Umeå Universitet
Department of Computing Science
09/12/2018

Project 1 - Robust and accurate root finding for real polynomials

Aladdin Persson (alhi0008@student.umu.se)

Version 2, Username: tfy17api
Teknisk-vetenskapliga beräkningar
Supervisors: Carl Christian Kjelgaard Mikkelsen,
Niclas Börlin

Contents

1 Introduction

With computers inability to calculate in exact arithmetic, problems oftentimes arise when blindly relying on an output value to be correct. In this project we wish to create a root finder for polynomials using the bisection method and that also computes roots reliably. This means that it interrupts the calculations and informs the user when they cannot be trusted. We also wish to compute, along with the approximated values of the roots, bounds for the unavoidable errors that follow when calculating in floating point arithmetic.

The Chebyshev polynomials of the first kind have many desirable testing properties to see if the code is working as intended. Among other attributes, one feature which is of particular convenience is that they have all their roots in $x \in [-1, 1]$ which makes initializing a bracket with the bisection method straightforward. A theory part for these polynomials is found in Section 2. When creating the software for finding the roots of a polynomial it is important to know when the calculations cannot be trusted and therefore when the calculations should be interrupted. This is explained in further depth in Section 3. Lastly relative errors for the roots and a concrete example is examined and why the user can trust the calculations are summarized in Section 4.

All MATLAB files necessary to run the scripts for this project are included in a separate zip file named P1Code.

2 Chebyshev Polynomials

The Chebyshev polynomials of the first kind are generated by the following linear reccurence relation

$$T_0 = 1 \tag{1}$$
$$T_1 = x \tag{2}$$
$$T_{j+1} = 2xT_j - T_{j-1},\ j \in \{1, 2, 3, ...\} \tag{3}$$

where the first T_n for $n = 2, 3, 4, 5, 6$ are given by

$$T_2 = 2x^2 - 1 \tag{4}$$
$$T_3 = 4x^3 - 3x \tag{5}$$
$$T_4 = 8x^4 - 8x^2 + 1 \tag{6}$$
$$T_5 = 16x^5 - 20x^3 + 5x \tag{7}$$
$$T_6 = 32x^6 - 48x^4 + 18x^2 - 1 \tag{8}$$

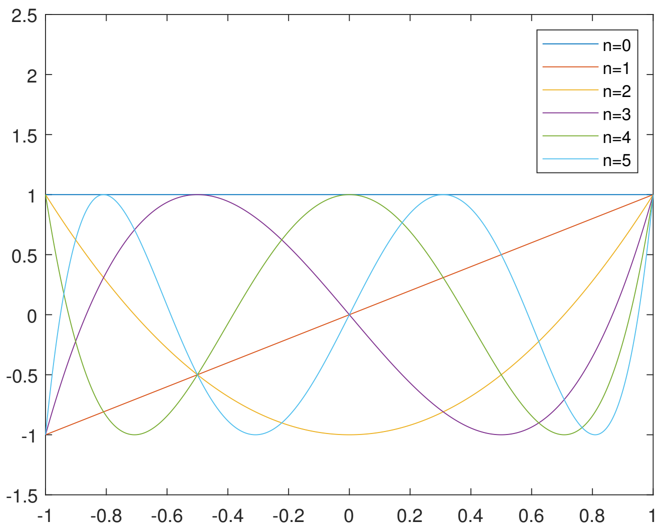

Figure 1 – Plot of the first $n = 6$ Chebyshev polynomials

2.1 Degree of a Chebyshev polynomial of the first kind

One can observe that for each n the corresponding Chebyshev polynomial has exactly degree n for the ones written above. We wish to show that this is in fact the case for all $n \in \mathbb{N}$.

Proof. Let $V \subseteq \mathbb{N}$ be given by

$$V = \left\{ n \in \mathbb{N} : T_n \text{ has degree } n \right\}$$

Our goal is to show that $V = \mathbb{N}$. It is clear that $n = 0, 1 \in V$.

Now assume that $n \in V$ such that

$$T_n = 2xT_{n-1} - T_{n-2} \tag{9}$$

has order n. We assume that T_{n-1} has degree $n-1$ and that T_{n-2} has degree $n-2$.

We wish to show that $n+1 \in V$ such that

$$T_{n+1} = 2xT_n - T_{n-1} \tag{10}$$

We have assumed that T_n has degree n where T_{n-1} therefore must have degree $n-1$. Since T_n of order n is multiplied by $2x$ this means it must have degree $n+1$. Therefore we have proved that $n+1 \in V$. By the principle of mathematical induction we conclude $V = \mathbb{N}$. This completes the proof.

□

2.2 Roots of a Chebyshev polynomial of the first kind

The roots for small n, such that of (4) or perhaps (5) are easily calculated. However for larger n the roots are not as easily calculated. For this the following theorem is used without proof

Theorem. *The Chebyshev polynomials of the first kind satisfy the equation*

$$T_n(cos(\theta)) = cos(n\theta),\ \theta \in \mathbb{R} \tag{11}$$

We wish to show that the n roots of T_n are given by

$$x_k = cos\left(\frac{(2k-1)\pi}{2n}\right),\ k = 1, 2, ..., n \tag{12}$$

Proof. According to theorem stated, $T_n(cos(\theta))$ can be written as $cos(n\theta)$. From previous proof we know that T_n has a polynomial of degree n. To find the roots we are therefore looking for

$$cos(n\theta) = 0 \tag{13}$$

We know this is equal to 0 if and only if

$$\theta_k = \frac{(2k-1)\pi}{2n}, k \in \mathbb{Z} \tag{14}$$

Referencing to the Fundamental Theorem of Algebra we know that a polynomial of degree n has n roots. We also know that

$$cos(x) = cos(-x) \tag{15}$$

Hence, if we are looking for unique roots and therefore unique solutions in (14) not only should we restrict to the interval $[-\pi, \pi]$, we should because of (15) also only look at $[0, \pi]$. Remembering that we need to look for n unique solutions we look at

$$k = 0 \implies \theta_0 = \frac{-\pi}{2n} \notin [0, \pi] \tag{16}$$
$$k = 1 \implies \theta_1 = \frac{\pi}{2n} \in [0, \pi] \tag{17}$$
$$k = n \implies \theta_n = \pi - \frac{\pi}{2n} \in [0, \pi]. \tag{18}$$
$$\tag{19}$$

Because of the relationship of

$$\theta_0 < \theta_i < \theta_n, i \in [1, n-1] \tag{20}$$

We can conlude that $k = 1, 2, 3, ..., n$ makes up a total of n unique solutions, where all $\theta_k \in [0, \pi]$.

To calculate the roots x_k we recognize that $x_k = cos(\theta_k)$

$$x_k = cos\left(\frac{(2k-1)\pi}{2n}\right), \ k = 1, 2, ..., n \tag{21}$$

This concludes the proof.

$\square$

3 Root finding software

A common method for finding a very close approximation to the roots of a polynomial is utilizing the bisection method. In theory this method is very robust but when applying it to computers with finite precision, it can result in code which is not reliable.

Specifically, let $y = f(x)$ where $f(x)$ is a polynomial. Whenever calculating $f(x)$ with a computer, we are in fact computing an approximation which we will denote $\hat{y}$. It is in many cases of interest to bound the error between the true value y and the approximated y which we can write as

$$|y - \hat{y}| \leq \mu \cdot u \tag{22}$$

where we let u denote the unit roundoff error

$$u = \begin{cases} 2^{-23} & \text{in single precision} \\ 2^{-52} & \text{in double precision} \end{cases} \tag{23}$$

and the method to calculate μ for a polynomial are found in [1] page 43, algorithm 9. The calculations compute the running error bound specifically using Horner's method which gives an upward estimate of the difference between the true value y and the approximated $\hat{y}$. When using the bisection method, which utilizes the intermediate value theorem we need to be certain that the computed sign is in fact correct. If it is the case that we cannot be certain that the computed sign is correct, the bisection algorithm will not work as intended. From (22) we know that

$$\hat{y} - \mu u \leq y \leq \hat{y} + \mu u \tag{24}$$

If it is the case which can be easily checked that

$$|\hat{y}| \leq \mu u \tag{25}$$

then the sign of the approximated $\hat{y}$ cannot be trusted. In terms of the bisection method then this means that it cannot be used with confidence in the result. The function (MyRoot.m) uses this fact and leaves a flag which tells the user that the results are unreliable. A minimal working example of this is found in the file (MyRootMWE1.m) which finds the roots for the Chebyshev polynomial T_{10} as further dicussed in Section 4.

4 Calculations

The bisection method that uses the additional functionality to check for sign certainty of approximated $\hat{y}$, outputs a bracket $[a, b]$ where we are sure that the true root r is between. It also outputs a value $\hat{r}$ which is the best approximation and that is the midpoint of $[a, b]$ such that

$$\hat{r} = \frac{a+b}{2}. \tag{26}$$

It is oftentimes of interest to know the relative error, and specifically an upward estimate of the relative error of our calculations which can be found by

$$y = \left|\frac{r-\hat{r}}{r}\right| \leq \frac{1}{2}\left|\frac{|b-a|}{r}\right| \leq \frac{1}{2}\left|\frac{|b-a|}{min(|a|,|b|)}\right|. \tag{27}$$

We recognize that in order to make an upward estimate in the last inequality it is necessary to choose the minimum of $(|b|, |a|)$ in the denominator as one can imagine that $\{r \in \mathbb{R} \ : r < 0\}$ as well as $\{r \in \mathbb{R} \ : r \geq 0\}$ where to make an upward estimate we always need to divide by the smallest of a, b which will be different depending on the sign of the root. A special case that needs to be taken into consideration is that a requirement to make a relative error is that $r \neq 0$ as then (27) is not defined. A practical solution to this when programming is to simply check if we are sure that the root cannot be at 0, i.e that the signs of b, a are the same.

Now let us consider the example of finding roots for

$$T_{10} = 512x^{10} - 1280x^8 + 1120x^6 - 400x^4 + 50x^2 - 1. \tag{28}$$

We will use the bisection method but with added functionality that uses the conclusion of Section 3 to check for situations where the evaluated sign of a function evaluation cannot be trusted. Additionally we also wish it to calculate the relative error according to (27) unless it is the special case mentioned previously where it cannot do so. Running the code (MyRootMWE1.m) that uses a more robust bisection method documented in (MyRoot.m), we receive a table of the calculated roots, running error bound, relative error bounds, etc. For this example this displays as the following

Table 1 – Resulting table of applying MyRoot to the polynomial T_{10} with $\delta = 10^{-15}$, $\epsilon = 10^{-15}$. Table displays the relevant resulting brackets [a,b], residuals, running error bound (REB), relative error (R.E)

flag	iter	a	b	$\hat{r}$	residual	REB	R.E	trust
3	38	-9.876883e-01	-9.876883e-01	-9.876883e-01	-3.967937e-13	6.340730e-13	7.295163e-14	0
3	39	-8.910065e-01	-8.910065e-01	-8.910065e-01	5.462297e-14	3.202772e-13	4.043375e-14	0
3	41	-7.071068e-01	-7.071068e-01	-7.071068e-01	2.353673e-14	7.860379e-14	1.271775e-14	0
3	44	-4.539905e-01	-4.539905e-01	-4.539905e-01	-1.665335e-15	8.994736e-15	2.506613e-15	0
3	46	-1.564345e-01	-1.564345e-01	-1.564345e-01	-2.220446e-16	5.506726e-16	1.774262e-15	0
1	46	1.564345e-01	1.564345e-01	1.564345e-01	-1.443290e-15	5.506726e-16	1.774262e-15	1
3	44	4.539905e-01	4.539905e-01	4.539905e-01	-1.665335e-15	8.994736e-15	2.506613e-15	0
3	41	7.071068e-01	7.071068e-01	7.071068e-01	2.353673e-14	7.860379e-14	1.271775e-14	0
3	39	8.910065e-01	8.910065e-01	8.910065e-01	5.462297e-14	3.202772e-13	4.043375e-14	0
3	38	9.876883e-01	9.876883e-01	9.876883e-01	-3.967937e-13	6.340730e-13	7.295163e-14	0

Note here that the last column named *trust* corresponds to a binary $0, 1$ which can be misleading but is of importance to make the user question the results. The relevant information is given by $flag$ and perhaps *trust* should not be emphasized too much but gives the user the information that if the code would have not been stopped then further calculations would not be reliable. As one can see the $flag$ is set to 3 for the entire list of roots and as detailed in documentation of (MyRoot.m) implies that it was stopped because of sign uncertainty of the last function evaluation. The code utilizes conclusions of Section 3 and summarized by (25) and this fact that the running error bound was greater or equal to our approximation $\hat{y}$ is what made the code not able to reliably continue. If we wanted to make the code run with better precision than what was given from (Table 1), it would be a necessity to make the running error bound smaller. The residual calculated here is the function evaluated at the point $\hat{r}$ which if less than ϵ will break, and it is important to distinguish this from the root error $r - \hat{r}$ which if less than delta will break. Both of ϵ and δ are constants specified by the user togethor with the number of maximum amount of iterations the program should be run which in turn impacts the resulting relative error.

To conclude (Table 1) shows the brackets $[a, b]$ to the point where we are certain that the computation has been correct and where $\hat{r}$ simply follows from (26).

5 Conclusions

From this project we have seen that even when applying a simple algorithm such as the bisection method whenever we implement this in practice with floating point arithmethic we need to be certain that the calculations are robust. In this case it means including error bounds that are inevitable when using inexact arithmetic and making critical interruptions when the root calculations can no longer be trusted. An interesting future improvement for this project would be to see how low the relative error can become by finding more effective ways of evaluating polynomials and also estimating the running error bound.

6 References

References

[1] Carl Christian Kjelgaard Mikkelsen: *An Introduction to Scientific Computing*, Department of Computing Science, Umeå University (2018)

7 Appendix

7.1 Appendix 1 - MyRoot.m

Source code for `MyRoot.m`.

```
function [x, flag, it, a, b, his, y, reb, res]=MyRoot(p,a0,
    b0,delta,eps,maxit)
% MyRoot  Finds roots of polynomials using the bisection
    method
%
% INPUT:
%   p         array of coefficients used by my_horner
%   a0, b0    the initial bracket
%   delta     return if current bracket is less than delta
%   eps       return if current residual is less than
    epsilon
%   maxit     return after maxit iterations
%
% OUTPUT:
%   x       final approximation of the root
%   flag    a flag signaling succes or failure,
%               flag  = -2 the initial bracket is bad
%               flag  = -1 the sign of f(a0) or f(b0) cannot
     be trusted
%               flag  =  0 maxit iterations completed
    without convergence
%               flag  >  0 then convergence has been
    achieved and if:
%                   flag = 1  then the last bracket is
    shorter than delta
%                   flag = 2  then the last function value is
     bounded by eps
```

```
%                   flag = 3  then the sign of the last function value cannot
%                                 be trusted
%   it      the number of iterations completed
%   a, b    a(j) and b(j) form the jth bracket around his(j)
%   his     a vector containing all computed approximations of the root
%   y       the computed values of y=p(his)
%   reb     the running error bounds for y
%   res     the residuals (res = residual)
%
% MIMIMAL WORKING EXAMPLE: MyRootMWE1.m

% PROGRAMMING by Carl Christian Kjelgaard Mikkelsen (spock@cs.umu.se)
%   2018-11-14 Skeleton extracted from working code MyRoot

% PROGRAMMING by Aladdin Persson (alhi0008@ad.umu.se)
%   2018-11-17 Initial programming
%   2018-12-08 Minor improvements to code

% Initialize the flag.
flag=0;

% Dummy initialization of *all* output arguments
x=NaN; it=0; his=NaN;
reb=zeros(maxit,1);
a=zeros(maxit,1);
y=zeros(maxit,1);
b=zeros(maxit,1);
res=zeros(1,maxit);

% Initialize search bracket (alpha,beta) such that alpha <= beta
alpha=min(a0,b0);
beta=max(a0,b0);

% Compute fa=p(alpha) and fb=p(beta) and associated error bounds
[fa, ~, reb_alpha]= my_horner(p, alpha);
```

```
[fb, ~, reb_beta]= my_horner(p, beta);

% Investigate if the flag should be -2 or -1
if sign(fa)*sign(fb) > 0
    flag=-2;
elseif (abs(fa) <= reb_alpha) || (abs(fb) <= reb_beta)
    flag=-1;
end

% Stop if difference less than delta
if abs(beta-alpha) <= delta
    flag=-2;
end

% If the sign is the same, then we cannot run bisection
if sign(fa)*sign(fb) > 0
    flag=-2;
end

% In case flag is initially less than zero, there was
    something wrong
if (flag<0)
    % The initial bracket is either bad or cannot be judged
    return
end

% Main loop
for j=1:maxit
    % Record the current search bracket
    a(j)=alpha; b(j)=beta;

    % Carefully compute the midpoint c of the current
        search bracket
    c=alpha+(beta-alpha)/2;

    % Evaluate fc = p(c) and the running error bound for fc
    % priori error from my_horner is not necessary for
        bisection.
    [fc,~,rebfc]=my_horner(p,c);

```

```
    % Save the current values
    x=c; his(j)=c; y(j)=fc; reb(j)=rebfc; res(j)=fc;

    % Check for small bracket
    if abs(alpha-beta)<=delta
        flag=1;
    end

    % Check for small residual
    if abs(fc)<=eps
        flag=2;
    end

    % Check if the computed sign of the p(c) cannot be trusted
    if abs(fc) <= rebfc
        flag=3;
    end

    % Check if we can break out of the loop
    if flag>0
        % Yes, there is no reason to continue
        break
    end

    % ---------------------------------------------------------------------
    % At this point we know that we need more iterations.
    % ---------------------------------------------------------------------

    % Rebracket the root and recycle the old function values
    if sign(fa)*sign(fc)==-1
        beta=c; fb=fc;
    else
        alpha=c; fa=fc;
    end
```

```
end

% Shrink the output to avoid tails of unnecessary zeros
a=a(1:j); b=b(1:j); his=his(1:j); res=res(1:j); reb=reb(1:j
    ); y=y(1:j);

% Return the number of iterations
it=j;
```

7.2 Appendix 2 - MyRootMWE1.m

Source code `MyRootMWE1.m`.

```
% Minimal working example for finding roots to polynmials
    using
% MyRoot.m which uses the bisection method.

% PROGRAMMING by Aladdin Persson (alhi0008@student.umu.se)
% 2018-11-21 Initial code
% 2018-11-24 Included relative error check

% clear variables for the sake of uneccessary bugs
clear all; close all; clc

% declare delta, eps which are used for bisection algorithm
delta=1e-13;
eps=1e-13;

% if not stopped by delta or eps, stop by maxiteration so
    that it does
% not run forever.
maxit=100;

% Initialize vector corresponding to polynomial
    coefficients in ascending
% order of exponent of x. Chebyshev polynomial for n = 10 (
    T_10)
p = [-1, 0, 50, 0, -400, 0, 1120, 0, -1280, 0, 512];

% declare potential points a, b for bisection
m = linspace(-1,1,102);
```

```

% Following initiaizations for using function displaydata
colheadings = {'flag','iter','a','b','root', 'residual', 'REB', ...
                    'R.E', 'trust'};
rowheadings={};
wid = [6 8,16,16,16,16,16,16, 5];
fms = {'d'};
colsep = ' | ';
rowending = ' ';
fileID=1;
data = [];

% Loop through all points declared by m and check if bisection algorithms
% find any roots between these two points.
relerror = [];

for j = 1:length(m)-1
    a0 = m(j); b0 = m(j+1);
    [x, flag, it, a, b, his, y, reb, res]=MyRoot(p,a0,b0,delta,eps,maxit);

    % if sign of computed cannot be trusted then trust = 0, else trust=1
    if flag == 3 || flag == -1
        trust=0;
    else
        trust=1;
    end

    % Only if there was something found from MyRoot do we which to store
    % the results
    if ~isnan(x) && it ~= 0

        % If the resulting bracket [a,b] are of different signs
        % we cannot trust the relative error as root=0 is a possibility.
```

```
            if sign(b(it))*sign(a(it))==-1
                trust=0;
                reler=NaN;
            else
                reler=1/2*(abs(b(it)-a(it)))/min(abs(a(it)),abs
                   (b(it)));
            end

            % Concatenating the new information to data matrix
            data = [data; flag, it, a(it), b(it), x, res(it),
               reb(it), ...
                     reler, trust];
        end
end

% Only if we found roots can we display, hence first check
   so we have data
if size(data)>0
    displaytable(data,colheadings,wid,fms,rowheadings,
       fileID,...
                     colsep,rowending);
end
```

7.3 Appendix 3 - MyChebyshev.m

Source code for `MyChebyshev.m`.

```
function y=MyChebyshev(n,x)

% MyChebyshev Evaluates the first n Chebyshev polynomials
%
% CALL SEQUENCE: y=MyChebyshev(n,x)
%
% INPUT:
%    n      the number of polynomials
%    x      a vector of length m containing the sample points
%
% OUTPUT:
%    y       a matrix of dimension m by n such that y(i,j) = T
%    (j,x(i))
%
```

```
% MINIMAL WORKING EXAMPLE: MyChebyshevMWE1.m

% PROGRAMMING by Carl Christian Kjelgaard Mikkelsen (
    spock@cs.ume.se)
%  2018-11-14 Skeleton extracted from working function

% PROGRAMMING by Aladdin Persson (alhi0008@student.umu.se)
%  2018-11-17 Initial programming
%  2018-11-27 Minor improvements and added comments

% Determine number of element in x
m = length(x);

% Reshape x as a column vector
x = reshape(x,[1,m]);

% Allocate space for output y
y = zeros(n, m);

% Initialize the first two columns of y
y(1,:)=1; y(2,:)=x(1,:);

% Calculate all remaining columns of y
for j = 3:n
    y(j, :) = 2.*x.*(y(j-1,:))-y(j-2,:);
end
```

7.4 Appendix 4 - MyChebyshevMWE1.m

Source code for `MyChebyshevMWE1.m`.

```
% MyChebyshevMWE1 Minimal working example for MyChebyshev
    function

% PROGRAMMING by Carl Christian Kjelgaard Mikkelsen
%   2018-11-14 Skeleton extracted from working code

% PROGRAMMING by Aladdin Persson
%   2018-11-17 Initial programming

% clear variables for the sake of uneccessary bugs
```

```
clear all; close all; clc

% Set number of polynomials
n = 6;

% Set number of sample points
m = 1000;

% Define sample points
x = linspace(-1,1,m);

% Generate function values
y = MyChebyshev(n, x);

% Plot all graphs with one command
plot(x,y)

% Adjust axis to make room for legend
ylim([-1.5,2.5])

% Construct and display legend
str=[];
for i=0:n-1
    str=[str strcat("n=",string(i))];
end

% Add legend
legend(str);
```

7.5 Appendix 5 - my_horner.m

Source code for `my_horner.m`.

```
function [y,aeb,reb,gamma]=my_horner(a,x)
% MY_HORNER   An implementation of Horner's method
%
% CALL SEQUENCE:
%
% [y]=my_horner(a,x)
%
% INPUT:
```

```
%   a       array of cofficients determining p
%   x       array of arguments to pass to p
%
% OUTPUT:
%   y       the computed value of the polynomial
%
% MINIMAL RUNNING EXAMPLE: my_horner

% Isolate the number of coefficients
m=numel(a);

% Isolate the degree of the polynomial
n=m-1;

% Both a and x must be in double precision or MATLAB works
    in single
if (strcmp(class(a),'double') && strcmp(class(x),'double'))
    % Set u to double precision unit roundoff
    u=2^-53;
else
    % Set u to single precision unit round off
    u=2^-24;
end

% Reshape the coefficient array as a row vector
aux=reshape(a,1,m);

% Determine the size of the input array x
sx=size(x);

% Initialize the output arrays
y=ones(sx)*aux(m); pt=ones(sx)*abs(aux(m));

% Initialize running error bound
mu=0;

% Main loop.
for j=1:n
    % Compute intermediate value
    z=y.*x;
```

```
    % Update polynomial p
    y=z+aux(m-j);
    % Update running error bound
    mu=mu.*abs(x)+abs(z)+abs(y);
    % Update polynomial pt
    pt=pt.*abs(x)+abs(aux(m-j));
end

% Compute the relvant gamma factor
gamma=(2*n*u)/(1-2*n*u);

% Compute the apriori error bound
aeb=pt.*gamma;

% Compute the running error bound
reb=mu.*u;
```