Metody Numeryczne [MNUM]
Projekt 1.40

Tomasz Pawlak, Informatyka, 304104
Prowadzący: dr hab. inż. Andrzej Karbowski

Zad 1.

„Napisać uniwersalną procedurę w Matlabie o odpowiednich parametrach wejścia i wyjścia (solwer), rozwiązującą układ n równań liniowych $\boldsymbol{A}x = \boldsymbol{b}$, wykorzystując podaną metodę. Nie sprawdzać w procedurze, czy dana macierz $\boldsymbol{A}$ spełnia wymagania stosowalności metody. Obliczyć błąd rozwiązania $\varepsilon = ||A\tilde{x} - b||_2$ (skorzystać z funkcji `norm` Matlaba). Proszę zastosować następnie swoją procedurę w programie do rozwiązania obydwu (jeśli można) lub jednego z układów równań dla podanych niżej macierzy A i wektorów b, przyjmując $n =$ 5, 10, 25, 50, 100, 200.
Metoda: **rozkład LU z pełnym wyborem elementu głównego**
Proszę wykonać wykres (wykresy) zależności błędu ε od liczby równań n."

Opis algorytmu:
Rozkład LU z pełnym wyborem elementu głównego bazuje na eliminacji Gaussa. Macierz $\boldsymbol{A}$ zostaje rozłożona na iloczyn dwóch macierzy: dolnej trójkątnej $\boldsymbol{L}$ oraz górnej trójkątnej $\boldsymbol{A}^{(n)} = \boldsymbol{U}$, gdzie n jest wymiarem macierzy $\boldsymbol{A}$. Metoda jest metodą skończoną – wynik uzyskujemy w skończonej liczbie kroków, a ilość iteracji obliczeń jest zależna od rozmiaru macierzy $\boldsymbol{A}$. Wektor $\boldsymbol{b}$ również ulega tym przekształceniom, a jego finalny wygląd zachowuje stosunek $\boldsymbol{L}\boldsymbol{b}^{(n)} = \boldsymbol{b}$. Gdy dysponujemy rozkładem $\boldsymbol{LU}$ macierzy oraz przekształconym wektorem $\boldsymbol{b}^{(n)}$ rozwiązanie możemy obliczyć rozwiązując równanie liniowe $\boldsymbol{U}\bar{x} = \boldsymbol{b}$, po czym należy pamiętać o zamianie kolejności rozwiązań: ($\overline{\boldsymbol{P}}x = \bar{x}$ **)**

Algorytm w Matlab solwera został zawarty w pliku *LU_solver.m*

Pierwszym krokiem rozwiązania jest inicjacja wartości: parametru n jako ilość wierszy macierzy $\boldsymbol{A}$. Dalej następuje inicjacja macierzy $\boldsymbol{L}$ jako $n \times n$ wymiarowa tablica zer oraz macierzy $\boldsymbol{P}$ i $\overline{\boldsymbol{P}}$, czyli macierzy zamiany wierszy i kolumn – początkowo przyjmują wartość macierzy jednostkowych wymiaru $n \times n$.

Następnym krokiem rozwiązania jest iteracyjny wybór elementu macierzy $\boldsymbol{A}$ o największym module zgodnie ze wzorem (2.35) z książki:

$$\left|a_{il}^{(k)}\right| = \max_{j,p}\left\{\left|a_{jp}^{(k)}\right|, \qquad j,p = k, k+1, \dots, n\right\}$$

Gdzie $k = 1$ dla iteracji początkowej, a dalej oznacza numer kolejnych iteracji.

Ustalone wartości wiersza i kolumny j, p zapisujemy pod postacią zmiennych $maxrow$ i $maxcol$, które potem wykorzystujemy w przekształcaniu wierszy i kolumn macierzy $\boldsymbol{A}$ $\boldsymbol{b}$ $\boldsymbol{L}$ oraz $\boldsymbol{P}$ i $\bar{\boldsymbol{P}}$.

Trzeci krok to iteracyjne wyznaczenie parametrów macierzy $\boldsymbol{L}$. W pierwotnej iteracji wyznaczamy pierwszą kolumnę macierzy oraz przekształcamy na jej podstawie wiersze macierzy $\boldsymbol{A}$ i $\boldsymbol{b}$.

Macierz $\boldsymbol{A}$ po wszystkich powyższych iteracyjnych przekształceniach jest macierzą $\boldsymbol{U}$ zgodnie z definicją algorytmu, zatem w programie należy przypisać A do zmiennej U.

Po przejściu algorytmu możemy również wyznaczyć rozwiązanie układu równań czyli wektor $\tilde{x}$, na podstawie algorytmu z linii 44-49 kodu. Pierwszy krok to inicjacja wektora rozwiązań oznaczonego jako X jako kolumna n zer. Podobnie do rozwiązania „papierowego" należy zacząć od ostatniego rzędu macierzy $\boldsymbol{U}$. Wzór zgodnie z zaleceniami jest przedstawieniem wzoru 2.17 czyli rozwiązania układu równań z macierzą trójkątną w sposób taki, by nie działać na iloczynie macierzy $\boldsymbol{U}$ oraz x, a na elementach.

Wektor rozwiązań obliczony w ten sposób zawiera poprawne wyniki lecz należy pamiętać tutaj o wspominanych wyżej zamianach kolumn rozkładu LU, które z pełnym wyborem elementu głównego. Poprawną kolejność uzyskamy operacją $x = \bar{P} \times x$ (Wymnożenie macierzy przestawień kolumn przez pierwotny wektor rozwiązań. $\bar{P}$ jest przestawioną macierzą jednostkową, zatem operacja zamienia tylko miejscami wartości x zgodnie z pierwotnym ich umiejscowieniem w układzie równań wejściowym $\boldsymbol{A}x = \boldsymbol{b}$.

Uruchomienie solwera:
Przykład uruchomienia wykonam dla macierzy z przykładu z przykładu 2.3 (str.44 z książki).

```
>> A = [3,1,6;2,1,3;1,1,1]

A =

     3     1     6
     2     1     3
     1     1     1

>> B = [2;7;4]

B =

     2
     7
     4

>> x = LU_solver(A,B)

x =

    19
    -7
    -8
```

W pliku *LU_solver.m* wynik jest obliczany zgodnie z powyższym algorytmem, a pierwszym wyjściem programu jest wektor rozwiązań. Możliwe jest wypisywanie wszystkich wyników rozkładu tj. x, $\boldsymbol{L}$, $\boldsymbol{U}$, $\boldsymbol{B}^{(n)}$, $\boldsymbol{P}$ oraz $\overline{\boldsymbol{P}}$. Wynik dla poprzednich danych:

```
>> [x,L,U,B,Prow,Pcol] = LU_solver(A,B)

x =

    19
    -7
    -8


L =

    1.0000         0         0
    0.1667    1.0000         0
    0.5000    0.6000    1.0000


U =

    6.0000    1.0000    3.0000
         0    0.8333    0.5000
         0         0    0.2000


B =

    2.0000
    3.6667
    3.8000


Prow =

     1     0     0
     0     0     1
     0     1     0


Pcol =

     0     0     1
     0     1     0
     1     0     0
```

Wyniki są jak widać identyczne jak w przykładzie książkowym.

Generacja macierzy:

Macierze z treści zadania zostaną wygenerowane z uruchomienia plików *make_array1.m* oraz *make_array2.m*:

1) $a_{ij} = \begin{cases} -17 & \text{dla } j = i \\ 3 - \frac{j}{n} & \text{dla } j = i-2 \text{ lub } j = i+2, \ b_i = 2.5 + 0.5i \\ 0 & \text{dla pozostałych} \end{cases}$

2) $a_{ij} = 4(i-j) + 2, j \neq i; \ a_{ii} = \frac{1}{3}; \ b_i = 3.5 - 0.4i$

Poniżej przykład wygenerowanych macierzy dla $n = 6$:

```
>> [A,B] = make_array1(6)

A =

  -17.0000         0    2.5000         0         0         0
         0  -17.0000         0    2.3333         0         0
    2.8333         0  -17.0000         0    2.1667         0
         0    2.6667         0  -17.0000         0    2.0000
         0         0    2.5000         0  -17.0000         0
         0         0         0    2.3333         0  -17.0000


B =

    3.0000
    3.5000
    4.0000
    4.5000
    5.0000
    5.5000
>> [A,B] = make_array2(6)

A =

    0.3333   -2.0000   -6.0000  -10.0000  -14.0000  -18.0000
    6.0000    0.3333   -2.0000   -6.0000  -10.0000  -14.0000
   10.0000    6.0000    0.3333   -2.0000   -6.0000  -10.0000
   14.0000   10.0000    6.0000    0.3333   -2.0000   -6.0000
   18.0000   14.0000   10.0000    6.0000    0.3333   -2.0000
   22.0000   18.0000   14.0000   10.0000    6.0000    0.3333


B =

    3.1000
    2.7000
    2.3000
    1.9000
    1.5000
    1.1000
```

Obie macierze nie są osobliwe (ich wyznaczniki nie są równe 0).

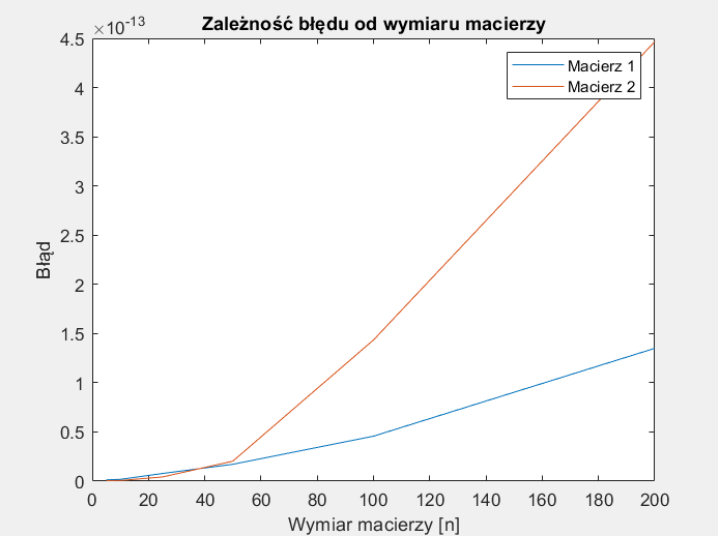
Obliczanie błędu rozwiązania ε:

Plik *zad1.m* pozwala wygenerować obliczenia równań liniowych dla obydwu macierzy oraz parametru $n = 5, 10, 25, 50, 100, 200$ oraz przedstawić je na wykresie:

1.) Wymiar macierzy dla której dokonywano obliczeń
2.) Norma euklidesowa macierzy błędów rozwiązań tj. $\varepsilon = ||\boldsymbol{A}\tilde{x} - \boldsymbol{b}||_2$, zapisana w programie jako `eps_i = norm(A*X-b, 2)`
3.) Stosunek wartości `eps_i/eps`, gdzie `eps` jest dokładnością arytmetyki wykorzystywanej przez Matlab

```
>> eps

ans =

   2.2204e-16
```

Wartości błędów wygenerowane funkcją `plot` z tego pliku są przedstawione na poniższym wykresie.



Wniosek:
Jak widać błąd dla drugiej Macierzy rośnie znacznie szybciej niżeli dla pierwszej od pewnego momentu. Nie wynika to ze złej metody obliczeniowej, ale z gorszego uwarunkowania danych w Macierzy drugiej:

```
>> [cond(make_array1(10)), cond(make_array2(10))]

ans =

    1.6933   75.1158
```

```
>> [cond(make_array1(100)), cond(make_array2(100))]

ans =

   1.0e+03 *

    0.0020    6.9881
```

```
>> [cond(make_array1(1000)), cond(make_array2(1000))]

ans =

   1.0e+05 *

    0.0000    6.9342
```

Porównując uwarunkowanie danych dla obu macierzy rzędów 10, 100 oraz 1000 widzimy że stosunek między nimi rośnie znacząco, a więc uwarunkowanie macierzy drugiej znacznie szybciej ulega pogorszeniu, co uzasadnia poprawność wykresu.

Zad 2.

„Napisać uniwersalną procedurę w Matlabie o odpowiednich parametrach wejścia i wyjścia, rozwiązującą układ n równań liniowych $\boldsymbol{A}x = \boldsymbol{b}$, wykorzystując metodę iteracyjną **Gaussa-Seidel 'a.** Nie sprawdzać w procedurze, czy dana macierz $\boldsymbol{A}$ spełnia wymagania stosowalności metody. Jej parametry wejściowe powinny zawierać m.in. wartość graniczną δ błędu między kolejnymi przybliżeniami rozwiązania, liczonego jako norma euklidesowa z ich różnicy (skorzystać z funkcji norm Matlaba). Przyjąć jako kryterium stopu warunek $\delta = 10^{-8} \triangleq 1e-8.$ Proszę zastosować tę procedurę do rozwiązania właściwego układu równań spośród przedstawionych poniżej dla $n = 5$, 10, 25, 50, 100, 200.
Proszę sprawdzić dokładność rozwiązania licząc także błąd ε i dla każdego układu równań wykonać rysunek zależności tego błędu od liczby równań n. Jeśli był rozwiązywany ten sam układ równań, co w p. 1, proszę porównać czasy obliczeń dla różnych algorytmów i wymiarów zadań."

Opis algorytmu:
Metoda Gaussa-Seidel'a jest metodą iteracyjną, różniącą się tym od metod skończonych, że liczba kroków nie jest tu odgórnie określona przez dane wejściowe, lecz zależy od tego jak dokładny wynik chcemy uzyskać (parametr stopu).

Algorytm zaczyna działanie od przybliżonego rozwiązania – danego lub zakładanego. Przybliżenie jest poprawiane z iteracji na iterację i zbiega do poprawnego rozwiązania wraz z ilością iteracji.

Warunkiem zbieżności kolejnych rozwiązań jest silna dominacja kolumnowa lub wierszowa macierzy – ta druga brzmi:

$$|a_{ii}| > \sum_{j=1, j\neq 1}^{n} |a_{ij}|, \qquad i = 1,2, \dots, n$$

Dla macierzy symetrycznych dodatkowo warunkiem dostatecznym zbieżności jest dodatnia określoność macierzy. Zgodnie z poleceniem nie jest ona sprawdzana w kodzie, lecz wynika z niej ciekawa sytuacja podczas próby rozwiązania Macierzy2, która tej normy nie spełnia.

W pliku *GS_solver.m* zawarłem funkcję zwracającą - macierz x, błąd rozwiązania liczony jako $\varepsilon = ||\boldsymbol{A}\tilde{x} - \boldsymbol{b}||_2$ (w Matlabie podstawiony pod zmienną euk) na podstawie wyniku iteracyjnych obliczeń wektora x, oraz liczbę iteracji k. Parametrami wejściowymi metody są macierze $\boldsymbol{A}$, $\boldsymbol{b}$, wektor estymowanych rozwiązań x_0 oraz próg tolerancji tol, czyli wartość graniczna błędu między rozwiązaniami obliczana jako $\delta = ||x^{(i+1)} - x^{(i)}||_2$.

Algorytm początkowo wykonuje inicjację oraz generację podziału macierzy $\boldsymbol{A}$ na trzy macierze: $\boldsymbol{L}$, $\boldsymbol{D}$ oraz $\boldsymbol{U}$ zgodnie z rozważaniami metody Jacobiego oraz Gaussa-Seidel'a omówione w podrozdziałach 2.6.1 oraz 2.6.2 książki.

Naszym zadaniem w celu poprawienia bieżącego rozwiązania będzie obliczenie układu równań:

$$\boldsymbol{D}x^{(i+1)} = -\boldsymbol{L}x^{(i+1)} - \boldsymbol{U}x^{(i)} + \boldsymbol{b}$$

Zmienne `Xj` oraz `Xi` programu są traktowane jako wektory $x^{(i+1)}$ oraz $x^{(i)}$. Inicjacja `Xj` polega na przypisaniu do niego wprowadzonej do funkcji *GS_solver* wartości `x0`, czyli pierwotnego przybliżenia. Dla tego wektora obliczamy drugą normę macierzy, a po jego porównaniu z progiem tolerancji - do iteratora *while*. Obliczamy w nim macierz w, której użyjemy do wyznaczania kolejnych wartości $x^{(i+1)}$.

Zgodnie z książką wiersze tej macierzy wynoszą:

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

Składowe nowego wektora są obliczane za pomocą metody z linii 28-33. Co jest przeniesieniem rozwiązania układu równań z macierzą trójkątną dolną opisanego w książce na str. 59 tak, aby operacje zgodnie z zaleceniem - były wykonywane na pojedynczych elementach, a nie jako iloczyny wektorowe. Wyliczenie normy euklidesowej macierzy różnicy wyników decyduje czy wynik zostanie poddany kolejnej operacji zwiększenia dokładności czy jest to już wynik, który należy uznać jako wartościowe przybliżenie wyniku.

Uruchomienie solwera:
Przykład uruchomienia solwera dla Macierzy1 6×6 wraz z kryterium stopu `1e-8` oraz wektorem losowych liczb z zakresu <0,1> :

```
>> [X, epsilon1, k] = GS_solver(A,B,rand(6,1),1e-8)

X =

   -0.2229
   -0.2537
   -0.3159
   -0.3482
   -0.3406
   -0.3713


epsilon1 =

   1.5732e-09


k =

     9
```

W przypadku próby wykonania operacji GS_solver dla macierzy drugiej wyniki nie są na tyle zbliżone do tych z pozostałych metod:

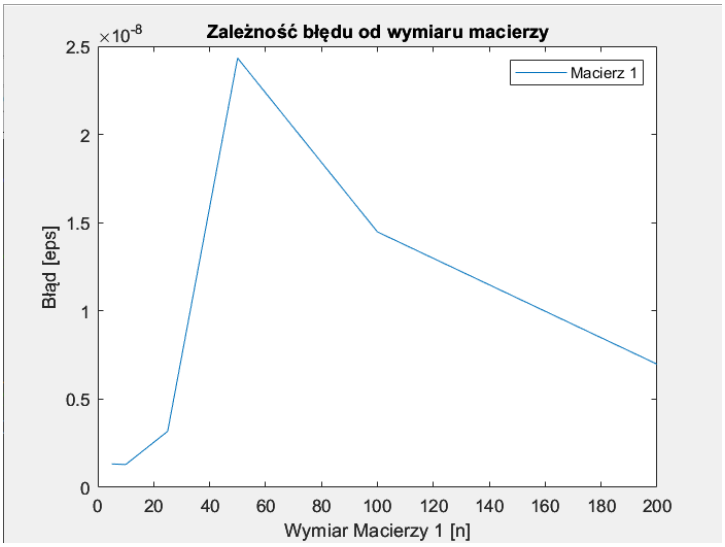
```
>> [X, epsilon1, k] = GS_solver(A,B,rand(6,1),1e-8)

X =

  1.0e+307 *

   -0.1193
    2.0540
      -Inf
       NaN
       NaN
       NaN


epsilon1 =

   NaN


k =

    41
```

Wynika to z niespełnienia kryterium dominacji kolumnowej ani wierszowej przez Macierz 2, co rzuca się w oczy przy jej wypisaniu na ekran Okna Poleceń:

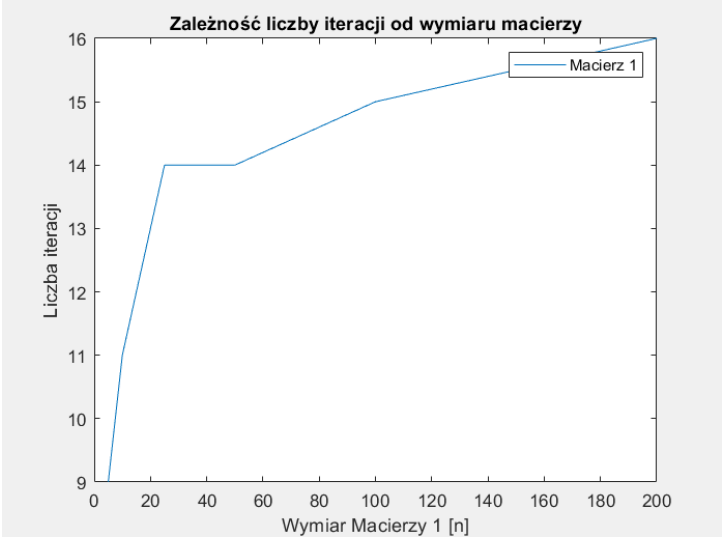
```
>> A

A =

    0.3333   -2.0000   -6.0000  -10.0000  -14.0000  -18.0000
    6.0000    0.3333   -2.0000   -6.0000  -10.0000  -14.0000
   10.0000    6.0000    0.3333   -2.0000   -6.0000  -10.0000
   14.0000   10.0000    6.0000    0.3333   -2.0000   -6.0000
   18.0000   14.0000   10.0000    6.0000    0.3333   -2.0000
   22.0000   18.0000   14.0000   10.0000    6.0000    0.3333
```

Porównanie błędu ε dla układów równań oraz czasów obliczeń algorytmów LU_Solver oraz GS_Solver oraz wnioski:

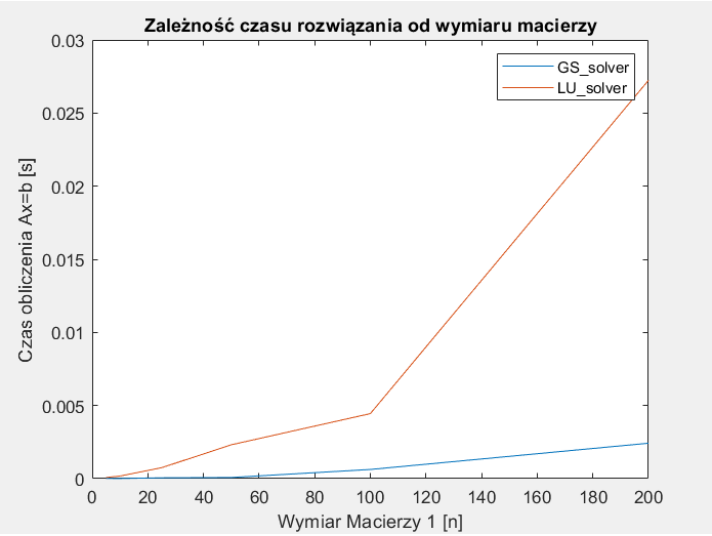
Błąd $\varepsilon = ||\boldsymbol{A}\tilde{x} - \boldsymbol{b}||_2$ liczony w *zad2.m* nie jest zależny wykładniczo jak to było w przypadku poprzedniego zadania, lecz oscyluje w okolicy wartości `1e-8` czyli tolerancji uznanej jako granica algorytmu:



Kolejny wykres wygenerowany z pliku prezentuje liczbę iteracji potrzebną przez program do osiągnięcia założonego poziomu tolerancji $1e-8$ – ciekawym zjawiskiem jest rosnącą zależność między wartościami, lecz tylko w zakresie od 9 do 16 iteracji:



Wyniki czasowe możemy porównywać sensownie tylko dla macierzy 1, a wyglądają one następująco dla procesora Intel Core i7-4700HQ @ 2,4GHz.



Możemy tu zauważyć dla metody Gaussa-Seidel'a liniowość zależności wymiar - czas, która mimo błędów 5 rzędów wielkości większych niżeli w LU daje zdecydowaną przewagę w przypadku rozwiązywania macierzy kwadratowych o znacznie większym wymiarze. Czas operacyjny solwera LU dla $n = 200$ liczący ca. 0,03s jest moim zdaniem niewielki i dla zastosowania algorytmu np. w systemach czasu rzeczywistego daje duże możliwości.

Zad 3.

„Dla podanych w tabeli danych pomiarowych (próbek) metodą najmniejszych kwadratów należy wyznaczyć funkcję wielomianową $y = f(x)$ (tzn. wektor współczynników) najlepiej aproksymującą te dane.

x_i	y_i
-10	-3.578
-8	-5.438
-6	-4.705
-4	-3.908
-2	-2.069
0	0.942
2	-0.725
4	-4.128
6	-11.160
8	-23.440
10	-42.417

Proszę przetestować wielomiany stopni: 3, 5, 7, 9, 10. Kod aproksymujący powinien być uniwersalną procedurą w Matlabie o odpowiednich parametrach wejścia i wyjścia.
W sprawozdaniu proszę przedstawić na rysunku otrzymaną funkcję na tle danych (funkcję aproksymującą proszę próbkować przynajmniej 10 razy częściej niż dane).
Do rozwiązania zadania najmniejszych kwadratów proszę wykorzystać najpierw **układ równań normalnych**, a potem **rozkład SVD**.
Do rozwiązywania układu równań i dekompozycji użyć solwerów Matlaba. Porównać efektywność obydwu podejść. Do liczenia wartości wielomianu użyć funkcji `polyval`.
Proszę obliczyć błąd aproksymacji w dwóch normach: euklidesowej oraz maksimum (nieskończoność). W obydwu przypadkach skorzystać z funkcji `norm` Matlaba."

Opis algorytmu:

LNZK polega na rozwiązaniu m równań z n niewiadomymi gdzie $m > n$. Należy znaleźć wektor taki, że:

$$\forall x \epsilon \boldsymbol{R} \quad ||b - A\hat{x}||_2 \leq ||b - Ax||_2$$

Gdzie można to zinterpretować jako minimalizacja sumy pól kwadratów odległości między wartością funkcji aproksymującej w punktach x_i oraz wartością ciągu aproksymowanego w tych punktach.

W przypadku **rozwiązania układu równań normalnych** metoda polega na rozwiązaniu układu równań:

$$A^T A\hat{x} = A^T b$$

Gdzie A jest macierzą współczynników, którą mój program oblicza w funkcji `coef_matrix`, a *b* jest wektorem wartości y danych podanych w poleceniu.

Rozwiązanie układu liniowego następuje za pomocą funkcji

```
x_normal=linsolve(A' * A,A' * y_data);
```

Zmiana została przeze mnie zastosowana ze względu na wektor Y w poleceniu – stąd też w nazwie data celem odróżnienia od wyników próbkowanych. Przyrostek normal odróżnia wyniki współczynników wielomianu od tego dla rozkładu SVD.

W przypadku słabego uwarunkowania $A^T A$ zalecane jest korzystanie z innych metod np. rozkład QR. W przypadku aproksymacji funkcją 9 oraz 10 potęgi mamy do czynienia z ostrzeżeniem wydanym przez środowisko Matlab, informującym o tym zjawisku:

```
Warning: Matrix is close to singular or badly scaled. Results may be inaccurate. RCOND =  6.300838e-19.
> In zad3 (line 23)

Warning: Matrix is close to singular or badly scaled. Results may be inaccurate. RCOND =  3.494536e-21.
> In zad3 (line 23)
```

Dla **rozkładu SVD** należy policzyć macierz *pseudoodwrotną* A^+ macierzy A zgodnie ze wzorami 3.39-3.40 z książki. Mamy tutaj możliwość skorzystania z funkcji pinv(A), a potem obliczyć wektor współczynników funkcji wielomianowej $\hat{x}$.

```
x_svd=pinv(A)*y_data
```

$$\hat{x} = A^+ b$$

Ze względu na zbytnie wykorzystanie zasobów Matlaba, obliczyłem to również według wzorów wykorzystując solwer SVD. Obydwie metody pozwalają uzyskać jednoznaczny wynik:

```
[U,E,V]=svd(A);
E=E(1:size(A,2),:);
Eplus=[inv(E) zeros(size(A,2),size(A,1)-size(A,2))];
pinvA=V*Eplus*U.';
x_svd=pinvA*y_data;
```

Jak można zauważyć zmieniam macierz $\mathbf{\Sigma}$, przycinając ją do wymiarów $n \times n$ macierzy A, co czyni ją kwadratową z diagonalnymi wartościami **σ**. Następnie zostaje ona odwrócona podczas tworzenia macierzy $\mathbf{\Sigma}^+$ oraz zgodnie z podręcznikiem uzupełniona rzędami zer do rozmiaru $n \times m$. Macierz odwrotną liczymy jako:

$$A^+ = V\Sigma^+ U^T$$

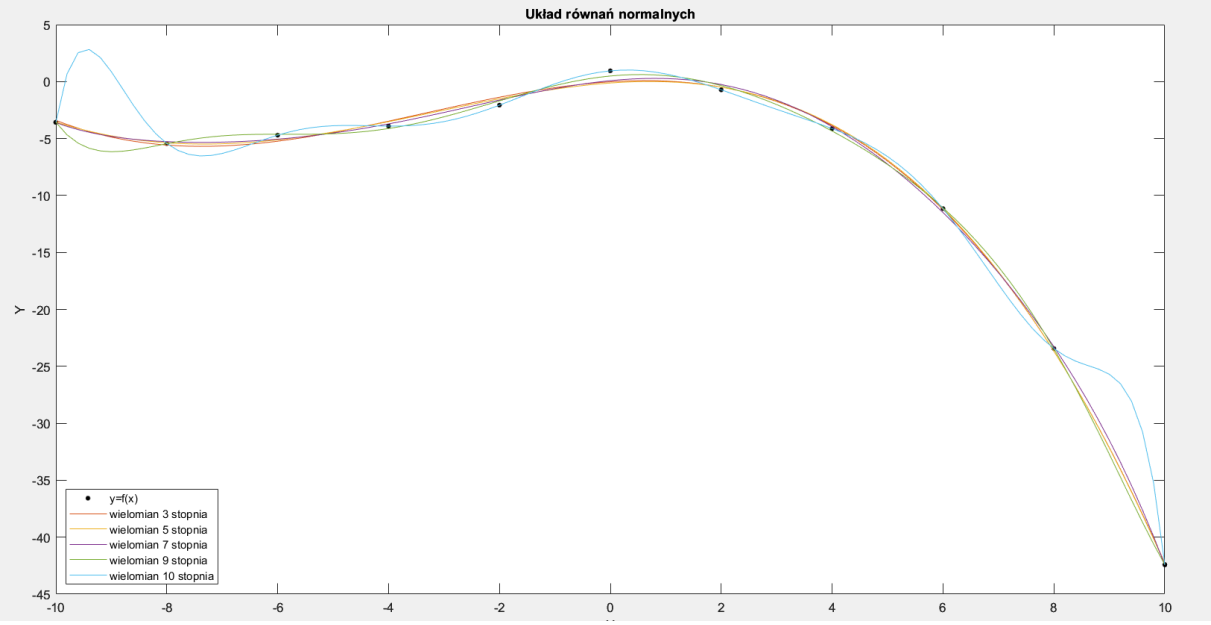
A posiadając macierz odwrotną w prosty sposób obliczamy wektor współczynników wielomianu jak wyżej.

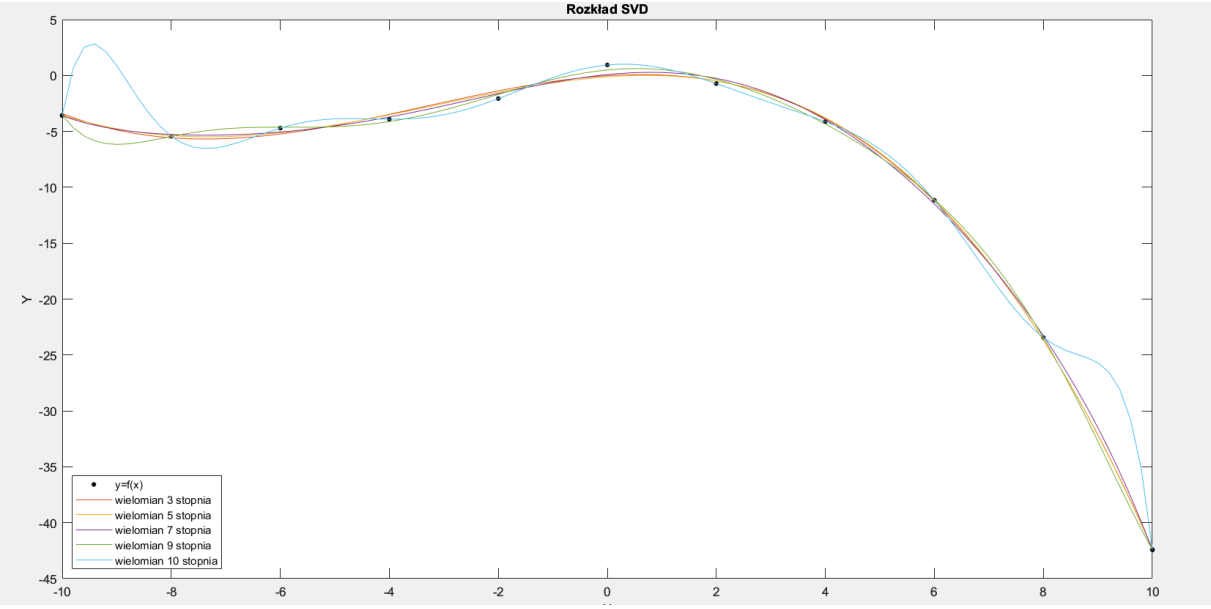
W pliku *coef_matrix.m* zawarłem generator macierzy współczynników. Jako parametr macierzy należy podać stopień wielomianu aproksymującego n oraz współrzędne wektora X, czyli rzędne punktów aproksymowanych. Dla naszych danych zwracana jest macierz o wymiarach $11 \times (stopień\_wielomianu + 1)$. Wartości tej macierzy są obliczane ze wzoru:

$$a_{ij} = X_i^{n+1-j}$$

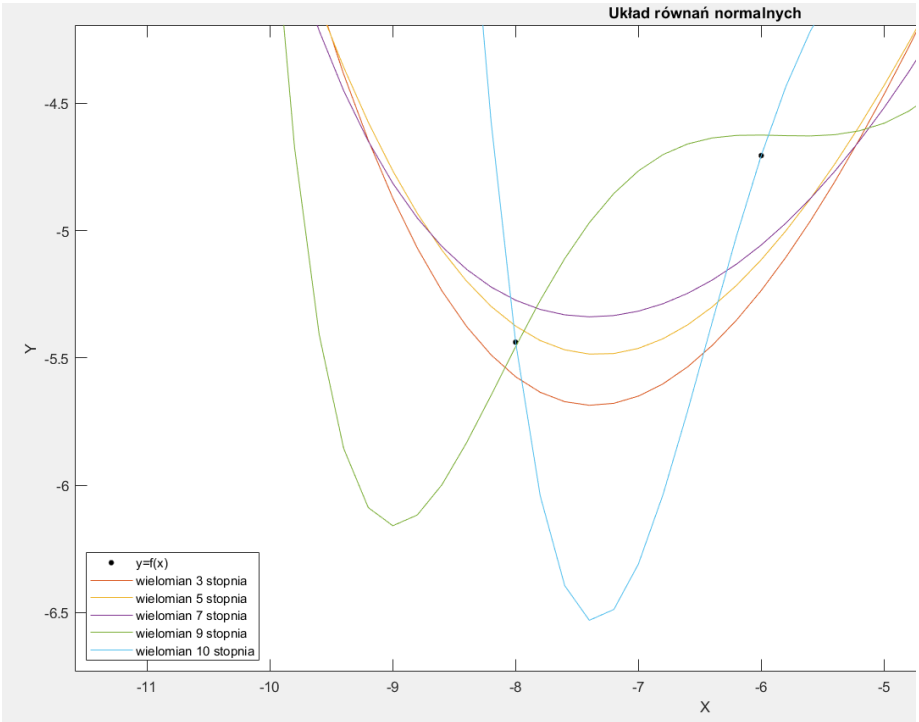
Implementacja funkcji algorytmów

Za pomocą `polyval` obliczamy wartość funkcji w punktach próbkowania (zdefiniowanych przez wektor `samples` na podstawie obliczonych wektorów. Po naniesieniu funkcji na wykres uzyskamy dwa nierozróżnialne obrazy (zapisane osobno, by można było obserwować wyniki w plikach *układ-rn-normalnych.fig* oraz *rozklad-svd-wielomian.fig*):



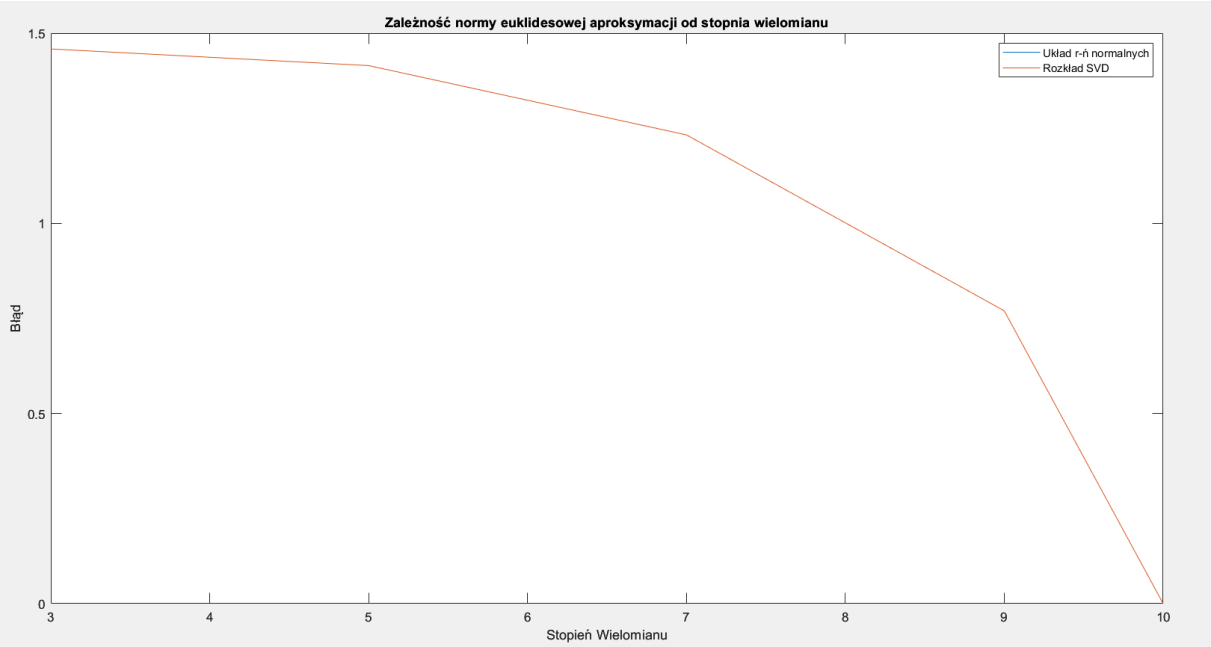


Generacja następuje po uruchomieniu polecenia zad3(). Widzimy, że aproksymacja jest bardzo dokładna na pierwszy rzut oka, a większość prostych w stosunkowo niewielkich odległościach mija punkty. Po przybliżeniu w punkcie możemy dostrzec w skali różnice pomiędzy wielomianami aproksymującymi:

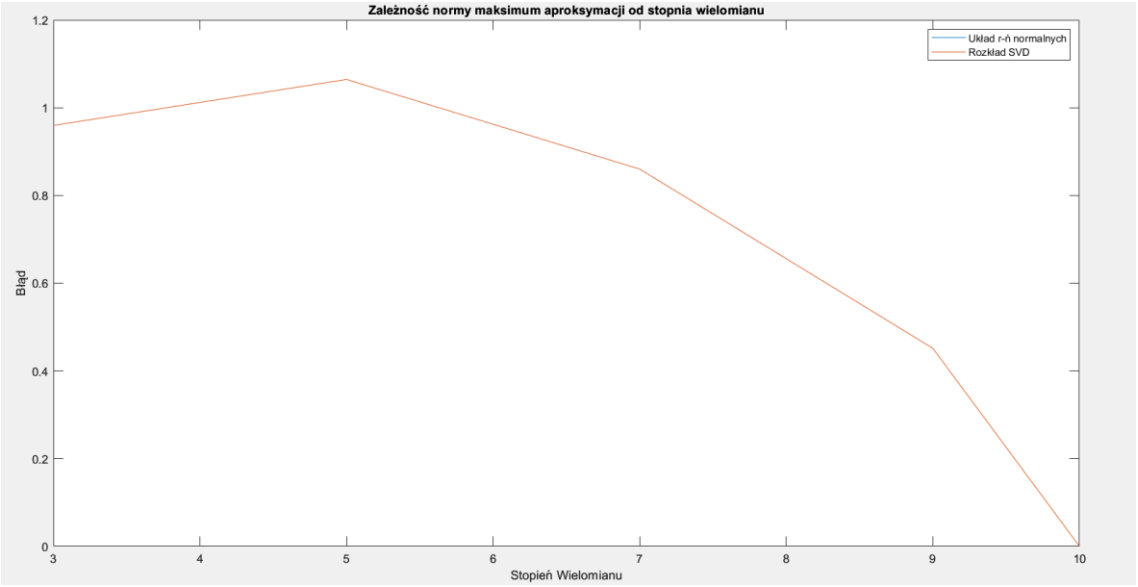


Jak widać 10 stopień wielomianu w przybliżeniu dokładnie przechodzi przez punkty. Najmniejszą dokładnością w przybliżonych dwóch punktach cechuje się wielomian 3 stopnia. Podobne wnioski można wyciągnąć oglądając wykresy norm zawarte w plikach *norma_eukl.fig* oraz *norma_maks.fig*.

Badanie normy drugiej oraz nieskończoność do szacowania błędu

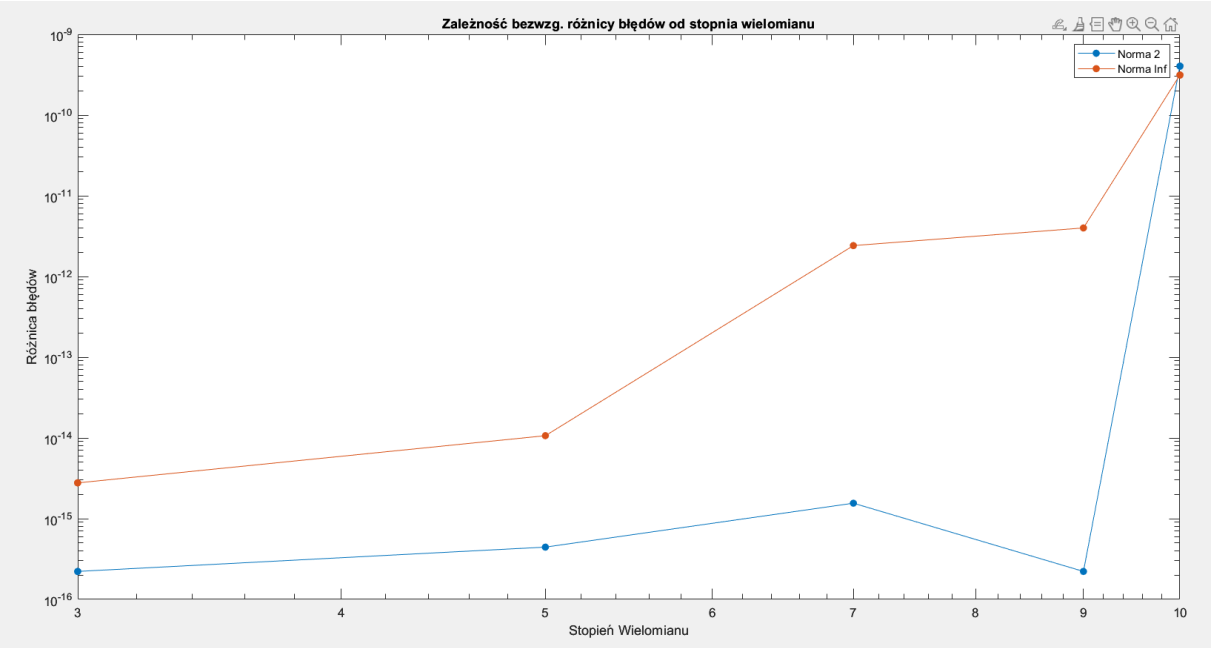


Jak widać norma euklidesowa maleje z iteracji na iterację (czyli ze stopnia na stopień wielomianu) co ma sens, biorąc pod uwagę moje odczucia i zdecydowanie lepszy wynik wielomianu 10-st. od reszty.



Wykres normy nieskończoność wykazuje delikatne pogorszenie dla 5 stopnia wielomianu względem 3 rzędu.

Wykresy dla obu algorytmów nachodzą na siebie, zatem niejasne jest czy w wykres nie wkradły się błędy lub metody wybrane przeze mnie są niesłuszne do tego zadania. Zatem dodatkowo postanowiłem wykonać wykres zależności bezwzględnej różnicy błędów między dwoma algorytmami dla każdej z obliczanej norm. Dla uproszczenia odczytu wykres ten jest rozwiązany w skali logarytmicznej oraz zawarty w pliku *roznica_bledow.fig*.



Wnioski

Błąd różnicy rośnie logarytmicznie zgodnie ze wzrostem skomplikowania obliczeniowego wynikającego z długości wektora odpowiedzi $\hat{x}$. Identycznie jak w przypadku poprzednich zadań rozmiar macierzy w dużej skali zwiększał epsilony reszty odstępstwa od danych faktycznych w realizacji maszynowej. Co ciekawe z danych warto zauważyć, że mimo rosnących obu błędów dokładność jest coraz lepsza wraz ze wzrostem stopnia wielomianu.