Metody numeryczne – Projekt 2.

Autor: Aleksandra Jamróz

Zadanie 1.

Treść:

Proszę znaleźć wszystkie pierwiastki funkcji

$$f(x)=3.1-3x-e^{-0.6x}$$

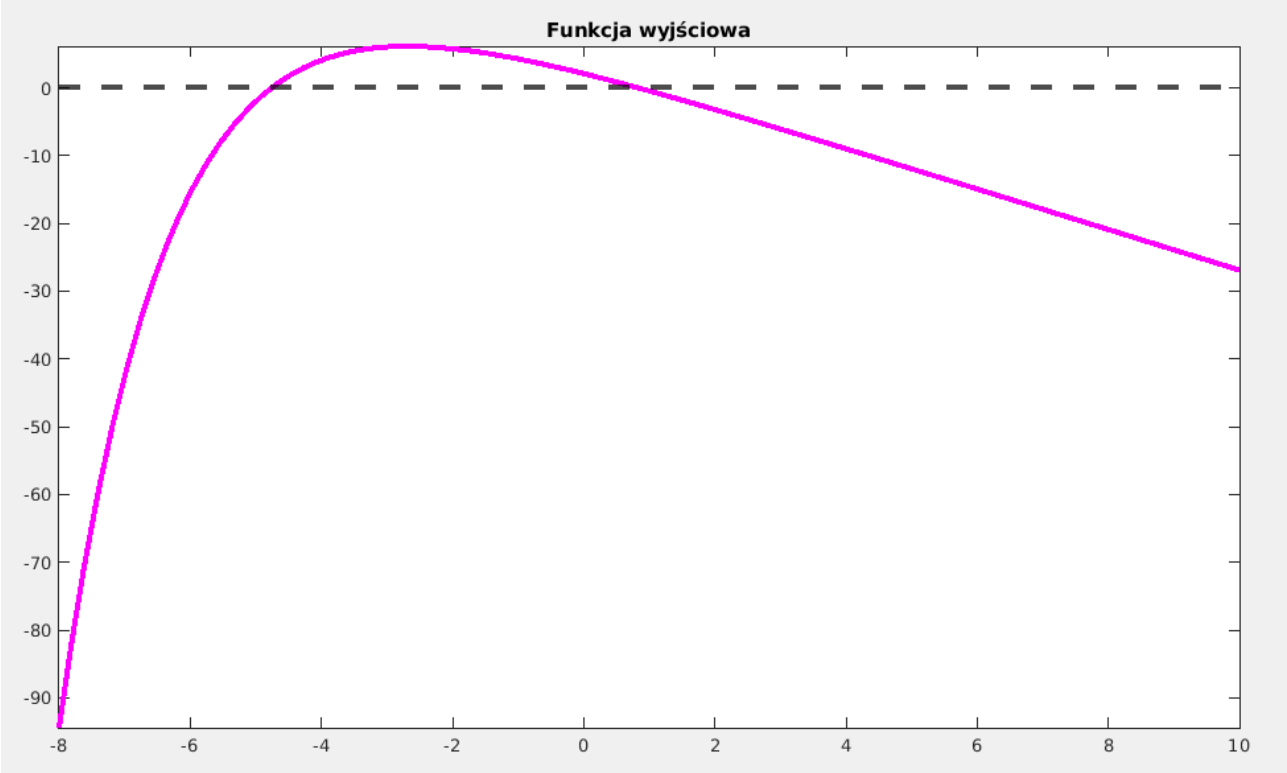
przedziale [–8, 10] używając:
a) własnego solwera z implementacją metody bisekcji.
b) podanego na stronie przedmiotu solwera newton.m z implementacją metody Newtona.

Rozwiązanie:

Rozwiązanie zaczęłam od zaimplementowania podanej funkcji oraz narysowania jej w podanym przedziale na wykresie.

```
function [y] = foo(x)
    y = 3.1 - 3*x - exp(1)^(-0.6*x);
end
```

Obraz 1: Implementacja funkcji f



Obraz 2: Wykres funkcji f w przedziale [-8 10]

Na wykresie oprócz funkcji zaznaczyłam prostą y=0. Dzięki temu widzimy, że funkcja f ma w zadanym przedziale 2 pierwiastki.

a) Rozwiązanie z metodą bisekcji

Metoda bisekcji to metoda iteracyjna, w której w celu znalezienia pierwiastka funkcji, przy każdej iteracji przechodzimy przez następujące kroki:

1. Wyznaczenie środka c aktualnego przedziału, zawierającego zero funkcji:

$$c_n = \frac{a_n + b_n}{2}, a_n - \textit{aktualny początek przedziału}, b_n - \textit{aktualny koniec przedziału}$$

2. Obliczamy iloczyny wartości funkcji na krańcach przedziału I w środku przedziału

$$ac = f(a_n) * f(c_n)$$
$$bc = f(b_n) * f(c_n)$$

3. Na nowe krańce przedziału zostaje wybrana para, dla której iloczyn wartości funkcji jest ujemny.

W tradycyjnym podejściu kroki powtarzamy tak długo aż błąd rozwiązania będzie zadowalający, lub kiedy przejdziemy przez określoną liczbę iteracji. W naszym przypadku warunkiem stopu jest uzyskanie błędu rozwiązania nie większego niż dokładność maszynowa eps.

```
function [xf, ff] = bisekcja(funkcja, a, b)

    c = (a + b) / 2;                    % wyznaczenie środka przedziału dla 1 iteracji
    fc = fukcja(c);                     % wyznaczenie wartości funkcji dla 1 iteracji
    while (abs(fc) > eps)               % sprawdzenie warunku stopu
        c = (a + b) / 2;                % wyznaczenie środka przedziału
        fc = fukcja(c);                 % obliczenie wartości funkcji dla środka przedziału
        if (funkcja(a) * fc < 0)        % obliczenie iloczynu f(a) * f(c), sprawdzenie czy ujemny
            b = c;                      % przypisanie c na nowy koniec przedziału
        else
            a = c;                      % przypisanie c na nowy początek przedziału
        end
    end
xf = c;
ff = funkcja(xf);
end
```

Obraz 3: Implementacja metody bisekcji

Niestety taki warunek stopu okazał się niewystarczający. Bez względu na liczbę iteracji, minimalną wartością funkcji, jaką udało mi się uzyskać za pomocą tej metody, to 7.1054e-15. Dokładność maszynowa matlaba to 2.2204e-16, a więc udało mi się osiągnąć błąd 32 razy większy od zamierzonego. Utrzymywał się on od 37 iteracji. Przyjęłam więc taki błąd za warunek stopu, a maksymalną liczbę iteracji podstawiałam 100.

```
function [xf, ff, iter] = bisekcja(funkcja, a, b, delta, imax)

    c = (a + b) / 2;                    % wyznaczenie środka przedziału dla 1 iteracji
    fc = funkcja(c);                    % wyznaczenie wartości funkcji dla 1 iteracji
    iter = 0;
    while (abs(fc) > delta && iter < imax) % sprawdzenie warunku stopu
        iter = iter + 1;
        if (funkcja(a) * fc < 0)        % obliczenie iloczynu f(a) * f(c), sprawdzenie czy ujemny
            b = c;                      % przypisanie c na nowy koniec przedziału
        else
            a = c;                      % przypisanie c na nowy początek przedziału
        end
        c = (a + b) / 2;                % wyznaczenie nowego środka przedziału
        fc = funkcja(c);                % obliczenie wartości funkcji dla środka przedziału
    end
xf = c;
ff = funkcja(xf);
end
```

Obraz 4: Implementacja metody bisekcji po zmianie warunku stopu i wprowadzeniu maksymalnej liczby iteracji

Przedziały izolacji kolejnych pierwiastków funkcji dla metody bisekcji należy wyznaczać automatycznie, dlatego napisałam do tego odrębną funkcję *przedzialy_izolacji*. Przyjmuje ona funkcję, początek i koniec przedziału w obrębie którego szukamy przedziałów izolacji oraz interwał po którym poruszamy się w trakcie szukania granic przedziałów. Zakładamy, że w obrębie jednego interwału nie znajdują się 2 pierwiastki funkcji. Zwraca ona 2 wektory, jeden z początkami, drugi z końcami przedziałów izolacji. Początki zakresów do metody Newtona wyznaczyłam samodzielnie z wykresu.

```
function [as, bs] = przedzialy_izolacji(f, a, b, interval)

    starting_point = a;
    curr_point = a;
    as = [];
    bs = [];
    while curr_point < b
        if foo(starting_point) * f(curr_point) < 0
            as = [as, starting_point];
            bs = [bs, curr_point];
            starting_point = curr_point;
        end
        curr_point = curr_point + interval;
    end
end
```

Obraz 5: Implementacja funkcji wyznaczającej przedziały izolacji pierwiastków funkcji f w zadanym przedziale

Po implementacji metod przeszłam do eksperymentów. Oto otrzymane wyniki:

Metoda bisekcji

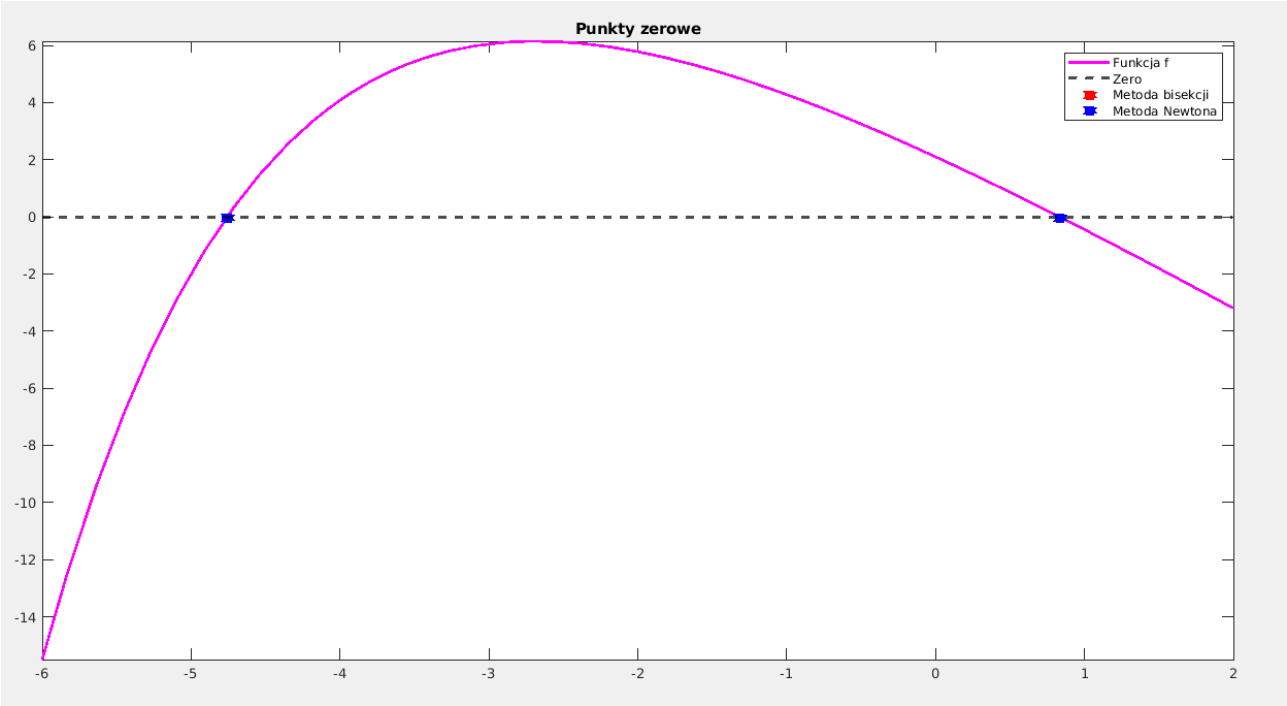
	1. przedział izolacji		2. przedział izolacji	
	Początek	Koniec	Początek	Koniec
Punkt	-8	-4.7	-4.7	0.9
Wartość funkcji w punkcie	-94.4104	0.4231	0.4231	-0.1827

Metoda Newtona

	1. pierwiastek	2. pierwiastek
Początek przedziału	-8	-2.5
Wartość funkcji w nim	-94.4104	6.1183

Porównanie metod

	1. pierwiastek		2. pierwiastek	
	Metoda bisekcji	Metoda Newtona	Metoda bisekcji	Metoda Newtona
Pierwiastek	-4.75840913130918	-4.75840913130918	0.830855667147**11**	0.830855667147**09**
Wartość funkcji w nim	-7.10542735760100e-15	-7.10542735760100e-15	-6.21724893790088e-15	-1.11022302462516e-15
Liczba iteracji	51	7	47	5
Różnica w rozwiązaniu	Identycze		Różnica na 14. oraz 15. miejscu po przecinku	



Obraz 6: Wykres z zaznaczonymi wyliczonymi punktami zerowymi

Na powyższym wykresie widoczny jest wykres funkcji f z zaznaczonymi punktami zerowymi. Nie widać punktów wyliczonych za pomocą metody bisekcji, gdyż są identycze z tymi wyliczonymi za pomocą metody Newtona. Dla lepszej czytelności wykresu, znając już wartości pierwiastków, ograniczyłam zakres z oryginalnego [-8, 10] do [-6,2].

Obie metody dobrze radzą sobie z zadaniem szukania pierwiastków funkcji. Znalezione przez nie pierwiaski nie różnią się, lub różnią się między sobą w minimalnym zakresie. Znaczącą różnicą jest jedynie liczba iteracji – metoda bisekcji potrzebuje ich znacznie więcej niż metoda Newtona.

Kod wykorzystany do przeprowadzenia eksperymentów można znaleźć w pliku *eksperymenty.m*.

Zadanie 2.

Treść:

Używając metody **Müllera MM1** proszę znaleźć wszystkie pierwiastki wielomianu czwartego stopnia:

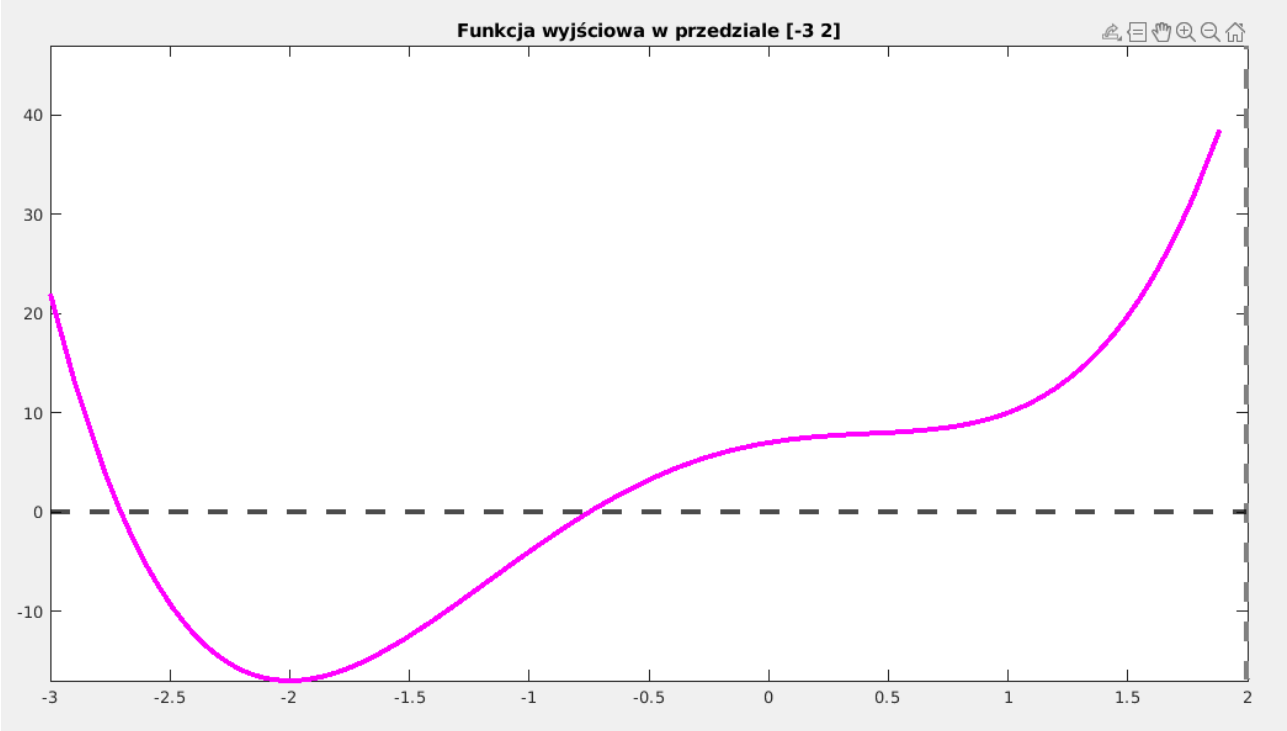
$$f(x)=a_4x^4+a_3x^3+a_2x^2+a_1x+a_0, [a_4 a_3 a_2 a_1 a_0]=[2\,3-6\,4\,7]$$

Rozwiązanie:

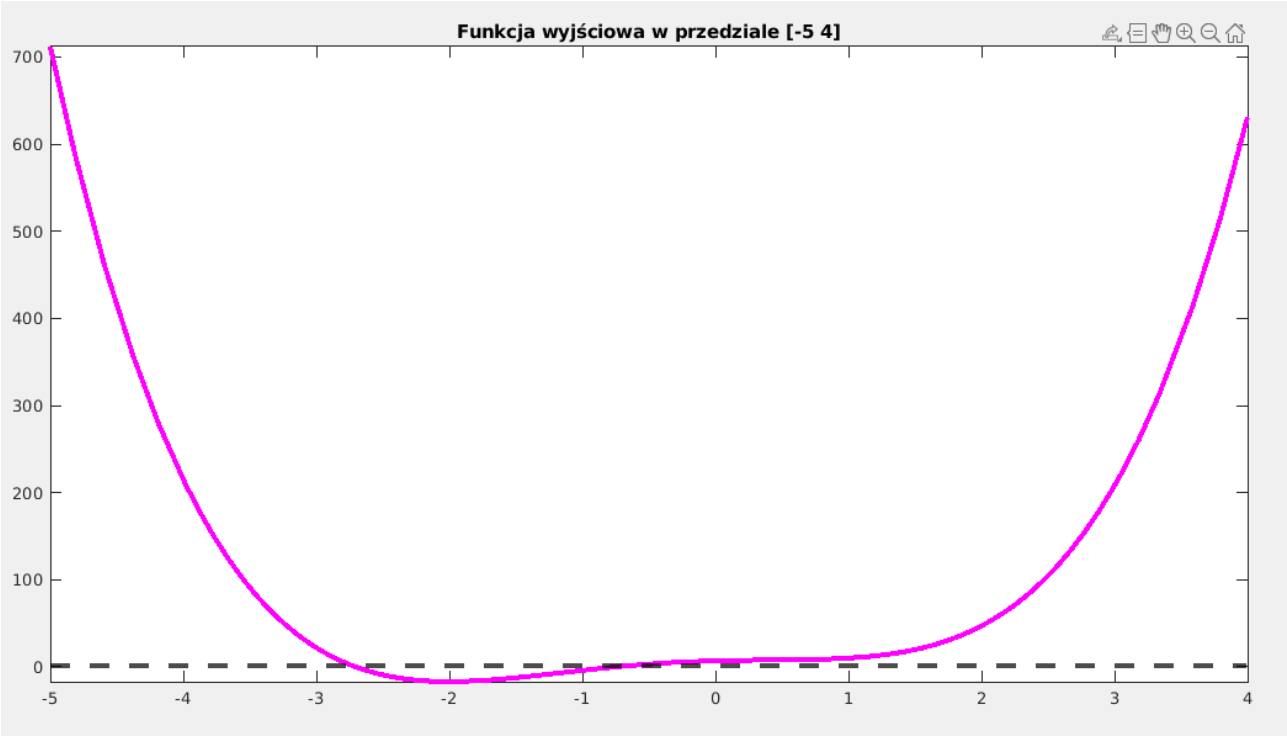
Rozwiązanie zaczęłam od zaimplementowania podanej funkcji oraz narysowania jej w podanym przedziale na wykresie.

```
function [y] = foo2(x)
    y = 2*x^4 + 3*x^3 -6*x^2 + 4*x + 7;
end
```

Obraz 7: Implementacja funkcji f



Obraz 8: Wykres fukcji z przybliżeniem na obszar największych zmian



Obraz 9: Wykres funkcji w oddaleniu

Metoda Müllera polaga na aproksymacji wielomianu w otoczeniu rozwiązania funkcją kwadratową. Jest metodą iteracyją, w której wykonujemy poniższe kroki do czasu uzyskania satysfakcjonującego rozwiązania.

1.Wyznaczamy 3 punkty: x_0, x_1, x_2 oraz wartości wielomianu w tych punktach $f(x_0), f(x_1), f(x_2)$. Przyjmujemy x_2 jako aktualną aproksymację rozwiązania.

2. Konstruujemy funkcję kwadratową przechodzącą przez te punkty:
Wprowadzamy zmienną przyrostową $z = x - x_2$. Zapisujemy również $z_0 = x_0 - x_2$ oraz $z_1 = x_1 - x_2$. Oznaczamy poszukiwaną parabolę przez $y(z) = az^2 + bz + c$. Zapisujemy równości:

$$az_0^2 + bz_0 + c = y(z_0) = f(x_0),$$
$$az_1^2 + bz_1 + c = y(z_1) = f(x_1),$$
$$c = y(0) = f(x_2).$$

Dzięki nim możemy zapisać układ równań liniowych i obliczyć współczynniki a i b.

$$az_0^2 + bz_0 = f(x_0) - f(x_2),$$
$$az_1^2 + bz_1 = f(x_1) - f(x_2).$$

3. Obliczamy miejsca zerowe paraboli za pomocą wzorów:

$$z_{plus} = \frac{-2c}{b + \sqrt{b^2 - 4ac}}$$
$$z_{minus} = \frac{-2c}{b - \sqrt{b^2 - 4ac}}$$

4. Wybieramy pierwiastek leżący bliżej aktualnego przybliżenia rozwiązania, to znaczy mający mniejszy moduł i mianujemy go nowym przybliżeniem rozwiązania:

$$x_{nowe} = x_2 + z_{min}$$

przy czym

$$z_{min} = z_{plus}, gdy |b + \sqrt{b^2 - 4ac}| \geqslant |b - \sqrt{b^2 - 4ac}|$$

$$z_{min} = z_{minus}, w\ przeciwnym\ przypadku \quad .$$

Ponadto spośród punktów x_0, x_1, x_2 wybieramy jeden leżący najdalej od nowego przybliżenia rozwiązania i odrzucamy go.

```
function [pierwiastek] = MM1(as, xs, delta)
    while abs(polyval(as, xs(3))) > delta
        % obliczenie wartości funkcji dla x0, x1 i x2
        f1 = polyval(as, xs(1));
        f2 = polyval(as, xs(2));
        f3 = polyval(as, xs(3));
        % zmienne przyrostowe
        z1 = xs(1) - xs(3);
        z2 = xs(2) - xs(3);
        % obliczenie współczynników a i b - wartości f(x0)-f(x2) oraz f(x1)-f(x2)
        r1 = f1 - f3;
        r2 = f2 - f3;
        % rozwiązanie układu równań
        a = (r1 * z2 - r2 * z1) / (z1*z2*(z1 - z2));
        b = (r2 * z1^2 - r1 * z2^2) / (z1*z2*(z1 - z2));
        c = f3;
        % obliczenie miejsc zerowych paraboli
        z_plus = (-2*c) / (b + sqrt(b^2 - 4*a*c));
        z_minus = (-2*c) / (b - sqrt(b^2 - 4*a*c));
        % wybranie nowego przybliżenia rozwiązania
        if abs(b + sqrt(b^2 - 4*a*c)) >= abs(b - sqrt(b^2 - 4*a*c))
            x_new = xs(3) + z_plus;
        else
            x_new = xs(3) + z_minus;
        end

        % wybranie x najbardziej oddalonego od nowego przybliżenia
        out_x = 1;  % indeks x najbardziej oddalonego
        delta_x = abs(x_new - xs(1));
        if abs(x_new - xs(2)) > delta_x
            delta_x = abs(x_new - xs(2));
            out_x = 2;
        end
        if abs(x_new - xs(3)) > delta_x
            out_x = 3;
        end
        % nowy wektor x
        xs(out_x) = [];
        xs = [xs, x_new];
    end
    pierwiastek = xs(3);
end
```

Obraz 10: Implementacja funkcji MM1

Aby znaleźć wszystkie pierwiastki wielomianu, należy na zmianę zastosować powyższą funkcję oraz zmniejszyć stopień wielomianu, dzieląc go na (x – znaleziony_właśnie_pierwiastek). W tym celu napisałam funkcję odpowiadającą za deflację czynnikiem liniowym o nazwie *deflacja*. Wykorzystuje ona sklejany schemat Hornera – połączenia schematu prostego i odwrotnego.

Schemat Hornera:

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

$$q_i=a_i+q_{i+1}\alpha, i=n, n-1, \dots, 1,0$$

Odwrotny schemat Hornera:

$$q_0=0$$

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

Sklejany schemat Hornera:

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

```
function q = deflacja(as, pierwiastek)
    n = length(as);          % pobranie aktualnej długości wektora współczynników
    k = floor(n/2);          % wyznaczenie granicy między prostym a odwotnym schematem
    q = zeros(1, n-1);       % inicjalizacja nowego wektora współczynników

    % Schemat prosty
    i = 1;
    while(i < k)
        if (i == 1)
            q(i) = as(i);
        else
            q(i) = as(i) + q(i - 1) * pierwiastek;
        end
        i = i + 1;
    end
    % Schemat odwrotny
    j = n;
    while (j >= k + 1)
        if (j ==n)
            q(j - 1) = -as(j) / pierwiastek;
        else
            q(j - 1) = (q(j) - as(j)) / pierwiastek;
        end
        j = j - 1;
    end
end
```

Obraz 11: Implementacja funkcji deflacja

.

Mając wszystkie elementy mogłam połączyć je w solwer przyjmujący współczynniki wielomianu i wyznaczający jego pierwiastki oraz wartości wielomianu w tych punktach.

```
function [pierwiastki, wartosci] = MM1_solver(as)
    as_in = as;                 % zapisanie wejściowych współczynników

    n = length(as) - 1;         % obliczenie stopnia wielomianu
    pierwiastki = zeros(n, 1);  % inicjalizacja wektora rozwiązań
    wartosci = zeros(n, 1);     % inicjalizacja wektora wartości

    x = [0,1,2];
    delta = 1e-8;

    for i = 1:n - 1
        pierwiastki(i) = MM1(as, x, delta);
        as = deflacja(as, pierwiastki(i));
        wartosci(i) = abs(polyval(as_in, pierwiastki(i)));
    end

    % obliczamy ostatni pierwiastek
    pierwiastki(n) = - as(2) / as(1);
    wartosci(n) = abs(polyval(as_in, pierwiastki(n)));
end
```

Obraz 12: Implentacja solwera

```
as = [2, 3, -6, 4, 7];
[pierwiastki, wartosci] = MM1_solver(as)
```

Obraz 13: Wywołanie solwera na moim wielomianie

```
pierwiastki =

   0.9774 + 0.8780i
   0.9774 - 0.8780i
  -0.7495 - 0.0000i
  -2.7054 + 0.0000i


wartosci =

   1.0e-09 *

    0.0000
    0.0138
    0.0049
    0.1262
```

Obraz 14: Wynik działania solwera

.

Jak widać solwer obliczył 4 pierwiastki wielomianu, 2 z nich rzeczywiste, 2 zawierają składową urojoną. Błąd rozwiązania był z każdym kolejnym pierwiastkiem większy, co jest skutkiem deflacji czynnikiem liniowym. Błąd przy pierwszym obliczonym pierwiastku wynosi 0.