

# Solving System of Nonlinear Equations

Yong-Ming Li

Intergraph Corporation

May, 2011

# 1 Introduction

The need to solve a system of nonlinear equations arises in many geometric computations such as intersection, minimum distance, creation of catenary curves (power lines and spider webs are examples of catenary curves), and so on. It is usually difficult to solve a system of nonlinear equations in spite of the fact that many researches have been done in this area. When all equations are  $C^2$  differentiable, Newton's method is preferable due to its fast convergence when sufficiently good estimations of solutions are available. However, Newton's method may diverge when poor estimations are provided.

In computing, for example, curve/curve intersection, we usually subdivide the curves at concerned domain until we can obtain the good estimations. Then, we use Newton's method to refine the solutions. In some cases (e.g., creation of catenary curves), however, good estimations may not be readily derived. Accordingly, we may not be able to obtain solutions via Newton's method. In this document we discuss:

- A generic Newton's method that can be used to solve any system of nonlinear equations.
- The Steepest Descent Algorithm that is used to obtain a good estimation for Newton's method.

# 2 Newton's Method

A system of nonlinear equations has the form

$$\begin{aligned} f_1(x_1, x_2, \dots, x_n) &= 0, \\ f_2(x_1, x_2, \dots, x_n) &= 0, \\ &\vdots \\ f_n(x_1, x_2, \dots, x_n) &= 0, \end{aligned} \tag{2.1}$$

where each function  $f_i$  can be thought as mapping a vector  $\mathbf{X} = (x_1, x_2, \dots, x_n)^T$  of the  $n$ -dimensional space  $\mathbb{R}^n$  into the real line  $\mathbb{R}$ . If vector notation is used to represent the variables  $x_1, x_2, \dots, x_n$ , then system (2.1) can be concisely written as

$$\mathbf{F}(\mathbf{X}) = \mathbf{0},$$

where  $\mathbf{F}$  is a mapping from  $\mathbb{R}^n$  into  $\mathbb{R}^n$  and the functions  $f_1, f_2, \dots, f_n$  are the coordinate functions of  $\mathbf{F}$ .

Defining the matrix  $J(\mathbf{X})$  by

$$J(\mathbf{X}) = \begin{pmatrix} \frac{\partial f_1}{\partial x_1}(\mathbf{X}) & \frac{\partial f_1}{\partial x_2}(\mathbf{X}) \cdots & \frac{\partial f_1}{\partial x_n}(\mathbf{X}) \\ \frac{\partial f_2}{\partial x_1}(\mathbf{X}) & \frac{\partial f_2}{\partial x_2}(\mathbf{X}) \cdots & \frac{\partial f_2}{\partial x_n}(\mathbf{X}) \\ \vdots & & \\ \frac{\partial f_n}{\partial x_1}(\mathbf{X}) & \frac{\partial f_n}{\partial x_2}(\mathbf{X}) \cdots & \frac{\partial f_n}{\partial x_n}(\mathbf{X}) \end{pmatrix},$$

then Newton's method for nonlinear systems is given by

$$\mathbf{X}^{(k)} = \mathbf{X}^{(k-1)} - J(\mathbf{X}^{(k-1)})^{-1} \mathbf{F}(\mathbf{X}^{(k-1)}) \quad (k = 1, 2, \dots)$$

It is noted that  $\mathbf{X}^{(0)}$  is the initial estimation and  $\mathbf{X}^{(k)}$  is the result of  $k^{\text{th}}$  iteration. The matrix  $J(\mathbf{X})$  is known as the *Jacobian matrix* and has a number of applications in analysis. Newton's method is generally expected to give quadratic convergence, provided that a sufficiently accurate starting value is known and the inverse of Jacobian matrix  $J(\mathbf{X})^{-1}$  exists. The weakness in Newton's method arises from the need to compute derivatives and inverse the matrix  $J(\mathbf{X})$  at each iteration step. In practice, explicit computation of  $J(\mathbf{X})^{-1}$  is avoided by performing the operation in a two-step manner. First, a vector  $\mathbf{Y}$  is found that satisfies

$$J(\mathbf{X}^{(k-1)}) \mathbf{Y} = \mathbf{F}(\mathbf{X}^{(k-1)}).$$

Then, the new approximation,  $\mathbf{X}^{(k)}$ , is obtained by subtracting  $\mathbf{Y}$  from  $\mathbf{X}^{(k-1)}$ .

With the initial estimation of  $\mathbf{X}$ , Newton's method is outlined below

Step 1: Memory allocation for  $\mathbf{F}$  ( $n$  doubles) and  $J$  ( $n \times n$  doubles) and initialization.

Step 2: While ( $k < \text{max\_iteration}$ ) do Steps 3 – 7

Step 3: Calculate  $\mathbf{F}$  and  $J$ .

Step 4: If  $\|\mathbf{F}\|_{\infty} < \varepsilon$ , terminate the loop and output the results ( $\varepsilon$  is the given tolerance).

Step 5: Solve  $n \times n$  linear system  $J(\mathbf{X})\mathbf{Y} = \mathbf{F}(\mathbf{X})$ .

Step 6: If  $\|\mathbf{Y}\|_{\infty} < \delta$ , terminate as no changes can be made ( $\delta$  is the machine tolerance).

Step 7: Set  $\mathbf{X} = \mathbf{X} - \mathbf{Y}$  and  $k = k + 1$ .

Step 8: Set error code to *unsuccessful* due to maximum number of iterations exceeded.

Since this new function needs to be accessible by any math function, it will be implemented as a BS-function in our `Ubspm2d` library. In order to make Newton's method generic, we also need to have a callback function to evaluate  $\mathbf{F}$  and  $J$ . This callback function will be implemented by callers. For this reason, our interface is defined as follows:

```
void BSnewton(
    int (*userEvalFunc)(          // callback function
        void *pUserData, // Input
        double *X,         // Input
        double *F,         // Output
        double *Jmat),      // Output

    void *pUserData, // data that defines F and J
    double tol, // tolerance
    int n, // number of variables/functions
    double *pRanges, // (optional) bounds for variables
    double *X, // array for variables
    double *F, // array for functions values
    BSrc *rc) // error code
```

To illustrate how this generic Newton method is used, we consider creation of catenary curve that may be represented in the following generic form<sup>[1]</sup>:

$$y(x) = u \cosh\left(\frac{x-v}{u}\right) + \beta$$

where  $u$ ,  $v$ , and  $\beta$  are three constants to be determined by the boundary conditions of the problem. Usually these conditions include two points from which the cable is being suspended and the length of the cable.

Assume the two points are given by  $\mathbf{p}_1 = (x_1, y_1)$  and  $\mathbf{p}_2 = (x_2, y_2)$ . Substituting them into the above equation we have

$$y_1 = u \cosh\left(\frac{x_1-v}{u}\right) + \beta \quad (2.2)$$

$$y_2 = u \cosh\left(\frac{x_2-v}{u}\right) + \beta \quad (2.3)$$

In addition to the above two equations, we will need one more equation to form a system of three equations in three unknowns. We further assume that the cable length  $L$  is given. Accordingly, the third equation would be the arc length of catenary curve, i.e.,

$$L = \int_{x_1}^{x_2} \sqrt{1 + y'(x)^2} dx.$$

Since

$$y'(x) = u \sinh\left(\frac{x-v}{u}\right) \left(\frac{1}{u}\right) = \sinh\left(\frac{x-v}{u}\right),$$

we have

$$\begin{aligned} L &= \int_{x_1}^{x_2} \sqrt{1 + \sinh^2\left(\frac{x-v}{u}\right)} dx = \int_{x_1}^{x_2} \cosh\left(\frac{x-v}{u}\right) dx \\ &= u \int_{x_1}^{x_2} \cosh\left(\frac{x-v}{u}\right) d\left(\frac{x-v}{u}\right) = u \sinh\left(\frac{x-v}{u}\right) \Big|_{x_1}^{x_2} \\ &= u \left[ \sinh\left(\frac{x_2-v}{u}\right) - \sinh\left(\frac{x_1-v}{u}\right) \right] \end{aligned} \quad (2.4)$$

Equations (2.2), (2.3), and (2.4) form a system of nonlinear equations. Let's rewrite them in the following forms:

$$\begin{aligned} f_1 &= u \cosh\left(\frac{x_1-v}{u}\right) + \beta - y_1 = 0 \\ f_2 &= u \cosh\left(\frac{x_2-v}{u}\right) + \beta - y_2 = 0 \\ f_3 &= u \left[ \sinh\left(\frac{x_2-v}{u}\right) - \sinh\left(\frac{x_1-v}{u}\right) \right] - L = 0 \end{aligned}$$

Then, the components of Jacobian matrix are:

$$\frac{\partial f_1}{\partial u} = \cosh\left(\frac{x_1 - v}{u}\right) - \frac{x_1 - v}{u} \sinh\left(\frac{x_1 - v}{u}\right)$$

$$\frac{\partial f_1}{\partial v} = -\sinh\left(\frac{x_1 - v}{u}\right)$$

$$\frac{\partial f_1}{\partial \beta} = 1$$

$$\frac{\partial f_2}{\partial u} = \cosh\left(\frac{x_2 - v}{u}\right) - \frac{x_2 - v}{u} \sinh\left(\frac{x_2 - v}{u}\right)$$

$$\frac{\partial f_2}{\partial v} = -\sinh\left(\frac{x_2 - v}{u}\right)$$

$$\frac{\partial f_3}{\partial \beta} = 1$$

$$\frac{\partial f_3}{\partial u} = \sinh\left(\frac{x_2 - v}{u}\right) - \sinh\left(\frac{x_1 - v}{u}\right) - \frac{1}{u} \left[ (x_2 - v) \cosh\left(\frac{x_2 - v}{u}\right) - (x_1 - v) \cosh\left(\frac{x_1 - v}{u}\right) \right]$$

$$\frac{\partial f_3}{\partial v} = \cosh\left(\frac{x_1 - v}{u}\right) - \cosh\left(\frac{x_2 - v}{u}\right)$$

$$\frac{\partial f_3}{\partial \beta} = 0$$

The following C code shows how Newton's method is used to find three unknowns  $u$ ,  $v$  and  $\beta$  (Pay attention to how user data is passed onto BSNewton and back to MyEvalFunc via pDataStruct)

```
// Callback function that evaluates functions and derivatives
static int MyEvalFunc(void *pDataStruct, double *X, double *F, double *Jmat)
{
    double    x1, x2, y1, y2, L, ch1, ch2, sh1, sh2, *pData;

    pData = (double *)pDataStruct;
    x1 = pData[0];
    y1 = pData[1];
    x2 = pData[2];
    y2 = pData[3];
    L = pData[4];

    ch1 = cosh((x1 - X[1]) / X[0]);
    ch2 = cosh((x2 - X[1]) / X[0]);
    sh1 = sinh((x1 - X[1]) / X[0]);
    sh2 = sinh((x2 - X[1]) / X[0]);

    F[0] = X[0] * ch1 + X[2] - y1; // f1
    F[1] = X[0] * ch2 + X[2] - y2; // f2
    F[2] = X[0] * (sh2 - sh1) - L; // f3

    Jmat[0] = ch1 - (x1 - X[1]) * sh1 / X[0];
    Jmat[1] = -sh1;
    Jmat[2] = 1.0;

    Jmat[3] = ch2 - (x2 - X[1]) * sh2 / X[0];
```

```

    Jmat[4] = -sh2;
    Jmat[5] = 1.0;

    Jmat[6] = sh2 - sh1 - (ch2 * (x2 - X[1]) - ch1 * (x1 - X[1])) / X[0];
    Jmat[7] = ch1 - ch2;
    Jmat[8] = 0.0;

    return IMSUCC;
} // End of MyEvalFunct

// Compute u, v, and beta from two points and cable length
static void GetThemByNewton(double tol, GPosition2d &pt1, GPosition2d &pt2,
                           double L, double& u, double& v, double& beta, int *rc)
{
    double    x1, x2, y1, y2, pData[5], X[3], F[3];
    BSrc      bsrc = 0;

    *rc = IMSUCC;
    x1 = pData[0] = pt1.x;
    y1 = pData[1] = pt1.y;
    x2 = pData[2] = pt2.x;
    y2 = pData[3] = pt2.y;
    pData[4] = L;

    X[0] = L / 4.0;
    X[1] = 0.5 * (x1 + x2) - 0.25 * (y2 - y1);
    X[2] = 0.7 * min(y1, y2);
    BSnewton(MyEvalFunct, pData, 1.0e-4, 3, NULL, X, F, &bsrc);
    if (BSERROR(bsrc))
    {
        *rc = IMCFAIL;
    }
    else
    {
        u = X[0];
        v = X[1];
        beta = X[2];
    }
} // End of GetThemByNewton

```

**Example:** With  $\mathbf{p}_1 = (x_1, y_1) = (-50, 100)$ ,  $\mathbf{p}_2 = (x_2, y_2) = (60, 120)$ ,  $L = 150$ , we take again  $u = L/3$ ,  $v = (x_1 + x_2)/2$ , and  $\beta = 0.7 \times \min(y_1, y_2)$  as initial approximations for  $u$ ,  $v$ , and  $\beta$ . After 7 iteration we obtain  $u = 39.72898$ ,  $v = -0.32893$  and  $\beta = 24.95907$  that guarantee  $f_1$ ,  $f_2$ , and  $f_3$  are all less than  $1.0e^{-8}$ .

In this example, we made a reasonably good estimation to the solution. If we set  $u = v = \beta = 1$  as a wild guess, Newton's method will fail. In the following section we discuss how to ensure it

will find the correct solution even with a poor estimation.

### 3 Steepest Descent Algorithm

Newton's method for solving nonlinear systems has a very fast convergence speed once a sufficiently accurate estimation is known. When poor estimation is given, it often diverges. In some computations such as minimum distance and curve/curve intersection, a good estimation can be obtained by performing subdivision (equivalent to bisection method). However, good estimations may not be readily available for other applications (e.g., creation of catenary curves). The *Steepest Descent* algorithm discussed in this section converges slowly to the solution, but it will often converge even for poor initial estimations. Therefore, this algorithm may be used for improving estimations. When sufficient accurate estimations are obtained, we then call Newton's method to refine the solutions at very fast convergence speed.

Let us define a mapping  $g : \mathbb{R}^n \rightarrow \mathbb{R}$ . The connection between the minimization of  $g$  from  $\mathbb{R}^n$  to  $\mathbb{R}$  and the solutions of a system of nonlinear equations is due to the fact that (2.2) has a solution at  $\mathbf{X} = (x_1, x_2, \dots, x_n)^T$  precisely when the function  $g$  defined by

$$g(x_1, x_2, \dots, x_n) = \sum_{i=0}^n f_i^2$$

has the minimal value 0. By introducing  $g$  we translate the multidimensional search into a sequence of searches in one dimension and hence avoid complex searches in multiple dimensions.

The Steepest descent is also known as the *gradient descent*. From vector calculus it is known that, for  $g : \mathbb{R}^n \rightarrow \mathbb{R}$ , the gradient of  $g$  at  $\mathbf{X}$  is denoted  $\nabla g$  whose components are the partial derivatives of  $g$ . That is:

$$\nabla g(\mathbf{X}) = \left( \frac{\partial g}{\partial x_1}, \frac{\partial g}{\partial x_2}, \dots, \frac{\partial g}{\partial x_n} \right)^T$$

The gradient of a function gives the direction of steepest increase (uphill). Thus a natural minimization algorithm is to go in the direction opposite the gradient (downhill) a certain amount. Let's evaluate  $g$  at the initial estimation  $\mathbf{X}$  and denote it by  $g_1$ . Since  $-\nabla g$  is the steepest descent direction, we expect to obtain smaller  $g$  if we evaluate it at

$$\mathbf{X}^{(3)} = \mathbf{X} - a_3 \nabla g(\mathbf{X}).$$

Different  $a_3$  results in different  $g_3 = g(\mathbf{X}^{(3)})$ . To derive the optimized step length  $a_3$ , we use the quadratic interpolation method that is outlined as follows<sup>[2,3]</sup>:

Step 1: Find  $a_3 \in (0, 1)$  such that  $g_3 < g(\mathbf{X})$ .

Step 2: Take  $a_2 = a_3/2$  and set  $\mathbf{X}^{(2)} = \mathbf{X} - a_2 \nabla g(\mathbf{X})$ .

Step 3: Evaluate  $g$  at  $\mathbf{X}^{(2)}$  and denote the result by  $g_2$ .

Step 4: Set  $a_0 = 0.5(a_2 - h_1/h_3)$  with

$$h_1 = (g_2 - g_1)/a_2,$$

$$h_2 = (g_3 - g_2)/(a_3 - a_2),$$

$$h_3 = (h_2 - h_1)/a_3.$$

Note these values are derived from quadratic interpolation at three points  $(0, g)$ ,  $(a_3, g_3)$ , and  $(a_2, g_2)$ .

Step 5: Compute the critical point  $\mathbf{X}^{(0)} = \mathbf{X} - a_0 \nabla g(\mathbf{X})$  and  $g_0 = g(\mathbf{X}^{(0)})$ .

Step 6: Select  $a$  from  $\{a_0, a_3\}$  so that  $g = g(\mathbf{X} - a \nabla g(\mathbf{X})) = \min\{g_0, g_1\}$ .

Step 7: Terminate and output  $\mathbf{X}$  if  $g < 0.1$ .

Step 8: Set  $\mathbf{X} = \mathbf{X} - a \nabla g(\mathbf{X})$  and  $g_1 = g$ . Go to Step 1.

To illustrate how the steepest descent algorithm can ensure the success of Newton's method, we use the same example discussed in the previous section. If we set  $u = v = \beta = 1$  and evaluate  $f_1$ ,  $f_2$ , and  $f_3$ , we have

```
X[0] 1.0000000000000000
X[1] 1.0000000000000000
X[2] 1.0000000000000000

F[0] 7.0467454121346943e+021
F[1] 2.1006052018952572e+025
F[2] 2.1013098764364706e+025
```

As it is seen, this is really a poor estimation and leads to a failure of Newton's method. By calling the steepest descent algorithm with 20 iterations (our maximum number of iteration), we obtain

```
X[0] 17.581266196429354
X[1] 2.3135043817601586
X[2] 0.97568540938299109

F[0] 73.719882310373237
F[1] 115.18909766891642
F[2] 255.39979963519886
```

It should be pointed out that the above result did not meet our convergence criterion  $g < 0.1$ . It would need 36 iterations to achieve so. Nevertheless this is a much better estimation than the original one. If we take the above values as our estimation for Newton's method, we obtain the converged solution after 6 iterations:

```
X[0] 39.728956078137841
X[1] -0.32892449943671215
X[2] 24.959086500355568

F[0] 2.5699194011963300e-005
F[1] 3.4360111058617804e-005
F[2] 5.3023404944951835e-005
```

## References

- [1] <http://en.wikipedia.org/wiki/Catenary>
- [2] Burden, R.J. and Faires, J.D. 2001), *Numerical Analysis*, 7th Ed., Brooks Cole.
- [3] Hildebrand, F.B.), *Introduction to Numerical Analysis*, 2nd Ed., Dover Publications, Inc.