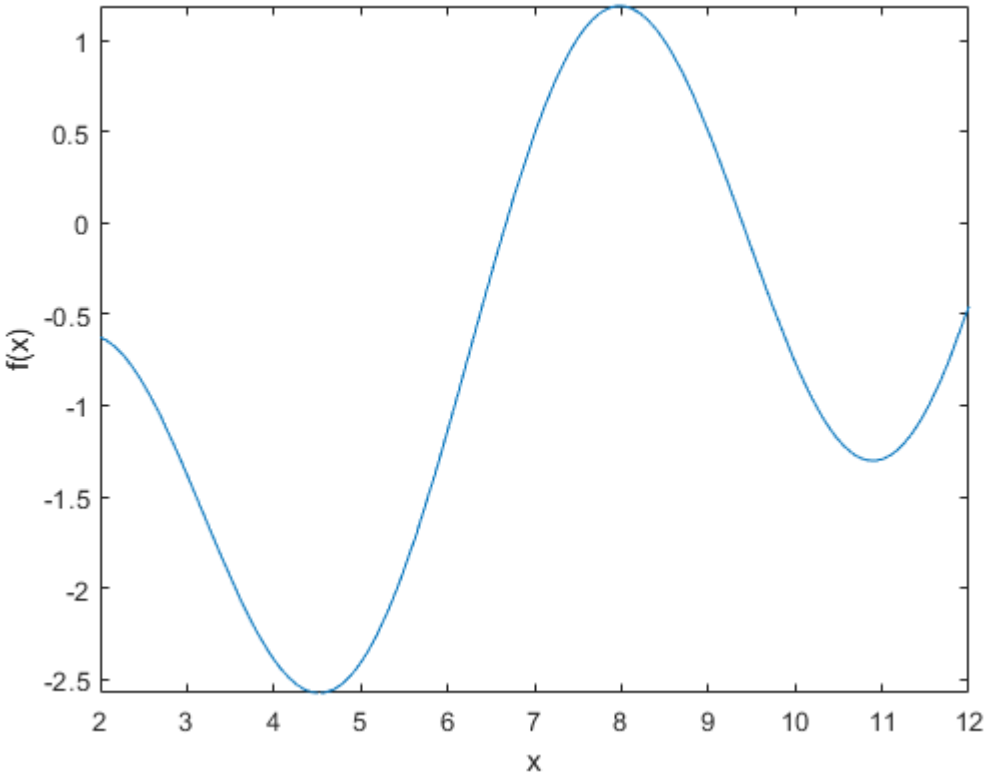
Mariusz Paluch

Sprawozdanie projekt nr 2 MNUM

1.1. **Badana funkcja**

$$f(x)=1.5\sin(x)+2\ln(x+2.5)-5 \; w \; przedziale [2,12]$$

Wykres badanej funkcji:



1.2. **Znajdowanie pierwiastków przy użyciu metody *regula falsi***

1.2.1. **Opis metody *regula falsi***

Metoda regula falsi, zwana także metodą false position, polega na tym, że aktualny przedział izolacji pierwiastka dzielony jest na dwa (najczęściej) nierówne podprzedziały, sieczną łączącą na płaszczyźnie punkty $(f(a_n), a_n)$ i $(f(b_n), b_n)$, przecinając oś rzędnych w punkcie oznaczonym jako c_n.

Wzór na c_n wygląda następująco:

$$c_n = \frac{a_n f(b_n) - b_n f(a_n)}{f(b_n) - f(a_n)}$$

Zmiana przedziału izolacji pierwiastka odbywa się poprzez zamianę wartości krańcowej przedziału na c_n. Gdy c_n jest ujemne to dolna część przedziału jest podmieniana, w przeciwnym wypadku to górna część jest podmieniana.

W sytuacji gdy c_n wychodzi poza początkowy przedział to c_n arbitralnie jest ustawiany jako środek przedziału [a,b]. Zapobiega to wychodzeniu poza poszukiwany zakres, co jest szczególnie ważne dla badanej funkcji, ponieważ nie posiada ona w przestrzeni liczb rzeczywistych wartości poniżej 0.

Metoda ta jest zawsze zbieżna, ale może być też wolno zbieżna, gdy jeden z końców izolacji pierwiastka pozostaje stały i metoda nie prowadzi do zmniejszania do zera przedziału izolacji pierwiastka. W takich sytuacjach należy stosować *zmodyfikowaną metodę regula falsi*. Podczas tego projektu stosuję jednak podstawową wersję.

1.2.2. **Kod algorytmu (falsePosition.m)**

```
function [xf, ff, iexe, texe] = falsePosition(f, a_start, b_start, delta, imax)
%
%   CEL
%      Poszukiwanie pierwiastka funkcji jednej zmiennej
%      metoda regula falsi
%
%   PARAMETRY WEJSCIOWE
%      f       -  funkcja dana jako wyrazenie
%      a_start -  wstępny początek przedziału
%      b_start -  wstępny koniec przedziału
%      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] = falsePosition(@ (x) sin(x), 2, 12, 1e-8, 100)

tic;
a = a_start; b = b_start; c = b_start;
fc = f(c);
```

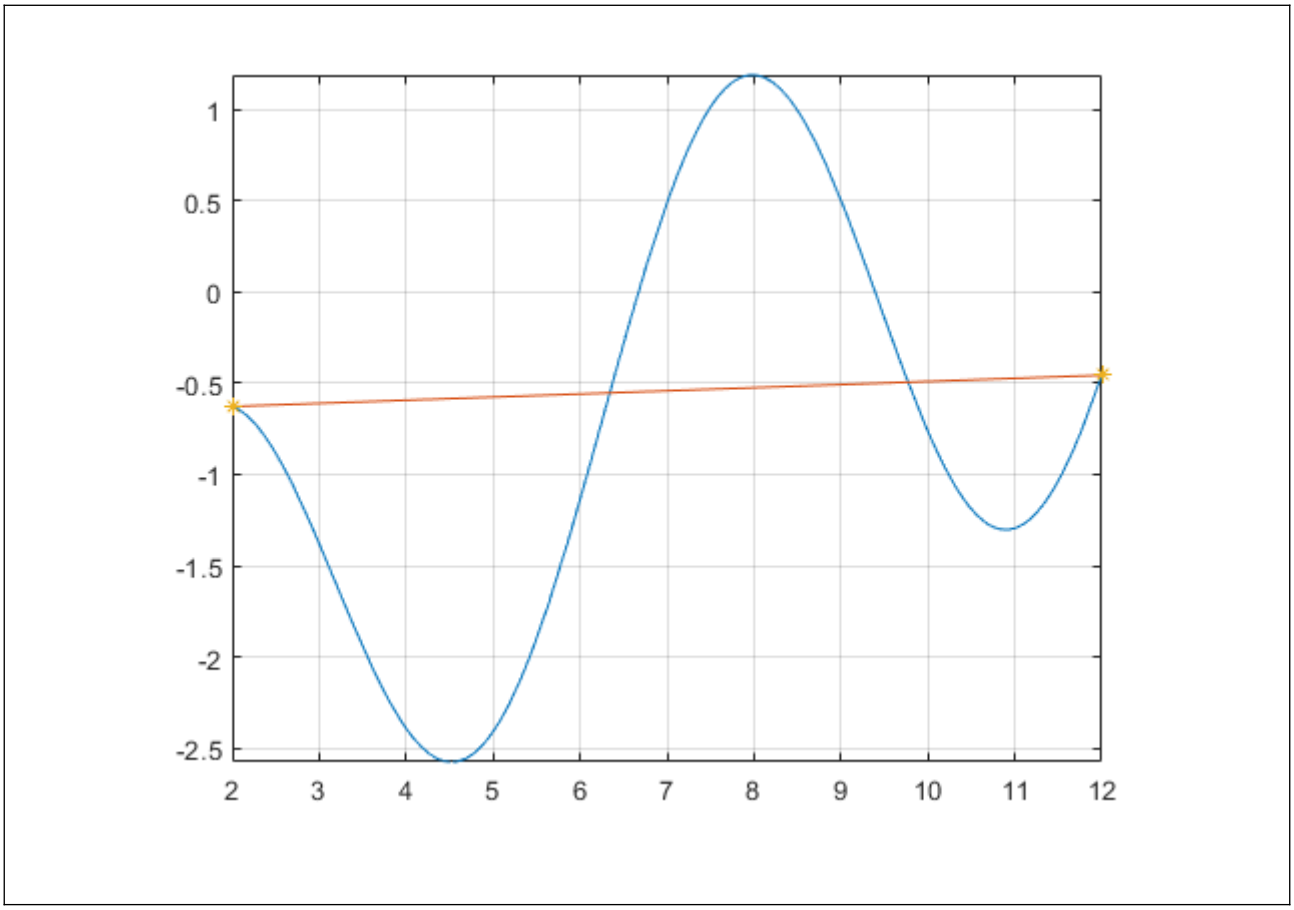
```
i = 0;
while abs(fc) > delta && i < imax
    %kod do wizualizacji działania algorytmu
    %test = [a b];
    %fplot(@(x) f(x), [2 12]);
    %hold on
    %grid on;
    %plot(test, arrayfun(f, test));
    %plot(test, arrayfun(f, test), '*');
    %input(i + ". next", 's')
    %hold off
    %clf;
    i = i + 1;
    fa = f(a); fb = f(b);
    c = (a*fb - b*fa) / (fb - fa);
    if c > b_start || c < a_start
       c = (a + b) / 2;
    end
    fc = f(c);
    if fc*fb > 0
      b = c;
    else
      a = c;
    end
end
texe=toc; iexe=i;
xf=c; ff = fc;
```

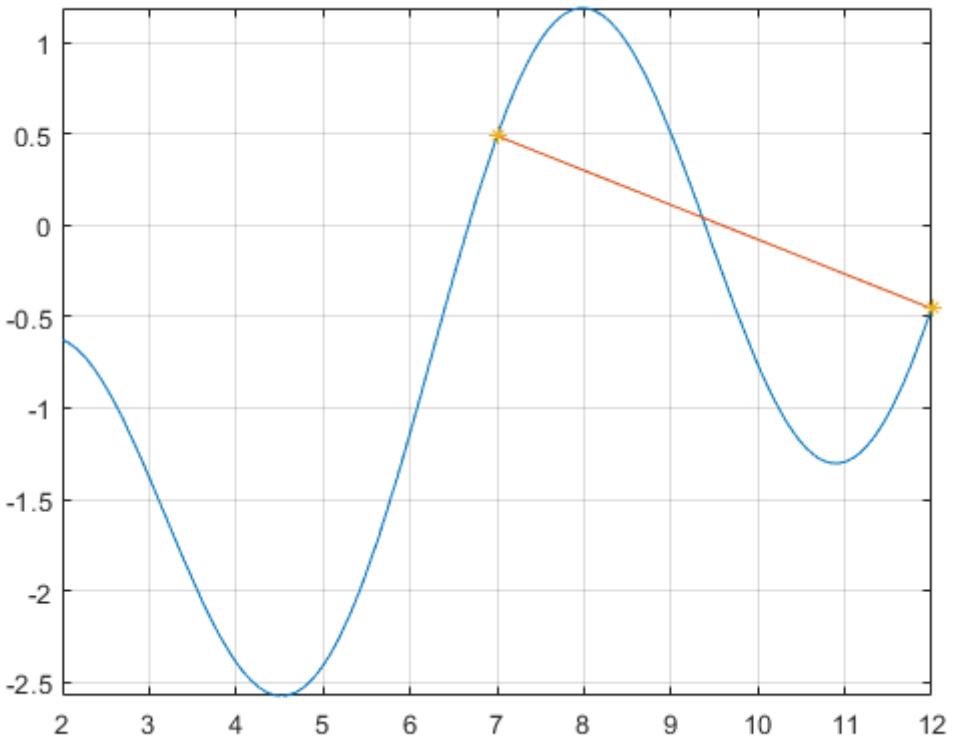
1.2.3. **Kod solvera (rootsFinder.m)**

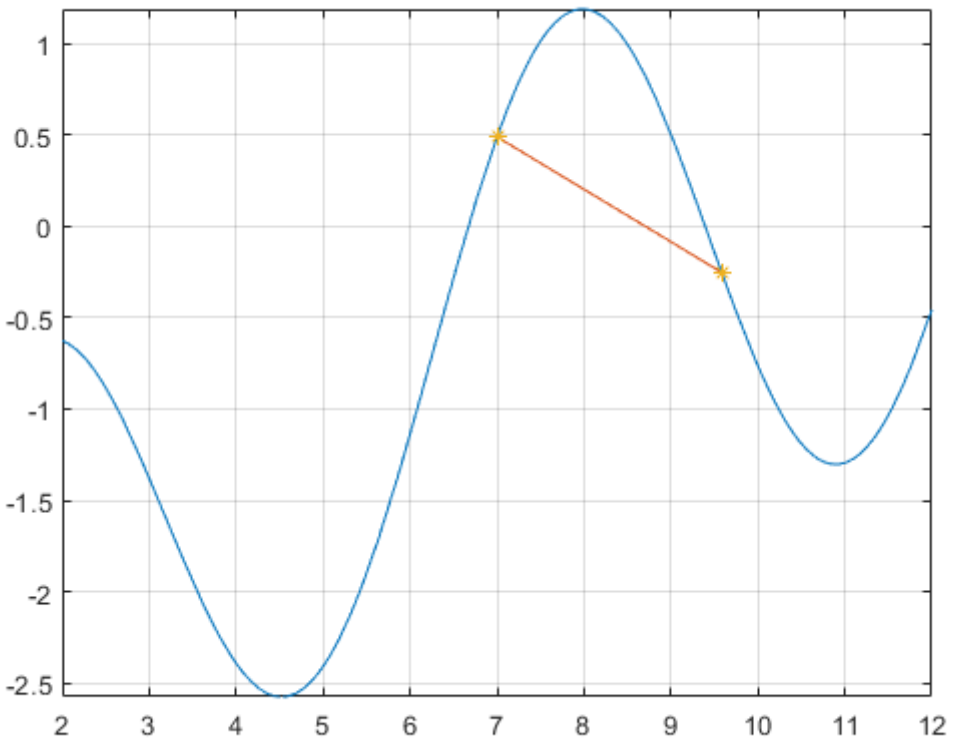
```
function [xs,ys,iexes,texes, starts, ends] = rootsFinder(f, root_method, ...
          x_start, x_end, roots_numb, delta, imax, epsilon)

xs = zeros(roots_numb, 1) + a_start - 1;
ys = zeros(roots_numb, 1);
starts = zeros(roots_numb, 1);
ends = zeros(roots_numb, 1);
iexes = zeros(roots_numb, 1);
texes = zeros(roots_numb, 1);
i = 1;
for a=x_start:epsilon:x_end
   for b=x_end:-epsilon:a
      if i > roots_numb
         break
```

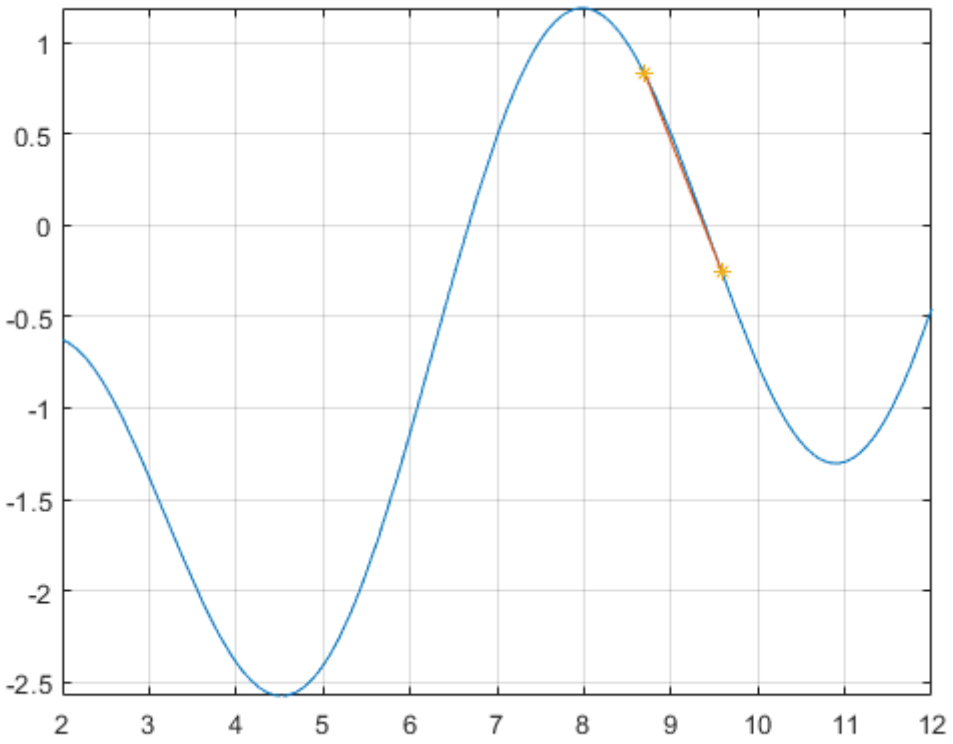
```
        end
        [x,y,iexe,texe] = root_method(f, a, b, delta, imax);
        if abs(y) < delta && x <= x_end && x >= x_start && min(abs(xs - x)) > 1e-4
            xs(i) = x;
            ys(i) = y;
            starts(i) = a;
            ends(i) = b;
            iexes(i) = iexe;
            texes(i) = texe;
            i = i + 1;
        end
    end
end
```

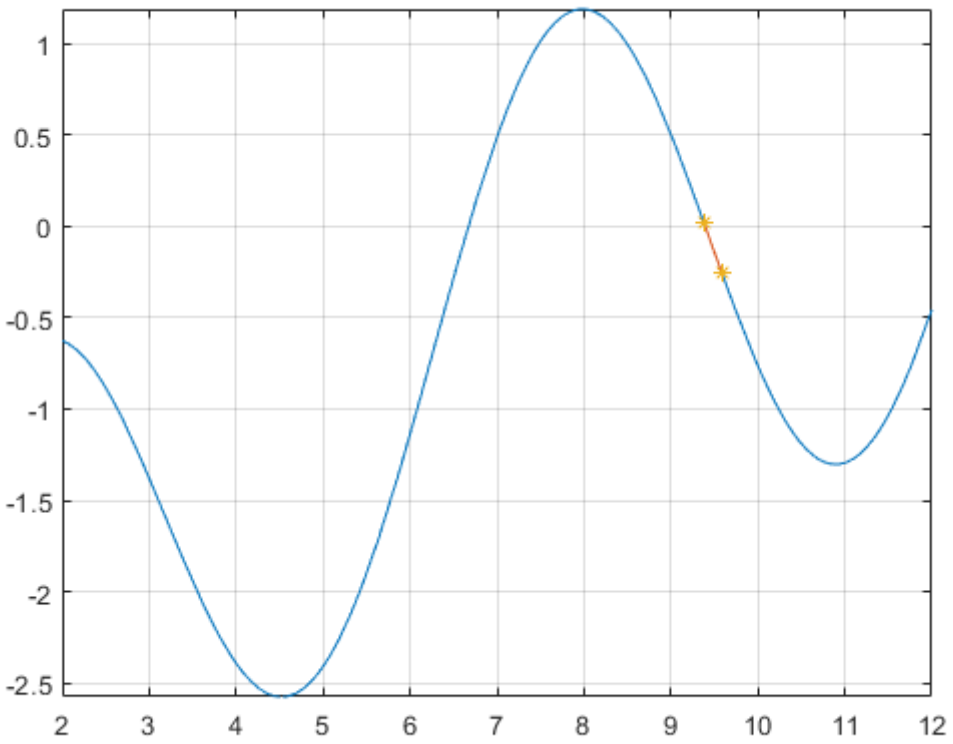
1.2.4. **Wizualizacja działania metody**

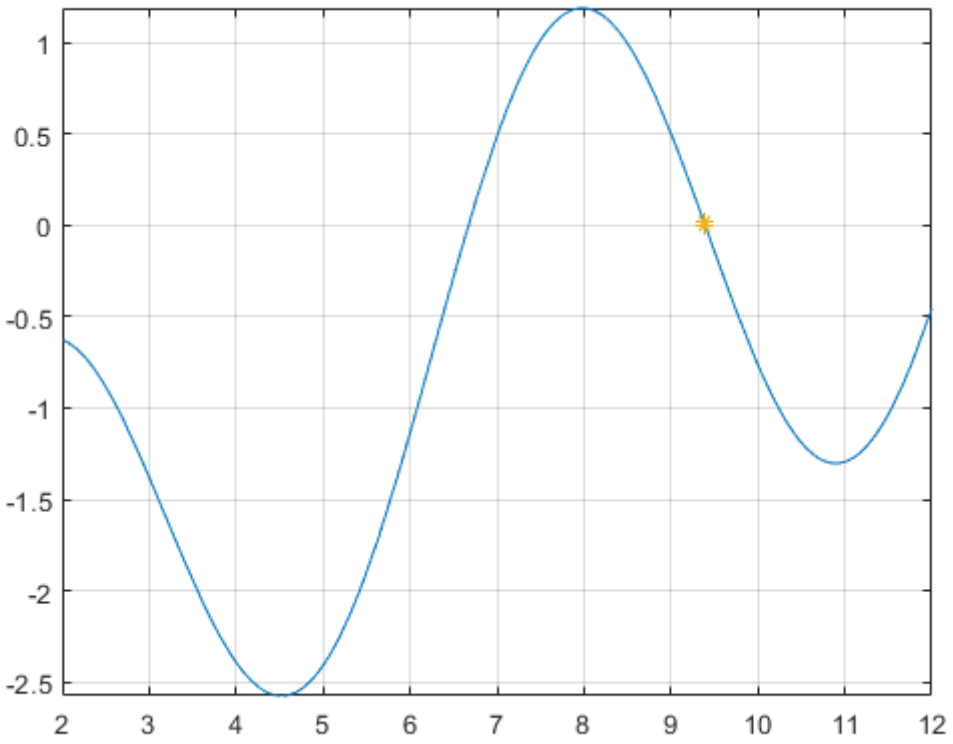


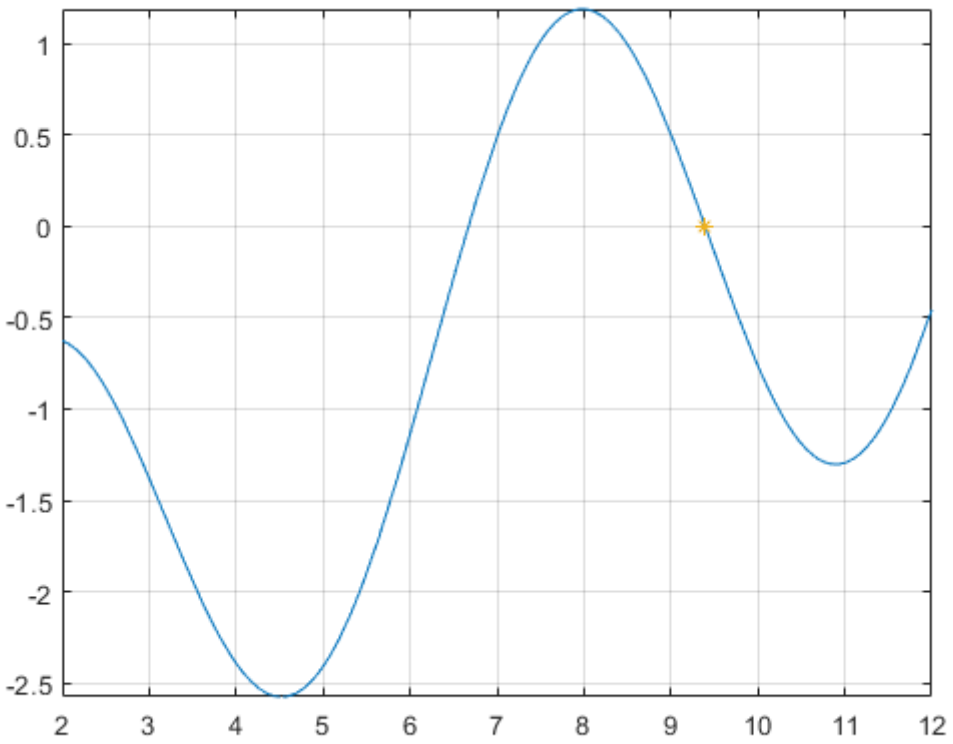

1
0.5
0
-0.5
-1
-1.5
-2
-2.5
2
3
4
5
6
7
8
9
10
11
12


1
0.5
0
-0.5
-1
-1.5
-2
-2.5
2
3
4
5
6
7
8
9
10
11
12

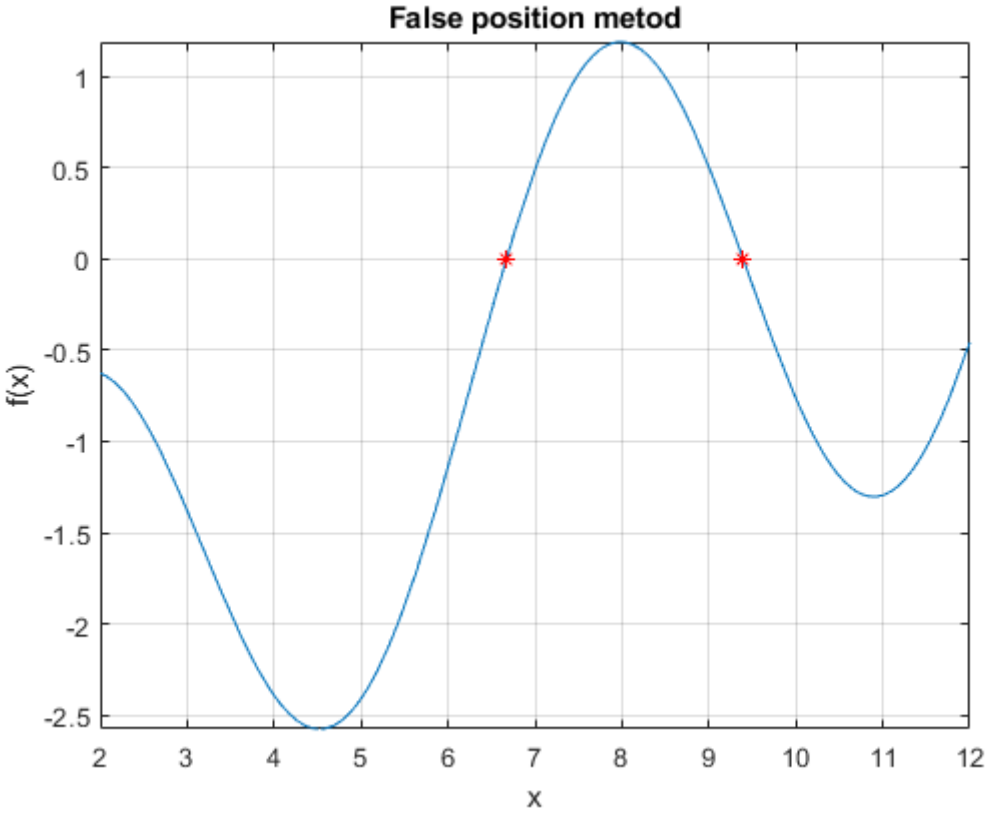

1
0.5
0
-0.5
-1
-1.5
-2
-2.5
2
3
4
5
6
7
8
9
10
11
12


1
0.5
0
-0.5
-1
-1.5
-2
-2.5
2
3
4
5
6
7
8
9
10
11
12


1
0.5
0
-0.5
-1
-1.5
-2
-2.5
2
3
4
5
6
7
8
9
10
11
12


1
0.5
0
-0.5
-1
-1.5
-2
-2.5
2
3
4
5
6
7
8
9
10
11
12

1.2.5. **Wyniki działania algorytmu**



Algorytm znalazł miejsca zerowe dla współrzędnej x równej 6.67 oraz 9.39.

Dla pierwszego wyniku algorytm znajduje wynik dla wstępnego przedziału [2,12], co zajmuje mu 7 iteracji w czasie 0.004614 s.

Dla drugiego rozwiązania początkowy przedział [2,9], co zajmuje mu 10 iteracji w czasie 0.000057s.

Dla pewnych przedziałów algorytm nie działa (np. [2, 10]), ponieważ zaczyna szukać rozwiązań wybiegających poza badanych zakres, co powoduje że algorytm zaczyna się blokować.

1.3. **Znajdowanie pierwiastków przy użyciu metody *Newtona***

1.3.1. **Opis metody *Newtona***

Metoda Newtona (stycznych), zakłada aproksymacje funkcji jej liniowym przybliżeniem wynikającym z uciętego rozwinięcia w szereg Taylora w aktualnym przybliżeniu pierwiastka

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

Następnie porównujemy prawą stronę przybliżenia do zera:

$$f(x_n)+f'(x_n)(x-x_n)=0$$

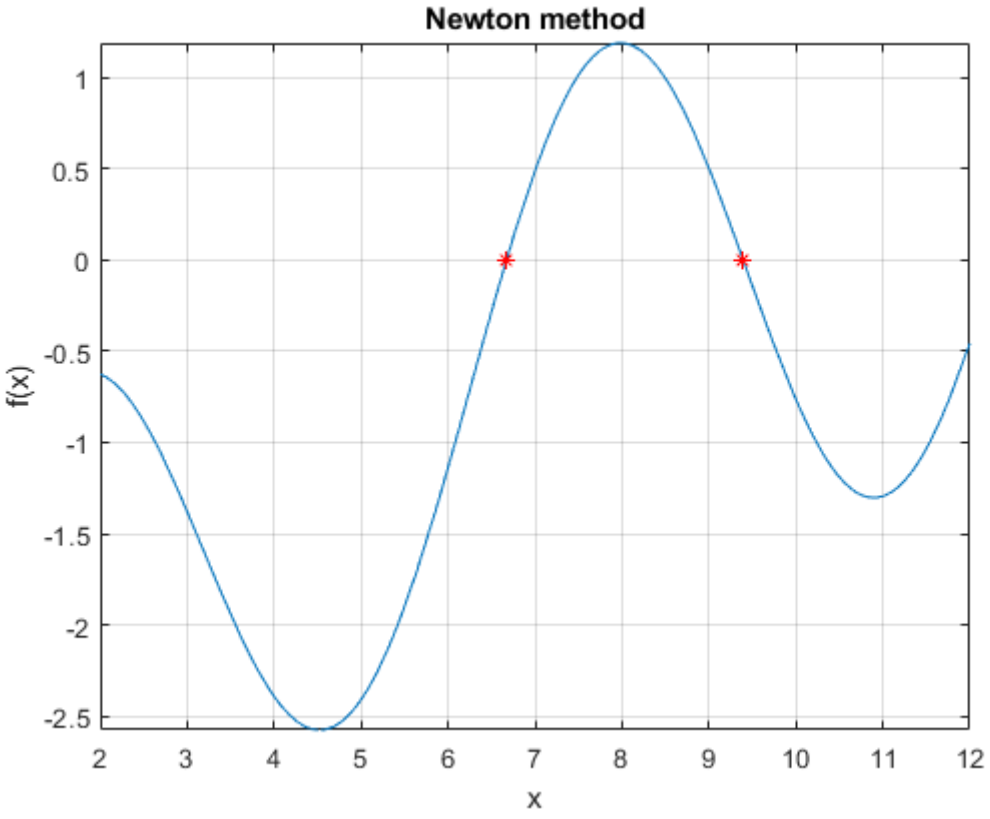
Po prostych przekształceniach otrzymujemy zależność iteracyjną

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

Metoda Newtona jest lokalnie zbieżna tzn. jeśli punkt początkowy będzie zbyt oddalony od pierwiastka to metoda może nie znaleźć rozwiązania.

Omawiana metoda dobrze sprawuje się dla funkcji, która w otoczeniu pierwiastka jest bardzo stroma, natomiast słabo sprawuje się w przypadku gdy funkcja jest prawie pozioma w okolicy pierwiastka (tzn. pochodna z funkcji ma w tym miejscu jest bardzo mała).

1.3.2. **Wyniki działania algorytmu**



Algorytm znalazł miejsca zerowe dla współrzędnej x równej 6.67 oraz 9.39.

Dla pierwszego wyniku algorytm znajduje wynik dla punktu początkowego x0=2.00 co zajmuje mu 8 iteracji oraz czas 0.000015s.

Dla drugiego rezultatu znaleziono wynik dla punktu początkowego x0=2.50 co zajmuje mu 10 iteracji oraz czas 0.000018s.

1.4. **Wnioski**

W obu przypadkach udało się uzyskać satysfakcjonujące wyniki. Metoda Newtona okazała się być wielokrotnie szybsza od metody regula falsi. Jest to najprawdopodobniej spowodowane tym, że metoda regula falsi musi wykonać więcej operacji niż metoda Newtona.

Funkcja zastosowana w zadaniu jest dobrze dostosowana do wymagań metody Newtona, ponieważ w okolicy miejsc zerowych jest ”stroma” (tzn. w tych okolicach pochodna jest duża).

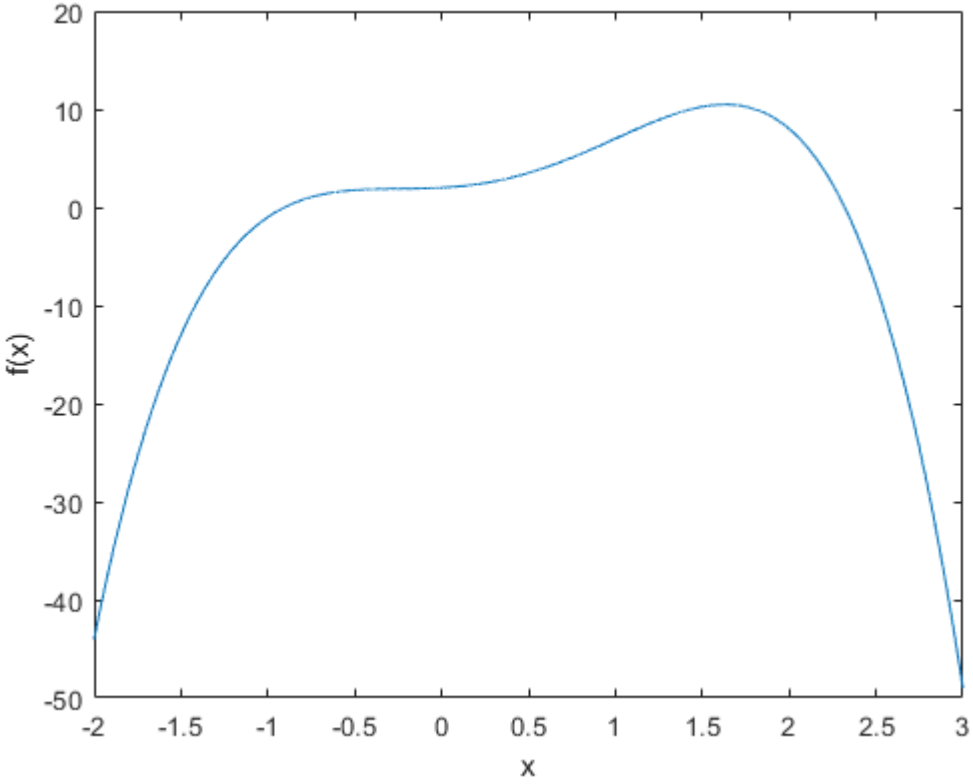
Metoda regula falsi dla zadanego zadania sprawuje się całkiem dobrze, jednak należy pamiętać, że nie musi tak być zawsze. W przypadku, gdy jeden z końców izolacji pierwiastka pozostaje stały. W takiej sytuacji będziemy mieli do czynienia z powolną zbieżnością.

Dodatkowo metoda ta może chcieć szukać rozwiązania poza dziedziną funkcji. W takiej sytuacji algorytm się zablokuje.

2.1. **Badany wielomian**

$$f(x)=-2x^4+3x^3+2x^2+x+2$$

Wykres badanego wielomianu dla $x\in[-2,3]$:



2.2. **Znajdowanie pierwiastków wielomianu przy użyciu metody *Laguerre'a***

2.2.1. **Opis metody *Laguerre'a (oraz deflacji czynnikiem liniowym)***

Metodę Laguerre'a definiuje taki oto wzór:

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

gdzie n to wymiar wielomianu, a znak w mianowniku wybieramy tak aby miał on jak największy moduł.

Omawiana metoda dla wielomianu o zerach rzeczywistych jest zbieżna z każdego rzeczywistego punktu startowego a więc jest zbieżna lokalnie.

Dla liczb zespolonych algorytm sprawuje się w zadowalający sposób, jednak mogą nastąpić przypadki niezbieżności.

Deflacja czynnikiem liniowym to metoda polegająca na dzieleniu wielomianu przez czynnik $(x-\alpha)$, przez co uzyskujemy wielomian

niższego rzędu, dzięki czemu nie wyznaczamy ponownie tego samego pierwiastka.

Jako $Q(x)$ oznaczmy wielomian uzyskany po podzieleniu $f(x)$ przez $(x-\alpha)$.

Mamy więc:

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

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

Do algorytmów dzielenie wielomianu przez jednomian $(x-\alpha)$ należą:

- Schemat Hornera (prosty):

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

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

- Odwrotny schemat Hornera

$$q_0=0$$

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

- Sklejany schemat Hornera – jest szczególnie dokładny dla wielomianów o większych rzędach.
  - Schemat działania:
    - $q_n, q_{n-1}, \ldots, q_{k+1}$ wyznaczamy zgodnie z prostym algorytmem Hornera
    - $q_1, q_2, \ldots, q_k$ wyznaczamy zgodnie z odwrotnym algorytmem Hornera

2.2.2. **Kod algorytmu *Laguerre'a (Laguerr.m)***

```
function [xk, y, iexe, texe] = Laguerr(a, x_start, delta, imax)
%
%   CEL
```

```
%       Poszukiwanie pierwiastka wielomianu
%       metoda Laguerre'a
%
%   PARAMETRY WEJSCIOWE
%       a       -  współczynniki wielomianu
%       x_start -  początkowy punkt przeszukiwań
%       delta   -  dokladnosc
%       imax    -  maksymalna liczba iteracji
%
%   PARAMETRY WYJSCIOWE
%       xk      -  rozwiazanie
%       y       -  wartosc funkcji w xk
%       iexe    -  liczba iteracji wykonanych
%       texe    -  czas obliczen [s]
%
%   PRZYKLADOWE WYWOLANIE
%       >> [xk, y, iexe, texe] = Laguerr([1,2,3,4]', 2, 1e-8, 100)

n = length(a);
xk = x_start;
der_a = polyder(a);
derSec_a = polyder(der_a);
f = @(x) (polyval(a,x));
df = @(x) (polyval(der_a,x));
dfSec = @(x) (polyval(derSec_a, x));
i = 0;
start = tic;
while abs(f(xk)) > delta && i < imax
    y = f(xk);
    derx = df(xk);
    derSecx = dfSec(xk);
    downPart = sqrt((n - 1)*((n-1)*(derx)^2) - n*y*derSecx);
    if abs(derx - downPart) > abs(derx + downPart)
        xk = xk - ((n*y) / (derx - downPart));
    else
        xk = xk - ((n*y) / (derx + downPart));
    end
    i = i + 1;
end
iexe = i;
texe = toc(start);
y = f(xk);
```

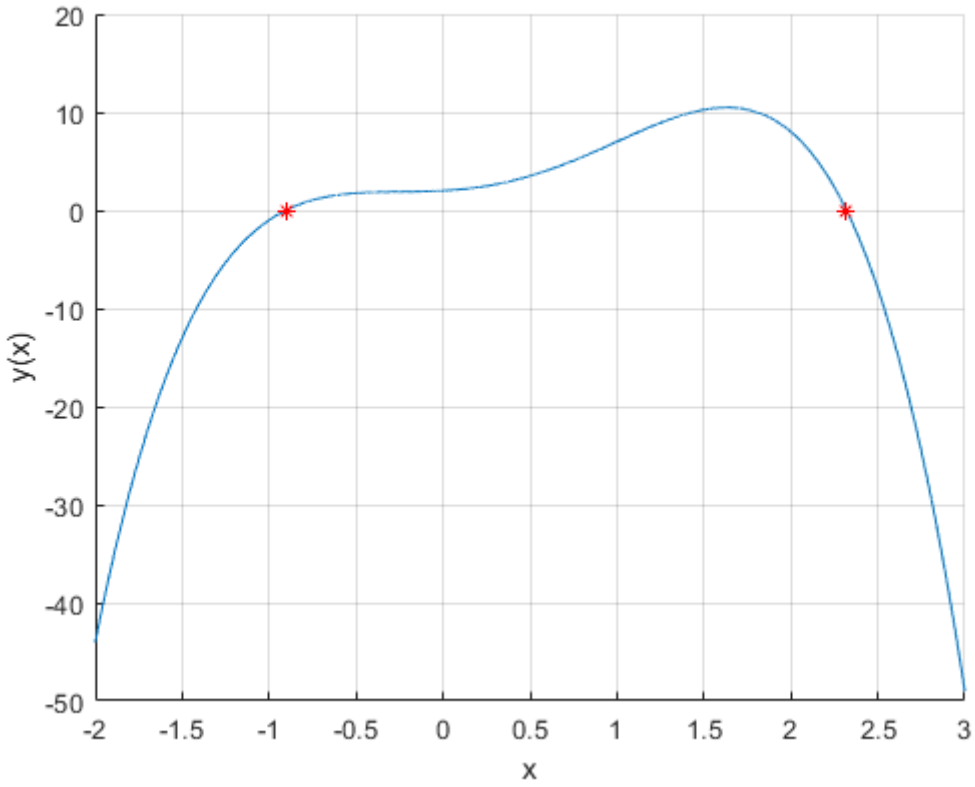
2.2.3. **Kod solvera (polyRootsFinder.m)**

```
function [xs, ys, iexes, texes] = polyRootsFinder(a, x_start, delta, max_iters)

n = length(a) - 1;
xs = zeros(n, 1);
ys = zeros(n, 1);
iexes = zeros(n,1);
texes = zeros(n, 1);
for i=1:n
    [x, y, iexe, texe] = Laguerr(a, x_start, delta, max_iters);
    xs(i) = x;
    ys(i) = y;
    texes(i) = texe;
    iexes(i) = iexe;
    a = polyReduce(a, x);
end
```

2.2.4. I**mplementacja deflacji czynnikiem liniowym**

```
function [q] = polyReduce(a, alpha)

n = length(a);
k = floor(n/2);

asc_a = flip(a);

q = zeros(n, 1);
q(1) = 0;
q(n) = a(n);

for i=n-1:-1:k+1
    q(i) = asc_a(i) + q(i+1)*alpha;
end

for i=1:n
    q(i + 1) = (q(i) - asc_a(i)) / alpha;
end

q = flip(q(2:n));
```

2.2.5. **Wyniki działania algorytmu**



Algorytm znalazł cztery miejsca zerowe (w tym dwa zespolone).

x	Liczba iteracji	Czas
0.04+0.689i	8	0.00591s
0.04-0.689i	5	0.00046s
2.32	4	0.00032s
-0.90	1	0.00024s

2.3. **Wnioski**

Algorytm bez problemu znalazł wszystkie miejsca zerowe wielomianu.

Metodzie tej zgodnie z przewidywaniami nie sprawiło problemu znalezienie rozwiązań w przestrzeni liczb zespolonych.

Dzięki zastosowaniu deflacji czynnikiem liniowym, znajdowanie kolejnych pierwiastków dla wielomianów niższych stopni wymaga znacznie mniej czasu i iteracji.