

Spis treści

Zadanie 1. <i>Metoda QR obliczania wartości własnych</i>	2
Zadanie 2. <i>Aproksymacja funkcji</i>	8
Załącznik 1. <i>Kod źródłowy zadania 1.</i>	15
Załącznik 2. <i>Kod źródłowy zadania 2.</i>	20

Zadanie 1. Metoda QR obliczania wartości własnych

Celem zadania jest:

1. napisanie programu obliczającego wartości własne macierzy metodą rozkładu QR:
 - a. z przesunięciami
 - b. bez przesunięć
2. przetestowanie napisanych funkcji na 30 macierzach kwadratowych o rozmiarze 5, 10 i 20
3. wyznaczenie średniej liczby iteracji potrzebnej do uzyskania wyniku o zadanej dokładności.

Koncepcja rozwiązania

1. Do wykonania rozkładu QR macierzy wykorzystana została ortogonalizacja Grama-Schmidta. Algorytm samego rozkładu można opisać następująco:
 - 1) Na podstawie macierzy wejściowej wyznacz rozmiar macierzy Q oraz R.
 - 2) Dokonaj ortogonalizacji Grama-Schmidta (algorytm opisany w poprzednim projekcie). Q - macierz ortogonalizowana, R – macierz współczynników.
 - 3) Znormalizuj otrzymane macierze Q oraz R.
 - a. Algorytm wyznaczania wartości własnych bez przesunięć:
 - i. Dopóki nie zostanie osiągnięta **tolerancja (0,00001)** lub **maksymalna liczba iteracji (200)** dopóty
 - ii. Dokonaj rozkładu macierzy wejściowej (A) na macierz Q i R.
 - b. Algorytm wyznaczania wartości własnych z przesunięciami:
 - i. wykonuj do momentu uzyskania macierzy 2x2
 - ii. wyznacz macierz dolnego prawego rogu (2x2) - M
 - iii. oblicz wartości własne macierzy M
 - iv. wybierz wartość własną bliższą elementowi prawego dolnego rogu (*przesunięcie*)
 - v. Dokonaj rozkładu QR na macierzy $A - I * \text{przesunięcie}$
 - vi. odtwórz macierz A jako $R * Q + I * \text{przesunięcie}$
 - vii. po osiągnięciu wymaganej dokładności zapisz otrzymaną wartość własną (dolny prawy róg)
 - viii. dokonaj deflacji macierzy
2. Została napisana funkcja tworząca macierze nie-/symetryczne o zadanym wymiarze i losowych wartościach. Dla ułatwienia zadania wartości losowane są liczbami całkowitymi z przedziału 0-100. Do sprawdzenia została użyta wbudowana matlabowa funkcja *qr()* wyznaczająca rozkład QR z dokładnością co do znaku oraz funkcja *eig()* wyznaczająca wektor wartości własnych.
3. Prócz wyznaczenia średniej liczby iteracji potrzebnej do uzyskania wyniku o zadanej dokładności sprawdzona została z ciekawości potencjalna korelacja ilości potrzebnych iteracji do uwarunkowania macierzy.

Sprawdzenie

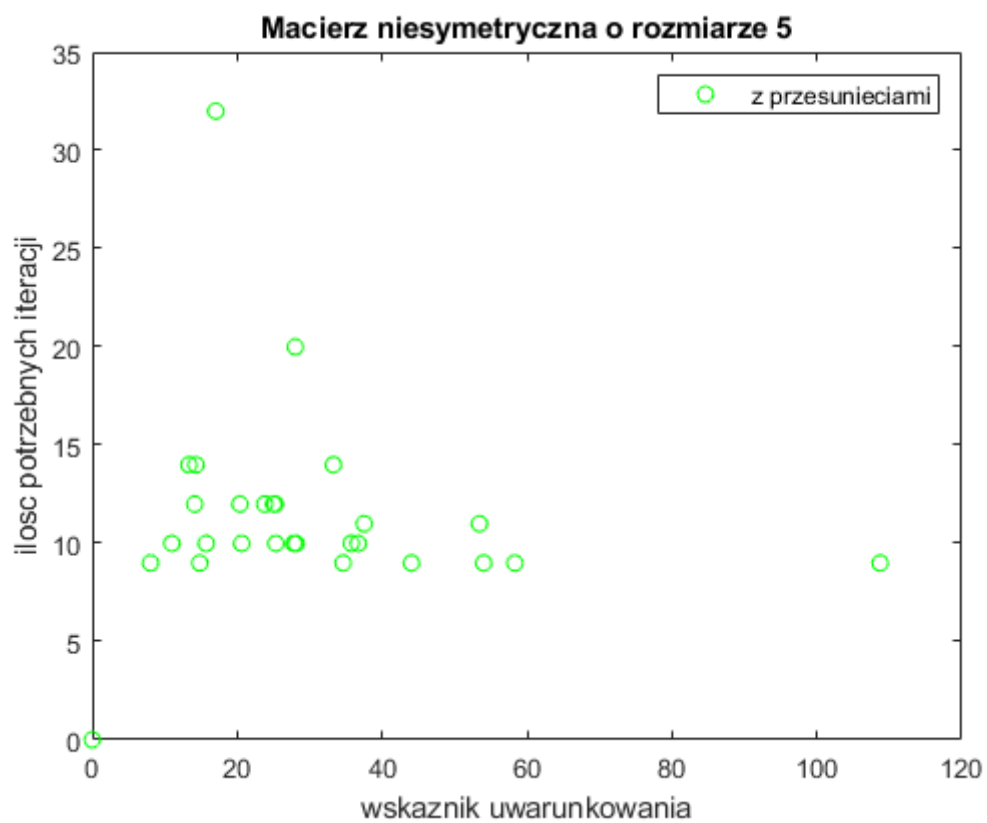
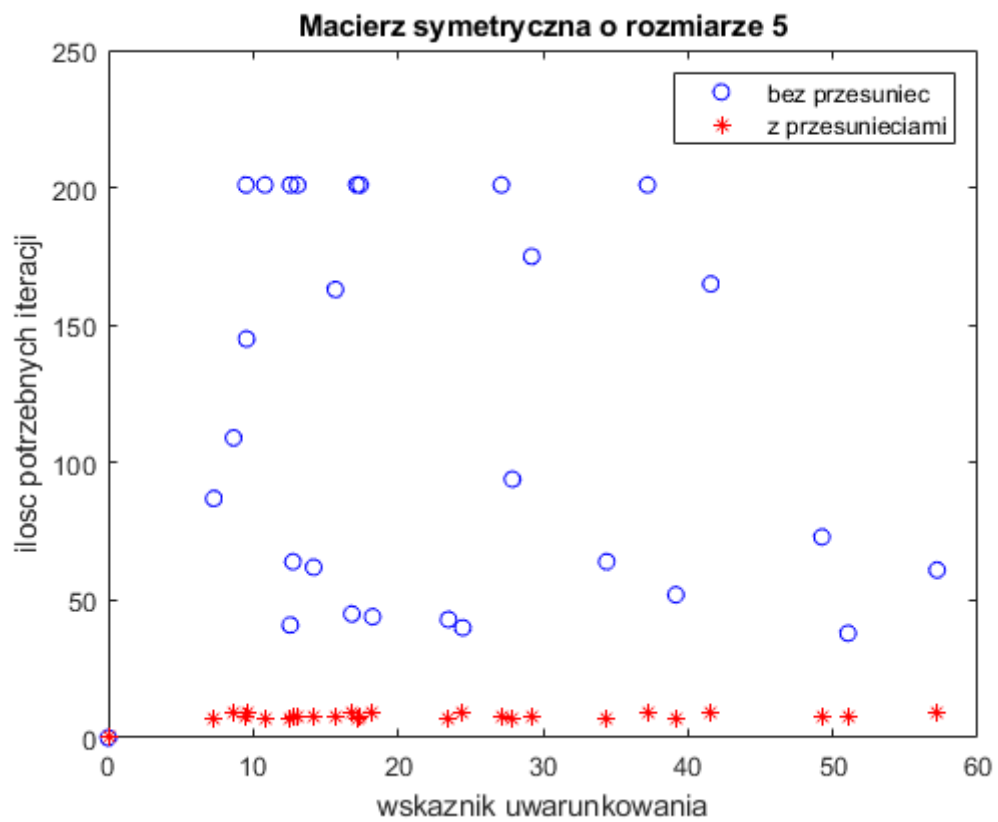
1. Do sprawdzenia poprawności wyznaczonych wartości własnych został wykorzystany mały program testujący, którego główną funkcjonalnością było sprawdzenie maksymalnego błędu:

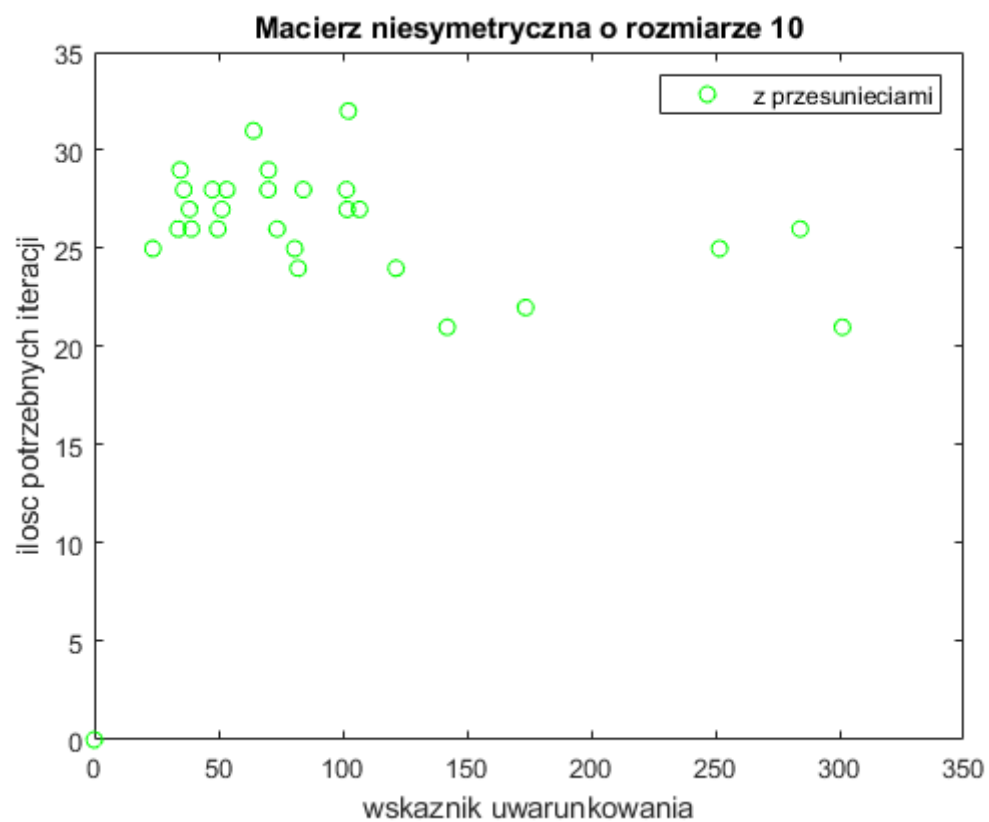
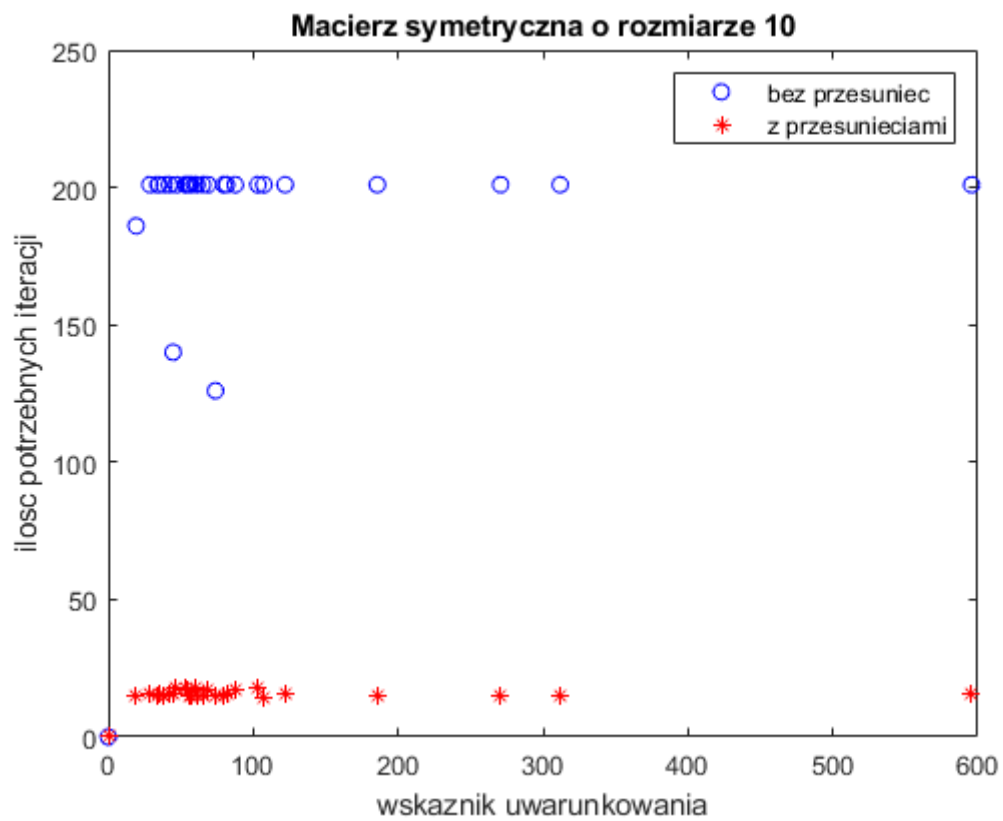
```
rozmiar = 20;  
A = macierz_symetryczna(rozmiar);  
[B iteracje_sym_bezprzes] = qr_bezprzesuniec(A);  
max(sort(B) - eig(A))
```

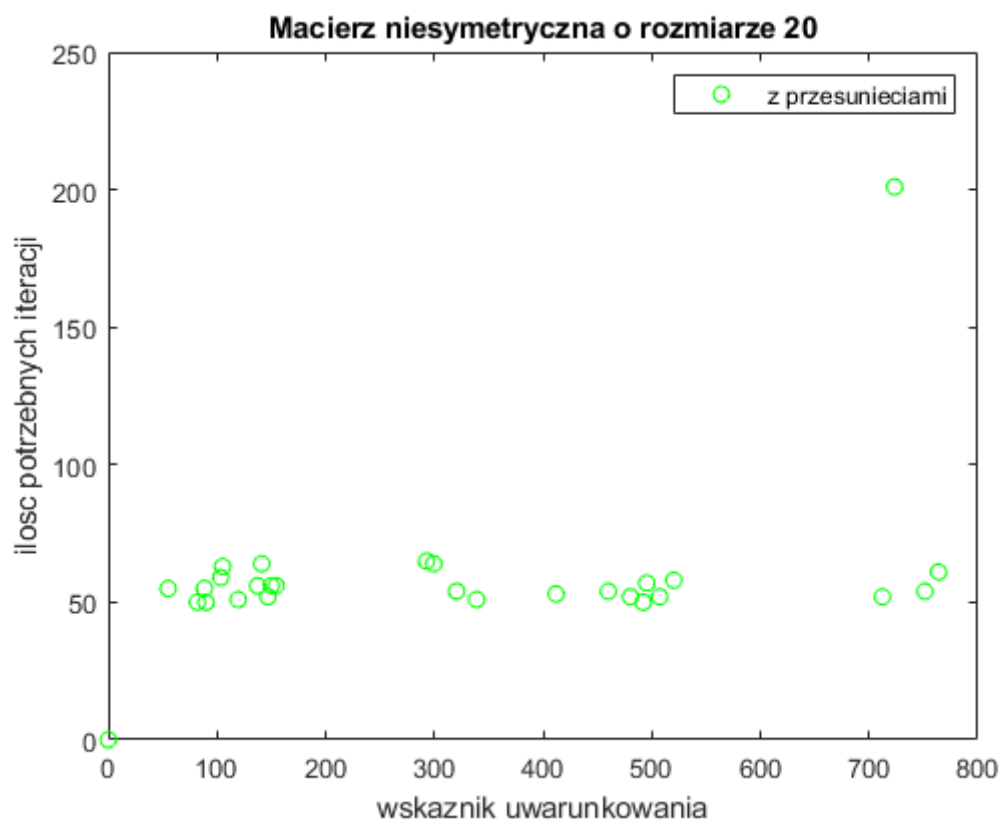
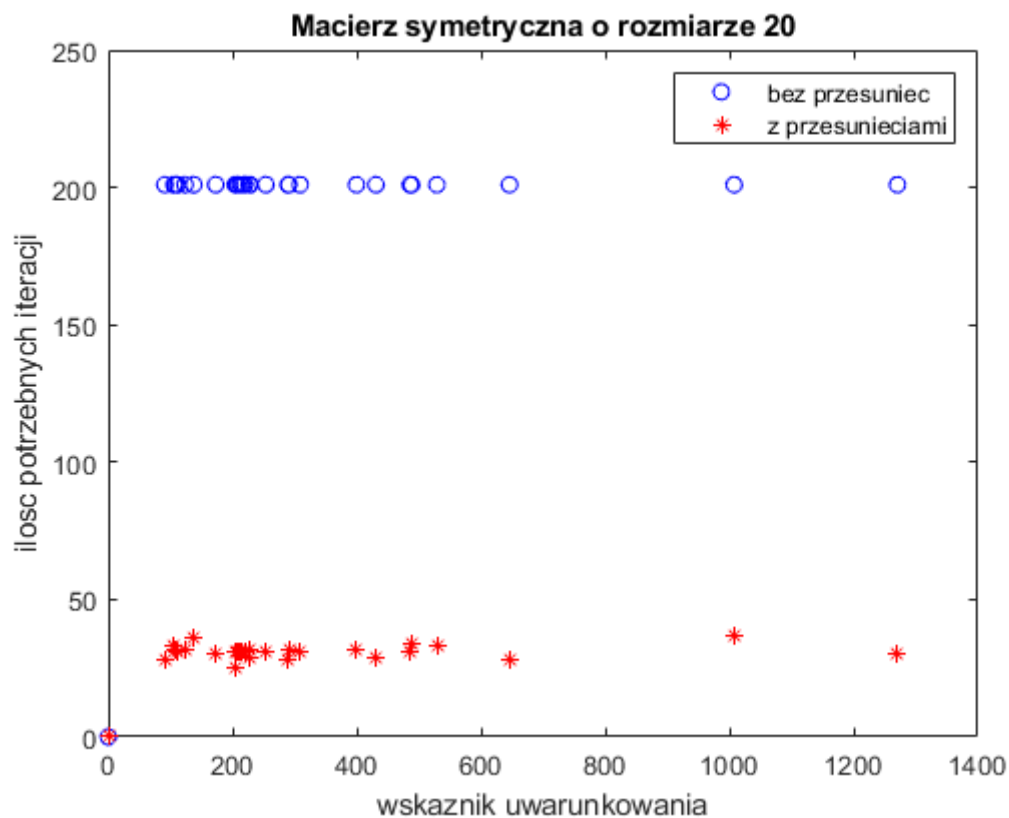
Przy testach wartość maksymalna była rzędu 10^{-13} (ostatni pomiar - $1.2079e-13$) dla metody z przesunięciami i macierzy 20x20. Z kolei dla tego samego rozmiaru bez przesunień dokładność ta była rzędu 0,1 (ostatnia wartość 0.1438, 20 to wielkość macierzy, przy której warunek o maksymalnej ilości iteracji przerywa wykonanie algorytmu w większości przypadków).

Zostało to uznane za wystarczającą dokładność. Niestety dla wartości zespolonych funkcja sort() nie działa tak jak funkcja sortująca eig() jednakże po wyświetleniu wartości własnych za pomocą autorskiej funkcji oraz eig() przy 10 różnych macierzach wyniki pokrywały się co zostało uznane za wystarczający test zgodności.

3.







Rozmiar\Metoda	Macierz symetryczna		Macierz niesymetryczna
	Bez przesunięć	Z przesunięciami	Z przesunięciami
5	105.7667	7.1667	10.6333
10	175.8667	14.3000	23.8000
20	180.9000	27.9667	54.8333

Komentarz

Metody działają poprawnie. Ta z przesunięciami okazała się ok 10 razy bardziej skuteczna. Przy macierzy 5x5 1/3 przebiegów algorytmu bez przesunięć kończyła się niepowodzeniem, przy macierzy 10x10 zaledwie 1/10 przebiegów algorytmu zakończył się powodzeniem zaś przy wielkości 20x20 ani jeden przebieg algorytmu nie skończył się w założonych 200 iteracjach. Świadczy to o jego wysokiej niewydajności. Dodanie prostego elementu przesunięcia znacznie zwiększa efektywność algorytmu.

Dla porównania zaledwie 1 na 30 prób przy algorytmie z przesunięciami dla macierzy niesymetrycznych zakończona była porażką (rozmiar 10 i 20). Zaś dla symetrycznych można powiedzieć, że ilość iteracji była w pewnym przybliżeniu stała.

Co ciekawe wskaźnik uwarunkowania nie miał wpływu na ilość wykonanych przez algorytm iteracji.

Zadanie 2. Aproksymacja funkcji

Celem zadania jest napisanie programu, który będzie aproksymował wielomianową funkcję na podstawie zadanych punktów dwiema metodami:

1. układu równań normalnych
2. układu równań liniowych wynikającego z rozkładu QR

Ponadto dla każdego układu należy obliczyć błąd rozwiązania jako normę residuum.

Koncepcja rozwiązania

Dla zestawu (x,y):

```
dane = [-5 -5.4606;-4 -3.8804;-3 -1.9699;-2 -1.6666;-1 -0.0764;0 -  
0.3971;1 -1.0303;2 -4.5483;3 -11.528;4 -21.6417;5 -34.4458];
```

wyznaczone zostały aproksymacje na podstawie obu metod. Wykorzystane do tego zostały funkcje, które za parametr przyjmują wektor x, y oraz stopień wielomianu, który ma służyć jako przybliżenie funkcji tworzącej dane.

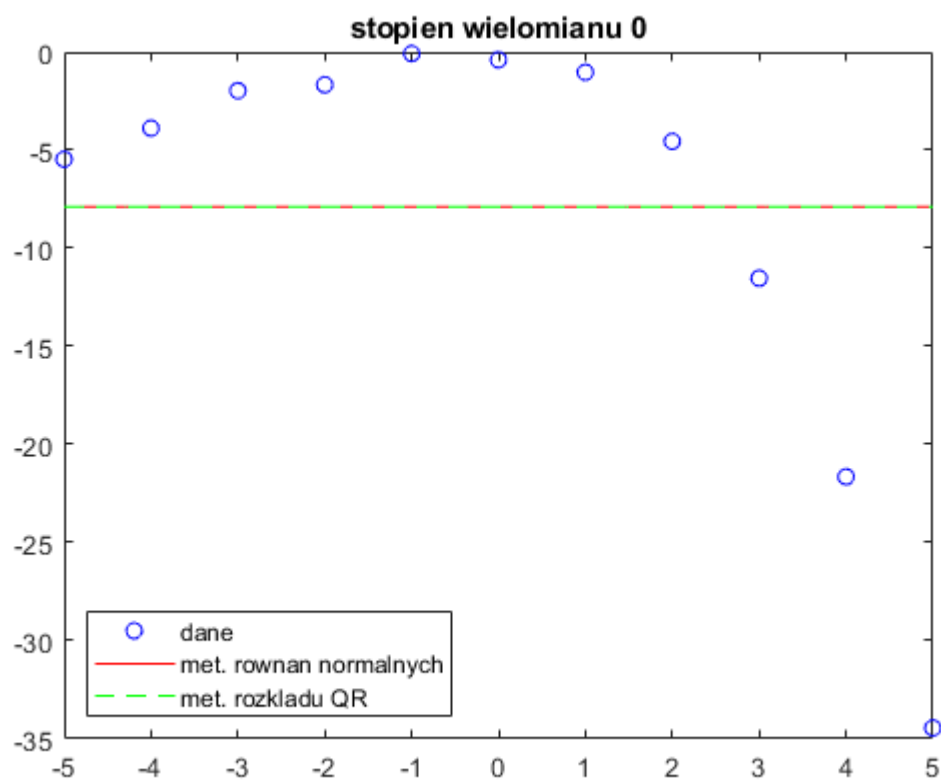
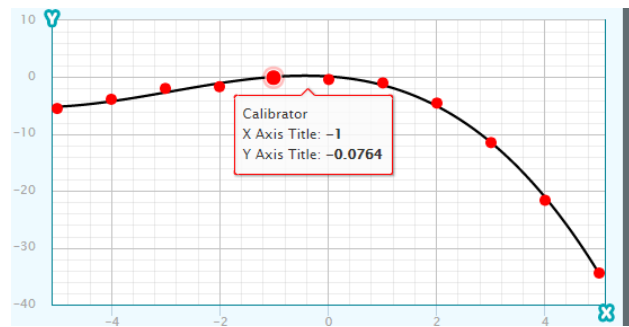
1. Algorytm układu równań normalnych:
 - 1) wyznaczenie macierzy (G) Grama jako iloczynu przekształceń – w tym przypadku sumowanie potęg x
 - 2) wyznaczenie macierzy prawej strony (P) jako *przekształcenie*korespondująca wartość wyjściowa (y)*
 - 3) obliczenie równania $GX = P$.
komentarz: pozwoliłem sobie użyć wbudowanej w Matlaba funkcji `\`, gdyż przy poprzednim zadaniu samodzielnie pisałem funkcję obliczającą równania tego typu.
 - 4) potraktuj wyjście jako zbiór współczynników kolejnych potęg x.
2. Algorytm oparty na rozkładzie QR różni się jedynie tym, że w kroku 3 nie zostaje wykonane obliczenie równania $GX = P$ tylko $Rx=Q^TP$.
Komentarz: do dokonania rozkładu QR została wykorzystana napisana przeze mnie metoda rozkładu QR z poprzedniego zadania.

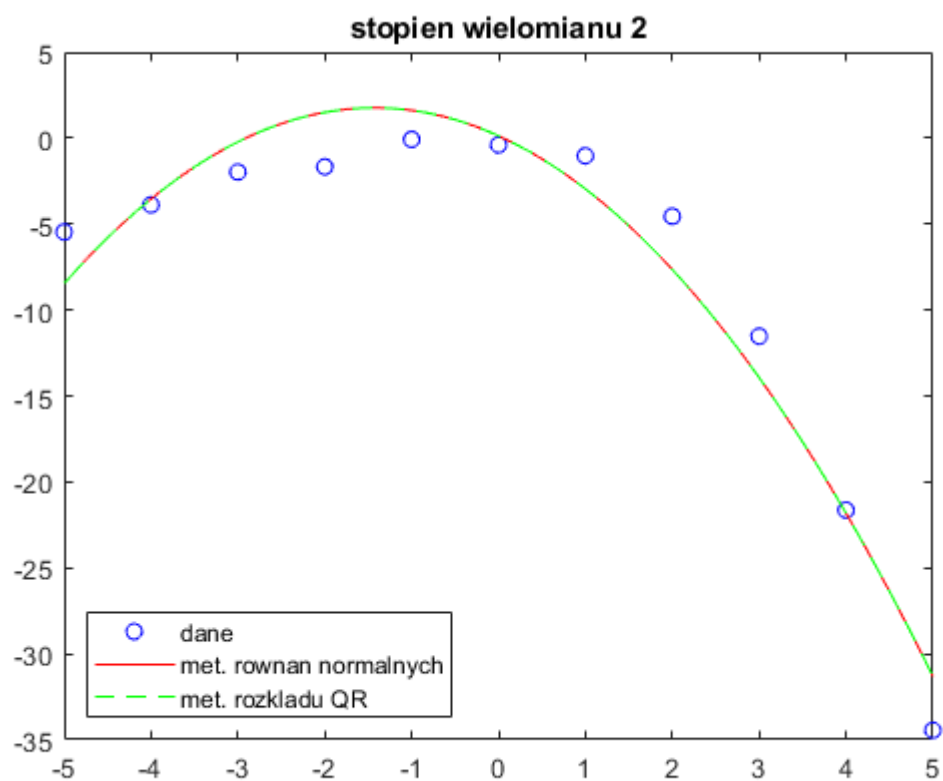
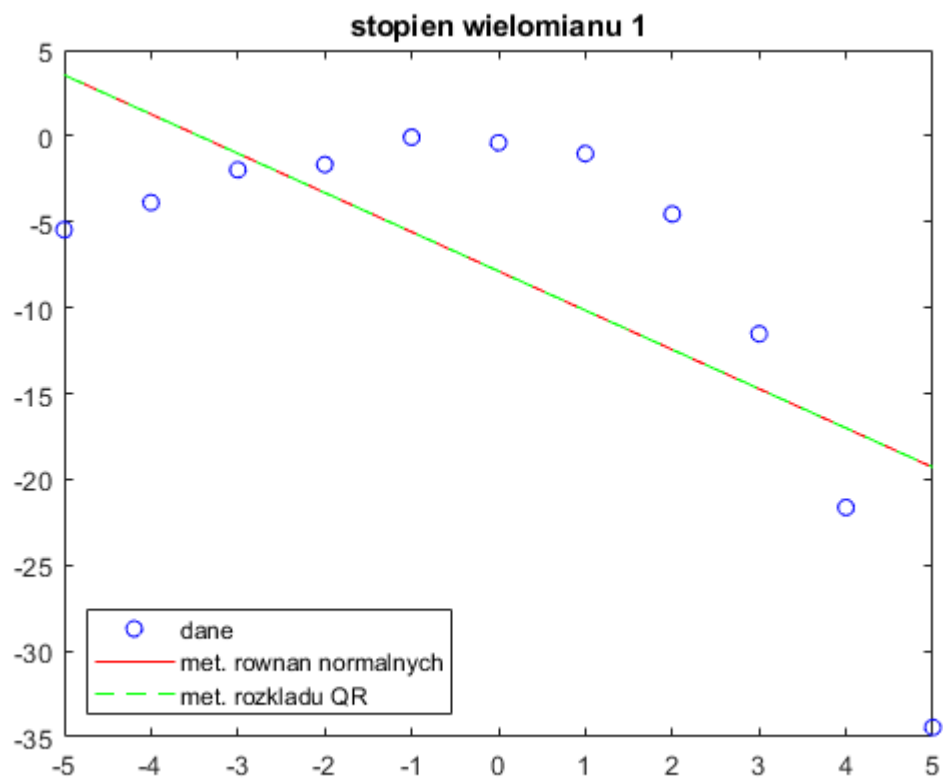
Sprawdzenie

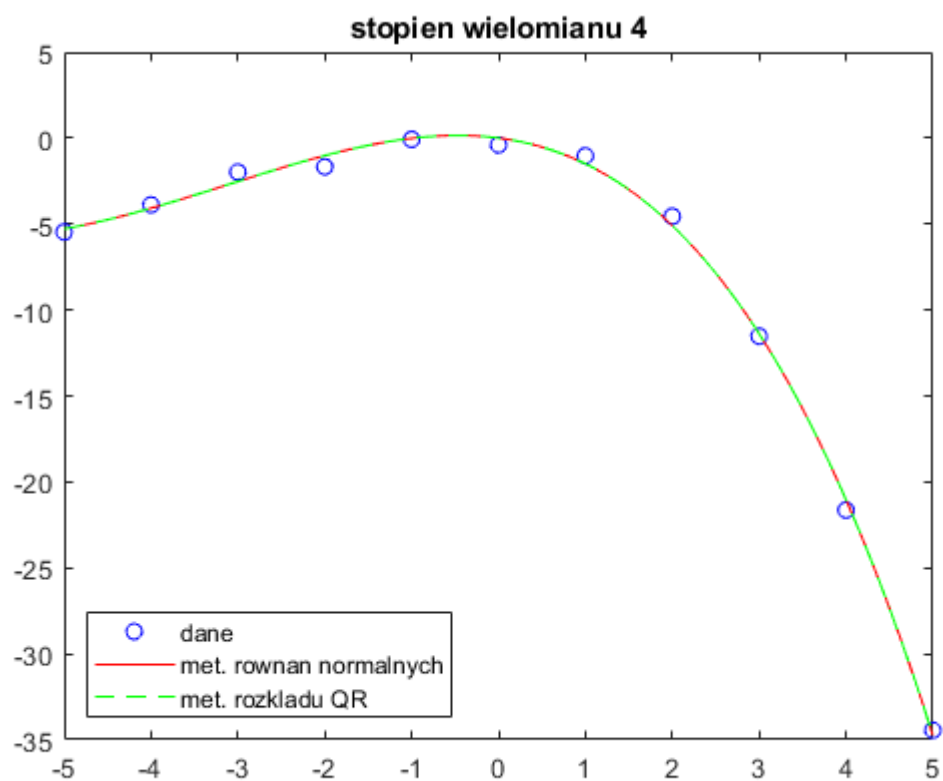
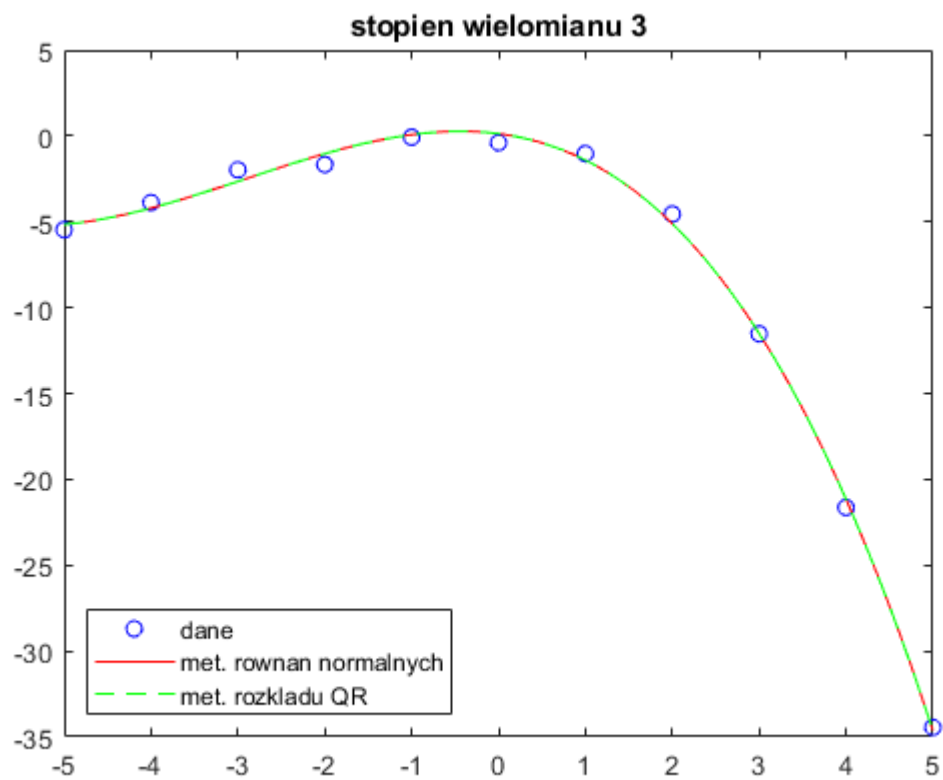
W celu sprawdzenia poprawności wykreowanych rozwiązań skorzystałem z:

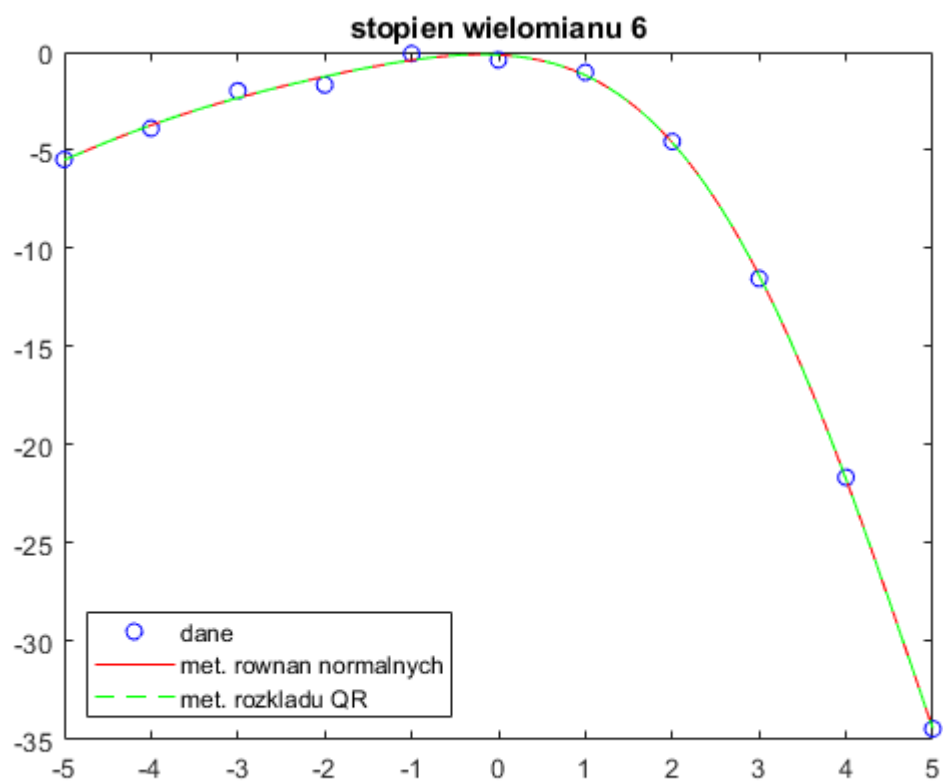
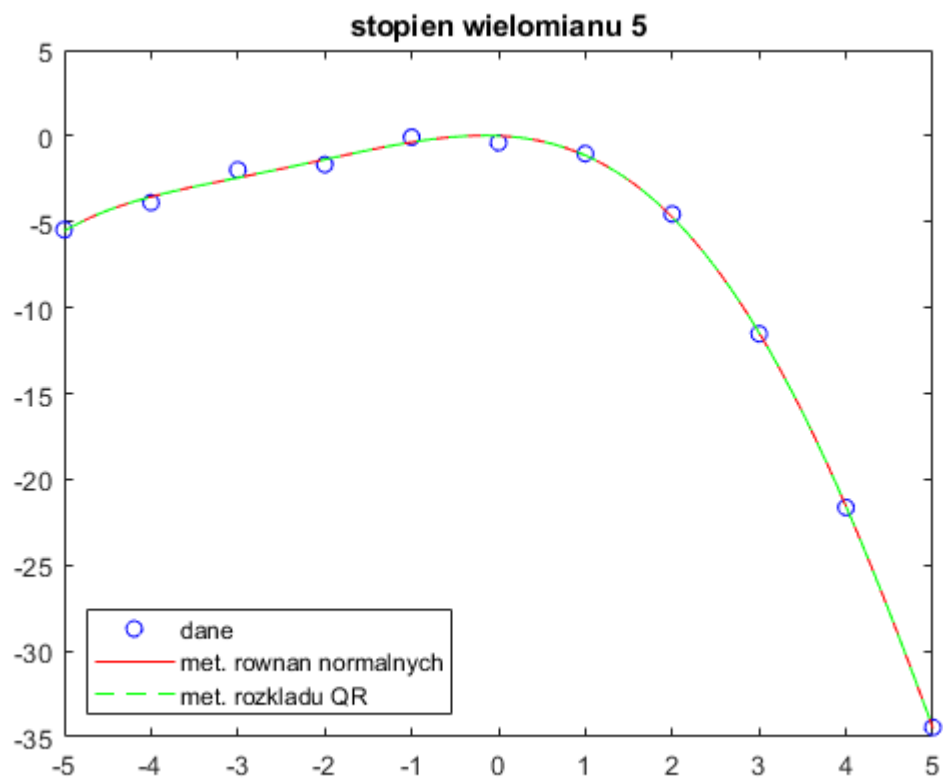
- efekt finalny: obliczania normy residuum
- poprawności rysowanych wykresów: strona, która na podstawie wyjścia generowanego przez mój program tworzyła wykresy, np. https://www.wolframalpha.com/input/?i=-0.09-0.8*x-0.65*x%5E2%2B0.13*x%5E3

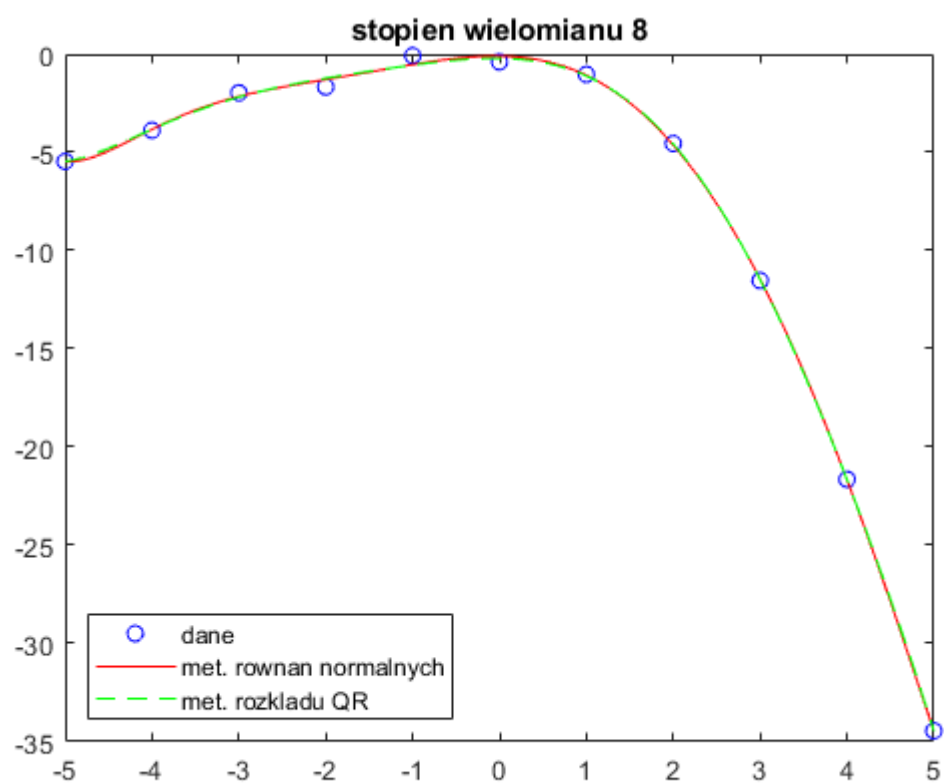
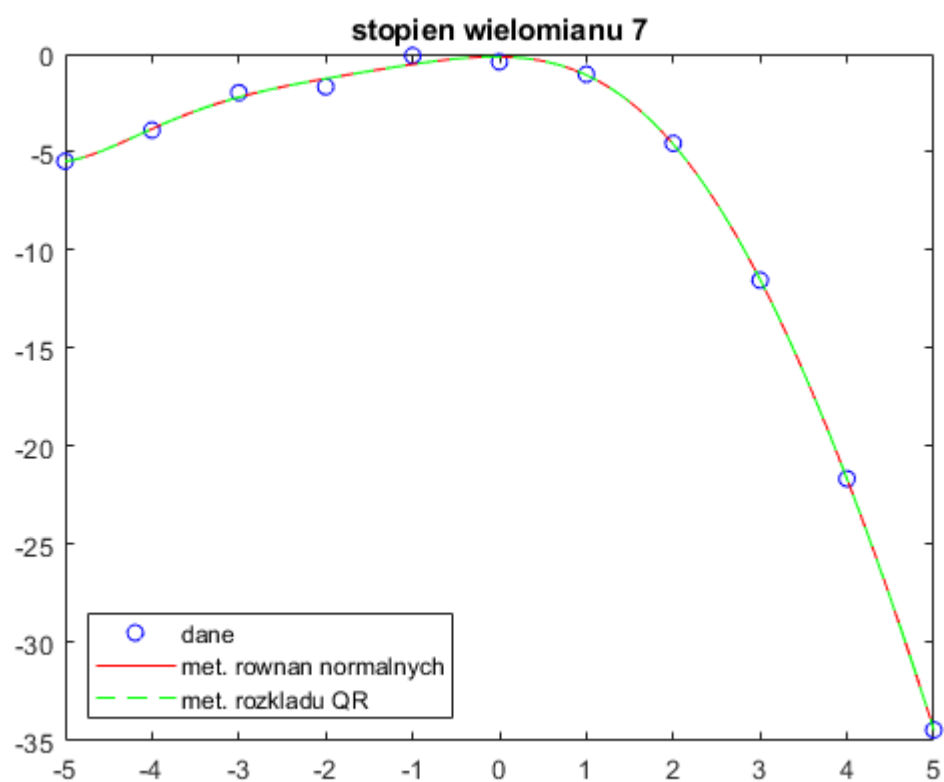
- dokładność przybliżenia: na podstawie danych weryfikowałem czy wykres jest dość dokładny jak na zadane wejście za pomocą <https://mycurvefit.com/> (przykład generowany przez stronę dla 3 stopnia)











```

norma_res = 9x2
met. rown norm      met QR
  34.3326          34.3326 - stopień 0
  24.5832          24.5832 - stopień 1
   7.3647          7.3647 - stopień 2
   1.4390          1.4390 - stopień 3
   1.3958          1.3958 - stopień 4
   0.8501          0.8501 - stopień 5
   0.7595          0.7595 - stopień 6
   0.7069          0.7069 - stopień 7
   0.6997          0.7181 - stopień 8

```

Komentarz

Obie metody dobrze dokonują aproksymacji dla zadanego zestawu danych już przy 3 stopniu wielomianu. Do 7. stopnia wielomianu różnice pomiędzy obiema metodami są wręcz niezauważalne – zarówno wykresy jak i norma residuum są takie same. Przy większym stopniu metoda QR traci nieznacznie na rzecz metody równań normalnych. Wynikać to może ze stosowania dodatkowego rozkładu, który przy większych macierzach nieco traci na dokładności – może to dziać się np. na etapie ortogonalizacji Grama-Schmidta gdyż występuje coraz większe pole do popełnienia błędów np. w elemencie sumy.

Załącznik 1. Kod źródłowy zadania 1.

program

```
clc;
clear;
iteracje_sym_bezprzes = zeros(30);
iteracje_sym_przes = zeros(30);
cond_sym = zeros(30);
cond_niesym = zeros(30);
iteracje_niesym_przes = zeros(30);
for rozmiar = [5 10 20]
    for i = 1:30
        A = macierz_symetryczna(rozmiar);
        [B iteracje_sym_bezprzes(i,rozmiar)] = qr_bezprzesuniec(A);
        [B iteracje_sym_przes(i,rozmiar)] = qr_przesuniecia(A);
        cond_sym(i,rozmiar) = cond(A);
        %eig(A);
        A = macierz_niesymetryczna(rozmiar);
        [B iteracje_niesym_przes(i,rozmiar)] = qr_przesuniecia(A);
        cond_niesym(i,rozmiar) = cond(A);
    end
end
```

statystyki i czyszczenie danych

```
for rozmiar = [5 10 20]
    rozmiar
    sr_sym_bezprzes = iteracje_sym_bezprzes(:,rozmiar)
    sr_sym_przes = iteracje_sym_przes(:,rozmiar)
    sr_niesym_przes = iteracje_niesym_przes(:,rozmiar)
    for i = 1:3
        [wart indx] = max(cond_sym(:,rozmiar));
        iteracje_sym_bezprzes(indx,rozmiar) = 0;
        iteracje_sym_przes(indx,rozmiar) = 0;
        cond_sym(indx,rozmiar) = 0;

        [wart indx] = max(cond_niesym(:,rozmiar));
        iteracje_niesym_przes(indx,rozmiar) = 0;
        cond_niesym(indx,rozmiar) = 0;
    end
end
```

```
for rozmiar = [5 10 20]
    rozmiar
    sr_sym_bezprzes = mean(iteracje_sym_bezprzes(:,rozmiar))
    sr_sym_przes = mean(iteracje_sym_przes(:,rozmiar))
    sr_niesym_przes = mean(iteracje_niesym_przes(:,rozmiar))
end
```

wykresy

```
for rozmiar = [5 10 20]
    figure
```

```

    plot(cond_sym(:,rozmiar),iteracje_sym_bezprzes(:,rozmiar),'bo',
cond_sym(:,rozmiar), iteracje_sym_przes(:,rozmiar), 'r*');
    title(['Macierz symetryczna o rozmiarze ' num2str(rozmiar)])
    xlabel('wskaznik uwarunkowania')
    ylabel('ilosc potrzebnych iteracji')
    legend({'bez przesuniec','z przesunieciami'},'Location','northeast');
    figure
    plot(cond_niesym(:,rozmiar),iteracje_niesym_przes(:,rozmiar),'go');
    title(['Macierz niesymetryczna o rozmiarze ' num2str(rozmiar)])
    xlabel('wskaznik uwarunkowania')
    ylabel('ilosc potrzebnych iteracji')
    legend({'z przesunieciami'},'Location','northeast');
end

```

funkcje pomocnicze

rozklad QR

```

function [Q R] = qr_rozklad(A)
    [r_wiersze r_kolumny] = size(A);
    Q = zeros(r_wiersze);
    if r_wiersze > r_kolumny
        R = eye(r_wiersze);
        Q = eye(r_wiersze);
    else
        R = eye(r_kolumny);
        Q = eye(r_wiersze);
    end
    %Gram-Schmidt
    for i = 1:r_kolumny
        Q(:,i) = A(:,i);
        for j = 1:(i-1)
            R(j,i) = mydot(Q(:,j),A(:,i))/mydot(Q(:,j),Q(:,j));
            Q(:,i) = Q(:,i) - R(j,i)*Q(:,j);
        end
    end
    Q = Q(1:r_wiersze,1:r_kolumny);
    %normalizacja
    N = zeros(r_wiersze);
    for i = 1:r_kolumny
        N(i,i) = norm(Q(:,i));
        Q(:,i) = Q(:,i)/N(i,i);
    end
    R = N*R;

    if r_wiersze > r_kolumny
        R = R(1:r_kolumny,1:r_kolumny);
    else
        R = R(1:r_wiersze,1:r_wiersze);
    end
end

```

algorytm obliczania wartosci wlasnych metoda QR bez przesuniec

```

function [wart_wlasne i] = qr_bezprzesuniec(A)

```



```

i = 0;
while tolerancja(A) > 0.00001 & i < 200+1
    [Q R] = qr_rozklad(A);
    A = R * Q;
    i = i+1;
end
wart_wlasne = wektor(A);
end

```

algorytm obliczania wartosci wlasnych metoda QR z przesunieciami

```

function [wart_wlasne i] = qr_przesuniecie(A)
    rozmiar = size(A,1);
    i = 0;
    wart_wlasne = zeros(rozmiar);
    wart_wlasne = wart_wlasne(:,1);
    for j = rozmiar:-1:2
        while max(abs(A(j,1:j-1))) > 0.00001 & i < 200+1
            mala_macierz = A(j-1:j,j-1:j); %
macierz 2x2,
            [x1 x2] = pierw_f_kwadratowej(mala_macierz);
            przesuniecie = blizsza_liczba(mala_macierz(2,2), x1, x2); % z
ktorej wyznaczana jest najlepsza wart. wlasna
            A = A - eye(j)*przesuniecie;
            [Q R] = qr_rozklad(A);
            A = R * Q + eye(j)*przesuniecie;
            i = i+1;
        end
        wart_wlasne(j) = A(j,j);
        if j > 2
            A = A(1:j-1,1:j-1); %deflacja
        else
            wart_wlasne(1) = A(1,1);
        end
    end
end
end

```

wyznaczanie pierw f. kwadratowej

```

function [x1 x2] = pierw_f_kwadratowej(mala_macierz)
    a = 1;
    b = -(mala_macierz(1,1)+mala_macierz(2,2));
    c = (mala_macierz(1,1)*mala_macierz(2,2))-
(mala_macierz(2,1)*mala_macierz(1,2));

    x1 = (-b + sqrt(b*b - 4*a*c))/(2*a);
    x2 = (-b - sqrt(b*b - 4*a*c))/(2*a);
    if abs(x2) > abs(x1)
        x1 = x2;
    end
    %drugi pierwiastek ze wzorów Viète'a
    x2 = ((-b)/a) - x1;
end

```

wybor pierwiastka blizszego d(n,n)

```
function x = blizsza_liczba(wlasciwa, x1, x2)
    if abs(wlasciwa-x1) < abs(wlasciwa-x2)
        x = x1;
    else
        x = x2;
    end
end
```

autorska implementacja matlabowej funkcji dot()

```
function md = mydot(A,B)
    rozmiar = size(A);
    md = 0;
    for i = 1:rozmiar
        md = md + A(i)*B(i);
    end
end
```

funkcja wektoryzujaca macierz diagonalna

```
function w = wektor(A)
    rozmiar = size(A);
    for i = 1:rozmiar
        w(i,1) = A(i,i);
    end
end
```

sprawdzenie tolerancji

```
function tol = tolerancja(A)
    rozmiar = size(A);
    A = abs(A);
    tol = 0;
    if rozmiar > 2
        for i = 1:rozmiar
            if max(A(i,i+1:end)) > tol
                tol = max(A(i,i+1:end));
            end
            if max(A(i,1:i-1)) > tol
                tol = max(A(i,1:i-1));
            end
        end
    else
        tol = 0;
    end
end
```

tworzenie macierzy symetrycznej o zadanym rozmiarze

```
function mac_sym = macierz_symetryczna(rozmiar)
    mac_sym = randi([0 50],rozmiar,rozmiar);
    mac_sym = mac_sym + mac_sym';
end
```

tworzenie macierzy niesymetrycznej o zadanym rozmiarze

```
function mac_nsym = macierz_niesymetryczna(rozmiar)
```

```
    mac_nsym = randi([0 100],rozmiar,rozmiar);  
end
```

norma residuum

```
function nr = norma_residuum(wspolczynniki, x, rozw)  
    residuum = wspolczynniki*x - rozw;  
    nr = norm(residuum);  
end
```

Załącznik 2. Kod źródłowy zadania 2.

program

```
clc;
clear;

dane = [-5 -5.4606;-4 -3.8804;-3 -1.9699;-2 -1.6666;-1 -0.0764;0 -0.3971;1 -
1.0303;2 -4.5483;3 -11.528;4 -21.6417;5 -34.4458];

x = linspace(-5,5,100);

max_stopien = 8;

norma_res = zeros(max_stopien);

norma_res = norma_res(:,1:2);

for stopien = 0:max_stopien
    figure

    funkcja = uklad_rownan_normalnych(dane, stopien);
    y = fun(funkcja, x);
    funkcja2 = uklad_qr(dane, stopien);
    y2 = fun(funkcja2, x);
    plot(dane(:,1),dane(:,2),'bo', x, y, 'r', x, y2, 'g--');
    legend({'dane','met. rownan normalnych','met. rozkladu
QR'}, 'Location', 'southwest');
    title(['stopien wielomianu ' num2str(stopien)]);
    norma_res(stopien+1,1) = norm(dane(:,2) - fun(funkcja, dane(:,1)));
    norma_res(stopien+1,2) = norm(dane(:,2) - fun(funkcja2, dane(:,1)));
end

norma_res
```

funkcje pomocnicze

uklad rownan normalnych

```
function wspolczynniki = uklad_rownan_normalnych(dane, st_wielomianu)

% wyznaczanie macierzy Grama - <przekształcenie_i,przekształcenie_j>

[r_wiersze, r_kolumny] = size(dane);

st_wielomianu = st_wielomianu + 1;

macierz_Grama = wyzn_macierz_Grama(dane, st_wielomianu);

% wektor prawej strony
```

```

prawa_strona = zeros(st_wielomianu);
prawa_strona = prawa_strona(:,1);
for i = 1:st_wielomianu
    for k = 1:r_wiersze
        prawa_strona(i) = prawa_strona(i) + (dane(k,1))^(i-1)*dane(k,2);
    end
end

wspolczynniki = macierz_Grama\prawa_strona;
end

```

układ wynikający z rozkładu QR

```

function wspolczynniki = ukklad_qr(dane, st_wielomianu)
    % wyznaczanie macierzy Grama - <przekształcenie_i,przekształcenie_j>
    [r_wiersze, r_kolumny] = size(dane);
    st_wielomianu = st_wielomianu + 1;
    macierz_Grama = wyzn_macierz_Grama(dane, st_wielomianu);

    % wektor prawej strony
    prawa_strona = zeros(st_wielomianu);
    prawa_strona = prawa_strona(:,1);
    for i = 1:st_wielomianu
        for k = 1:r_wiersze
            prawa_strona(i) = prawa_strona(i) + (dane(k,1))^(i-1)*dane(k,2);
        end
    end

    [Q R] = qr_rozklad(macierz_Grama);
    wspolczynniki = R\Q'*prawa_strona;
end

```

wyznaczenie macierzy Grama

```

function macierz_Grama = wyzn_macierz_Grama(dane, st_wielomianu)
    % wyznaczanie macierzy Grama - <przekształcenie_i,przekształcenie_j>
    macierz_Grama = zeros(st_wielomianu);

```

```

[r_wiersze, r_kolumny] = size(dane);
for i = 1:st_wielomianu
    for j = 1:st_wielomianu
        for k = 1:r_wiersze
            macierz_Grama(i,j) = macierz_Grama(i,j) + (dane(k,1))^(i+j-2);
        end
    end
end
end
end

```

wyznaczenie wyjść dla podanych x-ów i zadanej funkcji

```

function y = fun(funkcja, x)
    [temp rozmiar_x] = size(x);
    if rozmiar_x == 1
        x = x';
        rozmiar_x = temp;
    end
    st_wielomianu = size(funkcja);
    y = zeros(rozmiar_x);
    y = y(:,1);
    for i = 1:rozmiar_x
        for j = 1:st_wielomianu
            y(i,1) = y(i,1) + funkcja(j,1)*(x(1,i))^(j-1);
        end
    end
end
end

```

rozkład QR

```

function [Q R] = qr_rozklad(A)
    [r_wiersze r_kolumny] = size(A);
    Q = zeros(r_wiersze);
    if r_wiersze > r_kolumny
        R = eye(r_wiersze);
        Q = eye(r_wiersze);
    else

```

```

        R = eye(r_kolumny);
        Q = eye(r_wiersze);
    end
    %Gram-Schmidt
    for i = 1:r_kolumny
        Q(:,i) = A(:,i);
        for j = 1:(i-1)
            R(j,i) = mydot(Q(:,j),A(:,i))/mydot(Q(:,j),Q(:,j));
            Q(:,i) = Q(:,i) - R(j,i)*Q(:,j);
        end
    end
    Q = Q(1:r_wiersze,1:r_kolumny);
    %normalizacja
    N = zeros(r_wiersze);
    for i = 1:r_kolumny
        N(i,i) = norm(Q(:,i));
        Q(:,i) = Q(:,i)/N(i,i);
    end
    R = N*R;

    if r_wiersze > r_kolumny
        R = R(1:r_kolumny,1:r_kolumny);
    else
        R = R(1:r_wiersze,1:r_wiersze);
    end
end

```

autorska implementacja matlabowej funkcji dot()

```

function md = mydot(A,B)
    rozmiar = size(A);
    md = 0;
    for i = 1:rozmiar
        md = md + A(i)*B(i);
    end

```

```
end
```

funkcja wektoryzująca macierz diagonalną

```
function w = wektor(A)
    rozmiar = size(A);
    for i = 1:rozmiar
        w(i,1) = A(i,i);
    end
end
```