Maciej Suchocki I.SID 277377

MNUM PROJEKT 1 - zestaw 1.12

Zadanie 1

Zadanie polega na wyznaczeniu dokładności maszynowej komputera, co przekłada się na obliczenie tak zwanego epsilona maszynowego. Według definicji epsilon maszynowy jest to wartość określająca precyzję obliczeń numerycznych wykonywanych na liczbach zmiennoprzecinkowych, będąc jednocześnie najmniejszą liczbą dodatnią ***g***, dla której zachodzi relacja *fl(1+g) > 1, tzn.*

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

Algorytm wyznaczenia zakłada początkową dokładność liczby *g = 1*. Przy ciągłym dzieleniu tej liczby przez 2 za każdym razem sprawdzamy warunek *(1+g>1)*. Jeśli warunek ten nie będzie spełniony to liczba wyznaczona w poprzednim obiegu pętli jest naszym epsilonem maszynowym.

Kod programu:

```
function [] = Zadaniel()
    x = 1.5; %początkowe zainicjalizowanie zmiennej
    g = 1.0;
    while( x > 1 ) %przechodzenie przez pętle i dzielenie epsilona
        g = g/2;   %tak długo aż dodanie go nie wpłynie na wynik
        x = 1.0 + g;
    end
    g = g*2; %jeden przebieg pętli w tył
    fprintf('Wyznaczony epsilon maszynowy: %d \n',g);
end
```

Wyznaczony epsilon maszynowy: 2.220446e-16

Wynik ten jest zgodny ze standardem IEEE 754 dla liczb zmiennoprzecinkowych podwójnej precyzji.

Zadanie 2

Zadanie polega na napisaniu procedury rozwiązującej układ ***n*** równań liniowych $Ax = b$ wykorzystując **metodę faktoryzacji Cholesky'ego-Banachiewicza**. Metoda mówi o tym, że dla każdej symetrycznej dodatnio określonej macierzy ***A*** istnieje dokładnie jedna trójkątna dolna macierz ***L*** o dodatnich elementach diagonalnych taka, że:

$$\boldsymbol{A = LL^T}$$

Rozkład ten zapisuje się w następujący sposób:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ . & . & . & . \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & \cdots & 0 \\ l_{21} & l_{22} & \cdots & 0 \\ . & . & . & . \\ l_{n1} & l_{n2} & \cdots & l_{nn} \end{bmatrix} \begin{bmatrix} l_{11} & l_{21} & \cdots & l_{n1} \\ 0 & l_{22} & \cdots & l_{n2} \\ . & . & . & . \\ 0 & 0 & \cdots & l_{nn} \end{bmatrix}$$

Celem takiego rozkładu jest zastąpienie jednego układu równań o ***n*** niewiadomych opisanego macierzą pełną, dwoma układami równań o ***n*** niewiadomych, gdzie każdy z tych układów równań

opisane są macierzami trójkątnymi. Elementy macierzy wyznaczę dzięki zależności $\boldsymbol{A = L \cdot L^T}$, która jest źródłem niezależnych równań prowadzących do wzorów na poszczególne elementy macierzy $\boldsymbol{L}$:

$$l_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} l_{ik}^2}$$

$$l_{ji} = \left(a_{ji} - \sum_{k=1}^{i-1} l_{jk} \cdot l_{ik} \right) / l_{ii}, \quad i = 1, 2, ..., n, \quad j = i+1, i+2, ..., n$$

Powyższy krok został osiągnięty następującą implementacją:

```
function[Ll] = cholesky(A,n)
    Ll = zeros(n,n);
    for i = 1:n                                          %przechodzenie po kolumnach
        for j = i:n                                      %przechodzenie wierszy wewnątrz kolumny
            if(i == j)
                Ll(i,i) = sqrt(A(i,i)-sumDiag(Ll,i));    %przypisanie wartości na przekątnej macierzy
            else
                Ll(j,i) = (A(j,i)-sumRest(Ll,i,j))/Ll(i,i);   %przypisanie wartości w reszcie przypadków
            end
        end
    end
end
```

```
function[sum] = sumDiag(Ll,i)
    sum=0;
    for k = 1:1:i-1
       sum=sum+Ll(i,k)^2;
    end
end
```

```
function[sum] = sumRest(Ll,i,j)
    sum=0;
    for k = 1:1:i-1
        sum=sum+(Ll(j,k)*Ll(i,k));
    end
end
```

Metoda ta zakłada rozwiązywanie równań w kolejno podany sposób:

$$\boldsymbol{A = L \cdot L^T}$$
$$\boldsymbol{A \cdot X = L \cdot L^T \cdot x = b}$$
$$\boldsymbol{L^T \cdot x = y}$$
$$\boldsymbol{L \cdot y = b}$$

Mając macierz $\boldsymbol{L}$ oraz jej transpozycję możemy przejść do rozwiązywania układów równań dążących do wyznaczenia wektora $\boldsymbol{y}$, a w konsekwencji wektora wynikowego $\boldsymbol{x}$. Aby rozwiązać układ równań złożony z macierzy trójkątnej w zależności od jej rodzaju posłużymy się **algorytmem Forward Substitution** - z macierzą dolnotrójkątną ($\boldsymbol{L}$) lub **algorytmem Backward Substitution** - z macierzą górnotrójkątną ($\boldsymbol{L^T}$) w zależności od równania. Algorytmy te są symetryczne wobec siebie i polegają

na sukcesywnym wyznaczania wyników równań z coraz większą liczbą niewiadomych, zastępując wszystkie z nich oprócz jednego wynikami z poprzednich iteracji. Jedyną różnicą między nimi jest przechodzenie przez macierz albo od pierwszego wiersza do ostatniego albo od ostatniego wiersza do pierwszego. Zapis matematyczny tych algorytmów można przedstawić w następujący sposób:

Forward Substitution:

$$\begin{bmatrix} a_{11} & 0 & \dots & 0 \\ a_{21} & a_{22} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mm} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

$$x_1 = b_1/a_{11}$$
$$x_2 = (b_2 - a_{21}x_1)/a_{22}$$
$$x_3 = (b_3 - a_{31}x_1 - a_{32}x_2)/a_{33}$$
$$\vdots$$
$$x_m = (b_m - a_{m1}x_1 - a_{m2}x_2 - \dots - a_{m,m-1}x_{m-1})/a_{mm}$$

Backward Substitution:

$$\begin{bmatrix} a_{11} & \dots & a_{1,m-1} & a_{1m} \\ \vdots & \ddots & \vdots & \vdots \\ 0 & \dots & a_{m-1,m-1} & a_{m-1,m} \\ 0 & \dots & 0 & a_{mm} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_{m-1} \\ x_m \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ b_{m-1} \\ b_m \end{bmatrix}$$

$$x_m = b_m/a_{mm}$$
$$x_{m-1} = \left(b_{m-1} - a_{m-1,m}x_m\right)/a_{m-1,m-1}$$
$$x_{m-2} = \left(b_{m-2} - a_{m-2,m-1}x_{m-1} - a_{m-2,m}x_m\right)/a_{m-2,m-2}$$
$$\vdots$$
$$x_1 = (b_1 - a_{12}x_2 - a_{13}x_3 - \dots - a_{1,m}x_m)/a_{11}$$

Powyższe kroki rozwiązywania równań zostały osiągnięty następującą implementacją:

```
function [x] = solveEq(L1,b)
    b = b';                                %transponowanie macierzy
    y = forwardSubstitution(L1,b);
    x = backwardSubstitution(L1',y);
end
```

```
function [x] = forwardSubstitution(A,b)
    m = length(b);                                  %wyłuskanie ostatniego indeksu
    x(1,1) = b(1)/A(1,1);                           %wyliczenie pierwszego elementu
    for i = 2:m                                     %iteracja po wszystkich równaniach począwszy od drugiego wiersza
        x(i,1)=(b(i)-A(i,1:i-1)*x(1:i-1,1))./A(i,i);    %wyznaczanie niewiadomych
    end
end
```

```
function [x] = backwardSubstitution(A,b)
    m = length(b);                                  %wyłuskanie ostatniego indeksu
    x(m,1) = b(m)/A(m,m);                           %wyliczenie ostatniego elementu
    for i = m-1:-1:1                                %iteracja po wszystkich równaniach począwszy od przedostatniego wiersza
        x(i,1)=(b(i)-A(i,i+1:m)*x(i+1:m,1))./A(i,i);    %wyznaczenie niewiadomych
    end
end
```

Powyższe metody należy zastosować do rozwiązania poniższych układów równań dla rosnącej liczby równań n = 10,20,40,80,160,... . Dla każdego z rozwiązań należy również obliczyć błąd rozwiązania (liczony jako norma residuum) i dla każdego układu równań należy wykonać rysunek zależności tego błędu od liczby równań n.
Dane:

(a) $a_{ij} = \begin{cases} 10 & dla\ i = j \\ 4 & dla\ i = j-1\ lub\ i = j+1, \\ 0 & dla\ pozostałych \end{cases} \qquad b_i = 2,5 - 0,5i$

(b) $a_{ij} = 2(i+j) + 1;\ a_{ii} = 4n^2 - i;\ b_i = 2,5\ +\ 0,6i$

(c) $a_{ij} = \frac{1}{4(i+j+1)}\ ;\ a_{ii} = 0,2n + 0,3i;\ b_i = \frac{5}{3i}$

Do wygenerowania danych macierzy posłużyłem się poniższą implementacją:

(a)

```
function [A] = a_genA(n)
    v1 = ones(1,n)*10;
    v2 = ones(1,n-1)*4;
    A = diag(v1) + diag(v2,1) + diag(v2,-1);
end
```

```
function [b] = a_genB(n)
    bottom = 2.5 - 0.5*n;
    b = 2:-0.5:bottom;
end
```

(b)

```
function [A] = b_genA(n)
    vl = 4*n^2-1:-1:1:4*n^2-n;
    A = diag(vl);
        for i = 1:n-1
            j=i+1;
            while(j <= n)
                A(i,j) = 2*(i+j)+1; %dodanie wartości do odpowiedniego indeksu macierzy
                j=j+1;
            end
        end
    A = triu(A)+triu(A,1)'; %odbicie symetrzyczne wobec przekątnej
end
```

```
function [b] = b_genB(n)
    top = 2.5 + 0.6*n;
    b = 3.1:0.6:top;
end
```

(c)

```
function [A] = c_genA(n)
    vl = 0.2*n+0.3:0.3:0.2*n+0.3*n;
    A = diag(vl);
        for i = 1:n-1
            j=i+1;
            while(j <= n)
                A(i,j) = 1/(4*(i+j+1)); %dodanie wartości do odpowiedniego indeksu macierzy
                j=j+1;
            end
        end
    A = triu(A)+triu(A,1)'; %odbicie symetryczne wobec przekątnej
end
```

```
function [b] = c_genB(n)
    b = 1:1:n;
    b = (1./b).*(5/3); %odwrócenie i przemnożenie każdego elementu
end
```

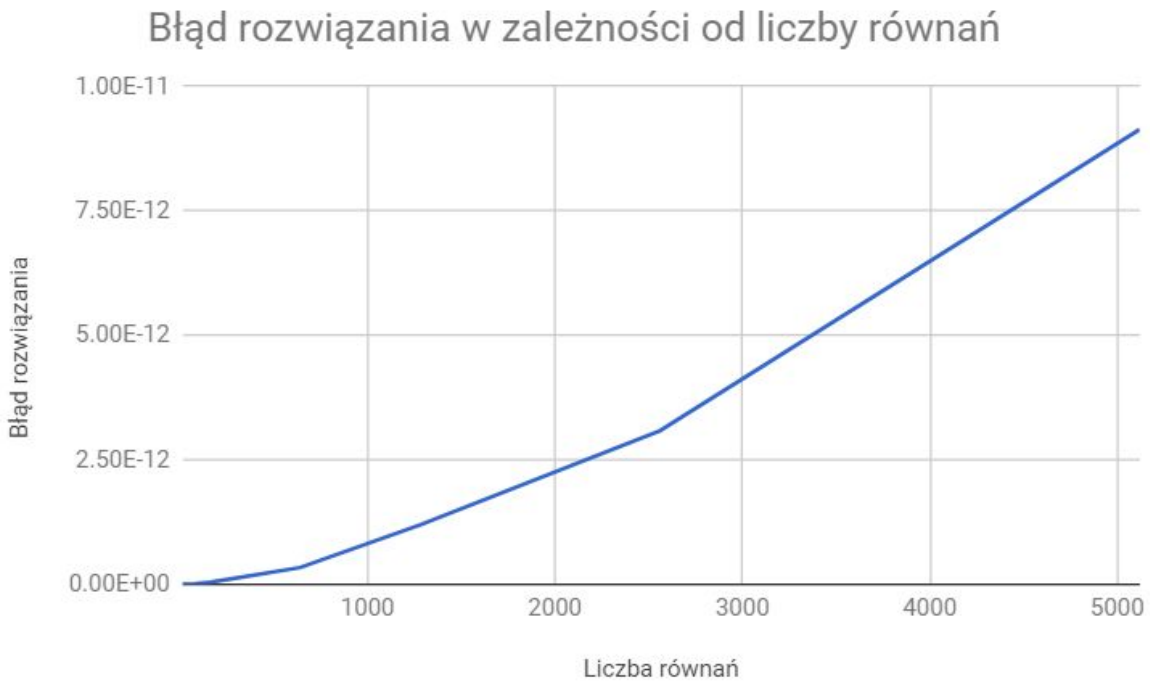
Implementacja głównego ciała zadania (dla każdego podpunktu wygląda ono identycznie - jedyną różnicą jest wywołanie innych metod generujących macierze danych):

```
function [] = Zadanie2a(n)
    A = a_genA(n);          %generator macierzy wejściowej A
    b = a_genB(n);          %generator macierzy wejściowej b
    b=b';

    tic                     %rozpoczęcie pomiaru czasu
    Ll = cholesky(A,n);     %rozkład metodą Cholesky'ego-Banachiewicza
    x = solveEq(Ll,b);      %rozwiązanie równania
    t = toc;                %koniec pomiaru czasu

    r = b - A * x;          %obliczenie residuum
    br = norm(r);           %obliczenie błędu rozwiązania jako normy z residuum

    fprintf("Zmierzony czas rozwiązania: %d \n",t);
    fprintf("Liczba równań i błąd rozwiązania: \n %d %d \n",n,br);
end
```

Wartości błędu rozwiązania w zależności od liczby równań i wykresy do każdego z zestawu danych:

Liczba równań	Błąd rozwiązania (a)	Błąd rozwiązania (b)	Błąd rozwiązania (c)
10	3.34E-16	1.88E-15	1.62E-16
20	1.68E-15	6.54E-15	5.22E-16
40	4.91E-15	1.69E-14	7.75E-16
80	1.30E-14	6.85E-14	5.78E-16
160	4.08E-14	2.24E-13	1.19E-15
320	1.39E-13	4.57E-13	4.71E-16
640	3.37E-13	1.96E-12	6.37E-16
1280	1.19E-12	7.71E-12	1.36E-15
2560	3.08E-12	4.62E-11	2.66E-15
5120	9.14E-12	1.86E-10	1.10E-14

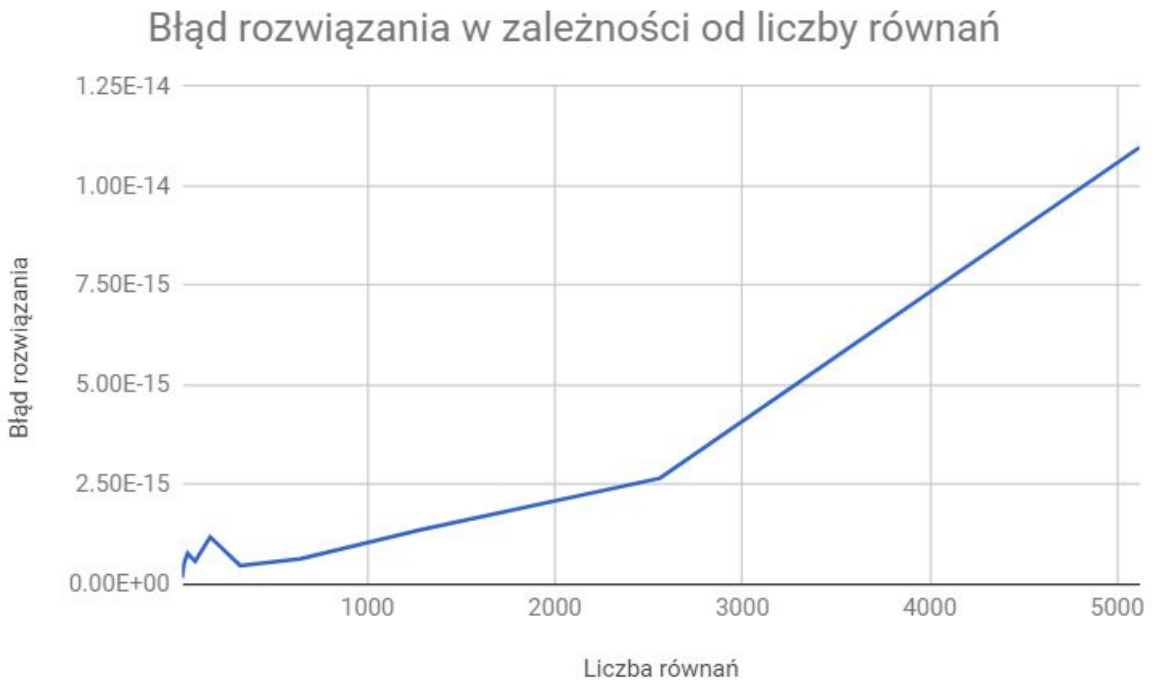
(a)


Błąd rozwiązania w zależności od liczby równań
1.00E-11
7.50E-12
5.00E-12
2.50E-12
0.00E+00
Błąd rozwiązania
1000
2000
3000
4000
5000
Liczba równań

(b)


Błąd rozwiązania w zależności od liczby równań
2.00E-10
1.50E-10
1.00E-10
5.00E-11
0.00E+00
Błąd rozwiązania
1000
2000
3000
4000
5000
Liczba równań

(c)



Omówienie wyników:
Iteracje zwiększania liczby równań kontynuowałem aż do momentu w którym czas rozwiązywania równań przekroczył 3 minuty. Taką sytuacje uzyskałem w każdym z przypadków (a,b oraz c) w momencie generowania 5120 równań. W każdym z podpunktów możemy zauważyć, że błąd rozwiązania jest niewielki i rośnie stosunkowo w zależności od liczby równań w sposób zbliżony do liniowego. Tak jak w przypadku ***(a)*** osiąga wartości rzędu e^{-12} to w przypadku ***(b)*** jest to rząd wielkości e^{-10}, a w przypadku ***(c)*** e^{-15}. Dane te wskazują, że dla każdego z tych przypadków macierz $\boldsymbol{A}$ związana z generowanymi danymi jest dobrze uwarunkowana, jest to związane z tym iż każda z nich jest diagonalnie silnie dominujący, czyli wartość bezwzględna na przekątnej macierzy są większe od sumy wartości bezwzględnych pozostałym elementów w wierszach. W przykładzie ***(a)*** wartości na głównej przekątnej są stałe, także mogliśmy się spodziewać równego wzrostu błędu do liczby równań. Podobne wyniki uzyskujemy w przykładzie (b) z powodu dużych różnic między wartościami osiągniętymi na diagonali, a resztą elementów macierzy - co prawda nie mają one wartości zerowych ale stosunek między wartościami wynikającymi z n^2, a wartościami $i+j$ jest tak wysoki szczególnie przy większej liczbie równań, że wykres z przykładu ***(b)*** mocno przypomina ten z przykładu ***(a)***, jednak błąd osiąga o 2 rzędy większe wartości. W przykładzie (c) możemy zauważyć podobne zachowanie jak w przykładzie (b) przy większej liczbie równań - wynika to z tego iż zmienne opisujące wartość poza główną przekątną zależą w sposób odwrotnie proporcjonalny do sumy indeksów macierzy, a wartość na diagonali w sposób wprost proporcjonalny. Jednakże przez taką zależność możemy zauważyć wahania w błędzie przy mniejszych ilościach równań, gdyż przez element $0.2n$ dodany do elementów głównej przekątnej w zależności od liczby równań wygenerowane macierze $\boldsymbol{A}$ są lepiej lub gorzej uwarunkowane.

Zadanie 3

Zadanie to polega na napisaniu procedury rozwiązującej układ ***n*** równań liniowych, wykorzystując metodę iteracyjną **Gaussa-Seidela** i użyć jej do rozwiązania układu równań liniowych z punktu 2(b). Parametry układu powinny zostać takie same i jedynie być uzupełnione o parametr wejściowy ε - błąd przybliżenia, liczony jako norma euklidesowa różnicy między kolejnymi przybliżeniami rozwiązania. W celu przeprowadzenia rozwiązywania równania $Ax = b$ najpierw należy sprawdzić warunek dostateczny zbieżności. Warunek ten mówi o tym iż suma wszystkich elementów w wierszy poza diagonalny macierzy $\boldsymbol{A}$ nie może być większa od elementu diagonalnego. Jeśli warunek ten jest spełniony możemy rozdzielić macierz na macierze podiagonalną ($\boldsymbol{L}$), diagonalną ($\boldsymbol{D}$) oraz naddiagonalną ($\boldsymbol{U}$):

$$\boldsymbol{A = L + D + U}$$

Warunek dostateczny zbieżności zaimplementowałem w następujący sposób:

```
function[bool] = warDost(A,n)
    for i = 1:n
        %Sumujemy wszystkie elementy w wierszu oprócz głównej przekątnej
        sum = 0;
        for j = 1:n
            if i~=j
                sum = sum + abs(A(i,j));
            end
        end
        %Sprawdzamy warunek silnej dominancji dla jednego wiersza
        if sum > abs(A(i,i))
            bool = 0;
            return;
        end
    end
    bool = 1;
end
```

A rozkład osiągnąłem poniższą implementacją:

```
function[L,D,U] = rozkladGaussSeidel(A,n)
    L = zeros(n,n); %macierz poddiagonalna
    U = zeros(n,n); %macierz naddiagonalna
    D = diag(diag(A)); %macierz diagonalna

    for i = 1:n
        for j = 1:n
            if(i<j)
                U(i,j) = A(i,j);
            elseif(i>j)
                L(i,j) = A(i,j);
            end
        end
    end
end
```

Układ równań przedstawia się teraz w następującej postaci:

$$(\boldsymbol{D}+\boldsymbol{L})\cdot\boldsymbol{x} = -\boldsymbol{U}\cdot\boldsymbol{x}+\boldsymbol{b}$$

Iterację naszego algorytmu możemy zapisać w następującej postaci:

$$(\boldsymbol{D}+\boldsymbol{L})\cdot\boldsymbol{x}^{(i+1)} = -\boldsymbol{U}\cdot\boldsymbol{x}^{(i)}+\boldsymbol{b}, \quad \boldsymbol{i = 0,1,2,...}$$

Po przekształceniu:

$$\boldsymbol{D}\cdot\boldsymbol{x}^{(i+1)} = -\boldsymbol{L}\cdot\boldsymbol{x}^{(i+1)} - \boldsymbol{U}\cdot\boldsymbol{x}^{i}+\boldsymbol{b}; \quad \boldsymbol{i = 0,1,2,...}$$

Dzięki temu otrzymujemy układ n równań skalarnych, gdzie składowe nowego wektora wyznaczamy kolejno przyjmując $\boldsymbol{w^{(i)} = U\cdot x^{(i)} - b}$:

$$x_1^{(i+1)} = -\frac{w_1^{(i)}}{d_{11}},$$

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

$$x_3^{(i+1)} = -\frac{-l_{31}\cdot x_1^{(i+1)} - l_{32}\cdot x_2^{(i+1)} - w_3^{(i)}}{d_{22}},$$

Itd.

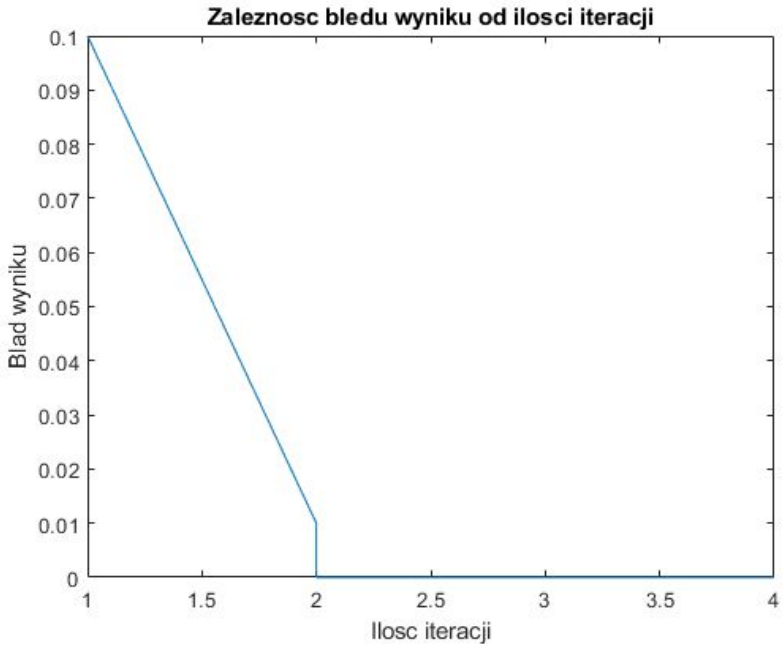
Rozwiązanie to osiągnąłem przy pomocy poniższej implementacji:

```
function[x,iter] = GaussSeidel(A,b,n,e)
    x = zeros(n,1);

    %Stworzenie macierzy naddiagonalne, poddiagonalnej i diagonalnej
    [L,D,U] = rozkladGaussSeidel(A,n);

    r = 1;
    iter = 1;
    while r>e                                   %kolejne iteracje
        y = x;                                  %zapamietujemy wektor x z poprzedniej iteracji
        w = U*x - b;
        for i = 1:n
            x(i) = (-L(i,:)*x- w(i))/D(i,i);
        end
    r = x-y;
    r = norm(r);                                %liczmy blad z nomrmy euklidesowej, po kazdej iteracji
    iter = iter + 1;                            %zliczamy ilosc iteracji
    end
end
```

Należy również nadmienić, że osiągnięcie zadanej dokładności podanej w zadaniu implikuje nam przerwanie iterowania algorytmu oraz wypisanie wyniku. Dodatkowo aby nasz ε był normą euklidesową różnicy między kolejnymi przybliżeniami rozwiązania to główne ciało zadania zdecydowałem się zaimplementować w następujący sposób:

```
function[] = Zadanie3(n,e2)
    i=1;
    e=1;
    A = b_genA(n);
    b = b_genB(n);

    %Sprawdzenie warunku dostatecznego zbieżności
    if(warDost(A,n) == 0)
        disp('Warunek silnej dominacji diagonalnej nie jest spelniony');
        return
    end

    while e>e2
        [x,iter(i)] = GaussSeidel(A,b,n,e);
        e = e/10;
        r(i) = e;
        i = i+1;
    end
    plot(iter, r);
    title('Zaleznosc bledu wyniku od ilosci iteracji')
    xlabel('Ilosc iteracji');
    ylabel('Blad wyniku');
end
```

Aby zaobserwować jaka jest zależność między błędem wyniku a liczbą operacji można przedstawić je za pomocą wykresu. Poniższe dane przedstawiają rozwiązanie dla liczby równań równej 2560.



Jak możemy zaobserwować na wykresie błąd wyniku zbiega do zera już po drugiej iteracji. Możemy z tego wywnioskować, iż dla podanych danych metoda Gaussa-Seidela daje znakomite rezultaty. Co więcej po przeprowadzeniu testów z liczbą równań równą 5120, 1280, 640, 320 wykres ten wyglądał prawie identycznie co wskazuje na to, że metoda ta również nadaje się dla tych danych i nie wpływa na nią wielkość macierzy. Należy również zauważyć, że rozwiązania tą metodą były o wiele szybsze od rozwiązań metodą **faktoryzacji Cholesky'ego-Banachiewicz** z zadania 2.