MNUM-PROJEKT 2, zadanie 2.24

Wojciech Grunwald (311566)

08.05.2023

1 Wstęp

Celem projektu jest znalezienie wszystkich pierwiastków danej funkcji, korzystając ze środowiska *Matlab* dwoma różnymi metodami:

1. siecznych (własna implementacja)
2. Newtona (zapewniony gotowy solwer)‘

Funkcja to:

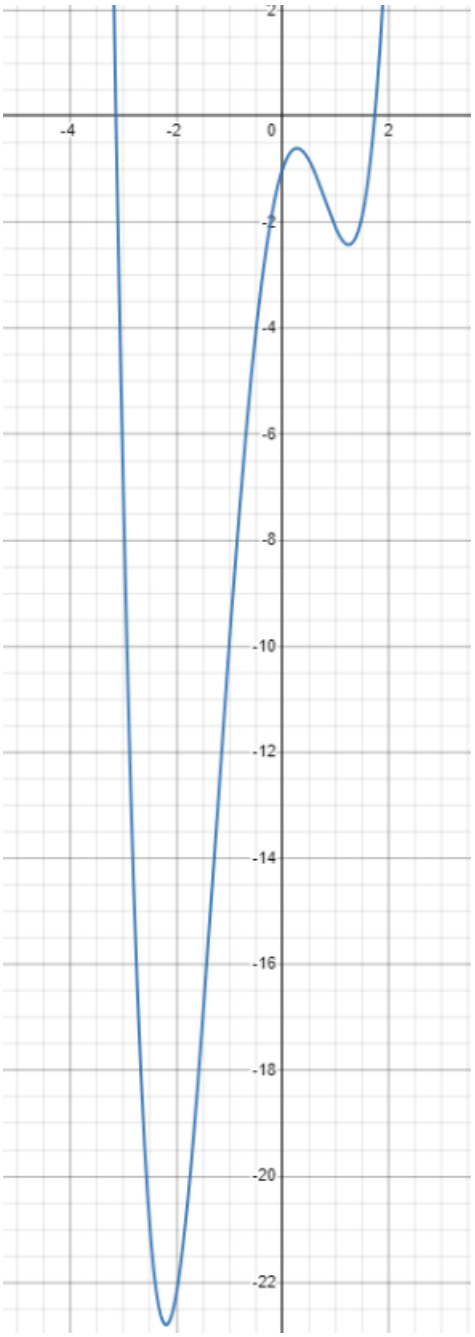
$$f(x) = 0.7cos(x) - ln(x+1) \tag{1}$$

w przedziale [2, 11]

Rysunek 1: Przebieg analizowanej funkcji

A także wszystkich pierwiastków wielomianu:

$$f(x) = x^4 + 0.9x^3 - 6x^2 + 3x - 1 \tag{2}$$



Rysunek 2: Przebieg analizowanego wielomianu

Za pomocą algorytmu Müllera MM1 (własna implementacja)

Algorytmy zaimplementowałem w *Matlabie* w wersji R2023a. Obliczenia były wykonywane na procesorze Intel Core i5-9300H CPU 2.40Ghz.

W folderze *Kod źródłowy* załączonym do sprawozdania znajdują się odpowiednie skrypty rozwiązujące zadania:

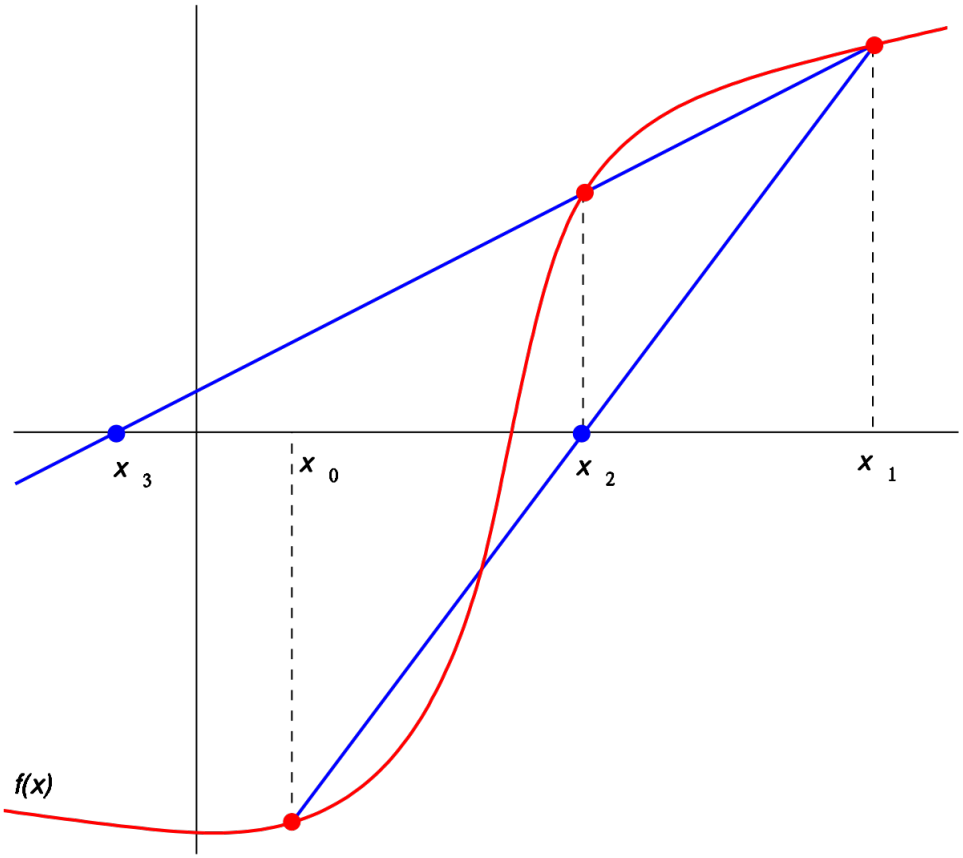
1. ZAD1.m - skrypt korzystający z własnego solwera siecznych i gotowego solwera Newtona

2. ZAD2.m - skrypt, korzystający z metody Müllera
3. secant.m - skrypt znajdujący pierwiastki metodą siecznych
4. newton.m - gotowy skrypt znajdujący pierwiastki metodą Newtona
5. Function.m - skrpyt z docelową funkcją
6. RangeMaker.m - skrypt generujący przedziały izolacji pierwiastków
7. horner.m - skrypt wykonujący deflację czynnikiem liniowym
8. MM1.m - skrypt znajdujący pierwiastki wielomianu metodą Müllera
   (a) groupConsec.m, groupLims.m, groupTrue.m - skrypty użyte w ramach załączonej licencji do wyznaczenia przedziałów izolacji w RangeMaker.m

2 Metoda siecznych znajdowania pierwiastka funkcji

2.1 Wprowadzenie teoretyczne

W analizie numerycznej metoda siecznych jest algorytmem znajdowania miejsc zerowych, który wykorzystuje kolejne miejsca zerowe siecznych linii w celu coraz lepszego przybliżenia miejsca zerowego funkcji f. Metoda siecznych można traktować jako przybliżenie różnicowe metody Newtona. Jednak metoda siecznych poprzedza metodę Newtona o ponad 3000 lat.



Rysunek 3: Metoda siecznych

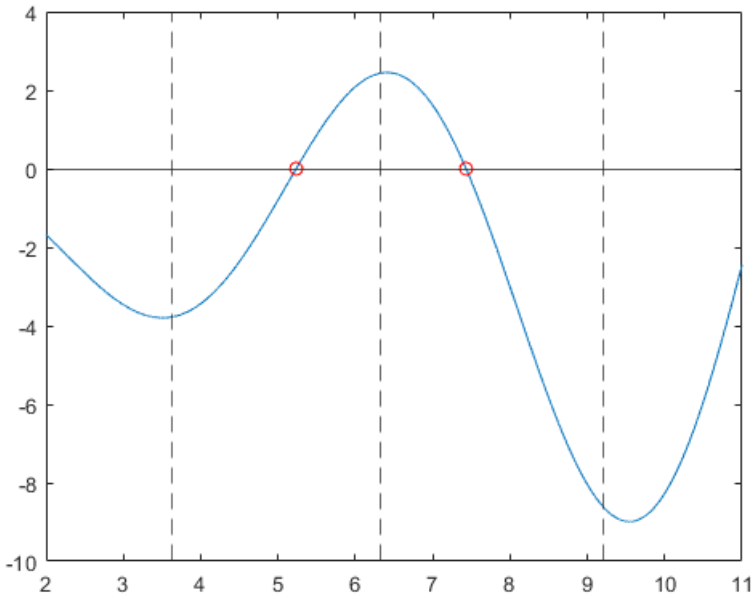
Aby znaleźć zero funkcji, metoda siecznych jest zdefiniowana przez relację rekurencyjną:

$$x_n = x_{n-1} - f(x_{n-1})\frac{x_{n-1} - x_{n-2}}{f(x_{n-1}) - f(x_{n-2})} \tag{3}$$

2.1.1 Znajdowanie przedziałów izolacji pierwiastków

Metoda ta polega na znajdowaniu prawidłowości w wektorze początkowego przedziału, z którego chcemy wydzielić przedziały izolacji pierwiastków.
Na początku wyznaczamy miejsca, w których zmienia się znak funkcji i bierzemy średnią arytmetyczną między punktem "startowym", a punktem "końcowym", tym samym otrzymując żądane przedziały izolacji pierwiastków (algorytm działa tak, że nie trzeba podawać liczby pierwiastków, znajdzie je wszystkie).

2.2 Testowanie własnego solwera



Rysunek 4: Znalezione pierwiastki z zaznaczonymi przerywanymi liniami przedziałami izolacji

Zostały znalezione dwa przedziały izolacji pierwiastków przez funkcję *RangeMaker.m*, co zgadza się z wykresem funkcji w zadanym przedziale.

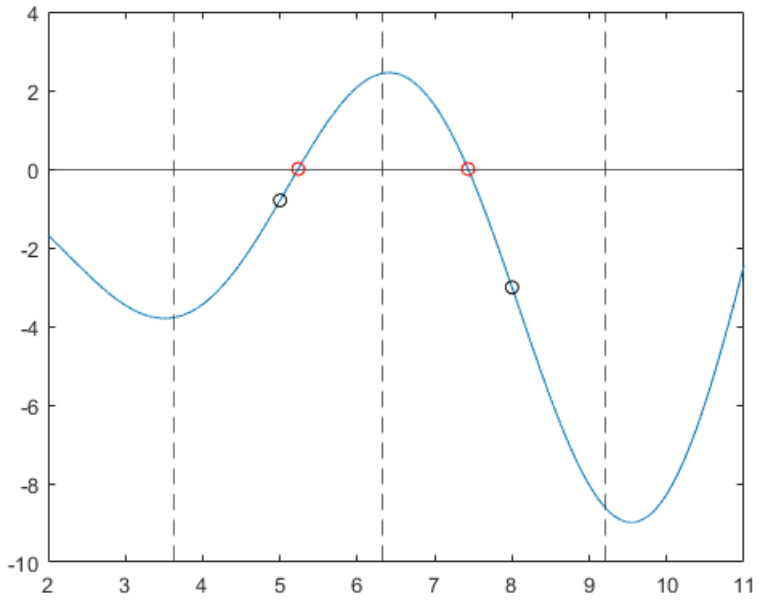
	Przedział izolacji	**f(a)**	**f(b)**
1	[3.6180, 6.3340]	-3.780552	2.435556
2	[6.3340, 9.2160]	2.435556	-8.635067

Tabela 1: Przedziały izolacji pierwiastków

	Znalezione pierwiastki x	**f(x)**	**Liczba iteracji**
1	5.2353	-1.585843e-12	5
2	7.4317	3.641532e-14	7

Tabela 2: Odnalezione pierwiastki metodą siecznych

3 Testowanie gotowego algorytmu Newtona



Rysunek 5: Znalezione pierwiastki z zaznaczonymi przerywanymi liniami przedziałami izolacji, czarnym kolorem oznaczone są punkty początkowe poszukiwań, a czerwonym kolorem znalezione pierwiastki

Przedziały izolacji te same. Punkt początkowy algorytmu brałem jako środek przedziału izolacji pierwiastka.

	Znal. pierwiastki x	**f(x)**	**Punkt początkowy x_{pp}**	**f(x_{pp})**	**Liczba iteracji**
1	5.2353	-5.329071e-15	5	-0.798942	7
2	7.4317	0	8	-3.012025	7

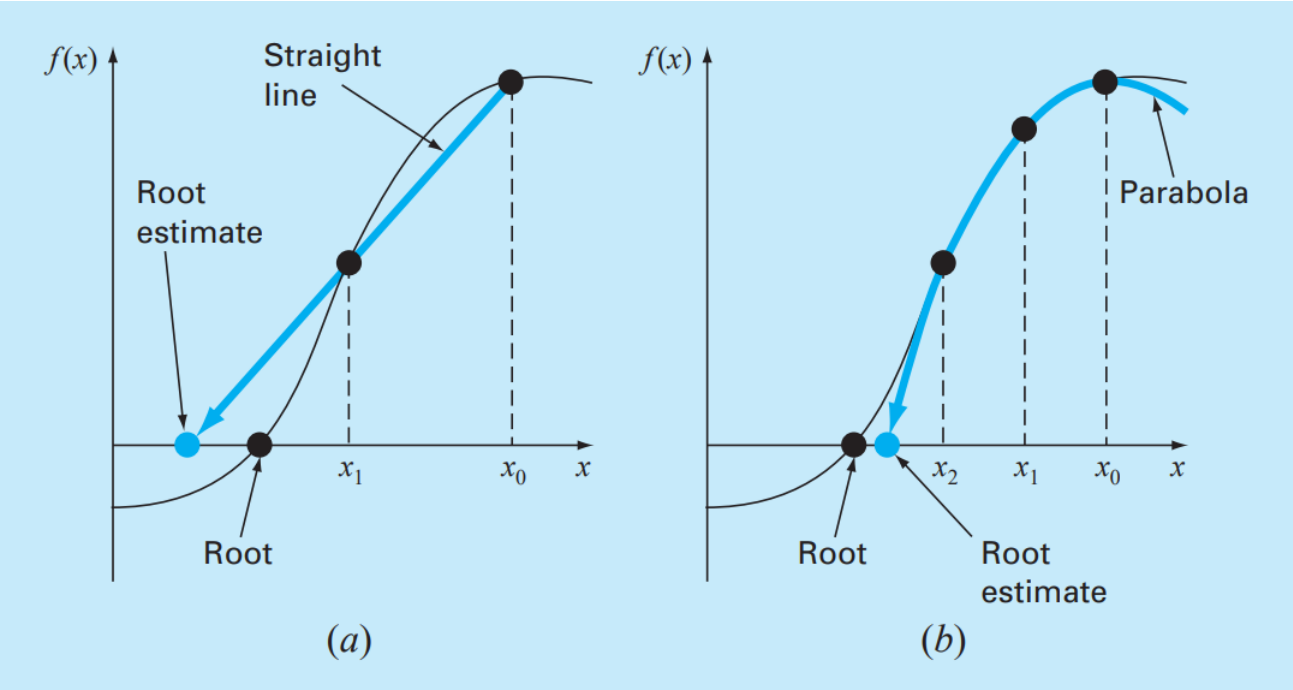
Tabela 3: Znalezione pierwiastki metodą Newtona

Porównując z wynikami dla metody siecznych można zauważyć, że dokładność dla metody Newtona jest większa niż dla metody siecznych przy podobnej liczbie iteracji. Metoda Newtona jest znana z szybkiego zbiegania do rozwiązania (tutaj można by stosować np. inne kryteria wyboru punktu początkowego, aby uzyskać jeszcze lepsze wyniki), zwłaszcza gdy początkowe przybliżenie jest dostatecznie bliskie rzeczywistego rozwiązania i gdy funkcja jest dobrze zachowana wokół pierwiastka. Jeśli te warunki są spełnione, metoda Newtona może osiągnąć wysoką dokładność.

4 Znalezienie wszystkich pierwiastków wielomianu metodą Müllera MM1

4.1 Wprowadzenie teoretyczne

Bierze pod uwagę wartości zespolone.
Polega ona na aproksymacji wielomianu w otoczeniu rozwiązania funkcją kwadratową. Jest pewnego rodzaju uogólnieniem metody siecznych - zamiast interpolacji w dwóch punktach funkcją liniową (tzn. sieczną) dokonujemy interpolacji w trzech punktach funkcją kwadratową. Istnieją dwa rodzaje algorytmu: MM1 i MM2. W tym projekcie wykorzystywana jest metoda MM1.



Rysunek 6: Porównanie metody siecznych i Müllera MM1

Metoda polega na kolejnym wyznaczaniu współczynników paraboli przechodzącej przez trzy punkty. Następnie można te współczynniki podstawić do wzoru na miejsca zerowe funkcji kwadratowej, aby otrzymać punkt przecięcia paraboli z osią x, czyli przybliżenie poszukiwanego pierwiastka. Ten sposób jest ułatwiony poprzez zapisanie równania parabolicznego w dogodnej formie:

$$f(x) = a(x - x_2)^2 + b(x - x_2) + c \tag{4}$$

Chcemy by parabola ta przecinała trzy punkty: $[x_0, f(x_0)], [x_1, f(x_1)], i[x_2, f(x_2)]$. Otrzymujemy więc równania, które prowadzą do:

$$\begin{aligned} f(x_0) &= a(x_0 - x_2)^2 + b(x_0 - x_2) + c \\ f(x_1) &= a(x_1 - x_2)^2 + b(x_1 - x_2) + c \\ f(x_2) &= a(x_2 - x_2)^2 + b(x_2 - x_2) + c \end{aligned} \tag{5}$$

Z ostatniego równania mamy $f(x_2) = c$. Aby wyznaczyć współczynniki a i b, wprowadźmy wyrażenia różnicowe:

$$h_0 = x_1 - x_0$$

$$h_1 = x_2 - x_1$$

$$\delta_0 = \frac{f(x_1) - f(x_0)}{x_1 - x_0}$$

$$\delta_1 = \frac{f(x_2) - f(x_1)}{x_2 - x_1} \tag{6}$$

Możemy te równania podstawić do równań (4) i uzyskać:

$$a = \frac{\delta_1 - \delta_0}{h_1 + h_0}$$

$$b = ah_1 + \delta_1$$

$$c = f(x_2) \tag{7}$$

W efekcie mamy, korzystając ze wzorów na równanie kwadratowe:

$$x_3 = x_2 + \frac{-2c}{b \pm \sqrt{b^2 - 4ac}} \tag{8}$$

Dla kolejnego przybliżenia rozwiązania bierzemy pierwiastek położony jak najbliżej x_2, tj. o mniejszym module.

Przy wyznaczaniu kolejnych pierwiastków stosujemy opisaną w następnym punkcie deflację czynnikiem liniowym - jak gdyby usuwamy z pierwotnego wielomianu uzyskany już pierwiastek i szukamy nowych.

4.1.1 Deflacja czynnikiem liniowym

Wprowadźmy f(x) i Q(x):

$$f(x) = a_n x^n + a_{n-1}x^{n-1} + ... + a_1 x + a_0 = (x - \alpha) \cdot Q(x) \tag{9}$$

$$Q(x) = q_n x^{n-1} + ... + q_2 x + q_1 \tag{10}$$

A także α - aktualny pierwiastek wyznaczony z metody

Deflacji czynnikiem liniowym dokonuje się przy pomocy *sklejanego schematu Hornera*, tzn. szczególnie dla wielomianów wyższego rzędu:

- $q_n, q_{n-1}, ..., q_{k+1}$ wyznaczamy zgodnie z algorytmem Hornera
- $q_1, q_2, ..., q_k$ wyznaczamy zgodnie z odwrotnym algorytmem Hornera

Stosujemy sklejany schemat Hornera, ponieważ wtedy uzyskujemy mniejsze błędy numeryczne kolejnych współczynników, niż w przypadku korzystania z jednej metody.
Algorytm Hornera sprowadza się do dzielenia wielomianu f(x) przez jednomian x-α)

Prosty schemat Hornera:

$$q_{n+1} = 0$$

$$q_i = a_i + q_{i+1}\alpha \quad i = n, n-1, ..., 0 \tag{11}$$

Odwrotny schemat Hornera:

$$q_0 = 0$$

$$q_{i+1} = \frac{q_i - a_i}{\alpha} \quad i = 0, 1, 2, ..., n-1 \tag{12}$$

4.2 Wyniki

	Znalezione pierwiastki x	**f(x)**	**Liczba iteracji**
1	0.2504+0.3461i	-1.998264e-10	7
2	0.2504 - 0.3461i	6.206588e-09	6
3	1.7433	8.881784e-16	1
4	-3.1440	-4.440892e-16	1

Tabela 4: Znalezione pierwiastki wielomianu

Z załączonej tabeli wynika, że algorytm znajduje dużo szybciej pierwiastki rzeczywiste niż zespolone. Ze względu na swoją technikę interpolacji kwadratowej metoda Müllera jest często przydatna i skuteczna w przypadku równań zespolonych.

5 Listingi funkcji

5.1 ZAD1.m

```
range = 2:0.001:11;

figure(1)
plot(range, arrayfun(@Function, range))
hold on
Output = RangeMaker(range)
yline(0,'-');

%Secant method
for i=Output
    a = i{1}(1);
    b = i{1}(2);
    X = sprintf('Secant: Function(a) is %f', Function(a));
    disp(X)
    Y = sprintf('Secant: Function(b) is %f', Function(b));
    disp(Y)
    [x, iter] = secant(@Function, a, b, 10^-8)
    W = sprintf('Secant: Function value of root: %d', Function(x));
    disp(W)
    Z = sprintf('Secant: Number of iterations: %d', iter);
    disp(Z)
    scatter(x, Function(x), 'r')
end
hold off

figure(2)
plot(range, arrayfun(@Function, range))
hold on
Output = RangeMaker(range);
yline(0, '-');
%Newton method
for i=Output
    a = i{1}(1);
    b = i{1}(2);
    pp = ceil((a+b)/2)
    Y = sprintf('Newton: Function(pp) is %f', Function(pp));
    disp(Y)
    [xf, ff, iexe, texe] = newton(@Function, (a+b)/2, 10^-8, 100);
    xf
    W = sprintf('Secant: Function value of root: %d', Function(xf));
    disp(W)
    Z = sprintf('Newton: Number of iterations: %d', iter);
    disp(Z)
    scatter(pp, Function(pp), 'black')
    scatter(xf, Function(xf), 'r')
end
```

```
hold off
```

5.2 secant.m

```
function [x2, iter] = secant(f, x0, x1, eps)
    % f - funkcja, ktorej pierwiastek znajdujemy
    % x1 - pocz tek zakresu izolacji
    % x2 - koniec zakresu izolacji
    % eps - dopuszczany b   d
    % x2 - pierwiastek o b   dzie co najwy ej eps
    % iter - wykonana liczba iteracji
    i=0;
    while 1
        i=i+1;
        x2 = x1 - f(x1)*(x1-x0)/(f(x1)-f(x0));
        x0 = x1;
        x1 = x2;
        if abs(f(x2))<=eps
            break
        end
    end
    iter=i;
end
```

5.3 newton.m

```
function [xf, ff, iexe, texe] = newton(f, x0, delta, imax)
%
%   CEL
%       Poszukiwanie pierwiastka funkcji jednej zmiennej
%       metoda Newtona (stycznych)
%
%   PARAMETRY WEJSCIOWE
%       f       -   funkcja dana jako wyrazenie
%       x0      -   punkt poczatkowy
%       delta   -   dokladnosc
%       imax    -   maksymalna liczba iteracji
%
%   PARAMETRY WYJSCIOWE
%       xf      -   rozwiazanie
%       ff      -   wartosc funkcji w xf
%       iexe    -   liczba iteracji wykonanych
%       texe    -   czas obliczen [s]
%
%   PRZYKLADOWE WYWOLANIE
%       >> [xf, ff, iexe, texe] = newton(@(x) sin(x), 2, 1e-8, 100)
%
%   AUTOR, PROJEKT
%       Andrzej Karbowski,
%       Projekt z "Metod numerycznych" MNUM,
%       WEiTI PW, sem. 22Z
```

```
%
syms X
% obliczenie pochodnej reprezentowanej jako funkcja anonimowa
df = matlabFunction(diff(f(X), X));
tic;
i = 0; x=x0; fx=feval(f,x);
while abs(fx) > delta && i < imax
     i = i + 1; xpop = x;
     % iteracyjne obliczanie nowego przyblizenia pierwiastka
     x = x - fx/df(x);
     fx=feval(f,x);
end
texe=toc; iexe=i;
xf=x; ff = fx;
```

5.4 Function.m

```
function f = Function(x)
    f = 0.7.*x.*cos(x)-log(x+1);
end
```

5.5 RangeMaker.m

```
function Ranges = RangeMaker(range)
    % range - pocz tkowy przedzia
    % Ranges - array cell z poszczeg lnymi przedzia ami izolacji
    Ranges = {};
    Function= @(x) 0.7*x.*cos(x)-log(x+1);
    G=groupConsec(groupTrue(Function(range)>0)-groupTrue(Function(range)<0))
        ;
    [starts,stops]=groupLims(G,1)
    endpoints=sort(round((starts+stops)/2));
    xline(range(endpoints),'--');
    for i=1:length(endpoints)-1
        Ranges{end+1} = [range(endpoints(i)), range(endpoints(i+1))];
    end
end
```

5.6 horner.m

```
function q = horner(a, alpha)
    % a - wsp  czynniki wielomianu poddawanego deflacji czynnikiem
       liniowym
    % alpha - aktualnie wyznaczony pierwiastek z metody
    % q - wsp  czynniki wielomianu poddanego deflacji czynnikiem liniowym
    q = a;
    q = [q, 0];
    q(1) = 0;
    for i=2:ceil(length(q)/2)
        q(i)=a(i-1) + q(i-1)*alpha;
    end
    for i=length(q)-1:-1:ceil(length(q)/2)+1
```

```
12          q(i)=(q(i+1)-a(i))/alpha;
13      end
14      q = q(2:end-1);
15  end
```

5.7 MM1.m

```
1   function [x3, iter] = MM1(p, x0, x1, x2, eps)
2       % p - wsp czynniki wielomianu
3       % x0, x1, x2 - punkty pocz tkowe algorytmu
4       % eps - dopuszczany b d
5       % x3 - pierwiastek o b dzie co najwy ej eps
6       % iter - wykonana liczba iteracji
7       i=0;
8       while 1
9           i=i+1;
10          h0 = x1-x0;
11          h1 = x2-x1;
12          d0 = (polyval(p, x1)-polyval(p, x0))/h0;
13          d1 = (polyval(p, x2)-polyval(p, x1))/h1;
14          a = (d1-d0)/(h1+h0);
15          b = a*h1+d1;
16          c = polyval(p, x2);
17          rad = sqrt(b*b-4*a*c);
18          if abs(b+rad)>abs(b-rad)
19              denom = b+rad;
20          else
21              denom = b-rad;
22          end
23          dx3 = -2*c/denom;
24          x3 = x2+dx3;
25          if abs(polyval(p, x3))<eps
26              break
27          end
28          x0 = x1;
29          x1 = x2;
30          x2 = x3;
31      end
32      iter=i;
33  end
```

5.8 ZAD2.m

```
1   x0 = -4;
2   x1 = 0;
3   x2 = 4;
4   range=-4:0.01:4;
5   coef = [1, 0.9, -6, 3, -1];
6
7   % plot(range, polyval(coef, range))
8   % yline(0, '-');
```

```
9
10 while (length(coef)>1)
11     [x, iter] = MM1(coef, x0, x1, x2, 10^-8)
12     W = sprintf('Muller: Function value of root: %d', polyval(coef, x));
13     disp(W)
14     coef = horner(coef, x)
15 end
```