Lecture `root4`: Nonlinear Equations: Systems

November 9, 2022

Summary: Comparison of the various nonlinear solvers discussed to date, and a look at solving systems of nonlinear equations. Some aspects taken from C. F. Van Loan's *Introduction to Scientific Computing*.

References: Some aspects taken from C. F. Van Loan's *Introduction to Scientific Computing*. See also T. Sauer's *Numerical Analysis*, Section 2.7, pages 131–135.

1 Nonlinear Systems

The past few lectures have covered various methods for solving nonlinear scalar equations. However, just as the scalar linear equation $ax = b$ generalizes to $A\mathbf{x} = \mathbf{b}$, nonlinear equations may arise as a system. The scalar form of a general nonlinear equation is

$$f(x) = 0. \tag{1}$$

We shall generalize this by denoting vector versions of x and f as

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix}, \quad \mathbf{f}(\mathbf{x}) = \begin{pmatrix} f_1(\mathbf{x}) \\ f_2(\mathbf{x}) \\ f_3(\mathbf{x}) \\ \vdots \\ f_n(\mathbf{x}) \end{pmatrix},$$

where the x_i are unknowns and the $f_i(\mathbf{x}) = f_i(x_1, x_2, \cdots, x_n)$ are nonlinear functions. A general nonlinear system can be then written compactly as

$$\mathbf{f}(\mathbf{x}) = \mathbf{0} \tag{2}$$

Example 1
Let

$$f_1(x_1, x_2, x_3) = x_1^3 + \sin e^{x_2}, \quad f_2(x_1, x_2, x_3) = \log(x_2 x_3), \quad f_3(x_1, x_2, x_3) = \frac{x_1^2}{x_3 + 4}$$

Then a 3×3 nonlinear system can be written as

$$\begin{pmatrix} f_1(\mathbf{x}) \\ f_2(\mathbf{x}) \\ f_3(\mathbf{x}) \end{pmatrix} = \begin{pmatrix} x_1^3 + \sin e^{x_2} \\ \log(x_2 x_3) \\ x_1^2/(x_3 + 4) \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

Notice that we have chosen the same size for the vectors $\mathbf{x}$ and $\mathbf{f}$. Unlike the situation with linear systems, there is no simple condition telling us whether or not $\mathbf{f}(\mathbf{x}) = \mathbf{b}$ has a unique solution. However, it is often the case that n nonlinear equations can be solved for n unknowns. For such "square" systems the counting is right, and in this lecture we deal exclusively with such systems. In attempting to solve a nonlinear system, are we allowed to use all of the methods and concepts developed for scalar nonlinear equations? To address this question, let's run through the methods we've talked about previously, but now in the context of systems.

1.1 Fixed-Point Iteration

Not much changes for the fixed–point method. We can still define $\tilde{\mathbf{f}}(\mathbf{x})$ as $\mathbf{f}(\mathbf{x}) + \mathbf{x}$, in which case the equation

$$\mathbf{x} = \tilde{\mathbf{f}}(\mathbf{x})$$

is equivalent to equation (2). Of course, this form is amenable to fixed-point iteration: given an initial guess $\mathbf{x}_0$, we can define successive iterates as

$$\mathbf{x}_{i+1} = \tilde{\mathbf{f}}(\mathbf{x}_i). \tag{3}$$

The same observations made for scalar equations hold for systems. In fact, even the Contraction Mapping Theorem is still valid, albeit in a more abstract setting. The scalar case of this theorem involved a condition on $f'(x)$. "$\mathbf{f}'$" for the system case doesn't really make sense as written, but we do have the concept of the *Jacobian matrix*:

$$\frac{\partial \mathbf{f}}{\partial \mathbf{x}} = D\mathbf{f} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_n} \end{pmatrix}, \tag{4}$$

where each entry

$$\frac{\partial f_k}{\partial x_j} = \frac{\partial f_k}{\partial x_j}(x_1, x_2, \cdots, x_n)$$

in the Jacobian matrix is a function of the n–point $\mathbf{x}$. In this sense $D\mathbf{f} = D\mathbf{f}(\mathbf{x})$. The definition (4) is the generalization of the "derivative" to multidimensional cases. It's more complicated: instead of a scalar f' that depends on x, it's a matrix $D\mathbf{f}$ that depends on $\mathbf{x}$. Nevertheless, one can state a condition directly analogous to $|f'(x)| < 1$, and this condition governs the conclusion of the *Contraction Mapping Theorem* for systems. Of course, we saw in the scalar case that the condition $|f'(x)| < 1$ is rather stringent: the function may not satisfy this condition. Inapplicability of the Contraction Mapping Theorem is likewise at issue for nonlinear systems.

1.2 Bisection

Our bisection algorithm for scalar nonlinear equations $f(x) = 0$ was geometrically motivated: if f is continuous and $f(a)f(b) < 0$, then it must pass through zero at some value of x between a and b. Unfortunately, there is no such result in the multidimensional case. No "multidimensional bisection" exists.

1.3 Newton's Method and related algorithms

We now turn to Newton–Raphson iteration.[1] Newton's Method can be formulated for the multidimensional case, and it is essentially a direct generalization of the scalar case: one approximates the function $\mathbf{f}$ by a linear function and uses the root of that linear function as the value of the next iterate. However, due to the introduced complexity, we shall simply state the algorithm without derivation:

$$\mathbf{x}_{i+1} = \mathbf{x}_i - \left(D\mathbf{f}(\mathbf{x}_i)\right)^{-1} \mathbf{f}(\mathbf{x}_i). \tag{5}$$

One can easily see the generalization from the scalar case, if we write the scalar Newton's method as

$$x_{i+1} = x_i - \left(f'(x_i)\right)^{-1} f(x_i). \tag{6}$$

We have simply replaced $(f'(x_i))^{-1}$ by the matrix $(D\mathbf{f}(\mathbf{x}_i))^{-1}$, and we now have a matrix-vector multiply between this matrix and the vector $\mathbf{f}(\mathbf{x}_i)$. While the generalization is straightforward, we note the following. First, we must assume that $D\mathbf{f}(\mathbf{x}_i)$ is an invertible matrix. This condition really is no different than requiring that $f'(x_i) \neq 0$ in the scalar case (6), but there are "more ways" for a matrix, here $D\mathbf{f}$, to be singular, which can lead to problems with the iteration in Newton's method for systems. Second, we must actually *compute* the matrix inverse of $D\mathbf{f}(\mathbf{x}_i)$ at *every* iteration! This is a very expensive proposition, especially if the system is large. Do note however, that the full inverse $(D\mathbf{f}(\mathbf{x}_i))^{-1}$ is not really needed, only the vector direction $\Delta\mathbf{x}_i = (D\mathbf{f}(\mathbf{x}_i))^{-1} \mathbf{f}(\mathbf{x}_i)$. This direction is the solution to the linear problem

$$D\mathbf{f}(\mathbf{x}_i) \cdot \Delta\mathbf{x}_i = \mathbf{f}(\mathbf{x}_i). \tag{7}$$

Third, even if we are willing and able to calculate matrix inverses repeatedly (or solve the related linear problems), we must calculate the Jacobian matrix $D\mathbf{f}$. Doing this by hand for a large system is intractable. For the example given above, the Jacobian matrix, obtained via straightforward calculation, is the following:

$$D\mathbf{f}(\mathbf{x}) = \begin{pmatrix} 3x_1^2 & e^{x_2}\cos e^{x_2} & 0 \\ 0 & x_2^{-1} & x_3^{-1} \\ 2x_1/(x_3+4) & 0 & -x_1^2/(x_3+4)^2 \end{pmatrix}. \tag{8}$$

While it is possible to compute the above Jacobian by hand, it's easy to imagine a large system for which such a hand calculation is just not practical.

This is the familiar problem of not being able to calculate the derivative, already seen in the last lecture for Newton's method applied to scalar nonlinear equations. We got around

[1] Joseph Raphson, an English mathematician and contemporary of Newton, also invented the method.

this problem with the Secant Method or by using finite–differences. Broyden's Method is a a multidimensional analog of the Secant Method, but we will not discuss it here (see Sauer's textbook, pages 135–137 for a description of the method). Instead, we shall content ourselves with presenting the finite–difference modification of Newton's Method.

Perhaps we cannot calculate the entries $(D\mathbf{f})_{jk}$ of the matrix $D\mathbf{f}$ directly. We shall then seek a finite–difference approximation g_{jk} to each entry $\partial f_k/\partial x_l$. To do this, we'll define a vector $\mathbf{x}+\delta\mathbf{e}_k$ which is a perturbation of the $\mathbf{x}$ vector:

$$\mathbf{x}+\delta\mathbf{e}_k = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_{k-1} \\ x_k \\ x_{k+1} \\ \vdots \\ x_n \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ \delta \\ 0 \\ \vdots \\ 0 \end{pmatrix} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_{k-1} \\ x_k+\delta \\ x_{k+1} \\ \vdots \\ x_n \end{pmatrix}, \qquad k = 1, 2, \ldots, n.$$

That is, $\mathbf{x}+\delta\mathbf{e}_k$ is the same as $\mathbf{x}$, except that the kth component has been perturbed by some small number δ. We then define our approximation g_{jk} to the partial derivatives as

$$\frac{\partial f_j}{\partial x_k}(\mathbf{x}_\mu) \approx g_{jk} = \frac{f_j(\mathbf{x}_\mu+\delta\mathbf{e}_k) - f_j(\mathbf{x}_\mu)}{\delta}, \qquad j,k = 1,2,\ldots,n,$$

where we now use μ as the iteration index in order to avoid confusion with k. Of course we again run into the problem of choosing a value for δ: it needs to be small, but not too small; say perhaps around 10^{-3}. Nevertheless, if we use this approximation for the Jacobian, then we can define a matrix G with entries given by $(G)_{jk} = g_{jk}$, and we have $G \approx D\mathbf{f}(\mathbf{x}_\mu)$. Then our approximate Newton's method reads

$$\mathbf{x}_{\mu+1} = \mathbf{x}_\mu - (G)^{-1}\,\mathbf{f}(\mathbf{x}_\mu). \tag{9}$$

The benefit of using this approximate method is that it does not require us to know anything about the Jacobian directly; however, computing the g_{jk} may require many function evaluations, which can be expensive. Nevertheless, (9) is one of the most widely–used methods for solving nonlinear systems.

2 Summary of nonlinear solvers

We end this lecture by briefly listing the advantages and disadvantages of the methods presented so far.

Fixed-point iteration:

- ⊕ Simple implementation
- ⊕ Contraction Mapping Theorem provides guaranteed convergence under certain conditions
- ⊕ Applicable to systems

⊖ The required assumptions to guarantee convergence are too strict. If these assumptions aren't satisfied, the method frequently diverges.

Bisection:

⊕ Easily understandable geometry and motivation

⊕ Guaranteed convergence if very simple conditions are met

⊖ Slow convergence

⊖ No generalization to nonlinear systems.

Newton-Raphson:

⊕ Derivation of the method mathematically founded

⊕ Fast convergence to roots

⊕ Can be generalized to systems

⊖ Requires explicit computation of derivatives/Jacobians

⊖ Can diverge if initial condition is not close enough to the root.

3 Final example

For this example we use $\mathbf{u} = (u, v)^T$ as the independent variable, rather than $\mathbf{x}$. Nonetheless, the nonlinear system $\mathbf{f}(\mathbf{u}) = \mathbf{0}$ is defined by

$$\mathbf{f}(\mathbf{u}) = \begin{pmatrix} -u^3 + v \\ u^2 + v^2 - 1 \end{pmatrix}.$$

Let us try to solve this system via both fixed-point and Newton iteration. The equations we are trying to simultaneously solve are

$$-u^3 + v = 0, \qquad u^2 + v^2 - 1 = 0. \tag{10}$$

Clearly, one way to write these equations is as follows:

$$v = u^3, \qquad u = \sqrt{1 - v^2},$$

corresponding to the fixed point scheme

$$v_{n+1} = u_n^3, \qquad u_{n+1} = \sqrt{1 - v_n^2} \qquad \text{(method 1)}.$$

We shall see that this scheme does not converge. To get another scheme, write the first equation in (10) as $u = v^{1/3}$ or $\frac{1}{2}u = \frac{1}{2}v^{1/3}$ or $u = \frac{1}{2}u + \frac{1}{2}v^{1/3}$. This yields a second scheme

$$u_{n+1} = \tfrac{1}{2}u_n + \tfrac{1}{2}v_n^{1/3}, \qquad v_{n+1} = \sqrt{1 - u_n^2} \qquad \text{(method 2)}.$$

Although not obvious, this fixed-point scheme does converge. We consider both fixed point schemes with the following MATLAB code.

```
% UNM CS/Math 375, Fall 2022
% Fixed Point Method for
%
%        -u^3     + v       = 0
%         u^2     + v^2 - 1 = 0
%
%  Comment/uncomment method 1 or 2 below as desired.
%  Method (1)
%           v = u^3
%           u = sqrt(1-v^2)
%
%  Method (2)
%           u = 0.5*u + 0.5*v^(1/3)
%           v = sqrt(1-u^2)
%
% function [u, v, err, k] = FixedPointExample(tol,u0,v0)
  function [u, v, err, k] = FixedPointExample(tol,u0,v0)
  err = 1000;
  maxk = 1000; k = 0;
  u = u0; v = v0;                     % Initial guess.
  while((err > tol) && (k < maxk))
    uold = u; vold = v;               % Save previous iterate.
%   v = uold^3;                       % METHOD (1)
%   u = sqrt(1-vold^2);
    u = 0.5*uold + 0.5*(vold)^(1/3);  % METHOD (2)
    v = sqrt(1 - uold^2);
    err = sqrt((u-uold)^2+(v-vold)^2);
    k = k+1;
  end
```

Here is the output using fixed-point Method 1.

```
>> format compact; format long
>> tol = 1e-8; u0=1; v0=1;
>> [u v err k] = FixedPointExample(tol,u0,v0)
u =  1
v =  1
err =  1
k =  1000
```

The algorithm ran until the maximum number 1000 of allowable iterations without convergence. Here is the output using fixed-point Method 2.

```
>> format compact; format long
>> tol = 1e-8; u0=1; v0=1;
>> [u v err k] = FixedPointExample(tol,u0,v0)
u =  0.826031356216796
v =  0.563624171881370
err =    5.57692000269224e-09
k =  38
```

Now we consider Newton's method. The Jacobian matrix is

$$D\mathbf{f}(\mathbf{u}) = \begin{pmatrix} -3u^2 & 1 \\ 2u & 2v \end{pmatrix}.$$

The following MATLAB code implements Newton iteration for this example.

```
% UNM CS/Math 375, Fall 2022
% Newton Method for solving the example problem
%
% -u^3 + v       = 0
%  u^2 + v^2 - 1 = 0
%
% function [u, v, err, k] = NewtonExample(tol,u0,v0)
  function [u, v, err, k] = NewtonExample(tol,u0,v0)
  err = 1000;
  maxk = 1000; k = 0;
  u = u0; v = v0;                     % Initial guess.
  while((err > tol) && (k < maxk))
    uold = u; vold = v;         % Save previous iterate.
    F = [-u^3 + v;              % Nonlinear function.
          u^2 + v^2  - 1];
    DF = [-3*u^2 , 1;           % Jacobian.
          2*u    , 2*v];
    Z = DF\F;                   % Compute Newton direction.
    u = u - Z(1);               % Get new iterate.
    v = v - Z(2);
    err = sqrt((u-uold)^2+(v-vold)^2);
    k = k+1;
  end
```

Here is the output using Newton's method.

```
>> format compact; format long
>> tol = 1e-8; u0=1; v0=1;
>> [u v err k] = NewtonExample(tol,u0,v0)
u =  0.826031357654187
v =  0.563624162161259
err =    8.36459786245049e-11
k =  5
```

Note that with Newton's method we have converged to the same root much quicker.