Solution of non-linear equations

By Gilberto E. Urroz, September 2004

In this document I present methods for the solution of single non-linear equations as well as for systems of such equations.

Solution of a single non-linear equation

Equations that can be cast in the form of a polynomial are referred to as ***algebraic equations***. Equations involving more complicated terms, such as trigonometric, hyperbolic, exponential, or logarithmic functions are referred to as **transcendental equations**. The methods presented in this section are numerical methods that can be applied to the solution of such equations, to which we will refer, in general, as ***non-linear equations***. In general, we will we searching for one, or more, solutions to the equation,

$$f(x) = 0.$$

We will present the ***Newton-Raphson*** algorithm, and the ***secant*** method. In the secant method we need to provide two initial values of x to get the algorithm started. In the Newton-Raphson methods only one initial value is required.

Because the solution is not exact, the algorithms for any of the methods presented herein will not provide the exact solution to the equation $f(x) = 0$, instead, we will stop the algorithm when the equation is satisfied within an allowed tolerance or error, ε. In mathematical terms this is expressed as

$$|f(x_R)| < \varepsilon.$$

The value of x for which the non-linear equation $f(x)=0$ is satisfied, i.e., $x = x_R$, will be the solution, or root, to the equation within an error of ε units.

The Newton-Raphson method

Consider the Taylor-series expansion of the function $f(x)$ about a value $x = x_o$:

$$f(x)= f(x_o)+f'(x_o)(x-x_o)+(f''(x_o)/2!)(x-x_o)2+\ldots.$$

Using only the first two terms of the expansion, a first approximation to the root of the equation

$$f(x) = 0$$

can be obtained from

$$f(x) = 0 \approx f(x_o)+f'(x_o)(x_1 -x_o)$$

Such approximation is given by,

$$x_1 = x_o - f(x_o)/f'(x_o).$$

The Newton-Raphson method consists in obtaining improved values of the approximate root through the recurrent application of equation above. For example, the second and third approximations to that root will be given by

$$x_2 = x_1 - f(x_1)/f'(x_1),$$

and

$$x_3 = x_2 - f(x_2)/f'(x_2),$$

respectively.

This iterative procedure can be generalized by writing the following equation, where i represents the iteration number:

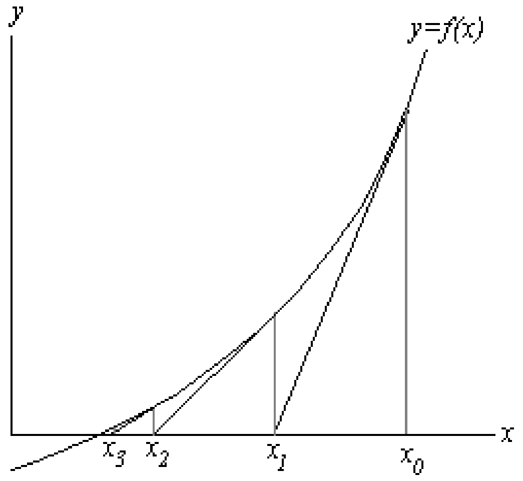
$$x_{i+1} = x_i - f(x_i)/f'(x_i).$$

After each iteration the program should check to see if the convergence condition, namely,

$$|f(x_{i+1})| < \varepsilon,$$

is satisfied.

The figure below illustrates the way in which the solution is found by using the Newton-Raphson method. Notice that the equation $f(x) = 0 \approx f(x_o) + f'(x_o)(x_1 - x_o)$ represents a straight line tangent to the curve $y = f(x)$ at $x = x_o$. This line intersects the x-axis (i.e., $y = f(x) = 0$) at the point x_1 as given by $x_1 = x_o - f(x_o)/f'(x_o)$. At that point we can construct another straight line tangent to $y = f(x)$ whose intersection with the x-axis is the new approximation to the root of $f(x) = 0$, namely, $x = x_2$. Proceeding with the iteration we can see that the intersection of consecutive tangent lines with the x-axis approaches the actual root relatively fast.



The Newton-Raphson method converges relatively fast for most functions regardless of the initial value chosen. The main disadvantage is that you need to know not only the function f(x), but also its derivative, f'(x), in order to achieve a solution. The secant method, discussed in the following section, utilizes an approximation to the derivative, thus obviating such requirement.

The programming algorithm of any of these methods must include the option of stopping the program if the number of iterations grows too large. How large is large? That will depend of the particular problem solved. However, any Newton-Raphson, or secant method solution that takes more than 1000 iterations to converge is either ill-posed or contains a logical error. Debugging of the program will be called for at this point by changing the initial values provided to the program, or by checking the program's logic.

A MATLAB function for the Newton-Raphson method

The function *newton*, listed below, implements the Newton-Raphson algorithm. It uses as arguments an initial value and expressions for f(x) and f'(x).

```
function [x,iter]=newton(x0,f,fp)
% newton-raphson algorithm
N = 100; eps = 1.e-5; %  define max. no. iterations and error
maxval = 10000.0;     %  define value for divergence
xx = x0;
while (N>0)
   xn = xx-f(xx)/fp(xx);
   if abs(f(xn))<eps
      x=xn;iter=100-N;
      return;
   end;
   if abs(f(xx))>maxval
      disp(['iterations = ',num2str(iter)]);
      error('Solution diverges');
      break;
   end;
   N = N - 1;
   xx = xn;
end;
error('No convergence');
break;
% end function
```

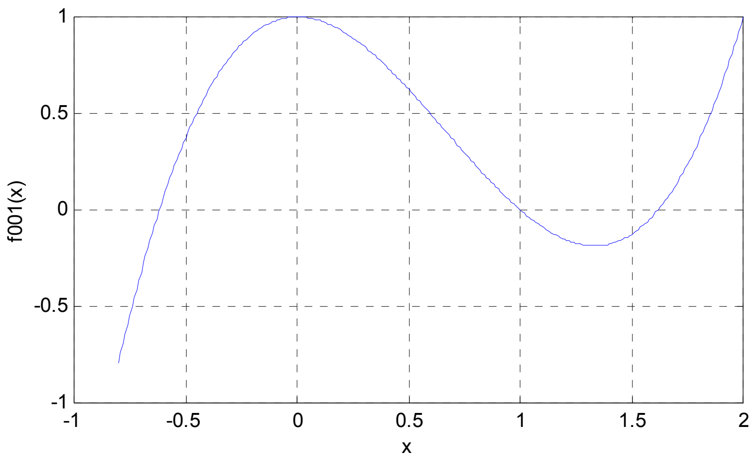
We will use the Newton-Raphson method to solve for the equation, $f(x) = x^3-2x^2+1 = 0$. The following MATLAB commands define the function *f001(x)* and its derivative, *f01p(x)*:

```
»f001 = inline('x.^3-2*x.^2+1','x')

f001 =

     Inline function:
     f001(x) = x.^3-2*x.^2+1
```

```
» f01p = inline('3*x.^2-2','x')

f01p =

     Inline function:
     f01p(x) = 3*x.^2-2
```

To have an idea of the location of the roots of this polynomial we'll plot the function using the following MATLAB commands:

```
» x = [-0.8:0.01:2.0]';y=f001(x);
» plot(x,y);xlabel('x');ylabel('f001(x)');
» grid on
```



We see that the function graph crosses the x-axis somewhere between –1.0 and –0.5, close to 1.0, and between 1.5 and 2.0. To activate the function we could use, for example, an initial value $x_0 = 2$:

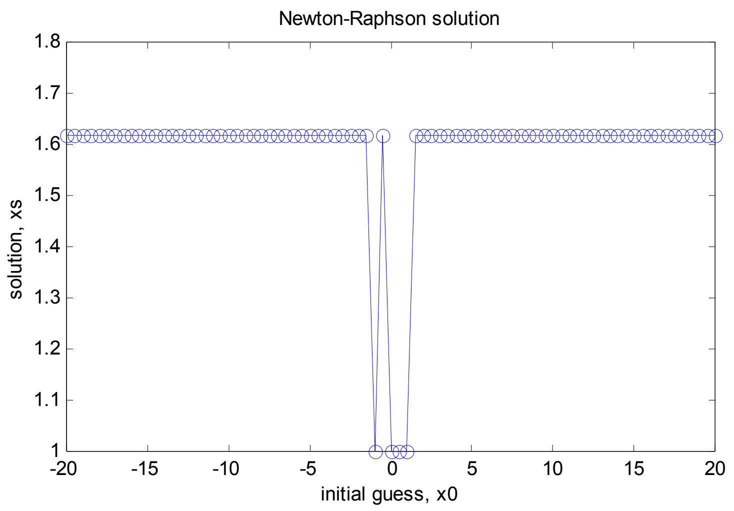
```
» [x,iterations] = newton(2,f001,f01p)

x =  1.6180


iterations = 39
```

The following command are aimed at obtaining vectors of the solutions provided by function *newton.m* for *f001(x)=0* for initial values in the vector *x0* such that $-20 < x_0 < 20$. The solutions found are stored in variable *xs* while the required number of iterations is stored in variable *is*.

```
» x0 = [-20:0.5:20]; xs = []; is = [];
EDU» for i = 1:length(x0)
          [xx,ii] = newton(x0(i),f001,f01p);
          xs = [xs,xx]; is = [is,ii];
     end
```

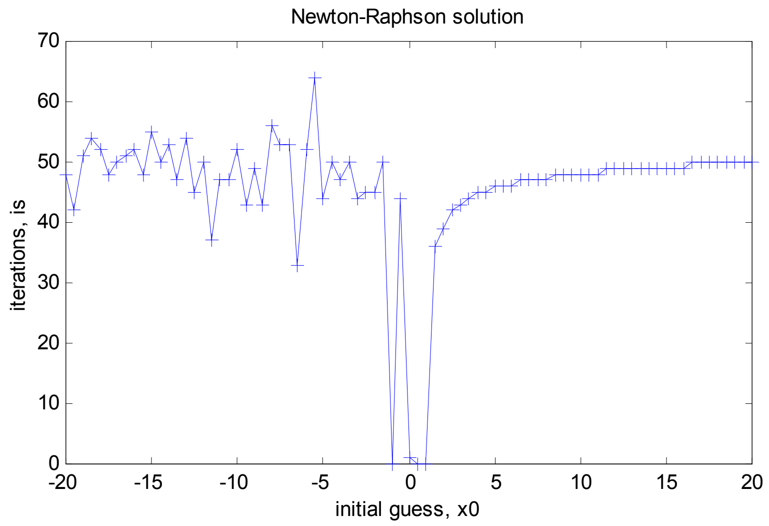
Plot of *xs* vs. *x0*, and of *is* vs. *x0* are shown next:

```
» figure(1);plot(x0,xs,'o');hold;plot(x0,xs,'-');hold;
» title('Newton-Raphson solution');
» xlabel('initial guess, x0');ylabel('solution, xs');
```



This figure shows that for initial guesses in the range $-20 < x_o < 20$, function *newton.m* converges mainly to the solution $x = 1.6180$, with few instances converting to the solutions $x = -1$ and $x = 1$.

```
» figure(2);plot(x0,is,'+'); hold;plot(x0,is,'-');hold;
» title('Newton-Raphson solution');
» xlabel('initial guess, x0');ylabel('iterations, is');
```



This figure shows that the number of iterations required to achieve a solution ranges from 0 to about 65. Most of the time, about 50 iterations are required. The following example shows a case in which the solution actually diverges:

```
» [x,iter] = newton(-20.75,f001,f01p)
iterations = 6
??? Error using ==> newton
Solution diverges
```

NOTE: If the function of interest is defined by an *m*-file, the reference to the function name in the call to *newton.m* should be placed between quotes.

The Secant Method

In the secant method, we replace the derivative $f'(x_i)$ in the Newton-Raphson method with

$$f'(x_i) \approx (f(x_i) - f(x_{i-1}))/(x_i - x_{i-1}).$$

With this replacement, the Newton-Raphson algorithm becomes

$$x_{i+1} = x_i - \frac{f(x_i)}{f(x_i) - f(x_{i-1})} \cdot (x_i - x_{i-1}).$$

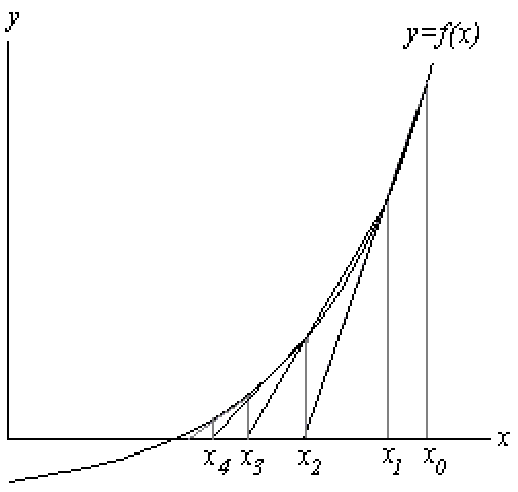
To get the method started we need two values of x, say x_o and x_1, to get the first approximation to the solution, namely,

$$x_2 = x_1 - \frac{f(x_1)}{f(x_1) - f(x_o)} \cdot (x_1 - x_0).$$

As with the Newton-Raphson method, the iteration is stopped when

$$|f(x_{i+1})| < \varepsilon.$$

Figure 4, below, illustrates the way that the secant method approximates the solution of the equation f(x) = 0.



A MATLAB function for the secant method

The function *secant*, listed below, uses the secant method to solve for non-linear equations. It requires two initial values and an expression for the function, f(x).

```
function [x,iter]=secant(x0,x00,f)
% newton-raphson algorithm
N = 100; eps = 1.e-5; %  define max. no. iterations and error
maxval = 10000.0;     %  define value for divergence
xx1 = x0; xx2 = x00;
while N>0
   gp = (f(xx2)-f(xx1))/(xx2-xx1);
   xn = xx1-f(xx1)/gp;
   if abs(f(xn))<eps
      x=xn;
      iter = 100-N;
      return;
   end;
   if abs(f(xn))>maxval
      iter=100-N;
      disp(['iterations = ',num2str(iter)]);
      error('Solution diverges');
      abort;
   end;
   N = N - 1;
   xx1 = xx2;
   xx2 = xn;
end;
iter=100-N;
disp(['iterations = ',iter]);
error('No convergence');
abort;
% end function
```

Application of secant method

We use the same function *f001(x) = 0* presented earlier. The following commands call the function *secant.txt* to obtain a solution to the equation:

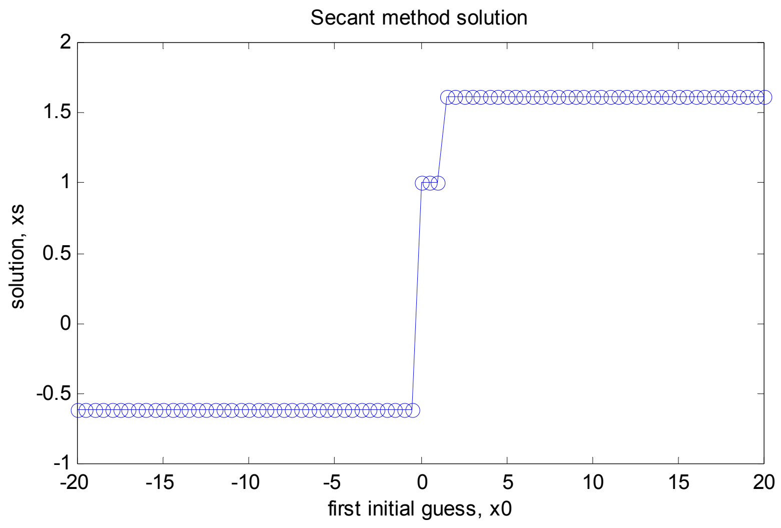
```
» [x,iter] = secant(-10.0,-9.8,f001)

x = -0.6180


iter = 11
```

The following command are aimed at obtaining vectors of the solutions provided by function *Newton.m* for *f001(x)=0* for initial values in the vector *x0* such that $-20 < x_0 < 20$. The solutions found are stored in variable *xs* while the required number of iterations is stored in variable *is*.

```
» x0 = [-20:0.5:20]; x00 = x0 + 0.5; xs = []; is = [];
» for i = 1:length(x0)
         [xx,ii] = secant(x0(i),x00(i),f001);
         xs = [xs, xx]; is = [is, ii];
     end
```

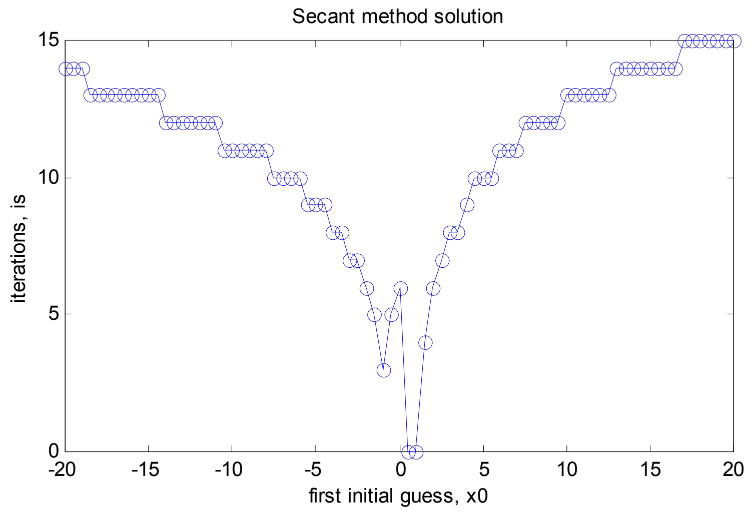
Plot of *xs* vs. *x0*, and of *is* vs. *x0* are shown next:

```
» figure(1);plot(x0,xs,'o');hold;plot(x0,xs,'-');hold;
» title('Secant method solution');
» xlabel('first initial guess, x0');ylabel('solution, xs');
```



This figure shows that for initial guesses in the range *–20* < x_o<*20*, function *newton.m* converges to the three solutions. Notice that initial guesses in the range *–20* < x_0 < *-1*, converge to *x* = *- 0.6180*; those in the range *–1*< *x* < *1*, converge to *x* = *1*; and, those in the range *1*<*x*<*20*, converge to *x* =*1.6180*.

```
» figure(2);plot(x0,is,'o');hold;plot(x0,is,'-');hold;
» xlabel('first initial guess, x0');ylabel('iterations, is');
» title('Secant method solution');
```



This figure shows that the number of iterations required to achieve a solution ranges from 0 to about 15. Notice also that, the closer the initial guess is to zero, the less number of iterations to convergence are required.

NOTE: If the function of interest is defined by an *m*-file, the reference to the function name in the call to *newton.m* should be placed between quotes.

Solving systems of non-linear equations

Consider the solution to a system of *n* non-linear equations in *n* unknowns given by

$$\begin{gathered} f_1(x_1,x_2,\ldots,x_n) = 0 \\ f_2(x_1,x_2,\ldots,x_n) = 0 \\ \cdot \\ \cdot \\ \cdot \\ f_n(x_1,x_2,\ldots,x_n) = 0 \end{gathered}$$

The system can be written in a single expression using vectors, i.e.,

$$\mathbf{f}(\mathbf{x}) = 0,$$

where the vector **x** contains the independent variables, and the vector **f** contains the functions $f_i(\mathbf{x})$:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad \mathbf{f}(\mathbf{x}) = \begin{bmatrix} f_1(x_1,x_2,\ldots,x_n) \\ f_2(x_1,x_2,\ldots,x_n) \\ \vdots \\ f_n(x_1,x_2,\ldots,x_n) \end{bmatrix} = \begin{bmatrix} f_1(\mathbf{x}) \\ f_2(\mathbf{x}) \\ \vdots \\ f_n(\mathbf{x}) \end{bmatrix}.$$

Newton-Raphson method to solve systems of non-linear equations

A Newton-Raphson method for solving the system of linear equations requires the evaluation of a matrix, known as the Jacobian of the system, which is defined as:

$$\mathbf{J} = \frac{\partial(f_1,f_2,\ldots,f_n)}{\partial(x_1,x_2,\ldots,x_n)} = \begin{bmatrix} \partial f_1/\partial x_1 & \partial f_1/\partial x_2 & \cdots & \partial f_1/\partial x_n \\ \partial f_2/\partial x_1 & \partial f_2/\partial x_2 & \cdots & \partial f_2/\partial x_n \\ \vdots & \vdots & \ddots & \vdots \\ \partial f_n/\partial x_1 & \partial f_n/\partial x_2 & \cdots & \partial f_n/\partial x_n \end{bmatrix} = [\frac{\partial f_i}{\partial x_j}]_{n\times n}.$$

If $\mathbf{x} = \mathbf{x}_0$ (a vector) represents the first guess for the solution, successive approximations to the solution are obtained from

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \mathbf{J}^{-1}\cdot\mathbf{f}(\mathbf{x}_n) = \mathbf{x}_n - \Delta\mathbf{x}_n,$$

with $\Delta\mathbf{x}_n = \mathbf{x}_{n+1} - \mathbf{x}_n$.

A convergence criterion for the solution of a system of non-linear equation could be, for example, that the maximum of the absolute values of the functions $f_i(\mathbf{x}_n)$ is smaller than a certain tolerance ε, i.e.,

$$\max_i |f_i(\mathbf{x}_n)| < \varepsilon.$$

Another possibility for convergence is that the magnitude of the vector $\mathbf{f}(\mathbf{x}_n)$ be smaller than the tolerance, i.e.,

$$|\mathbf{f}(\mathbf{x}_n)| < \varepsilon.$$

We can also use as convergence criteria the difference between consecutive values of the solution, i.e.,

$$\max_i |(x_i)_{n+1} - (x_i)_n| < \varepsilon.,$$

or,

$$|\Delta\mathbf{x}_n| = |\mathbf{x}_{n+1} - \mathbf{x}_n| < \varepsilon.$$

The main complication with using Newton-Raphson to solve a system of non-linear equations is having to define all the functions $\partial f_i/\partial x_j$, for $i,j = 1,2, \ldots, n$, included in the Jacobian. As the number of equations and unknowns, n, increases, so does the number of elements in the Jacobian, n^2.

MATLAB function for Newton-Raphson method for a system of non-linear equations

The following MATLAB function, *newtonm,* calculates the solution to a system of n non-linear equations, $\mathbf{f}(\mathbf{x}) = 0$, given the vector of functions $\mathbf{f}$ and the Jacobian $\mathbf{J}$, as well as an initial guess for the solution $\mathbf{x}_0$.

```
function [x,iter] = newtonm(x0,f,J)

% Newton-Raphson method applied to a
% system of linear equations f(x) = 0,
% given the jacobian function J, with
% J = del(f1,f2,...,fn)/del(x1,x2,...,xn)
% x = [x1;x2;...;xn], f = [f1;f2;...;fn]
% x0 is an initial guess of the solution

N = 100;           % define max. number of iterations
epsilon = 1e-10;    % define tolerance
maxval = 10000.0;   % define value for divergence
xx = x0;            % load initial guess

while (N>0)
   JJ = feval(J,xx);
   if abs(det(JJ))<epsilon
      error('newtonm - Jacobian is singular - try new x0');
      abort;
   end;
   xn = xx - inv(JJ)*feval(f,xx);
   if abs(feval(f,xn))<epsilon
      x=xn;
      iter = 100-N;
      return;
   end;
   if abs(feval(f,xx))>maxval
      iter = 100-N;
      disp(['iterations = ',num2str(iter)]);
      error('Solution diverges');
      abort;
   end;
   N = N - 1;
   xx = xn;
end;
error('No convergence after 100 iterations.');
abort;
% end function
```

The functions **f** and the Jacobian **J** need to be defined as separate functions. To illustrate the definition of the functions consider the system of non-linear equations:

$$f_1(x_1,x_2) = x_1^2 + x_2^2 - 50 = 0,$$
$$f_2(x_1,x_2) = x_1 \cdot x_2 - 25 = 0,$$

whose Jacobian is

$$\mathbf{J} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 2x_1 & 2x_2 \\ x_2 & x_1 \end{bmatrix}.$$

We can define the function **f** as the following user-defined MATLAB function *f2*:

```
function [f] = f2(x)
% f2(x) = 0, with x = [x(1);x(2)]
% represents a system of 2 non-linear equations
f1 = x(1)^2 + x(2)^2 - 50;
f2 = x(1)*x(2) -25;
f  = [f1;f2];
% end function
```

The corresponding Jacobian is calculated using the user-defined MATLAB function *jacob2x2*:

```
function [J] = jacob2x2(x)
% Evaluates the Jacobian of a 2x2
% system of non-linear equations
J(1,1) = 2*x(1); J(1,2) = 2*x(2);
J(2,1) = x(2);   J(2,2) = x(1);
% end function
```

Illustrating the Newton-Raphson algorithm for a system of two non-linear equations

Before using function *newtonm*, we will perform some step-by-step calculations to illustrate the algorithm. We start by defining an initial guess for the solution as:

```
» x0 = [2;1]

x0 =

     2
     1
```

Let's calculate the function **f**(**x**) at **x** = $\mathbf{x}_0$ to see how far we are from a solution:

```
» f2(x0)

ans =

   -45
   -23
```

Obviously, the function **f**($\mathbf{x}_0$) is far away from being zero. Thus, we proceed to calculate a better approximation by calculating the Jacobian **J**($\mathbf{x}_0$):

```
» J0 = jacob2x2(x0)

J0 =

     4     2
     1     2
```

The new approximation to the solution, $\mathbf{x}_1$, is calculated as:

```
» x1 = x0 - inv(J0)*f2(x0)

x1 =

    9.3333
    8.8333
```

Evaluating the functions at $\mathbf{x}_1$ produces:

```
f2(x1)

ans =   115.1389
```

Still far away from convergence. Let's calculate a new approximation, $\mathbf{x}_2$:

```
» x2 = x1-inv(jacob2x2(x1))*f2(x1)

x2 =

    6.0428
    5.7928
```

Evaluating the functions at $\mathbf{x}_2$ indicates that the values of the functions are decreasing:

```
» f2(x2)

ans =

   20.0723
   10.0049
```

A new approximation and the corresponding function evaluations are:

```
» x3 = x2 - inv(jacob2x2(x2))*f2(x2)

x3 =

    5.1337
    5.0087

E» f2(x3)

ans =

    1.4414
    0.7129
```

The functions are getting even smaller suggesting convergence towards a solution.

Solution using function newtonm

Next, we use function *newtonm* to solve the problem postulated earlier.

A call to the function using the values of x0, f2, and jacob2x2 is:

```
» [x,iter] = newtonm(x0,'f2','jacob2x2')

x =

    5.0000
    5.0000


iter =

    16
```

The result shows the number of iterations required for convergence (16) and the solution found as $x_1 = 5.0000$ and $x_2 = 5.000$. Evaluating the functions for those solutions results in:

```
» f2(x)

ans =

  1.0e-010 *
    0.2910
   -0.1455
```

The values of the functions are close enough to zero (error in the order of 10^{-11}).

"Secant" method to solve systems of non-linear equations

In this section we present a method for solving systems of non-linear equations through the Newton-Raphson algorithm, namely, $\mathbf{x}_{n+1} = \mathbf{x}_n - \mathbf{J}^{-1}\cdot\mathbf{f}(\mathbf{x}_n)$, but approximating the Jacobian through finite differences. This approach is a generalization of the secant method for a single non-linear equation. For that reason, we refer to the method applied to a system of non-linear equations as a "secant" method, although the geometric origin of the term not longer applies.

The "secant" method for a system of non-linear equations free us from having to define the n^2 functions necessary to define the Jacobian for a system of n equations. Instead, we approximate the partial derivatives in the Jacobian with

$$\frac{\partial f_i}{\partial x_j} \approx \frac{f_i(x_1, x_2, \cdots, x_j + \Delta x, \cdots, x_n) - f_i(x_1, x_2, \cdots, x_j, \cdots, x_n)}{\Delta x},$$

where Δx is a small increment in the independent variables. Notice that $\partial f_i/\partial x_j$ represents element J_{ij} in the jacobian $\boldsymbol{J} = \partial(f_1,f_2,\ldots,f_n)/\partial(x_1,x_2,\ldots,x_n)$.

To calculate the Jacobian we proceed by columns, i.e., column j of the Jacobian will be calculated as shown in the function *jacobFD* (*jacob*ian calculated through *F*inite *D*ifferences) listed below:

```
function [J] = jacobFD(f,x,delx)
% Calculates the Jacobian of the
% system of non-linear equations:
% f(x) = 0, through finite differences.
% The Jacobian is built by columns

[m n] = size(x);

for j = 1:m
   xx = x;
   xx(j) = x(j) + delx;
   J(:,j) = (f(xx)-f(x))/delx;
end;
% end function
```

Notice that for each column (i.e., each value of j) we define a variable *xx* which is first made equal to x, and then the j-th element is incremented by *delx*, before calculating the j-th column of the Jacobian, namely, J(:,j). This is the MATLAB implementation of the finite difference approximation for the Jacobian elements $J_{ij} = \partial f_i/\partial x_j$ as defined earlier.

Illustrating the "secant" algorithm for a system of two non-linear equations

To illustrate the application of the "secant" algorithm we use again the system of two non-linear equations defined earlier through the function *f2*.

We choose an initial guess for the solution as $\mathbf{x}_0 = [2;3]$, and an increment in the independent variables of $\Delta x = 0.1$:

```
x0 = [2;3]

x0 =

     2
     3

EDU» dx = 0.1

dx =  0.1000
```

Variable J0 will store the Jacobian corresponding to $\mathbf{x}_0$ calculated through finite differences with the value of Δx defined above:

```
» J0 = jacobFD('f2',x0,dx)

J0 =

    4.1000    6.1000
    3.0000    2.0000
```

A new estimate for the solution, namely, $\mathbf{x}_1$, is calculated using the Newton-Raphson algorithm:

```
» x1 = x0 - inv(J0)*f2(x0)

x1 =

    6.1485
    6.2772
```

The finite-difference Jacobian corresponding to $\mathbf{x}_1$ gets stored in J1:

```
» J1 = jacobFD('f2',x1,dx)

J1 =

   12.3970   12.6545
    6.2772    6.1485
```

And a new approximation for the solution ($\mathbf{x}_2$) is calculated as:

```
» x2 = x1 - inv(J1)*f2(x1)

x2 =
    4.6671
    5.5784
```

The next two approximations to the solution (**x**$_3$ and **x**$_4$) are calculated without first storing the corresponding finite-difference Jacobians:

```
» x3 = x2 - inv(jacobFD('f2',x2,dx))*f2(x2)

x3 =

    4.7676
    5.2365

EDU» x4 = x3 - inv(jacobFD('f2',x3,dx))*f2(x3)

x4 =

    4.8826
    5.1175
```

To check the value of the functions at **x** = **x**$_4$ we use:

```
» f2(x4)

ans =

    0.0278
   -0.0137
```

The functions are close to zero, but not yet at an acceptable error (i.e., something in the order of 10^{-6}). Therefore, we try one more approximation to the solution, i.e., **x**$_5$:

```
» x5 = x4 - inv(jacobFD('f2',x4,dx))*f2(x4)

x5 =

    4.9413
    5.0587
```

The functions are even closer to zero than before, suggesting a convergence to a solution.

```
» f2(x5)

ans =

    0.0069
   -0.0034
```

MATLAB function for "secant" method to solve systems of non-linear equations

To make the process of achieving a solution automatic, we propose the following MATLAB user -defined function, *secantm*:

```
function [x,iter] = secantm(x0,dx,f)

% Secant-type method applied to a
% system of linear equations f(x) = 0,
% given the jacobian function J, with
% The Jacobian built by columns.
% x = [x1;x2;...;xn], f = [f1;f2;...;fn]
% x0 is the initial guess of the solution
% dx is an increment in x1,x2,... variables


N = 100;              % define max. number of iterations
epsilon = 1.0e-10;    % define tolerance
maxval = 10000.0;     % define value for divergence

if abs(dx)<epsilon
   error('dx = 0, use different values');
   break;
end;

xn  = x0;            % load initial guess

[n m] = size(x0);

while (N>0)
   JJ = [1,2;2,3]; xx = zeros(n,1);
   for j = 1:n                      % Estimating
      xx = xn;                      % Jacobian by
      xx(j) = xn(j) + dx;           % finite
      fxx = feval(f,xx);
      fxn = feval(f,xn);
      JJ(:,j) = (fxx-fxn)/dx; % differences
   end;                             % by columns

   if abs(det(JJ))<epsilon
      error('newtonm - Jacobian is singular - try new x0,dx');
      break;
   end;

   xnp1 = xn - inv(JJ)*fxn;
   fnp1 = feval(f,xnp1);

   if abs(fnp1)<epsilon
      x=xnp1;
      disp(['iterations: ', num2str(100-N)]);
      return;
   end;

   if abs(fnp1)>maxval
      disp(['iterations: ', num2str(100-N)]);
      error('Solution diverges');
      break;
   end;

   N = N - 1;
   xn  = xnp1;
end;
error('No convergence');
break;
% end function
```

Solution using function secantm

To solve the system represented by function *f2*, we now use function *secantm*. The following call to function *secantm* produces a solution after 18 iterations:

```
» [x,iter] = secantm(x0,dx,'f2')
iterations: 18

x =

    5.0000
    5.0000


iter =

    18
```

Solving equations with Matlab function *fzero*

Matlab provides function *fzero* for the solution of single non-linear equations. Use

```
» help fzero
```

to obtain additional information on function *fzero*. Also, read Chapter 7 (*Function Functions*) in the *Using Matlab* guide.

For the solution of systems of non-linear equations Matlab provides function *fsolve* as part of the *Optimization* package. Since this is an add-on package, function *fsolve* is not available in the student version of Matlab. If using the full version of Matlab, check the help facility for function *fsolve* by using:

```
» help fsolve
```

If function *fsolve* is not available in the Matlab installation you are using, you can always use function *secantm* (or *newtonm*) to solve systems of non-linear equations.