Table of Contents

```
%{
Paul Yuska
EP501 HW2
Nonlinear Equations and Root Finding

This is the top-level script. Run this.
%}

clc
clear
close all

fprintf('EP501 - HW2 - Paul Yuska\n\n')
```

```
EP501 - HW2 - Paul Yuska
```

1. Finding roots of functions lacking a closed form

Write a code block that uses nwtnapprox.m to find the first, and then the first six roots of the order-zero Bessel function

```
% Generate order-zero Bessel function

% domain; have to manually specify to match number of desired roots
 (for
% now)
dom = 0:0.1:20;
bess_res = besselj(0,dom);
% figure
% plot(dom,bess_res)
% grid on
% grid minor
% From plot, first root somewhere between x = 2 and x = 3

numroots = 6;
bess_root_guess = zeros(1,numroots); %
k = 1; % bess_root_indices index
for i = 1:length(dom)-1
    if bess_res(i)*bess_res(i+1) < 0
```

```
            % flipped sign, found a root
            bess_root_guess(k) = i;
            k = k+1;
        else
            % same sign, did not find a root, keep looking
            continue
        end % if - bess_res
    end % for - i

    % Find first root
    eps = 0.01; % small deviation from x to approximate derivative at a
     point
    maxIter = 100; % maximum no. of iterations
    toler = 1e-8;

    bess_roots = zeros(numroots,3);
    fprintf('====== PART 1
     ==================================================')
    fprintf('\nFinding zeros of the 0th order Bessel function of the first
     type:\n\n')

    ordinal = {'First','Second','Third','Fourth','Fifth','Sixth'};
    for j = 1:numroots
        initX = dom(bess_root_guess(j));
        [root,it,succ] = nwtnapprox(@(z)
     besselj(0,z),eps,initX,maxIter,toler);
        bess_roots(j,1) = root;
        bess_roots(j,2) = it;
        bess_roots(j,3) = succ;
        % Print output
        if succ
            fprintf('%s root for J_0(x) at x = %.4f (%d iterations)\n',...
                ordinal{j},bess_roots(j,1),bess_roots(j,2))
        else
            disp('Failed to converge')
        end
    end

    fprintf('\nEvaluating the built-in Bessel function ''besselj'' at the
     roots\n')
    fprintf('found by the code (''bess_roots'') as ''>
     besselj(0,bess_roots)''\n')
    fprintf('gives zeros within the specified tolerance:\n\n')
    disp(besselj(0,bess_roots(:,1)))

    %{
    can't find a print book that tabulates zeros of 0th-order Bessel
     functions
    of the first kind. (checked about 10 PDE/ODE books in the library)

    however, evaluating the MATLAB built-in bessel function of the first
     type
    and 0th order for the roots found in problem 1 as so:
```

```
besselj(0,bess_roots)

results in an array of zeros within the specified tolerance, as
 expected.
%}

fprintf('====== PART 2
 ==================================================')
fprintf('\n\n')
```

```
====== PART 1 ==================================================
Finding zeros of the 0th order Bessel function of the first type:

First root for J_0(x) at x = 2.4048 (2 iterations)
Second root for J_0(x) at x = 5.5201 (2 iterations)
Third root for J_0(x) at x = 8.6537 (3 iterations)
Fourth root for J_0(x) at x = 11.7915 (3 iterations)
Fifth root for J_0(x) at x = 14.9309 (2 iterations)
Sixth root for J_0(x) at x = 18.0711 (2 iterations)

Evaluating the built-in Bessel function 'besselj' at the roots
found by the code ('bess_roots') as '> besselj(0,bess_roots)'
gives zeros within the specified tolerance:

   1.0e-08 *

    0.5649
    0.5428
    0.0009
   -0.0005
   -0.1166
    0.1364

====== PART 2 ==================================================
```

2. Numerical solution for multiple polynomial roots

a. Modify the newton_exact.m function from the course repo to find all real roots of a polynomial. Ignore repeated roots, if any.

```
% SEE PART B

% clear data from problem 1, keep some things that do not change
clearvars -except toler maxIter
%{
% define test function
% testFun = @(x) x.^5-15*x.^4+85*x.^3-225*x.^2+274*x-120;
% testFunDeriv = @(x) 5*x.^4-60*x.^3+255*x.^2-450*x+274;
% numroots = 5; % manually/visually determined from polynomial

% figure
% fplot(testFun,[0 6])
% grid on
% axis([0.5 5.5 -4 4])
% from plot, roots appear to be near or at x = 1, 2, 3, 4, and 5
% bad_root_guesses = [0.5,1.5,2.5,3.5,4.5];

ends up with:
1st root @ x = 1
2nd root @ x = 2
3rd root @ x = 5
4th root @ x = 1
5th root @ x = 4
tangent lines at these x guesses very conveniently point toward other
 roots
easy to tell that these roots are wrong for this function. how to tell
 for
more complicated functions?
%}

% b. Modify a. to account for complex roots. Should semi-autonomously
 find
% all roots of a polynomial, real or complex. ASSUMES NO MULTIPLICITY
 OF
% ROOTS.

% Method: Characterize polynomial using Descartes' rule of signs. Use
% Newton's method to (hopefully) find a real root. If one is found,
 try
% different initial x-values to see if any other real roots can be
 found.
% Set domain for x-values to be [-10,10]. If no additional real roots
 are
% found, try a complex x-value

testFun = [1 -3 4 -2]; % coefficients of polynomial, should be real
lenF = length(testFun);
testFunDeriv = zeros(1,lenF-1);
sumDesc = [0;0]; % array to hold sign changes for Descartes' rule of
 signs
for j = 1:lenF-1
    testFunDeriv(j) = testFun(j)*(lenF-j); % derivative power rule

    % implement Descartes' rule of signs
    if testFun(j)*testFun(j+1) < 0 % if sign change for f(x)
        sumDesc(1) = sumDesc(1) + 1; % then note it in array
    elseif (testFun(j)*(-1)^(lenF-j)) ...
            *(testFun(j+1)*(-1)^(lenF-j)) < 0 % if sign change for f(-
x)
        sumDesc(2) = sumDesc(2) + 1; % then note it in array
    end
end

% sumDesc(1) is number of positive real roots (or less by an even
 integer),
% but if sumDesc(1) > 0, must be at least 1 positive real root. same
 logic
% applies to sumDesc(2), but for negative real roots.

%{
if (order of polynomial) - (number of real roots) is not divisible by
 2
with no remainder, then there is a single complex root, which is an
 error.
complex roots should come in conjugate pairs
%}
if mod((lenF-1)-sum(sumDesc),2) ~= 0
    error('Only one complex root found; should find multiple of 2')
end

roots = zeros(lenF-1,1);
numRealRoots = sum(sumDesc); % number of real roots
realRootsFound = 0;
rootsFound = 0;
domain = -10:10; % domain of initial x-values
k = 0; % internal counter
cmplx = 0;
%{
2 cases to handle
CASE 1: 3rd-order polynomial, 3 real roots
CASE 2: 3rd-order polynomial, 1 real root, 2 complex conjugate roots

1) check if new polynomial is of order 2 (if yes, break -> quadratic)
2) use Newton's method to find 1 real root
3) increment 'rootsFound' counter
4) add root to 'roots' array
5) perform polynomial deflation to get new polynomial
6) check if all real roots predicted by Descartes' rule of signs have
 been
   found
7) if no, create new real initial guess; if yes, create new complex
 guess
8) use Newton's method to find another root
9) if Newton's method results in same root, go to 4)
10) if new root, increment 'rootsFound' and add to 'roots' array

%}

while rootsFound < lenF - 3 % should have 3 terms remaining (a,b,c)
    % if all real roots have been found, reset counter for initial x
 guess;
    % alternatively, if the entire domain has been traversed, start
    % guessing complex initial values
    if (realRootsFound == numRealRoots) || k == length(domain)
        k = 1;
        cmplx = 1;
    end

    % generate new guess
```

```
        if cmplx == 0
            % keep guessing real initial x-values
            k = k+1;
            initX = domain(k);
        else
            % all predicted real roots have been found, guess complex
            k = k+1;
            initX = complex(domain(k));
        end

        % use modified version of newton_exact.m that allows for
     representation
        % of polynomials as vectors of coefficients instead of fn. handles
        [root,it,succ] = newton_exact_polyvec(testFun,testFunDeriv,...
            initX,maxIter,toler);

        % sum is zero if root has not already been found, nonzero
     otherwise
        if sum(abs(roots-root)<toler) ~= 0
            continue
        else
            % new, unique root; add to roots array
            roots(k) = root;
            if cmplx == 0
                % still finding real roots
                realRootsFound = realRootsFound + 1;
                rootsFound = rootsFound + 1;
            else
                % moved on to finding complex roots
                rootsFound = rootsFound + 1;
            end
        end

        [Q,R] = polydiv(testFun,root);

        % calculate new coefficient vectors
        testFun = Q;
        lenF = lenF-1;
        testFunDeriv = zeros(1,lenF-1);
        for j = 1:lenF-1
            testFunDeriv(j) = testFun(j)*(lenF-j); % derivative power rule
        end
    end

    % then solve for roots analytically using quadratic formula
    [qd1,qd2] = quadratic(testFun);

    roots(end-1) = qd1;
    roots(end) = qd2;

    fprintf('The roots of the polynomial are:\n\nx = ')
    for k = 1:length(roots)-1
    fprintf('%.4f + %.4fi, ',real(roots(k)),imag(roots(k)))
    end
```

```
fprintf('and %.4f + %.4fi\n',real(roots(end)),imag(roots(end)))

fprintf('\n====== PART 3
 =================================================\n')
```

```
The roots of the polynomial are:

x = 1.0000 + 0.0000i, 1.0000 + 1.0000i, and 1.0000 + -1.0000i

====== PART 3 =================================================
```

3. Polynomial deflation & use in root finding for high-order polynomials

```
clearvars -except testFun testFunDeriv maxIter toler

% a. Create a function that analytically solves a quadratic for roots
 (both
% real and complex): fn quadratic.m

% b. Write a polynomial division algorithm
polynomial = [1 -15 85 -225 274 -120]; % equation 4.28 from textbook
alpha = 5; % given divisor
[Q1,R1] = polydiv(polynomial,alpha);

fprintf('\nDividing the polynomial P(x) given by Eq 4.28 by a factor
 of (x-5)\n')
fprintf('(N = 5) results in a deflated polynomial Q(x) and a remainder
 R.\n')
fprintf('The coefficients of Q are: \n\n')
fprintf('Q = [ '); fprintf('%.4f ',Q1); fprintf('], and\n')
fprintf('R = %.4f, which implies that N = 5 is a root of P\n',R1)

% c. Find a root of P_n (x) using Newton's method, then deflate P by
 that
% root
numroots = 5;
roots = zeros(1,numroots);
root_guesses = [0.9 1.9 2.9 3.9 4.9];

% define vectors for coefficients of polynomial and its derivative
c = polynomial;
lenc = length(c);
d = zeros(1,lenc-1);
for j = 1:lenc-1
    d(j) = c(j)*(lenc-j); % derivative power rule
end

% deflate polynomial until we have a quadratic
for k = 1:lenc-3 % should have 3 terms remaining (a,b,c)
    % use modified version of newton_exact.m that allows for
 representation
```

```
    % of polynomials as vectors of coefficients instead of fn. handles
    [root,it,succ] = newton_exact_polyvec(c,d,root_guesses(k),...
        maxIter,toler);
    roots(k) = root;
    [Q,R] = polydiv(c,root);

    % calculate new coefficient vectors
    c = Q;
    lenc = lenc-1;
    d = zeros(1,lenc-1);
    for j = 1:lenc-1
        d(j) = c(j)*(lenc-j); % derivative power rule
    end
end

% then solve for roots analytically using quadratic formula
[qd1,qd2] = quadratic(c);

roots(end-1) = qd1;
roots(end) = qd2;

% note that the generated solution is already accurate to 6 decimal
 places,
% but if it were not, polished roots/solutions would be found as
 follows:

% define vectors for coefficients of polynomial and its derivative
testFun = polynomial;
lenF = length(testFun);
testFunDeriv = zeros(1,lenF-1);
for j = 1:lenF-1
    testFunDeriv(j) = testFun(j)*(lenF-j); % derivative power rule
end

polished = zeros(1,length(roots));
for k = 1:length(roots)
    [root,it,succ] =
 newton_exact_polyvec(testFun,testFunDeriv,roots(k),...
        maxIter,toler);
    polished(k) = root;
end
fprintf('\nPolished roots:\n')
fprintf('%.6f\n',polished)

fprintf('\n====== PART 4
 ==================================================\n')
```

```
Dividing the polynomial P(x) given by Eq 4.28 by a factor of (x-5)
(N = 5) results in a deflated polynomial Q(x) and a remainder R.
The coefficients of Q are:

Q = [ 1.0000 -10.0000 35.0000 -50.0000 24.0000 ], and
R = 0.0000, which implies that N = 5 is a root of P
```

```
Polished roots:
1.000000
2.000000
3.000000
5.000000
4.000000

====== PART 4 ===================================================
```

4. Multivariate root finding

```
fprintf('Should have left a little more time to do this part :/\n')
```

```
Should have left a little more time to do this part :/
```

Published with MATLAB® R2018b