Lecture – 11

Solution of Nonlinear Equations - III

Efficiency of a method

The efficiency index of an iterative method is defined by

$$E = r^{1/n}$$

r: rate of convergence of the method
n: total number of functions and derivative evaluations at each step of iteration.

Obviously, if the value of the index is larger, then the method is more efficient.

Example:

Method	n	r	E
Secant	1	1.62	1.62
Newton	2	2	1.41

Methods for multiple roots

Definition: If we can write f(x) = 0 as $f(x) = (x - x^*)^m g(x) = 0$, where g(x) is bounded and g(x*) = 0, then x* is called a multiple root of multiplicity m.

If x* is a multiple root of multiplicity m of equation f(x) = 0, then we have from definition of multiple root:

$$f(x^*) = f'(x^*) = \cdots\cdots = f^{(m-1)}(x^*) = 0 \text{ and } f^{m}(x^*) \neq 0$$

Note: It can be verified that all iterative methods discussed have only linear rate of convergence when m > 1. For example, for Newton-Raphson method, we get the error equation as

$$\varepsilon_{k+1} = \left(1 - \frac{1}{m}\right)\varepsilon_k + \frac{1}{m^2(m+1)}\varepsilon_k^2 \frac{f^{m+1}(x^*)}{f^{m}(x^*)} + O(\varepsilon_k^3)$$

if $m \neq 1$, $\varepsilon_{k+1} = \left(1 - \frac{1}{m}\right)\varepsilon_k + O(\varepsilon_k^2)$, Which shows the linear convergence.

When the multiplicity of the root is known in advance we can modify the methods by introducing parameters dependent on the multiplicity of the root to increase their order of convergence. For example – Newton-Raphson method

$$x_{k+1} = x_k - \alpha \frac{f_k}{f_k^{'}}$$

a: arbitrary parameter to be determined.

Error equation for this method: $\varepsilon_{k+1} = \left(1 - \frac{\alpha}{m}\right)\varepsilon_k + \frac{\alpha}{m^2(m+1)}\varepsilon_k^2 \frac{f^{m+1}(x^*)}{f^m(x^*)} + O(\varepsilon_k^3)$.

If the method has quadratic rate of convergence, the coefficient of ε_k must vanish, which gives

$$1 - \frac{\alpha}{m} = 0 \Rightarrow \alpha = m$$

Thus the method: $$x_{k+1} = x_k - m\frac{f_k}{f_k^{'}}.$$

If the multiplicity is not known in advance, then use the following procedure:

It is known that f(x) = 0 has a root x* of multiplicity m, then $f'(x) = 0$ has the same root x* and its of multiplicity (m-1). Hence $g(x) = \frac{f(x)}{f'(x)}$ has a simple root x*, we can now use Newton-Raphson method $x_{k+1} = x_k - \frac{g_k}{g_k^{'}}$ to find approximate value of the multiple root x*. Simplification gives $x_{k+1} = x_k - \frac{f_k f_k^{'}}{f_k^{'2} - f_k f_k^{''}}$.

Verify this method has second order convergence.

Zeros of Polynomials

Till now we have discussed methods for finding single zero of an arbitrary function in one dimension. For a special case of a polynomial p(x) of degree n, one often may need to find all "n" of its zeros, which may be complex even if the coefficients of the polynomial are real. There are several approaches available:

1. Use one of the methods such as Newton to find a single root x_1, then consider a deflated polynomial $p(x)/(x - x_1)$ of degree one less. Repeat until all zeros have been found.

Its is a good idea to go back and refine each root using original polynomial p(x) to avoid contamination due to rounding error in forming the deflated polynomial.

2. Form companion matrix of polynomial and compute the eigen values. This is used by MATLAB.

3. Use method designed specifically for finding all roots of polynomial.

System of Non-linear Equations

System of equations tend to be more difficult to solve than single nonlinear equations for a number of reasons:

1. A much wider range of behavior is possible. So that theoretical analysis of the existence and number of solutions is much more complex.

2. No single way, in general, to guarantee convergence to desire solution ot to bracket solution to produce absolutely safe method.

3. Computational overhead increases rapidly with dimension of the problem.

Fixed Point Iteration for System of equations

Fixed point problem for $\boldsymbol{g}:\Re^n \to \Re^n$ is to find a vector x such that $\boldsymbol{x} = \boldsymbol{g(x)}$ corresponding fixed point iteration is simply $\boldsymbol{x}_{k+1} = \boldsymbol{g}(\boldsymbol{x}_k)$ given some starting point $\boldsymbol{x}_0$.

Newton's Method

Many methods for solving nonlinear equations in one-dimension do not generalized directly to n-dimensions. The most popular method that does generalized is Newton's method.

For a differentiable function $\boldsymbol{f}:\Re^n \to \Re^n$

Truncated Taylor series: $\boldsymbol{f}(\boldsymbol{x}+\boldsymbol{s}) \approx \boldsymbol{f}(\boldsymbol{x}) + \boldsymbol{J}_f(\boldsymbol{x})\boldsymbol{s}$, where $\boldsymbol{J}_f(\boldsymbol{x})$ is the Jacobian matrix

$$\{\boldsymbol{J}(\boldsymbol{x})\}_{ij} = \frac{\partial \boldsymbol{f}_i(\boldsymbol{x})}{\partial \boldsymbol{x}_j}$$

If "s" satisfies the linear system $\boldsymbol{J}_f(\boldsymbol{x})\boldsymbol{s} = -\boldsymbol{f(x)}$, then "$\boldsymbol{x}+\boldsymbol{s}$" is taken as an approximate zero of $\boldsymbol{f}$.

In this sense Newton's method replaces a system of nonlinear equations with a system of linear equations, but since the solution of the two systems are not identical in general, the process must be repeated until the approximate solution is reached as accurate as desired.

Algorithm

$\boldsymbol{x}_0$ = Initial guess

for k = 0, 1, 2, 3, ….

Solve $\boldsymbol{J}_f(\boldsymbol{x}_k)\boldsymbol{s}_k = -\boldsymbol{f}(\boldsymbol{x}_k)$ for $\boldsymbol{s}_k$ (Compute Newton Step)

$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \boldsymbol{s}_k$ (Update solution)

end

Cost of Newton's Method

Cost per iteration of Newton's method for dense problem in n dimensions is substantial.

1. Computing Jacobian matrix costs $\boldsymbol{n}^2$ scalar function evaluations.

2. Solving linear system costs $\boldsymbol{O}(\boldsymbol{n}^3)$ operations.

Secant Updating Method

The partial derivatives that make up the Jacobian matrix could be replaced by finite difference approximation along each coordinate direction, but this would entail additional function evaluation purely for the purpose of obtaining derivative information. Instead we take our own cue from the secant method for nonlinear equations in one dimension which avoids explicitly computing derivatives by approximating the derivatives based on the change in function values between successive iterates.

Secant updating methods reduce cost of the Newton's method by

1. Using function values at successive iterates to build approximate Jacobian and avoiding explicit evaluation of derivatives.

2. Updating factorization of approximate Jacobian rather than refactoring it at each iteration.

Note: Most secant updating methods have superlinear but not quadratic convergence rate; often cost less overall than Newton's method.

One of the Simples and most effective secant updating method for solving nonlinear systems is **Broyden's Methods.**

Broyden's Methods

This method begins with an approximate Jacobian matrix and updates it (or a factorization of it) at each iteration,

Algorithm

$\boldsymbol{x}_0$ = Initial guess

$\boldsymbol{B}_0$ = Initial Jacobian approximation (Can be true Jacobian or
Finite difference approximation or
to avoid derivatives we simply start with $\boldsymbol{B}_0 = \mathbf{I}$)

for k = 0, 1, 2, 3, ….

Solve $\boldsymbol{B}_k \boldsymbol{s}_k = -\boldsymbol{f}(\boldsymbol{x}_k)$ for $\boldsymbol{s}_k$ (Compute Newton like Step)

$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \boldsymbol{s}_k$ (Update solution)

$\boldsymbol{y}_k = \boldsymbol{f}(\boldsymbol{x}_{k+1}) - \boldsymbol{f}(\boldsymbol{x}_k)$

$\boldsymbol{B}_{k+1} = \boldsymbol{B}_k + \left((\boldsymbol{y}_k - \boldsymbol{B}_k \boldsymbol{s}_k)\boldsymbol{s}_k^T\right) \Big/ \boldsymbol{s}_k^T \boldsymbol{s}_k$ (Update approximate Jacobian)

end

The motivation for the formula for the update Jacobian approximation $\boldsymbol{B}_{k+1}$ is that it gives the least change to $\boldsymbol{B}_k$ subject to satisfying the secant equation

$$\boldsymbol{B}_{k+1}(\boldsymbol{x}_{k+1} - \boldsymbol{x}_k) = \boldsymbol{f}(\boldsymbol{x}_{k+1}) - \boldsymbol{f}(\boldsymbol{x}_k)$$

In this way the sequence of matrices $\boldsymbol{B}_k$ gains and maintains information about the behavior of the function f along the various directors generated by the algorithm, without the need for the function to be sampled purely for the purpose of obtaining derivative information.

Updating $\boldsymbol{B}_k$ as just indicated would still leave on needing to solve a linear system at each iteration at a cost of $\boldsymbol{O}(\boldsymbol{n}^3)$ arithmetic.

Therefore, in practice a factorization of $\boldsymbol{B}_k$ is updated instead updating $\boldsymbol{B}_k$ directly, so that total cost per iteration is only $\boldsymbol{O}(\boldsymbol{n}^3)$.