A New Newton’s Method with Diagonal Jacobian Approximation for Systems of Nonlinear Equations

M.Y. Waziri, W.J. Leong, M.A. Hassan and M. Monsi
Department of Mathematics, Faculty of Science, University Putra Malaysia 43400 Serdang, Malaysia

Abstract: Problem statement: The major weaknesses of Newton method for nonlinear equations entail computation of Jacobian matrix and solving systems of n linear equations in each of the iterations. Approach: In some extent function derivatives are quit costly and Jacobian is computationally expensive which requires evaluation (storage) of n×n matrix in every iteration. Results: This storage requirement became unrealistic when n becomes large. We proposed a new method that approximates Jacobian into diagonal matrix which aims at reducing the storage requirement, computational cost and CPU time, as well as avoiding solving n linear equations in each iterations. Conclusion/Recommendations: The proposed method is significantly cheaper than Newton’s method and very much faster than fixed Newton’s method also suitable for small, medium and large scale nonlinear systems with dense or sparse Jacobian. Numerical experiments were carried out which shows that, the proposed method is very encouraging.

Key words: Nonlinear equations, large scale systems, Newton’s method, diagonal updating, Jacobian approximation

INTRODUCTION

Consider the system of nonlinear equations:

\[ F(x) = 0 \]  \hspace{1cm} (1)

where, \( F(x) : \mathbb{R}^n \rightarrow \mathbb{R}^n \) with the following properties:

- There exist \( x^* \) with \( F(x^*) = 0 \)
- \( F \) is continuously differentiable in a neighbourhood of \( x^* \)
- \( F(x^*) = J_F(x^*) \neq 0 \)

The most well-known method for solving (1), is the classical Newton’s method. However, the Newton’s method for nonlinear equations has the following general form: Given an initial point \( x_0 \), we compute a sequence of corrections \( \{s_k\} \) and iterates \( \{x_k\} \) as follows:

**Algorithm CN (Newton’s method):** where, \( k = 0, 1, 2... \) and \( J_F(x_k) \) is the Jacobian matrix of \( F \), then:

**Stage 1:** Solve \( J_F(x_k)s_k = -F(x_k) \)
**Stage 2:** Update \( x_{k+1} = x_k + s_k \)
**Stage 3:** Repeat 1-2 until converges.

The convergence of Algorithm CN is attractive. However, the method depends on a good starting point (Dennis, 1983). Newton’s method will converges to \( x^* \) provided the initial guess \( x_0 \) is sufficiently close to the \( x^* \) and \( J_F(x^*) \neq 0 \) with \( J_F(x) \) Lipchitz continuous and the rate is quadratic (Dennis, 1983), i.e.:

\[ \|x_{k+1} - x^*\| \leq h\|x_k - x^*\| \]  \hspace{1cm} (2)

For some \( h \).

Even though it has good qualities, CN method has some major shortfalls as the dimension of the systems increases which includes (Dennis, 1983) for details):

- Computation and storage of Jacobian in each iteration
- Solving system of n linear equations in each iteration
- More CPU time consumption as the equations dimension increases

There are several strategies to overcome the above drawbacks. The first is fixed Newton method, i.e., by setting \( J_F(x_k) \equiv J_F(x_0) \) for \( k>0 \). Fixed Newton is the easiest and simplest strategy to overcome the shortfalls...
for systems of nonlinear equations and it follows the following steps:

Algorithm FN (fixed Newton): Let $x_0$ be given:

Step 1: Solve $J(x_0)s_k = -F(x_k)$
Step 2: Set $x_{k+1} = x_k + s_k$ for $k = 0, 1, 2...$

FN method diminishes both the computation of the Jacobian (except for the first iteration) as well as avoiding solving $n$ linear system in each iteration but is significantly slower (Natasa and Zorna, 2001). The second strategy is inexact Newton method. This method avoids solving Newton equation (Stage 1 of Algorithm CN) by taking the correction $\{s_k\}$ satisfying (Dembo et al., 1982; Eisenstat and Walker, 1985):

$$r_k = J(x_k)s_k + F(x_k)$$

Inexact Newton method is given by the following algorithm:

Algorithm INM (Inexact Newton): Let $x_0$ be given:

Step 1: Find some $s_k$ which satisfies:

$$J(x_k)s_k = -F(x_k) + \gamma_k$$

Where:

$$\|\gamma_k\| \leq \eta_k \|F(x_k)\|$$

Step 2: Set:

$$x_{k+1} = x_k + s_k$$

where, $\{\eta_k\}$ is a forcing sequence. Letting $\eta_k \equiv 0$ it gives Newton method.

Another modification is quasi-Newton’s method, the method is the famous method that replaces derivatives computation with direct function computation and also replaces Jacobian or its inverse with an approximation which can be updated at each iterations (Lam, 1978; Denis, 1971). There are quite many modifications introduced to conquer some of the shortfalls (Drangoslav and Natasa, 1996; Hao and Qin, 2008; Natasa and Zorna, 2001), but most of the modifications requires to computes and store an $n \times n$ matrix (Jacobian) in each iterations (Natasa and Zorna, 2001). In some cases when the number of equations is sufficiently large it becomes computationally expensive and requires evaluation (and storage) of generally fully populated $J_p(x_k)$ of dimension $n \times n$ which requires more CPU time, that is why Newton method cannot handle large-scale system of nonlinear equations. In this study we propose a method that reduces computational cost, storage requirement, CPU time and also eliminates the need for solving $n$ linear system in each iteration. This is made possible by approximating the Jacobian into diagonal matrix. The proposed method is significantly cheaper than Newton method, so much faster than Fixed Newton’s method and is suitable for both small, medium and large scale systems of equations.

MATERIALS AND METHODS

A new Newton method with diagonal Jacobian: Consider the Taylor expansion of $F(x)$ about $x_k$:

$$F(x) = F(x_k) + F'(x_k)(x - x_k) + o(\|x - x_k\|)$$

Then the incomplete Taylor series expansion of $F(x)$ is given by:

$$\hat{F}(x) = F(x_k) + F'(x_k)(x - x_k) + o\left(\|x - x_k\|\right)$$

where, $F'(x_k)$ is the Jacobian of $F$ at $x_k$.

In order to incorporate correct information on the Jacobian matrix to the updating matrix, from (4) we impose the following condition (Albert and Snyman, 2007):

$$\hat{F}(x_{k+1}) = F(x_{k+1})$$

where, $\hat{F}(x_{k+1})$ is an approximated $F$ evaluates at $x_{k+1}$.

Then (4) turns into:

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

Hence we have:

$$F'(x(x_{k+1} - x_k)) \approx F(x_k) - F(x_{k+1})$$

We propose the approximation of $F(x_k)$ by a diagonal matrix. i.e.:

$$F(x_k) \approx D_k$$

where $D_k$ is a given diagonal matrix, updated at each iteration. Then (7) turns to:

$$D_{k+1}(x_{k+1} - x_k) \approx F(x_{k+1}) - F(x_k)$$
Since we require \( D \) to be a diagonal matrix, say \( D = \text{diag}(d_1, d_2, \ldots, d_n) \), we consider to let components of the vector \( \frac{F(x_i) - F(x_{i-1})}{x_i - x_{i-1}} \) as the diagonal elements of \( D_k \)
from (9) it follows that:

\[
d^{(i)}_{k+1} = \frac{F(x_{k+1}) - F(x_k)}{x^{(i)}_{k+1} - x^{(i)}_k}
\]

(10)

Hence:

\[
D_{k+1} = \text{diag}(d^{(i)}_{k+1})
\]

(11)

for \( k = 1, 2, \ldots, n \) and \( k = 0, 1, 2, \ldots \).

Where:

\[
F(x_{k+1}) = \text{The } i^{\text{th}} \text{ component of the vector } F(x_{k+1})
\]

\[
F(x_k) = \text{The } i^{\text{th}} \text{ component of the vector } F(x_k)
\]

\[
x_{k+1}^{(i)} = \text{The } i^{\text{th}} \text{ component of the vector } x_{k+1}
\]

\[
x_k^{(i)} = \text{The } i^{\text{th}} \text{ component of the vector } x_k
\]

\[
d^{(i)}_{k+1} = \text{The } i^{\text{th}} \text{ diagonal element of } D_{k+1}
\]

We use (14) (to safeguard very small \( x^{(i)}_{k+1} - x^{(i)}_k \) if only denominator is not equal to zero \( |x^{(i)}_{k+1} - x^{(i)}_k| > 10^{-8} \) for \( i = 1, 2, \ldots \), else set \( d^{(i)}_{k+1} = d^{(i)}_k \).

We propose the update for our proposed method (NDJ) as below:

\[
x_{k+1} = x_k - D_k^{-1}F(x_k)
\]

(12)

where, \( D_k \) is defined by (11), provided

\[
|x^{(i)}_{k+1} - x^{(i)}_k| > 10^{-8}.
\]

Else set \( d^{(i)}_{k+1} = d^{(i)}_k \) for \( k = 1, 2, \ldots \).

**Algorithm NDJ:** Consider \( F(x) : \mathbb{R}^n \rightarrow \mathbb{R}^n \) with the same property as (1):

Step 1: Given \( x_0 \) and \( D_0 = I_n \), set \( k = 0 \)

Step 2: Compute \( F(x_k) \)

Step 3: Compute \( x_{k+1} = x_k - D_k^{-1}F(x_k) \) where \( D_k \)

is defined by (11), provided \( |x^{(i)}_{k+1} - x^{(i)}_k| > 10^{-8} \)

else set \( d^{(i)}_{k+1} = d^{(i)}_k \) for \( k = 1, 2, \ldots, n \)

Step 4: If \( \|x_{k+1} - x_k\| + \|F(x_{k+1})\| \leq 10^{-8} \) stop else set

\( k = k + 1 \) and go to Step 2

**RESULTS**

In order to demonstrate the performance method NDJ, four prominent methods are compared and the comparison was based upon the following criterion:

- NDJ stands for method proposed in this study
- The Newton method (CN)
- The Fixed Newton method (FN)
- The Incomplete Jacobian Newton method (IJN)
- MRVF denotes Newton-like method with the modification of right-hand side vector

The MRVF was proposed in (Natasa and Zorna, 2001) and IJN proposed by (Hao and Qin, 2008). The stopping criterion used is \( \|x_{k+1} - x_k\| + \|F(x_k)\| \leq 10^{-8} \). We implemented the five methods (CN, FN, MRVF, IJN and NDJ) using MATLAB 7.0. All the calculations were carried out in double precision computer. We introduced the following notations: \( N \): number of iterations and CPU: CPU time in seconds. Problem 1-3 is to show the fitness of our method (NDJ) to small scale and Problem 4-11 are for large scale systems with dense or sparse Jacobian.

**Problem 1:** Consider the system of two nonlinear equations (Dennis, 1983):

\[
F(x) = \begin{cases} 
  x_1 + x_2 - 3 \\
  x_1^2 + x_2^2 - 9 
\end{cases} 
\]

\( x_0 = (1.5) \)

**Problem 2:** Consider the system of three nonlinear equations (Hao and Qin, 2008):

\[
F(x) = \begin{cases} 
  (x_1^2 + x_2^3 + x_3^3 + 1)(x_1 - 1) + x_1(x_2 + x_3 - 2) \\
  (x_1^2 + x_2^4 + x_3^4 + 1)(x_2 - 1) + x_2(x_1 + x_3 - 2) \\
  (x_1^3 + x_2^5 + x_3^5 + 1)(x_3 - 1) + x_3(x_1 + x_2 - 2) \\
  \end{cases} 
\]

\( x_0 = (3, -3, 3) \)

**Problem 3:** Consider the system of five nonlinear equations (Hao and Qin, 2008):

\[
F(x) = \begin{cases} 
  (x_1^2 + x_2^2 + x_3^2 + x_1^2 + x_2^2 + 1)(x_1 - 1) + x_1(x_2 + x_3 - 2) \\
  + x_2(x_1 + x_2 + x_3 - 4) \\
  (x_1^3 + x_2^3 + x_3^3 + x_1^3 + x_2^3 + 1)(x_2 - 1) + x_2(x_1 + x_3 - 2) \\
  + x_3(x_1 + x_2 + x_3 - 4) \\
  (x_1^4 + x_2^4 + x_3^4 + x_1^4 + x_2^4 + 1)(x_3 - 1) + x_3(x_1 + x_2 - 2) \\
  \end{cases} 
\]

\( x_0 = (-1.5, 3.5, -1.5, 3.5, -1.5) \)
Problem 4: Singular Broyden (Gomes-Ruggiero et al., 1982; Broyden, 1965):

\[ f_i(x) = ((3-hx_i)x_1 - 2x_2 + 1)^2 \]
\[ f_i(x) = ((3-hx_i)x_1 - x_{i-1} - 2x_{i+1} + 1)^2 \]
\[ f_n(x) = ((3-hx_n)x_n - x_{n-1} + 1)^2 \]
\[ x_i^0 = -1 \text{ and } h = 2 \]

Problem 5: Generalized function of Rosenbrock (Luksan, 1994):

\[ f_i(x) = -4c(x_2 - x_i^2)x_i - 1 - 2(1-x_i) \]
\[ f_i(x) = 2c(x_i - x_{i-1}) - 4c(x_{i+1} - x_i^2)x_i \]
\[ = 2(1-x_i), \quad i = 2,...,n-1 \]
\[ f_n = 2c(x_n - x_{n-1}^2), \quad x_i^2 = 1.2, \text{ and } c = 2 \]

Problem 6: Extend Rosenbrock (Hao and Qin, 2008):

\[ f_i(x) = -400x_i(x_2 - x_i^2) - 2(1-x_i) + x_i \left( \sum_{j=1}^{n} x_j \right) - n + 1 \]
\[ f_i(x) = 200(x_i - x_{i-1}^2) - 400x_i(x_{i+1} - x_i^2) - 2(1-x_i) \]
\[ \quad + x_i \left( \sum_{j=1}^{n} x_j \right) - n + 1, \quad i = 2,...,n-1 \]
\[ f_n = 200(x_n - x_{n-1}^2) + x_n \left( \sum_{j=1}^{n} x_j \right) - n + 1 \]
\[ x_0 = (1.2, 1.2, 1.2, 1.2,...)^T \]

Problem 7: Trigonometric-Exponential system (Luksan, 1994):

\[ f_i(x) = 3x_i^2 + 2x_i - 5 + \sin(x_i - x_2)\sin(x_i + x_2) \]
\[ f_i(x) = 3x_i^2 + 2x_{i+1} - 5 + \sin(x_i - x_{i+1})\sin(x_i + x_{i+1}) \]
\[ + 4x_i - x_{i+1}\exp(x_{i+1} - x_i) - 3 \]
\[ f_n(x) = 4x_n - x_{n-1}\exp(x_{n-1} - x_n) - 3 \]
\[ x_i^0 = 0, \quad i = 2,...,n-1 \]

Problem 8: System of n linear equation (Hao and Qin, 2008):

\[ F_j(x) = \left( \sum_{i=1}^{n} x_i^2 + 1 \right)(x_j - 1) + x_j \sum_{j=1}^{n} x_j - n + 1, \quad j = 1, 2,...,n-1 \]
\[ F_n(x) = \left( \sum_{i=1}^{n} x_i^2 + 1 \right)(x_n - 1) \]
\[ x_0 = (-1.53, 5, -1.5, 3.5, ...)^T \]

Problem 9: Structured Jacobian problem (Luksan, 1994):

\[ f_i(x) = -2x_i^2 + 3x_i - 2x_2 + 3x_{n-1} - x_{n-2} + 0.5x_{n-1} - x_n + 1 \]
\[ f_i(x) = -2x_i^2 + 3x_i - x_{i-1} - 2x_{i+1} + 3x_{n-1} - x_{n-2} + 0.5x_{n-1} - x_n + 1 \]
\[ f_i(x) = -2x_i^2 + 3x_i - x_{i-1} + 3x_{n-1} - x_{n-2} + 0.5x_{n-1} - x_n + 1 \]
\[ x_i^0 = -1, \quad i = 2,...,n \]

Problem 10: Broyden tridiagonal (More et al., 1981):

\[ f_i(x) = (3 - 2x_i)x_i - 2x_2 + 1 \]
\[ f_i(x) = (3 - 2x_i)x_i - x_{i-1} - 2x_{i+1} + 1 \]
\[ f_n(x) = (3 - 2x_n)x_n - x_{n-1} + 1 \]
\[ x_i^0 = 0 \]

Problem 11: Spedicato4 (More et al., 1981):

\[ F_i(x) = \frac{1}{10}(x_i - x_{i-1}) \quad \text{if } i \text{ odd} \]
\[ F_i(x) = -10(x_i - x_{i-1}) \quad \text{if } i \text{ even} \]
\[ x_0 = (-1.2, 1.2, 1.2, 1.2, 1.2,...)^T \]

**DISCUSSION**

In Table 10, the robustness index is given by (Natasa and Zorna, 2001):

\[ V_j = \frac{t_j}{n_j} \]

where, \( t_j \) number of success by method \( j \) and \( n_j \) is the number of problems attempted by method \( j \) and the large value of index shows better result and the best possible result is 1.

From Table 1, it can be seen that our proposed method (NDJ) is Cheaper than Newton method (CN), Fixed Newton method (FN) and better than (IJN) and (MRVF). However our method is slower than Newton method (CN) but much faster than Fixed Newton method (FN). In addition CN, MRVF and IJN required to computes and stores the Jacobian in each iteration where as NDJ only vector storage.

Table 1: Results of problems 1-3 (number of iteration/CPU time)

<table>
<thead>
<tr>
<th>Problems</th>
<th>CN</th>
<th>FN</th>
<th>MRVF</th>
<th>IJN</th>
<th>NDJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem 1</td>
<td>4/0.0001</td>
<td>16/0.0002</td>
<td>52(( \alpha = -0.5 ))</td>
<td>*</td>
<td>6/0.00010</td>
</tr>
<tr>
<td>Problem 2</td>
<td>7/0.0003</td>
<td>22/0.0002</td>
<td>*(( \alpha = -0.08 ))</td>
<td>*</td>
<td>11/0.0002</td>
</tr>
<tr>
<td>Problem 3</td>
<td>6/0.0008</td>
<td>59/0.0004</td>
<td>*(( \alpha = -0.08 ))</td>
<td>*</td>
<td>8/0.00020</td>
</tr>
</tbody>
</table>

* Means that particular method fails to converge.
Moreover from Table 2-9, our method (NDJ) also uses less computational cost and less CPU time than other four methods (FN, MRVF, CN and IJN), that is why our method (NDJ) is significantly cheaper than CN, MRVF and IJN and also much faster than FN and MRVF, this is more noticeable when the dimension increases. In all the test problem, our method (NDJ) shows a promising result. We can observe that as the dimension of the systems increases, global cost for FN, MRVF, CN and IJN methods increases exponentially where as our method (NDJ) grows linearly. This is because they all require to compute Jacobian matrix in each iterations.
### Table 6: Computational results for solving problem 8 (number of iteration/CPU time)

<table>
<thead>
<tr>
<th>n</th>
<th>CN (α = −0.01)</th>
<th>FN</th>
<th>MRVF</th>
<th>IJN</th>
<th>NDJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>8/0.0034</td>
<td>43/0.0029</td>
<td>58/0.0031</td>
<td>26/0.0014</td>
<td>28/0.0012</td>
</tr>
<tr>
<td>50</td>
<td>9/0.0054</td>
<td>49/0.0052</td>
<td>62/0.0050</td>
<td>28/0.0018</td>
<td>30/0.0014</td>
</tr>
<tr>
<td>80</td>
<td>9/0.0672</td>
<td>53/0.0583</td>
<td>79/0.0602</td>
<td>30/0.0024</td>
<td>32/0.0020</td>
</tr>
<tr>
<td>100</td>
<td>9/5.6500</td>
<td>565/1.2800</td>
<td>86/5.407</td>
<td>30/0.0099</td>
<td>33/0.0073</td>
</tr>
<tr>
<td>200</td>
<td>9/8.6590</td>
<td>607/6.6420</td>
<td>*</td>
<td>31/0.0157</td>
<td>34/0.0093</td>
</tr>
<tr>
<td>500</td>
<td>9/19.2550</td>
<td>62/16.8163</td>
<td>*</td>
<td>32/0.0263</td>
<td>36/0.0137</td>
</tr>
<tr>
<td>1000</td>
<td>9/196.1002</td>
<td>69/188.5926</td>
<td>*</td>
<td>34/0.0481</td>
<td>38/0.0210</td>
</tr>
<tr>
<td>5000</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>34/0.0319</td>
<td>38/0.0221</td>
</tr>
<tr>
<td>10000</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>34/0.2564</td>
<td>38/0.1486</td>
</tr>
<tr>
<td>50000</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>44/5.7020</td>
<td>52/0.9126</td>
</tr>
</tbody>
</table>

*: Means that particular method fails to converge

### Table 7: Computational results for solving problem 9 (number of iteration/CPU time)

<table>
<thead>
<tr>
<th>n</th>
<th>CN (α = −0.3)</th>
<th>FN</th>
<th>MRVF</th>
<th>IJN</th>
<th>NDJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>5/0.0024</td>
<td>12/0.0021</td>
<td>6/0.0023</td>
<td>18/0.0017</td>
<td>12/0.0011</td>
</tr>
<tr>
<td>50</td>
<td>5/0.0025</td>
<td>14/0.0022</td>
<td>6/0.0024</td>
<td>18/0.0018</td>
<td>16/0.0012</td>
</tr>
<tr>
<td>80</td>
<td>5/0.0031</td>
<td>15/0.0026</td>
<td>7/0.0027</td>
<td>20/0.0026</td>
<td>18/0.0018</td>
</tr>
<tr>
<td>100</td>
<td>5/0.0057</td>
<td>16/0.0048</td>
<td>8/0.0053</td>
<td>21/0.0026</td>
<td>18/0.0020</td>
</tr>
<tr>
<td>200</td>
<td>5/0.0125</td>
<td>16/0.0102</td>
<td>8/0.0119</td>
<td>23/0.0059</td>
<td>21/0.0032</td>
</tr>
<tr>
<td>500</td>
<td>5/2.9229</td>
<td>17/1.2160</td>
<td>8/1.3510</td>
<td>25/0.0495</td>
<td>23/0.0275</td>
</tr>
<tr>
<td>1000</td>
<td>5/16.0154</td>
<td>17/13.7132</td>
<td>9/14.5723</td>
<td>25/0.7141</td>
<td>23/0.3974</td>
</tr>
<tr>
<td>5000</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>27/1.0112</td>
<td>24/0.8623</td>
</tr>
<tr>
<td>10000</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>30/2.5921</td>
<td>25/1.0045</td>
</tr>
<tr>
<td>50000</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>39/29.0927</td>
<td>28/4.2741</td>
</tr>
</tbody>
</table>

*: Means that particular method fails to converge

### Table 8: Computational results for solving problem 10 (number of iteration/CPU time)

<table>
<thead>
<tr>
<th>n</th>
<th>CN (α = −0.08)</th>
<th>FN</th>
<th>MRVF</th>
<th>IJN</th>
<th>NDJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>9/0.0040</td>
<td>43/0.0271</td>
<td>12/0.0035</td>
<td>21/0.0034</td>
<td>24/0.0028</td>
</tr>
<tr>
<td>50</td>
<td>9/0.0063</td>
<td>54/0.1753</td>
<td>14/0.0052</td>
<td>24/0.0042</td>
<td>24/0.0036</td>
</tr>
<tr>
<td>80</td>
<td>11/0.0141</td>
<td>68/1.0642</td>
<td>15/0.0099</td>
<td>24/0.0067</td>
<td>26/0.0043</td>
</tr>
<tr>
<td>100</td>
<td>12/0.0197</td>
<td>79/1.9843</td>
<td>15/0.0132</td>
<td>26/0.0084</td>
<td>28/0.0069</td>
</tr>
<tr>
<td>200</td>
<td>12/0.0264</td>
<td>824/8.164</td>
<td>16/0.0207</td>
<td>27/0.0106</td>
<td>35/0.0083</td>
</tr>
<tr>
<td>500</td>
<td>12/0.0319</td>
<td>*</td>
<td>18/0.0297</td>
<td>27/0.0129</td>
<td>35/0.0094</td>
</tr>
<tr>
<td>1000</td>
<td>12/0.1279</td>
<td>*</td>
<td>22/0.0942</td>
<td>30/0.0281</td>
<td>35/0.0101</td>
</tr>
<tr>
<td>5000</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>31/0.0438</td>
<td>35/0.0185</td>
</tr>
<tr>
<td>10000</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>33/0.0818</td>
<td>38/0.0371</td>
</tr>
<tr>
<td>50000</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>51/9.0713</td>
<td>59/0.8410</td>
</tr>
</tbody>
</table>

*: Means that particular method fails to converge

### Table 9: Computational results for solving problem 11 (number of iteration/CPU time)

<table>
<thead>
<tr>
<th>n</th>
<th>CN (α = −0.5)</th>
<th>FN</th>
<th>MRVF</th>
<th>IJN</th>
<th>NDJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>8/0.0037</td>
<td>28/0.0027</td>
<td>38/0.0054</td>
<td>40/0.0023</td>
<td>34/0.0016</td>
</tr>
<tr>
<td>50</td>
<td>8/0.0042</td>
<td>35/0.0031</td>
<td>49/0.0096</td>
<td>43/0.0035</td>
<td>36/0.0023</td>
</tr>
<tr>
<td>80</td>
<td>8/0.0210</td>
<td>39/0.0194</td>
<td>63/0.0358</td>
<td>45/0.0130</td>
<td>38/0.0089</td>
</tr>
<tr>
<td>100</td>
<td>8/0.0279</td>
<td>42/0.0204</td>
<td>*</td>
<td>45/0.0167</td>
<td>38/0.0106</td>
</tr>
<tr>
<td>200</td>
<td>8/0.0514</td>
<td>47/0.4632</td>
<td>*</td>
<td>48/0.0268</td>
<td>39/0.0153</td>
</tr>
<tr>
<td>500</td>
<td>8/0.1826</td>
<td>54/0.4835</td>
<td>*</td>
<td>50/0.0618</td>
<td>43/0.0449</td>
</tr>
<tr>
<td>1000</td>
<td>8/1.2915</td>
<td>62/1.8165</td>
<td>*</td>
<td>54/0.0917</td>
<td>45/0.0615</td>
</tr>
<tr>
<td>5000</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>60/0.5041</td>
<td>48/0.3103</td>
</tr>
<tr>
<td>10000</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>68/1.3612</td>
<td>48/0.5190</td>
</tr>
<tr>
<td>50000</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>87/12.3091</td>
<td>54/1.7209</td>
</tr>
</tbody>
</table>

*: Means that particular method fails to converge

### Table 10: Robustness index table

<table>
<thead>
<tr>
<th>CN</th>
<th>FN</th>
<th>MRVF</th>
<th>IJN</th>
<th>NDJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.7108</td>
<td>0.5542</td>
<td>0.5904</td>
<td>0.9639</td>
<td>1</td>
</tr>
</tbody>
</table>

Another advantage of our method over FN, MRVF, CN and IJN methods is the storage requirement for the Jacobian matrix to only a vector storage. For instance when n = 1000, CN, MRVF and IJN stores 1000×1000 fully populated matrix which is equivalent to 10^6 memory locations but NDJ stores 1000 vectors equivalents to 1000 memory locations only, in general our method has reduced the matrix storage to some
vector storage only. These results indicated that NDJ is an improvement; with emphasis on eliminating matrix storage, reducing computational cost and CPU time, as well as avoiding solving n system of equations in each iteration only.

CONCLUSION

In this study, a modification of classical Newton’s method for solving system of nonlinear equations is presented. This method (NDJ) approximates the Jacobian into a diagonal matrix, by replacing the derivative computation by a direct function computation. Hence it reduces the computational cost, storage requirements of a matrix, CPU time and avoids solving n linear system in each iteration. The numerical results of the proposed method (NDJ) are very encouraging and it shows that its global cost increases linearly as the dimension of the nonlinear systems increases whereas in CN, FN, MRVF and IJN methods it increases exponentially; furthermore it shows that NDJ can achieve good performance for large nonlinear systems. Therefore to this end we can say that our method (NDJ) is significantly cheaper than the four methods and much faster than FN and MRVF. Finally we can concludes that our method (NDJ) is suitable for small, medium and large scale systems especially when the function derivatives are costly or can’t be done precisely.

REFERENCES