Journal of Mathematics and Statistics 6 (3): 246-252, 2010
ISSN 1549-3644

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:

$$F(x) = 0 \quad (1)$$

where, $F(x) : R^n \rightarrow R^n$ 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.:

$$\left\|x_{k+1} - x^*\right\| \leq h\left\|x_k - x^*\right\| \quad (2)$$

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

Corresponding Author: M.Y. Waziri, Department of Mathematics, Faculty of Science,
University Putra Malaysia 43400 Serdang, Malaysia

for systems of nonlinear equations and it follows the following steps:

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

Step 1: Solve $J_F(x_0)s_k = -F(x_k)$
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):

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

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:

$$J_F(x_k)\, s_k = -F(x_k) + \gamma_k$$

Where:

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

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×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 $J_F(x_k)$ of dimension n×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:

$$F(x) = F(x_k) + F'(x_k)(x - x_k) + o(\|x - x_k\|^2) \quad (3)$$

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

$$\hat{F}(x) = F(x_k) + F'(x_k)(x - x_k) + o(\|x - x_k\|^2) \quad (4)$$

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):

$$\hat{F}(x_{k+1}) = F(x_{k+1}) \quad (5)$$

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

Then (4) turns into:

$$F(x_{k+1}) \approx F(x_k) + F'(x_k)(x_{k+1} - x_k) \quad (6)$$

Hence we have:

$$F'(x)(x_{k+1} - x_k) \approx F(x_k) - F(x_{k+1}) \quad (7)$$

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

$$F'(x_k) \approx D_k \quad (8)$$

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

$$D_{k+1}(x_{k+1} - x_k) \approx F(x_{k+1}) - F(x_k) \quad (9)$$

Since we require D to be a diagonal matrix, says $D = diag(d^1, d^2,..., d^n)$, we consider to let components of the vector $\frac{F(x_k) - F(x_{k+1})}{x_{k+1} - x_k}$ as the diagonal elements of D_k, from (9) it follows that:

$$d_{k+1}^{(i)} = \frac{F_i(x_{k+1}) - F_i(x_k)}{x_{k+1}^{(i)} - x_k^{(i)}} \quad (10)$$

Hence:

$$D_{k+1} = diag(d_{k+1}^{(i)}) \quad (11)$$

for i = 1, 2, . . ., n and k = 0, 1, 2, . . ., n.

Where:

$F_i(x_{k+1})$ = The i^{th} component of the vector $F(x_{k+1})$
$F_i(x_k)$ = The i^{th} component of the vector $F(x_k)$
$x_{k+1}^{(i)}$ = The i^{th} component of the vector x_{k+1}
$x_k^{(i)}$ = The i^{th} component of the vector x_k
$d_{k+1}^{(i)}$ = The i^{th} diagonal element of D_{k+1} respectively

We use (14) (to safeguard very small $x_{k+1}^{(i)} - x_k^{(i)}$ if only denominator is not equal to zero $\left|x_{k+1}^{(i)} - x_k^{(i)}\right| > 10^{-8}$ for i = 1, 2,..., else set $d_k^{(i)} = d_{k-1}^{(i)}$.

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

$$x_{k+1} = x_k - D_k^{-1}F(x_k) \quad (12)$$

where, D_k is defined by (11), provided $\left|x_{k+1}^{(i)} - x_k^{(i)}\right| > 10^{-8}$. Else set $d_k^{(i)} = d_{k-1}^{(i)}$ for k = 1, 2, ...

Algorithm NDJ: Consider F(x): $\Re^n \to \Re^n$ 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} = $x_{k+1} = x_k - D_k^{-1}F(x_k)$ where D_k is defined by (11), provided $\left|x_{k+1}^{(i)} - x_k^{(i)}\right| > 10^{-8}$ else set $d_k^{(i)} = d_{k-1}^{(i)}$ for k = 1, 2,...,n
Step 4: If $\left\|x_{k+1} - x_k\right\| + \left\|F(x_k)\right\| \le 10^{-8}$ 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: Number of iterations, CPU time in seconds, storage requirement and robustness index. The methods are namely:

- 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 $\|x_{k+1}-x_k\|+\|F(x_k)\| \le 10^{-8}$. 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):

$$F(x) = \begin{cases} x_1 + x_2 - 3 \\ x_1^2 + x_2^2 - 9 \end{cases} \quad x_0 = (1,5)$$

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

$$F(x) = \begin{cases} (x_1^2 + x_2^2 + x_3^2 + 1)(x_1 - 1) + x_1(x_2 + x_3) - 2 \\ (x_1^2 + x_2^2 + x_3^2 + 1)(x_2 - 1) + x_2(x_1 + x_3) - 2 \\ (x_1^2 + x_2^2 + x_3^2 + 1)(x_3 - 1) + x_3(x_1 + x_2) - 2 \\ x_0 = (3,-3,3) \end{cases}$$

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

$$F(x) = \begin{cases} (x_1^2 + x_2^2 + x_3^2x_4^2 + x_5^2 + 1)(x_1 - 1) \\ +x_1(x_2 + x_3 + x_4) - 4 \\ (x_1^2 + x_2^2 + x_3^2x_4^2 + x_5^2 + 1)(x_2 - 1) \\ +x_2(x_1 + x_3 + x_4) - 4 \\ (x_1^2 + x_2^2 + x_3^2x_4^2 + x_5^2 + 1)(x_3 - 1) \\ +x_3(x_1 + x_2 + x_4) - 4 \\ (x_1^2 + x_2^2 + x_3^2x_4^2 + x_5^2 + 1)(x_4 - 1) \\ +x_4(x_1 + x_2 + x_3) - 4 \\ (x_1^2 + x_2^2 + x_3^2x_4^2 + x_5^2 + 1)(x_5 - 1) \end{cases}$$

$$x_0 = (-1.5, 3.5, -1.5, 3.5, -1.5)$$

Problem 4: Singular Broyden (Gomes-Ruggiero *et al*., 1982; Broyden, 1965):

$$f_1(x) = ((3-hx_1)x_1 - 2x_2 + 1)^2$$
$$f_i(x) = ((3-hx_i)x_i - x_{i-1} - 2x_{i+1} + 1)^2$$
$$f_n(x) = ((3-hx_n)x_n - x_{n-1} + 1)^2$$
$$x_i^0 = -1 \text{ and } h = 2$$

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

$$f_1(x) = -4c(x_2 - x_1^2)x_1 - 1 - 2(1-x_1)$$
$$f_i(x) = 2c(x_i - x_{i-1}) - 4c(x_{i+1} - x_i^2)x_i - 2(1-x_i), \quad i = 2,...,n-1$$
$$f_n = 2c(x_n - x_{n-1}^2), \quad x_i^0 = 1.2. \text{ and } c = 2$$

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

$$f_1(x) = -400x_1(x_2 - x_1^2) - 2(1-x_1) + x_1\left(\sum_{j=2}^{n} x_j\right) - n + 1$$
$$f_j(x) = 200(x_i - x_{i-1}^2) - 400x_i(x_{i+1}x_i^2) - 2(1-x_i) + x_i\left(\sum_{j\neq 1}^{n} x_j\right) - n + 1, \quad i = 2,...,n-1$$
$$f_n = 200(x_n - x_{n-1}^2) + x_n\left(\sum_{j\neq i}^{n} x_j\right) - n + 1$$
$$x_0 = (1.2,1,1.2,1,1.2,...)^T$$

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

$$f_1(x) = 3x_1^2 + 2x_2 - 5 + \sin(x_1 - x_2)\sin(x_1 + x_2)$$
$$f_j(x) = 3x_i^2 + 2x_{i+1} - 5 + \sin(x_i - x_{i+1})\sin(x_i + x_{i+1}) + 4x_i - x_{i-1}\exp(x_{i-1}x_i) - 3$$
$$f_n(x) = 4x_n - x_{n-1}\exp(x_{i-1} - x_i) - 3$$
$$x_i^0 = 0, \quad i = 2,...,n-1$$

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

$$F_j(x) = \left(\sum_{i=1}^{n} x_i^2 + 1\right)(x_j - 1) + x_j\sum_{i\neq j} x_i - n + 1,$$
$$F_n(x) = \left(\sum_{i=1}^{n} x_i^2 + 1\right)(x_n - 1)$$
$$j = 1,2,...,n-1$$
$$x_0 = (-1.53.5,-1.5,3.5,...)^T$$

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

$$f_1(x) = -2x_1^2 + 3x_1 - 2x_2 + 3x_{n-4} - x_{n-3} - x_{n-2} + 0.5x_{n-1} - x_n + 1$$
$$f_i(x) = -2x_i^2 + 3x_i - x_{i-1} - 2x_{i+1} + 3x_{n-4} - x_{n-3} - x_{n-2} + 0.5x_{n-1} - x_n + 1$$
$$f_n(x) = -2x_n^2 + 3x_n - x_{n-1} + 3x_{n-4} - x_{n-3} - x_{n-2} + 0.5x_{n-1} - x_n + 1$$
$$x_i^0 = -1, \quad i = 2,...,n$$

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

$$f_1(x) = (3-2x_1)x_1 - 2x_2 + 1$$
$$f_i(x) = (3-2x_i)x_i - x_{i-1} - 2x_{i+1} + 1$$
$$f_n(x) = (3-2x_n)x_n - x_{n-1} + 1$$
$$x_i^0 = 0$$

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

$$F_i(x) = \begin{cases} 1-x_i & \text{if i odd} \\ 10(x_i - x_{i-1}) & \text{if i even} \end{cases}$$
$$x_0 = (-1.2,...,-1.2,1)^T$$

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 problem 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)

Problems	CN	FN	MRVF	IJN	NDJ
Problem 1	4/0.0001	16/0.0002	52(α = –0.5)	*	6/0.00010
Problem 2	7/0.0003	22/0.0002	*(α = –0.08)	*	11/0.0002
Problem 3	6/0.0008	59/0.0004	*(α = –0.08)	*	8/0.00020

*: Means that particular method fails to converge

Table 2: Computational results for solving problem 4 (number of iteration/CPU time)

n	CN	FN	MRVF($\alpha = -0.08$)	IJN	NDJ
25	10/0.0045	609/6.71600	398/3.12650	21/0.0041	12/0.0032
50	10/0.0078	864/15.9725	467/6.98610	21/0.0046	12/0.0040
80	10/0.0162	968/24.8130	608/16.4521	22/0.0083	13/0.0051
100	13/0.0290	*	789/20.9741	23/0.0093	14/0.0072
200	13/0.0310	*	912/25.3170	23/0.0120	16/0.0094
500	13/0.0456	*	978/28.8672	32/0.0169	20/0.0108
1000	13/0.1981	*	*	35/0.0328	24/0.0142
5000	*	*	*	35/0.0526	24/0.0266
10000	*	*	*	35/0.0919	25/0.0695
50000	*	*	*	64/14.0945	36/1.3085

*: Means that particular method fails to converge

Table 3: Computational results for solving problem 5 (number of iteration/CPU time)

n	CN	FN	MRVF ($\alpha = -0.08$)	IJN	NDJ
25	4/0.0037	14/0.0026	7/0.0036	12/0.0023	11/0.0019
50	4/0.0042	14/0.0031	7/0.0040	15/0.0028	13/0.0021
80	4/0.0048	14/0.0035	7/0.0043	17/0.0030	13/0.0023
100	4/0.0067	14/0.0059	7/0.0054	17/0.0036	16/0.0025
200	4/0.0218	14/0.0145	7/0.0200	20/0.0059	17/0.0032
500	4/0.9934	17/0.0921	7/0.8132	21/0.0086	17/0.0059
1000	4/5.4910	18/0.1246	9/4.3292	22/0.0989	19/0.0899
5000	*	*	*	24/0.1460	20/0.0982
10000	*	*	*	25/0.2981	22/0.1284
50000	*	*	*	37/10.5629	27/2.0956

*: Means that particular method fails to converge

Table 4: Computational results for solving problem 6 (Number of iteration/CPU time)

n	CN	FN	MRVF ($\alpha = -0.08$)	IJN	NDJ
25	6/0.024000	45/0.02300	42/0.02400	7/0.00130	14/0.0013
50	6/0.031000	69/0.02800	47/0.03000	9/0.00201	6/0.00190
80	6/0.049700	73/0.03100	50/0.03540	18/0.0024	21/0.0021
100	6/1.043100	73/0.98010	51/1.00090	22/0.0041	27/0.0034
200	6/54.45500	78/21.8710	51/34.7210	26/0.0064	34/0.0045
500	6/106.7143	78/55.0377	64/87.5410	43/0.0176	54/0.0099
1000	6/109.6140	90/68.6521	87/95.9642	84/0.0487	85/0.0224
5000	*	*	*	91/0.1348	98/0.0943
10000	*	*	*	92/0.9610	98/0.5887
50000	*	*	*	108/20.3108	140/5.0612

*: Means that particular method fails to converge

Table 5: Computational results for solving problem 7 (number of iteration/CPU time)

n	CN	FN	MRVF ($\alpha = -0.08$)	IJN	NDJ
25	4/0.0434	*	10/0.0342	41/0.0215	20/0.0191
50	4/0.0480	*	10/0.0040	17/0.0362	20/0.0294
80	4/0.0973	*	12/0.0830	17/0.0749	21/0.0610
100	4/0.1516	*	12/0.0967	20/0.0841	21/0.0630
200	5/0.7898	*	12/0.4712	23/0.2654	25/0.0956
500	5/1.3091	*	14/0.7823	25/0.5134	28/0.1720
1000	5/6.1698	*	15/1.3514	28/0.8135	32/0.2561
5000	*	*	*	32/0.9104	35/0.4942
10000	*	*	*	33/0.9671	35/0.5261
50000	*	*	*	52/8.0160	58/1.0086

*: 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) growths 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)

n	CN	FN	MRVF ($\alpha = -0.01$)	IJN	NDJ
25	8/0.0034	43/0.0029	58/0.0031	26/0.0014	28/0.0012
50	9/0.0054	49/0.0052	62/0.0050	28/0.0018	30/0.0014
80	9/0.0672	53/0.0583	79/0.0602	30/0.0024	32/0.0020
100	9/5.6500	56/5.1280	86/5.407	30/0.0099	33/0.0073
200	9/8.6590	60/7.6420	*	31/0.0157	34/0.0093
500	9/19.2550	62/16.8163	*	32/0.0263	36/0.0137
1000	9/196.1002	69/188.5926	*	34/0.0481	38/0.0210
5000	*	*	*	34/0.0319	38/0.0221
10000	*	*	*	34/0.2564	38/0.1486
50000	*	*	*	44/5.7020	52/0.9126

*: Means that particular method fails to converge

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

n	CN	FN	MRVF ($\alpha = -0.3$)	IJN	NDJ
25	5/0.0024	12/0.0021	6/0.0023	18/0.0017	12/0.0011
50	5/0.0025	14/0.0022	6/0.0024	18/0.0018	16/0.0012
80	5/0.0031	15/0.0026	7/0.0027	20/0.0020	18/0.0018
100	5/0.0057	16/0.0048	8/0.0053	21/0.0026	18/0.0020
200	5/0.0125	16/0.0102	8/0.0119	23/0.0059	21/0.0032
500	5/2.9229	17/1.2160	8/1.3510	25/0.0495	23/0.0275
1000	5/16.0154	17/13.7132	9/14.5723	25/0.7141	23/0.3974
5000	*	*	*	27/1.1012	24/0.8623
10000	*	*	*	30/2.5921	25/1.0045
50000	*	*	*	39/29.0927	28/4.2741

*: Means that particular method fails to converge

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

n	CN	FN	MRVF ($\alpha = -0.08$)	IJN	NDJ
25	9/0.0040	43/0.0271	12/0.0035	21/0.0034	24/0.0028
50	9/0.0063	54/0.1753	14/0.0052	24/0.0042	24/0.0036
80	11/0.0141	68/1.0642	15/0.0099	24/0.0067	26/0.0043
100	12/0.0197	79/1.9843	15/0.0132	26/0.0084	28/0.0069
200	12/0.0264	82/4.8164	16/0.0207	27/0.0106	35/0.0083
500	12/0.0319	*	18/0.0297	27/0.0129	35/0.0094
1000	12/0.1279	*	22/0.0942	30/0.0281	35/0.0101
5000	*	*	*	31/0.0438	35/0.0185
10000	*	*	*	33/0.0818	38/0.0371
50000	*	*	*	51/9.0713	59/0.8410

*: Means that particular method fails to converge

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

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

*: Means that particular method fails to converge

Table 10: Robustness index table

	CN	FN	MRVF	IJN	NDJ
V	0.7108	0.5542	0.5904	0.9639	1

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 stores1000 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

Hao, L. and N. Qin, 2008. Incomplete Jacobian Newton method for nonlinear equation. Comput. Math. Appli., 56: 218-227.

Albert, A. and J.E. Snyman, 2007. Incomplete series expansion for function approximation. J. Struct. Multdisc. Optim., 34: 21-40.

Broyden, C.G., 1965. A class of methods for solving nonlinear simultaneous equations. Math. Comp., 19: 577-593.

Dembo, R.S., S.C. Eisenstat and T. Steihaug, 1982. Inexact Newton methods. SIAM J. Num. Anal., 19: 400-408.

Denis, J.E., 971. On the convergence of Broydens method for nonlinear systems of equations. J. Math. Comput., 25: 559-567.

Dennis, J.E., 1983. Numerical Methods for Unconstrained Optimization and Nonlinear Equations. 3rd Edn., Prince-Hall, Inc., Englewood Cliffs, New Jersey, pp: 378.

Drangoslav, H. and K. Natasa, 1996. Quasi-Newton's method with corrections. Novi Sad J. Math., 26: 115-127.

Eisenstat, S.C. and Walker, 1985. Choosing the forcing terms in inexact newton methos. SIAM J. Sci. Comput., 17: 16-32.

Gomes-Ruggiero, M.A., J.M. Martinez and A.C. Moretti, 1982. Comparing algorithms for solving sparse nonlinear systems of equations. SIAM J. Sci. Comput., 13: 459-483.

Lam, B., 1978. On the convergence of a Quasi-Newtons method for sparse nonlinear. Syst. Math. Comp., 32: 447-451.

Luksan, 1994. Inexact trust region method for large sparse systems of nonlinear equations. J. Optimiz. Theory Appli., 81: 569-590.

More, J.J., B.S. Grabow and K.E. Hillstrom, 1981. Testing unconstrained optimization software. ACM Trans. Math. Software, 7: 17-41.

Natasa, K. and L. Zorna, 2001. Newton-like method with modification of the righthand vector. J. Math. Comp., 71: 237-250.