ŁUKASZ STANISZEWSKI, NR INDEKSU: 304098

METODY NUMERYCZNE

PROJEKT 3



# Znajdowanie zer funkcji

Celem tego zadania było znalezienie wszystkich zer funkcji

w przedziale , 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 . 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 i w każdej iteracji kolejno:

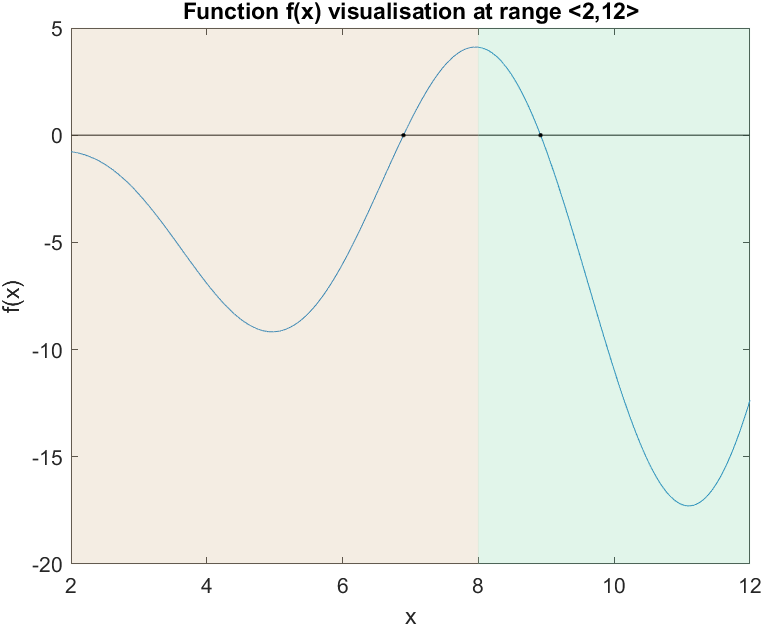
* Bieżący przedział zawierający zero funkcji jest dzielony na dwie połowy, a punktem środkowym podziału jest .
* Obliczany jest iloczyn , jeśli iloczyn ten jest ujemny, wtedy nowym bieżącym przedziałem staje się przedział , inaczej .

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. , gdzie – 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 - – wynika z przyrównania do zera sformułowanej lokalnej liniowej aproksymacji funkcji – w ten sposób, po odpowiednich przekształceniach, otrzymujemy zależność iteracyjną wyprowadzającą wzór na :

Realizację zadania rozpocząłem od narysowania funkcji i wyznaczenia przedziału izolacji dla poszczególnych pierwiastków. Na podstawie niżej zamieszczonego wykresu można zauważyć, że funkcja przyjmuje wartość **0** w dwóch miejscach, dodatkowo naniosłem na wykres przedziały, w których pierwiastki są izolowane – dla mniejszego z nich będzie to przedział (kolor kremowy), natomiast dla większego – przedział (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 **,** maksymalna liczba iteracji, dokładność ; a także dla drugiego z pierwiastków z parametrami: przedział początkowy: **,** maksymalna liczba iteracji , dokładność .

>> 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 , maksymalna liczba iteracji**,** dokładność ; a także dla drugiego z pierwiastków z parametrami: punkt początkowy: , maksymalna liczba iteracji , dokładność .

>> 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 oraz 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 |  |  |  |  |  |
| Bisekcji |  |  |  |  |  |
| Newtona |  |  |  |  |  |
| Newtona |  |  |  |  |  |
| Bisekcji |  |  |  |  |  |
| Bisekcji |  |  |  |  |  |
| Newtona |  |  |  |  |  |
| Newtona |  |  |  |  |  |

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 oraz – 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 lub .

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 z ilorazem zbieżności , gdzie metoda Newtona jest zbieżna kwadratowo, bo jej rząd zbieżności – co oznacza, że jest ona zdecydowanie szybsza.

# Znajdowanie pierwiastków wielomianu

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

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

W tym zadaniu, koniecznym było znalezienie zer wielomianów, tzn. takich -ów, dla których . Istotnym jest zwrócenie uwagi na fakt, że wielomian -tego stopnia, czyli będący postaci

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:

gdzie 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**.** , gdzie – założona dokładność rozwiązania.

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

oraz odwrotnego schematu Hornera (z założeniem ), opisywanego wzorami

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

* – wyznaczamy zgodnie z algorytmem Hornera prostym,
* – wyznaczamy zgodnie z odwrotnym algorytmem Hornera.

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

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 z maksymalną liczbą iteracji oraz dokładnością znajdowania zera rzędu . 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** |
|  |  |  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

Przedstawione wyżej wyniki pokazują, że metoda Laguerre’a jest metodą bardzo szybką – w przypadku tego wielomianu i dokładności rzędu 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.