ŁUKASZ STANISZEWSKI, NR INDEKSU: 304098

METODY NUMERYCZNE

PROJEKT 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 () wyniku działań elementarnych była nie większa od dokładności maszynowej komputera (), co można zapisać (wzór 1.5):

.

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

,

gdzie oznacza zbiór liczb maszynowych oraz .

Przy założeniu, że podstawą w reprezentacji zmiennoprzecinkowej liczb jest 2, zaczynając od , w pętli dzieliłem liczbę przez do momentu, aż kolejny taki iloraz zsumowany z liczbą przestanie dawać liczbę większą od , 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 mojego komputera wynosi , co jest zgodne ze standardem **IEEE754** dla liczb w podwójnej precyzji (gdzie mantysa zapisywana jest właśnie na **52 bitach**).

# Zadanie 2 – Eliminacja Gaussa z Częściowym Wyborem Elementu Podstawowego

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

1. Eliminacji zmiennych wykonywanym w krokach, gdzie w wyniku odpowiednich przekształceń zarówno na macierzy jak i wektorze , otrzymamy równoważny układ równań z macierzą trójkątną górną w miejscu macierzy .
2. Zastosowania algorytmu rozwiązania 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 konieczność wybrania ze zbioru elementów macierzy z kolumny z wierszy o indeksie niemniejszym niż , element o największym module (wzór 2.28):

i zamianę w macierzy rozszerzonej miejscami wiersza wraz z wierszem .

Działanie te nie dość, że prowadzi do mniejszych błędów numerycznych, to również eliminuje sytuację, gdy w -tym kroku . Tak więc realizację całego algorytmu można opisać w punktach:

1. Połączenie macierzy A oraz wektora b w macierz rozszerzoną :
2. W krokach wykonanie etapu eliminacji zmiennych, gdzie **-ty** krok tego działania to zamiana **-tego** wiersza macierzy rozszerzonej z wierszem , w którym znajduje się wcześniej opisany element o największym module (wybierany częściowo), a następnie wykonanie działania:

, gdzie oraz to **-ty** wiersz macierzy rozszerzonej.

1. W wyniku powyższego etapu, po krokach, uzyskany układ równań to , gdzie to macierz trójkątna górna.
2. Na końcu, w celu zdobycia wektora , koniecznym jest rozwiązanie układu , stosując klasyczny algorytm rozwiązania układu równań z macierzą trójkątną polegający na przejściu od wiersza ostatniego w górę i wykonaniu działania (wzór 2.17):

dla

*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 rozwiazanie ukladu rownan liniowych Ax=b.

n = size(A, 1); % liczba rownan

Ab = [A b]; % dla ulatwienia laczymy poziomo A z b (macierz rozszerzona)

for k=1:n % k to krok eliminacji gaussa

% wybor elementu glownego

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 pomnozone 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 przeksztalcony wektor

% prawych stron b i rozwiazujemy rownanie od dolu

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 z gęstą macierzą i wektora 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 dla .

W pierwszym kroku, dla macierzy oraz wektora z podpunktu (a) obliczyłem błąd rozwiązania dla każdego przy użyciu skryptu:

n\_of\_equations = [5, 10, 50, 100, 200]; % mozliwe n

errors\_1 = zeros(size(n\_of\_equations,2), 1); % wektor bledow

% wyznaczanie bledu dla kazdego 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

% wyswietlenie wektora bledow

disp(errors\_1);

% zobrazowanie zaleznosci w postaci wykresu

plot(n\_of\_equations, errors\_1);

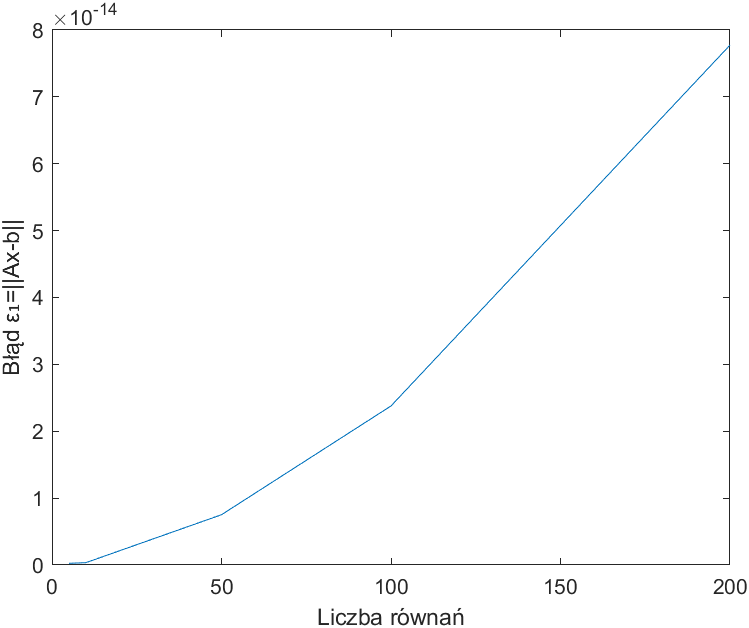
xlabel("Liczba równań");

ylabel("Błąd ε₁=||Ax-b||");

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

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

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

oraz rysunek zależności do liczby równań :

**Wykres 1 – zależność** **błędu rozwiązania do liczby układów równań 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 oraz wektora 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)

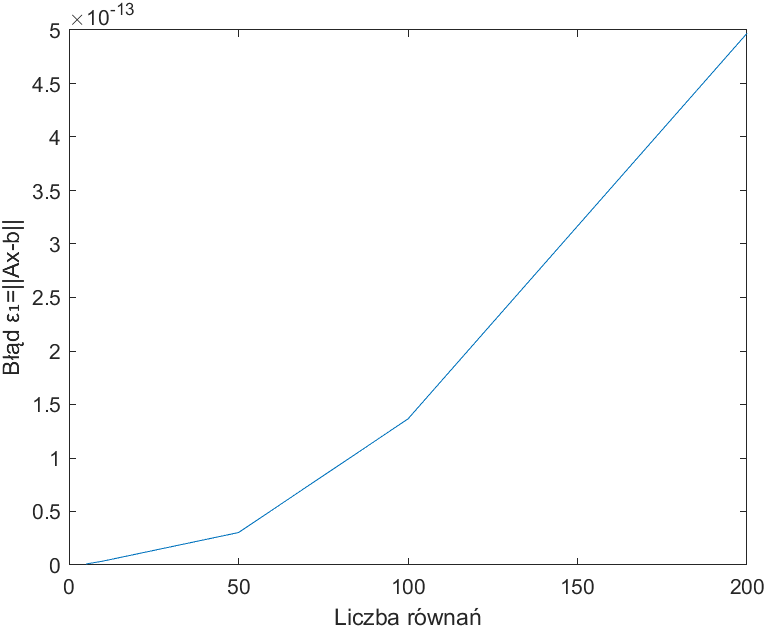
x = solveGaussPartial(A, b);

…

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

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

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

a także rysunek zależności do liczby równań :

**Wykres 2 – zależność** **błędu rozwiązania do liczby układów równań 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 , co oznacza, że metoda ta z *całkiem* dużą dokładnością daje poprawne rozwiązanie.

# 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 na sumę macierzy poddiagonalnej , macierzy diagonalnej oraz macierzy naddiagonalnej **:** .
2. Dopóki nie będzie spełniony warunek stopu (w tym przypadku dopóki norma euklidesowa różnicy między kolejnymi przybliżeniami rozwiązania nie będzie mniejsza niż podane ), wykonywaniu przybliżenia wektora (wzór 2.59):

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

1. Macierz diagonalna , na którą m.in. dekomponuje się macierz , jest macierzą nieosobliwą (wartości na diagonali są niezerowe).
2. Aby metoda Jacobiego była zbieżna, macierz musi charakteryzować się silną diagonalną dominacją wierszową lub kolumnową, tzn. musi spełniać jeden z poniższych warunków:
   1. dla każdego **– dominacja wierszowa,**
   2. dla każdego **– 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 z punktu (a):

>> b = checkDominance(A)

b =

logical

1

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

>> b = checkDominance(A)

b =

logical

0

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

Funkcja, rozwiązująca układ równań liniowych metodą Jacobiego z podanym błędem granicznym liczonym jako norma euklidesowa różnicy kolejnych przybliżeń 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;

% dopoki nie jest przekroczony błąd graniczny

while lim\_error >= e2

x\_before = x;

% tworzymy wektor x dla kolejnego kroku przyblizenia

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 roznica w postaci normy z różnicy kolejnych przyblizen

lim\_error = norm(x - x\_before, 2);

end

end

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

1. Dla błędu granicznego :

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

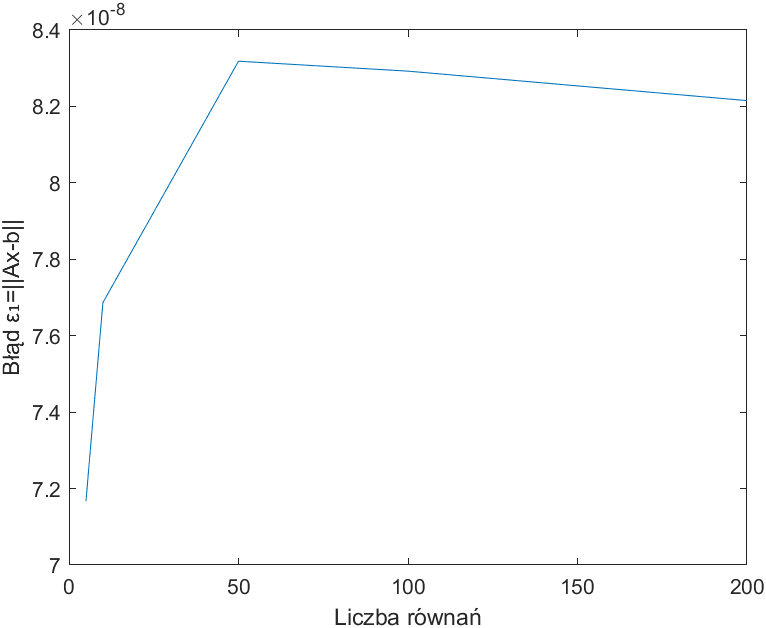


Tabela 3 i Wykres 3 – reprezentują zależnośc błędu rozwiąania do liczby równań liniowych do rozwiązania układu równań liniowych (a) dla metody Jacobiego dla błędu granicznego **.**

1. Dla błędu granicznego :

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

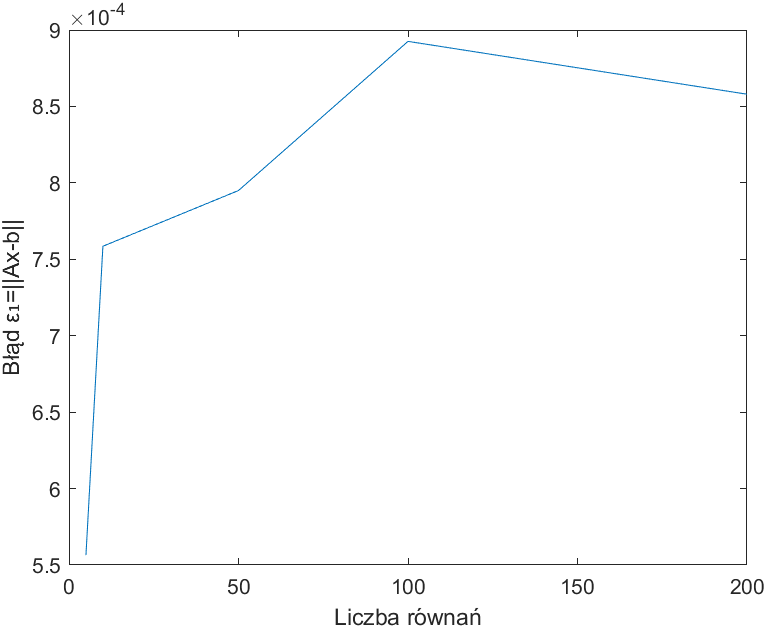


Tabela 4 i Wykres 4 – reprezentują zależność błędu rozwiąania do liczby równań liniowych do rozwiązania układu równań liniowych (a) dla metody Jacobiego i błędu granicznego **.**

1. Dla błędu granicznego :

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

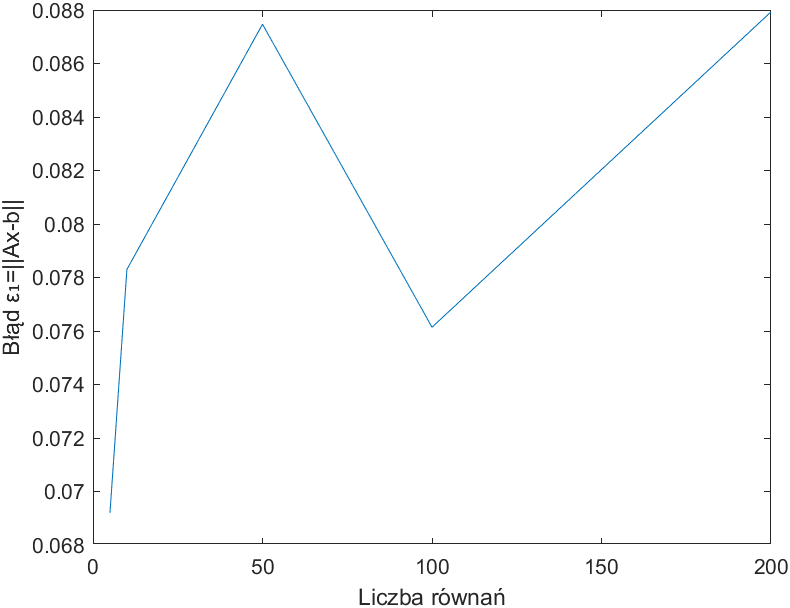


Tabela 5 i Wykres 5 – reprezentują zależność błędu rozwiąania do liczby równań liniowych do rozwiązania układu równań liniowych (a) dla metody Jacobiego i błędu granicznego **.**

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