Feliks Bańka, 323733

MNUM – Projekt 3

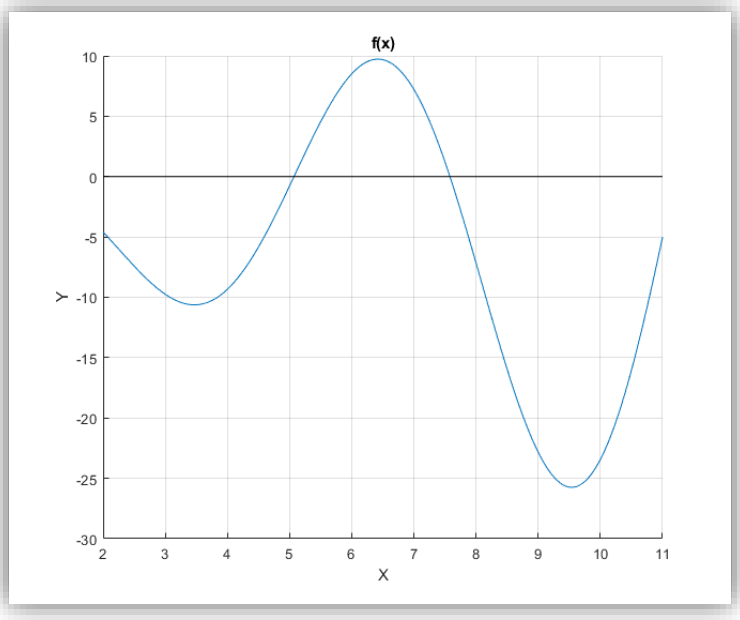
Zadanie 1

1. Proszę znaleźć wszystkie pierwiastki funkcji

$$f(x) = 2.2x\cos(x) - 2\ln(x+2) \quad \text{w przedziale} \quad [2, 11]$$

używając dla każdego zera:
a) własnego solwera z implementacją metody **bisekcji**.
b) podanego na stronie przedmiotu solwera `newton.m` z implementacja metody Newtona.

Obie wymienione metody są metodami iteracyjnymi, co oznacza, iż z każdą kolejną iteracją nasz x (rozumiany jako pierwiastek) jest coraz bliżej faktycznego rozwiązania. Zanim wykonamy kolejne iteracje, musimy wyznaczyć (oszacować) przedział, w którym będzie się znajdować pierwiastek równania (przedział izolacji pierwiastka). Najłatwiej wykorzystać jest do tego program, który rysuje przybliżony wykres funkcji i odczytać z niego przedziały.



Z powyższego wykresu (wygenerowanego w programie MATLAB) możemy oszacować, iż pierwiastki znajdują się w okolicach $x = 5$ oraz $x = 7{,}5$, a w związku z tym możemy podzielić

nasz przedział $[2; 11]$ na dwa mniejsze przedziały np. $[2; 6{,}5]$ i $[6{,}5; 11]$, które są naszymi przedziałami izolacji pierwiastka.
Na skutek każdej kolejnej iteracji wartość $f(x)$ zbliża się coraz bardziej do 0.
$\lim_{n\to\infty} f(x_n) = 0$, gdzie n to liczba iteracji.

Metoda bisekcji

W tej metodzie nasz początkowo oszacowany przedział oznaczamy jako przedział dla iteracji (*n*=0) $[a, b] = [a_0, b_0]$.

W każdej iteracji (*n*-tej) dzielimy nasz przedział na dwa mniejsze (równe):

$[a_n, c_n],\ [b_n, c_n]$

Dla $c_n = \frac{a_n + b_n}{2}$

A następnie wyznaczamy nowy przedział $[a_{n+1}, b_{n+1}]$, równy temu przedziałowi $f(a_n) \times f(c_n)$ lub $f(b_n) \times f(c_n)$, którego iloczyn jest ujemny (co oznacza, że właśnie w tym przedziale znajduje się pierwiastek równania).

Tą procedurę powtarzamy, aż do momentu, gdy $f(c_n) \leq \delta$, gdzie δ to wybrana przez nas dokładność rozwiązania (w naszym przypadku jest to $1e-8$).

Niestety metoda ta wymaga, od nas również sprawdzenia czy długość przedziału $|b_n - a_n|$ nie jest zbyt mała, gdyż wtedy pochodna w otoczeniu zera funkcji może być z mała, co będzie prowadziło do nieprecyzyjnego wyniku.

Natomiast zaletą tej metodę jest fakt, iż jest ona zbieżna liniowo ($p = 1$), co oznacza, iż znajdzie ona przybliżoną wartość pierwiastka niezależnie od początkowych wartości na końcach przedziałów (choćby $|b_n - a_n|$ była bardzo duża) – zbieżność tej metody jest globalna. Jedyny warunek jaki musi zostać spełniony jest ciągłość funkcji (oraz oczywiście istnienie pierwiastków).

Implementacja w Matlab'ie

```
function [c, fc, i] = bisection(f, a, b, delta)
%...

    if nargin < 4
        delta = 1e-8;
    end

    fc=inf; % to run while loop
    i=0;
    while (abs(fc) > delta && abs(b-a)>1e-8)
        i=i+1;

        c=(a+b)/2; % gets points c between a and b
        fc=f(c);
        fa=f(a);

        if (fa*fc<0) % if root is between a and c
            b=c;
        else
            a=c;
        end
    end
end
```

Metoda Newtona

Metoda Newtona do wyznaczenia pierwiastków równania wykorzystuje rozwiniecie w szereg Taylora. Na początku musimy wyznaczyć aktualne przybliżenie pierwiastka $x_n = x_0$, gdzie x_0 jest oszacowaną przez nas wartością (np. z wykresu).

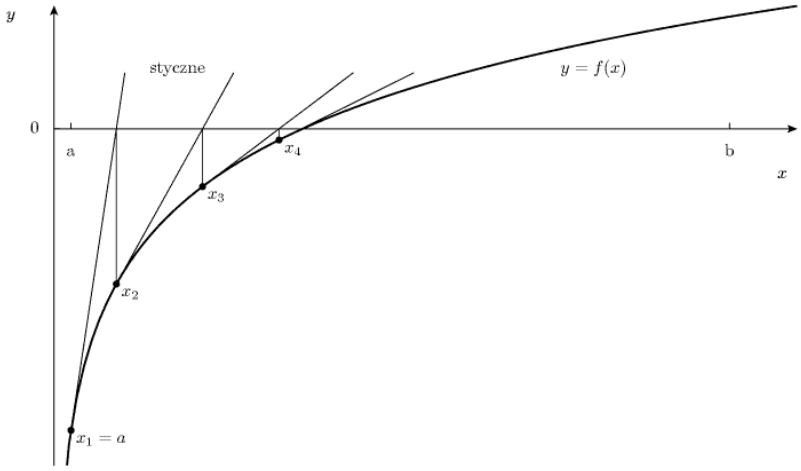
Następnie po zapisaniu funkcji jako

$$f(x) \approx f(x_n) + f'(x_n)(x - x_n)$$

Oraz przyrównaniu jej od zera możemy wyznaczyć wartość x_{n+1}. Wartość ta będzie wynosić:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Funkcja ta jest tym dokładniejsza im bardziej stroma jest funkcja o otoczeniu pierwiastka, przy czym nie zalecane jest jej używanie, kiedy w pobliżu pierwiastka jest ona prawie pozioma (bo wtedy wartość $f'(x)$ jest bardzo mała).



https://pl.wikipedia.org/wiki/Metoda_Newtona#/media/Plik:Metoda_Newtona-4_kroki.png

Wyniki

Poniżej prezentują się liczby iteracji potrzebnych do uzyskania precyzji rzędu $1e-8$, dla poszczególnych metod w zależności od: pp – punktu początkowego (lub $x0$ dla metody Newtona), f(pp) – wartości funkcji w pp, pk – punktu końcowego, (pk) – wartości funkcji w pk.

Pierwszy pierwiastek (z przedziału $[2; 6{,}5]$)

Dane				Liczba iteracji	
pp ($x0$ dla Newtona)	f(pp)	pk	f(pk)	Bisekcja	Newton
2	-4.6036	6.5	9.6851	29	11
4	-9.3356	6	8.5154	30	5
4.5	-5.8305	5.5	4.5451	29	3
4.9	-1.8524	5.2	1.4117	27	3
5	-0.7715	5.1	0.3207	26	2

Metoda bisekcji

Wyniki pierwszych 10 iteracji dla zawężonego przedziału $[4; 6]$

Nr. Iteracji	Przybliżony argument x	Wartość funkcji f(x)
1	5	-0.77154
2	5.5	4.545098
3	5.25	1.952584
4	5.125	0.594085
5	5.0625	-0.08941
6	5.09375	0.252359
7	5.078125	0.081455
8	5.070313	-0.00399
9	5.074219	0.038733
10	5.072266	0.017373

Metoda Newtona

Wyniki iteracji dla punktu początkowego $x0 = 4$

Nr. Iteracji	Przybliżony argument x	Wartość funkcji f(x)
1	5.9097	7.9689
2	4.6911	-4.0215
3	5.0943	0.2589
4	5.0707	1.30E-05
5	5.0707	1.79E-13

Drugi pierwiastek (z przedziału $[6{,}5; 11]$)

Dane				Liczba iteracji	
pp ($x0$ dla Newtona)	f(pp)	pk	f(pk)	Bisekcja	Newton
6.5[1]	9.6852	11	-5.0228	32	4
6.5[1]	9.6852	8.5	-15.9604	30	4
7	7.2156	8	-7.1659	29	4
7.4	2.6581	7.7	-1.9460	23	3
7.5	1.2169	7.6	-0.3225	23	3

[1] - Metoda Newtona w dwóch pierwszych przedziałach znajdowała inny pierwiastek (spoza naszego zakresu), dlatego w przedziale zamiast punktu 6.5 został wykorzystany punkt 8.5.

Metoda bisekcji

Wyniki pierwszych 10 iteracji dla zawężonego przedziału $[7; 8]$

Nr. Iteracji	Przybliżony argument x	Wartość funkcji f(x)
1	7.5000	1.2169
2	7.7500	-2.7848
3	7.6250	-0.7210
4	7.5625	0.2654
5	7.5938	-0.2236
6	7.5781	0.0220
7	7.5859	-0.1006
8	7.5820	-0.0392
9	7.5801	-0.0086
10	7.5791	0.0067

Metoda Newtona

Wyniki iteracji dla punktu początkowego $x0 = 7$

Nr. Iteracji	Przybliżony argument x	Wartość funkcji f(x)
1	7.8312	-4.1783
2	7.5907	-0.1761
3	7.5796	-0.00054
4	7.5795	-5.16E-09

Wnioski

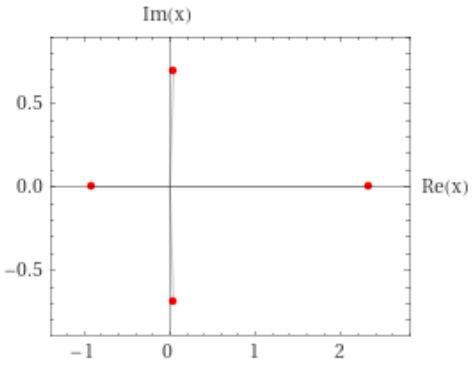
W obu przypadkach metoda bisekcji potrzebowała około 30 iteracji, podczas gdy metodzie Newtona wystarczyło za każdym razem ≤ 5. Jest to bardzo duża wada tej metody, gdyż jest ona mało wydajna. Z drugiej strony jednak, należy pamiętać, że metoda Newtona jest zbieżna lokalnie (a nie globalnie) co prowadzi do takich sytuacji, jak w punkcie [1] gdzie znaleziony został pierwiastek z innego zakresu. Było to spowodowane faktem, iż punkt $x0 = 6.5$ znajduje się blisko ekstremum.

Zadanie 2

2. Używając metody **Laguerre'a** proszę znaleźć wszystkie pierwiastki wielomianu czwartego stopnia:

$$f(x) = a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0, \qquad [a_4\ a_3\ a_2\ a_1\ a_0] = [-2\ 3\ 3\ 1\ 2]$$

Graficzne przedstawienie pierwiastków wielomianu (z wykorzystaniem wolframalpha):



Pierwiastki wielomianu

Wiemy, że każdy wielomian n-tego stopnia ma dokładnie n pierwiastków w dziedzinie zespolonej. Pierwiastki te mogą być zarówno: rzeczywiste lub zespolone (zawsze jako para liczba-sprężenie) i jednokrotne lub wielokrotne.

Do znalezienia tych pierwiastków możemy oczywiście wykorzystać znane już algorytmy, te same co do równań nieliniowych, jednak o wiele efektywniejsze jest wykorzystanie specjalnych algorytmów – przystosowanych do znajdowania pierwiastków wielomianów, które wykorzystują specjalne własności wielomianów (np. wielokrotną różniczkowalność).

Dodatkowo w naszym algorytmie można uwzględnić fakt, iż dla każdego pierwiastka zespolonego, jego sprzężenie jest również pierwiastek wielomianu, dzięki czemu od razu otrzymujemy dwa pierwiastki w jednym wykonaniu algorytmu.

Metoda Laguerre'a

Wzór na tę metodę wygląda następująco:

$$x_{k+1} = x_k - \frac{nf(x_k)}{f'(x_k) \pm \sqrt{(n-1)\big((n-1)f'(x_k)^2 - nf(x_k)f''(x_k)\big)}}$$

Gdzie:

n – stopień wielomianu,

$\pm$ - wybieramy tak, aby mianownik był jak największy (co do modułu)

Implementacja w Matlab'ie

```
function [x, y, i] = laguerre(f, x0, delta)
%...

n = size(f,2)-1;
df = polyder(f);  %first differitial of f
d2f = polyder(df); %second differitial of f
X(1)=x0;
i=1;

while (abs(polyval(f,X(i))) > delta)
    yf=polyval(f,X(i));
    ydf=polyval(df,X(i));
    yd2f=polyval(d2f,X(i));

    G=sqrt((n-1)*((n-1)*ydf^2-n*yf*yd2f));

    D1=ydf-G;
    D2=ydf+G;
    if abs(D1) > abs(D2) % chooses bigger absolute value
        D=D1;
    else
        D=D2;
    end

    X(i+1)=X(i)-n*yf/D;
    i=i+1;
end
x=X(i);
y=polyval(f,x);
end
```

Gdzie G oraz D:

$$x_{k+1} = x_k - \frac{nf(x_k)}{D_{1,2} \pm G} = x_k - \frac{nf(x_k)}{D}$$

Deflacja czynnikiem liniowym

Aby jeszcze usprawnić nasz algorytm, możemy zastawać tzw. deflację czynnikiem liniowym. Mianowicie po znalezieniu jednego pierwiastka (α), jeszcze przed wyznaczeniem kolejnego dzielimy nasz wielomian przez czynnik $(x - \alpha)$, dzięki czemu otrzymujemy prostszy wielomian (niższego rzędu) i nie wyznaczymy już drugi raz tego samego pierwiastka.

Jeśli:

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

$$Q(x) = q_n x^{n-1} + \cdots + q_2 x + q$$

To nasz algorytm dzielenie wielomianu $f(x)$ przez jednomian $(x-\alpha)$ możemy zapisać jako:

$$q_{n+1} \stackrel{\text{def}}{=} 0$$

$$q_i = a_i + q_{i+1}\alpha$$

gdzie $i = n, n-1, \ldots, 0$ - i jest to tzw. Schemat Hornera.

Implementacja w Matlab'ie

```
1  function [Q] = horner(a,root)
2  %...
19
20      n = length(a)-1; % degree of Q(x)
21      q=zeros(1,n-1);
22      q(1) = a(1);
23      for i = 2:n
24          q(i) =  a(i) + q(i-1)*root;
25      end
26      Q=q;
27  end
```

Wyniki

Bez deflacji czynnikiem liniowym:

Liczba iteracji	x (pierwiastek)	f(x)
3	-0.9035	-5.30E-12
3	2.3197	1.06E-10
5	0.0419 + 0.6895i	0
5	0.0419 - 0.6895i	0

Z deflacją:

Liczba iteracji	x (pierwiastek)	f(x)
3	-0.9035	-5.30E-12
3	2.3197	6.60E-11
2	0.0419 - 0.6895i	0
2	0.0419 + 0.6895i	0

Wnioski

Jak widać algorytm Laguerre'a ten jest bardzo wydajny, gdyż potrzebne było łącznie tylko 16 iteracji, aby znaleźć wszystkie pierwiastki, jednakże, jeśli wykorzystamy jeszcze deflację czynnikiem liniowym to liczba ta spada do zaledwie 10 iteracji. Co więcej niewykorzystanie schematu Hornera wymaga od nas odpowiedniego dobrania wartości $x0$, tak aby wszystkie pierwiastki były unikalne (bez powtórzeń), a korzystając z deflacji możemy podać dowolne punkty, które będą miały wpływ jedynie na liczbę iteracji (i to w niewielkim stopniu, bo przy podaniu np. $x0 = 1000$ nasz algorytm potrzebujemy łącznie jedynie 16 iteracji).