MNUM – PROJEKT, zadanie 1.9

Bartosz Cywiński

Spis treści

Zadanie 1. Dokładność maszynowa komputera

Opis zastosowanego algorytmu

Dokładność maszynową eps można określić jako najmniejszą dodatnią liczbę maszynową g taką, że zachodzi relacja $fl(1+g) > 1$, to znaczy:

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

Kod programu

```
function [eps] = machinePrecision()
% MACHINEPRECISION Funkcja wyznacza dokładność maszynową komputera

g = 1.0;
a = 1.0 + g;
% Dopóki liczba a jest uznawana przez komputer jako większa od zera
while (a > 1.0)
    eps = g; % Zapamiętuję poprzednią liczbę g
    g = g / 2.0; % Zmniejszam liczbę g
    a = 1.0 + g;
end
end
```

Prezentacja wyników

```
>> [x] = machinePrecision()

x =

   2.2204e-16

>> eps

ans =

   2.2204e-16
```

Wnioski

Uzyskany wynik jest zgodny z *IEEE Standard 754 (double precision)*. Ponadto wynik ten jest taki sam, jak wynik uzyskany po wykonaniu funkcji eps wbudowanej w Matlaba.

Zadanie 2. Rozwiązywanie układów równań za pomocą metody rozkładu LU

Opis zastosowanego algorytmu

Do zastosowania metody rozkładu LU do rozwiązania układów równań, konieczne jest rozłożenie macierzy $\mathbf{A}$ na iloczyn dwóch macierzy: dolnej trójkątnej $\boldsymbol{L}$ $(lower\ triangular)$ i górnej trójkątnej $\boldsymbol{A}^{(n)} = \boldsymbol{U}$ $(upper\ triangular)$. Do uzyskania tych macierzy prowadzi postępowanie metodą eliminacji Gaussa:

W k-tym kroku eliminujemy za pomocą równania k-tego zmienną x_k z równań $k+1, k+2, \dots, n$ – odejmując od każdego z nich, kolejno, równanie k-te pomnożone przez

$$l_{ik} \stackrel{\text{def}}{=} \frac{a_{ik}^{(k)}}{a_{kk}^{(k)}}, \qquad i = k+1, k+2, \dots, n$$

W taki sposób otrzymujemy kolejne wartości macierzy $\boldsymbol{L}$.

Natomiast przekształcenia macierzy $\boldsymbol{A}$ w k-tym kroku dane są zależnością:

$$a_{ij}^{(k+1)} = a_{ij}^{(k)} - l_{ik} a_{kj}^{(k)}, \qquad j = k, k+1, \dots, n$$

Ponadto wiedząc, że

$$\boldsymbol{U} = \boldsymbol{A}^{(n)}$$

Doprowadziliśmy do rozkładu LU macierzy.

Jeśli dysponujemy jedynie rozkładem LU macierzy $\boldsymbol{A}$ (mamy jedynie oryginalny wektor prawych stron b, to aby rozwiązać układ równań $\boldsymbol{LU}x = b$, rozwiązujemy dwa równoważne mu układy równań z macierzami trójkątnymi (dolną i górną):

$$\boldsymbol{L}y = b, a\ następnie$$
$$\boldsymbol{U}x = y$$

Powyższe układy równań rozwiązujemy według wzoru:

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

Przy czym dla macierzy $\boldsymbol{L}$ zamiast zaczynać od wiersza ostatniego i idąc wierszami „w górę”, zaczynamy od wiersza pierwszego i idziemy wierszami „w dół”.

W wyniku rozwiązania powyższych dwóch układów równań, otrzymamy wektor rozwiązań x.

Kod programu

```
function [x] = LUDecomposition(A, b, n)
%LUDECOMPOSITION Funkcja rozwiązująca układy równań metodą rozkładu LU

L = eye(n, n);
y = zeros(n,1);
x = zeros(n,1);

if det(A) == 0
    disp('Macierz A nie jest nieosobliwa');
    return
end

for k = 1:n
    for i = k+1:n
        % Wykonuję eliminację Gaussa
        l_ik = A(i,k) / A(k,k);
        L(i,k) = l_ik;
        A(i,:) = A(i,:) - (l_ik*A(k,:));
    end
end
U = A;

% Dysponujemy jedynie rozkładem LU macierzy ->
% Nie mamy przekształconego wektora b

% Rozwiązujemy równanie Ly = b
for k = 1:n
    y(k) = (b(k) - L(k,:)*y) / L(k,k);
end

% Następnie rozwiązujemy równanie Ux = y
for k = n:-1:1
    x(k) = (y(k) - U(k,:)*x) / U(k,k);
end

end
```

Prezentacja wyników

Najpierw sprawdziłem działanie programu dla zadania wymiaru $n = 5$ z gęstą macierzą $A = GG^T$, gdzie macierz G została wygenerowana poleceniem `10*rand(5)` i wektorem b wygenerowanym poleceniem `30*rand(5,1)`.

```
>> n = 5;
>> G = 10*rand(5);
>> A = G*G'

A =

  114.8258  100.1373   99.4403  163.8907  131.6661
  100.1373  201.9229  161.2894  193.2571  204.7168
   99.4403  161.2894  279.0966  262.2748  282.8958
  163.8907  193.2571  262.2748  348.6600  328.3957
  131.6661  204.7168  282.8958  328.3957  335.2644
```

```
>> b = 30*rand(5, 1)

b =

   22.7322
   22.2940
   11.7668
   19.6643
    5.1356

>> [x] = LUDecomposition(A,b,n)

x =

  -52.3508
   25.8180
   25.0521
   88.9812
 -103.4872

>> A\b

ans =

  -52.3508
   25.8180
   25.0521
   88.9812
 -103.4872

>> e1 = euclideanNorm(A*x-b)

e1 =

   3.9951e-12
```

Poprawność wyniku sprawdziłem porównując wynik otrzymany używając napisanego przeze mnie programu z wynikiem otrzymanym rozwiązując układ równań za pomocą funkcji Matlaba (A\b – rozwiązuje układ równań $Ax = b$).

Napisałem następnie dwie funkcje rozwiązujące układy równań wyznaczone według poniższych wzorów, dla ich rosnącej liczby $n = 5, 10, 50, 100, 200$ oraz liczące błąd dla każdego układu równań.

Punkt A:

$$a_{ij} = \begin{cases} 7, & dla\ i = j \\ 3, & dla\ i = j - 1\ lub\ i = j + 1 \\ 0, & dla\ pozostałych \end{cases}$$

$$b_i = 2{,}5 + 0{,}5i$$

Punkt B:

$$a_{ij} = 4(i - j) + 2$$

$$a_{ii} = \frac{1}{3}$$

$$b_i = 3{,}5 - 0{,}4i$$

```
function [errors] = AtestLU(n)
%ATESTLU Funkcja testująca działanie metody rozkładu LU na układach równań
%wygenerowanych zgodnie z pkt. A

errors = zeros(size(n));
i = 1;
for n_i = n
    [A, b] = genEquationsA(n_i);
    tic
    x = LUDecomposition(A, b, n_i);
    toc
    errors(i) = euclideanNorm(A*x - b);
    i = i + 1;
end

plot(n, errors);
title('Zależność błędu rozwiązania od liczby równań n - punkt A')
xlabel('Liczba równań (n)');
ylabel('Błąd');
end
```

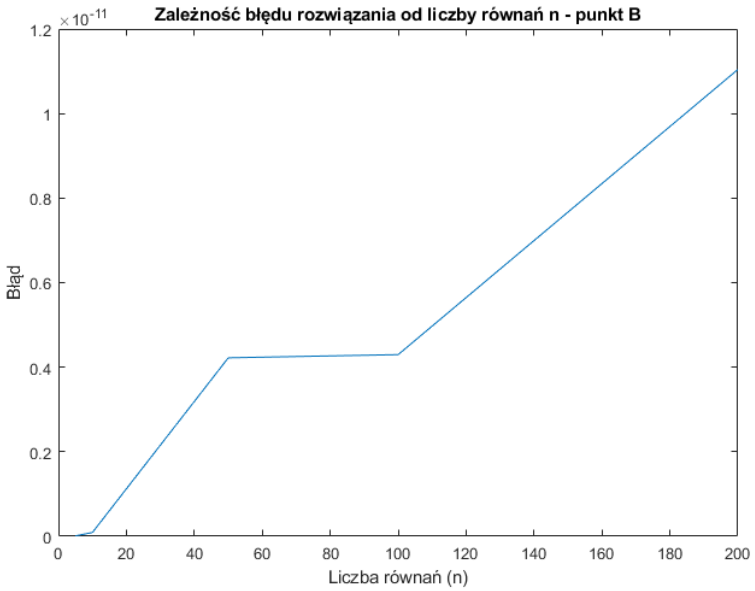
Analogicznie została napisana funkcja `BtestLU()`.

Do liczenia normy euklidesowej napisałem pomocniczą funkcję:

```
function [res] = euclideanNorm(x)
%EUCLIDEANNORM Funkcja licząca normę euklidesową

res = sqrt(sum(abs(x).^2));
end
```

Wykresy


```
Elapsed time is 0.000030 seconds.
Elapsed time is 0.000017 seconds.
Elapsed time is 0.000138 seconds.
Elapsed time is 0.000478 seconds.
Elapsed time is 0.002903 seconds.

errors =

   1.0e-13 *

    0.0126    0.0126    0.0933    0.1909    0.5901
```




```
Elapsed time is 0.000070 seconds.
Elapsed time is 0.000040 seconds.
Elapsed time is 0.000157 seconds.
Elapsed time is 0.000459 seconds.
Elapsed time is 0.003330 seconds.
```

```
errors =

   1.0e-10 *

    0.0001    0.0009    0.0422    0.0430    0.1104
```

Wnioski

Błędy rozwiązań układów równań z punktu B są około o 3 rzędy wielkości większe niż błędy rozwiązań układów równań z punktu A, natomiast w obydwu przypadkach są to nieduże błędy. W obydwu przypadkach błąd rośnie wraz ze wzrostem liczby równań, ponieważ im więcej równań tym więcej mnożeń i dodawań, a z każdą elementarną operacją arytmetyczną związany jest błąd. Ponadto wszystkie błędy ulegają propagacji i modyfikacji przy kolejnych operacjach, co prowadzi do coraz większej kumulacji błędów.

Błędy w punkcie A są mniejsze przez to, że macierz generowana w punkcie A to macierz rzadka, natomiast macierz generowana w punkcie B to macierz gęsta. Większość elementarnych operacji arytmetycznych w punkcie A nie gubi precyzji, ponieważ jest to dodawanie zera lub mnożenie przez zero.

Czas obliczeń w obydwu podpunktach był bardzo podobny.

Zadanie 3. Rozwiązywanie układów równań za pomocą metody iteracyjnej Gaussa-Seidela

Opis zastosowanego algorytmu

Metoda Gaussa-Seidela jest zbieżna, kiedy $\boldsymbol{A}$ jest silnie diagonalnie dominująca wierszowo lub kolumnowo - wartości bezwzględne elementów na głównej przekątnej są większe od sumy wartości bezwzględnych pozostałych elementów w wierszach/kolumnach. Do sprawdzenia czy macierz spełnia te założenia napisałem pomocniczą funkcję:

```
function [boolean] = isDiagonallyDominant(A, n)
%ISDIAGONALLYDOMINANT Funckja sprawdzająca, czy macierz A jest silnie
%diagonalnie dominująca wierszowo lub kolumnowo

isRowDominant = 1;
isColumnDominant = 1;

% Sprawdzam czy macierz A jest silnie diagonalnie dominująca wierszowo
for i = 1:n
    a = abs(A(i,i)); % Element na głównej przekątnej

    % Suma elementów w wierszu bez elementu na głównej przekątnej
    abs_sum = sum(abs(A(i, :))) - a;
    if a < abs_sum
        isRowDominant = 0;
        break
    end
end

% Sprawdzam czy macierz A jest silnie diagonalnie dominująca kolumnowo
for i = 1:n
    a = abs(A(i,i)); % Element na głównej przekątnej

    % Suma elementów w kolumnie bez elementu na głównej przekątnej
    abs_sum = sum(abs(A(:, i))) - a;
    if a < abs_sum
        isColumnDominant = 0;
        break
    end
end

if isRowDominant == 1 || isColumnDominant == 1
    boolean = 1;
else
    boolean = 0;
end

end
```

Na początku należy zdekomponować macierz $\boldsymbol{A}$:

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

Gdzie $\boldsymbol{L}$ jest macierzą poddiagonalną, $\boldsymbol{D}$ diagonalną, a $\boldsymbol{U}$ naddiagonalną.

Następnie w każdej iteracji zastępujemy obecny wektor nowym wektorem z wcześniejszej iteracji. Wyznaczamy:

$$w^{(i)} = \boldsymbol{U}x^{(i)} - b$$

Składowe nowego wektora wyznaczamy kolejno, według wzoru:

$$x_k^{(i+1)} = \frac{-\boldsymbol{L}_{kj}x_j^{(i+1)} - w_k^{(i)}}{\boldsymbol{D}_{kk}}, \qquad j = 1, 2, \dots, n$$

Iteracje przerywamy, kiedy norma euklidesowa różnicy między kolejnymi przybliżeniami rozwiązania jest nie większa niż błąd graniczny ε_2, który jest parametrem wejściowym programu:

$$\left\|x^{(i+1)} - x^{(i)}\right\| \leq \epsilon_2$$

Kod programu

```
function [x] = GaussSeidelMethod(A, b, n, e2)
%GAUSSSEIDELMETHOD Funkcja rozwiązująca układy równań metodą iteracyjną
%Gaussa-Seidela
L = zeros(n, n); % macierz poddiagonalna
D = zeros(n, n); % macierz diagonalna
U = zeros(n, n); % macierz naddiagonalna
x_new = zeros(n, 1);

% Sprawdzenie czy macierz A jest silnie diagonalnie dominująca, by
% określić czy metoda jest zbieżna
if isDiagonallyDominant(A, n) == 0
    disp('Macierz A nie jest silnie diagonalnie dominująca');
    return
end

% Dekompozycja macierzy A
for i = 1:n
    for j = 1:n
        if i == j
            D(i, j) = A(i, j);
        elseif i < j
            U(i, j) = A(i, j);
        else
            L(i, j) = A(i, j);
        end
    end
end

error = 1.0;
% Iterowanie jest przerywane, gdy norma euklidesowa różnicy między
% kolejnymi przybliżeniami rozwiązania jest nie większa niż błąd graniczny
while error > e2
    x_curr = x_new;
    % Kolejno wyznaczam składowe nowego wektora
    w = U*x_curr - b;
    for i = 1:n
        x_new(i) = (-L(i,:)*x_new - w(i)) / D(i, i);
    end

    error = euclideanNorm(x_new - x_curr);
end

x = x_new;
end
```

Prezentacja wyników

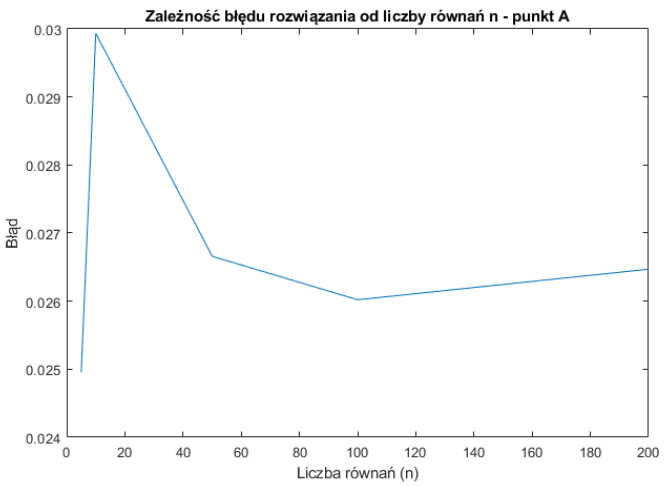
Napisałem dwie funkcje rozwiązujące układy równań wygenerowanych według wcześniej podanych wzorów.

```
function [errors] = AtestGaussSeidel(n, e2)
%ATESTGAUSSSEIDEL Funkcja testująca działanie metody Gaussa-Seidela na
%układach równań wygenerowanych zgodnie z pkt. A

errors = zeros(size(n));
i = 1;
for n_i = n
    [A, b] = genEquationsA(n_i);
    tic
    x = GaussSeidelMethod(A, b, n_i, e2);
    toc
    errors(i) = euclideanNorm(A*x - b);
    i = i + 1;
end

plot(n, errors);
title('Zależność błędu rozwiązania od liczby równań n – punkt A')
xlabel('Liczba równań (n)');
ylabel('Błąd');
end
```

Rozwiązanie układów równań z punktu B za pomocą metody Gaussa-Seidela jest niemożliwe, ponieważ macierz $\boldsymbol{A}$ nie jest silnie diagonalnie dominująca.

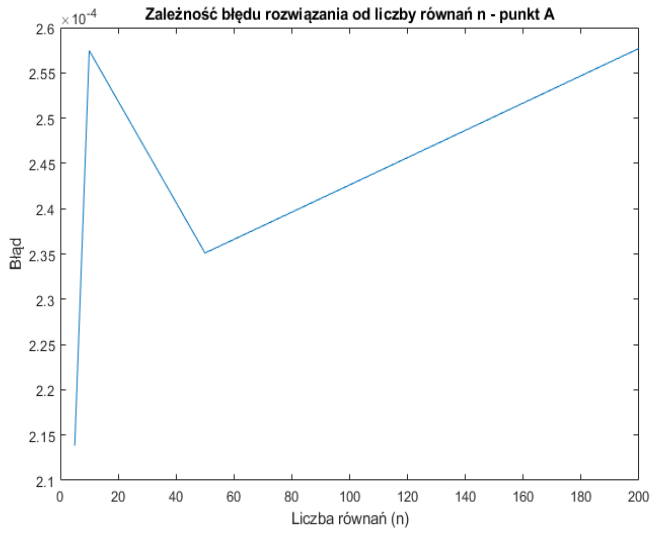
Wykresy

Błąd graniczny $\epsilon_2 = 10^{-2}$:



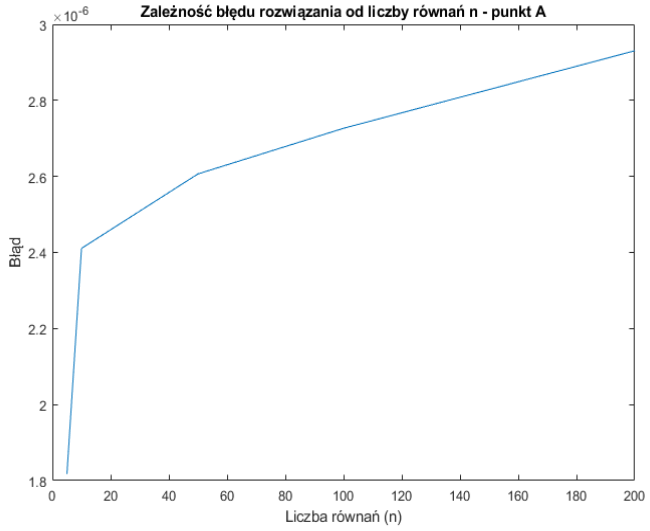

```
>> errors =
AtestGaussSeidel(n,1e-2)
Elapsed time is 0.000210 seconds.
Elapsed time is 0.000044 seconds.
Elapsed time is 0.000201 seconds.
Elapsed time is 0.000256 seconds.
Elapsed time is 0.000861 seconds.

errors =

    0.0250    0.0299    0.0267
0.0260    0.0265
```

Błąd graniczny $\epsilon_2 = 10^{-4}$:



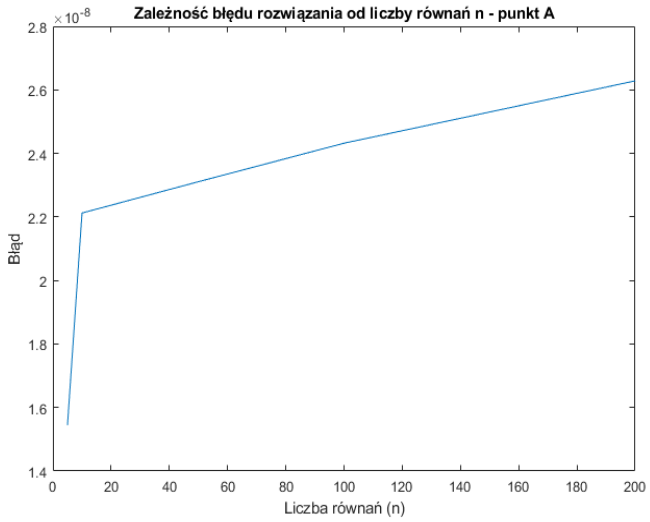

```
>> errors =
AtestGaussSeidel(n,1e-4)
Elapsed time is 0.000555 seconds.
Elapsed time is 0.000090 seconds.
Elapsed time is 0.000235 seconds.
Elapsed time is 0.000425 seconds.
Elapsed time is 0.001326 seconds.

errors =

   1.0e-03 *

    0.2138    0.2575    0.2351
0.2426    0.2577
```

Błąd graniczny $\epsilon_2 = 10^{-6}$:



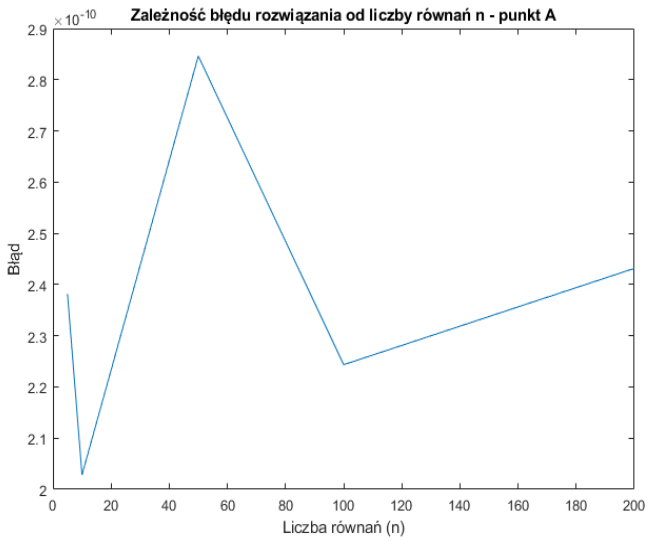

```
>> errors =
AtestGaussSeidel(n,1e-6)
Elapsed time is 0.001040 seconds.
Elapsed time is 0.000123 seconds.
Elapsed time is 0.000334 seconds.
Elapsed time is 0.000667 seconds.
Elapsed time is 0.002216 seconds.

errors =

   1.0e-05 *

    0.1817    0.2411    0.2607
0.2727    0.2930
```

Błąd graniczny $\epsilon_2 = 10^{-8}$:



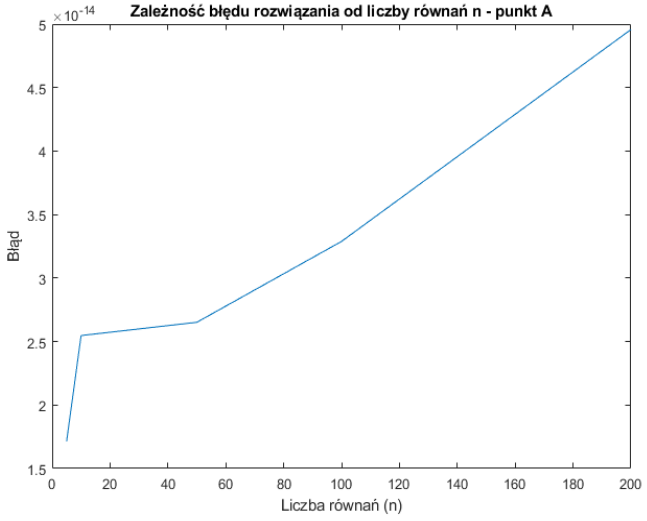

```
>> errors =
AtestGaussSeidel(n,1e-8)

Elapsed time is 0.000101 seconds.
Elapsed time is 0.000141 seconds.
Elapsed time is 0.000374 seconds.
Elapsed time is 0.000805 seconds.
Elapsed time is 0.002545 seconds.

errors =

   1.0e-07 *

    0.1544    0.2212    0.2311
0.2432    0.2628
```

Błąd graniczny $\epsilon_2 = 10^{-10}$:




```
>> errors =
AtestGaussSeidel(n,1e-10)

Elapsed time is 0.000536 seconds.
Elapsed time is 0.000137 seconds.
Elapsed time is 0.000456 seconds.
Elapsed time is 0.001041 seconds.
Elapsed time is 0.003380 seconds.

errors =

   1.0e-09 *

    0.2382    0.2028    0.2846
0.2243    0.2432
```

Błąd graniczny $\epsilon_2 = 10^{-14}$:




```
>> errors =
AtestGaussSeidel(n,1e-14)
Elapsed time is 0.000600 seconds.
Elapsed time is 0.000143 seconds.
Elapsed time is 0.000612 seconds.
Elapsed time is 0.001390 seconds.
Elapsed time is 0.004354 seconds.

errors =

   1.0e-13 *

    0.1714    0.2547    0.2652
0.3289    0.495
```

Wnioski

Patrząc na wykresy można zauważyć, że w porównaniu z metodą rozkładu LU, błąd w metodzie Gaussa-Seidela nie zawsze rośnie wraz ze wzrostem liczby równań.

Ciekawym jest, że rząd wielkości błędu jest zawsze taki sam jak rząd wielkości błędu granicznego ϵ_2. Oczywistym wnioskiem jest, że im mniejszy błąd graniczny, tym mniejszy błąd rozwiązania, gdyż wraz ze zmniejszaniem się błędu granicznego, nowy wektor rozwiązań wyznaczany jest przez większą liczbę iteracji.

Porównując czasy obliczenia układów równań metodą rozkładu LU oraz metodą Gaussa-Seidela z błędem rzędu 10^{-14} zauważyć można, że metoda Gaussa-Seidela jest liczona znacznie dłużej od metody rozkładu LU. Z drugiej strony, metoda Gaussa-Seidela jest bardziej uniwersalna, ponieważ ograniczając liczbę iteracji można znacznie ograniczyć czas potrzebny na obliczenia, uzyskując i tak już często dość dokładne rozwiązanie.