadania otrzymujemy wynik pozytywny, co znaczy, że algorytm powinien poprawnie rozwiązać ten układ. Tak też się dzieje, gdyż błąd rozwiązania wynosi zaledwie 4.409 \*10-16. Błąd był liczony dla algorytmu wykonującego 100 iteracji, natomiast, żeby dokładnie wyliczyć ile tak naprawdę ich potrzeba napisałem kolejną funkcję, która wyznacza różnice między kolejnymi iteracjami. Wykres przedstawiam poniżej.

Widać na nim dokładnie, że w przypadku tych danych wystarczy około 50 iteracji i różnica między kolejnymi wynikami maleje do wartości rzędu 10-18. Algorytm wygląda więc następująco:

function [X] = Jacobi\_solver(mat)

s = size(mat);

n\_rows = s(1); %liczba zmiennych w uk?adzie

A = mat(:, 1:n\_rows);

B = mat(:, n\_rows+1);

[L, D, U] = split(A);

a = D~=0;

D(a) = D(a).^-1;

N = D;

M = -N \* (L + U);

result = zeros(n\_rows, 1);

prev\_result = result;

NB = N\*B;

for x=1 : 50

for k=1 : n\_rows

s = 0;

for col=1 : n\_rows

s = s + M(k, col) \* prev\_result(col);

end

prev\_iter\_sum = s;

x\_k = prev\_iter\_sum + NB(k);

result(k) = x\_k;

end

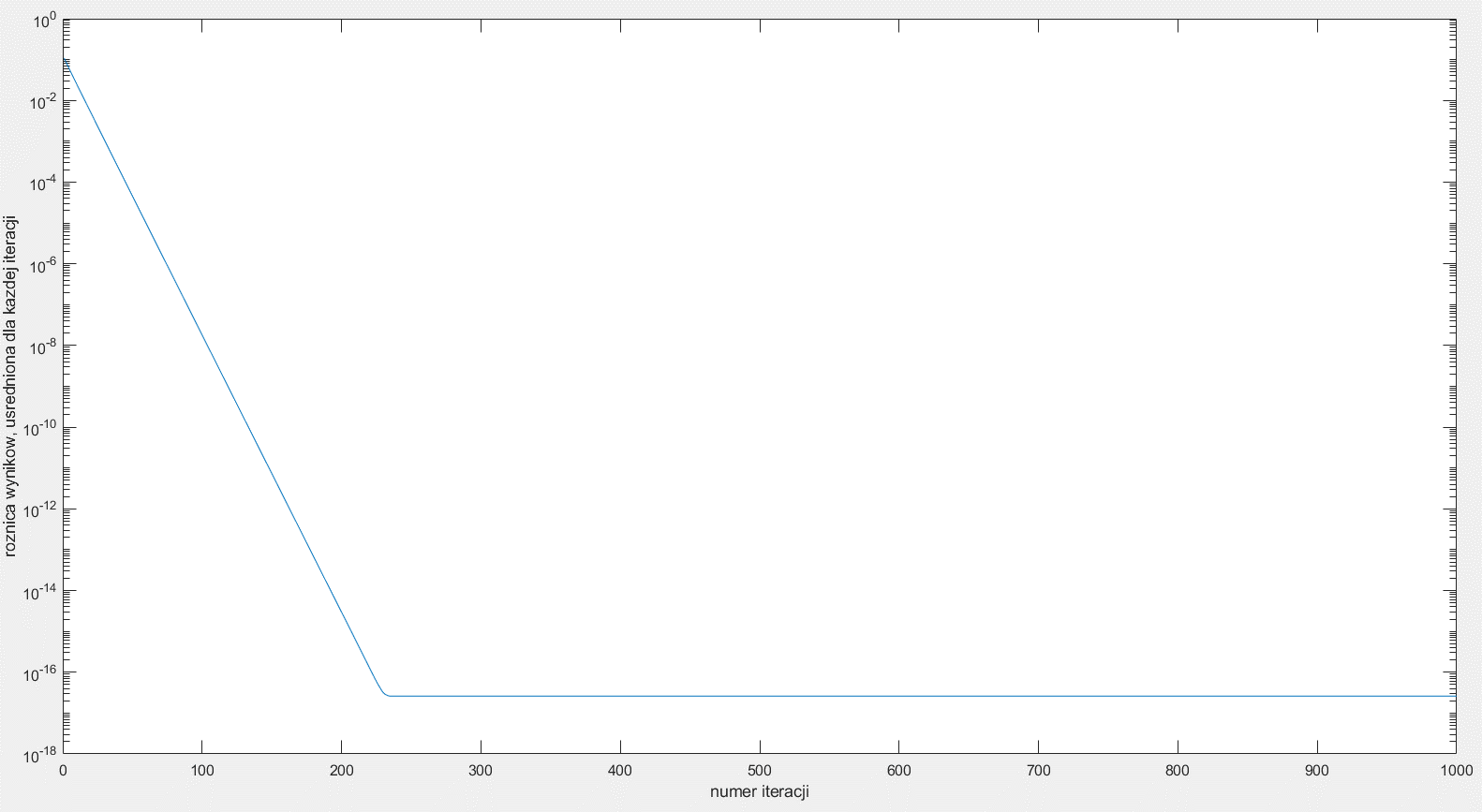
prev\_result = result;

end

X = result;

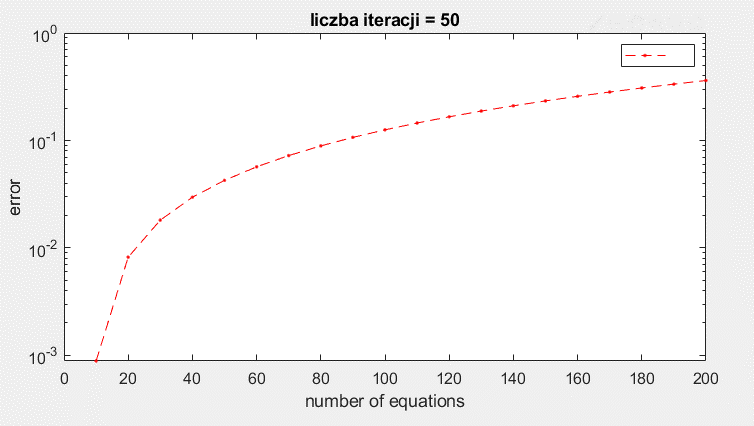
end

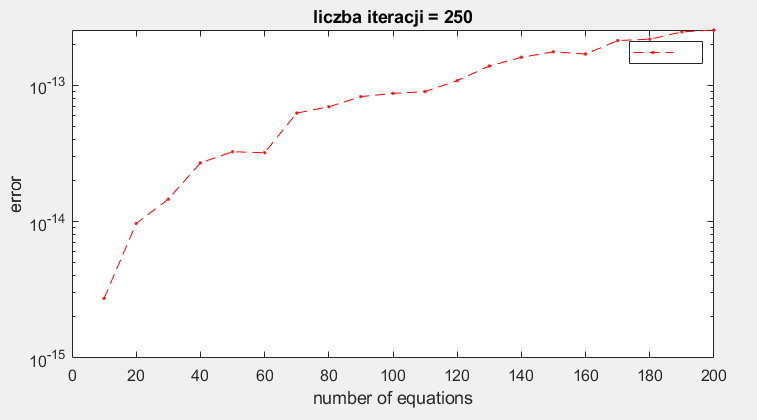
Trochę inaczej sprawa wygląda, gdy chcemy podstawić algorytm Jacobiego do rozwiązania układów z zadania 2.

Test stopu dla zestawu pierwszego wygląda następująco:

Na wykresie widać, że różnica osiąga asymptotycznie wartość 10-16 już po około 250 iteracjach, co oznacza, że algorytm Jacobiego dla tych danych jest zbieżny. Potwierdza to też moja funkcja

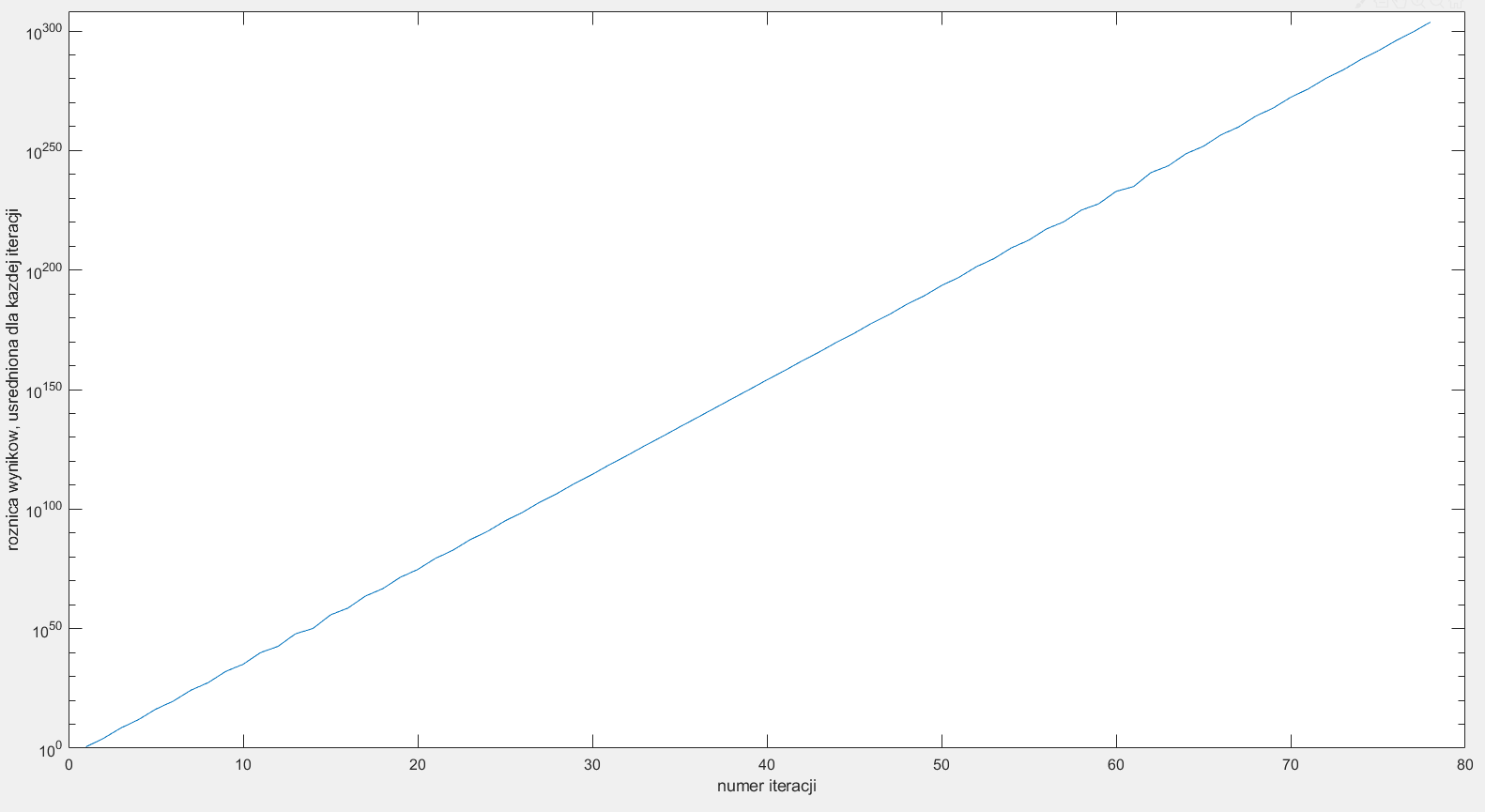
Następne dwa wykresy dotyczą zależności błędu od liczby równań w układzie dla zestawu nr 1.





Widać, że błąd zmniejszył się diametralnie, gdy zwiększyła się liczba iteracji do 250. Jest on teraz zbliżony do błędu przy zastosowaniu algorytmu eliminacji Gaussa, z poprzedniego zadania.

Dla zestawu drugiego widzimy natomiast, że różnica nie maleje, ale rośnie, bardzo szybko osiągając ogromne wartości rzędu 10300 !

Wykres świadczy o tym, że algorytm Jacobiego dla zestawu drugiego nie jest zbieżny, co potwierdza moja funkcja check(), która zwraca fałsz.

Podobnie dzieje się dla zestawu trzeciego, który również nie nadaje się do rozwiązania metodą iteracyjną.

