ŁUKASZ STANISZEWSKI, NR INDEKSU: 304098

METODY NUMERYCZNE

PROJEKT 3 – ZADANIA 3.8

1 ZNAJDOWANIE ZER FUNKCJI

Celem tego zadania było znalezienie wszystkich zer funkcji

$$\boldsymbol{f(x) = 1.1 \cdot x \cdot sin(x) - 2 \cdot \ln(x+2)}$$

w przedziale $[\mathbf{2, 12}]$, używając dla każdego z zer programu z implementacją:

- metody bisekcji,
- metody Newtona.

W tym projekcie skupiono się na metodach iteracyjnych, służących do znajdowania zer funkcji, tzn. wyznaczania rozwiązania pojedynczego równania nieliniowego $\boldsymbol{f(x) = 0}$. Aby można było takie równanie rozwiązać, konieczne jest na początku zdefiniowanie przedziału izolacji pierwiastka – przedziału, w którym znajduje się konkretnie jeden pierwiastek równania.

Pierwsza z metod – metoda bijekcji (połowienia) zakłada schemat, w którym to zaczyna się z początkowego przedziału – przedziału izolacji pierwiastka $[\boldsymbol{a_0, b_0}]$ i w każdej iteracji kolejno:

- Bieżący przedział zawierający zero funkcji $[\boldsymbol{a_n, b_n}]$ jest dzielony na dwie połowy, a punktem środkowym podziału jest $\boldsymbol{c_n = (a_n + b_n)/2}$.
- Obliczany jest iloczyn $\boldsymbol{f(a_n) \cdot f(c_n)}$, jeśli iloczyn ten jest ujemny, wtedy nowym bieżącym przedziałem staje się przedział $[\boldsymbol{a_n, c_n}]$, inaczej $[\boldsymbol{c_n, b_n}]$.

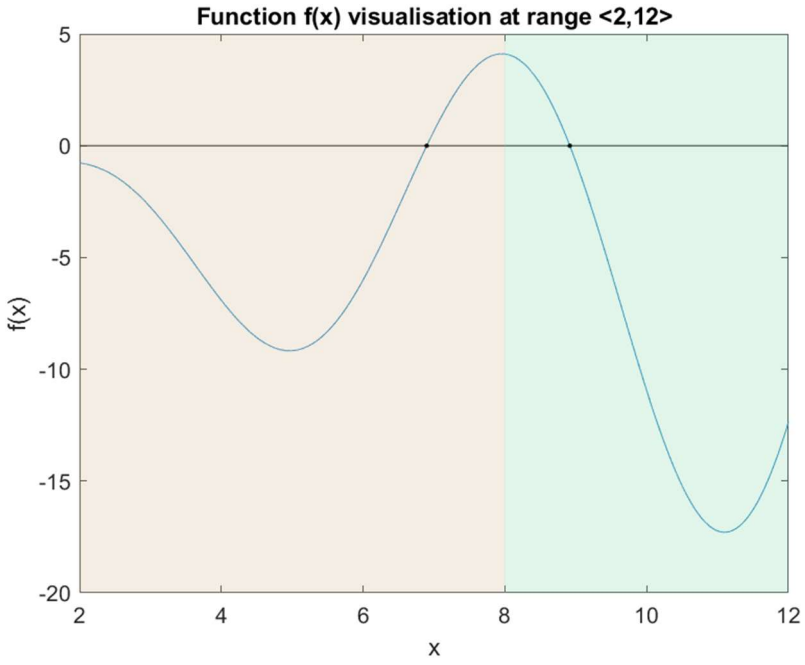
Iteracja ta trwa tak długo, dopóki nie spełnimy któregoś z warunków kończących iterację:

- Przekroczona zostaje maksymalna liczba iteracji.
- Długość przedziału bieżącego jest mniejsza niż minimalna możliwa, zadana jako parametr.
- Dokładność rozwiązania jest wystarczająca, tzn. $\boldsymbol{|f(c_n)| < \delta}$, gdzie $\boldsymbol{\delta}$ – założona dokładność rozwiązania.

Druga z metod – metoda Newtona, inaczej zwana metodą stycznych - jest metodą zakładającą aproksymację funkcji jej liniowym przybliżeniem, wynikającym z uciętego rozwinięcia w szereg Taylora, gdzie następny punkt - $\boldsymbol{x_{n+1}}$ – wynika z przyrównania do zera sformułowanej lokalnej liniowej aproksymacji funkcji $\boldsymbol{f(x)}$ – w ten sposób, po odpowiednich przekształceniach, otrzymujemy zależność iteracyjną wyprowadzającą wzór na $\boldsymbol{x_{n+1}}$:

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

Realizację zadania rozpocząłem od narysowania funkcji $\boldsymbol{f(x)}$ i wyznaczenia przedziału izolacji dla poszczególnych pierwiastków. Na podstawie niżej zamieszczonego wykresu można zauważyć, że funkcja $\boldsymbol{f(x)}$ przyjmuje wartość $\mathbf{0}$ w dwóch miejscach, dodatkowo naniosłem na wykres przedziały, w których pierwiastki są izolowane – dla mniejszego z nich będzie to przedział $[\mathbf{2, 8}]$ (kolor kremowy), natomiast dla większego – przedział $[\mathbf{8, 12}]$ (kolor zielony).



W wyniku realizacji zadania metodą bijekcji stworzyłem solver, realizujący te zadanie – znajduje się on w pliku bisectionSolver.m:

```
function [c] = bisectionSolver(a0, b0, function_f, delta, max_iter)
%BISECTIONSOLVER Solver finfing zero in given range using bisection method
    n_iter = 1;
    a = a0;
    b = b0;
    c = (a + b)/2;
    f_c = function_f(c);
    next_args(n_iter) = c;
    next_vals(n_iter) = f_c;
    while abs(f_c)>delta && max_iter >= n_iter && (b-a)>delta
        f_a = function_f(a);
        if f_a * f_c < 0
           b = c;
        else
           a = c;
        end
        c = (a + b)/2;
        f_c = function_f(c);
        n_iter = n_iter + 1;
        next_args(n_iter) = c;
        next_vals(n_iter) = f_c;
    end
    disp(["Liczba iteracji:", n_iter]);
    disp(["Punkt końcowy:", c]);
    disp(["Wartość dla punktu końcowego:", f_c]);
end
```

Następnie, uruchomiłem solver dla pierwszego z pierwiastków z parametrami: przedział początkowy $[\mathbf{2},\mathbf{8}]$, maksymalna liczba iteracji $= \mathbf{50}$, dokładność $\boldsymbol{\delta} = \mathbf{0.00001}$; a także dla drugiego z pierwiastków z parametrami: przedział początkowy: $[\mathbf{8},\mathbf{12}]$, maksymalna liczba iteracji $= \mathbf{50}$, dokładność $\boldsymbol{\delta} = \mathbf{0.00001}$.

```
>> x = bisectionSolver(2, 8, @funct, 0.00001, 50);
    "Liczba iteracji:"    "19"

    "Punkt końcowy:"    "6.8972"

    "Wartość dla punktu końcowego:"    "-5.8549e-06"

>> x = bisectionSolver(8, 12, @funct, 0.00001, 50);
    "Liczba iteracji:"    "19"

    "Punkt końcowy:"    "8.9156"

    "Wartość dla punktu końcowego:"    "-8.9011e-06"
```

W wyniku realizacji zadania metodą Newtona stworzyłem solver, realizujący te zadanie – znajduje się on w pliku newtonSolver.m:

```
function [x_n] = newtonSolver(x0, func, d_func, delta, max_iter)
%NEWTONSOLVER Solver finding zero in given range using Newton method
    x_n = x0;
    n_iter = 1;
    x_next = x_n;
    while abs(func(x_n)) >= delta && n_iter <= max_iter
        x_n = x_next;
        x_next = x_n - func(x_n) / d_func(x_n);
        n_iter = n_iter + 1;
    end
    disp(["Liczba iteracji:", n_iter]);
    disp(["Punkt końcowy:", x_n]);
    disp(["Wartość dla punktu końcowego:", func(x_n)]);
end
```

Następnie, uruchomiłem solver dla pierwszego z pierwiastków z parametrami: punkt początkowy $\boldsymbol{x = 3}$, maksymalna liczba iteracji = $\mathbf{50}$, dokładność $\boldsymbol{\delta = 0.00001}$; a także dla drugiego z pierwiastków z parametrami: punkt początkowy: $\boldsymbol{x = 11}$, maksymalna liczba iteracji = $\mathbf{50}$, dokładność $\boldsymbol{\delta = 0.00001}$.

```
>> x = newtonSolver(3, @funct, @d_funct, 0.00001, 50);
    "Liczba iteracji:"    "11"
    "Punkt końcowy:"    "-9.1293+0.6202i"
    "Wartość dla punktu końcowego:"    "-3.0263e-11+1.2955e-10i"

>> x = newtonSolver(11, @funct, @d_funct, 0.00001, 50);
    "Liczba iteracji:"    "7"
    "Punkt końcowy:"    "-3.2376+1.0862i"
    "Wartość dla punktu końcowego"    "-1.7596e-08+1.2877e-08i"
```

Można zauważyć, że metoda Newtona dla podanych wartości punktu początkowego nie znajduje odpowiednich zer – wynika to ze zbieżności lokalnej metody Newtona – zastosowaliśmy ją w punkcie zbyt oddalonym od rozwiązania (poza jej obszarem atrakcji) – i stała się rozbieżna.

W następnych krokach zdecydowałem zbliżać punkty początkowe w kierunku rozwiązania, żeby sprawdzić od jakiego punktu metoda Newtona będzie znów zbieżna.

```
>> x = newtonSolver(10, @funct, @d_funct, 0.00001, 50);

    "Liczba iteracji:"    "5"

    "Punkt końcowy:"    "8.9156"

    "Wartość dla punktu końcowego:"    "-1.0355e-08"


>> x = newtonSolver(6, @funct, @d_funct, 0.00001, 50);

    "Liczba iteracji:"    "6"

    "Punkt końcowy:"    "6.8972"

    "Wartość dla punktu końcowego:"    "-3.7965e-11"
```

Możemy zauważyć, że dla punktów początkowych $x_0 = 6$ oraz $x_0 = 10$ metoda znajduje już odpowiednie zera, dodatkowo działa bardzo efektywnie – potrzebuje maksymalnie 6 iteracji aby dojść do rozwiązania.

Dla porównania, metoda bijekcji do znalezienia zer w bliższym przedziale z taką samą dokładnością potrzebuje zdecydowanie więcej iteracji – aż 3 razy:

```
>> x = bisectionSolver(6, 8, @funct, 0.00001, 50);

    "Liczba iteracji: "    "19"

    "Punkt końcowy: "    "6.8972"

    "Wartość dla punktu końcowego"    "-5.8549e-06"

>> x = bisectionSolver(8, 10, @funct, 0.00001, 50);

    "Liczba iteracji: "    "18"

    "Punkt końcowy: "    "8.9156"

    "Wartość dla punktu końcowego"    "-8.9011e-06"
```

Na koniec zamieszczam zestawienie porównujące algorytmy:

METODA	PUNKT/PRZEDZIAŁ POCZĄTKOWY (PP)	WARTOŚĆ W PP	PUNKT KOŃCOWY (PK)	WARTOŚĆ W PK	LICZBA ITERACJI
Bisekcji	$[2,8]$	$[-0.77,4.1]$	6.8972	$-5.8549 * 10^{-6}$	19
Bisekcji	$[8,12]$	$[4.1,-12.4]$	8.9156	$-8.9011 * 10^{-6}$	19
Newtona	3	-2.8	$-3.2376 + 1.0862i$	$-3.0263 * 10^{-11} + 1.2955 - 10\mathrm{i}$	11
Newtona	11	-17.2	$-9.1293 + 0.6202i$	$-1.7596 * 10^{-8} + 1.2877 - 08\mathrm{i}$	7
Bisekcji	$[6,8]$	$[-6,4.1]$	6.8972	$-5.8549 * 10^{-6}$	19
Bisekcji	$[8,10]$	$[4.1,-3.1]$	8.9156	$-8.9011 * 10^{-6}$	18
Newtona	6	-6	6.8972	$-3.7965 * 10^{-11}$	6
Newtona	10	-3.1	8.9156	$-1.0355 * 10^{-8}$	5

Na podstawie zamieszczonych wyżej eksperymentów, można zauważyć, że to co najbardziej różni obie metody – bijekcji i Newtona – jest ich zbieżność. Metoda bijekcji charakteryzuje się tym, że jest zbieżna globalnie – zawsze znajdzie pierwiastek równania, jeśli tylko zadamy jej przedział izolacji pierwiastka – może być dowolnie duży, jednak musi spełniać warunki bycia przedziałem izolacji – w przedziale tym musi znajdować się dokładnie jeden pierwiastek – ten którego szukamy.

Metoda Newtona natomiast nie jest już niezawodna – jest zbieżna lokalnie, co znaczy, że jeśli zaczniemy stosować ją w punkcie zbytnio oddalonym od rozwiązania (czyli poza obszarem atrakcji pierwiastka) to może być ona rozbieżna – co zauważono dla punktów początkowych $\boldsymbol{x = 3}$ oraz $\boldsymbol{x = 11}$ – nie udało się wtedy znaleźć pierwiastka równania. Sytuacje, w których metoda ta nie zadziała to przede wszystkim źle podany przedział izolacji pierwiastka, a także uznanie za punkt startowy punktu będącego poza obszarem atrakcji pierwiastka – m.in. wtedy, gdy pochodna funkcji zmienia swój znak (monotoniczność funkcji się zmienia) na przedziale $[punkt\_poczatkowy, pierwiastek]$ lub $[punkt\_poczatkowy, pierwiastek]$.

Innym aspektem, w jakim te dwa podejścia się różnią jest iloraz zbieżności – możemy zauważyć, że jeśli dobierzemy podobne przedziały / punkty początkowe dla obu metod i będzie punkt początkowy należał do obszaru atrakcji pierwiastka, wtedy metoda Newtona działa zdecydowanie szybciej, co wynika z ilorazu zbieżności obu metod – metoda bisekcji jest zbieżna liniowo (z rzędem zbieżności $\boldsymbol{p = 1}$) z ilorazem zbieżności $\boldsymbol{k = 0.5}$, gdzie metoda Newtona jest zbieżna kwadratowo, bo jej rząd zbieżności $\boldsymbol{p = 2}$ – co oznacza, że jest ona zdecydowanie szybsza.

2 ZNAJDOWANIE PIERWIASTKÓW WIELOMIANU

Celem tego zadania było użycie metody Laguerre'a w celu znalezienia wszystkich pierwiastków wielomianu:

$$\boldsymbol{f(x) = -2x^4 + 5x^3 + 7x^2 + x + 3}$$

Aby wyznaczyć wszystkie pierwiastki tego wielomianu, koniecznym było zastosowanie deflacji czynnikiem liniowym.

W tym zadaniu, koniecznym było znalezienie zer wielomianów, tzn. takich x-ów, dla których $\boldsymbol{f(x) = 0}$. Istotnym jest zwrócenie uwagi na fakt, że wielomian n-tego stopnia, czyli będący postaci

$$\boldsymbol{f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0}$$

posiada dokładnie n takich pierwiastków, dodatkowo pierwiastki te mogą być zarówno rzeczywiste jak i zespolone. Do szukania pierwiastków zarówno rzeczywistych, jak i zespolonych, istnieją specjalistyczne, bardziej złożone niż do tej pory, metody – jedną z nich jest metoda Laguerre'a.

Metoda Laguerre'a jest metodą iteracyjną poszukującą zera wielomianu, którą w ramach pojedynczej iteracji definiuje wzór mówiący o aktualizacji szukanego pierwiastka:

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

gdzie n to stopień wielomianu, a znak w mianowniku wybierany jest tak, żeby miał większy moduł. Metoda Laguerre'a jest metodą iteracyjną – trzeba dla niej zdefiniować kryterium stopu, w tym projekcie skorzystano z dwóch:

- Przekroczenia maksymalnej liczby wykonanych iteracji.
- Wystąpienia wystarczającej dokładności rozwiązania, tzn. $|f(x_k)| < \delta$, gdzie δ – założona dokładność rozwiązania.

Dodatkowo, aby móc znaleźć wszystkie pierwiastki danego wielomianu, koniecznym jest, po znalezieniu pierwiastka α, uproszczenie wielomianu dzieląc go przez czynnik $(x - \alpha)$. Takie działanie nazywamy deflacją czynnikiem liniowym. Aby ją wykonać, korzystamy z dwóch alternatywnych podejść do schematu Hornera - prostego schematu Hornera opisanego wzorami

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

oraz odwrotnego schematu Hornera (z założeniem $\alpha \neq 0$), opisywanego wzorami

$$\begin{cases} q_0 = 0 \\ q_{i+1} = \frac{q_i - a_i}{\alpha}; \quad i = 0, 1, \dots, n-1 \end{cases}$$

Podejście ze standardowymi schematami Hornera może okazać się błędne – generować błędy numeryczne. Dlatego też, często korzystamy z tzw. sklejanego schematu Hornera, gdzie:

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

Istotne jest tu ustalenie miejsca podziału. Zakładając, że b_i oznacza współczynniki q_i wyznaczone algorytmem Hornera, a c_i oznacza współczynniki q_i wyznaczone odwrotnym algorytmem Hornera, **k** najczęściej wybieramy w następujący sposób:

$$\frac{|c_k - b_k|}{|a_k| + |c_k|} = \min_{|a_j|+|c_j|>0} \frac{|c_j - b_j|}{|a_j| + |c_j|}$$

Implementując rozwiązanie, stworzyłem specjalne solvery realizujące zadanie. Stworzyłem specjalne funkcje pomocnicze:

- getPolyVal.m – oblicza wartość wielomianu o zadanych współczynnikach dla zadanego punktu x,
- getPolyValDeriv.m – oblicza wartość pochodnej pierwszego i drugiego stopnia dla wielomianu o zadanych współczynnikach i dla zadanego x,
- getHorner.m oraz getHornerReverse.m – realizujące podstawowe schematy Hornera (prosty i odwrócony).

Koniecznym było tu stworzenie solvera znajdującego pojedynczy pierwiastek wielomianu – jest to funkcja laguerreSolver.m, której kod jest następujący:

```
function [xk] = laguerreSolver(coeff, start_x, max_iters, delta)
%LAGUERRESOLVER znajduje pojedyncze zero wielomianu metodą Laguerre
    curr_degree = size(coeff, 1) - 1;
    xk = start_x;
    iter = 1;
    while iter <= max_iters && abs(getPolyVal(coeff, xk)) > delta
        % pobranie pochodnych
        [val_deriv1, val_deriv2] = getPolyValDeriv(coeff, xk);
        val_poly = getPolyVal(coeff, xk);
        % dobranie znaku w mianowniku tak aby zapewnic najmniejszy modul
        denominator_sqrt = sqrt( (curr_degree-1) * ( (curr_degree-1) *
(val_deriv1^2) - curr_degree*val_poly*val_deriv2) );
        if abs (val_deriv1+denominator_sqrt) > abs(val_deriv1-denominator_sqrt)
            denominator_sqrt = val_deriv1+denominator_sqrt;
        else
            denominator_sqrt = val_deriv1-denominator_sqrt;
        end
        % aktualizacja xk
        xk = xk - curr_degree * val_poly / denominator_sqrt;
        iter = iter + 1;
    end
    disp(['Found zero for x=',num2str(xk),' in ', num2str(iter),' iterations!']);
end
```

Dodatkowo, koniecznym było stworzenia solvera realizującego deflację czynnikiem liniowym, która korzysta, w sposób efektywny ze schematu Hornera oraz jego wersji w postaci odwróconej.

```
function [new_coeff] = getDeflation(coeff, alfa)
%GETDEFLATION przeprowadza deflację w sposób optymalny dla wielomianow
% wyzszego rzedu
    hornerCoeffs = getHorner(coeff, alfa);
    if alfa ~= 0
        hornerRevCoeffs = getHornerReverse(coeff, alfa);
        % wybieranie najlepszego k zgodnie ze wzorem
        best_k = 0;
        best_k_val = inf;
        temp_coeff = coeff(1:size(coeff, 1)-1, 1);
        for temp_k = 1:size(hornerRevCoeffs,1)
            aj = temp_coeff(temp_k);
            bj = hornerCoeffs(temp_k);
            cj = hornerRevCoeffs(temp_k);
            if abs(aj) + abs(cj) > 0
                if abs(cj-bj)/(abs(aj)+abs(cj)) < best_k_val
                    best_k = temp_k;
                    best_k_val = abs(cj-bj)/(abs(aj)+abs(cj));
                end
            end
        end
        for k=best_k:size(hornerCoeffs,1)
            hornerCoeffs(k,1) = hornerRevCoeffs(k,1);
        end
    end
    new_coeff = hornerCoeffs;
end
```

Na samym końcu, stworzyłem solver, który łączy zaimplementowane przeze mnie funkcje i dla zadanego zbioru współczynników wielomianu i miejsca startu (oraz kryteriów stopu) znajduje wszystkie pierwiastki wielomianu. Kod ten został stworzony w postaci funkcji laguerreDeflationSolver.m:

```
function [ret_zeros] = laguerreDeflationSolver(coeff, start_x, max_iters, delta)
%LAGUERREDEFLATIONSOLVER znajduje wszystkie zera wielomianu metodą Laguerre
%dzieki zastosowaniu deflacji czynnikiem zerowym (optymalna forma)
    curr_degree = size(coeff, 1) - 1;
    ret_zeros = zeros(curr_degree, 1);
    curr_zeros_found = 0;
    xk = start_x;
    while curr_degree>=1
        % znajdowanie pojedynczego zera
        xk = laguerreSolver(coeff, xk, max_iters, delta);
        ret_zeros(curr_zeros_found+1) = xk;
        curr_zeros_found = curr_zeros_found + 1;
        curr_degree = curr_degree - 1;
        % zastosowanie deflacji
        coeff = getDeflation(coeff, xk);
    end

end
```

Następnie, zbadałem skuteczność algorytmu dla przydzielonego mi w ramach zadania wielomianu $f(x)$ z maksymalną liczbą iteracji 100 oraz dokładnością znajdowania zera rzędu 10^{-4}. Wyniki badań zamieściłem niżej:

```
>> coeff = [-2; 5; 7; 1; 3];
>> y = laguerreDeflationSolver(coeff,15, 100, 0.0001);
      Found zero for x=3.5569 in 3 iterations!
      Found zero for x=0.046717+0.60368i in 4 iterations!
      Found zero for x=0.046718-0.60368i in 2 iterations!
      Found zero for x=-1.1503+8.2937e-07i in 2 iterations!
>> y = laguerreDeflationSolver(coeff,0, 100, 0.0001);
      Found zero for x=0.046717-0.60368i in 4 iterations!
      Found zero for x=-1.1503+1.7177e-07i in 4 iterations!
      Found zero for x=0.046717+0.60368i in 2 iterations!
      Found zero for x=3.5569+1.225e-06i in 2 iterations!
>> y = laguerreDeflationSolver(coeff,-5, 100, 0.0001);
      Found zero for x=-1.1503 in 4 iterations!
      Found zero for x=0.046717-0.60368i in 4 iterations!
      Found zero for x=0.046717+0.60368i in 2 iterations!
      Found zero for x=3.5569-5.6421e-07i in 2 iterations!
>> y = laguerreDeflationSolver(coeff,1000, 100, 0.0001);
      Found zero for x=3.5569 in 3 iterations!
      Found zero for x=0.046717+0.60368i in 4 iterations!
      Found zero for x=0.046718-0.60368i in 2 iterations!
      Found zero for x=-1.1503+8.2937e-07i in 2 iterations!
>> y = laguerreDeflationSolver(coeff,-1000000, 100, 0.0001);
      Found zero for x=-1.1503 in 4 iterations!
      Found zero for x=0.046717-0.60368i in 4 iterations!
      Found zero for x=0.046717+0.60368i in 2 iterations!
      Found zero for x=3.5569-5.6421e-07i in 2 iterations!
```

Na końcu postanowiłem stworzyć zestawienie przedstawiające działanie algorytmu dla różnych punktów:

Punkt początkowy (pp)	Wartość f(x) dla punktu początkowego	Kolejne znalezione pierwiastki	Wartości f(x) dla kolejnych pierwiastków	Liczba iteracji do otrzymania pierwiastka
15	-82782	3.5569	-0.0039	3
		$0.046717 + 0.60368i$	$3.85 * 10^{-5} + 7.38 * 10^{-6}i$	4
		$0.046718 - 0.60368i$	$3.52 * 10^{-5} - 1.84 * 10^{-5}i$	2
		$-1.1503 + 8.2937 * 10^{-7}i$	$3.88 * 10^{-5} + 1.4 * 10^{-5}i$	2
0	3	$0.046717 - 0.60368i$	$3.85 * 10^{-5} + 7.38 * 10^{-6}i$	4
		$-1.1503 + 1.7177 * 10^{-7}i$	$3.88 * 10^{-5} + 2.91 * 10^{-6}i$	4
		$0.046717 + 0.60368i$	$3.85 * 10^{-5} + 7.38 * 10^{-6}i$	2
		$3.5569 + 1.2246 * 10^{-6}i$	$-0.0039 - 0.0001i$	2
-5	-1702	-1.1503	$3.88 * 10^{-5}$	4
		$0.046717 - 0.60368i$	$3.85 * 10^{-5} + 7.38 * 10^{-6}i$	4
		$0.046717 + 0.60368i$	$3.85 * 10^{-5} + 7.38 * 10^{-6}i$	2
		$3.5569 - 5.6421 * 10^{-7}i$	-0.0039	2
1000	$-1.9950 * 10^{12}$	3.5569	-0.0039	3
		$0.046717 + 0.60368i$	$3.85 * 10^{-5} + 7.38 * 10^{-6}i$	4
		$0.046718 - 0.60368i$	$3.52 * 10^{-5} - 1.84 * 10^{-5}i$	2
		$-1.1503 + 8.2937 * 10^{-7}i$	$3.88 * 10^{-5} + 1.4 * 10^{-5}i$	2
-1000000	$-2 * 10^{24}$	-1.1503	$3.88 * 10^{-5}$	4
		$0.046717 - 0.60368i$	$3.85 * 10^{-5} + 7.38 * 10^{-6}i$	4
		$0.046717 + 0.60368i$	$3.85 * 10^{-5} + 7.38 * 10^{-6}i$	2
		$3.5569 - 5.6421 * 10^{-7}i$	-0.0039	2

Przedstawione wyżej wyniki pokazują, że metoda Laguerre'a jest metodą bardzo szybką – w przypadku tego wielomianu i dokładności rzędu $\mathbf{10^{-4}}$ znajduje pojedyncze pierwiastki w maksymalnie **5** iteracji. Dodatkowo, w jej przypadku podanie innego punktu początku poszukiwań nie skutkuje nieznalezieniem pierwiastków, a jedynie znalezieniem ich w innej kolejności. Dodatkowo warto zauważyć, że metoda znajduje również bardzo dobrze i szybko pierwiastki zespolone. Można wysunąć wniosek, że w przypadku tego zadania, jest ona zbieżna globalnie. Zastosowanie metody deflacji czynnikiem liniowym w połączeniu z metodą Laguerre'a, pozwoliło na znalezienie wszystkich pierwiastków wielomianu.