Informatica
Numerieke wiskunde
Prof. Koen Van Den Abeele
Andries Vansweevelt

Verslag project numerieke wiskunde

Academiejaar 2017-2018

Thomas Bamelis en Michiel Provoost

Inleiding

Dit verslag behandeld de vragen en antwoorden gesteld in het eind-project numerieke wiskunde 2017. De hoofdvraag in dit project is hoe wortels/nulpunten van een willekeurige veelterm kunnen gevonden worden met de methode van Newton-Raphson en de methode van Bairstow. Verder behandelt dit verslag de performantie en fouten op beide algoritmen en sluit af met deze met elkaar te vergelijken. Beide methoden werden geïmplementeerd in matlab en getest op 10 gegeven veeltermen, met complexe en reële nulpunten. Verder werden verschillende opgedragen ease-of-use features geïmplementeerd, zoals een figuur plotten indien enkel de coëfficiënten gegeven.
Ook word kort aandacht besteed aan een analyse van een bepaald geval, waarin bekeken wordt welke startwaarden voor beide methoden tot welke (of geen) nulpunten leiden.

1 Newton-Raphson

Dit hoofdstuk bespreekt de implementatie van Newton-Raphson, waarin onder andere de methode van Horner, samen met een analyse van de performantie en fouten.

1.1 Evaluatie veelterm en zijn afgeleide m.b.v. het Hornerschema

Hier wordt opdracht 1 besproken.

Met het uitgebreide Hornerschema, besproken in [1], kan een veelterm geëvalueerd in een gegeven punt. Daarna kan met een deel van de oplossing verder gewerkt worden om 1-per-1 de afgeleiden van de veelterm te evalueren in hetzelfde punt.

1.1.1 Implementatie

Dit werd als volgt geïmplementeerd:

```
1  function y = my_polyval( p, x, m)
2  %my_polyval Geeft de functiewaarde in c van de eerste m afgeleiden van p
3  %terug, via het uitgebreid schema van Horner.
4  %
5  %   Signatuur: y = my_polyval( p, x, m)
6  %
7  %   @param p
8  %       De veelterm die geevalueerd zal worden, voorgesteld door zijn
9  %       coefficienten als vector, met de hoogste graad term als eerste
10 %       element enzovoort. Dit is een 1 x n vector.
11 %
12 %   @param x
13 %       Het punt waarin de functies zullen geevalueerd worden.
14 %
15 %   @param m
16 %       Het aantal keer dat p moet afgeleid worden.
17 %
18 %   @return y
19 %       De vector die de functiewaarden in x van de 0'de tot de m'de
```

```
20 %        afgeleide van p bevat.
21 %
22
23
24
25
26 graadStartPlusEen = size( p );
27 graadStartPlusEen = graadStartPlusEen(2);
28
29
30 y = zeros( m + 1, 1);
31
32 % p gebruiken, want minder geheugen gebruikt (call by value)
33
34
35 for afgeleide = 1:m+1
36
37     % 1ste niet kopieren want die zit al in p.
38
39     for coefficient = 2:(graadStartPlusEen - afgeleide+1)
40
41         %De coefficienten aanpassen. Alle ellementen van p voor
42         %p(coefficient) zijn al aangepast, dus de 2de term
43         %in onderstaande formule ook.
44
45         p(coefficient) = p(coefficient) + p(coefficient - 1)*x;
46
47     end
48
49     %De functiewaarde in x van deze afgeleide berekenen en opslaan.
50     %y begint bij 1 en eindigt bij m + 1, dus 1 verder zetten.
51
52     y(afgeleide) = factorial(afgeleide-1) * p(graadStartPlusEen - afgeleide
           +1);
53
54
55
56 end
57 end
```

Zoals in [1] wordt aangetoond, worden de m eerste afgeleid (inclusief $f^{(0)}$) gevonden, door m keer het schema van Horner toe te passen op p waarna de functiewaarde van i'de afgeleide gegeven wordt door $k!b_{n-k}^{k+1}$. Merk op dat in de 'afgeleide' iteratie, de 'afgeleide' - 1 'ste afgeleide behandeld wordt om beter matlabs vector nummering in te kunnen spelen.
Het 1'ste element van p, dus de coefficient van de hoogste graad term, wordt nooit aangepast omdat volgens Horner $a_0 = b_0$. Daarna worden de rest van de coefficienten aangepast volgens $b_i = a_i + b_{i-1} * x$. Er wordt rekening gehouden met het feit dat de graad van de veelterm bij iedere iteratie van de afgeleide met 1 verlaagd wordt.

Een bijkomende opdracht was om zo weinig mogelijk geheugen te gebruiken. Om hier mee om te gaan werd in de vector p zelf gewerkt. Dit is geen enkel probleem omdat je nooit de oude vorige elementen nodig hebt en omdat matlab call-by-value is. Dit wil zeggen dat matlab p kopieert en de kopie meegeeft aan de functie waar mee gewerkt mag worden zonder dat de originele p wordt aangepast. Het enige

geheugen dat extra wordt aangemaakt, is het geheugen gebruikt om de functie waarden terug te geven een de oproeper van de functie. Dit wil zeggen dat het geheugengebruik van de functie van orde $\Theta(n+m)$ is, met n de graad + 1 van p en m het aantal keer dat afgeleid moet worden.

1.1.2 Fouten

De stabiliteit van deze methode komt op 2 manieren in gedrang:

- In lijn 45, als p(coefficient) en p(coefficient - 1)*x bijna even groot zijn en tegengesteld teken hebben.
- In lijn 52, als de 'afgeleide' afgeleide zeer groot word, zullen afrondingsfouten groter worden door de faculteit.

1.2 Een nulpunt vinden met Newton-Raphson

Hier wordt opdracht 2 besproken.

Indien een willekeurig veelterm gegeven wordt en een startwaarde, kan via de formule van Newton-Raphson ($x_{i+1} = x_i + \frac{p(x_i)}{p'(x_i)}$) een nieuwe x gevonden worden, die dichter bij het nulpunt ligt als er convergentie optreedt.

1.2.1 Implementatie

```
1  function w = newtonraphson( p, start, tol)
2  %newtonraphson Geeft het nulpunt van p terug die gevonden word via de
3  %startwaarde start met fouten tolerantie tol (standaard 10^-6),
4  %via de methode van Newton-Raphson.
5  %
6  %   Signatuur: w = newtonraphson( p, start, tol)
7  %
8  %   @param p
9  %       De veelterm die geevalueerd zal worden, voorgesteld door zijn
10 %       coefficienten als vector, met de hoogste graad term als eerste
11 %       element enzovoort. Dit is een 1 x n vector.
12 %
13 %   @param start
14 %       De startwaarde nodig om Newton-Raphson aan te vatten.
15 %
16 %   @param tol
17 %       De tolerantie van de fout, indien niet meegegeven 10^-6.
18 %
19 %   @return w
20 %       Het nulpunt gevonden aan de hand van de starwaarde binnen de
21 %       tolerantie tol indien de fouttolerantie van de gevonden waarden
22 %       kleiner of gelijk is aan tol, anders NaN.
23 %
24 %
25 %
26
27
28 % p mag geen 0de graads zijn.
29 s = size(p);
30 s = s(2);
31 if s <= 1
32     disp("Functie moet minstens graad 1 zijn!");
33     w = NaN;
```

```
34      return
35  end
36
37  %Check of tol gegeven is.
38
39  if nargin == 2
40      tol = 10^(-6);
41  end
42
43  % Initialiseer de startwaarden.
44
45  vorigeX = start;
46  huidigeX = start;
47
48  % De fout benadering initialiseren zodat deze in de while geraakt.
49
50  foutAprox = tol + 1;
51
52  iterator = 0;
53  while foutAprox > tol && iterator < 10000
54
55      vorigeX = huidigeX;
56
57      % Bereken p(vorigeX) en p'(vorigeX).
58
59      [ px, afgpx] = my_polyval( p, vorigeX, 1);
60
61
62      % Stabiliteit beschermen tegen een kleine afgeleide.
63
64      if afgpx < 1
65
66          stabV = 100;
67
68      else
69
70          stabV = 1;
71
72      end
73
74
75      % px gedeeld door afgpx
76
77      deling = px / ( afgpx * stabV );
78      deling = deling * stabV;
79
80      % Bereken de huidigeX
81
82      huidigeX = vorigeX - deling;
83
84
85      foutAprox = abs(huidigeX - vorigeX);
86      iterator = iterator + 1;
87  end
88
```

```
89 % Checken of de fout benadering kleiner is dan tol.
90 if abs(huidigeX - vorigeX) > tol
91     w = NaN;
92     return
93 end
94
95 w = huidigeX;
96
97 end
```

Het iteratief berekenen van de volgende x, wordt gestopt op 2 manieren:

1. Door de 'zwakke' stopwaarde, als de nieuwe x en de vorige x in absolute waarde kleiner zijn dan de meegegeven tolerantie.
   In dit geval wordt de functie aanroep als succesvol gezien.

2. Na 10000 iteraties wordt de lus verplicht afgebroken, om niet in een oneindige lus terecht te komen.

Bij iedere iteratie wordt de 'zwakke' fout opnieuw berekent en gecheckt of aan de voorwaarde voldaan wordt. Op het einde van het algoritme wordt nog een laatste check gedaan om te zien of er voldaan is aan de meegegeven tolerantie. Indien wel wordt de gevonden benadering van het nulpunt meegegeven, anders NaN. Er werd nog een extra geïmplementeerd om de stabiliteit te bevorderen die in de volgende sectie besproken wordt.

1.2.2 Fouten

De grootste fout wordt verkregen door een kleine $p^{'}(x_i)$ in de berekening van een nieuwe x, door de deling in lijn 77. Om hiermee om te gaan wordt indien $p^{'}(x_i)$ kleiner dan 1 wordt, in de deling een $p^{'}(x_i)$ vermenigvuldigt wordt met 100 om het verderzetten van de fout op $p^{'}(x_i)$ te verzachten. Hierna wordt de volledige quotiënt opnieuw vermenigvuldigt met 100. De 100 kan groter genomen worden, maar dan worden de afrondingsfouten voor een grote $p(x)_i$ ook groter.

1.3 Deflatie met Horner

Hier wordt opdracht 3 besproken.

Nadat een nulpunt met Newton-Raphson gevonden wordt, kan de (x - gevondenNulpunt) factor weggedeeld worden met het alom bekende schema van Horner. Er moest ook voorzien worden dat als het gegeven nulpunt complex is, zijn complex toegevoegde ook weggedeeld wordt.

1.3.1 Implementatie

```
1  function q = horner( p, x)
2  %horner voert een deflatie stap uit op p zodat p een graad lager wordt
3  %door de factor met nulpunt x, en indien x complex is ook zijn complex
4  %toegvoegde, weg te delen via de methode van Horner.
5  %
6  %   Signatuur: q = horner( p, x)
7  %
8  %   @param p
9  %       De veelterm die de deflatie ondergaat, voorgesteld door zijn
10 %       coefficienten als vector, met de hoogste graad term als eerste
11 %       element enzovoort. Dit is een 1 x n vector.
12 %
13 %   @param x
14 %       Het nulpunt die zal weggedeeld worden in p.
```

```
15 %
16 %    @result q
17 %        De veelterm die het resultaat is van de deflatie van p, voorgesteld
18 %        door zijn coefficienten als vector, met de hoogste graad term als
19 %        eerste element enzovoort. Dit is een 1 x n vector.
20 %
21
22
23 % Initialiseer q.
24 grootteP = size(p);
25 grootteP = grootteP(2);
26 q = zeros(1, (grootteP - 1));
27
28 %Stop als p te klein is.
29
30 if grootteP <= 1
31    disp("Graad p is te laag!");
32    q = NaN;
33    return
34 end
35
36 if grootteP <= 2 && not( isreal(x) )
37    disp("Graad p is te laag voor een complex getal!");
38    q = NaN;
39    return
40 end
41
42 % Coefficienten van de hoogste graad termen van p en q zijn gelijk.
43
44 q(1) = p(1);
45
46 % Start en eindwaarde for loop elk 1 hoger voor gemak in matlab (rest niet
47 % nodig).
48 for coefficient = 2:(grootteP - 1)
49
50    q(coefficient) = p(coefficient) + q(coefficient - 1)*x;
51
52 end
53
54
55
56 % Complex toegevoegde ook wegdelen indien x complex is.
57 if not( isreal(x) )
58
59     x = conj( x );
60
61     % Kopieer q om verder te kunnen werken ermee.
62
63     a = q;
64
65     % q nog een graad lager initialiseren.
66
67     q = zeros( 1, grootteP - 2);
68
69     % Coefficienten van de hoogste graad termen van p en q zijn gelijk.
```

```
70
71      q(1) = a(1);
72
73      % Start en eindwaarde for loop elk 1 hoger voor gemak in matlab (rest
            niet
74      % nodig).;
75      for coefficient = 2:(grootteP - 2)
76
77        q(coefficient) = a(coefficient) + q(coefficient - 1)*x;
78
79      end
80
81  end
82
83
84  end
```

De implementatie is gelijk aan die van sectie 1.1 met 1 iteratie, waarbij de eerste n-1 coëfficiënten worden teruggegeven na de iteratie. Dit stelt dus een veelterm van 1 graad lager dan p voor. Dit geld echter enkel als het nulpunt reëel is. In dit geval word een veelterm die 2 graden lager is dan p teruggegeven, waaruit beide nulpunten op dezelfde manier zijn weggedeeld. Tijdens het wegdelen van het complex toegevoegde wordt dus met een input van 1 graad lager een output van 2 graden lager gegeven.

1.3.2 Fouten

De stabiliteit van deze methode lijd onder dezelfde problemen als 1.1:

- In lijn 50, als p(coefficient) en q(coefficient - 1)*x bijna even groot zijn en tegengesteld teken hebben.
- In lijn 77, als a(coefficient) en q(coefficient - 1)*x bijna even groot zijn en tegengesteld teken hebben.

1.4 Alle nulpunten vinden

Hier wordt opdracht 3 besproken.

Met de hiervoor besproken methoden werd zoals gevraagd een implementatie geschreven die alle nulpunten berekent van een gegeven veelterm. Deze methode vraagt om een veelterm, een startwaarde en een tolerantie die 10^{-6} is indien niet meegegeven. Er wordt een nulpunt bepaald met de methode van Newton-Raphson met de gegeven startwaarde, waarna dit nulpunt weggedeeld wordt met de methode van Horner. Er wordt met de nieuwe veelterm die door Horner werd teruggegeven verder gewerkt en er wordt opnieuw een nulpunt met Newton-Raphson bepaald met als startwaarde het vorige gevonden nulpunt. Dit wordt gedaan tot alle nulpunten van de veelterm gevonden worden.
Als Newton-Raphson niet convergeert, en dus NaN wordt teruggegeven, wordt de huidige functie geplot en om een nieuw startpunt gevraagd aan de gebruiker.
Uiteindelijk wordt een vector met alle nulpunten van p teruggegeven.

1.4.1 Implementatie

```
1  function ws = newtonraphsondef( p, start, tol)
2  %newtonraphsondef Geeft alle nulpunten van de veelterm p terug met fouten
3  %tolerantie tol (standaard 10^-6).
4  %
5  %   Signatuur: ws = newtonraphsondef( p, start, tol)
6  %
7  %   @param p
8  %       De veelterm die geevalueerd zal worden, voorgesteld door zijn
```

```
9  %       coefficienten als vector, met de hoogste graad term als eerste
10 %       element enzovoort. Dit is een 1 x n vector.
11 %
12 %   @param start
13 %       De startwaarde nodig om Newton-Raphson aan te vatten.
14 %
15 %   @param tol
16 %       De tolerantie van de fout, indien niet meegegeven 10^-6
17 %
18 %   @return ws
19 %       de nulpunten gevonden aan de hand van de starwaarde binnen de
20 %       tolerantie tol indien de fouttolerantie van de gevonden waarden
21 %       kleiner of gelijk is aan tol, anders NaN.
22 %
23
24
25 %ZORG DAT JE ALLES VIND, DOE DIT DOOR NIEUWE STARTWAARDEN TE DOEN INGEVEN
26 %INDIEN NIET ALLES WERD GEVONDEN, DOE DIT ZOALS AANGEGEVEN IN DE OPGAVE
27
28 %zeg dat ze desnoods een complexe startwaarde moeten geven,
29 %want enkel zo kun je complexe nulpunten vinden.
30
31
32 % p mag geen 0de graads zijn.
33
34 grootteP = size(p);
35 grootteP = grootteP(2);
36 if grootteP <= 1
37     disp("Functie moet minstens graad 1 zijn!");
38     ws = NaN;
39     return
40 end
41
42 % Check of tol gegeven is.
43
44 if nargin == 2
45     tol = 10^(-6);
46 end
47
48 % Initialiseer ws.
49
50 ws = zeros( 1, grootteP - 1);
51
52 % Maak de arrays aan om te plotten indien geen nulpunt gevonden word.
53 plotGrootte = 5000;
54 startPlot = -500;
55 eindPlot = 500;
56 X = zeros( plotGrootte, 1);
57 Y = zeros( plotGrootte, 1);
58 stapGrootte = (abs(startPlot) + abs(eindPlot)) / plotGrootte;
59
60 % Itereer aan de hand van het aantal gevonden nulpunten.
61
62 nbGevondenNulpunten = 0;
63
```

```
64  while nbGevondenNulpunten ~= grootteP - 1
65
66      if(size(p,2) > 2)
67          huidigNulpunt = newtonraphson( p, start, tol);
68      else
69          huidigNulpunt = -p(2)/p(1);
70      end
71      if (abs(imag(huidigNulpunt)) < tol)
72          huidigNulpunt = real(huidigNulpunt);
73      end
74
75      if ~isnan(huidigNulpunt)
76          %Nulpunt gevonden binnen de tolerantie:
77          %Deel nulpunt weg en voeg nulpunt toe aan ws
78
79          nbGevondenNulpunten = nbGevondenNulpunten + 1 ;
80
81
82          p = horner( p, huidigNulpunt);
83
84
85          ws(nbGevondenNulpunten) = huidigNulpunt;
86
87          %Indien het nulpunt complex is werd zijn toegevoegde ook
88          %weggedeelt:
89          if not(isreal(huidigNulpunt))
90              nbGevondenNulpunten = nbGevondenNulpunten + 1 ;
91              ws(nbGevondenNulpunten) = conj(huidigNulpunt);
92          end
93
94      else
95          % Geen nulpunt gevonden, laad nieuwe startwaarde
96
97          % Maak arrays om te plotten.
98
99          xWaarde = startPlot;
100
101         huidigeGrootteP = size(p);
102         huidigeGrootteP = huidigeGrootteP(2);
103
104         % Bereken alle koppels om te plotten.
105         for i = 1:plotGrootte
106
107             yWaarde = 0;
108             exponent = 0;
109
110             % Bereken de y waarde.
111             % Begin achteraan de lijst, dus met de constante.
112             for j = huidigeGrootteP:-1:1
113
114                 yWaarde = yWaarde + p(j) * xWaarde^(exponent);
115
116                 exponent = exponent + 1;
117
118             end
```

```
119
120                 X(i) = xWaarde;
121                 Y(i) = yWaarde;
122
123                 xWaarde = xWaarde + stapGrootte;
124             end
125
126             % Plot de functie.
127
128             plot(X,Y);
129
130             % Vraag om juiste input
131
132             input_hf = str2double(input('Geen nulpunt gevonden, geef nieuwe (
                    misschien complexe) startwaarde: ','s'));
133             while isnan(input_hf) || fix(input_hf) ~= input_hf
134                 input_hf = str2double(input('Geef een getal in: ','s'));
135             end
136
137             start = input_hf;
138
139         end
140
141
142 end
143 end
```

1.4.2 Fouten

Omdat deze functie op zich niet veel rekenwerk bevat, maar dit wordt doorgegeven aan de opgeroepen functie, kan hier niet veel over de fout gepraat worden.
Bij het plotten van de functie zijn kleine fouten niet zo belangrijk, omdat dit op een grote schaal gebeurt en omdat een plot sowieso nooit als betrouwbaar mag gezien worden.

2 Bairstow

De methode van Bairstow werkt door telkens bij iedere stap 2 nulpunten af te zonderen. Dit zorgt ervoor dat de methode nog sneller convergeert. Wel hebben we om dit efficiënt te doen verlopen het dubbelrijige Hornerschema nodig. Dit hoofdstuk bespreekt de implementatie van dat schema, de implementatie van het algoritme van Bairstow zelf en hoe we dan een algoritme implementeren die aan de hand van de methode van Bairstow alle nulpunten van een gegeven veelterm teruggeeft.

2.1 Evaluatie via dubbelrijige Horner

Om de 2 nulpunten effeciënt te kunnen evalueren is er een nieuwe functie geïmplementeerd die het dubbelrijige algoritme van Horner gebruikt. Dit laat ons toen om in $(2n+1)V$ en $2nO$ de beide te evalueren in plaats van $(4n-2)V$ en $(3n-2)O$ [1].

2.1.1 Implementatie

```
1 function q = doubleHorner(p,rho,mu)
2 %Dubbelrijige horner voert 2 deflatie stappen uit op p zodat p 2 graden
3 %lager wordtdoor de factor x^2 + rho*x +mu  weg te delen via
```

```
4  %de methode van Horner.
5  %
6  %    Signatuur: q = doubleHorner( rho, mu)
7  %
8  %    @param p
9  %        De veelterm die de deflatie ondergaat, voorgesteld door zijn
10 %        coefficienten als vector, met de hoogste graad term als eerste
11 %        element enzovoort. Dit is een 1 x n vector.
12 %
13 %    @param rho
14 %        de   c  o  ffici   van x in de term die zal worden weggedeelt.
15 %    @param mu
16 %        de   c  o  ffici   in de term die zal worden weggedeelt.
17 %
18 %    @result q
19 %        De veelterm die het resultaat is van de deflatie van p, voorgesteld
20 %        door zijn coefficienten als vector, met de hoogste graad term als
21 %        eerste element enzovoort. Dit is een 1 x n vector.
22 %
23 grootteP = size( p );
24 grootteP = grootteP(2);
25 % De eerste twee speciale gevallen doen
26
27      q(1) = p(1);
28      q(2) =  p(2) + q(1) * rho;
29
30      for graad = 3:(grootteP-2)
31
32        %Dubbelrijig horner
33
34          q(graad) = p(graad) + q(graad - 1) * rho - q(graad - 2) * mu;
35
36      end
37
38 end
```

De implementatie heeft invloeden met die van sectie 1.1 met 1 iteratie. Echter zal nu bij elke coëfficiënt direct het gevolg van het afzonderen van beide nulpunten in rekening worden gebracht. Daarom baseert deze methode zich ook op de 2 voorgaande coëfficiënten. wat de basisstap wat uitgebreider maakt. De methode berekent dus een veelterm van 2 graden lager dan p.

2.1.2 Fouten

De stabiliteit van deze methode lijd onder dezelfde problemen als 1.1:

- In lijn 28, als p(2) en q(1)*rho bijna even groot zijn en tegengesteld teken hebben.
- In lijn 34, als p(graad) en q(graad - 1)*rho of q(graad - 1)*rho en -q(graad-2)*mu bijna even groot zijn en tegengesteld teken hebben.

2.1.3 Vinden van de 2 nulpunten uit de 2de-graadsfunctie

Hier wordt opdracht 5 besproken.

Omdat Bairstow telkens een 2-de graadsveelterm afzondert, hebben we nog een methode nodig die efficiënt de wortels uit die veelterm haalt. Daarvoor gebruiken we de functie quadroots. Deze berekent de 1 wortel uit de veelterm en gaat dan iteratief tewerk om de 2e wortel te bepalen.

2.1.4 Implementatie

```
function ws = quadroots( p)
%quadroots Berekent de nulpunten van de 2de graads functie met de gekende
%formules en gebruikt deze als startwaarden voor newton rhapson om het
%nulpunt naar de gewenste tolerantie te brengen.
%
%   Signatuur: ws = quadroots( p, tol)
%
%   @param p
%       De 2de graads veelterm die geevalueerd zal worden, voorgesteld door
%       zijn coefficienten als vector, met de hoogste graad term als eerste
%       element enzovoort.
%
%
%   @return ws
%       De 2 nulpunten van de p.

% Indien a klein is, wordt de noemer en teller met 100 vermenigvuldigt voor
% een verbetering in de stabiliteit


if p(1) < 1

    % 128 is 2^7, die normaal goed is voor de stabiliteit, want dit is
    % de basis voor floats volgens de IEEE standaard, dus zorgt dit voor
    % zo min mogelijk nauwkeurigheidsverlies.

    stabiliteitsVerhoger = 128;
else
    stabiliteitsVerhoger = 1;
end

xEen = ( -p(2) + sqrt(p(2)^2 - 4*p(1)*p(3)) ) / ( 2 * p(1) *
    stabiliteitsVerhoger );

% Teller nu verhogen, niet ervoor want dan wordt teller groter voor er
% gedeeld wordt, wat een slechtere stabiliteit geeft.

xEen = xEen * stabiliteitsVerhoger;


% Volgens de stabiele methode berekenen als xEen niet te dicht van 0 ligt,
% anders de gewone formule toepassen.

if xEen > 0.0000000005

    % Opnieuw eerst de stabiliteitsVerhoger aanpassen

    if xEen < 1
        stabiliteitsVerhoger = 128;
    else
        stabiliteitsVerhoger = 1;
    end

    xTwee = p(3) / ( p(1) * xEen * stabiliteitsVerhoger);
```

```
54      xTwee = xTwee * stabiliteitsVerhoger;
55
56  else
57
58      % stabiliteitsVerhoger heeft hier al de juiste waarde
59
60      xTwee = ( -p(2) - sqrt(p(2)^2 - 4*p(1)*p(3)) ) / ( 2 * p(1) *
            stabiliteitsVerhoger );
61      xTwee = xTwee * stabiliteitsVerhoger;
62
63  end
64
65
66
67  ws = [ xEen xTwee ];
68
69  end
```

Een nuttige extra parameter en bijdrage aan de implementatie zou zijn dat men kan de stabiliteitsverhoger instellen naargelang de grootte van het probleem. Dit zou de robuustheid van het algoritme zeker ten goede komen.

2.2 Een nulpunt vinden met Bairstow

Hier wordt opdracht 6 besproken.

Indien een willekeurig veelterm gegeven wordt en 2 startwaardes, kan via de formules van Bairstow een nieuwe ρ en μ gevonden worden, die dichter bij het nulpunt ligt als er convergentie optreedt. Doordat we telkens 2 nulpunten afzonderen zal deze kwadratisch convergeren.

2.2.1 Implementatie

```
1  function ws = bairstow( p, start, tol)
2  %bairstow Geeft 2 nulpunten van de functie p aan de hand van de startwaarde
3  % met een gegeven tolerantie, gevonden met de methode van bairstow.
4  %
5  %   Signatuur: ws = bairstow( p, start, tol)
6  %
7  %   @param p
8  %       De veelterm die geevalueerd zal worden, voorgesteld door zijn
9  %       coefficienten als vector, met de hoogste graad term als eerste
10 %       element enzovoort. Dit is een 1 x n vector.
11 %
12 %   @param start
13 %       De 2 startwaarden nodig om Bairstow aan te vatten.
14 %       Dit is een 1 x 2 vector met [ a b ]
15 %
16 %   @param tol
17 %       De tolerantie van de fout, indien niet meegegeven 10^-6.
18 %
19 %   @return ws
20 %       De 2 nulpunten gevonden aan de hand van de starwaarde binnen de
21 %       tolerantie tol.
22 %
```

```
23
24 % Grootte van p bepalen.
25
26
27 grootteP = size( p );
28 grootteP = grootteP(2);
29
30
31 % Kijken of p minstens 3de graads is.
32
33 if grootteP < 3
34
35     disp("De meegegeven veelterm moet minstens van graad 2 zijn.")
36     ws = NaN;
37     return
38
39 end
40
41
42 %Check of tol gegeven is.
43
44 if nargin == 2
45     tol = 10^(-6);
46 end
47
48
49 % Werk vector initialiseren.
50
51
52 % Fout initialiseren.
53
54 fout = tol + 1;
55
56 iterator = 0;
57 mu = start(1)+ start(2);
58 rho = start(1)*start(2);
59 while fout > tol && iterator < 10000
60
61     werkVector(1) = p(1);
62     werkVector(2) =  p(2) + werkVector(1) * rho;
63
64     % De eerste keer dubbelrijig horner ( b's berekenen)
65
66     for graad = 3:grootteP
67
68       %Dubbelrijig horner
69
70         werkVector(graad) = p(graad) + werkVector(graad - 1) * rho +
              werkVector(graad - 2) * mu;
71
72     end
73
74     % Het 2 x 2 stelsel oplossen om delta p en delta u te vinden
75     dbNaardrho(1) = 0;
76     dbNaardrho(2) = werkVector(1);
```

```
77      dbNaardmu(1) = 0;
78      dbNaardmu(2) = 0;
79       for graad = 3:grootteP
80            dbNaardmu(graad) = dbNaardrho(graad-1);
81            dbNaardrho(graad) = werkVector(graad-1) + rho*dbNaardrho(graad-1)
                  + mu*dbNaardrho(graad-2);
82       end
83      D =dbNaardrho(grootteP-1)*dbNaardmu(grootteP) - dbNaardrho(grootteP)*
            dbNaardmu(grootteP-1);
84
85      deltaRho = -1/D *(werkVector(grootteP-1)*dbNaardmu(grootteP) -
            werkVector(grootteP)*dbNaardmu(grootteP-1));
86      deltaMu = -1/D * (werkVector(grootteP)*dbNaardrho(grootteP-1) -
            werkVector(grootteP-1)*dbNaardrho(grootteP));
87
88      fout =  max(abs(werkVector(grootteP)),abs(werkVector(grootteP-1)));
89
90      if fout > tol
91           rho = rho + deltaRho;
92           mu = mu + deltaMu;
93      end
94      iterator = iterator + 1;
95  end
96
97  ws = quadroots( [1,-rho,-mu] );
98
99  end
```

Het iteratief berekenen van de volgende ρ en μ, wordt op eenzelfde manier gestopt als 1.2.1.

2.2.2 Fouten

De grootste fout wordt verkregen door een kleine D in de berekening van $\delta\rho$ en $\delta\mu$, door de deling in lijn 85 en 86.

2.3 Alle nulpunten vinden

Hier wordt opdracht 7 besproken.

Met al deze werd zoals gevraagd een implementatie gemaakt die alle nulpunten berekent van een gegeven veelterm. Deze methode vraagt om een veelterm, 2 (mogelijks complexe) startwaarden en een tolerantie die 10^{-6} is indien niet meegegeven. Er worden 2 nulpunten bepaald met de methode van Bairstow met de gegeven startwaarde, waarna die nulpunten weggedeeld wordt met de methode van de dubbelrijige Horner. Er wordt met de nieuwe veelterm die door Horner werd teruggegeven verder gewerkt en er wordt opnieuw een nulpunt met Bairstow bepaald met als startwaarden de vorige gevonden nulpunten. Indien de graad van p nu kleiner zou zijn dan 2, dan kunnen we rechtstreeks het laatste nulpunt berekenen.
Dit wordt gedaan tot alle nulpunten van de veelterm gevonden worden.
Als Bairstow niet convergeert, en dus NaN wordt teruggegeven, wordt de huidige functie geplot en om nieuwe startpunten gevraagd aan de gebruiker.
Uiteindelijk wordt een vector met alle nulpunten van p teruggegeven.

2.3.1 Implementatie

```
1  function ws = bairstowdef( p, start, tol)
2  %bairstowdef Berekent alle nulpunten van p met de methode van Bairstow.
```

```
3  %
4  %    Signatuur: ws = bairstowdef( p, start, tol)
5  %
6  %    @param p
7  %        De veelterm die geevalueerd zal worden, voorgesteld door zijn
8  %        coefficienten als vector, met de hoogste graad term als eerste
9  %        element enzovoort. Dit is een 1 x n vector.
10 %
11 %    @param start
12 %        De 2 startwaarden nodig om Bairstow aan te vatten.
13 %        Dit is een 1 x 2 vector met [ a b ]
14 %
15 %    @param tol
16 %        De tolerantie van de fout, indien niet meegegeven 10^-6.
17 %
18 %    @return ws
19 %        De gevonden nulpunten.
20 %
21
22
23 % Bairstow tot graad kleiner is dan drie, dan quadroots of gewoon lineair
24 %ZORG DAT JE ALLES VIND, DOE DIT DOOR NIEUWE STARTWAARDEN TE DOEN INGEVEN
25 %INDIEN NIET ALLES WERD GEVONDEN, DOE DIT ZOALS AANGEGEVEN IN DE OPGAVE
26
27 %zeg dat ze desnoods een complexe startwaarde moeten geven,
28 %want enkel zo kun je complexe nulpunten vinden.
29
30
31 % p mag geen 0de graads zijn.
32
33 grootteP = size(p);
34 grootteP = grootteP(2);
35 if grootteP <= 3
36     disp("Functie moet minstens graad 3 zijn!");
37     ws = NaN;
38     return
39 end
40
41 % Check of tol gegeven is.
42
43 if nargin == 2
44     tol = 10^(-6);
45 end
46
47 % Initialiseer ws.
48
49 ws = zeros( 1, grootteP - 1);
50
51 % Maak de arrays aan om te plotten indien geen nulpunt gevonden word.
52 plotGrootte = 5000;
53 startPlot = -500;
54 eindPlot = 500;
55 X = zeros( plotGrootte, 1);
56 Y = zeros( plotGrootte, 1);
57 stapGrootte = (abs(startPlot) + abs(eindPlot)) / plotGrootte;
```

```
58
59 % Itereer aan de hand van het aantal gevonden nulpunten.
60
61 nbGevondenNulpunten = 0;
62
63 while nbGevondenNulpunten ~= grootteP - 1
64
65     if(grootteP - 1 - nbGevondenNulpunten == 1)
66         huidigNulpunt = -p(2)/p(1);
67     else
68         huidigNulpunt = bairstow( p, start, tol);
69         if (abs(imag(huidigNulpunt(2))) < tol)
70         huidigNulpunt(2) = real(huidigNulpunt(2));
71         end
72     end
73     if (abs(imag(huidigNulpunt(1))) < tol)
74         huidigNulpunt(1) = real(huidigNulpunt(1));
75     end
76
77     huidigeGrootteNulpunt = size(huidigNulpunt);
78     huidigeGrootteNulpunt = huidigeGrootteNulpunt(2);
79
80     if ~isnan(huidigNulpunt(1))
81         %Nulpunt gevonden binnen de tolerantie:
82         %Deel nulpunt weg en voeg nulpunt toe aan ws
83
84         nbGevondenNulpunten = nbGevondenNulpunten + huidigeGrootteNulpunt;
85
86         if(huidigeGrootteNulpunt > 1)
87             p = doubleHorner(p, huidigNulpunt(1)+ huidigNulpunt(2),
                    huidigNulpunt(1)* huidigNulpunt(2));
88             ws(nbGevondenNulpunten-1) = huidigNulpunt(2);
89         end
90         start = 0.9 * huidigNulpunt;
91         ws(nbGevondenNulpunten) = huidigNulpunt(1);
92
93     else
94         % Geen nulpunt gevonden, laad nieuwe startwaarde
95
96         % Maak arrays om te plotten.
97
98         xWaarde = startPlot;
99
100        huidigeGrootteP = size(p);
101        huidigeGrootteP = huidigeGrootteP(2);
102
103        % Bereken alle koppels om te plotten.
104        for i = 1:plotGrootte
105
106            yWaarde = 0;
107            exponent = 0;
108
109            % Bereken de y waarde.
110            % Begin achteraan de lijst, dus met de constante.
111            for j = huidigeGrootteP:-1:1
```

```
112
113                     yWaarde = yWaarde + p(j) * xWaarde^(exponent);
114
115                     exponent = exponent + 1;
116
117                 end
118
119                 X(i) = xWaarde;
120                 Y(i) = yWaarde;
121
122                 xWaarde = xWaarde + stapGrootte;
123             end
124
125             % Plot de functie.
126
127             plot(X,Y);
128
129             % Vraag om juiste input
130
131             input_hf = str2double(input('Geen nulpunt gevonden, geef 2 nieuwe (
                    misschien complexe) startwaarden: ','s'));
132             while isnan(input_hf(1)) || fix(input_hf(1)) ~= input_hf(1) ||
                    isnan(input_hf(2)) || fix(input_hf(2)) ~= input_hf(2)
133                 input_hf = str2double(input('Geef 2 getallen in: ','s'));
134             end
135
136             start = input_hf(1);
137         end
138
139
140 end
141
142 end
```

2.3.2 Fouten

Deze functie zelf bevat opnieuw niet veel pijnpunten.
Het plotten voldoet aan dezelfde voorwaarden als bij Newton-Raphson.

3 testen van robuustheid

Hier wordt opdracht 8 besproken.

Beide functies (newtonraphsondef en bairstowdef) werden uitgevoerd op de volledige testbank. Daaruit bleek dat Newton-Raphson minder snel gaat convergeren dan Bairstow. Maar beide implementaties leverden betrouwbare resultaten en kunnen toch als robuust beschouwd worden.

4 Convplot

Hier wordt opdracht 9 besproken.

Er zijn verschillende implementaties voor convplot, en deze worden apart besproken.

4.1 convplot voor Newton-Raphson

De convplot voor Newton-Raphson is als volgt geïmplementeerd:

```
function convplotnr(p, x0, a, b, tol)
    % function convplot(p, x0, a, b, tol)
    %   Deze functie maakt een plot die aangeeft welke startwaarden naar
    %   welk nulpunt convergeren.
    %
    % Inputs
    %   p    : Vector met de   c  o  fficien   van de veelterm
    %   x0   : Een startwaarde voor het vinden van alle nulpunten
    %   a    : Een vector met alle waardes voor het  r e le  deel van de
    %            startwaarde die gebruikt moeten worden
    %   b    : Een vector met alle waardes voor het imaginaire deel van de
    %            startwaarde die gebruikt moeten worden
    %   tol  : Een gewenste tolerantie

    %Check of tol gegeven is.

    if nargin == 4
        tol = 10^(-6);
    end

    % Zoek alle nulpunten van de veelterm en plot deze
    styles = {'b', 'k', 'c', 'm', 'g', 'y'};
    ws = newtonraphsondef(p, x0, tol);
    if size(ws, 2) < size(p, 2)-1
        error('not all zeros found')
    end
    figure
    for j = 1:size(ws, 2)
        plot(real(ws(j)), imag(ws(j)), [styles{j} '*'], 'MarkerSize', 20, '
            Linewidth', 5)
        prepleg{j} = num2str(ws(j));
        hold on
    end
    legend(prepleg{:}, 'Location', 'EastOutside')


    for k = a
        for l = b

            huidigNulpunt = newtonraphson( p, k + 1i*l);

            if isnan(huidigNulpunt)
                plot( k, l, ['r' '.'], 'MarkerSize', 20, 'Linewidth', 5)
            else

                kleur = 0;

                for j = 1:size(ws, 2)

                    % Kijken of het nulpunt hetzelfde is
                    verschil = abs( ws(j) - huidigNulpunt );

                    if verschil <= tol
```

```
53                          kleur = j;
54                      end
55
56                  end
57
58                  if kleur == 0
59                      error("Nulpunten komen niet overeen");
60                  end
61
62                  plot( k, l, [styles{kleur} '.'], 'MarkerSize', 20, '
                        Linewidth', 5)
63              end
64          end
65      end
66      axis([min(a), max(a), min(b), max(b)])
67      print('nrPlot','-depsc');
68  end
```

Wat we in feite doen is een lijst opstellen met voor elke nulwaarde van newtonraphsondef op onze gegeven starwaarde een kleur. En dan de gehele lijst met mogelijke startwaarden overlopen en dan kijken welk punt (binnen de tolerantie) gelijk is aan het verkregen nulpunt. Indien we een Nan terugkregen plotten we een rood punt.

4.2 convplot voor Bairstow

De convplot voor Bairstow is als volgt geïmplementeerd:

```
1  function convplotbr(p, x0, a, b, tol)
2      % function convplot(p, x0, a, b, tol)
3      %   Deze functie maakt een plot die aangeeft welke startwaarden naar
4      %   welk nulpunt convergeren.
5      %
6      % Inputs
7      %   p   : Vector met de   c  o  fficien   van de veelterm
8      %   x0  : Een startwaarde voor het vinden van alle nulpunten Dit is
9      %   een 1 x2 vector[ a b]
10     %   a   : Een vector met alle waardes voor het  r e le  deel van de
11     %           startwaarde die gebruikt moeten worden
12     %   b   : Een vector met alle waardes voor het imaginaire deel van de
13     %           startwaarde die gebruikt moeten worden
14     %   tol : Een gewenste tolerantie
15
16     %Check of tol gegeven is.
17
18     if nargin == 4
19         tol = 10^(-6);
20     end
21
22     % Zoek alle nulpunten van de veelterm en plot deze
23     styles = {'b', 'k', 'c', 'm', 'g', 'y'};
24     ws = bairstowdef(p, x0, tol);
25     if size(ws, 2) < size(p, 2)-1
26         error('not all zeros found')
27     end
28     figure
```

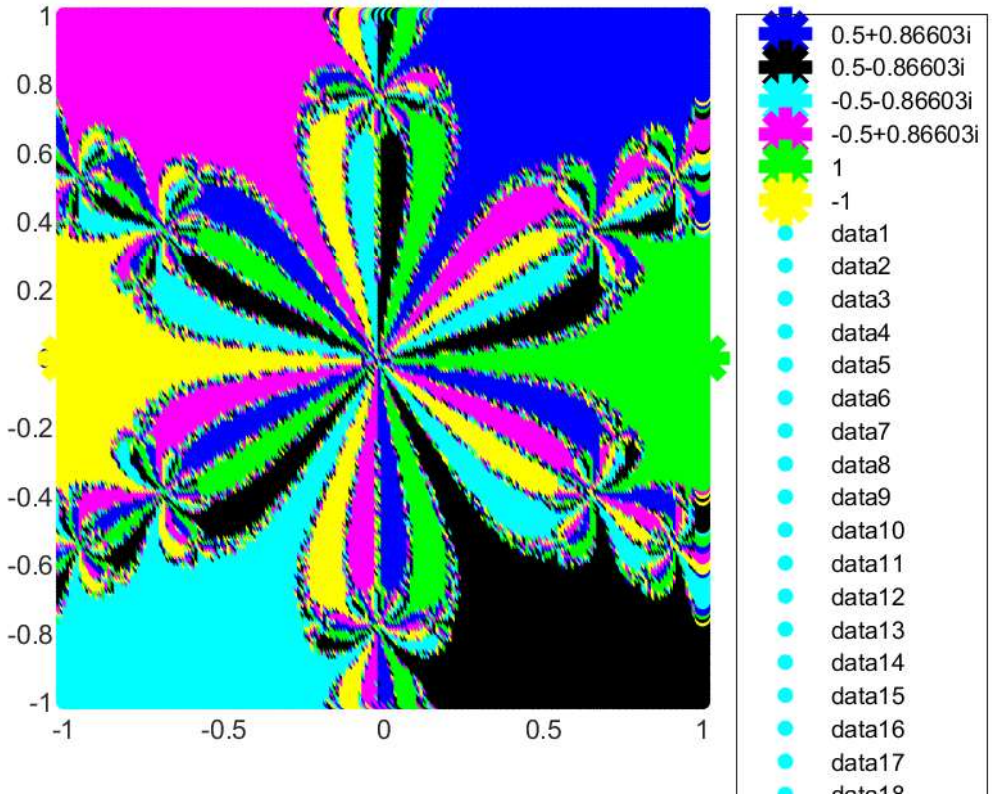
```
29      for j = 1:size(ws, 2)
30          plot(real(ws(j)), imag(ws(j)), [styles{j} '*'], 'MarkerSize', 20, '
                Linewidth', 5)
31          prepleg{j} = num2str(ws(j));
32          hold on
33      end
34      legend(prepleg{:}, 'Location', 'EastOutside')
35
36
37      for k = a
38          for l = b
39
40              huidigNulpunt = bairstow( p, [k + 1i*l,k - 1i*l]);
41
42              if isnan(huidigNulpunt(1))
43                  plot( k, l, ['r' '.'], 'MarkerSize', 20, 'Linewidth', 5)
44              else
45
46                  kleur1 = 0;
47
48                  for j = 1:size(ws, 2)
49
50                      % Kijken of het nulpunt hetzelfde is
51                      verschil1 = abs(ws(j) - huidigNulpunt(1));
52
53                      if verschil1 <= tol
54                          kleur1 = j;
55                      end
56
57                  end
58
59                  if kleur1 == 0
60                      error("Nulpunten komen niet overeen" + huidigNulpunt(1)
                            );
61                  end
62
63                  plot( k, l, [styles{kleur1} '.'], 'MarkerSize', 20, '
                        Linewidth', 5)
64                  if(size(huidigNulpunt,2) > 1)
65                      kleur2 = 0;
66
67                      for j = 1:size(ws, 2)
68
69                          % Kijken of het nulpunt hetzelfde is
70                          verschil2 =  abs(ws(j) - huidigNulpunt(2));
71
72                          if verschil2 <= tol
73                              kleur2 = j;
74                          end
75
76                      end
77
78                      if kleur2 == 0
79                          error("Nulpunten komen niet overeen" +
                                huidigNulpunt(2));
```

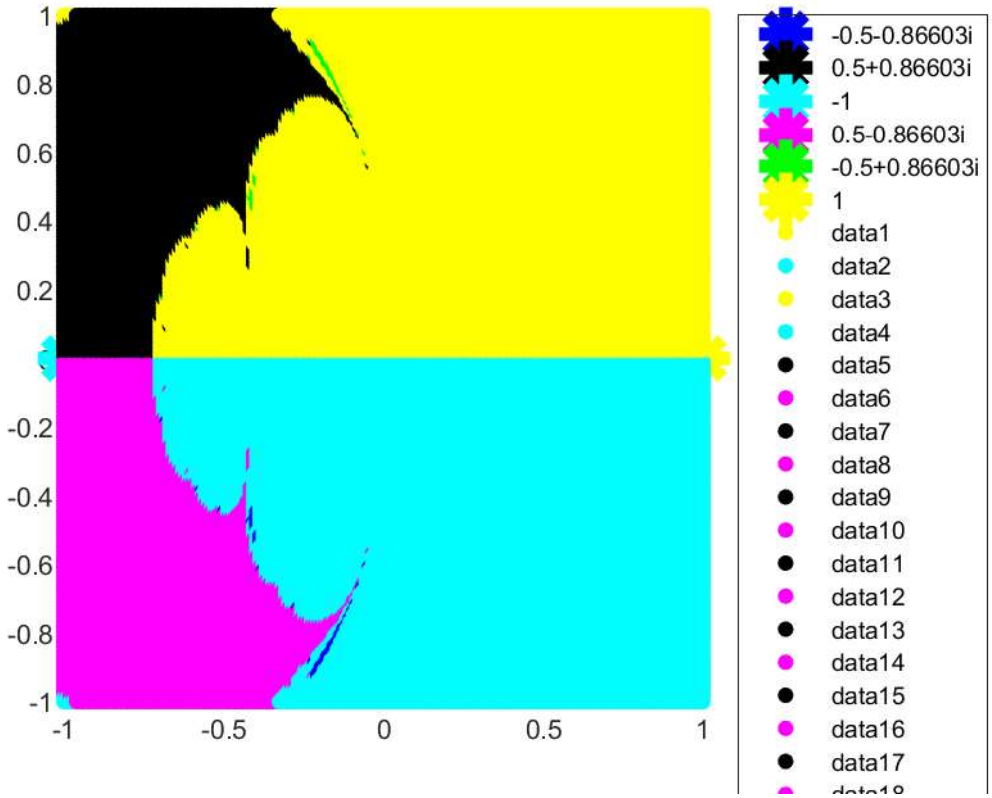
```
80                          end
81
82                          plot( k, -l, [styles{kleur2} '.'], 'MarkerSize', 20, '
                                Linewidth', 5)
83                      end
84                  end
85              end
86          end
87          axis([min(a), max(a), min(b), max(b)])
88          print('brPlot','-depsc');
89  end
```

Deze implementatie is in feite bijna identiek aan die van Newton-Raphson. Behalve dat we moeten rekening houden met het feit dat we 2 nulpunten kunnen meekrijgen.

4.3 convplots van Bairstow en Newton-Raphson

Hier wordt opdracht 10 besproken.


1
0.8
0.6
0.4
0.2
-0.2
-0.4
-0.6
-0.8
-1
-1
-0.5
0
0.5
1
0.5+0.86603i
0.5-0.86603i
-0.5-0.86603i
-0.5+0.86603i
1
-1
data1
data2
data3
data4
data5
data6
data7
data8
data9
data10
data11
data12
data13
data14
data15
data16
data17


1
0.8
0.6
0.4
0.2
-0.2
-0.4
-0.6
-0.8
-1
-1
-0.5
0
0.5
1
-0.5-0.86603i
0.5+0.86603i
-1
0.5-0.86603i
-0.5+0.86603i
1
data1
data2
data3
data4
data5
data6
data7
data8
data9
data10
data11
data12
data13
data14
data15
data16
data17

We kunnen enkel dingen toch duidelijk aanhalen uit de 2 plots:

- Bairstow heeft veel minder logische en gelijk verdeelde vlakken. Dit komt waarschijnlijk door de grote sprongen die het kan maken. Dit omdat het met 2 variabelen werkt.
- Newton-Raphson heeft een zeer afgebakende structuur en alles convergeert naar iets in de dichte omgeving. Dit waarschijnlijk omdat de afgeleiden ervoor zorgen dat we aan een soort van hillclimbing gaan doen.
- Aan de randen van Bairstow kunnen we waar nemen dat we convergeren naar totaal andere punten. Deze zone is zeer klein in vergelijking met de geweven structuur van Newton-Raphson. We kunnen dus veel beter voorspellen waarnaar een punt zal convergeren bij Bairstow als we de algemene plot weten. Terwijl Newton-Raphson duidelijker rond het punt convergeert maar minder voorspelbaar is op de randen.

5 Newton-Raphson VS Bairstow

Hier wordt opdracht 11 besproken.

Men kan zien dat Bairstow heel wat complexer is dan Newton-Raphson en ook altijd convergeert. Het convergeert ook in veel groter zones dan Newton-Raphson. Bairstow laat wel veel meer ruimte voor het opblazen van fouten en is een stuk geheugenintensiever dan newton-Raphson. Men kan zelf op beperkte instanties op beperkte machines een verschil in performatie voelen. Maar het geeft wel over het algemeen de beste resultaten en convergeert dus ook veel sneller. Het zal dus een betere methode zijn, maar om op krachtigere machines te draaien.

Besluit

We zien dat Newton-Raphson en bairstow vrij eenvoudige theoretische modellen zijn, maar dat deze we vrij snel door kleine verschillen zich van elkaar onderscheiden. De implementatie is ook een stuk verschillend. Toch komen de pijnpunten in vorm grotendeels overeen.

Referenties

[1] Adhemar Bultheel. *Inleiding tot de numerieke wiskunde.* Acco, 2006.