Metody Numeryczne [MNUM]
Projekt 2.40

Tomasz Pawlak, Informatyka, 304104
Prowadzący: dr hab. inż. Andrzej Karbowski

Zad 1.

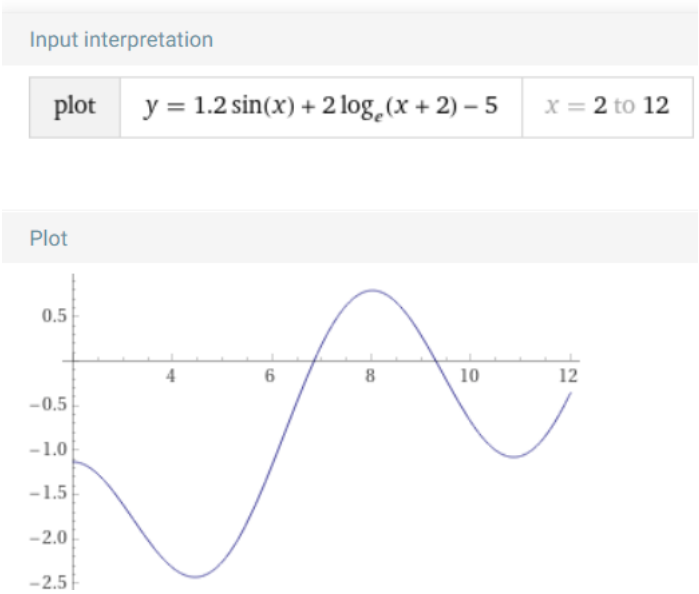
„Proszę znaleźć wszystkie pierwiastki funkcji

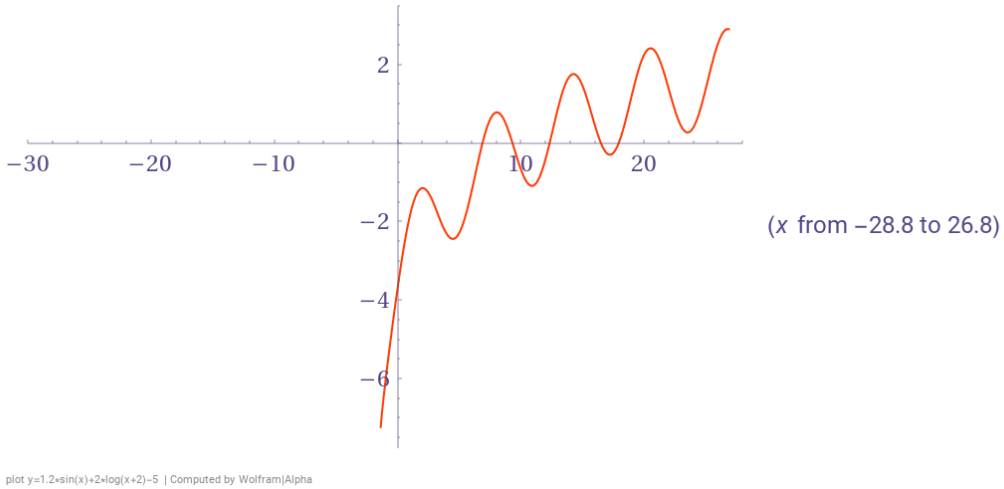
$$f(x) = 1.2 \sin(x) + 2 \ln(x + 2) - 5 \text{ w przedziale } [2, 12]$$

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

Opis algorytmów podstawowych:

Podstawowe algorytmy metody siecznych (zawarty w pliku secant.py) oraz metody Netwona (zawarty w pliku newton.py) są metodami iteracyjnymi, to znaczy ich wyniki są implikowane przez w przypadku 1 – przedział początkowej izolacji pierwiastka, a w drugim punkt początkowy aproksymujący pierwiastek rozwiązania. Do poprawnego wyboru tych punktów można wykorzystać wykres funkcji. Ze względu na czytelność wykorzystałem tu narzędzie WolframAlpha dla podanej w poleceniu funkcji:





Wykorzystując Metodę siecznych musimy wystartować z racjonalnego przedziału izolacji czyli takiego, którego obu końce znajdują się po przeciwnych stronach prostej x=0. Metoda Newtona wykorzystuje przybliżenie pierwiastka zatem nie ma znaczenia czy wybierzemy punkt, którego wartość jest dodatnia czy ujemna, ważne by metoda nie utknęła w ekstremum lokalnym.

Metoda siecznych polega na prowadzeniu siecznych łączących wartości punktów na granicach przedziału izolacji oraz iteracyjnym poprawianiu go. Jeśli granice przedziału izolacji oznaczymy jako a oraz b to nowy punkt możemy obliczyć zgodnie ze wzorem 6.6 str.139 podręcznika przekształconym dla naszych potrzeb i zwiększonej czytelności:

$$x_{n+1} = \frac{a * f(b) - b * f(a)}{f(b) - f(a)}$$

Metoda jest zbieżna tylko lokalnie co oznacza, że wybranie zbyt obszernego przedziału izolacji może skończyć się niepowodzeniem dla metody.

Implementacja i wywołanie Metody Siecznych

Funkcja podstawowa implementująca poszukiwanie pierwiastka funkcji jednej zmiennej metodą siecznych znajduje się w pliku `secant.py`. Został zaimplementowana według powyższego opisu.

```
tic;
i = 0; xf=a;
while abs(f(xf)) > delta && i < imax
    xf = (a*f(b)-b*f(a))/(f(b)-f(a));
    a = b;
    b = xf;
    i = i + 1;
end
texe=toc; iexe=i;
ff=f(xf);
```

Tak jak w przypadku dostarczonej implementacji metody stycznych Newtona wywołujemy ją z parametrami takimi jak: funkcja dana jako wyrażenie, przedział izolacji pierwiastka, dokładność do jakiej powinien zbiegać nasz algorytm oraz maksymalna liczba iteracji.

W przypadku naszej funkcji możemy określić dwa przedziały izolacji odpowiadające dwóm miejscom zerowym. Wywołując dla drugiego z nich (pierwiastek o większej wartości) zastosujmy pierw szeroki zakres [-1 25], wynik jest niepoprawny:

```
>> [xf, ff, iexe, texe] = secant(@(x)1.2*sin(x)+2*log(x+2)-5, -1, 25, 1e-8, 15)

xf =

 -14.2789 + 0.0000i


ff =

  -1.1722 + 6.2832i


iexe =

    15


texe =

    0.0064
```

Algorytmowi nie udało się uzyskać zbieżności w przedziale w związku z czym po upływie 15 operacji, które przewidziałem dla niego jako dopuszczalne wynik jest liczbą zespoloną, zawężając przedział do racjonalnego [8 11] uzyskamy rezultat, który możemy uznać za dopuszczalny:

```
>> [xf, ff, iexe, texe] = secant(@(x)1.2*sin(x)+2*log(x+2)-5, 8, 11, 1e-8, 15)

xf =

    9.2990


ff =

  -5.5955e-14


iexe =

     5


texe =

   2.9370e-04
```

W pierwszym punkcie wybrano przedział, w którym funkcja posiada wiele maksimów oraz minimów lokalnych, a także kilka miejsc zerowych. Wybór mniejszego przedziału pozwala uzyskać zbieżność.

Metoda stycznych Newtona dostarczona w solwerze z pliku `newton.m` ze strony przedmiotu różni się od mojej implementacji tym, że wykorzystuje aproksymacje funkcji jej liniowym przybliżeniem wynikającym z uciętego rozwinięcia w szereg Taylora w punkcie będącym danym przybliżeniem pierwiastka x_n. Iteracyjna zależność pomiędzy nowym, a starym przybliżeniem zależy od pochodnej wartości funkcji w punkcie x_n i prezentuje się następująco:

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

Zbieżność metody Newtona jest również lokalna, zatem od wyboru punktu początkowego również będzie zależeć czy algorytm poradzi sobie ze znalezieniem rozwiązania. Zbieżność jest jednak kwadratowa – lepsza od metody stycznych. Zatem jeśli wybierzemy odpowiedni punkt x_n metoda powinna poradzić sobie szybciej (w mniejszej liczbie iteracji), aby poprawić rozwiązanie do spełnienia tolerancji.

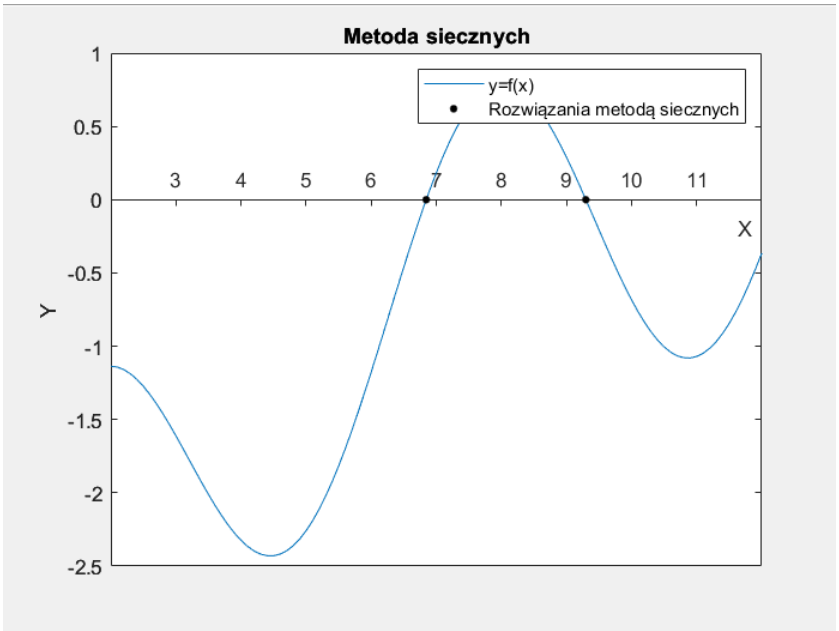
Metoda Newtona osiąga najlepszą efektywność w przypadku dużego spadku krzywej $f(x)$ w otoczeniu danego pierwiastka. W przypadku przeciwnym (gdy mamy mały skok w otoczeniu pierwiastka, czyli stosunkowo małą wartość $f'(x)$) zaleca się wybrać inną metodę.

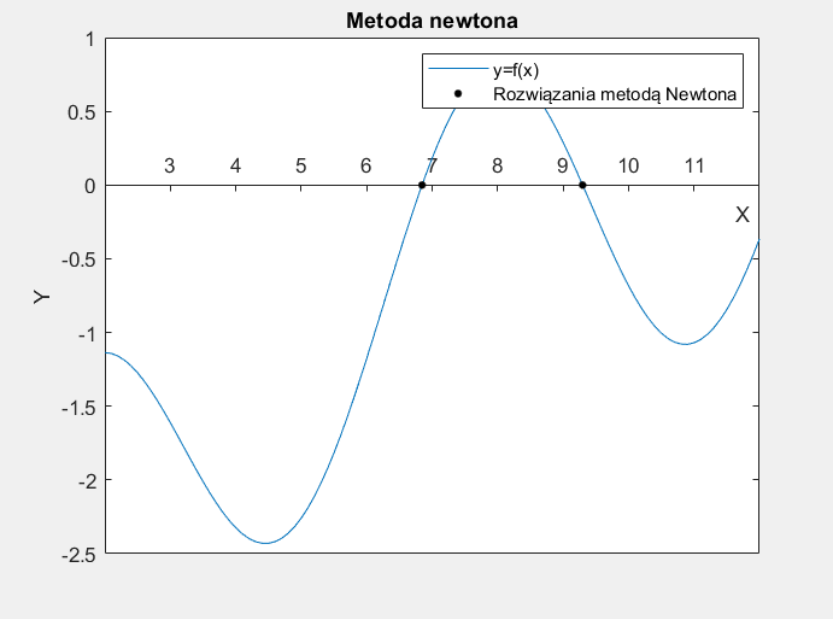
Opis algorytmu liczącego wszystkie pierwiastki z przedziału

W pliku `z1_all.m` zawarłem porównanie działania obu algorytmów oraz możliwość uruchomienia ich w celu znalezienia wszystkich miejsc zerowych z przdziału. Metoda zwraca dwie macierze wyników dla obu algorytmów. W przypadku znalezienia wyniku spoza przedziału funkcja informuje nas o nie znalezieniu rozwiązania. Dla funkcji danej w treści polecenia poprawnym wywołaniem będzie np.

```
>> [tabS, tabN, t] = z1_all(@(x)1.2*sin(x)+2*log(x+2)-5,[2 12],[6 8 8 10],[7 9],1e-8,15)

tabS =

    6.8458    0.0000
    9.2990    0.0000


tabN =

    6.8458   -0.0000
    9.2990   -0.0000


t =

   1.0e-03 *

    0.1055
    0.2015
```

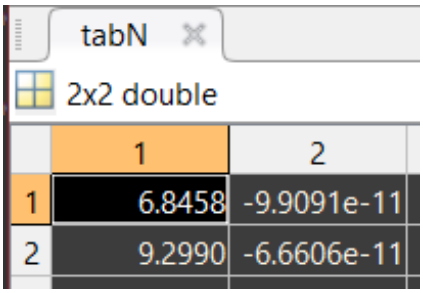
Dla metody siecznych wybrałem tutaj przedziały początkowe 6-8 oraz 8-10 ze względu na wizualizacje funkcji wykonaną w pierwszym paragrafie tego polecenia. Punktami przybliżeń dla metody Newtona zostały x=7 oraz x=9. Metoda generuje również wizualizację funkcji w przedziale [2 12], czyli zadanym przedziale wyszukiwania pierwiastków

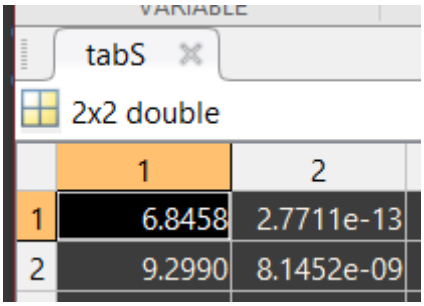




Wyjściem solwera jest również czas obliczania obu przybliżeń. Metoda Newtona poradziła sobie z zadaniem w dwukrotnie większym czasie.

Oba pierwiastki w Metodzie Newtona i Metodzie Siecznych spełniają dokładność założoną przez tolerancję. Po otworzeniu macierzy w arkuszu kalkulacyjnym widzimy, że dokładność obu pierwiastków w metodzie Newtona jest podobna. Natomiast w metodzie Siecznych drugi posiada dokładność 4 rzędy wielkości gorszą:





Porównanie wydajności algorytmów dla jednego miejsca zerowego

Plik `zad1.m` oraz zawarta w nim funkcja umożliwia nam generację macierzy kolejnych iteracji w celu znalezienia jednego zera obiema metodami. Aby przeanalizować wyniki poprzedniego wywołania uruchomimy ją dla pierwiastka $x \approx 9.2990$ na takich samych jak poprzednio przedziałach –[2 12](przedział obserwacji) [8 10](przedział początkowy dla metody siecznych) oraz punktu $x_n \approx 9$ dla metody Newtona.

```
>> [tabS, tabN, imaxs, imaxn]  = z1(@(x)1.2*sin(x)+2*log(x+2)-5,[2 12],8, 10,9,1e-8,15)
Secant (x,y)=9.298965e+00,8.145216e-09
Newton (x,y)=9.298965e+00,-6.660628e-11

tabS =

    9.0741    0.2214
    9.3008   -0.0019
    9.2989    0.0001
    9.2990    0.0000


tabN =

    9.3185   -0.0198
    9.2990   -0.0000
    9.2990   -0.0000


imaxs =

     4

imaxn =

     3
```

Metoda Newtona wykonała jedną operację mniej, a mimo to znalazła pierwiastek z większą dokładnością. Wyniki kolejnych iteracji w arkuszu kalkulacyjnym prezentują się następująco:

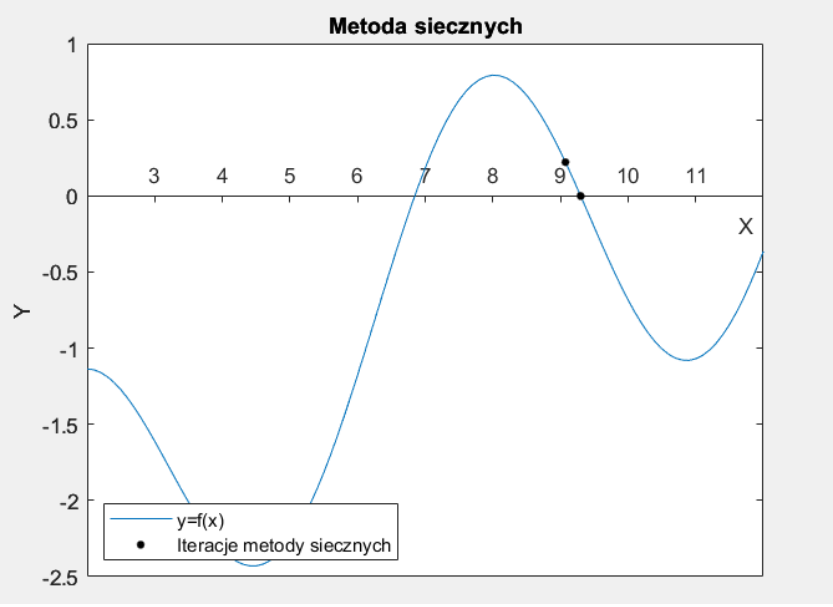
Dla **Metody Siecznych:**

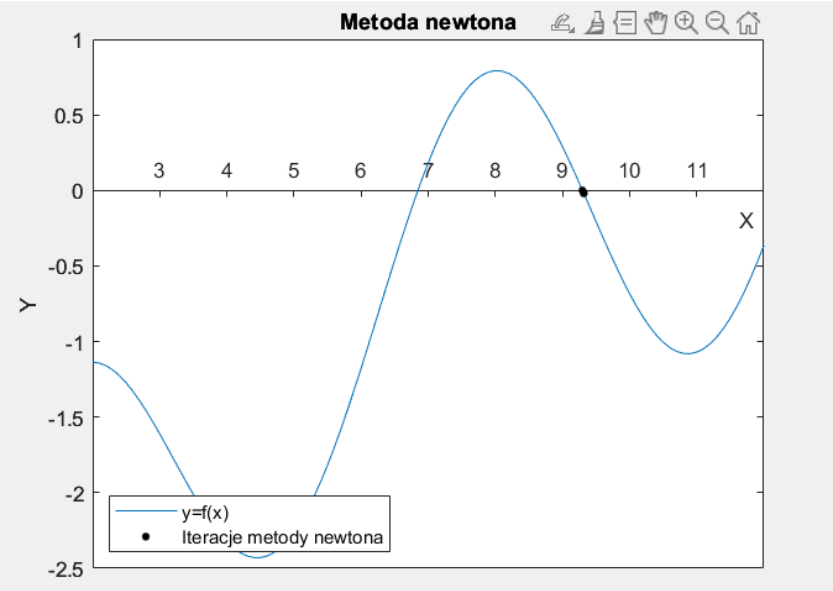
	1	2
1	9.074140673850450	0.221421108060397
2	9.300807287982238	-0.001867625087742
3	9.298911409721219	5.414298529782258e-05
4	9.298964823299190	8.145216057187099e-09

Dla **Metody Stycznych (Newtona):**

	1	2
1	9.318508592842512	-0.019838018089340
2	9.298993140620560	-2.869174717545775e-05
3	9.298964831401568	-6.660627605015179e-11

Większa wydajność metody Newtona jest najprawdopodobniej spowodowana wystarczającym stromym zboczem i wartością pochodnej w punkcie x_0. Wizualizacja kolejnych iteracji obydwu metod zostały zawarte na wykresach generowanych przy wywołaniu pliku `z1` z podanymi parametrami:





Podsumowanie:

- Metoda Newtona mimo, że wykonuje się dwukrotnie dłużej daje zdecydowanie lepsze rezultaty.

- Zbieżność metody siecznych jest mniejsza, przez co w przypadkach testowych potrzebowała więcej iteracji by osiągnąć wynik mieszczący się w progu tolerancji.

- Obie metody są zbieżne lokalnie, przez co wymagane jest przy użyciu każdej z nich odpowiednie szacowanie przedziałów oraz przybliżeń pierwiastków korzystając z wykresu funkcji. Możliwe sytuacje, w których metody zawodzą to wybranie przedziału, w którym pochodna zmienia znak lub przedziału z ekstremum lokalnym, gdzie metody mogą błądzić

Zad 2.

„Używając metody Muellera MM2 proszę znaleźć wszystkie pierwiastki wielomianu czwartego stopnia:

$$f(x) = a_4x^4 + a_3x^3 + a_2x^2 + a_1x^1 + a_0, \qquad [a_4\ a_3\ a_2\ a_1\ a_0] = [2\ 3\ -6\ 4\ 7]''$$

Opis algorytmu:

Druga Metoda Muellera wykorzystuje informacje o wartości wielomianu, pierwszej oraz drugiej jego pochodnej w przybliżeniu pierwiastka danego jako parametr metody i oznaczanego jako x_k. Z definicji paraboli $y(z)$ wynika, że przybliżenia zera można wyznaczyć za pomocą funkcji kwadratowej, której pierwiastek o mniejszym module jest iteracyjnym poprawianiem rozwiązania x_k:

$$z_{+,-} = \frac{-2f(x_k)}{f'(x_k) \pm \sqrt{(f'(x_k)^2 - 2f(x_k)f''(x_k)}}$$

$$x_{k+1} = x_k + z_{min,} \qquad \text{gdzie } z_{min} = \min\{z_+, z_-\}$$

Ze względu na naturę wielomianu – posiadanie dokładnie n pierwiastków, które mogą być rzeczywiste lub zespolone (będą wtedy występować parzyście jako liczby sprzężone ze sobą) oraz mogą być pojedyncze lub wielokrotne - należy wykorzystywać arytmetykę liczb zespolonych korzystając z MM2, aczkolwiek Matlab implementuje ją domyślnie.

Implementacja bazowego algorytmu MM2:

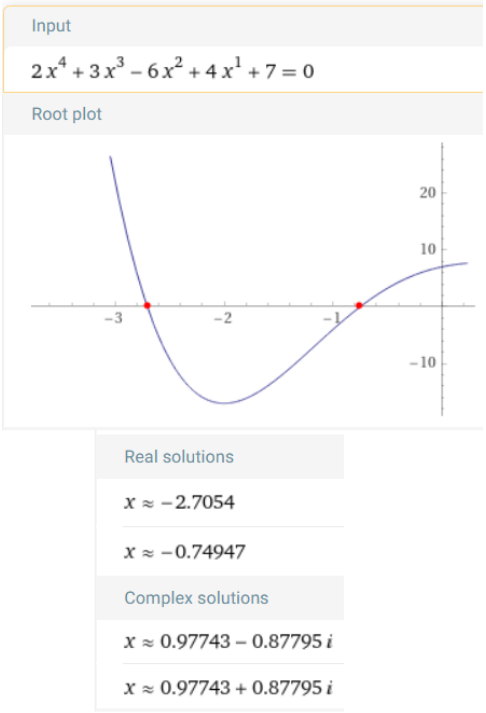
Zawarta w pliku `MM2.m` implementacja pozwala na znalezienie jednego pierwiastku – najbliższego dla punktu początkowego. Wejściem do solwera jest tablica współczynników wielomianu, zatem do obliczania pochodnych oraz wartości wielomianu zastosowałem metody `polyder` i `polyval`. Komentarze, którymi opatrzony jest kod pomagają zrozumieć mnogą ilość nieczytelnych wywołań ww. funkcji celem spełnienia rozwiązań wzorów z punktu wyżej:

```
%while (abs(f(x_k)) > 1e-8)
while abs(polyval(p,x_k)) > tol
    %sqrt_ = sqrt(df(x_k)^2-2*f(x_k)*d2f(x_k));
    sqrt_ = sqrt(polyval(polyder(p),x_k)^2-2*polyval(p,x_k)*polyval(polyder(polyder(p)),x_k));

    %z1 = -2*f(x_k)/(df(x_k)+sqrt_);
    z1 = -2*polyval(p,x_k)/(polyval(polyder(p),x_k)+sqrt_);

    %z2 = -2*f(x_k)/(df(x_k)-sqrt_);
    z2 = -2*polyval(p,x_k)/(polyval(polyder(p),x_k)-sqrt_);

    %Wybór elementu paraboli o mniejszym module
    %if abs(df(x_k)+sqrt_) > abs(df(x_k)-sqrt_)
    if abs(polyval(polyder(p),x_k)+sqrt_) > abs(polyval(polyder(p),x_k)-sqrt_)
        z_min = z1;
    else
        z_min = z2;
    end
    r = x_k+z_min;
    x_k = r;
end
```

Kod jest implementacją wzorów podanych w punkcie wyżej – jako `z1` i `z2` oznaczamy tutaj z_+, z_-. Wynik jest także poprawiany iteracyjnie od początkowej wartości, aż do osiągnięcia przez wielomian przybliżenia zera, dla którego $f(x)$ będzie mniejsze od tolerancji czyli zgodnie z poleceniem i przykładowym wywołaniem `1e-8`.

Przykład uruchomienia bazowego programu:

Zgodnie z wizualizacją wielomianu wykonaną w programie WolframAlpha oraz w pliku `plot_function.m`:

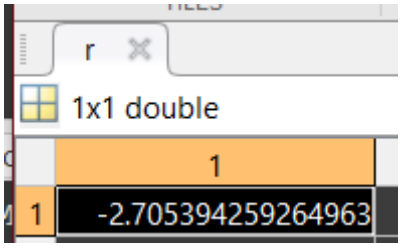


Powinniśmy spodziewać się pierwiastku naszego wielomianu w okolicy punktu $x = -2.5$. Zatem możemy uruchomić program z argumentami:

```
>> r = MM2([2 3 -6 4 7], -2.5, 1e-8)
```

Jak się okazuje metoda radzi sobie z tą próbą. Otrzymujemy wynik, który w przybliżeniu zostaje przedstawiony jako -2.7054, a po wyświetleniu w arkuszu kalkulacyjnym Matlaba prezentuje się jako -2.70539425926496:

```
>> r = MM2([2 3 -6 4 7], -2.5, 1e-8)

r =

   -2.7054

>> polyval([2 3 -6 4 7],r)

ans =

   1.1546e-14
```



Gdy obliczymy wartość wielomianu w tym punkcie okazuje się, że przybliżenie zera zgodnie z założeniami jest mniejsze od zadanej tolerancji – dokładnie o 6 rzędów wielkości.

Deflacja czynnikiem liniowym (prosty schemat Hornera) oraz wykorzystanie MM2 do znalezienia wszystkich pierwiastków wielomianu:

Program zawarty w pliku `z2.m` wykorzystuje `MM2.m` tylekrotnie ile pierwiastków posiada dany w argumencie wielomian:

```
poly_ = p;
% Stopień wielomianu jednoznacznie definiuje liczbę pierwiastków
deg=length(p)-1;
% Inicjacja tablicy pierwiastków
roots=zeros(deg,1);
for i=1:deg
    %Wykorzystanie metody MM2
    roots(i)=MM2(p, x_k, tol);

    n=length(p);
    new_p=zeros(n,1);
    % Deflacja czynnikiem liniowym - schemat Hornera
    for j = 2:n
        new_p(j)=p(j-1)+(new_p(j-1)*roots(i));
    end
    % Wielomian zostaje uproszczony po podzieleniu go przez (x-roots(i))
    p = new_p;
end
```

Deflacja czynnikiem liniowym i schemat Hornera zakłada tutaj wyznaczanie nowego wielomianu na podstawie współczynników `p(j-i)` i znalezionego zera `roots(i)` zgodnie ze wzorem:

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

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

Gdzie:
α – rozwiązanie znalezione w danej iteracji
q_i – współczynniki wielomianu obliczanego
a_i – współczynniki wielomianu pierwotnego

Zastosowałem prosty schemat Hornera. Jego wady to m.in. kumulacja błędów przybliżeń kolejnych zer doprowadzająca do błędów deflacji oraz sama metoda wyznaczania zer, która nie startuje od najmniejszych modułów, lecz od ustalonego punktu – co skutkuje zwiększeniem błędów numerycznych.

Dalsze linie kodu tego programu służą do generacji wykresu funkcji oraz zaznaczenia na niej przybliżeń pierwiastków rzeczywistych. Ze względu na złożoność i niską czytelność nie zastosowałem wykresów zespolonych pierwiastków.

Przykład uruchomienia pełnego programu:

```
>> roots = z2([2 3 -6 4 7], 0, 1e-8)

roots =

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

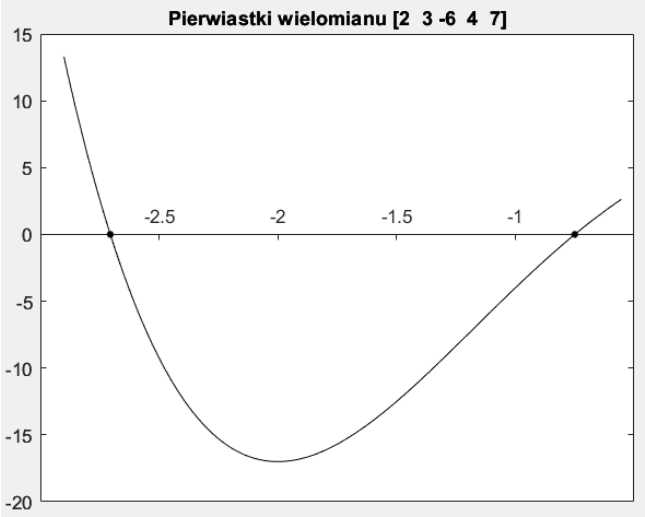
Jak widać pierwszy pierwiastek został wyznaczony jako zespolony, lecz w rozwinięciu widać że jest to kwestia typu danych zastosowanych dla całej tabeli:

	1
1	-0.749467969444041 + 0.000000000000000i
2	0.977431114354501 - 0.877953159884669i
3	0.977431114354501 + 0.877953159884669i
4	-2.705394259264962 + 0.000000000000000i

Wartości wielomianu w tych punktach możemy obliczyć wykonując operację:

```
>> polyval([2 3 -6 4 7], roots)

ans =

   1.0e-13 *

  -0.4352 + 0.0000i
  -0.4263 + 0.0044i
  -0.4086 - 0.0266i
  -0.3908 + 0.0000i
```

Wynik jest zgodny z oczekiwaniami, ponieważ wartości bezwzględne wszystkich liczb są tu mniejsze od zadanej tolerancji. Program rysuje też pierwiastki rzeczywiste co można zauważyć tutaj:



Wynik zgadza się z obserwacjami.

Próby wykonałem również dla punktu przybliżenia stosunkowo - znacznie oddalonego od punktów podejrzanych o bycie pierwiastkami – wyniki są podobnie zadowalające, porównałem je z wynikami dla poprzedniego punktu:

```
>> z2([2 3 -6 4 7], 0, 1e-8) - z2([2 3 -6 4 7], -1e6, 1e-8)

ans =

   1.0e-10 *

  -0.0000 + 0.0000i
   0.5714 - 0.0000i
   0.0000 + 0.0000i
   0.0000 + 0.0000i
```

W przypadku pierwiastków kilkukrotnych metoda sprawuje się niezbyt satysfakcjonująco:

Wykorzystałem przypadek $f(x) = (x-3)^5 =$ `[1 -15 90 -270 405 -243]`

```
>> roots = z2([1 -15 90 -270 405 -243], 0, 1e-8)

roots =

   2.9796 + 0.0115i
   2.9828 - 0.0159i
   3.0046 + 0.0230i
   3.0098 - 0.0213i
   3.0233 + 0.0027i
```

Już pierwszy pierwiastek odstaje od naszych oczekiwań. Żaden z pierwiastków nie jest jednak rzeczywisty. Wartości bezwzględne algorytmów również odstają od wartości 3 znacznie:

```
>> abs(roots)

ans =

    2.9796
    2.9828
    3.0047
    3.0098
    3.0233
```

Jednakże wartości wielomianu dla wyników cały czas spełniają zadaną tolerancję:

```
>> polyval([1 -15 90 -270 405 -243], roots)

ans =

   1.0e-08 *

   0.5898 + 0.3848i
   0.5918 + 0.3853i
   0.5888 + 0.3866i
   0.5920 + 0.3873i
   0.5902 + 0.3881i
```