2 ROOTS OF SINGLE NONLINEAR EQUATIONS

2.1 PRELIMINARIES

> *"A common problem encountered in engineering analysis is this: given a function f(x), determine the value of x for which f(x) = 0."* ([p. 143 Kiusalaas 2005)
>
> *"Nearly all functional relations occurring in practice are nonlinear. However, many nonlinear relationships can be described locally (in a more or less restricted domain) by linear models, with sufficient accuracy. Sill there are important phenomena such as saturation, solution branching, chaos etc. which can only be described by nonlinear models*." ([p. 272] Ueberhuber 1997)

Nonlinear problems in physics and engineering can be expressed symbolically as $f(\xi) = 0$, where f represents a *single* continuous real-valued function specified on a closed interval, [a, b], where $\xi \in [a, b]$ is termed a zero of the function, f. or, alternatively as the solution to the equation, $f(x) = 0$ ([p. 13] Engeln-Mullges & Uhlig 1996). Frequently, single nonlinear functions are expressed as $f(x) = g(x) - \beta = 0$, where g(x) is a *known* function of variable x, and β is some given constant parameter, e.g., $f(x) = x^2e^x - 1$, where $g(x) = x^2e^x$ and the parameter $\beta = 1$.

Systems of *n* nonlinear equations are represented by $n \geq 2$ –

$$\begin{cases} f_1(x_1,x_2,\ldots,x_n) = 0, \\ f_2(x_1,x_2,\ldots,x_n) = 0, \\ \qquad\vdots \\ f_n(x_1,x_2,\ldots,x_n) = 0, \end{cases}$$

employing

$$\underset{\rightarrow}{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}, \quad \underset{\rightarrow}{f} = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{pmatrix},$$

and, consequently, systems of nonlinear equations can be represented formally in a vector format by

$$\underset{\rightarrow}{f}(\underset{\rightarrow}{x}) = 0.$$

Examples of systems of nonlinear equations can be found in:

- flows in channels,
- circuit design, which require solutions of constant-coefficient, ordinary differential equations,
- behavior of elastic phenomena,
- vibration analysis,
- solutions to nonlinear partial differential equations, e.g., fluid dynamics.

In this course we are primarily interested in the solutions of *single* nonlinear equations which can be represented in many forms. For example, scalar nonlinear functions can be algebraic polynomials expressed here as

$$f(x) = \sum_{k=0}^{n} \alpha_k x^k,$$

where $a_n \neq 0$, and n is termed the degree of the polynomial; all other nonlinear equations which cannot be represented as such algebraic polynomials are termed *transcendental* ([p. 13] Engeln-Mullges & Uhlig 1996).

Roots of algebraic polynomials and transcendental nonlinear functions can be either real or complex, but complex roots are seldom of interest except for the case of algebraic polynomials describing say damped vibrations ([p. 143] Kiusalaas 2005). The present investigation will focus on solutions of transcendental nonlinear equation but I will discuss the solutions of algebraic polynomials in the final section 8.

Inevitably root finding employs iteration where some initial guess or trial solution is systematically refined until a predefined convergence criterion is met. Success depends critically on the quality of that first guess. The crucial selection of the trial solution depends, not on the choice of a numerical solver, but on the fundamental understanding of the problem at hand ([p. 341] Press et al. 1992; [p. 144] Kiusalaas 2005).

Please note the following -

> *"In general, there is no guarantee that an equation has a unique solution. ⋯ Theory says that either the solver will converge to a root or it will fail in some well-defined manner. No theory can say that the iterations will converge to a solution that you want*" ([p. 20] Kelley 2003)

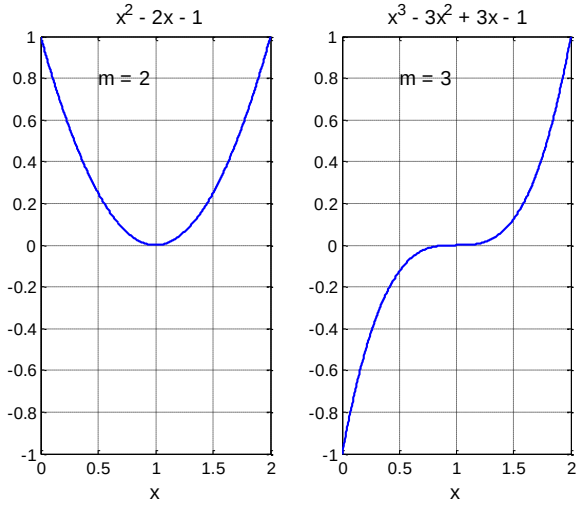
In *general*, single nonlinear transcendental equations can possess any number of roots (e.g. [p. 218] Heath 2002):

- $f(x) = e^x + 1$ has no solutions,
- $f(x) = e^{-x} - x$ has a single solution,
- $f(x) = x^2 - 4\sin x$ has two solutions,
- $f(x) = \sin(x)$ has an infinite number of solutions.

Moreover, single nonlinear equations can have multiple roots. Roots have multiplicity of *m* if

$$f(x) = df(x)/dx = d^2f(x)/dx^2 \ \ldots = df^m(x)/dx^m = 0.$$

In the presence of multiple roots convergence from a trial solution can be slow. For *odd* multiplicities f(x) changes sign at roots; for *even* multiplicities there are no sign changes at roots and thus bracketing algorithms, based on changes of the function sign do not work. Moreover, functions with zeros of high multiplicity are ill-conditioned with respect to function evaluations. Below the panel on the left illustrates a function with an even (m = 2) number of roots; the panel on the right shows a function with an odd (m = 3) number of roots –



2.1 INTERVAL BISECTION

If one has a *continuous* function, f(x), defined over $a \le x \le b$, such that $f(a)f(b) < 0$, then the intermediate value theorem of calculus states that *at least one* value of x can be found that satisfies $f(x) = 0$ in the domain ([p. 302] Ueberhuber 1997). The bisection algorithm starts with halving the initial interval. Setting $m = 0.5(a + b)$, one then determine which sub-interval contains a sign change. If $f(a)f(m) < 0$, the root occurs in the left-hand panel, then one resets $b = m$. Alternatively, if $f(m)f(b) < 0$, the root occurs in the right-hand panel, then one sets $a = m$. In either case the interval is now one half its original size. The iteration is continued. The appropriate sub-panel is halved again, and the process is continued until the panel width is less than a given, predefined tolerance. After n iterations the interval width must be $(b_o - a_o)/2^n$, where b_o and a_o refer to the boundaries of the original interval. The absolute error on the location of a root is $|a - b| < tol$; the number of iterations required to reach this uncertainty is $|\log_{10}[(b_o - a_o)/tol]/\log_{10}(2)|$.

Sample Bisection Calculation by *Hand* (Estimated root in red)

```
>> f = @(x) x.^3 + 2*x.^2 - 3*x - 1;    %  1 < x < 2
>> f([1 1.5 2])                          % root bounded by [1 1.5]
ans =   -1     2.375           9
>> f([1 1.25 1.5])                       % root bounded by [1 1.25]
ans =   -1      0.32813      2.375
>> f([1 1.125 1.25])                     % root bounded by [1.125 1.25]
ans =   -1    -0.41992       0.32813
>> (1.125 + 1.25)/2
ans =  1.1875
>> f([ 1.125 1.1875 1.25])               % root bounded by [1.1875 1.25]
ans = -0.41992    -0.067627      0.32813
>> ((1.125 + 1.25)/2 + 1.25)/2
ans = 1.2188
>> f([1.1875 1.2188 1.25])               % root bounded by [1.1875 1.2188]
ans =-0.067627      0.12504      0.32813
```

This manual iteration process can be implemented in the following for-loop if-construct, where the values of a and b are overwritten at each iteration -

```
for k = 1:kmax
  m = (a + b)/2;
     if sign(f(m)) == sign(f(b))    % logic employed in bisect.m  02/07/12
       b = m;
     else
       a = m;
     end
end % end for loop
```

Fifteen Iterations of Bisection Method Applied to $f(x) = x^3 + 2x^2 - 3x - 1$ for $a_o = 1$ & $b_o = 2$

```
>> bisect_script_1 (k_max = 15 using the above for loop/if construct):
Enclosing Interval     Approximation
(1.000000,2.000000)    1.500000
(1.000000,1.500000)    1.250000
(1.000000,1.250000)    1.125000
(1.125000,1.250000)    1.187500
(1.187500,1.250000)    1.218750
(1.187500,1.218750)    1.203125
(1.187500,1.203125)    1.195313
(1.195313,1.203125)    1.199219
(1.195313,1.199219)    1.197266
(1.197266,1.199219)    1.198242
(1.198242,1.199219)    1.198730
(1.198242,1.198730)    1.198486
(1.198486,1.198730)    1.198608
(1.198608,1.198730)    1.198669
(1.198669,1.198730)    1.198700
```

Identical to Table 2.1 ([p. 63] Bradie 2006)

The previous for-loop code fragment is employed in **example_bisection_01.m** where the square root of 2.0 is calculated using the function, `f(x) = @(x) (x^2 - 2.0)`, $a_0 = 1.0$, $b_0 = 2.0$, and tol was set equal to eps of 2.22×10^{-16}, the relative floating-point accuracy. An **fprintf** statement was inserted after the line, m = (a + b)/2, in order to output to the command window a, b, and m, at each iteration step, as tabulated below -

```
>> example_bisection_01
example_bisection_01.m:

     Enclosing Interval                              Approximation
k =  1 (+1.0000000000000000, +2.0000000000000000)  +1.5000000000000000
k =  2 (+1.0000000000000000, +1.5000000000000000)  +1.2500000000000000
k =  3 (+1.2500000000000000, +1.5000000000000000)  +1.3750000000000000
k =  4 (+1.3750000000000000, +1.5000000000000000)  +1.4375000000000000
k =  5 (+1.3750000000000000, +1.4375000000000000)  +1.4062500000000000
k =  6 (+1.4062500000000000, +1.4375000000000000)  +1.4218750000000000
k =  7 (+1.4062500000000000, +1.4218750000000000)  +1.4140625000000000
k =  8 (+1.4140625000000000, +1.4218750000000000)  +1.4179687500000000
k =  9 (+1.4140625000000000, +1.4179687500000000)  +1.4160156250000000

k = 45 (+1.4142135623730496, +1.4142135623731065)  +1.4142135623730780
k = 46 (+1.4142135623730780, +1.4142135623731065)  +1.4142135623730923
k = 47 (+1.4142135623730923, +1.4142135623731065)  +1.4142135623730994
k = 48 (+1.4142135623730923, +1.4142135623730994)  +1.4142135623730958
k = 49 (+1.4142135623730923, +1.4142135623730958)  +1.4142135623730940
k = 50 (+1.4142135623730940, +1.4142135623730958)  +1.4142135623730949
k = 51 (+1.4142135623730949, +1.4142135623730958)  +1.4142135623730954
k = 52 (+1.4142135623730949, +1.4142135623730954)  +1.4142135623730951
k = 53 (+1.4142135623730949, +1.4142135623730951)  +1.4142135623730949

root =   1.4142135623730949 sqrt(2) =   1.4142135623730951
```

Note that in **example_bisection_01.m** the stopping criterion was the absolute error of the root location was less than a predefined tolerance, $|b - a| < tol$, here tol = eps. As noted by Bradie ([p. 62] 2006), other criteria are possible – limiting the *relative* error on the root location –

$$|a - b| < tol|b|,$$

or constraining the root value, itself –

$$|f(m)| < tol.$$

As Bradie ([p 64] 2006) observed no single stopping criterion *"works well in all cases"*.

In the following, the function **bisect.m** will be employed to derive roots via the bisection process. This functions possesses the calling sequence,

```
[x, iter] = bisect(f, range, tol)
```

where the input arguments are: f(x) is the input function, f(x), range = [a_o, b_o], which brackets a root, f(a)f(b) < 0, tol is an *optional* parameter which constrains the absolute error on the root location, |a - b| < tol (the default tol = 10^3eps). The output arguments are: *x* is the calculated root, and *iter* is the number of iterations required to constrain the location of a root be less than tol. At the command prompt type

```
>> help bisect
```

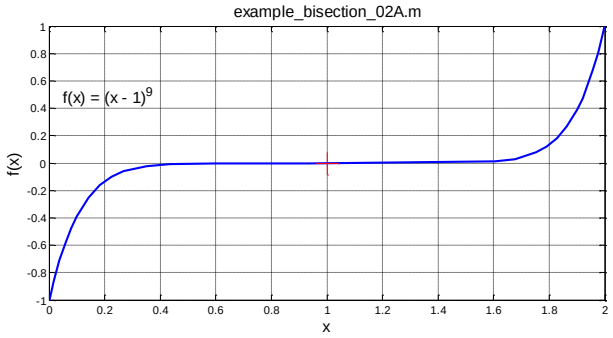
to ascertain calling sequence of a function. Here the resultant root values were represented as red crosses in the following f(x) plots.

Our first example function:

```
>> f = @(x) x.^3 +2*x.^2 -3*x - 1;
>> [x, k] = bisect(f, [1, 2]);
>> [x, f(x)]
ans = 1.198691243515953e+000   -2.726707748479385e-013
>> k
k = 44
```

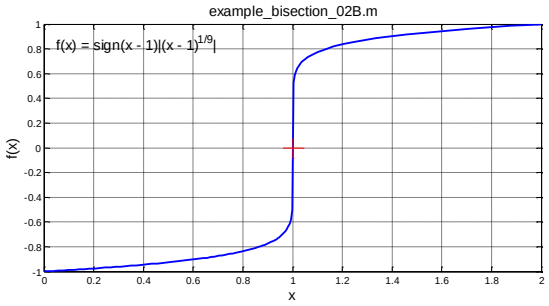
In **example_bisection_2A.m**, the root of f(x) = $(1 - x)^9$, is derived and its sensitivity to variations in tol is investigated. (Here a_o = -0.412 and b_o = 2.199 were chosen specifically to asymmetric around the root of 0.0.)

```
>> example_bisection_02A(10000*eps)
at m = +9.999999999997e-001 iter =  42  f(m) = -8.061165e-114
>> example_bisection_02A(1000*eps)
at m = +1.000000000000e+000 iter =  45  f(m) = -1.650032e-120
>> example_bisection_02A(100*eps)
at m = +1.000000000000e+000 iter =  48  f(m) = -6.770071e-132
>> example_bisection_02A(eps)
at m = +1.000000000000e+000 iter =  54  f(m) = +0.000000e+000
```



In **example_bisection_2B.m**, the root of f(x) = sign(1 - x)$(1 - x)^{1/9}$, $0 \le x \le 2$ is derived and its sensitivity to variations in tol is investigated. (Here a_o = -0.412 and b_o = 2.199 were chosen specifically to asymmetric around root of 0.0.)

```
>> example_bisection_02B(1000*eps)
at m = +1.000000000000e+000 iter =  45  f(m) = -3.320501e-002
>> example_bisection_02B(100*eps)
at m = +1.000000000000e+000 iter =  48  f(m) = -2.402285e-002
>> example_bisection_02B(10*eps)
at m = +1.000000000000e+000 iter =  52  f(m) = +1.822702e-002
>> example_bisection_02B(eps)
at m = +1.000000000000e+000 iter =  54  f(m) = +0.000000e+000
```



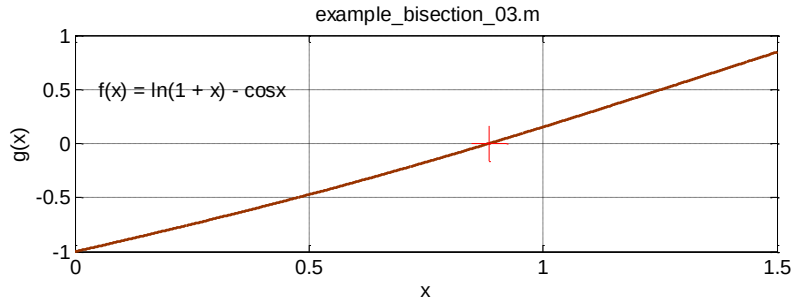
The bisection approach is *certain* to converge, but it is slow as we shall see that for smooth functions, other algorithms, e.g., the Newton's method, can achieve the required error tolerance, tol, in less than ten steps, if the initial guess is sufficiently accurate. However, the use of bisection is appropriate for root finding for badly behaved ($df/dx = 0$ & $d^2f/dx^2 = 0$) functions.

If f(x) is *not* continuous and the bracketed region encloses a singularity, the bisection approach converges on the singularity. For example, note the following –

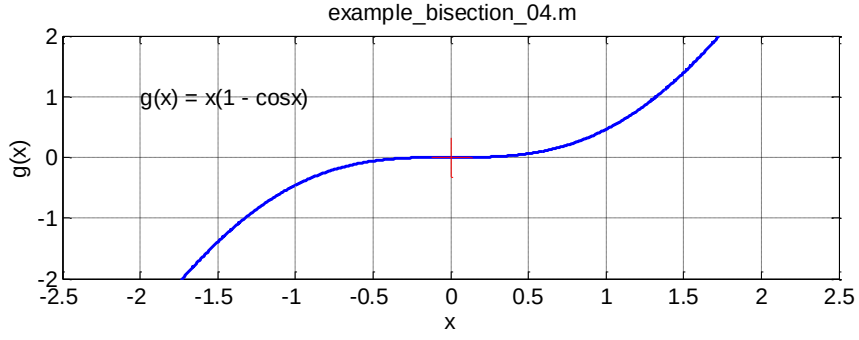
```
>> [x, k] = bisect(@tan, [0, 2])
x =1.570796326794891
k = 45
>> tan(x)
ans = 1.855183328867170e+014
```

Always plot the function, *if possible*, to identify *approximately* where a root occurs. For example, inspect the following examples -

```
>> example_bisection_03
at m = +8.845106161658e-001 iter =  47  f(m) = +0.000000e+000
```




```
>> example_bisection_04
% NOTE this function possesses a multiplicity of 3 at the root of 0!
at m = +1.053670820136e-008 iter =  50  f(m) = +0.000000e+000
```



2.3. NEWTON'S METHOD

This method uses more information than does the bisection method; this approach employs evaluations of *both* the function and its *derivative*. This method is based on a Taylor series of f(x) about the root value x ([p. 155] Kiusalaas 2005)

$$f(x_{k+1}) = f(x_k) + (x_{k+1} - x_k)f'(x_k) + (x_{k+1} - x_k)^2 f''(x_k)/2! + \cdots$$

If x_{k+1} is identified as the root of $f(x_{k+1}) = 0$, then this equation becomes

$$0 = f(x_k) + (x_{k+1} - x_k)df(x_k)/dx + (x_{k+1} - x_k)^2 d^2 f(x_k)/dx^2/2!$$

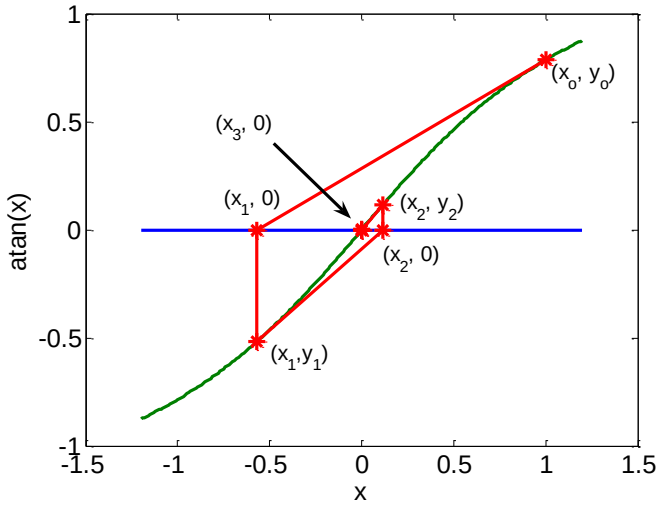
Assuming that x_k is close to x_{k+1}, the quadratic term in $(x_{k+1} - x_k)$ can be discarded and one solves for x_{k+1}

$$x_{k+1} \cong x_k - \frac{f(x_k)}{df(x_k)/dx}.$$

One must supply an initial guess, x_0, to start the iterative Newton process.

The tabulation below, and the associated plot, illustrate the first three iterations of Newton algorithm applied to f(x) = atan(x) and its df/dx = 1/(1 + x^2) for x_0 = 1 ([p. 3] Kelley 2003) -

```
Iteration        x value
  k =    0   x = +1.00000000000000e+000
  k =    1   x = -5.70796326794897e-001
  k =    2   x = +1.16859903998913e-001
  k =    3   x = -1.06102211704472e-003
```



For Newton's algorithm to be applicable:

- function, f(x), must be sufficiently *smooth* - Lipschitz continuous ([p. 17] Kelley, 2003). Thus, the function must *not* contain:
  - Non-differentiable functions – absolute values, sign functions, or fractional powers.
  - Internal interpolations from tables of data.
- the derivative, f'(x), must be calculated *accurately* and *analytically*,
- the initial guess must be *close* to the actual solution,
- the function *actually* has a root!

If you cannot derive df(x)/dx, observe that you can use MATLAB to do analytic derivations using the *Symbolic Math Toolbox.* This toolbox augments the numeric capabilities of MATLAB and it can be accessed *directly* from the command line. You perform analytic math operations through the creation of *symbolic objects*. Symbolic objects are a special data type inaugurated by the Symbolic Math Toolbox software. These objects allow you to perform mathematical operations in the MATLAB workspace analytically, without calculating numeric values.

I suggest that once you have created an analytic derivative, remove the symbolic objects from the desktop via the **clear** command, then represent these derivatives as anonymous functions.

Remember, before you can create a derivative you *must* create a symbolic object via the **syms** command!

```
                        % MATLAB 7.12.0.635 (R2011a)
>> syms x fs  dfs       % Employ syms x  OR x = sym('x')
>> fs = 3*x^4 - 6*x^2 + x*sin(pi*x);  % DO NOT USE . OPERATORS HERE
>> dfs = diff(fs, x);
 >> df = matlabFunction(dfs);  % convert from symbolic to anonymous fn
>> diff(atan(x), x)
ans = 1/(x^2 + 1)
>> clear x f
>> df = @(x) sin(pi*x) - 12*x + 12*x^3 + pi*x*cos(pi*x);
>> dg = @(x) 1/(x^2 + 1);
```

In the following examples, the function **newton** has the calling sequence -

```
[x, k] = newton(f, fprime, x0, tol)
```

where the input arguments are: f(x) is the input function, f(x), fprime(x) is the derivative function, x0 is the initial guest of the root value, tol is an *optional* parameter which constrains the absolute error on the root location, |a – b| < tol (the default tol = 10^3eps). The output arguments are: *x* is the calculated root, and *iter* is the number of iterations required to constrain the location of a root be less than tol.

Our first example function is the following –

```
>> f  = @(x) x.^3 +2*x.^2 -3*x - 1;
>> df = @(x) 3*x.^2 +4*x - 3;
>> [x,k] = newton(f, df, 1.5);
>> [x f(x)]
ans = 1.198691243515997e+000 0.0000000000000e+000
>> k
k = 6
```

Newton's method applied to multiple roots:

```
>> f = @(x) x^2 - 1.0; df = @(x) 2*x;          % Simple root (m = 1)
>> g = @(x) x^2 -2*x + 1; dg = @(x) 2*x - 2.0;% Multiple root (m = 2)
>> [xs, ks] = newtonM(f, df, 2.0, 1e06*eps);  % newtonM outputs all root estimates
>> [xm, km] = newtonM(g, dg, 2.0, 1e06*eps);
>> k = (1:6)'; [k xs(k)' xm(k)']
```

k	x_s	x_m
1.000000000000000	1.250000000000000	1.500000000000000
2.000000000000000	1.025000000000000	1.250000000000000
3.000000000000000	1.000304878048780	1.125000000000000
4.000000000000000	1.000000046461147	1.062500000000000
5.000000000000000	1.000000000000001	1.031250000000000
6.000000000000000	1.000000000000000	1.015625000000000

The x_s column (simple root) shows quadratic convergence but the x_m column (m = 2 root) illustrates linear converge (e.g. [p. 231] Heath 2002).

Now using the examples from the previous section, we find that

```
>> example_newton_03
example_newton_03.m:  x0 = +1.500
at m = +8.845106161659e-001 iter =   5  f(m) = +0.000000e+000

>> example_newton_04  % multiplicity of 3 at x = 0.0  ← NOTE slow convergence
example_newton_04.m:  x0 = +2.500
at m = +1.053670820136e-008 iter =  50  f(m) = +0.000000e+000

% Perverse example ([p. 121] Moler 2005)
>> example_newton_05
example_newton_05.m:  x0 = +3.000
at m =                 +Inf iter = 1002  f(m) = +Inf
```

Effect of varying th location of the start value, x_0, is now investigated -

Applying Newton's algorithm to the determination of a root of atan(x), starting with a guess, x_o = 1.0, produces:

```
>> df_dx = @(x)  (1 + x.^2).^(-1);
>> [x, k] = newton(@atan, df_dx, 1);
```

```
Iteration        x value
k =   1  x = -5.70796326794897e-001
k =   2  x = +1.16859903998913e-001
k =   3  x = -1.06102211704472e-003
k =   4  x = +7.96309604410642e-010
k =   5  x = +0.00000000000000e+000
k =   6  x = +0.00000000000000e+000
```

However, using a *much* poorer guess, x_o = 10.0, generates -

```
>> [x, k] = newton(@atan, df_dx, 10.0);
```

```
Iteration        x value
k =   1  x = -1.38583895104677e+002
k =   2  x = +2.98923207390070e+004
k =   3  x = -1.40352659289208e+009
k =   4  x = +3.09429109913164e+018
k =   5  x = -1.50398052679975e+037
k =   6  x = +3.55307441454522e+074
k =   7  x = -1.98302634370687e+149
k =   8  x = +6.17698923363772e+298
k =   9  x =                  -Inf
k =  10  x =                   NaN
```

Clearly, a disadvantage of this algorithm is that a root may drift far from where it should be, as seen here, if the initial guess is poor. Thus in this case the Newton algorithm can supply grossly inaccurate estimates for a value of a root.

Unlike bisection *Newton's method is not guaranteed* for arbitrary initial value x_0 (e.g. [p. 98] Bradie 2006).

How does Newton's method fare with polynomials?

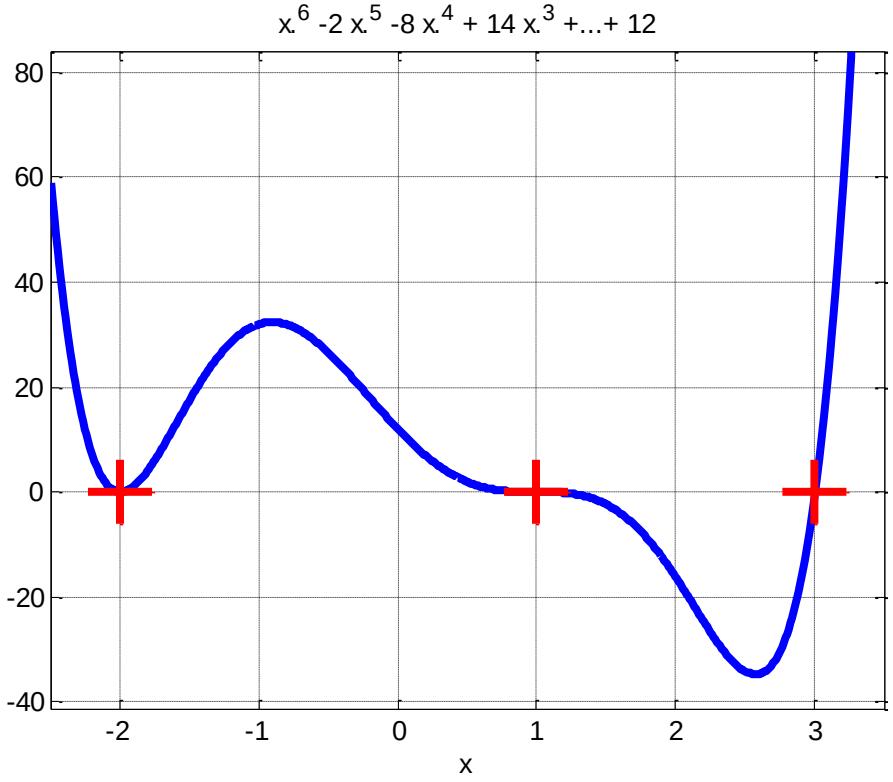
% 6th Order Polynomial Example <= [Newton's method NOT the best approach!]

```
>> g =  @(x)  x^6 -2*x^5 -8*x^4 + 14*x^3 + 11*x^2 -28*x + 12;
>> gprime = @(x) 6*x^5-10*x^4-32*x^3+42*x^2+22*x-28;
```

```
>> ezplot(g, [-2.5, 3.5])
>> grid on; hold on;

>> x = newton(g, gprime, 0.5)
k =  29   x =  0.99999463597284
>> plot(x, g(x), 'r+', 'MarkerSize', 16);
>> x = newton(g, gprime, 3.5)
k =  10   x =  3.00000000000000
>> plot(x, g(x), 'r+', 'MarkerSize', 16);
>> x = newton(g, gprime, -3.5)
k =  34   x = -2.00000000897261
>> plot(x, g(x), 'r+', 'MarkerSize', 16);
```

6th order equation *must* have 6 roots. Where are the other three roots? Are they complex or multiple? Inspection suggests multiple roots at x = 1 be investigated!



Acton (1990) and Ueberhuber (1997) suggest that, to avoid multiple zeros, one should solve the *modified* equation, defining the parameter, *u*, as

$$u(x) = \frac{f(x)}{f'(x)} \equiv 0,$$

and now solve for u(x) = 0 instead of f(x) = 0. One iterates solutions from u(x) and u'(x),

$$x^{k+1} = x^k - \frac{u(x^k)}{u'(x^k)} = x^k - \frac{f(x^k) f'(x^k)}{\left[f'(x^k)\right]^2 - f(x^k) f''(x^k)},$$

instead of

$$x^{k+1} = x^k - \frac{f(x^k)}{f'(x^k)},$$

the normal form for Newton's algorithm. This brute force approach to the derivation of multiple roots requires f''(x), as well as f'(x), be derived.

2.4. SECANT METHOD

Please inspect the **function [x, iter] = secant(f, range, tol),** which has the same calling sequence as the function **bisect.**

In many cases of practical interest either df/dx(x) does not exist in an analytic form or its determination incurs a high computational cost. The following method is similar to Newton's method, but it approximates the derivative in Newton's method by a truncated Taylor series, $f'(x) = (f(x_{k+1}) - f(x_k))/(x_{k+1} - x_k)$. Its convergence is intermediate between bisection and Newton's methods. Unfortunately, as with Newton's method, roots do not necessarily remain bracketed.

Our first example function is the following -

```
>> f = @(x) x.^3 +2*x.^2 -3*x - 1;
>> [x, k] = secant(f, [1, 2]);
>> [x f(x)]
ans =  1.198691243515997e+000    0
>> k
k = 8
```

For this function and the same start values the bisection algorithm required 44 iterations to reach the same level of accuracy.

Running the examples from the previous section, one finds that -

```
>> example_secant_03
at m = +8.845106161659e-001 iter =   7  f(m) = +0.000000e+000
>> example_secant_04
at m = -1.654361225106e-024 iter =  67  f(m) = -0.000000e+000
>> example_secant_05
at m = +2.000000000000e+000 iter =  66  f(m) = -1.219713e-007
>> example_secant_05(eps)
at m = +2.000000000000e+000 iter =  74  f(m) = +0.000000e+000
```

2.5. INVERSE QUADRATIC INTERPOLATION (IQI)

Please inspect the **function [x, iter] = iqi(f, range, tol),** which has the same calling sequence as the function **bisect.**

The secant method uses two points to estimate a root via a straight line. Why not employ three points and estimate a root via a parabolic fit? The problem with this approach is that a parabolic (quadratic) fit may find imaginary values, i.e., the fit does not intersect the x axis. The solution to this particular problem is to perform a quadratic fit as x as a function of y, x(y), which *always* intersects the x axis ([p. 123] Moler 2004).

The IQI method starts when the root of f(x) has been bracketed to the interval, $[x_1, x_2]$. Next a bisection step is performed, $x_3 = 0.5(x_1 + x_2)$. One has three points on the x(y). Thus a parabola is derived from

three points, (y_1, x_1), (y_3, x_3), and (y_2, x_2), creating a three-point Lagrange's interpolant through these points ([p. 104] Kiusalaas 2005) -

$$x(y) \equiv x_1 L_1(y) + x_2 L_2(y) + x_3 L_3(y),$$

$$L_1(y) = \frac{(y - y_2)(y - y_3)}{(y_1 - y_2)(y_1 - y_3)}, \quad L_2(y) = \frac{(y - y_1)(y - y_3)}{(y_2 - y_1)(y_2 - y_3)}, \quad L_3(y) = \frac{(y - y_1)(y - y_2)}{(y_3 - y_1)(y_3 - y_2)}.$$

Thus

$$x(y) = \frac{(y - y_2)(y - y_3)}{(y_1 - y_2)(y_1 - y_3)} x_1 + \frac{(y - y_1)(y - y_3)}{(y_2 - y_1)(y_2 - y_3)} x_2 + \frac{(y - y_1)(y - y_2)}{(y_3 - y_1)(y_3 - y_2)} x_3.$$

Setting y = 0, and simplifying ([p. 151] Kiusalaas 2005)

$$x(0) = \frac{y_2 y_3}{(y_1 - y_2)(y_1 - y_3)} x_1 + \frac{y_1 y_3}{(y_2 - y_1)(y_2 - y_3)} x_2 + \frac{y_1 y_2}{(y_3 - y_1)(y_3 - y_2)} x_3$$

$$= -\frac{y_2 y_3 x_1 (y_2 - y_3) + y_3 y_1 x_2 (y_3 - y_1) + y_1 y_2 x_3 (y_1 - y_2)}{(y_1 - y_2)(y_2 - y_3)(y_3 - y_1)}.$$

Finally rearranging, the estimate for the location of the root is

$$x_0 = x_3 + y_3 \frac{x_3 (y_1 - y_2)(y_2 - y_3 + y_1) + y_2 x_1 (y_2 - y_3) + y_1 x_2 (y_3 - y_1)}{(y_2 - y_1)(y_3 - y_1)(y_2 - y_3)}.$$

Since this implementation requires that $x_1 < x_3 < x_2$, interval boundaries are rearranged as x_0 is determined -

```
if x0 < x3
   x2 = x3; y2 = y3;
else
   x1 = x3; y1 = y3;
end
x3 = x0;
```

and a stopping criterion of $|x0 - x_3| < \text{eps}$ is employed in **iqi.m**.

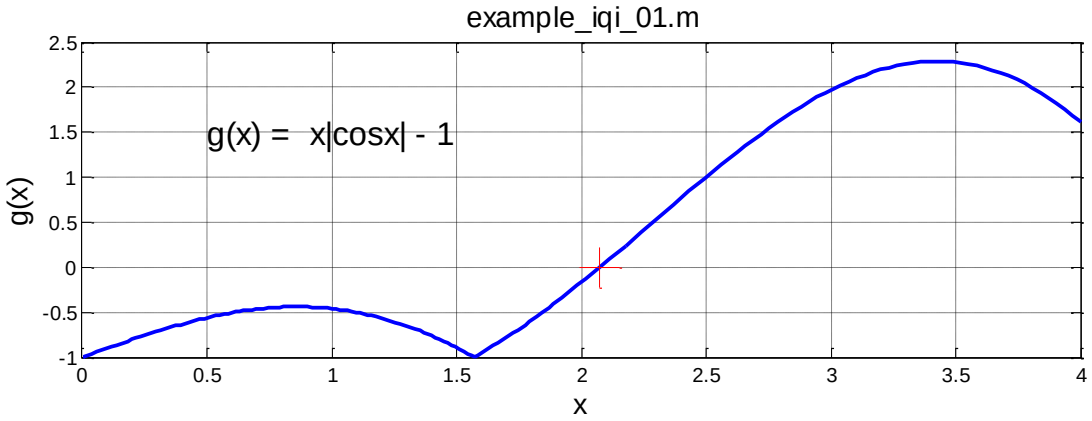
Our first example function is the following

```
>> f = @(x) x.^3 +2*x.^2 -3*x - 1;
>> [x, k] = iqi(f, [1, 2])
x = 1.198691243515997
k = 5
>> f(x)
ans = 0
```

For this function and the same start values the bisection algorithm required 44 iterations to reach the same level of accuracy.

Introducing another function, illustrated on the next page, one finds that

```
>> example_iqi_01
at m = +2.073932809091e+000 iter =   5  f(m) = +2.220446e-016
k =   1  x = 2.04835235489160
k =   2  x = 2.07468494487604
k =   3  x = 2.07393127045195
k =   4  x = 2.07393280909989
k =   5  x = 2.07393280909122
```



Employing our standard examples one finds that

```
>> example_iqi_03(eps)
at m = +8.845106161659e-001 iter =   4  f(m) = +0.000000e+000
>> example_iqi_04(eps)
at m = -6.617444900424e-024 iter =  64  f(m) = -0.000000e+000
>> example_iqi_05(eps)
at m = +2.000000000000e+000 iter =  98  f(m) = +0.000000e+000
```

2.6. BRENT'S ALGORITHM

Please inspect the **function [x, iter] = brentK(f, range, tol),** which has the same calling sequence as the function **bisect.**

This composite method combines robust, but slow bisection with high-order (fast) secant and quadratic interpolations. Brent *guarantees* convergence if the root is initially bracketed by the input values. The algorithm guards against interpolation procedures which generate values outside the bracketed regions; in this situation it disregards the interpolation estimate and then takes a bisection step.

The outline of this method ([p. 124] Moler 2004) is the following:

1. Use bracketing values such that sign(f(a)) ~= sign(f(b)).
2. Employ a secant step to supply a value *c* between *a* and *b*.
3. Arrange the values of *a*, *b*, and *c* such that –
   f(a)f(b) < 0,
   |f(b)| ≤ |f(a)|,
   c is the former *b* value,
4. If $c \neq a$, consider an IQI step.
5. If $c = a$, consider a secant step.
6. If either IQI or secant step is in the interval [*a b*], use it.
7. If the step is not in [*a b*], employ a bisection step.

This function is the recommended algorithm for general one-dimensional root finding ([p. 354] Press et al. 1992).

Employing our standard examples one finds that

```
>> example_brent_03(eps)
at m = +8.845106161659e-001 iter =   5  f(m) = +0.000000e+000
>> example_brent_04(eps)
at m = +6.911646535666e-006 iter =  42  f(m) = +1.650878e-016
>> example_brent_05(eps)
at m = +2.000000000000e+000 iter = 100  f(m) = +0.000000e+000
```

2.7. INTRINSIC MATLAB FUNCTION, `fzero`.

This algorithm is MATLAB's *implementation* of the Brent algorithm.

```
>> help fzero

fzero  Scalar nonlinear zero finding.
X = fzero(FUN, X0) tries to find a zero of the function FUN near X0. FUN ac-
cepts real scalar input X and returns a real scalar function value F evaluat-
ed at X.  The value X returned by fzero is near a point where FUN changes
sign (if FUN is continuous), or NaN if the search fails.

[X,FVAL]= FZERO(FUN,...) returns the value of the objective function, de-
scribed in FUN, at X.

[X,FVAL,EXITFLAG] = FZERO(...) returns a string EXITFLAG that describes the
exit condition of FZERO. If EXITFLAG is:
             > 0 then FZERO found a zero X
             < 0 then no interval was found with a sign change, or
                 NaN or Inf function value was encountered.
```

LIMITATIONS: The `fzero` command finds a point where the function changes sign. If the function is continuous, this is also a point where the function has a value near zero. If the function is discontinuous, `fzero` may return values that are discontinuous points instead of zeros. For example, consider `fzero(@tan, 1.0, eps)`.

```
>> [x] = fzero(@tan, 1, eps)
x = 1.570796326794898        ← π/2    % NOTE: Not return warning or error
>> [x, fv, flag] = fzero(@tan, 1, eps)
x =     1.570796326794898
fv =   -9.532973119934769e+014  % NOT 0.0
flag = -5     % NOT 1
```

Always use return variable flag and fv with fzero!

`fzero` returns a NaN, and an *error* message, if it cannot find a root:

```
>> fg = @(x) (abs(x) + 1);
>> fzero(fg, 3)
Exiting fzero: aborting search for an interval containing a sign change
because NaN or Inf function value encountered during search (Function
value at -Inf is Inf) Check function or try again with a different
starting value.  ans = NaN
```

Employing our standard examples one finds

```
>> f   = @(x) (x^2 - 2.0);  % example_bisection_01.m
>> a   = 1.0; b   = 2.0;
>> [x, fv, flag, out] = fzero(f, [a,b], eps);
>> x
x =   1.414213562373095
>> fv
fv = -4.440892098500626e-016
>> flag
flag = 1
>> out
out=intervaliterations: 0
            iterations: 7
             funcCount: 9
             algorithm: 'bisection, interpolation'
               message: 'Zero found in the interval [1, 2]'
```

```
>>  f  = @(x) (x - 1.0)^9;  % example_bisection_02A.m
a  = -0.412;  b  = +2.199;
>> [x, fv, flag, out] = fzero(f, [a,b], eps);
>> x
x =  1
>> fv
fv =   0
>> flag
flag = 1
>> out
out=intervaliterations: 0
            iterations: 141
             funcCount: 143
             algorithm: 'bisection, interpolation'
               message: 'Zero found in the interval [-0.412, 2.199]'
```

```
>>  g = @(x) (log(1.0 + x) - cos(x));   % example_bisection_03.m
    a  = 0.0; b  = 1.5;
>> x
x =  0.884510616165853
>> fv
fv = 1.110223024625157e-016
>> flag
flag = 1
>> out
out=intervaliterations: 0
            iterations: 7
             funcCount: 9
             algorithm: 'bisection, interpolation'
               message: 'Zero found in the interval [0, 1.5]'
```

```
>> g = @(x) x.*(1.0 - cos(x));   % example_bisect_04.m
   a  = -3.5; b  = +4.85;        % multiplicity of 3 at x = 0
>> [x, fv, flag, out] = fzero(g, [a,b], eps);
>> x
```

```
x =  -5.653934028213319e-009
>> fv
fv = 0
>> flag
flag = 1
>> out
out=intervaliterations: 0
            iterations: 79
             funcCount: 81
             algorithm: 'bisection, interpolation'
               message: 'Zero found in the interval [-3.5, 4.85]'
```

```
>> g = @(x) sign(x -2.0).*sqrt(abs(x - 2.0)); % example_bisect_05.m
   a  = -1.5;  b  = +5.70;
>> [x, fv, flag, out] = fzero(g, [a,b], eps);
>> x
x = 2.000000000000000
>> fv
fv = 2.107342425544702e-008
>> flag
flag = 1
>> out
out=intervaliterations: 0
            iterations: 29
             funcCount: 31
             algorithm: 'bisection, interpolation'
               message: 'Zero found in the interval [-1.5, 5.7]'
```

To find all the options for **fzero**:

```
>> options = optimset('fzero')
```

2.8. ZEROS OF POLYNOMIALS

In MATLAB, $P_n(x)$, a polynomial of degree *n* is represented in the form

$$P_n(x) = c_1x^n + c_2x^{n-1} + \cdots + c_nx + c_{n+1},$$

where the coefficients c_k can be either real or complex valued. Thus the coefficients of the following third-order polynomial

$$P_3(x) = x^3 - 2x - 5,$$

are

```
c = [1 0 -2 -5].          % Note the 0 value!
```

Note these following points from Press et al. ([p. 362] 1992). Observe that a polynomial of degree *n* has *n* roots; these roots can be real or complex. Moreover, they are not necessarily distinct. Note, if the coefficients a_k are *real*, complex roots occur in conjugate pairs, i.e. if a complex root $x_1 = r + wi$, where $i = \sqrt{-1}$, r is the real part of the complex root, and w is the imaginary part, then r - wi is also a root. In the case of complex coefficients, the complex roots are not necessarily related.

Multiple roots and closely spaced roots create significant difficulties for root-finding algorithms ([p. 362] Press et al. 1992). They remark, even for the simple case of a function $(x - a)^2$, with a double real root at x = a, bracketing methods, such as **fzero**, cannot work since function does *not* change sign at *a*, and slope-monitoring methods, such as **newton**, work poorly, with large roundoff errors, since *both* the function and its slope disappear at the multiple root.

2.8.1 Evaluating Polynomials

Note that polynomials can be evaluated for arbitrary x values from left to right via a simple for loop -

```
>> n = length(c) - 1;
>> p  = 0.0;
>>    for k = 1:n + 1
        p = p + c(k)*x^(n + 1 - k);
      end
```

However, the number of multiplications is reduced, and roundoff error is decreased, if a polynomial is evaluated in opposite order from right to left (e.g. [p 172] Kiusalaas 2005). For example,

$$P_4(x) = c_1x^4 + c_2x^3 + c_3x^2 + c_4x + c_5,$$

can be replaced by

$$P_4(x) = c_5 + x[c_4 + x[c_3 + x[c_2 + xc_1]]] \rightarrow$$
$$P_0(x) = c_1$$
$$P_1(x) = c_2 + xP_0(x)$$
$$P_2(x) = c_3 + xP_1(x)$$
$$P_3(x) = c_4 + xP_2(x)$$
$$P_4(x) = c_5 + xP_3(x).$$

Thus an n-degree polynomial can be evaluated *recursively* -

$$P_0(x) = c_1$$
$$P_k(x) = c_{1+k} + xP_{k-1}(x), \quad k = 1{:}n.$$

Similarly, evaluations of the first and second derivative of $P_n(x)$, required by the Laguerre method, are

$$P_0'(x) = 0,\ P_k'(x) = P_{k-1}(x) + xP_{k-1}'(x),$$
$$P_0''(x) = 0,\ P_k''(x) = 2P_{k-1}'(x) + xP_{k-1}''(x), \quad k = 1{:}n.$$

These recurse are employed in **evaluate_poly.m** -

```
%     p    = Pn(x)         polynomial evaluated at x
%     dp   = dPn(x)/dx     first derivative of polynomial evaluated at x
%     d2p  = d2Pn(x)/dx2   second derivative of polynomial evaluated at x
% Kiusalaas, J. 2005, Numerical Methods in Engineering in MATLAB [p. 173]
%  [Cambridge: Cambridge, UK]

   n   = length(c) - 1;
   p   = c(1);
   dp  = 0.0; d2p = 0.0;
    for k = 1:n
```

```
    d2p = 2.0*dp + x*d2p;
    dp  = p + x*dp;
    p   = p*x + c(k + 1);
end
```

2.8.2 Deflation of Polynomials

In the determination of the roots of a polynomial the overall effort is reduced by employing *deflation* ([p. 362] Press et al. 1992). When each root, *a*, is derived, the original polynomial is factored into a product of a reduced order polynomial and a term involving the root *a*, $P_n(x) = (x - a)P_{n-1}(x)$. Since the roots of $Q_{n-1}(x)$ are *exactly* the remaining roots of $P_n(x)$, the work in finding the roots of $P_n(x)$ declines as the order of reduced polynomial decreases ([p. 362] Press et al. 1992).

After the root a of $P_n(x) = 0$ has been determined, $P_n(x)$ is factored in the following way

$$P_n(x) = (x - a)P_{n-1}(x).$$

Expressing

$$P_{n-1}(x) = h_1x^{n-1}+h_2x^{n-2}+ \cdots + h_1,$$

then

$$c_1x^n + c_2x^{n-1}+ c_3x^{n-2} +\cdots + c_1 = (x - a)\left(h_1x^{n-1}+h_2x^{n-2}+ \cdots + h_1\right).$$

Equating the coefficients of the corresponding powers of x, one finds that

$$h_1 = c_1,\ h_2 = c_2 + ah_1,\ \ h_3 = c_3 + ah_2,\ \cdots h_n = c_n + ah_{n-1},$$

which are encapsulated in the function **deflation_poly.m**

```
n = length(c) - 1;
h = zeros(n, 1);
h(1) = c(1);
   for k = 2:n
       h(k) = c(k) + a*h(k - 1);
   end
```

2.8.3 Laguerre's Method

In the words of Press et al. ([p. 365] 1992) "*to motivate (although not rigorously derive) the Laguerre formulas we can note the following relations between the polynomial and its derivatives*" -

$$P_n(x) = (x - x_1)(x - x_2) \cdots (x - x_n)$$

$$\ln|P_n(x)| = \ln|x - x_1| + \ln|x - x_2| + \cdots + \ln|x - x_n|$$

$$\frac{d\ln|P_n(x)|}{dx} =+\frac{1}{x - x_1}+\frac{1}{x - x_2}+ \cdots +\frac{1}{x - x_n} = \frac{P_n'}{P_n} \equiv G(x).$$

Observe that absolute-value operator is absent from the definition of the parameter, G. Consequently, it may be either positive or negative ([p. 187] Acton 1990). Continuing, we see that

$$-\frac{d^2 \ln|P_n(x)|}{dx^2} = +\frac{1}{(x - x_1)^2} + \frac{1}{(x - x_2)^2} + \cdots + \frac{1}{(x - x_1)^2}$$
$$= \left[\frac{P_n'}{P_n}\right]^2 - \frac{P_n''}{P_n} \equiv H(x),$$

Thus the parameter H is *always* positive. Using these relations, Acton ([p. 187] 1990) makes what he terms "*a rather drastic set of assumptions*": that the root we search for, x_1, located at a distance, *a*, from the current guess for the root's location, *x*, and this root is *isolated* from all the remaining roots which are bunched at some distance, *b*, from x_1 (e.g. [p. 365] Press et al. 1992). Thus

$$x - x_1 = a,$$
$$x - x_k = b, \quad k = 2{:}n.$$

Consequently, we can represent the parameters *G* and *H* in terms of *a*, *b*, and *n*, as

$$\frac{1}{a} + \frac{n - 1}{b} = G,$$
$$\frac{1}{a^2} + \frac{n - 1}{b^2} = H,$$

which leads to the solution, solving the above quadratic equation, expressed as a function of 1/a,

$$a = \frac{n}{G \pm \sqrt{(n - 1)(nH - G^2)}},$$

where the sign in the denominator should be selected to make the denominator *larger* in the absolute sense. Observe that the quantity inside the square root can be negative, and, consequently, the root a can be complex.

Thus the iterative algorithm for finding roots of a polynomial via Laguerre's method is the following ([p. 188] Acton 1990; [p. 175] Kiusalaas 2005), given the coefficients, c, of polynomial, $P_n(x)$,

1. Choose a trial root x_0 and calculate, $P_n(x)$, $P_n'(x)$, and $P_n''(x)$, from **evaluate_poly.m**.
2. Then calculate G(x) and H(x) from $P_n(x)$, $P_n'(x)$, and $P_n''(x)$ in **laguerre.m**.
3. Derive the improved value of the root, *a*, from the above corresponding to the *larger* magnitude denominator.
4. Repeat step 2 and 3 until $|P_n(x)| <$ tol, or $|x - a| <$ tol.

Following this recipe one creates the following function, named, **laguerre.m** -

```
function x = laguerre(c, x, tol)

    if nargin < 2; tol = 1.0e3*eps; end

    n     = length(c);
    kmax  = 50;
      for k = 1: kmax
         [p, dp, d2p] = evaluate_poly(c, x);
```

```
        if abs(p)  < tol; return; end   % Exit if |Pn(x)| < tol
        G            = dp/p;
        H            = G*G - d2p/p;
        sq           = sqrt( (n - 1.0)*(n*H - G*G) );
          if abs(G + sq) > abs(G - sq)
            dx  = n/(G + sq);
          else
            dx  = n/(G - sq);
          end
       x = x - dx;
          if abs(dx) < tol; return; end  % Exit if |x - a| < tol
    end % end kth loop
x = Inf;
disp('laguerre.m: fell thru loop');
```

This function is called by the function, **roots_via_laguerre.m**, with the following calling sequence

```
r = roots_via_laguerre(c, tol)
```

which calculates *all* the roots, r, of a polynomial with the coefficients, c = [c_1; c_2; c_3; …, c_n]. Following Press et al. [p. 367], the initial trial guess for is zero. Once the first root is calculated using the above **laguerre.m**, the coefficient vector, c, is deflated to a length *n* employing **deflation_poly.m**. The function **laguerre.m** is applied to the deflated polynomial to determine the second root. This approach is continued until all the roots have been determined.

Now **roots_via_laguerre.m** is applied to some example polynomials-

```
% Bradie Example 2.18, p. 127]
>> c = [1; 2; 4; -2; -5];
>> r = roots_via_laguerre(c, 10*eps)
r =  -1.0000 + 2.0000i
     -1.0000 - 2.0000i
     +1.0000
     -1.0000

% Bradie Example 2.19 & 2.20, p. 130]
>> c =[ +16; +70; -169; -580; +75];
>> r = roots_via_laguerre(c, 10*eps)
r =  -5.0000
     +3.0000
     -2.5000
     +0.1250

% Bradie Example 2.21, p. 132]
>> format long
>> c = [1; -3.4; +5.4531; -4.20772; +1.50924; -0.20304];
>> r = roots_via_laguerre(c, 10*eps)
r =   1.000000000000000 + 1.000000000000000i
      1.000000000000000 - 1.000000000000000i
      0.480000000000255
      0.469999999999625
      0.450000000000120

% Bradie Example 2.21, p. 132]
>> a = [+4; -9; +3; +5; -3];
>> r = roots_via_laguerre(a, 10*eps)
r =   +1.000002211208340 + 0.000003829892696i
      -0.750000000000000 - 0.000000000000000i
      +1.000002211208320 - 0.000003829892696i
      +0.999995577583341
```

In section 3 the roots of a sixth-order polynomial were derived via Newton's method. Real roots were found at x = -2, +1, and +3. What were the remaining three roots?

```
>> clear
>> c = [1 -2 -8 +14 +11 -28 +12];
>> r = roots_via_laguerre(c, 10*eps)
r =   +3.000000000000000 - 0.000000000000000i
      -1.999999999980782 - 0.000000010290980i
      -2.000000000000043 + 0.000000004097850i
      +1.000000402235582 + 0.000000696667681i
      +1.000000402235124 - 0.000000696667681i
      +0.999999195529295


>> syms x f                          % create polynomial symbolically
>> f = (x + 3)^2 *(x - 2)^3;         % x = +2 multiplicity of 3
>> expand(f)                         % x = -3 multiplicity of 2
>> clear x, f

ans =  x^5 - 15*x^3 + 10*x^2 + 60*x - 72
>> c = [1; 0.0; -15; +10; +60; -72];
>> r = roots_via_laguerre(c, 10*eps)
r =   -3.000000000000000 - 0.000000009684359i
      +1.999994556588395 - 0.000000177926717i
      -2.999999992398868 + 0.000000000000012i
      +1.999998794741930 - 0.000002689634385i
      +1.999993161496112
```

2.8.4 Eigenvalue Method

As you may have learned in a linear algebra class, the eigenvalues of a matrix *A* are roots of the characteristic polynomial, P(x) = det[**A** - x**I**], where **I** is the identity matrix ([p. 368] Press et al. 1992). As is well known, root-finding is not an *effective* algorithm for determining eigenvalues ([p. 368] Press et al. 1992). However, the reverse is not true - efficient eigenvalue algorithms can be employed to determine roots of arbitrary polynomials. The nxn companion matrix, *C* (e.g. [p. 161] Heath 2002),

$$C = \begin{bmatrix} -c_2/c_1 & -c_3/c_1 & \cdots & -c_{n-1}/c_1 & -c_n/c_1 \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & 0 & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix}$$

has, as entries, the normalized coefficients of the polynomial (with sign reversed) in the first row, ones on the sub-diagonal, and zeros everywhere else; the eigenvalues of this companion matrix[1] are the roots of the c vector. The intrinsic function, **roots**, determines roots of a polynomial by this eigenvalue method, but it uses a *simple* calling sequence -

[1] One can calculate the roots of this companion matrix via the following MATLAB commands:

```
>> n = length(c);
>> A = diag(ones(n - 2, 1), -1);
>> A(1,:) = [-c(2:n)./c(1)]';
>> r = eig(A);
```

```
>> help roots
 ROOTS  Find polynomial roots.
    ROOTS(c) computes the roots of the polynomial whose coefficients
    are the elements of the vector c. If c has N+1 components,
    the polynomial is c(1)*X^N + ... + c(N)*X + c(N+1).
```

Observe that you need not know anything about eigenvalues and linear algebra but use the function, **roots**.

Employing examples from section 8.3 one sees that

```
% Bradie Example 2.18, p. 127]
>> c = [1; 2; 4; -2; -5];
>> r = roots(c)
r =  -1.000000000000001 + 2.000000000000002i
     -1.000000000000001 - 2.000000000000002i
     +1.000000000000000
     -1.000000000000000

>> % Bradie Example 2.19 & 2.20, p. 130]
c =[ +16; +70; -169; -580; +75];
>> r = roots(c)
r =  -5.000000000000005
     +3.000000000000003
     -2.500000000000000
     +0.125000000000000

>> % Bradie Example 2.21, p. 132]
c = [1; -3.4; +5.4531; -4.20772; +1.50924; -0.20304];
>> r = roots(c)
r =  1.000000000000000 + 0.999999999999999i
     1.000000000000000 - 0.999999999999999i
     0.479999999999730
     0.470000000000385
     0.449999999999885

>> % Bradie Example 2.21, p. 132]
a = [+4; -9; +3; +5; -3];
>> r = roots(a)
r =  -0.750000000000000
     +1.000008049507745
     +0.999995975246127 + 0.000006971024313i
     +0.999995975246127 - 0.000006971024313i

>> syms x f                        % create polynomial symbolically
>> f = (x + 3)^2 *(x - 2)^3;       % x = +2 multiplicity of 3
>> expand(f)                       % x = -3 multiplicity of 2
ans =  x^5 - 15*x^3 + 10*x^2 + 60*x - 72
>> c = [1; 0.0; -15; +10; +60; -72];

>> r = roots(c)
r =  -3.000000000000001 + 0.000000012342640i
     -3.000000000000001 - 0.000000012342640i
     +2.000004878243328 + 0.000008449428862i
     +2.000004878243328 - 0.000008449428862i
     +1.999990243513349
```

In section 3 the roots of a sixth-order polynomial were derived via Newton's method. Real roots were found at x = -2, +1, and +3. What were the remaining three roots?

```
>> clear
>> c = [1 -2 -8 +14 +11 -28 +12];
>> r = roots(c)
r =   +3.000000000000000
      -2.000000000000001 + 0.000000017548098i
      -2.000000000000001 - 0.000000017548098i
      +1.000007062508807
      +0.999996468745598 + 0.000006116299241i
      +0.999996468745598 - 0.000006116299241i
```

8.9. REFERENCES

Acton, F. S, 1990, *Numerical Methods That Work* (Mathematical Society of America: Washington DC).

Bradie, B. 2006, *A Friendly Introduction to Numerical Analysis* (Pearson, Prentice Hall: Upper Saddle River, NJ).

Engeln-Mullges, G. & Uhlig, F. 1996, *Numerical Algorithms with Fortran* (Springer-Verlag: Berlin).

Heath, M. T. 2002, *Scientific Computing: An Introductory Survey*, 2nd Ed. (McGraw-Hill: NY, NY).

Kelley, C. T. 2003, *Solving Nonlinear Equations with Newton's Method* (SIAM: Philadelphia).

Kiusalaas, J, 2005, *Numerical Methods in Engineering with MATLAB®* (Cambridge University Press, Cambridge, UK).

Moler, C. 2004, *Numerical Computing with MATLAB* (SIAM: Philadelphia).

Press, W. H., Teukolsky, S. A., Vetterling, W. T., & Flannery, B. P. 1992, *Numerical Recipes in Fortran: The Art of Scientific Computing, 2nd Ed.* (Cambridge University Press, Cambridge, UK).

Ueberhuber, C. W.1997, *Numerical Computation 2: Methods, Software, and Analysis* (Springer-Verlag: Berlin).

2.10 QUESTIONS

2.10.1 Review Questions

(a) What is a nonlinear equation?

(b) Is the bisection method (i) efficient? (ii) robust? Does it (iii) require a minimal amount of additional knowledge? (iv) require *the* function, f(x), to satisfy only minimum smoothness properties? (v) generalize easily to several functions in several variables?

(c) Answer similar questions for the Newton and secant methods.

(d) State at least two advantages and two disadvantages of Newton's method.

(e) State at least one advantage and one disadvantage of the secant method over Newton's.

(f) What is meant by a *bracket* for a nonlinear function in one dimension?

(g) What condition ensures that the bisection method will find a zero of a continuous nonlinear function *f* in the interval [a, b]?

2.10.2 Exercises

1. (a) If the bisection method for finding a zero of a function starts with an initial bracket of length 1, what is the length of the interval containing the root after six iterations?
(*b)* Do you need to know the particular function *f* to answer the question in part *a?*
(c) If we assume that it is started with a bracket for the solution in which there is a sign change, is the convergence rate of the bisection method dependent on whether the solution sought is a simple root or a multiple root? Why?

2. (a) What does it mean for a root of an equation to be a *multiple* root?
(b) What is the effect of a multiple root on the convergence rate of the bisection method?
(c) What is the effect of a multiple root on the convergence rate of Newton's method?

3. Which of the following behaviors are possible in using Newton's method for solving a nonlinear equation?
(*a)* It may converge linearly.
(b) It may converge quadratically.
(c) It may not converge at all.

4. What is the convergence rate for Newton's method for finding the root $x = 2$ of each of the
(a) $(x - 1)(x - 2)^2 = 0$
(b) $(x - 1)^2(x - 2) = 0$

5.. In using the secant method for solving a one-dimensional nonlinear equation,
(a) How many starting guesses for the solution are required?
(b) How many new function evaluations are required per iteration?

6. In bracketing a zero of a nonlinear function, one needs to determine if two function values, say *f(a)* and *f(b),* differ in sign. Is the following a good way to test for this condition: if *(f(a) f(b)* < 0) ...? Why?

7. List one advantage and one disadvantage of the secant method compared with the bisection method for finding a simple zero of a single nonlinear equation.

8. List one advantage and one disadvantage of the secant method compared with Newton's method for solving a nonlinear equation in one dimension.

9. Rank the following methods 1 through 3, from slowest convergence rate to fastest convergence rate, for finding a simple root of a nonlinear equation in one dimension:
(a) Bisection method
(b) Newton's method
(c) Secant method

10. What is meant by *inverse* interpolation? Why is it useful for root finding problems in one dimension?

11. Suggest a procedure for safeguarding the secant method for solving a one-dimensional nonlinear equation so that it will still converge even if started far from a root.

13. (a) How many zeros does the function

$$f(x) = \sin(10x) - x$$

have? (Hint: Sketching the graph of the function will be very helpful.) (b) Find all of the zeros of this function. (Hint: You will need a different starting point or initial bracketing interval for each root.)

14. Consider the problem of finding the smallest positive root of the nonlinear equation

$$\cos(x) + 1/(1 + e^{-2x}) = O.$$

Solve this problem using the starting point $x_o = 3$.

15. In celestial mechanics, *Kepler's equation*

$$M = E - e\sin(E)$$

relates the mean anomaly M to the eccentric anomaly E of an elliptical orbit of eccentricity e, where $0 < e < 1$. Solve Kepler's equation for the eccentric anomaly E corresponding to a mean anomaly of $M = 1$ (radians) and an eccentricity of e = 0.5.

16. In neutron transport theory, the critical length of a fuel rod is determined by the roots of the equation

$$\cot(x) = (x^2 - 1)/(2x).$$

Use a zero finder to determine the smallest positive root of this equation.

17. The natural frequencies of vibration of a uniform beam of unit length, clamped on one end and free on the other, satisfy the equation

$$\tan(x)\tanh(x) = -1.$$

Use a zero finder to determine the smallest positive root of this equation.

18. The vertical distance y that a parachutist falls before opening the parachute is given by the equation

$$y = \log(\cosh(t(gk)^{0.5}))/k,$$

where t is the elapsed time in seconds, $g = 9.8065$ m/s^2 is the acceleration due to gravity, and $k = 0.00341$ m^{-1} is a constant related to air resistance. Use a zero finder to determine the elapsed time required to fall 1000 m.

20. Find all the roots of the function

$$f(x) = \begin{cases} \text{sinx/x}, & x \neq 0.0, \\ 1.0, & x = 0.0, \end{cases}$$

in the interval [-10,10] for tol =10^{-7}.
[This function is special enough to have a name. It is called the **`sinc`** function.]

21. The function

$$f(x) = \alpha\cosh(x/4) - x$$

has two roots for α = 2 and none for α = 10. Is there an α for which there is precisely one root? If yes, then find such an α.

22. Verify that the following equations have a root in the interval [0 1]

a) $e^{-x} - x = 0$,

b) $\ln_e(1 + x) - \cos x = 0$,

c) $\cos x - x = 0$.

23. Observe that the following equations have a root in the provided interval.

a) $1 - \ln_e(x) = 0$, [2, 3], r = e.

b) $x^6 - 3 = 0$, [1, 2], $r = \sqrt[6]{3}$.

c) $x^3 + x^2 - 3x - 3 = 0$, [1, 2], $r = \sqrt{3}$.

24. The equation

$$x^2 - 1 + \cos(\sqrt{2}x) - \sin(\sqrt{2}x) = 0,$$

has two real roots. One of them is at x = 0. Determine the interval that contains this root and then approximate its value to six decimal places.

25. A function is

$$f(x) = \frac{x}{1+x^2} - \frac{500}{841}\left(1 - \frac{21x}{125}\right).$$

Find a root, accurate to six decimal places by Newton's method.

26. A function is

$$f(x) = 1.05 - 1.04x + \ln_e(x).$$

Find a root, accurate to six decimal places by Newton's method.

27 The following exercises examine the influence of start values on the efficiency of the secant method. In each exercise apply the secant method using the indicated start values. Iterate until the error is ≤ $5\text{x}10^{-7}$. Record and compare the final approximation and the number of iterations in each case.

a) $f(x) = \tan(\pi x) - x - 6 = 0$

i) [0.0 0.48], ii) [0.4 0.48], iii) [0.24 0.48].

b) $f(x) = x^3 + 2x^2 - 3x - 1 = 0$

i) [1 2], ii) [3 2], iii) [2 1], iv)[2 3].

28. Determine all the roots for the following polynomials. Use a tolerance of 5×10^{-11}.

a) $f(x) = +2x^5 - 6x^4 + 5x^3 + x^2 + 2$

b) $f(x) = -3x^6 + x^3 + 10x - 1$

c) $f(x) = +x^6 + x^5 - 9x^4 - 8x^3 + 29x^2 - 4x + 4$

d) $f(x) = 16x^4 - 40x^3 + 5x^2 + 20x + 6$

e) $f(x) = 10x^3 - 8.3x^2 + 2.295x - 0.21141$

29. The Chebyshev polynomials, $T_m(x)$, as a class of special functions. They satisfy a two-term recurrence relation –

$$T_{m+1}(x) = 2xT_m(x) - T_{m-1}(x),$$

with $T_0(x) = 1$ and $T_1(x) = x$.

a) Using this recurrence formula find $T_6(x)$.
b) Locate all the roots of the polynomial, $T_6(x)$.

30. The time dependence of a concentration, C(t), is

$$C(t) = \frac{3t^2 + t}{50 + t^3}.$$

At what t is C the greatest?

31. Use bisection to find the root of $f(x) = x^3 - 10x^2 + 5$ to a tolerance of 10^{-6} when the root is bracketed by [0.6 0.8].

32. Find all the zeros of f(x) x – tanx in the interval [0, 20] by the method of bisection.

33. Use inverse quadratic interpolation to find the root of $f(x) = x^3 - 10x^2 + 5$ to a tolerance of 10^{-6} when the root is bracketed by [0.6 0.8].

34. The smallest positive, nonzero root of coshxcosx -1 is in the vicinity of 4.5. Compute the root via Newton's method.

35. Determine the two roots of sinx + 3cosx – 2 = 0 that occur in the interval [-2, 2] via Newton's method.