6. ROOTS OF SYSTEMS OF NONLINEAR EQUATIONS

6.1 PRELIMINARIES

A system of *n* nonlinear equations can be represented in an equation-based format (n ≥ 2) -

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

However, employing vector notation,

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

then systems of nonlinear equations can be represented *formally* in a compact vector format by

$$\underline{\mathbf{f}}(\mathbf{x}) = 0.$$

Please note that a fundamental shortcoming of *all* algorithms for solving *systems* of nonlinear equations is that root bracketing, so effective in isolating the interval where roots of *single* nonlinear equations occur, does not exist in multiple space dimensions.

6.2 NEWTON'S METHOD FOR SYSTEMS OF NONLINEAR EQUATIONS

This method, applicable to computing roots in a single equation, was presented in chapter 2. This method is based on employing a truncated Taylor series about the known value x_k to estimate the value of unknown root, x_{k+1}, where $f(x_k + h) = 0$, and $h = x_{k+1} - x_k$,

Single (scalar) Equation:

$$f(x+h) \cong f(x) + f'(x)h \Rightarrow$$
$$f(x+h) \cong 0$$
$$f(x) + f'(x)h \cong 0$$
$$h = -f(x)/f'(x) \cong 0$$

Defining h,

$$h \equiv x_{k+1} - x_k = -f(x_k)/f'(x_k) \Rightarrow$$
$$x_{k+1} = x_k - f(x_k)/f'(x_k).$$

For a *system* of nonlinear equations *both* x & f(x) are vectors of the identical length, where x_k represents an n-dimensional vector at the k^{th} iteration state.

Vector Equation:

$$f(x) = \begin{bmatrix} f_1(x) \\ f_2(x) \\ \vdots \\ f_n(x) \end{bmatrix},$$

$$(x_{k+1} - x_k)f'(x_k) = -f(x_k), \text{ [single equation]}$$

$$(x_{k+1} - x_k)\bullet J(x_k) = -f(x_k), \text{ [analogous n-dimensional equation]}$$

$$J_{ij}(x_k) = \frac{\partial f_i}{\partial x_j}, \text{ [n-by-n matrix, termed Jacobian]}$$

Thus one solves following ***linear*** system at each k^{th} step to approximate *vector* x at the k + 1 step -

$$J(x_k)\bullet s_k = -f(x_k),$$
$$x_{k+1} = x_k + s_k,$$

Note that evaluating such a vector system is *costly* - determining *each* approximation, x_{k+1}, requires

- Calculating the Jacobian matrix, J, requires n^2 scalar evaluations,
- Solving the J-s linear system requires ~ $2n^3/3$ floating point operations at each iteration.

The disadvantages with employing the Newton method are:

- as was noted in the case of employing the Newton method to solve the case of a single nonlinear equation, the convergence of the solution vector depends on the accuracy of the initial guess in a very problem specific manner.
- analytic derivatives of *each* function with respect to *each* variable must be supplied to create n^2 terms of the Jacobian, which can be a very tedious task by hand.

The code encapsulated in **`newton_sys.m`** is listed below:

NOTE: the initial guess, x0, is a n-by-1 vector, the output solution, x, is also a n-by-1 vector, but the input parameter, Js, is a n-by-n matrix.

If one has a three-equation system,

$$f_1(x_1, x_2, x_3) = 0$$
$$f_2(x_1, x_2, x_3) = 0$$
$$f_3(x_1, x_2, x_3) = 0$$

$$J_s = \begin{bmatrix} \partial f_1/\partial x_1 & \partial f_1/\partial x_2 & \partial f_1/\partial x_3 \\ \partial f_2/\partial x_1 & \partial f_1/\partial x_2 & \partial f_2/\partial x_3 \\ \partial f_3/\partial x_1 & \partial f_3/\partial x_2 & \partial f_3/\partial x_3 \end{bmatrix}$$

```
  function  [x, fv, k] = newton_sys(fs, Js, x0, tol)
% function  [x, fv, k] = newton_sys(fs, Js, x0, tol)
% INPUT:
%          fs        = system of nonlinear equations
%          Js        = Jacobian of the system of nonlinear equations
%          x0        = initial guess
%          tol       = absolute accuracy of fit
```

```
% OUTPUT:
%          x         = root
%          fv        = fs(root)
%          k         = number of iterations
% Numerical Recipes in Fortran: The Art of Scientific Computing, 2nd Ed.
% Press, W H, Teukolsky, S A, Vetterling, W T, & Flannery, B p, (1992), p. 374
    x        = x0;   fv = nan(size(x0)); % dummy fs value
    iter_max = 50;
      for k = 1:iter_max
        f      = fs(x);
        J      = Js(x);
        err    = norm(f, 2);
        if err <= tol; fv = f; return; end
        s      = linsolve(-J, f); % Solve via intrinsic LU decomposition
        x      = x + s;
      end
 x = nan(size(x0));
 disp('newton_sys.m: Not converged');
```

As the first example let us attempt to determine the intercepts of the two following *implicitly defined* functions,

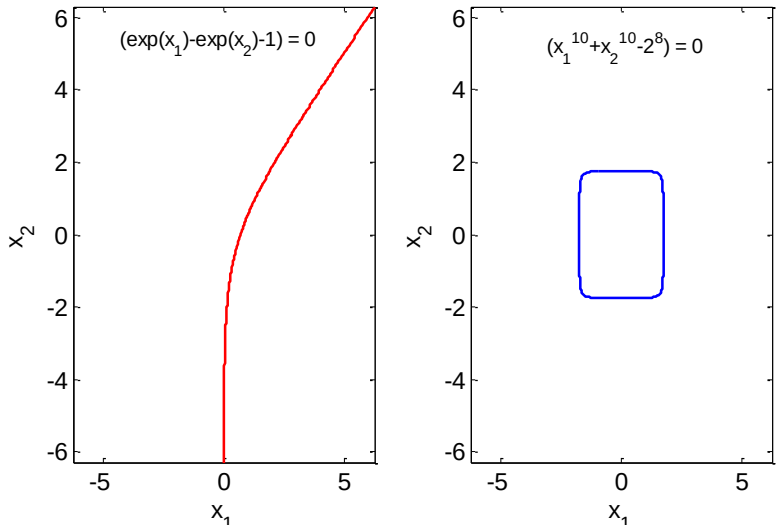
$$e^{x} - e^{y} - 1 = 0,$$
$$x^{10} + y^{10} - 2^{8} = 0.$$

In MATLAB how does one plot an implicit function, f(x, y) = 0? One uses the MATLAB built-in function, **ezplot.** The function, **ezplot,** graphs a function which depends on two *scalar* arguments, say x & y, or x1 & x2. Unfortunately, the input arguments cannot be a single vector, x, or two vector components, x(1), x(2)!

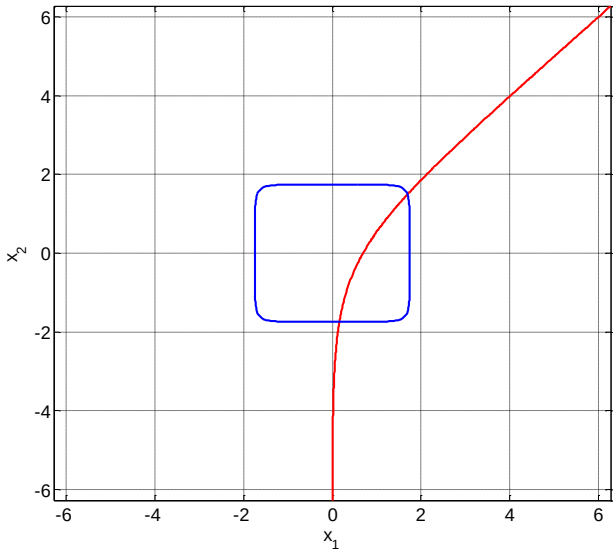
```
>> doc ezplot  % there one finds the following useful information
```

ezplot(fun2) plots fun2(x, y) = 0 over the default domain $-2\pi < x < 2\pi$, $-2\pi < y < 2\pi$.
ezplot(fun2,[x_{min}, x_{max}, y_{min}, y_{max}]) plots fun2(x,y) = 0 over $x_{min} < x < x_{max}$ and $y_{min} < y < y_{max}$.
ezplot(fun2,[min, max]) plots fun2(x, y) = 0 over min < x < max and min < y < max.

```
>> f2a = @(x1, x2) (exp(x1) - exp(x2) - 1);
>> f2b = @(x1, x2) (x1.^10 + x2.^10 - 2^8);
>> subplot(1, 2, 1); ezplot(f2a);  % Using the default x & y limits
>> subplot(1, 2, 2); ezplot(f2b);
>> figure(2); ezplot(f2a); hold on; ezplot(f2b); grid on;
```



```
>> figure(2); ezplot(f2a); hold on; ezplot(f2b); grid on;
```

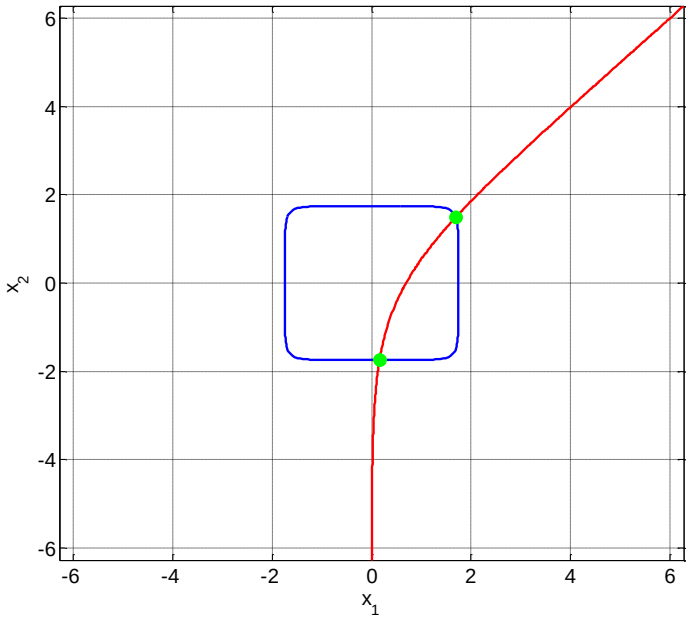


The root solvers employ vector (*not scalar*) input & output arguments:

$$x = \begin{bmatrix} x(1) \\ x(2) \end{bmatrix}, \; f(x) = \begin{bmatrix} f(1) \\ f(2) \end{bmatrix} = \begin{bmatrix} (\exp(x(1)) - \exp(x(2)) - 1) \\ (x(1)^{10} + x(2)^{10} - 2^8) \end{bmatrix},$$

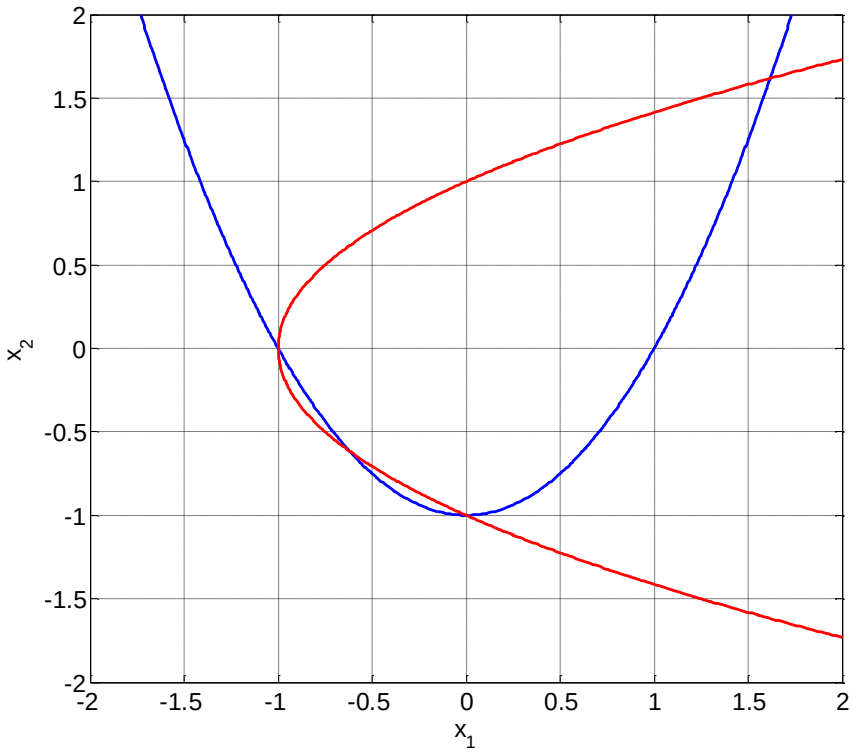
$$J = \begin{bmatrix} \partial f_1/\partial x_1 & \partial f_1/\partial x_2 \\ \partial f_2/\partial x_1 & \partial f_2/\partial x_2 \end{bmatrix} = \begin{bmatrix} \exp(x(1)) & -\exp(x(2)) \\ 10x(1)^9 & 10x(2)^9 \end{bmatrix}$$

```
>> f = @(x) [(exp(x(1)) - exp(x(2)) - 1); (x(1)^10 + x(2)^10 - 2^8)];
>> J = @(x) [exp(x(1)), -exp(x(2)); 10*x(1)^9,  10*x(2)^9];

>> x0A = [0.0 -2.0]';  % from inspection of above plot
>> [x, fv, k] = newton_sys(f, J, x0A, 1.0e-12);
>> xA
xA =  0.16155
      -1.7411
>> fv
fv =           0
     1.1369e-013
>> k
K  =  7


>> x0B = [1.8 1.8]';   % from inspection of above plot

>> [xB, fv, k] = newton_sys(f, J, x0B, 1.0e-12);
>> xB
xB =   1.6984
       1.4963
>> fv
fv =    8.8818e-016
       -5.6843e-014
>> k
k =  7

>> hold on; plot(xA(1), xA(2), 'go', xB(1), xB(2), 'go');
```



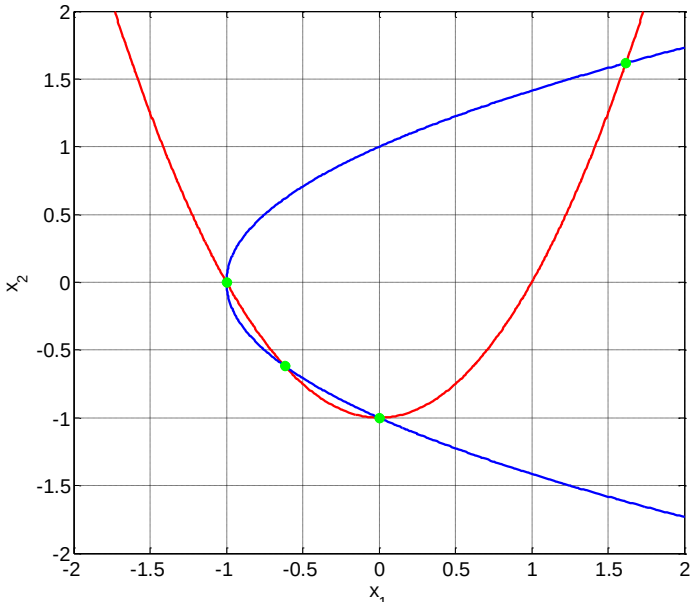
Determine intersects of $x^2 - y + \gamma = 0$ and $-x + y^2 + \gamma = 0$ for $\gamma = -1.0$.

```
>> gamma = -1.0;
>> f = @(x) [(x(1).^2 - x(2) + gamma); (-x(1) + x(2).^2 + gamma)];
>> J = @(x) [ 2.0*x(1) -1.0; -1.0 2.0*x(2)];

>> f1 = @(x1, x2) (x1.^2 -x2 +gamma);
>> f2 = @(x1, x2) (-x1 +x2.^2 +gamma);

>> ezplot(f1, [-2 2]); hold on; ezplot(f2, [-2 2]); grid on;
```




```
>> x0A = [-1 0]';          % By inspection of the above plot
>> x0B = [-0.6 -0.6]';
```

```
>> x0C = [0.0 -1.0]';
>> x0D = [1.6 1.6]';
>> [xA, fv, k] = newton_sys(f, J, x0A, 1.0e-12);
>> xA
xA =  -1
       0
>> fv
fv =   0
       0
>> k
k =  1

>> [xB, fv, k] = newton_sys(f, J, x0B, 1.0e-12);
>> xB
xB = -0.61803
     -0.61803
>> fv
fv = 0
     0
>> k
k =  4
>>
>> [xC, fv, k] = newton_sys(f, J, x0C, 1.0e-12);
>> xC
xC = 0
    -1
>> fv
fv = 0
     0
>> k
k =  1

>> [xD, fv, k] = newton_sys(f, J, x0D, 1.0e-12);
>> xD
xD =    1.618
        1.618
>> fv
fv = 0
     0
>> k
k =  4
>> hold on; plot(xA(1), xA(2), 'go', xB(1), xB(2), 'go', xC(1), xC(2), 'go')
```



6.3 BROYDEN (SECANT) METHOD

The above requirement for n^2 derivatives in the Jacobian, J(x), makes the Newton approach unusable for all but the smallest systems, This problem lead to the development of methods termed *quasi-Newton*, where the required partial derivatives are derived from the functions themselves. The following discussion/derivation of one of the more common quasi-Newton methods, the Broyden technique, is taken from Lindfield & Penny (2000).

The Broyden approach is based on solving -

$$(x_{k+1} - x_k) = -J(x_k)^{-1} f(x_k)$$

- An initial guess, x0, is assumed.
- An identity matrix is assumed to be the initial approximation for the inverse of the Jacobian, J^{-1}.
- At the kth step $p = J^{-1} * f(x_k)$.
- Then the k + 1 step Is determined -- $x_{k+1} = x_k + p$.
- Then vector norm, $|| f(x_{k+1})||$, is calculated. If this value is less than a predefined tolerance, *tol*, the routine is exited, otherwise the iteration continues.
- The following code updates the Jacobian -

$$y_k \equiv (f_{k+1} - f_k), \; p_k \equiv J_k^{-1} * f(x_k),$$

$$J_{k+1}^{-1} = \frac{J_k^{-1} - (J_k^{-1} * y_k - p_k) * p_k^T * J_k^{-1}}{p_k^T * J_k^{-1} * y_k}.$$

NOTE: the initial guess, x0, is a n-by-1 vector, the output solution, x, is also a n-by-1 vecto..

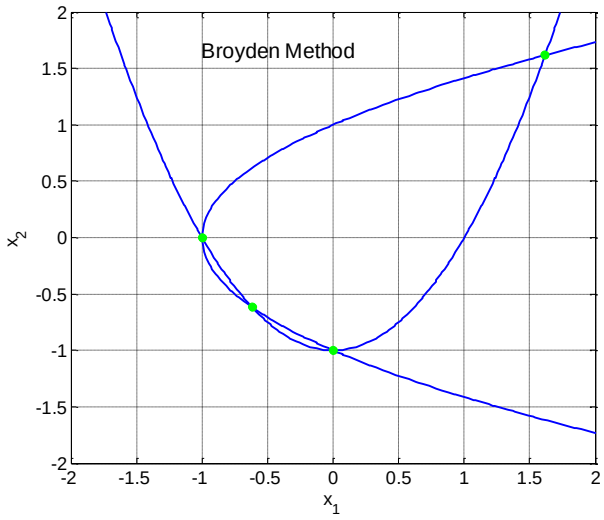
```
  function  [x, fv, k] = broyden_sys(fs, x0, tol)
% function  [x, fv, k] = broyden_sys(fs, x0, tol)
% INPUT:
%          fs        = system of nonlinear equations
%          x0        = initial guess
%          tol       = absolute accuracy of fit
% OUTPUT
%          x         = root
%          fv        = fs(root)
%          k         = number of iterations
% Numerical Methods Using MATLAB, 2nd Ed. G Linbdfield & J Penny
% 2000, (Prentice Hall: Saddle River, NJ).  [p. 153]
  fv = nan(size(x0));  % dummy fs value
  iter = 100;
  n    = length(x0);
  % NOTE: initial guess for Jacobian is eye(n)
  J    = eye(n);
  x    = x0;
     for k = 1:iter
        fn     = fs(x);
        pr     = -J*fn;
        x1     = x + pr;
        fn_old = fn;
        x      = x1;
        fn     = fs(x);
        % Update Jacobian
        y      = fn - fn_old;
        J_old  = J;
        old_yp = J_old*y - pr;
        pJ     = pr'*J_old;
            for i = 1:n
```

```
            for j = 1:n; M(i, j) = old_yp(i)*pJ(j); end
        end
        J      = J_old - M./(pr'*J_old*y);
        if norm(abs(fn)) < tol;  fv = fn; return; end
    end
  x = nan(size(x0));
 disp('Not converged');
```

The code **broyden_sys.m** was used to run the previous example -

```
>> x0A = [-1 0]';
>> x0B = [-0.6 -0.6]';
>> x0C = [0.0 -1.0]';
>> x0D = [1.6 1.6]';

>> [xA, fv, k] = broyden_sys(f, x0A, 1.0e-12);
>> xA
xA = -1
      0
>> fv
fv =  0
      0
>> k
k =   1

>> [xB, fv, k] = broyden_sys(f, x0B, 1.0e-12);
>> xB
xB = -0.61803
     -0.61803
>> fv
fv =  3.6193e-014
      3.6193e-014
>> k
k =  5

>> [xC, fv, k] = broyden_sys(f, x0C, 1.0e-12);
>> xC
xC = 0
    -1
>> fv
fv = 0
     0
>> k
k =  1

>> [xD, fv, k] = broyden_sys(f, x0D, 1.0e-12);
>> xD
xD =  1.618
      1.618
>> fv
fv =  0
      0
>> k
k =   5
```

The Broyden method took 1, 5, 1, and 5 iterations to determine these roots. The Newton method took 1, 4, 1, and 4 iterations to determine these roots with the identical starting values.



6.4 MATLAB INTRINSIC - **`fsolve`** [function in the *Optimization Toolbox*]

The *major* strength of **`fsolve`** is that it solves systems of nonlinear equations with much poorer initial guesses than **`broyden_sys.m`** or **`newton_sys.m`**. The following somewhat opaque discussion is based on MATLAB's on-line documentation. I suggest that you treat **`fsolve`** as a black box!

Limitations:
"The function to be solved must be *continuous*. When successful, fsolve only gives one root. fsolve may converge to a nonzero point, in which case, try other starting values. fsolve only handles real variables. When x has complex variables, the variables must be split into real and imaginary parts."

Syntax:

```
x = fsolve(fun,x0)                  % Starts at x0 & attempts to solve equations in fun.
x = fsolve(fun,x0,options)          % Solves fun with optimization parameters set in the
                                    structure option, Use optiset to set these parameters.
[x,fval] = fsolve(...)              % Returns fun(x) as well as solution x.
[x,fval,exitflag] = fsolve(...)     % Returns exitflag.
```

`exitflag` Describes the exit condition:

+1	Function converged to a solution x.
+2	Change in x was smaller than the specified tolerance.
+3	Change in the residual was smaller than the specified tolerance.
+4	Magnitude of search direction was smaller than the specified tolerance.
+0	Number of iterations exceeded MaxIter or number of function evaluations exceeded FunEvals.
-1	Output function terminated the algorithm.
-2	Algorithm appears to be converging to a point that is not a root.
-3	Trust region radius became too small (trust-region-dogleg algorithm) or regularization parameter became too large (levenberg-marquardt algorithm).
-4	Line search cannot sufficiently decrease the residual along the current search direction.

`output` Structure containing information about the optimization. Some of the fields of interest to us in the structure, **`output`**, are –

`iterations`	Number of iterations taken.
`funcCount`	Number of function evaluations.

algorithm	Algorithm used.

Options:
The function **optimset** can be employed to modify the values of the default parameters in the structure, **options.**

```
>> options = optimset('param1',value1,'param2',value2,...)
```

DiagnosticsDisplay	diagnostic information about the function to be minimized.
Display	Level of display. **'off'** displays no output; **'iter'** displays output at each iteration; '**final**' (default) displays just the final output.
Jacobian	If **'on', fsolve** uses a user-defined Jacobian (defined in fun), for the objective function. If **'off', fsolve** approximates the Jacobian using finite differences.
MaxFunEvals	Maximum number of function evaluations allowed.
MaxIter	Maximum number of iterations allowed.
TolFun	Termination tolerance on the function value.
TolX	Termination tolerance on x.
LargeScale	Use large-scale algorithm if possible when set to **'on'**. Use medium-scale algorithm when set to **'off'.** The default for **fsolve** is **'off.'**
NonlEqnAlgorithm	Choose Levenberg-Marquardt or Gauss-Newton over the trust region dogleg algorithm. **(LargeScale = 'off'.)**
LineSearchTypeLine	search algorithm choice. **(LargeScale = 'off'.)**

Simple Example:

```
>> opt   = optimset('TolFun', 1.0e-12, 'TolX',1.0e-12, 'Display', 'off');
>> gamma = -1.0;
>> f     = @(x)[ (x(1).^2 - x(2) + gamma); ( x(2).^2 - x(1) + gamma) ];
>> x0    = [-0.2 -1.2]';
>> [x1, fval, flag, output] = fsolve(f, x0, opt);
>> x1
x1 = 1.004e-015
              -1
>> fval
fval = 4.4409e-016
      -1.1102e-016
>> flag
flag = 1

>> x0   = [-0.75, -0.75]';
>> [x2, fval, flag, output] = fsolve(f, x0, opt);
>> x2
x2 = -0.61803
     -0.61803
>> fval
fval = -1.1102e-016
       -1.1102e-016
>> flag
flag = 1

>>  x0 = [-1.25 +0.25]';
>> [x3, fval, flag, output] = fsolve(f, x0, opt);
>> x3
x3 =   -1
  9.3947e-017
>> fval
fval =-1.1102e-016
            0
>> flag
flag = 1
>>
```

```
>> x0   = [1.5 1.5]';
>> [x4, fval, flag, output] = fsolve(f, x0, opt);
>> x4
x4 =    1.618
        1.618
>> fval
fval =-2.2204e-016
      -2.2204e-016
>> flag
flag = 1
```

Consider the following system -

$$\sin x_1 + x_2^2 + \ln_e x_3 - 7 = 0,$$

$$3x_1 + 2^{x_2} - x_3^3 - 1 = 0,$$

$$x_1 + x_2 + x_3 - 5 = 0,$$

```
>> fs4 = @(x) [( sin(x(1)) + x(2).^2 + log(x(3)) -7.0 );
        ( 3*x(1) + 2.^x(2) - x(3).^3 - 1.0)     ;
        ( x(1) + x(2) + x(3) - 5.0) ];
```

I guessed a start value of 1 -

```
>> opt = optimset('TolFun', 1.0e-12, 'TolX',1.0e-12, 'Display', 'off');
>> [x, fval, flag, output] = fsolve(fs4, [1 1 1]', opt);
>> x
x =    0.74927
       2.3867
       1.8641
>> fval
fval =
     0
     0
     0
>> flag
flag =
     1
```

Consider the system -

$$5\cos(x_1) + 6\cos(x_1 + x_2) - 10 = 0$$
$$5\sin(x_1) + 6\sin(x_1 + x_2) - 4 = 0.$$

```
fs10 = @(x) [(5*cos(x(1))+6*cos(x(1) + x(2))-10);
       (5*sin(x(1))+6*sin(x(1) + x(2)) -4)];
```

I guessed a start value of 1 -

```
>> opt = optimset('TolFun', 1.0e-12, 'TolX',1.0e-12, 'Display', 'off');
>> [x, fval, flag, output] = fsolve(fs10, [1 1]', opt);

>> x
x =   0.15598
      0.41114
>> fval
fval =
     0
     0
>> flag
```

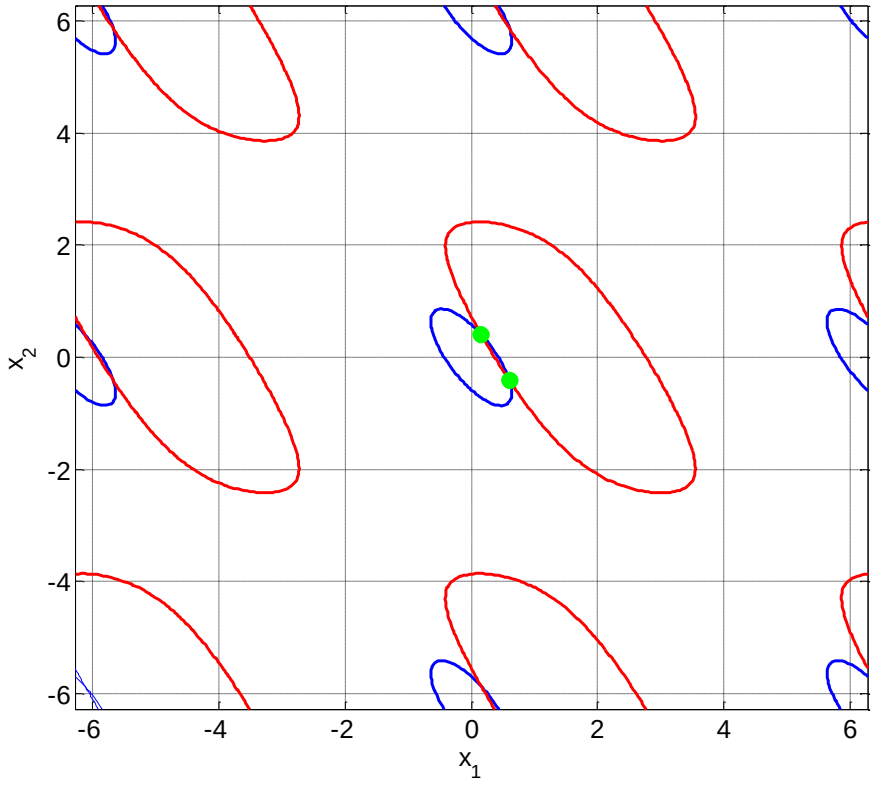
```
flag =

     1
```

Now I try a start value of 0:

```
>> [xB, fval, flag, output] = fsolve(fs10, [0 0]', opt);
>> xB
xB =  0.60503
     -0.41114
>> fval
fval = 0
       0
>> flag
flag = 1
```

How many roots are there? -- I counted seven roots in $-2\pi \le x \le +2\pi$ and $-2\pi \le y \le +2\pi$

```
>> fs10A = @(x1, x2) (5*cos(x1)+6*cos(x1+x2)-10);
>> fs10B = @(x1, x2) (5*sin(x1)+6*sin(x1+x2)-4);
>> ezplot(fs10A); hold on;   % blue
>> grid on;
>>  ezplot(fs10B); % red
>> hold on; plot(x(1), x(2), 'go')
>> hold on; plot(xB(1), xB(2), 'go')
```



6.5 REFERENCES

Lindfield, G. & Penny, J. 2000, *Numerical Methods Using MATLAB, 2nd Ed.* (Prentice Hall: Saddle River, NJ).

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

Quarteroni, A., & Saleri, F. 2003, *Scientific Computing with MATLAB* (Springer-Verlag: Berlin).

6.6 QUESTIONS

1. Solve the following the following system of nonlinear equations

$$\begin{aligned} 16x_1^4+16x_2^4+ x_3^4 &= 16 \\ x_1^2 + x_2^2+ x_3^2 &= 3 \\ x_1^3 - x_2 &= 0 \end{aligned}$$

   using Newton's method. As a starting guess, you make take each variable to be 1.0. NOTE exponent change.

2. The derivation of a Gaussian quadrature rule (which we will consider in a few weeks) on the interval [-1 +1] using the method of undetermined coefficients leads to the following system of nonlinear equations for the nodes x_1, x_2, and the weights w_1 and w_2:

$$\begin{aligned} w_1 + w_2 &= 2 \\ w_1x_1 + w_2x_2 &= 0 \\ w_1x_1^2 + w_2x_2^2 &= 2/3 \\ w_1x_1^3 + w_2x_2^3 &= 0. \end{aligned}$$

   Solve this system for x_1, x_2, w_1, and w_2. How many different solutions can you find?

3. Solve the following system of nonlinear equations:

$$\begin{aligned} \sin x_1 + x_2^2 + \ln_e x_3 &= 3 \\ 3x_1 + 2^{x_2} - x_3^3 &= 0 \\ x_1^2 + x_2^2 + x_3^3 &= 6. \end{aligned}$$

   Try to find as many different solutions as you can. You should find at least four real solutions (there are also complex solutions). NOTE change of constant.

4. Each of the following systems of nonlinear equations may present some difficulty in computing a solution. In some cases, the nonlinear solver fail may to converge

or may converge to a point other than a solution. When this happens, try to explain the reason for the observed behavior.

a)
$$x_1 + x_2(x_2(5 - x_2) - 2) = 13,$$
$$x_1 + x_2(x_2(1 + x_2) - 14) = 29,$$
starting from $x_1 = 15$, $x_2 = -2$.

b)
$$x_1^2 + x_2^2 + x_3^2 = 5,$$
$$x_1 + x_2 = 1,$$
$$x_1 + x_3 = 3,$$
starting from $x_1 = (1+\sqrt{3})/2$, $x_2 = (1-\sqrt{3})/2$, and $x_3 = \sqrt{3}$.

c)
$$x_1 + 10x_2 = 0,$$
$$\sqrt{5}(x_3 - x_4) = 0,$$
$$(x_2 - x_3)^2 = 0,$$
$$10(x_1 - x_4)^2 = 0,$$
starting from $x_1=1$, $x_2= 2$, $x_3= 1$, $x_4= 1$.

d)
$$10^4 x_1 x_2 = 1$$
$$e^{-x_1} + e^{-x_2} = 1.0001$$
starting from $x_1= 0$ and $x_2 = 1$.

e)
$$x_1 = 0$$
$$10x_1/(x_1+0.1) + 2x_2^2 = 0,$$
starting from $x_1=1.8$ and $x_2 = 0$.