Imię i nazwisko: Ada Wojterska

Temat projektu:

**Metoda Newtona wyznaczania zer wielomianów w i w^2, gdzie . (Porównanie szybkości zbieżności w przypadku funkcji o zerach jednokrotnych i dwukrotnych.) Do obliczania wartości wielomianu i jego pochodnej zastosować metodę Hornera.**

**Część 1 – opis metody**

Metoda Newtona (stycznych) jest iteracyjną metodą pozwalającą na przybliżanie miejsc zerowych funkcji. Polega na wykorzystywaniu lokalnej informacji o funkcji w punkcie ​, aby obliczyć nowe przybliżenie pierwiastka. Konkretnie, przy każdym kroku iteracji korzysta się z wartości funkcji oraz jej pochodnej w punkcie ​, aby obliczyć nowe przybliżenie ​. Wzór iteracyjny metody Newtona ma postać:

Powyższy wzór wynika z następujących przekształceń:

Niech będzie funkcją klasy . Wtedy rozwinięcie funkcji f w szereg Taylora w otoczeniu punktu jest następujące:

Ostatni składnik możemy zaniedbać w przybliżeniu liniowym, które wykorzystujemy w tej metodzie. Oznaczmy jako nasze przybliżenie wielomianem stopnia pierwszego.

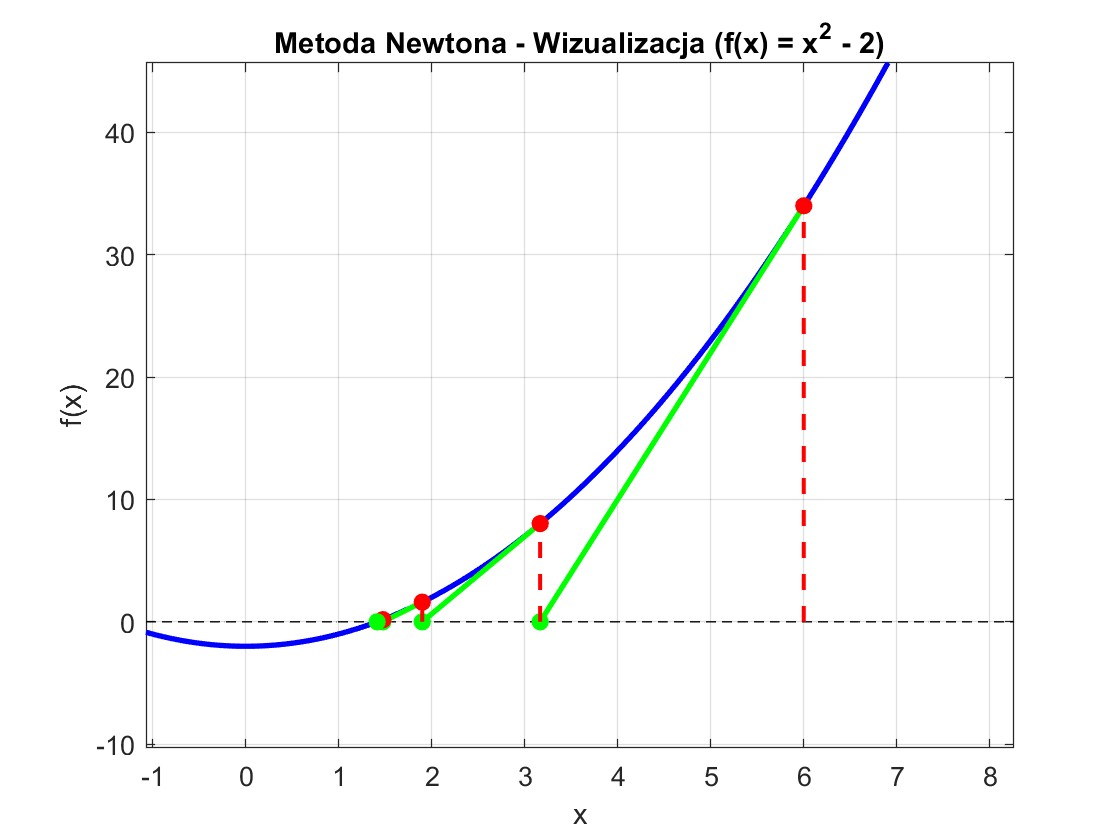
Aby znaleźć pierwiastek funkcji, chcemy, aby .

Podstawiając rozwinięcie Taylora:

Wtedy:

W ten sposób otrzymujemy pokazany na początku wzór iteracyjny metody Newtona. Kolejne przybliżenia w tej metodzie są więc miejscami zerowymi stycznej do f poprowadzonej w punkcie .

Wizualizacja iteracji dla metody Newtona:



Istnieją różne warunki stopu, które definiują nam moment w którym zakończymy nasze iteracje. Warunek który będę wykorzystywała:

| −| < d,

gdzie d to określona przez nas tolerancja.

Zbieżność metody Newtona dla zer jednokrotnych i dwukrotnych:

Definicja wykładnika zbieżności (z wykładu):

Niech jest błędem w k-tym kroku (gdzie a to miejsce zerowe funkcji).

Jeśli istnieją liczby p i C (C > 0), że:

,

To p jest wykładnikiem zbieżności metody.

W naszym przypadku możemy udowodnić, że dla metody Newtona wykładnik ten jest równy 2 dla zer jednokrotnych oraz 1 dla zer wielokrotnych, co oznacza, że metoda powinna prowadzić nas do rozwiązania zdecydowanie szybciej w przypadku zer jednokrotnych.

**Część 2 – opis programu**

Funkcje:

* find\_roots.m
* myhorner.m
* deflate.m

Skrypty testujące:

* main.m i dodatkowe do ciekawych przypadków ciekawe1.m, ciekawe2.m…

Dodatkowe – niezwiązane konkretnie z programem:

* wykres\_newton.m – generowanie wizualizacji kolejnych iteracji w metodzie Newtona.

find\_roots.m

Jest to główna funkcja, na której opiera się działanie programu.

Argumenty wejściowe:

* coeffcients - współczynniki wielomianu od do .
* max\_iter - maksymalna liczba iteracji szukania miejsca zerowego funkcji f.
* tol - tolerancja - jeżeli | - | < tol, funkcja uzna, że znalazła miejsce zerowe
* x\_poczatkowy - początkowe przybliżenie miejsca zerowego

Argumenty wyjściowe:

* roots - wszystkie znalezione miejsca zerowe funkcji, w kolejności znalezienia
* iterations = liczba iteracji potrzebna do znalezienia kolejnych miejsc zerowych

Funkcja iteruje tyle razy, ile miejsc zerowych maksymalnie może mieć wielomian (czyli najwyższa potęga x minus 1). Za każdym razem zeruje ona liczbę iteracji, a następnie, począwszy od – x\_poczatkowego maksymalnie max\_iter razy oblicza wartość i pochodną wielomianu w punkcie , a potem wyznacza . Następnie sprawdza warunek stopu i wychodzi z pętli (wpisując uprzednio znaleziony pierwiastek na listę pierwiastków i liczbę iteracji na listę liczby iteracji), jeżeli został on spełniony lub zwiększa liczbę iteracji i uaktualnia , jeżeli tak się nie stało. Jeżeli po max\_iter liczbie iteracji nie uda mu się spełnić warunku stopu, pozostawia NaN na liście.

myhorner.m

Jest to pomocnicza funkcja obliczająca wartość i pochodną wielomianu w punkcie.

Argumenty wejściowe:

* coeffcients - współczynniki wielomianu od do .
* x - argument x dla którego obliczana jest wartość funkcji i pierwsza pochodna.

Argumenty wyjściowe:

* w – wartość wielomianu w punkcie x
* p – wartość pochodnej w punkcie x

Funka wykorzystuje zoptymalizowany algorytm Hornera obliczania wartości funkcji w punkcie i jej pochodnej w tym punkcie.

Wykorzystuje on zapisanie funkcji w poniższej postaci:

A następnie obliczanie wartości funkcji począwszy od najbardziej zagłębionego nawiasu.

Podobnie w przypadku pochodnej, zapisujemy ją w postaci:

,

Gdzie = + x \* ( czyli obliczona wcześniej wartość wielomianu ).

W tym wypadku też zaczynamy obliczanie od najbardziej zagłębionego nawiasu.

deflate.m

Jest to pomocnicza funkcja dzieląca wielomian przez dwumian (x – root).

Argumenty wejściowe:

* coeffcients - współczynniki wielomianu od do .
* root – znaleziony pierwiastek wielomianu.

Argumenty wyjściowe:

* new\_coefficients - współczynniki wielomianu w postaci [ , … , ] (gdzie m = n - 1)

Funkcja zaczyna działanie od najwyższego współczynnika ​, który staje się pierwszym elementem ciągu wynikowego b (współczynnik .

Następnie iteracyjnie obliczamy nowe współczynniki reszty ciągu zmniejszając n o 1:

Każdy kolejny współczynnik jest sumą współczynnika i iloczynu ​, gdzie to wynik poprzedniej iteracji.

Ostatni obliczony element to reszta z dzielenia – u nas 0.

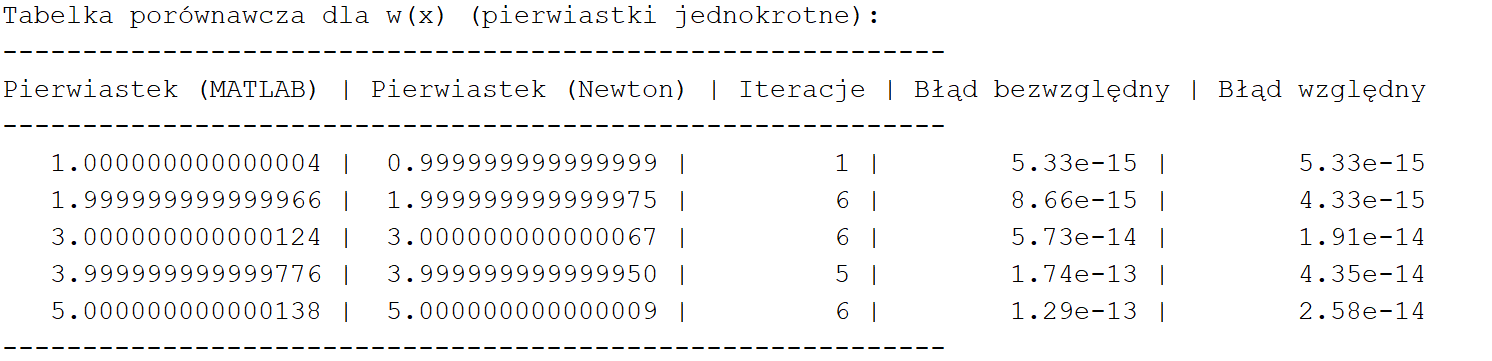
**Część 3 – ciekawe przypadki**

**Przypadek 1**

Przypadek ten będzie bardzo podstawowy, ale bardzo dobrze zobrazuje nam problem spowolnienia metody Newtona, gdy mamy do czynienia z zerami wielokrotnymi – w tym przypadku podwójnymi.

Wykorzystałam wielomian:

W przypadku zer jednokrotnych potrzebujemy zwykle tylko kilku iteracji, aby uzyskać pożądaną przez nas dokładność (u mnie ). to 4.5, a maksymalna liczba iteracji 100 000.



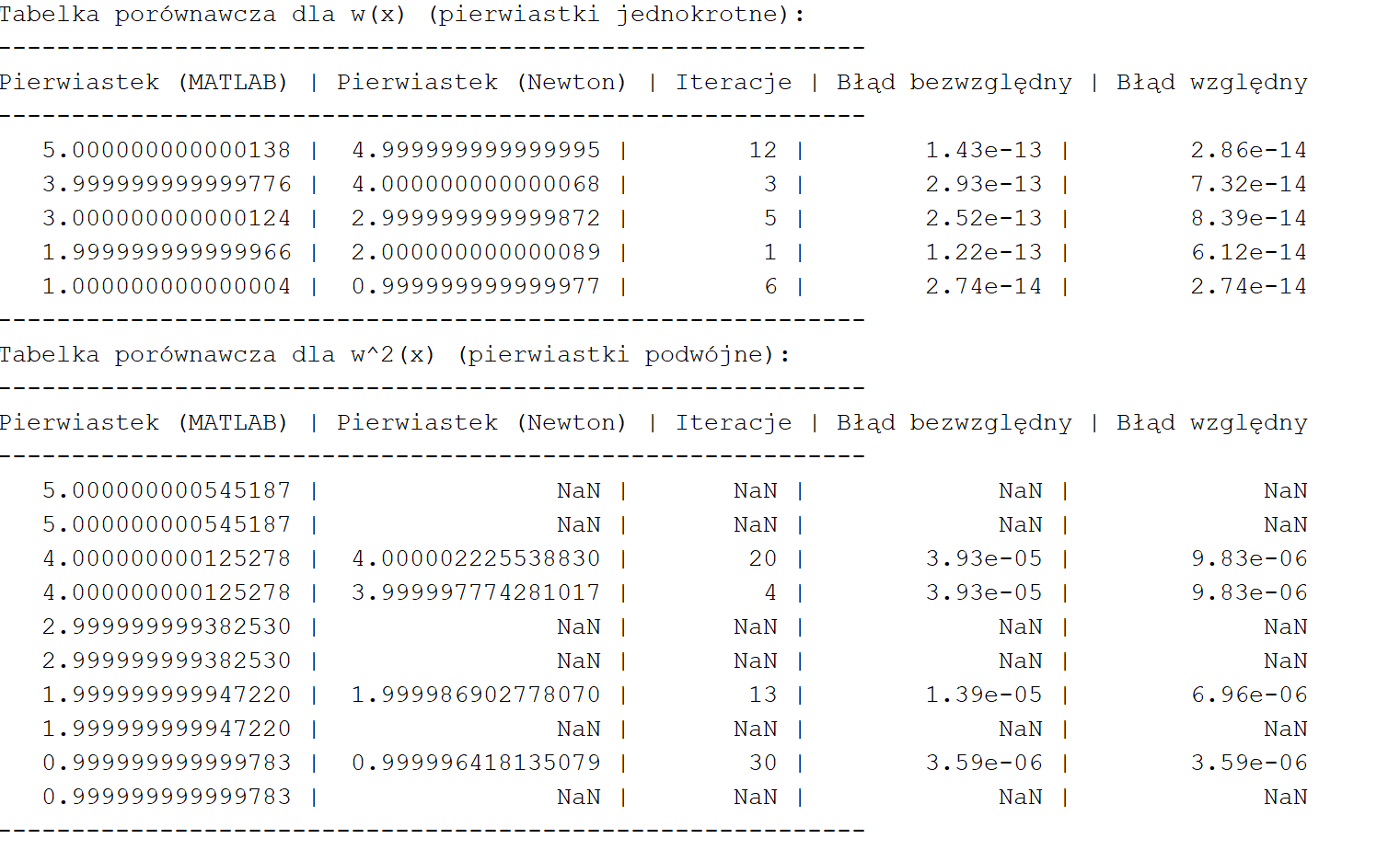
Inaczej sytuacja wygląda przy zerach dwukrotnych:



W tym przypadku często potrzebujemy kilkudziesięciu iteracji, aby uzyskać satysfakcjonujące przybliżenie. Co ciekawe, rzędy błędów w przypadku zer jednokrotnych są zdecydowanie niższe.

Co więcej, testując metodę dla różnych zauważyłam, że nawet przy teoretycznie dobrych argumentach początkowych i umożliwieniu funkcji iterowania milion razy, dla niektórych wartości funkcja podniesiona do kwadratu nie zbiega do swoich pierwiastków, co nie ma miejsca w przypadku zer jednokrotnych.

Przykład dla = 4.23:

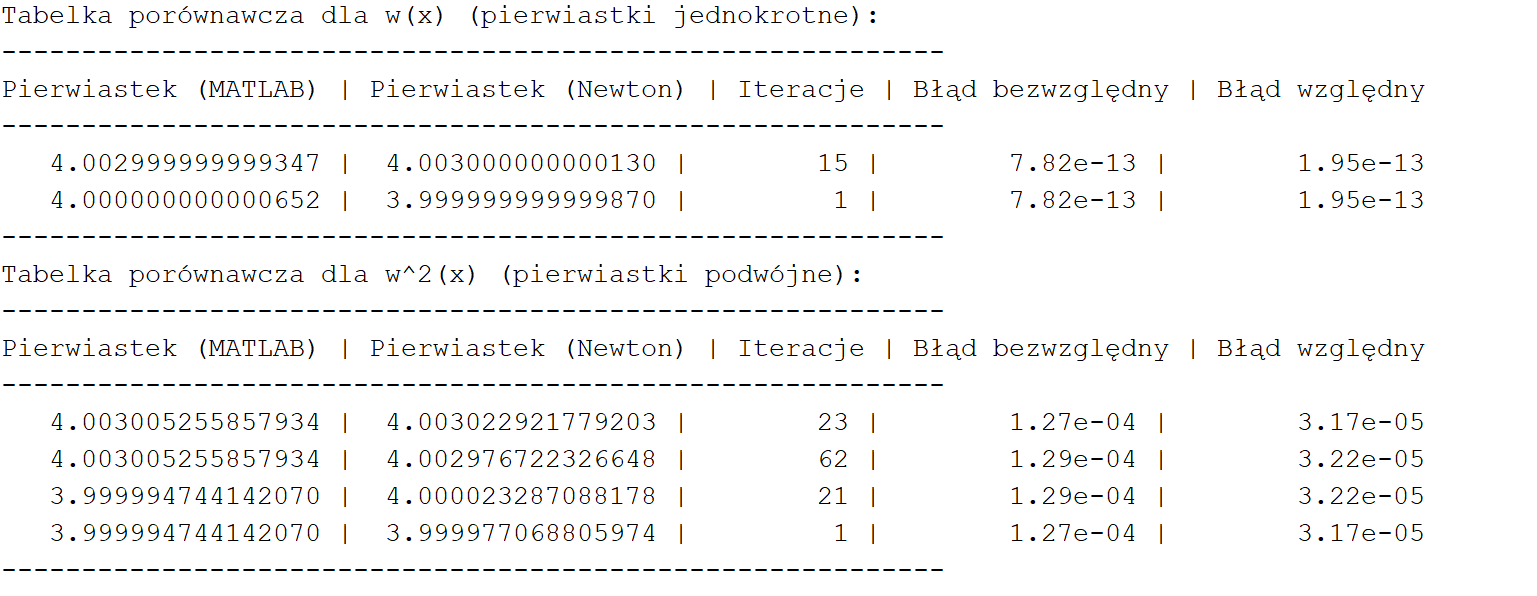


Kiedy wypisywałam jednak wartość, która była ostatnim przybliżeniem pierwiastka przed zakończeniem iteracji, zawsze była zbliżona do oczekiwanej. Z moich przypuszczeń wynika, że mogło więc dojść do sytuacji, w której w kolejnych iteracjach „przeskakiwano” przez miejsce zerowe funkcji.

**Przypadek 2**

Tym razem postanowiłam rozważyć funkcje, której pierwiastki położone są bardzo blisko siebie. Wybrany wielomian w(x) ma postać:

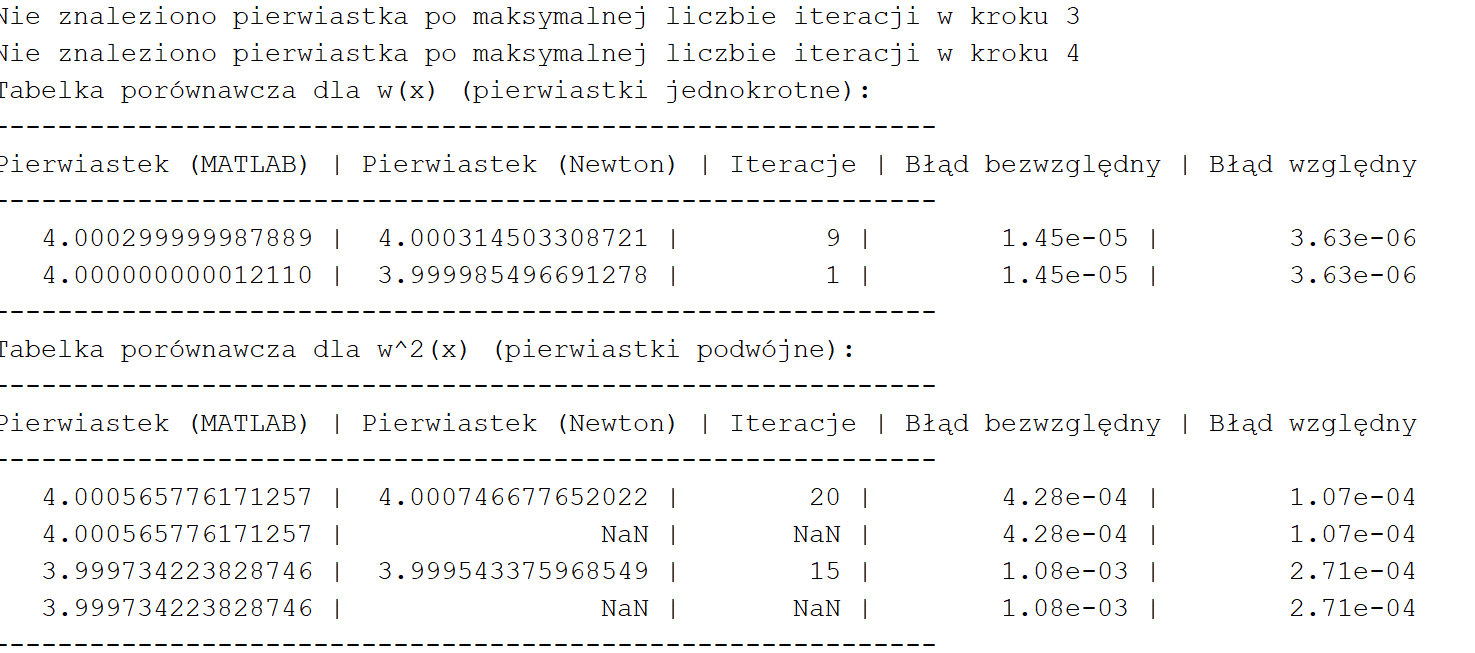
W tym przypadku, nawet podczas wybierania punktu początkowego dość odległego od pierwiastków (np. 10), program radził sobie dość dobrze. Wybrana przeze mnie tolerancja to . Co ciekawe, w tym przypadku pierwiastki znajdowane przez funkcję roots(), miały bardzo podobny błąd do tych znajdowanych przez mój algorytm. Tak jak w poprzednim przykładzie, w przypadku wielomianu z pierwiastkami dwukrotnymi, potrzebowaliśmy zdecydowanie więcej iteracji, by je przybliżyć, a błąd względny i bezwzględny był zdecydowanie większy.



Rozważyłam więc wielomian, którego pierwiastki znajdują się jeszcze bliżej siebie.

W tym wypadku mój algorytm działał już zdecydowanie gorzej, nawet po zmianie dokładności na lub i miejsca początkowego na 4.001. Dopiero przy rozpoczęciu iteracji np. w = 4.00015 udało mi się dwukrotnie znaleźć wszystkie miejsca zerowe wielomianu podniesionego do kwadratu. Także w tym wypadku jednak zarówno wyniki funkcji roots() i moje miały podobny błąd.

Wyniki dla = 4.1, tol = :

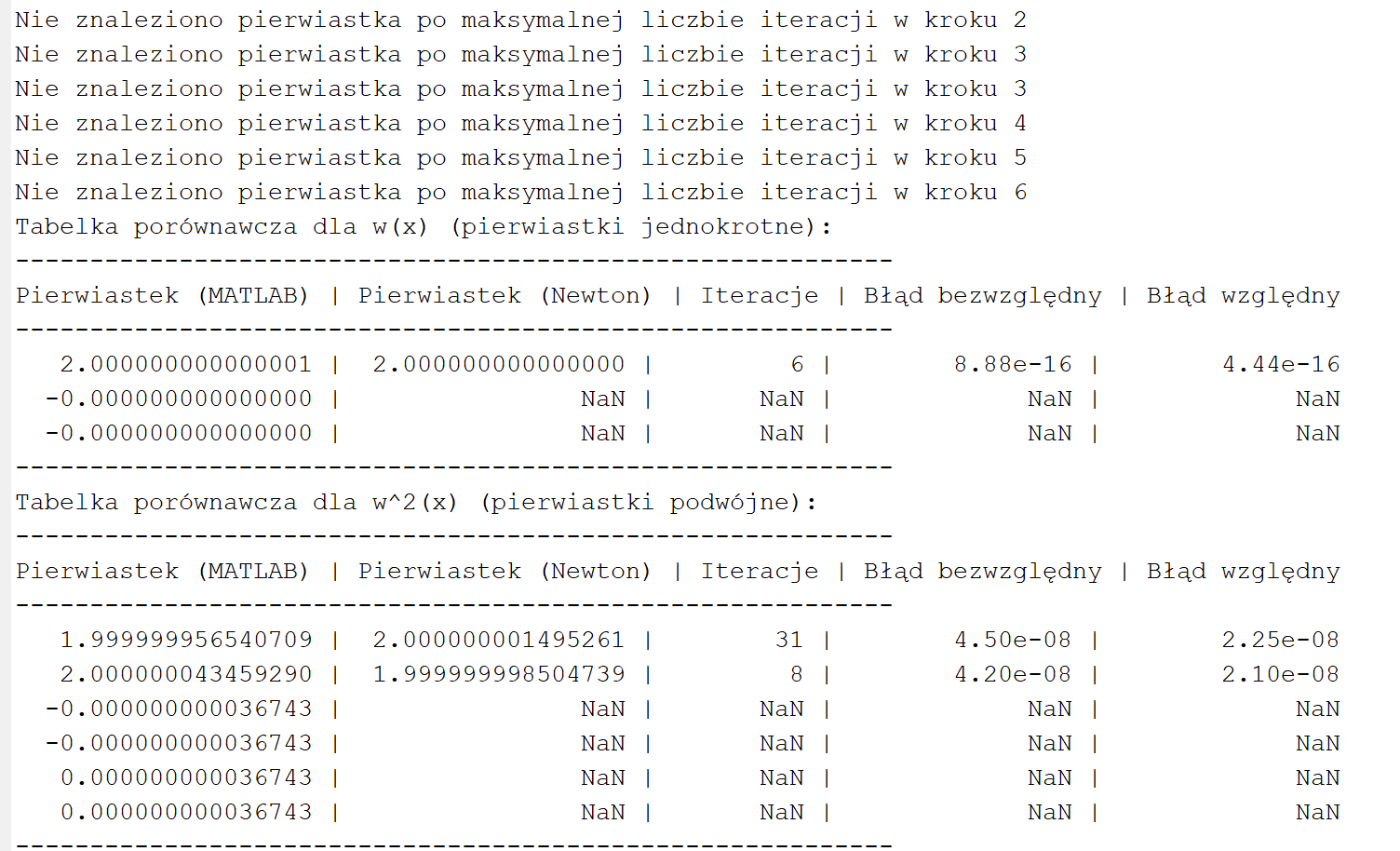


**Przypadek 3**

W kolejnym przypadku postanowiłam rozważyć funkcje, mające rozwiązania zespolone. To właśnie ten przykład doprowadził do poprawienia moich funkcji i wprowadzenia wartości NaN w przypadku nieznalezienia miejsca zerowego. Algorytm nadal szuka jednak maksymalnej możliwej liczby pierwiastków (najwyższa potęga x – 1).

W tym wypadku moja funkcja wypisuję więc NaN, a funkcja roots() część rzeczywistą pierwiastka zespolonego.

Przykład dla max\_iter = 1000000, tol = , = 4 – jednak wyniki są podobne bez względu na parametry:



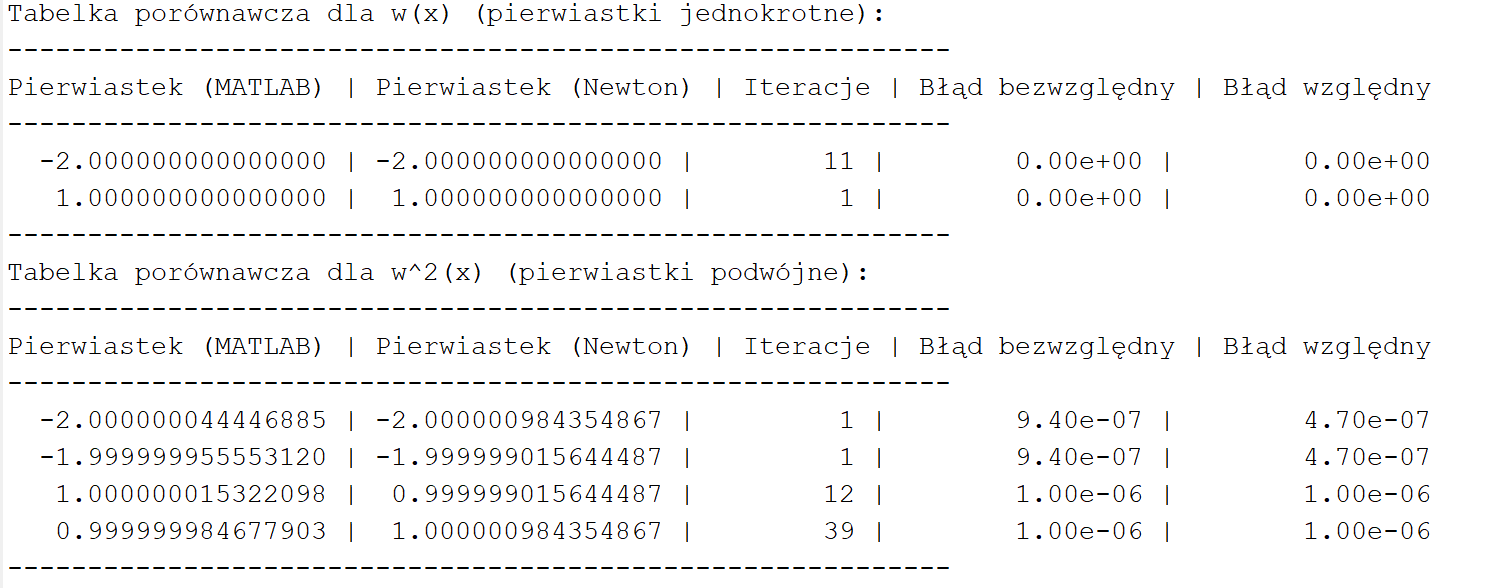
Wielomian podniesiony do kwadratu nie był w tym przypadku problemem, potrzebna była jedynie większa liczba iteracji i ponownie błąd przybliżenia był większy.

**Przypadek 4**

W następnym przykładzie chciałam świadomie doprowadzić do wyzerowania pochodnej. Ten przykład także pozwolił mi ulepszyć mój program, ponieważ początkowo nie obsługiwał on takich przypadków i kończył się błędem. Postanowiłam losować nowe przybliżenie w momencie, w którym pochodna jest równa 0. Początkowo testowałam również opcję losowania nowego przybliżenia, gdy pochodna jest mniejsza od tolerancji, jednak prowadziło to do dodatkowych błędów w przypadku zer wielokrotnych (prawdopodobnie dlatego, że pochodna tam jest właśnie często bardzo mała).

Wybrany wielomian:

W punkcie x = - jego pierwsza pochodna się zeruje, dlatego wybrałam dokładnie taki punkt początkowy. Teoretycznie problem powinien pojawić się już przy pierwszej iteracji.



Jak widać program w tym momencie bardzo dobrze już radzi sobie z taką sytuacją, błędy są podobne do innych przypadków. Liczba iteracji jest zwiększona, ponieważ nowa wartość początkowa jest losowana z przedziału [-1000, 1000], więc iteracje mogą rozpocząć się w dość odległym miejscu. Dodatkowo, przy każdym wywołaniu ich liczba jest inna, właśnie ze względu na losowość przy doborze początkowego x.

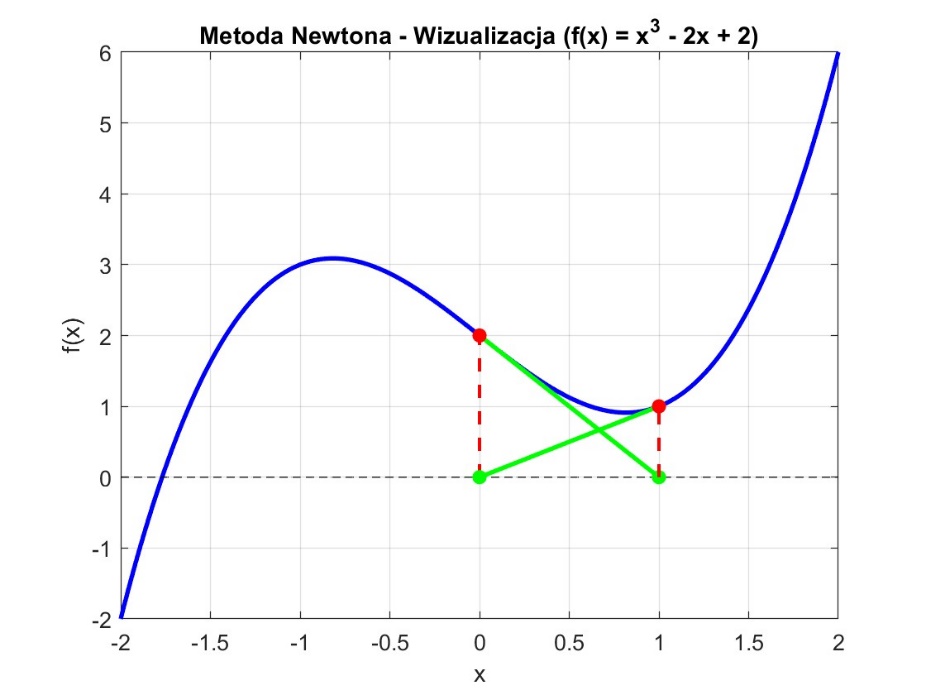
**Przypadek 5**

W kolejnym przypadku moim zamiarem było specjalne doprowadzenie do oscylacji i sprawdzenie zachowania się metody Newtona w tej sytuacji.

Wybrany wielomian:

Teoretycznie oscylacje powinny mieć miejsce pomiędzy oraz .

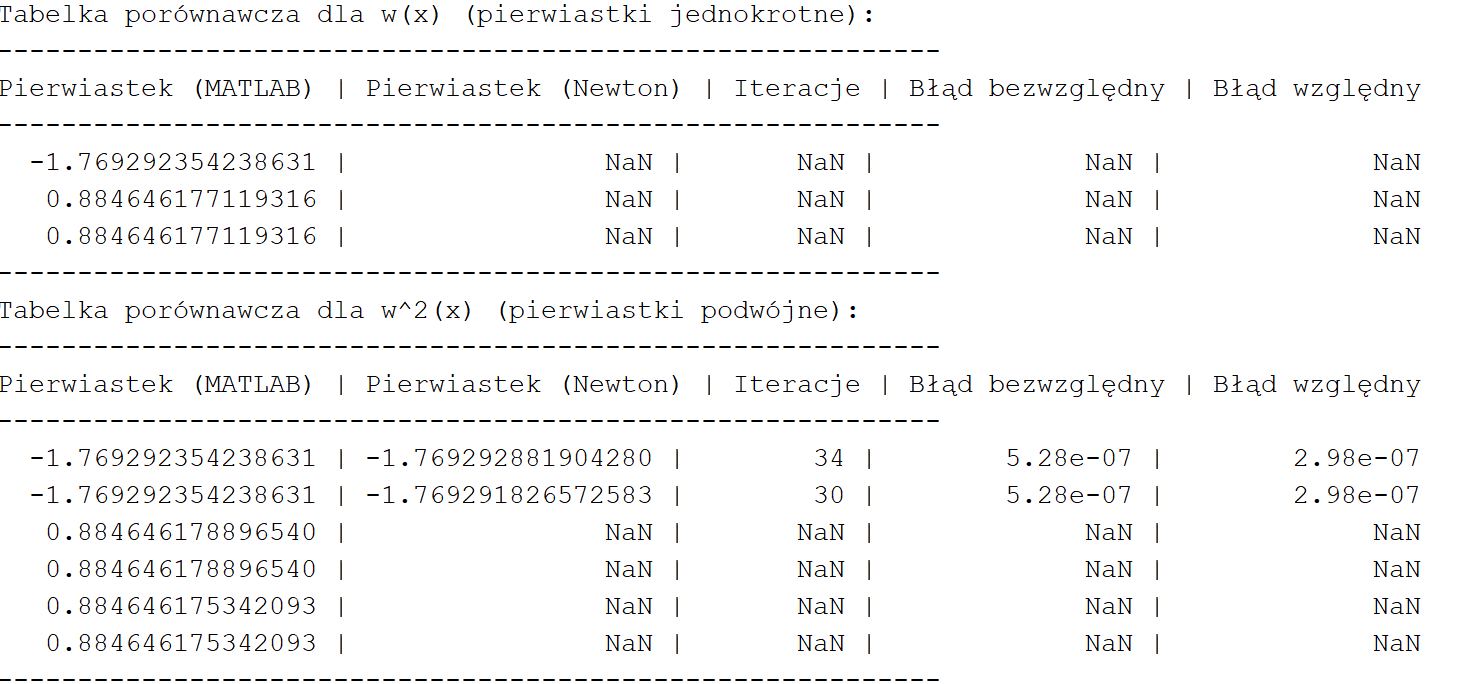
Wizualizacja oscylacji między punktami oraz :



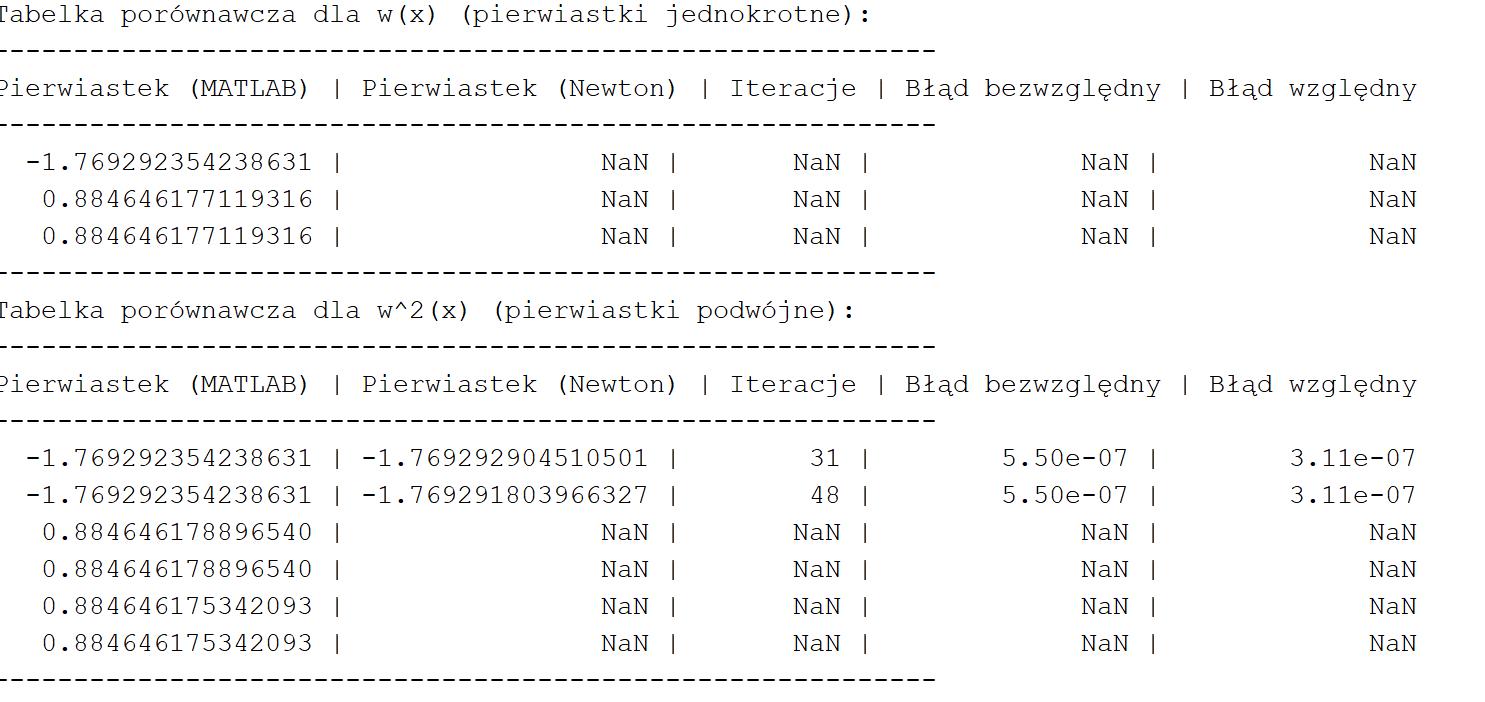
Sprawdziłam więc zachowanie funkcji dla . Zgodnie z przewidywaniami, w przypadku wielomianu niepodniesionego do kwadratu, milion iteracji nie starcza do znalezienia jedynego rzeczywistego rozwiązania. Zmiana argumentu na odległy od 0 oczywiście rozwiązuje ten problem. W przypadku wielomianu podniesionego do kwadratu oscylacje już nie występują, więc pierwiastek jest dość szybko znajdowany. Jest to przykład bardzo konkretnej sytuacji, w której wielomian z zerami wielokrotnymi zbiega szybciej, niż z jednokrotnymi.

Postanowiłam sprawdzić także, jak daleko od 0 musimy rozpocząć iteracje, aby oscylacje nie były problemem. Nawet dla otrzymujemy wartość NaN. Dla mamy już zbieżność, ale liczba iteracji dla zera jednokrotnego jest większa niż dla zera dwukrotnego.

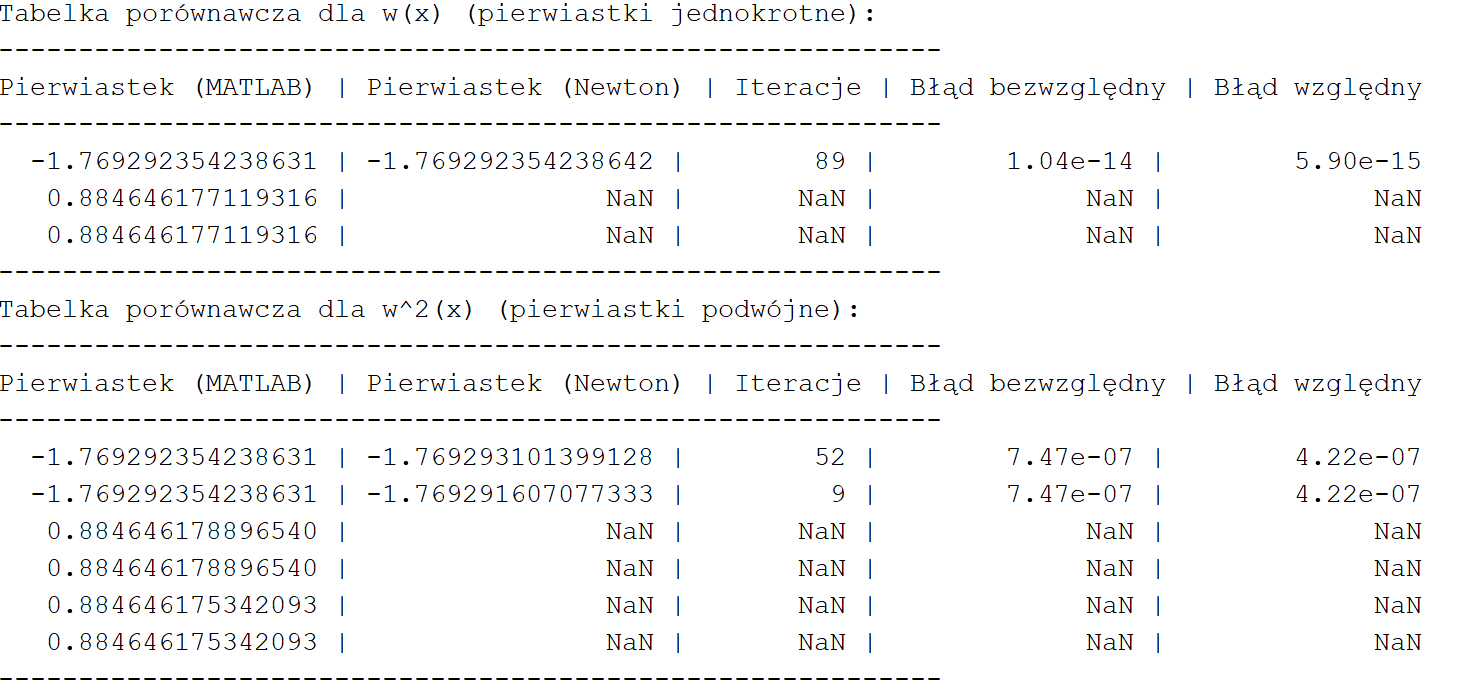
Dla



Dla



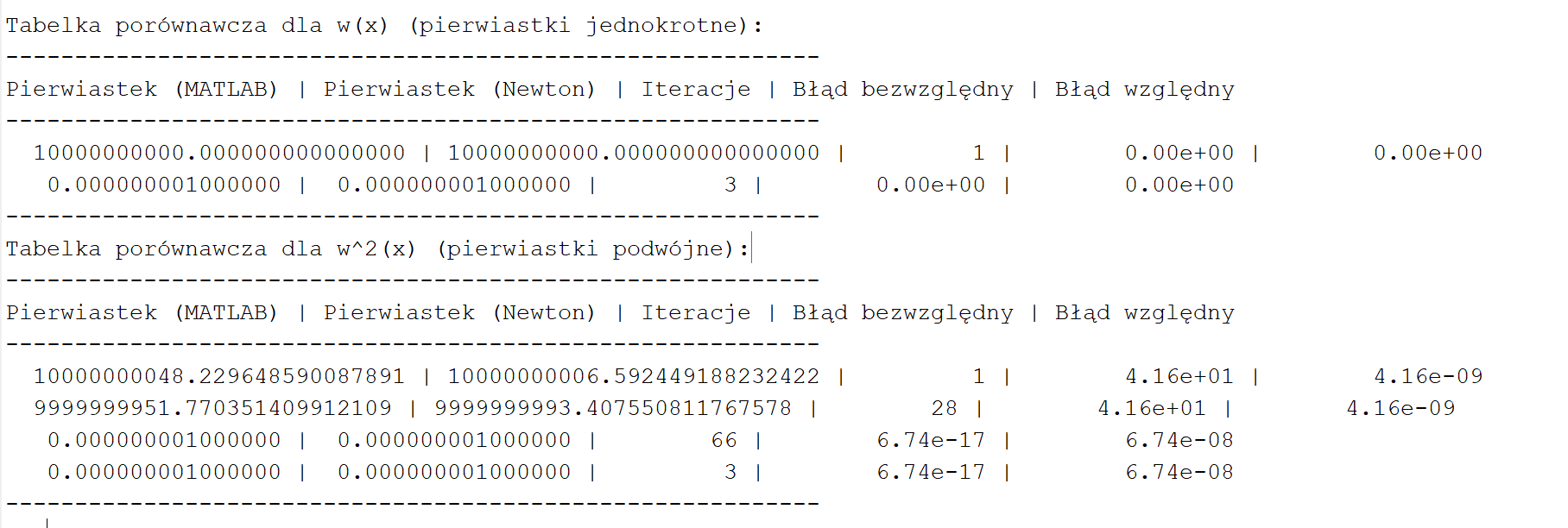
Dla



**Przypadek 6**

Ostatnim rozważanym przeze mnie przykładem będzie ten , w którym pierwiastki wielomianu są bardzo odległe od siebie, a ich rzędy wielkości są zdecydowanie inne. Wielomian który rozważę, to:

Za wybrałam 1000, jednak wyniki były podobne także przy innych przybliżeniach. Program bardzo dobrze poradził sobie w przypadku jednokrotnych pierwiastków, potrzebował tylko 4 iteracje na znalezienie z bardzo dużym przybliżeniem obu z nich. W przypadku zer dwukrotnych liczba iteracji wzrasta niezwykle znacząco, błędy są też większe, ale program nadal sobie radzi. Co więcej, błędy metody Newtona są często mniejsze niż te zwracane przez wbudowaną funkcję w MATLAB.



**Podsumowanie**

Analiza wszystkich przypadków prowadzi nas do kilku spostrzeżeń. Metoda Newtona zazwyczaj potrzebuje mniej iteracji, gdy ma do czynienia z zerami jednokrotnymi, co potwierdzają współczynniki zbieżności dla zer jedno- i wielokrotnych. W przypadku wielomianów o pierwiastkach jednokrotnych rzadziej też dochodzi do sytuacji, w których ciężko znaleźć , dzięki któremu znajdziemy wszystkie miejsca zerowe wielomianu. Musimy także liczyć się z większymi błędami względnymi i bezwzględnymi, gdy nasz wielomian posiada pierwiastki wielokrotne.