Advanced Numerical Techniques for the Solution of Single Nonlinear Algebraic Equations and Systems of Nonlinear Algebraic Equations

David Keffer
Department of Chemical Engineering
University of Tennessee, Knoxville
June 1999

Table of Contents

Introduction

As chemical engineers, we are frequently asked to solve nonlinear algebraic equations, either individually or in systems. To this end, we learned as undergraduates a few useful techniques, such as the Method of Successive Substitution, the Bisection method, the Secant Method, and the Newton-Raphson method for solving a single non-linear (or linear, of course) algebraic equation. For systems of nonlinear algebraic equations, we were probably taught the multivariate variations of the Method of Successive Substitution and Newton-Raphson method. Each of these techniques has some shortcomings and some strengths. With these tools, we ought to be able to solve any system of nonlinear algebraic equations. Why do we need to learn anything more about this topic?

There are two reasons to advance our understanding of the numerical solution of systems of non-linear algebraic equations. The reasons are (1) none of the methods listed above give a good, stable, and easily coded and easily generalized algorithm for solving systems of nonlinear equations and (2) none of the methods listed above will find a minimum if the root does not exist. The first reason is one of practicality; sure we can solve a system of nonlinear algebraic equations using the multivariate Newton Raphson Method, but, for a system of n equations, we have to first analytically evaluate the functional form of n^2 partial derivatives. This drawback makes the multivariate Newton-Raphson method basically useless for large problems. The second reason is one of optimization. Frequently, there isn't a root, only an extremum. The techniques will not find a root of the derivative, only of the function.

In Section I, we begin our discussion with some single equation techniques, which we will then expand to multivariate problems in Section II.

I. Single Non-linear Algebraic Equation, Advanced Numerical Techniques

A. Newton Raphson Method with Numerical Derivatives

A full review of the elementary techniques, such as the Newton Raphson method are available in the course notes for ChE 301. Check those notes out if you need a review of the method. Here, we are going to dive right in.

We have this general problem:

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

The basis of the Newton-Raphson method lies in the fact that we can approximate the derivative of $f(x)$ numerically.

$$f'(x_0) = \frac{df(x_0)}{dx} \approx \frac{f(x_0) - f(x_1)}{x_0 - x_1} \tag{I.2}$$

Now, initially, we don't know the roots of $f(x)$. Let's say our initial guess is x_0. Let's say that we want x_1 to be a solution to $f(x) = 0$. Let's rearrange equation (2) to solve for x_1.

$$x_1 = x_0 - \frac{f(x_0) - f(x_1)}{f'(x_0)} \tag{I.3}$$

Now, if x_1 is a solution to $f(x) = 0$, then $f(x_1) = 0$ and equation (3) becomes:

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} \tag{I.4}$$

This is the Newton-Raphson Method. Based on the x_0, $f(x_0)$, $f'(x_0)$, we then estimate the root to be at x_1. Of course, this is just an estimate. The root will not actually be at x_1. Therefore, we can do the Newton-Raphson Method again, a second iteration. In fact, we repeat these iteration using the formula

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \tag{I.5}$$

until the difference between x_{i+1} and x_i is small enough to satisfy us. At this point, we say that the Newton-Raphson method has converged to a solution. To rigorously state this, we define the absolute error at the ith iteration, e_i, as:

$$e_{a_i} = x_{i+1} - x_i \tag{I.6}$$

and the relative error as

$$e_{r_i} = \frac{x_{i+1} - x_i}{x_{i+1}} \tag{I.7}$$

We say the problem has converged when whichever error that is deemed important is within some acceptable tolerance, for example

$$\text{abs}\left(e_{r_i}\right) \leq \text{tolerance} \tag{I.8}$$

The Newton-Raphson method requires you to calculate the first derivative of the equation, $f'(x)$. That stinks. We don't want to calculate the derivative. What we want is a program where we can input the function and find the root. If we have to calculate the derivative, then we aren't going to use the Newton-Raphson method.

What do we do? We evaluate the derivative numerically. Any decent numerical method text book will have a table of formulae for calculating numerical derivatives. There are many formula, each formed by truncating the Taylor Series Expansion at a different order. There are formulae available for each derivative, the first, second, third, etc. as well, again based on the Taylor Series Expansion. Finally for each formula, there are three 3 choices: forward, backward, and centered finite-divided-difference formulae. Forward finite-divided-difference formulae evaluate the derivative, $f'(x_i)$, based on the function values, $f(x_{i+1})$ and $f(x_i)$. Backward finite-divided-difference formulae evaluate the derivative, $f'(x_i)$, based on the function values, $f(x_{i-1})$ and $f(x_i)$. Centered finite-divided-difference formulae evaluate the derivative, $f'(x_i)$, based on the function values, $f(x_{i+1})$, $f(x_i)$, and $f(x_{i-1})$. Consequently, centered finite-divided-difference formulae are the most accurate.

The centered finite-divided-difference formulae for the first derivative with error of order h^2 and error of order h^4 are respectively:

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_{i-1})}{(x_{i+1} - x_{i-1})} = \frac{f(x_{i+1}) - f(x_{i-1})}{2h} \tag{I.9}$$

$$f'(x_i) = \frac{-f(x_{i+2}) + 8f(x_{i+1}) - 8f(x_{i-1}) + f(x_{i-2})}{12h} \tag{I.10}$$

where h is the distance between points.

The centered finite-divided-difference formulae with error of order h^2 and error of order h^4 are respectively:

$$f''(x_i) = \frac{f(x_{i+1}) - 2f(x_i) + f(x_{i-1})}{h^2} \tag{I.11}$$

$$f'(x_i) = \frac{-f(x_{i+2}) + 16f(x_{i+1}) - 30f(x_i) + 16f(x_{i-1}) - f(x_{i-2})}{12h^2} \tag{I.12}$$

See a numerical methods reference for the forward and backward formulae and for the higher-order derivatives.

We can use these equations to estimate the derivative required in the Newton-Raphson method. That way, we can write a computer code that only requires us to input the function, in order to get the root.

The only question is how do you pick h, the spacing between datapoints? Well one very simple prescription that works most of the time is

$$h = \min(0.01x_i, 0.01) \tag{I.13}$$

Thus if x is a large number, h = 0.01. If x is a small number, h < 0.01. This seems to work well for me. Of course, pathological cases can be imagined which would foil this scheme. I don't know if this scheme has a name. I just call it the Newton-Raphson method with numerical derivatives.

Here is an example of the Newton-Raphson method with numerical derivatives using the centered finite-divided-difference formulae for the first derivative with error of order h^4, coded for MATLAB.

This method retains the strengths and weaknesses of the Newton-Raphson method but eliminates the need to analytically determine the functional form of the derivatives.

One disadvantage of the numerical derivative scheme is that it now increases the number of function evaluations at each iteration from 2 (the function and the analytical derivative) to 5 (the function five times). It is a small price to pay if the alternative is not solving the problem.

```
function x = NRND1(x0,tol,iprint)
%
%  function x = NRND1(x0,tol,iprint)
%  This function performs single variable Newton-Raphson
%  with numerical approximations to the derivative
%
%  x0 is the initial guess
%  tol is the absolute tolerance on the function
%  iprint = 1 if you want iteration information
%
%  the function must be located in fnctn.m
%
%  author:  David Keffer, University of Tennessee, Knoxville
%  date:  May, 1999
%

%
%  maximum iterations and size of h for derivative calc.
maxit = 1000;
dxcon = 0.01;

%
%  initialization
%
x = x0;
err = 100.0;
iter = 0;
%
% convergence loop
%
while ( err > tol )
   dx = min(dxcon*x,dxcon);
   i12dx = 1.0/(12.0*dx);
   %
   %  evaluate function
   %
   f = fnctn(x);
   %
   %  evaluate numerical derivative
   %
   xp(1) = x - 2*dx;
   xp(2) = x - dx;
   xp(3) = x + dx;
   xp(4) = x + 2*dx;
   for j = 1:1:4
      fp(j) = fnctn(xp(j));
   end
   df = i12dx*( -fp(4) + 8*fp(3) - 8*fp(2) + fp(1) );
   %
   %  evaluate new estimate of solution
   %
   xold = x;
   x = xold - f/df;
   %
   %  check for convergence
   %
   iter = iter +1;
   err = abs(x - xold);
   if (iprint == 1)
      fprintf (1,'iter = %4i, err = %9.2e, x = %9.2e, f =
%9.2e, df = %9.2e \n ', iter, err, x, f, df);
   end
   if ( iter > maxit)
      f
      df
      x
      error ('maximum number of iterations exceeded');
   end
end
```

B. Brent's Line Minimization Method

Brent's Line Minimization Method is described in Numerical Recipes by Press, Flannery, Teukolsky, and Vetterling (Cambridge, 1986).

Press et al. has a very nice of describing their mathematical problems and corresponding numerical techniques. Their presentation is short and sweet and very understandable. You ought to read the introduction of the Chapter on Minimization and Maximization of functions. Then you ought to read the Section in that chapter, on Parabolic Interpolation and Brent's Method in One-Dimension. They give a version of the code in FORTRAN.

I am only going to add a few points that I think they have missed.

What is the difference between finding a root and optimizing a function?

Finding a root means locating the x that satisifies

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

Optimizing a function means locating the x that satisfies

$$f'(x) = 0 \tag{I.14}$$

Clearly, the same work is involved. We can see this by making $g(x) = f'(x)$ and then, instead of being asked to optimize $f(x)$, we are asked to find the root of $g(x)$. It takes the same numerical work. However, if we are asked to perform an optimization, and we wish to use the rootfinding programs that we already have, we have analytically evaluate the derivative. Oh boy! We don't like to do that. In fact, we don't like to do it so much that, most of the time, if we have to do it, we will opt to use a different technique. Such is life. So, what happens? Well, people have come up with different algorithms to optimize functions.

Before we talk about some of the specific optimization routines, let me point out that:

> *Any optimization routine can be used as a root-finding routine by asking it to find the minimum of the absolute value of the function. In other finding the root of f(x) is equivalent to optimizing |f(x)|.*

This idea will allow us to use the optimization routines, not only to optimize but also to find the roots of non-linear algebraic equations.

Now we are ready to talk about Line Minimization. Again this discussion is intended to supplement, not replace, the discussion of Press et al.

Brent's Line Minimization Routine does several things

- initially brackets the minimum
- uses a parabolic interpolation to find the minimum

- avoids unnecessary function evaluations
- changes algorithms to maintain efficiency near the minimum
- is robust for cooperative and noncooperative functions

Brent's Line Minimization Routine is an example of a hodge-podge algorithm put together from much experience that WORKS. The routine works and that's what matters most. It is able to work by keeping track of six positions and the function values at those positions

- a, one side of the bracket
- b, the other side of the bracket
- u, most recent point of evaluation
- x, the point with the current least value
- w, the point with the current second-least value
- v, previous value of w

Read the description and the FORTRAN code in Press et al. for further information.

A version of the same Brent routine and the bracketing routine, written in MATLAB, are available on this course's website.

II. Systems of Non-linear Algebraic Equations, Advanced Numerical Techniques

Well, what we have said in the above section for single equations can also be said for systems of nonlinear algebraic equations. Right now, (1) we don't have a good method that doesn't require analytical numerical derivatives, and (2), we don't have a way to optimize, if there is not a root.

In this section, we discuss four methods that allow us to deal with these two problems.

A. Multivariate Newton-Raphson method with numerical derivatives

Consider the system of n nonlinear algebraic equations and n unknowns.

$$\begin{aligned} &f_1(x_1, x_2, x_3, \ldots x_{n-1}, x_n) = 0 \\ &f_2(x_1, x_2, x_3, \ldots x_{n-1}, x_n) = 0 \\ &f_3(x_1, x_2, x_3, \ldots x_{n-1}, x_n) = 0 \\ &\ldots \\ &f_{n-1}(x_1, x_2, x_3, \ldots x_{n-1}, x_n) = 0 \\ &f_n(x_1, x_2, x_3, \ldots x_{n-1}, x_n) = 0 \end{aligned} \tag{II.1}$$

This is the most general form of the problems that face chemical engineers and encompass linear equations (when all the functions, f, are linear) and single equations (when n = 1).

One technique used to solve this problem is called the multivariate Newton-Raphson Method (MNRM). The idea follows from the single-variable case. The basic idea stems from the fact that the total derivative of a function, f_j, is

$$df_j = \left(\frac{\partial f_j}{\partial x_1}\right) dx_1 + \left(\frac{\partial f_j}{\partial x_2}\right) dx_2 \tag{II.2}$$

for the case of two variables or

$$df_j = \sum_{i=1}^{n} \left(\frac{\partial f_j}{\partial x_i}\right) dx_i \tag{II.3}$$

for the n variable case. We can discretize this expression and write:

$$f_j\left(\{x\}^{(2)}\right) - f_j\left(\{x\}^{(1)}\right) = \sum_{i=1}^{n}\left(\frac{\partial f_j}{\partial x_i}\right)\left(x_i^{(2)} - x_i^{(1)}\right) \tag{II.4}$$

where the j is the index over functions, the i is the index over variables and the superscript in parentheses stands for the iteration. Note that if we have only one variable (n=1), this reduces to the Newton-Raphson method that we have already learned.

Now consider that we have n equations so that j = 1 to n. We have a system of equations which we can write in matrix notation as:

$$\underline{f}\left(\{x\}^{(2)}\right) - \underline{f}\left(\{x\}^{(1)}\right) = \sum_{i=1}^{n}\left(\frac{\partial \underline{f}}{\partial x_i}\right)\left(x_i^{(2)} - x_i^{(1)}\right) \tag{II.5}$$

Just as in the single variable case, we want our next iteration to take us to the root so we assume that $\underline{f}\left(\{x\}^{(2)}\right) = 0$. We can then write this system in matrix notation as:

$$\underline{\underline{J}}^{(k)}\underline{\delta x}^{(k)} = -\underline{R}^{(k)} \tag{II.6}$$

where $\underline{R}^{(k)}$ is called the residual vector at the k^{th} iteration and is defined as

$$\underline{R}^{(k)} = \underline{f}\left(\{x\}^{(k)}\right) \tag{II.7}$$

where $\underline{\underline{J}}^{(k)}$ is called the Jacobian matrix at the k^{th} iterationand is defined as

$$\left(\underline{\underline{J}}^{(k)}\right)_{j,i} = \left(\frac{\partial f_j^{(k)}}{\partial x_i^{(k)}}\right) \tag{II.8}$$

and where

$$\underline{\delta x}^{(k)} = \underline{x}^{(k+1)} - \underline{x}^{(k)} \tag{II.9}$$

so that the new guess for the $\underline{x}$ is

$$\underline{x}^{(k+1)} = \underline{x}^{(k)} + \underline{\delta x}^{(k)} \tag{II.10}$$

The algorithm for solving the multivariate Newton-Raphson follows analogously from the single variable NRM. The steps are as follows:

1. Make an initial guess for $\underline{X}$
2. calculate the Jacobian and the Residual.
3. Solve equation II.6
4. Calculate new $\underline{X}$ from equation II.10
5. If the solution has not converged, loop back to step 2.

The multivariate Newton-Raphson Method suffers from the same short-comings as the single-variable Newton-Raphson Method.

(1) You need a good initial guess.
(2) You don't get quadratic convergence until you are close to the solution.
(3) If the partial derivatives are zero, the method blows up. If the partial derivatives are close to zero, the method may not converge.

Additionally, we do not want to analytically determine the functional form of n^2 partial derivatives. So we need to modify the multivariate Newton-Raphson method by adding numerical partial derivatives. Well, the formulae for partial derivatives become more complicated when you have mixed second-partials. However, all we need here is the first partial. Thus we use the same formula (equation I.10) that we used for the single equation case. The difference is now we evaluate n^2 partial derivatives, the partial of each of the n functions with respect to each of the n variables. The Jacobian is composed of these partials.

One disadvantage of the numerical derivative scheme is that it now increases the number of function evaluations at each iteration from 2n (each function and each analytical derivative) to $n+4n^2$ (the function $n+4n^2$ times). It is a large price to pay but, if the alternative is not solving the problem at all, often times, especially if the problem is small, we will pay it.

Here is an example of the multivariate Newton-Raphson method with numerical partial derivatives using the centered finite-divided-difference formulae for the first partial derivatives with error of order h^4, coded for MATLAB.

```
function [f,x] = NRNDN(x0,tol,iprint)
%
%  function x = NRNDN(x0,tol,iprint)
%  This function performs multivariate Newton-Raphson
%  with numerical approximations to the partial derivatives
%
%  x0 is the vector of initial guesses
%  tol is the absolute tolerance on each function
%  iprint = 1 if you want iteration information
%
%  the functions must be located in syseqninput.m
%
%  author:  David Keffer, University of Tennessee, Knoxville
%  date:  May, 1999
%

%
%  maximum iterations
maxit = 1000;
n = max(size(x0));
dxcon = zeros(n,1);
dxcon(:) = 0.01;
Residual = zeros(n,1);
dx = zeros(n,1);
Jacobian = zeros(n,n);
xp = zeros(4);
fp = zeros(n,n,4);
%
%  initialization
%
x = x0;
err = 100.0;
iter = 0;
%
% convergence loop
%
while ( err > tol )
   for j = 1:1:n
    dx(j) = min(dxcon(j)*x(j),dxcon(j));
      i12dx(j) = 1.0/(12.0*dx(j));
   end
   %
   %  evaluate function
   %
   Residual = syseqninput(x);
   Residual = Residual';
   %
   %  set up a grid for evaluation of numerical partial
derivatives
   %  evaluate function at grid points
   %
   for j = 1:1:n
      for i = 1:1:n
         xeval(i) = x(i);
      end
      xp(1) = x(j) - 2*dx(j);
    xp(2) = x(j) - dx(j);
    xp(3) = x(j) + dx(j);
      xp(4) = x(j) + 2*dx(j);
      for k = 1:1:4
         xeval(j) = xp(k);
           fp(:,j,k) = syseqninput(xeval);
      end
   end
   %
   %  calculate numerical derivative
   %
   for i = 1:1:n
      for j = 1:1:n
         Jacobian(i,j) = i12dx(j)*( -fp(i,j,4) + 8*fp(i,j,3) -
8*fp(i,j,2) + fp(i,j,1) );
      end
   end
   %
   %  evaluate new estimate of solution
   %
   xold = x;
   invJ = inv(Jacobian);
   deltax = -invJ*Residual;
   for j = 1:1:n
      x(j) = xold(j) + deltax(j);
   end
   %
   %  check for convergence
   %
   iter = iter +1;
   err = sqrt( sum(deltax.^2) /n  ) ;
   if (iprint == 1)
      fprintf (1,'iter = %4i, err = %9.2e \n ', iter, err);
      if (n == 6)
         fprintf (1, 'x = %9.2e %9.2e %9.2e %9.2e %9.2e %9.2e
\n ', x);
         fprintf (1, 'f = %9.2e %9.2e %9.2e %9.2e %9.2e %9.2e
\n ', Residual);
      end
   end
   if ( iter > maxit)
      Residual
      error ('maximum number of iterations exceeded');
   end
end
f = sqrt(sum(Residual.*Residual)/n);
%
```

B. Nelder and Mead's Downhill Simplex method

Nelder and Mead's Downhill Simple Method is described in Numerical Recipes by Press, Flannery, Teukolsky, and Vetterling (Cambridge, 1986).

Press et al. has a very nice of describing their mathematical problems and corresponding numerical techniques. Their presentation is short and sweet and very understandable. You ought to read the introduction of the Chapter on Minimization and Maximization of functions. Then you ought to read the Section in that chapter, on Downhill Simplex Method in Multidimensions. They give a version of the code in FORTRAN.

I am only going to summarize.

The simplex method is a quick and dirty method. It is frequently extremely computationally-inefficient.

For a discussion of the geometric basis of the simplex method, I direct you to Press et al. Read the description there and the FORTRAN code in Press et al. for further information.

A version of the same Simplex routine, written in MATLAB, is available on this course's website.

C. Powell's Direction Set method

Powell's Direction Set method is described in Numerical Recipes by Press, Flannery, Teukolsky, and Vetterling (Cambridge, 1986). They give a version of the code in FORTRAN.

I am only going to summarize their discussion.

In n-dimensional space, you can define a basis set of n orthogonal directions which span the space. (By spanning the space, I mean that any position in the space is a linear combination of the vectors in the basis set.)

Now, we can imagine that for a given n-dimensional root-finding or optimization problem, there are some ways to obtain a special set of directions, such that performing iterative line minimizations along these (constantly updated) directions leads quickly to the root of extremum.

Powell developed a quadratically convergent method which will obtain a direction set. He then performs line minimizations along these directions, updating the directions as he goes, in order to keep them as conjugate as possible. The directions sets are constantly updated by throwing out the “best” vector from the previous iteration. This allows one new vector to be created that will probably have a large projection on the old vector but, since the old vector has been discarded, will still be linearly independent of the other vectors in the basis set.

A version of the Direction Set routine, written in MATLAB, is available on this course's website.

D. Polak and Ribiere's Conjugate Gradient method

Polak and Ribiere's Conjugate Gradient method is described in Numerical Recipes by Press, Flannery, Teukolsky, and Vetterling (Cambridge, 1986). They give a version of the code in FORTRAN.

I am only going to summarize their discussion.

As was the case in the direction set method, in n-dimensional space, you can define a basis set of n orthogonal directions which span the space. (By spanning the space, I mean that any position in the space is a linear combination of the vectors in the basis set.)

However, we can imagine a special set of directions, which optimize the efficiency of our solution algorithm.

This good special set of directions are called "conjugate gradients". The goal of a set of conjugate gradients is that minimization along one direction should not affect minimization along the other. The goal then is to reduce an n-dimensional minimization to n 1-dimensional minimizations! Wow! That is cool.

In this context, a conjugate set of vectors means a set of vectors, with which the second mixed partials of the objective functions are all zero.

A version of the Conjugate Gradient routine, written in MATLAB, is available on this course's website.