Coding with MATLAB – Part II

Example 1. The following M-file is developed to implement an incremental search to locate the roots of a function `func` within the range from `xmin` to `xmax`. An optional argument `ns` allows the user to specify the number of intervals within the range. If `ns` is omitted, it is automatically set to `50`.

```
function xb = incsearch(func,xmin,xmax,ns)
% incsearch: incremental search root locator
% xb = incsearch(func,xmin,xmax,ns):
% finds brackets of x that contain sign changes
% of a function on an interval
% input:
% func = name of function
% xmin, xmax = endpoints of interval
% ns = number of subintervals (default = 50)
% output:
% xb(k,1) is the lower bound of the kth sign change
% xb(k,2) is the upper bound of the kth sign change
% If no brackets found, xb = [].
% You can define the function using @, for example f=@(x) 0.3*x-
sin(x)
if nargin < 3,
    error('at least 3 arguments required'),
end
if nargin < 4,
    ns = 50;
end %if ns blank set to 50
% Incremental search
x = linspace(xmin,xmax,ns);
f = func(x);
nb = 0;
xb = []; %xb is null unless sign change detected
for k = 1:length(x)-1
    if sign(f(k)) ~= sign(f(k+1)) %check for sign change
        nb = nb + 1;
```

```
        xb(nb,1) = x(k);
        xb(nb,2) = x(k+1);
    end
end
if isempty(xb) %display that no brackets were found
    disp('no brackets found')
    disp('check interval or increase ns')
else
    disp('number of brackets:') %display number of brackets
    disp(nb)
end
```

Example 2. The following M-file implements the bisection method. It is passed the function (`func`) along with lower (`xl`) and upper (`xu`) guesses. In addition, an optional stopping criterion (`es`) and maximum iterations (`maxit`) can be entered.

```
Function [root,fx,ea,iter]=bisect(func,xl,xu,es,maxit,varargin)
% bisect: root location zeroes
% [root,fx,ea,iter]=bisect(func,xl,xu,es,maxit,p1,p2,...):
% uses bisection method to find the root of func
% input:
% func = name of function
% xl, xu = lower and upper guesses
% es = desired relative error (default = 0.0001%)
% maxit = maximum allowable iterations (default = 50)
% p1,p2,... = additional parameters used by func
% output:
% root = real root
% fx = function value at root
% ea = approximate relative error (%)
% iter = number of iterations
if nargin<3,
    error('at least 3 input arguments required'),
end
test = func(xl,varargin{:})*func(xu,varargin{:});
if test>0,
    error('no sign change'),
end
if nargin<4|isempty(es),
    es=0.0001;
end
if nargin<5|isempty(maxit),
    maxit=50;
end
iter = 0;
xr = xl;
ea = 100;
while (1)
    xrold = xr;
    xr = (xl + xu)/2;
    iter = iter + 1;
    if xr ~= 0,
        ea = abs((xr - xrold)/xr)*100;
    end
    test = func(xl,varargin{:})*func(xr,varargin{:});
    if test < 0
        xu = xr;
    elseif test > 0
        xl = xr;
```

```
        else
            ea = 0;
        end
        if ea <= es | iter >= maxit,
            break,
        end
    end
    root = xr;
    fx = func(xr, varargin{:});
```

Practice at home: Develop your own M-file for bisection in a similar fashion. However, rather than using the maximum iterations and relative error, employ the absolute error as your stopping criterion to find the minimum number of iterations first. The `ceil` function must be used to round the least number of iteration up to the nearest integer. The first line of your function should be

```
function [root,Ea,ea,n] = bisectnew(func,xl,xu,Ead,varargin)
```

Note that Ead = the desired approximate absolute error and ea = the approximate percent relative error.

Example 3. The following M-file implements the Newton's method. It is passed the function (`func`), the derivative of function (`dfunc`), and initial guess (`xr`). In addition, an optional stopping criterion (`es`) and/or maximum iterations (`maxit`) can be entered.

```
function [root,ea,iter] =
newtraph(func,dfunc,xr,es,maxit,varargin)
% newtraph: Newton ? Raphson root location zeroes
% [root,ea,iter] = newtraph(func,dfunc,xr,es,maxit,p1,p2,...):
% uses Newton-Raphson method to find the root of func
% input:
% func = name of function
% dfunc = name of derivative of function
% xr = initial guess
% es = desired relative error (default = 0.0001%)
% maxit = maximum allowable iterations (default = 50)
% p1,p2,... = additional parameters used by function
% output:
% root = real root
% ea = approximate relative error (%)
% iter = number of iterations
if nargin<3
    error('at least 3 input arguments required')
end
if nargin<4|isempty(es)
    es=0.0001;
end
if nargin<5|isempty(maxit)
    maxit = 50;
end
iter = 0;
tic
while (1)
    xrold = xr;
    xr = xr - func(xr)/dfunc(xr);
    iter = iter + 1;
    if xr ~= 0
        ea = abs((xr - xrold)/xr)*100;
    end
    if ea <= es | iter >= maxit
        break
    end
end
toc
root = xr;
```

MATLAB `roots` function.

If you are dealing with a problem where you must determine a single real root of a polynomial, the techniques such as bisection and the Newton-Raphson method can have utility. However, in many cases, engineers desire to determine all the roots, both real and complex. Unfortunately, simple techniques like bisection and Newton-Raphson are not available for determining all the roots of higher-order polynomials. However, MATLAB has an excellent built-in capability, the roots function, for this task.

The roots function has the syntax,

```
>>x = roots(c)
```

where `x` is a column vector containing the roots and `c` is a row vector containing the polynomial's coefficients.

In fact for the polynomial equation

$$a_n x^2 + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 = 0$$

we define $c = [a_n, a_{n-1}, \dots, a_1, a_0]$ and run `x=roots(c)`. The output will be a vector of roots.

Example. For the equation

$$-3x^5 + 2x^2 - 6x - 15 = 0$$

try the following:

```
>> xr = roots([-3, 0, 0, 2, -6, -15])
```

What will you see if you run the following?

```
>> coef = poly(xr)
```

Note. The `roots` function has an inverse called `poly`, which when passed the values of the roots, will return the polynomial's coefficients.