ECE 540 **Spring 2018**

Project 1: Finding Roots and Computing Functions

Section 1: Bracketing Method

a) Code for this part can be found in Appendix A.

b) The command line and program output are also found in Appendix A. The first set of xStep andxOverlap used found two roots of 0 and 1.5874. Another two examples were run, showing how a root can be found multiple times or be missed completely.

Section 2: Taylor's Series and Newton-Raphson

a) Code for this part can be found in Appendix B.

b) In this section we will write out the Newton-Raphson algorithm for a function that can be used to solve for the log of a number. We start with equation 1

$$f(x)=b-e^{x} \tag{1}$$

The derivative of f(x), f'(x) is

$$f'(x)=-e^{x} \tag{2}$$

Using these in the Newton-Raphson algorithm we have

$$x_{i+1}=x_i-\frac{(b-e^{x^{x}})}{-e} \tag{3}$$

This can be simplified to

$$x_{i+1}=x_i-1+\frac{b}{e^{x_i}} \tag{4}$$

This equation will move towards log(b).

c) A function that uses equation 4 can be found in Appendix B.

d) The test script begins by computing the log of 3, 10 and 200, using the Taylor's Series (TS_log),Newton-Raphson (NR_log) and the built-in function (log). The absolute relative error between TS_log and log, and NR_log and log were computed and printed for each of the three values. The errors can be seen at the start of the program output section in Appendix B. The errors are all less than the precision of double precision numbers, which is good.

e) This part is about testing the time required to compute the log for a large range of x, using the three functions. The script for this is in Appendix B, along with the output for the program.

Project 1: Finding Roots and Computing Functions

In this output we see that three approaches all have reasonably low error, although log(x) had the lowest, as we might expect. As to the speed of the calculations log(x) was again the fastest, but interestingly, NR_log took about half of the time required by TS_log.

Appendix A: Code for Section 1 – Function Version 1

```
function roots = BiSectionSearch( xStart, xStop, xStep, xOverlap, Function )
%
% Function that uses Bisection to search for root of f(x) % Inputs: xStart and
xStep - the end points over which the search is to occur.
%         sStep - Size of intervals used in search.
%         xOverlap - Overlap between intervals.
%         Function - function to be searched.
% Outputs: roots - array of roots found in the search.
%


% Insure that xStart is less that xStop.
if  xStart > xStop     temp = xStart;
xStart = xStop;     xStop = temp;
end
% and that xStep is positive.
xStep = abs( xStep );
xOverlap = abs( xOverlap );
% set initial search points.
x0 = xStart; x1 = x0 +
xStep; xNext = x1-xOverlap;

% and roots is an empty matrix to start.
roots = []; while  x0 < xStop


    fx0 = Function(x0);
fx1 = Function(x1);
    % if f(x) at the initial end points have the opposite sign
    if fx0*fx1 < 0
        % Do search for root of critical point.
while abs(x0-x1)/abs(x0+x1) > 2*eps
x2 = (x0+x1)/2; % x2 is mid point.
fx2 = Function(x2); % compute f(x2)
            if fx0*fx2 > 0 % if fx0 and fx2 are of the same sign
x0 = x2; % replace x0 with x2.                 fx0 = fx2;
else
                if fx1*fx2 > 0 % if fx1 and fx2 are of the same sign
x1 = x2; % replace x1 with x2.                     fx1 = fx2;
else                     x0 = x2;                    x1 = x2;
end % of if replacing x2 or done.             end % of if replacing
x0         end % of while doing bisection         % save off root
that was found.         roots = [roots; x0];     end % if not a
bracket.

    % Move to next interval
    x0 = xNext;     x1 = x0 + xStep;
xNext = x1-xOverlap; end % of while
moving through interval.
```

Due to the simple nature of this test a set of command line entry were entered. The command lines and their output are show here.

```
>>roots = BiSectionSearch(-5,5,1.5,0.1,@(x)(x.^10-10*x.^5+exp(-x)-1) )

roots =             0
1.587387276961554
```

Project 1: Finding Roots and Computing Functions

```
>> roots = BiSectionSearch(-5,5,0.5,0.25,@(x)(x.^10-10*x.^5+exp(-x)-1) )

roots =            0
1.587387276961554
   1.587387276961554

>> roots = BiSectionSearch(-5,5,2.5,0.1,@(x)(x.^10-10*x.^5+exp(-x)-1) )

roots =   []
```

From these results we can see that if we make the intervals overlap too much, we will can find a root on multiple times. Of course if we make the interval to large, we can step over two roots and miss them.

A second version of the function was possible, as shown here.

```
function roots = BiSectionSearch2( xStart, xStop, xStep, xOverlap, Function )
%
% Function that uses Bisection to search for root of f(x) % Inputs: xStart and
xStep - the end points over which the search is to occur.
%         sStep - Size of intervals used in search.
%         xOverlap - Overlap between intervals.
%         Function - function to be searched.
% Outputs: roots - array of roots found in the search.
%

% Insure that xStart is less that xStop.
if  xStart > xStop     temp = xStart;
xStart = xStop;     xStop = temp; end
% and that xStep is positive.
xStep = abs( xStep );
xOverlap = abs( xOverlap );
% set initial search points.
x0 = xStart; x1 = x0 +
xStep; xNext = x1-xOverlap;
% and roots is an empty matrix to start. roots
= [];
% loop until interval moves past the end of the range.
while  x0 < xStop

    % Call BiSection, from class webpage,
    % to check out interval (x0, x1).     r = BiSection(x0, x1, Function ); %
Note print out removed from BiSection.m

    % if a solution was found,
if length(r) ~= 0
        roots = [roots; r(end)]; % save end value.
end
    % Move to next interval
    x0 = xNext;     x1 = x0 + xStep;
xNext = x1-xOverlap;    end % of while
moving through interval.
% return not necessary at end of function.
return;
```

As a test that the second version also works, the following single test was run.

Project 1: Finding Roots and Computing Functions

```
>> roots = BiSectionSearch2(-5,5,1.5,0.1,@(x)(x.^10-10*x.^5+exp(-x)-1))

roots =      0
1.587387276961553
```

Appendix B: Code for solving Section 2 – Function for part a)

```
function [y,terms] = TS_log( x, MaxTerms )

if margin < 2
    MaxTerms = 1000;
end % Normalize x to < 2, keeping track of correction
factor.
offset = 0;   % Correction factor.
while x > 1.8     x = x/2.0;  % Adjust x,     offset = offset +
0.69314718055994529; % Accumulate offset. end % of
normalization loop.
% Set up Taylor's Series.
y = 0;
x = x-1;  % TS for log uses x-1 everywhere x_k = x;  % Keeps x ^ k.
  MinusOne_k = 1; % Keeps (-1)^k terms = 1;    % loop counter and for
  dividing. yOld = y - 1; % initial Old value, allowing loop to
  start.
% Loop until solution does not change. while y ~= yOld  ...
% y not changing       && terms < MaxTerms  % not at maximum
number of terms.      yOld = y;  % Keep old value.     y = y
+ MinusOne_k*x_k/terms;  % add in next term     terms =
terms + 1;                 % Update counter,
    MinusOne_k = -MinusOne_k;  % (-1)^k and
x_k = x_k*x;               % x^k
end
% Add correction factor to output
y = y + offset;
```

Function for part c)

```
function [y,Iterations] = NR_log( x, MaxIterations)

% Set default on MatTerms
if margin < 2
    MaxIterations = 1000;
end
% Initial guess set to (exponent + 1) * log(2)
[~,e] = log2(x);
y = (e+1)*0.69314718055994529;
% Set old value to help start loop.
yOld = y - 1;
Iterations = 0; % no iterations yet.
% Loop until while abs(( y - yOld )/yOld) > eps ... % relative
change is small.
      && Iterations < MaxIterations              % and less than Max iterations.
yOld = y;                 % save old value     y = y - 1 + x * exp(-y);  %
Perform next iteration
    Iterations = Iterations + 1;        % Count number iteration.
end % of while loop
```

Script for testing the accuracy of TS_log, and NR_log.

```
% Script to compare the results from TS_log and NR_log
% Compute results for log(3), using three approaches.
```

Project 1: Finding Roots and Computing Functions

```
y1 = TS_log(3.0);
y2 = NR_log(3.0);
y = log(3.0);
% Print relative errors for each. fprintf('Relative Error for TS_log
for 3 = %g\n', abs((y1-y)/y) ); fprintf('Relative Error for NR_log
for 3 = %g\n\n', abs((y2-y)/y) );
% Compute results for log(10), using three approaches.
y1 = TS_log(10.0); y2 = NR_log(10.0); y = log(10.0);

% Print relative errors for each. fprintf('Relative Error for TS_log
for 10 = %g\n', abs((y1-y)/y) ); fprintf('Relative Error for NR_log
for 10 = %g\n\n', abs((y2-y)/y) );
% Compute results for log(200), using three approaches.
y1 = TS_log(200.0); y2 = NR_log(200.0); y = log(200.0);
% Print relative errors for each. fprintf('Relative Error for TS_log
for 200 = %g\n', abs((y1-y)/y) ); fprintf('Relative Error for NR_log
for 200 = %g\n\n', abs((y2-y)/y) ); % search for maximum error for
TS_log computation for 1 to 100000.
MaxError = 0.0; % Start error off as small.
NonConvergent = 0;

tic % start timer
% Loop x through test values.
for x = 1:(1/3):100000
 [y,terms] = TS_log( x ); % Testing built in function.
if terms < 1000
     error = abs( exp(y)-x )/x; % Save off maximum
     MaxError = max( MaxError, error );
 else      NonConvergent =
NonConvergent + 1;  end % of max error
check.
end % of loop through test values. time = toc; % Stop
timer and print time. fprintf( 'Elapsed time is %g
seconds.\n', time );

% search for maximum error for TS_log computation for 1 to 100000.
MaxError = 0.0; % Start error off as small.
NonConvergent = 0;

tic % start timer
% Loop x through test values.
for x = 1:(1/3):100000
 [y,terms] = NR_log( x ); % Testing built in function.
if terms < 1000
     error = abs( exp(y)-x )/x; % Save off maximum
     MaxError = max( MaxError, error );
 else      NonConvergent =
NonConvergent + 1;  end % of max error
check.
end % of loop through test values. time =
toc; % Stop timer and print time.
fprintf( 'Elapsed time is %g seconds.\n', time );
MaxError % Print out Maximum Error.
NonConvergent
 % search for maximum error for log computation for 1 to
100000.
MaxError = 0.0; % Start error off as small.
NonConvergent = 0;
terms = 0; tic %
start timer
% Loop x through test values.
```

Project 1: Finding Roots and Computing Functions

```
for x = 1:(1/3):100000  y = log( x ); %
Testing built in function.  if terms < 1000
     error = abs( exp(y)-x )/x; % Save off maximum
     MaxError = max( MaxError, error );
 else      NonConvergent =
NonConvergent + 1;  end % of max error
check.
end % of loop through test values. time =
toc; % Stop timer and print time.
fprintf( 'Elapsed time is %g seconds.\n', time );
MaxError % Print out Maximum Error.
NonConvergent
```

PROGRAM OUTPUT

```
Relative Error for TS_log for 3 = 0
Relative Error for NR_log for 3 = 0

Relative Error for TS_log for 10 = 1.92865e-16
Relative Error for NR_log for 10 = 0

Relative Error for TS_log for 200 = 0
Relative Error for NR_log for 200 = 1.67634e-16
```

Time,

error and number of nonconvergent for TS_log

```
Elapsed time is 17.973018 seconds.

MaxError =  2.7663e-15

NonConvergent =     0
```

Time, error and number of nonconvergent for NR_log

```
Elapsed time is 5.89898 seconds.

MaxError = 1.2044e-15

NonConvergent =     0
```

Time, error and number of nonconvergent for built-in log

```
Elapsed time is 0.581958 seconds.

MaxError =   1.0338e-15

NonConvergent =     0
```