# Lecture 13

*Nonlinear systems of equations*

## 1 Introduction

We have investigated the solution of one nonlinear equation in one unknown: $f(x)=0$. What about multiple nonlinear equations in multiple unknowns? To get started, consider one equation in two unknowns

$$f(x,y)=0$$

To be specific, let's take

$$f(x,y)=x^2+y^2-4=0$$

Although we can solve this for *y*

$$y=\pm\sqrt{4-x^2}$$

this does not give us a unique solution $(x,y)$. Rather it defines *y* as a (two valued) function of *x*. In general $f(x,y)=0$ defines a contour in the *x,y* plane (Fig. 1). With two unknowns we need two equations to define a solution. Suppose our second equation is

$$g(x,y)=y-\sin(x)=0$$

This also defines *y* as a function of *x*

$$y=\sin(x)$$

and a contour in the *x,y* plane. An intersection of those contours (Fig. 1) is the solution of the system of equations

$$\begin{aligned} f(x,y)&=0 \\ g(x,y)&=0 \end{aligned} \tag{1}$$

In general one equation in *n* unknowns defines a "surface" of dimension $n-1$. Two equations define a "surface" of dimension $n-2$, and *k* equations define a "surface" of dimension $n-k$. A point is a "surface" of dimension 0, and to define a point we need to have *n* equations in *n* unknowns.

A graphical approach to root finding gets progressively more difficult as the number of dimensions grows. Beyond two dimensions it is rarely an option.

## 2 Notation

For a general system of *n* equations in *n* unknowns it's not convenient to use different letters *x,y,z* for the unknowns or *f,g,h* for the functions. A better notation is to use indices so that the unknown variables are represented as $x_i : 1\le i\le n$. Our system of 2 equations (1) would then be written as

$$\begin{aligned} f_1(x_1,x_2)&=0 \\ f_2(x_1,x_2)&=0 \end{aligned}$$

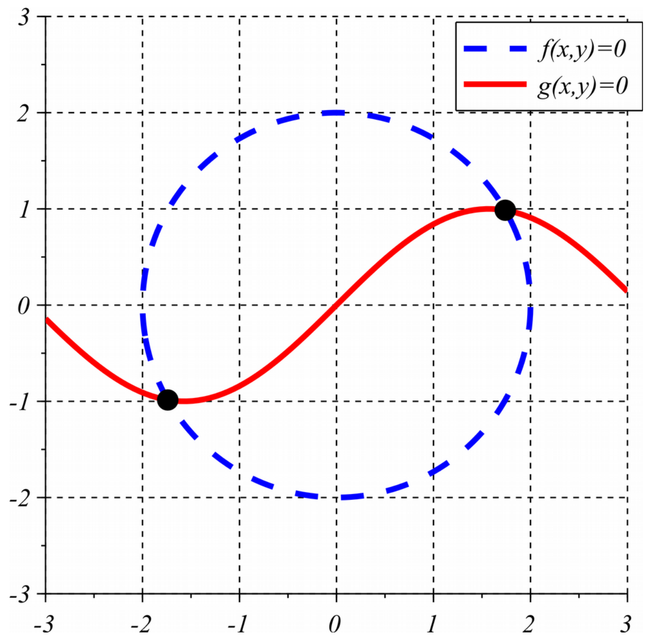

*Fig. 1: Intersections of the f(x,y)=0 and g(x,y)=0 contours define the solutions of the system of two equations in two unknowns.*

A system of *n* equations would have the form

$$\begin{array}{c} f_1(x_1, x_2, \ldots, x_n) = 0 \\ f_2(x_1, x_2, \ldots, x_n) = 0 \\ \vdots \\ f_n(x_1, x_2, \ldots, x_n) = 0 \end{array} \tag{2}$$

Now, we can treat the arrays of unknowns $x_1, x_2, \ldots, x_n$ and functions $f_1, f_2, \ldots, f_n$ as vectors to arrive at the compact notation

$$\mathbf{f}(\mathbf{x}) = \mathbf{0} \tag{3}$$

The notation in (3) is simply a shorthand representation for the system of (2), but when written using vectors it displays the same form as the scalar root-finding problem $f(x)=0$ .

## 3 Challenges

In general solving a system of the form (3) is a difficult problem. To quote the classic work *Numerical Recipes in C* [1,Section 9.6]

> There are *no* good, general methods for solving systems of more than one nonlinear equation. Furthermore, it is not hard to see why (very likely) there *never will be* any good, general methods.

The single largest problem is that in two or more dimensions we loose the concept of bracketing a root. Going back to the notation (1), suppose the point **a** is $(x_a, y_a)$ and the point **b** is $(x_b, y_b)$ and

$$f(x_a, y_a) f(x_b, y_b) < 0$$
$$g(x_a, y_a) g(x_b, y_b) < 0$$

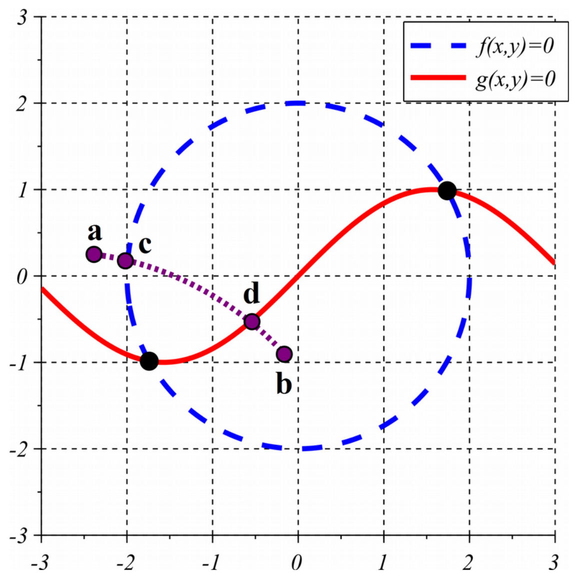

*Fig. 2: Continuous function $f(x, y), g(x, y)$ both change signs between point* **a** *and point* **b**, *but $f(x, y)=0$ and $g(x, y)=0$ are unlikely to occur at the same point.*

Provided $f$ and $g$ are continuous along a path connecting **a** and **b**, on that path there will be a point **c** where $f(x_c, y_c)=0$. Likewise there will be a point **d** where $g(x_d, y_d)=0$. However we have no way of knowing if these two points will coincide, as they must at a solution of (1). In fact, based on Fig. 2 we can see that it is quite unlikely that they will. Thus there is no procedure analogous to bisection that can guarantee we find a root with any given precision.

Without bracketing and bisection methods we are left with the possibility of implementing some form of root polishing. Recall that in one-dimension these methods were not guaranteed to find a solution, even if one or more exists. Typically they need to start reasonably close to a solution to converge.

The one-dimensional root-polishing methods we investigated were: fixed-point iteration, Newton's method and the secant method. We will develop multidimensional versions of those below.

## 4 Fixed-point iteration

Consider the following equations

$$\mathbf{0} = \mathbf{f}(\mathbf{r})$$
$$\mathbf{0} = \mathbf{A}\,\mathbf{f}(\mathbf{r})$$
$$\mathbf{r} = \mathbf{r} + \mathbf{A}\,\mathbf{f}(\mathbf{r}) = \mathbf{g}(\mathbf{r})$$

where **r** is an $n$-dimensional column vector that forms a solution of our problem and **f** is an $n$-dimensional column-vector function. The first equation is simply a statement of our root finding problem. In the second equation we have multiplied both sides by an $n$-by-$n$ matrix **A**. In the last equation we have added **r** to both sides and defined the right-hand side as the function $\mathbf{g}(\mathbf{r})$.

Let's now write

$$\mathbf{x}=\mathbf{g}(\mathbf{x}) \tag{4}$$

This has solution $\mathbf{x}=\mathbf{r}$ . Following the one-dimensional algorithm we expect that the sequence of vectors $\mathbf{x}_k$ where

$$\mathbf{x}_{k+1}=\mathbf{g}(\mathbf{x}_k) \tag{5}$$

might converge to **r** under the right conditions. Note that the notation $\mathbf{x}_k$ refers to the *k*th *vector* in a sequence of vectors and *not* the *k*th component of the vector **x**, which would be written $x_k$ .

In terms of components (4) gives us *n* equations

$$x_i=g_i(x_1, x_2, x_3, \ldots, x_n) \ , \ i=1,2,3,\ldots, n$$

Let's write

$$x_i=r_i+e_i$$

where $e_i=x_i-r_i$ is the error in the *i*th component of **x**. Supposing that all the $e_i$ values are small enough that first-order Taylor series are accurate we write (5) as

$$r_i+e_i=g_i(\mathbf{r})+\frac{\partial g_i}{\partial x_1}e_1+\frac{\partial g_i}{\partial x_2}e_2+\cdots+\frac{\partial g_i}{\partial x_n}e_n \tag{6}$$

or

$$e_i=\sum_{j=1}^{n}\frac{\partial g_i}{\partial x_j}e_j \tag{7}$$

since $r_i=g_i(\mathbf{r})$ . Now, define the *n*-by-*n* matrix **J** to have the elements

$$J_{ij}=\frac{\partial g_i}{\partial x_j}$$

This is called the *Jacobian matrix* of the vector function $\mathbf{g}(\mathbf{x})$ . We also define the *n*-by-1 column vector **e** to have elements $e_i$ . Then (7) can be written

$$\mathbf{e}_{k+1}=\mathbf{J}\,\mathbf{e}_k \tag{8}$$

where, again, $\mathbf{e}_k$ is the *k*th *vector* in the sequence of error vectors, and is not to be confused with $e_k$ which is the $k$th *element* in a particular vector.

Intuitively the sequence (8) will converge provided $\|\mathbf{J}\,\mathbf{e}_k\|\leq\alpha\|\mathbf{e}_k\|$ for some $0\leq\alpha<1$ . That is, the norm of the error vector decreases by a fixed factor at each iteration so that

$$\|\mathbf{e}_k\|\leq\alpha^k\|\mathbf{e}_0\|\rightarrow 0 \ \text{ as } \ k\rightarrow\infty$$

In principle it is always possible to find a matrix **A** so that (8) converges. However it's a difficult problem since **A** has $n^2$ components, and it is rarely practical to do so. Instead, we might try to manipulate our system of equations into the form (4), in various ways until we find one that converges.

*Example* 1: Consider the system

$$f(x,y)=x^2+y^2-4=0$$
$$g(x,y)=y-\sin(x)=0$$

Let's write this in the iterative form

$$x=x+x^2+y^2-4$$
$$y=y+y-\sin(x)$$

From Fig. 1 we see that $x=2, y=1$ is near a root. Starting at this point our iteration gives us

| $k$ | $x$ | $y$ |
|---|---|---|
| 1 | 3 | 1.85888 |
| 2 | 11.455435 | 4.6138744 |
| 3 | 159.97026 | 8.9794083 |

which is clearly not converging. However writing

$$x=\sqrt{4-y^2}$$
$$y=\sin(x)$$

we obtain

| $k$ | $x$ | $y$ |
|---|---|---|
| 1 | 1.7320508 | 0.9870266 |
| 2 | 1.7394765 | 0.9858072 |
| 3 | 1.7401679 | 0.9856909 |
| $\vdots$ | $\vdots$ | $\vdots$ |
| 7 | 1.7402407 | 0.9856786 |

which does converge.

## 5 The Newton-Raphson method

In one-dimension, the idea behind Newton's method is to approximate a function by its tangent line

$$f(x_{k+1})=f(x_k)+f'(x_k)(x_{k+1}-x_k)=0$$

and solve for the root of that line to get

$$x_{k+1}=x_k-\frac{f(x_k)}{f'(x_k)}$$

In the $n$-dimensional case, if we know $f_i(\mathbf{x})$, for a small change to $\mathbf{x}$ we can approximate the change in the function by a first-order Taylor series

$$f_i(x_1+u_1, x_2+u_2, \ldots, x_n+u_n) \approx f_i(\mathbf{x}) + \frac{\partial f_i}{\partial x_1}u_1 + \frac{\partial f_i}{\partial x_2}u_2 + \cdots + \frac{\partial f_i}{\partial x_n}u_n$$

where the derivatives are evaluated at **x** and **u** is the small change in **x**. Setting this expression equal to zero we have

$$\frac{\partial f_i}{\partial x_1}u_1 + \frac{\partial f_i}{\partial x_2}u_2 + \cdots + \frac{\partial f_i}{\partial x_n}u_n = -f_i(\mathbf{x})$$

Doing this for $i=1,2,\ldots,n$ we get the system of equations

$$\begin{aligned}
\frac{\partial f_1}{\partial x_1}u_1 + \frac{\partial f_1}{\partial x_2}u_2 + \cdots + \frac{\partial f_1}{\partial x_n}u_n &= -f_1(\mathbf{x}) \\
\frac{\partial f_2}{\partial x_1}u_1 + \frac{\partial f_2}{\partial x_2}u_2 + \cdots + \frac{\partial f_2}{\partial x_n}u_n &= -f_2(\mathbf{x}) \\
&\vdots \\
\frac{\partial f_n}{\partial x_1}u_1 + \frac{\partial f_n}{\partial x_2}u_2 + \cdots + \frac{\partial f_n}{\partial x_n}u_n &= -f_n(\mathbf{x})
\end{aligned}$$

Defining the *Jacobian matrix* **J** to have elements

$$J_{ij} = \frac{\partial f_i}{\partial x_j}$$

we then have the linear system

$$\mathbf{J}\,\mathbf{u} = -\mathbf{f}$$

Solving this we update **x** and repeat until convergence. As in all root-polishing methods, convergence involves some guesswork. Typically we assume convergence when $\|\mathbf{u}\|$ is smaller than some tolerance. The result is the *Newton-Raphson method*.

*Newton-Raphson method*

*Given* $\mathbf{x}_k$ *calculate*

$\mathbf{f}_k = \mathbf{f}(\mathbf{x}_k)$ *and* $\mathbf{J}_k$ *with elements* $J_{ij} = \dfrac{\partial f_i}{\partial x_j}$ *evaluated at* $\mathbf{x}_k$

*solve for* $\mathbf{u}_k = -\mathbf{J}_k^{-1}\mathbf{f}_k$

*update* $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{u}_k$

*repeat until* $\|\mathbf{u}_k\|$ *is "small enough"*

Let's see how this works on our previous example problem.

*Example* 2: We wish to solve $\mathbf{f}(\mathbf{x}) = \mathbf{0}$ where

$$\mathbf{f}(\mathbf{x}) = \begin{pmatrix} x_1^2 + x_2^2 - 4 \\ x_2 - \sin(x_1) \end{pmatrix}$$

The Jacobian matrix is

$$\mathbf{J}=\begin{pmatrix} 2x_1 & 2x_2 \\ -\cos(x_1) & 1 \end{pmatrix}$$

The iteration

$$\mathbf{x}\leftarrow\mathbf{x}-\mathbf{J}^{-1}\mathbf{f}$$

starting at $x=2, y=1$ produces

```
-->deff('y=f(x)','y=[x(1)^2+x(2)^2-4;x(2)-sin(x(1))]');

-->deff('y=J(x)','y=[2*x(1),2*x(2);-cos(x(1)),1]');

-->x = [2;1];

-->x = x-J(x)\f(x)

 x  =

    1.7415812
    1.0168376

-->x = x-J(x)\f(x)
 x  =

    1.7405501
    0.9856269

-->x = x-J(x)\f(x)
 x  =

    1.7402407
    0.9856787
```

When started “near” a root, the Newton-Raphson method converges quadratically. Like all root-polishing methods there is no guarantee that it will converge, even if a root exists. A Scilab implementation of the Newton-Raphson method is given in the Appendix as `rootNewtonRaphson`.

## 6 Broyden's method

The Newton-Raphson method requires the calculation of the Jacobian matrix, *n* derivatives of *n* functions, at each iteration. It might not be possible to calculate this analytically, or it might not be convenient to do so. In the one-dimensional problem we avoided calculating derivatives by approximating the derivative by

$$f'(x_k)\approx\frac{f(x_{k+1})-f(x_k)}{x_{k+1}-x_k}$$

This led to the secant method. A similar approach in the multidimensional case leads to *Broyden's method*. We approximate the Jacobian **J** by a matrix **B** and otherwise follow the Newton-Raphson method. Assume we start with some $\mathbf{x}_k$, $\mathbf{f}_k=\mathbf{f}(\mathbf{x}_k)$ and some estimate $\mathbf{B}_k$ for the Jacobian. We solve for

$$\mathbf{u}_k = -\mathbf{B}_k^{-1}\mathbf{f}_k$$

update our root estimate

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{u}_k$$

calculate

$$\mathbf{f}_{k+1} = \mathbf{f}(\mathbf{x}_{k+1})$$

and update our Jacobian estimate using Broyden's formula

$$\mathbf{B}_{k+1} = \mathbf{B}_k + \mathbf{f}_{k+1}\frac{\mathbf{u}_k^T}{\|\mathbf{u}_k\|^2}.$$

The motivation for Broyden's formula is that we want to have

$$\mathbf{B}_{k+1}\mathbf{u}_k = \mathbf{f}_{k+1} - \mathbf{f}_k$$

This is analogous to the one-dimensional formula $f'\Delta x = \Delta f$ where the Jacobian estimate $\mathbf{B}$ takes the place of the function derivative. Since

$$\frac{\mathbf{u}_k^T}{\|\mathbf{u}_k\|^2}\mathbf{u}_k = 1$$

we have

$$\mathbf{B}_{k+1}\mathbf{u}_k = \mathbf{B}_k\mathbf{u}_k + \mathbf{f}_{k+1} = \mathbf{f}_{k+1} - \mathbf{f}_k$$

In summary:

*Broyden's method*

*Given* $\mathbf{x}_k$, $\mathbf{f}_k = \mathbf{f}(\mathbf{x}_k)$ *and* $\mathbf{B}_k$

*solve for* $\mathbf{u}_k = -\mathbf{B}_k^{-1}\mathbf{f}_k$

*update* $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{u}_k$

*calculate* $\mathbf{f}_{k+1} = \mathbf{f}(\mathbf{x}_{k+1})$

*update* $\mathbf{B}_{k+1} = \mathbf{B}_k + \mathbf{f}_{k+1}\frac{\mathbf{u}_k^T}{\|\mathbf{u}_k\|^2}.$

*repeat until* $\|\mathbf{u}_k\|$ *is "small enough"*

If we have no information to guide us in forming the initial estimate $\mathbf{B}_0$ we can take it to be the identify matrix. Broyden's update formula will usually cause $\mathbf{B}_k$ to quickly form a good estimate of $\mathbf{J}_k$. A Scilab implement appears in the Appendix as `rootBroyden`.

## The fsolve command (Scilab)

We have already learned how to use the `fsolve` command to find roots of one-dimensional functions. Precisely the same syntax applies in the case of an *n*-dimensional function

```
    r = fsolve(x0,f);
```

The only difference is that `r`, `x0` and `f` are now *n*-dimensional. If the Jacobian can be explicitly calculated that can be added as an additional argument

```
r = fsolve(x0,f,J);
```

and `fsolve` will utilize that for faster convergence.

## 7 References

1. Press, W. H., Flannery, B. P., Teukolsky, S. A. and Vetterling, W. T., *Numerical Recipes in C*, Cambridge, 1988, ISBN: 0-521-35465-X.

## 8 Appendix – Scilab code

### 8.1 Newton-Raphson method

```
//////////////////////////////////////////////////////////////////////
// rootNewtonRaphson.sci
// 2014-06-04, Scott Hudson, for pedagogic purposes
// Implements Newton-Raphson method for finding a root f(x) = 0
// where f and x are n-by-1 vectors.
// Requires two functions: y=f(x) and y=J(x) where J(x) is the n-by-n
// Jacobian of f(x). Search starts at x0. Root is returned as r,
// niter is number of iterations performed. Termination when change
// in x is less than tol or MAX_ITERS exceeded.
//////////////////////////////////////////////////////////////////////
function [r, nIters]=rootNewtonRaphson(x0, f, J, tol)
  MAX_ITERS = 40; //give up after this many iterations
  nIters = 1; //1st iteration
  r = x0-J(x0)\f(x0); //Newton's formula for next root estimate
  while (max(abs(r-x0))>tol) & (nIters<=MAX_ITERS)
    nIters = nIters+1; //keep track of # of iterations
    x0 = r; //current root estimate is last output of formula
    r = x0-J(x0)\f(x0); //Newton's formula for next root estimate
  end
endfunction
```

### 8.2 Broyden's method

```
//////////////////////////////////////////////////////////////////////
// rootBroyden.sci
// 2014-06-12, Scott Hudson, for pedagogic purposes
// Implements Broyden's method for finding a root of f(x)=0
// where f and x are n-by-1 vectors. x0 is initial guess for root
// and tol is termination tolerance for change in x.
//////////////////////////////////////////////////////////////////////
function [r, nIters]=rootBroyden(x0, f, tol)
  MAX_ITERS = 40; //give up after this many iterations
  xk = x0;
  n = length(xk);
  fk = f(xk);
  Bk = eye(n,n);
  uk = -Bk\fk;
  nIters = 0;
  while (max(abs(uk))>tol) & (nIters<=MAX_ITERS)
    xk = xk+uk;
    fk = f(xk);
    Bk = Bk+fk*(uk')/(uk'*uk);
    uk = -Bk\fk;
    nIters = nIters+1;
  end
  r = xk+uk;
endfunction
```

## 9 Appendix – Matlab code

### 9.1 Newton-Raphson method

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% rootNewtonRaphson.m
%% 2014-06-04, Scott Hudson, for pedagogic purposes
%% Implements Newton-Raphson method for finding a root f(x) = 0
%% where f and x are n-by-1 vectors.
%% Requires two functions: y=f(x) and y=J(x) where J(x) is the n-by-n
%% Jacobian of f(x). Search starts at x0. Root is returned as r,
%% niter is number of iterations performed. Termination when change
%% in x is less than tol or MAX_ITERS exceeded.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [r,nIters] = rootNewtonRaphson(x0,f,J,tol)
  MAX_ITERS = 40; %%give up after this many iterations
  nIters = 1; %%1st iteration
  r = x0-J(x0)\f(x0); %%Newton's formula for next root estimate
  while (max(abs(r-x0))>tol) && (nIters<=MAX_ITERS)
    nIters = nIters+1; %%keep track of # of iterations
    x0 = r; %%current root estimate is last output of formula
    r = x0-J(x0)\f(x0); %%Newton's formula for next root estimate
  end
end
```

### 9.2 Broyden's method

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% rootBroyden.m
%% 2014-06-12, Scott Hudson, for pedagogic purposes
%% Implements Broyden's method for finding a root of f(x)=0
%% where f and x are n-by-1 vectors. x0 is initial guess for root
%% and tol is termination tolerance for change in x.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [r,nIters] = rootBroyden(x0,f,tol)
  MAX_ITERS = 40; %%give up after this many iterations
  xk = x0;
  n = length(xk);
  fk = f(xk);
  Bk = eye(n,n);
  uk = -Bk\fk;
  nIters = 0;
  while (max(abs(uk))>tol) && (nIters<=MAX_ITERS)
    xk = xk+uk;
    fk = f(xk);
    Bk = Bk+fk*(uk')/(uk'*uk);
    uk = -Bk\fk;
    nIters = nIters+1;
  end
  r = xk+uk;
end
```