J. Appl. Math. & Informatics Vol. **30***(2012), No. 3 - 4, pp. 395 - 411*
Website: http://www.kcam.biz

COMPUTATIONAL PITFALLS OF HIGH-ORDER METHODS FOR NONLINEAR EQUATIONS

SYAMAL K. SEN, RAVI P. AGARWAL* AND SANJAY K. KHATTRI

Abstract. Several methods with order higher than that of Newton methods which are of order 2 have been reported in literature for solving nonlinear equations. The focus of most of these methods was to economize on/minimize the number of function evaluations per iterations. We have demonstrated here that there are several computational pit-falls, such as the violation of fixed-point theorem, that one could encounter while using these methods. Further it was also shown that the overall computational complexity could be more in these high-order methods than that in the second-order Newton method.

AMS Mathematics Subject Classification : 65H04, 65H05, 41A25.
Key words and phrases : Convergence order, Fixed-point iteration, Newton method, nonlinear equations, optimal iterative methods.

1. Introduction

Many problems in science and engineering require solving the nonlinear equation

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

[1-13]. Some of the best known and probably the most used methods for solving the preceding equation are the Newton methods. The classical Newton method (NM) is as follows, where x_0 is an initial approximation (supplied) and k is a positive integer, say 4,

$$\begin{aligned} x_{i+1} &= x_i - \frac{f(x_i)}{f'(x_i)}, \quad i = 0, 1, 2, \ldots, \text{till} \\ & \frac{\|x_{i+1} - x_i\|}{\|x_{i+1}\|} \leq 0.5 \times 10^{-k} \quad \text{and} \quad |f'(x_i)| \neq 0 \end{aligned} \tag{2}$$

assuming that the iterates x_i converge. The Newton method converges quadratically [1-13]. There exists many modifications of the Newton method, which

Received April 28, 2011. Revised June 29, 2011. Accepted September 2, 2011. *Corresponding author.

© 2012 Korean SIGCAM and KSCAM.

improve the convergence rate [1-21 and references therein]. Here we consider two iterative schemes one fourth order and the other eighth order that are optimal according to Kung-Traub conjecture which states that an optimal iterative method based on $n + 1$ function evaluations could achieve a convergence order of 2^n. We demonstrate through numerical examples that both the schemes have the following computational drawbacks.

(i) Both these iteration schemes might not remain fixed-point (unlike Newton methods).

(ii) They could encounter division by zero or 0/0 form even for a distinct zero of a function (unlike Newton methods).

(iii) Computational complexity could be significantly more in these higher order schemes if the initial approximation is far away from the true zero (unlike Newton methods). Or, in other words, the high-order schemes could be more expensive computationally.

(iv) Number of function-evaluations per iteration is more in these high-order schemes (order ≥ 4) than that in Newton methods.

(v) Total number of function-evaluations for a specified initial approximation could be significantly more in these high order methods than in Newton methods.

Roughly speaking, if the order of an iterative scheme or, equivalently, the order of convergence of an iterative scheme is p and if the *ith* iterate is correct up to t digits, then the $(i + 1)st$ iterate will be correct up to about pt digits. For instance, if the eighth iterate of the classical (second order) Newton scheme for computing a zero of the function (polynomial) $f(x) = x^3 - 27$ is correct up to 4 digits, then the ninth iterate will be correct up to about $4 \times 2 = 8$ digits. This can be readily seen using the Matlab commands

```
>>format long g;x=27, for i=1:10,i ,x=x-(x^3-27)/(3*x^2),end;
```

where the initial approximation (0^{th} iterate) is chosen as $x_0 = 27$ (not a good one), the precision of computation used by Matlab was 15 digits. The 8^{th} iterate was then $x_8 = 3.00007335660249$ while the ninth iterate $x_9 = 3.00000000179367$.

Recently, even sixteenth order methods for nonlinear equations have been reported [16]. It is difficult to derive an iterative scheme of any arbitrary order of convergence. The questions then arise: Is there any optimal order of convergence of an iterative scheme that minimizes the amount of computation for solving any non-linear equation for a specified accuracy? Does the order of convergence always imply the same order of accuracy depicted in the foregoing numerical example? As to the first question, our numerical experiments over a large spectrum of problems consisting of zero-finding problems, higher order schemes need not be computationally more economical than a second or a third order process. Certainly, such a statement is shown mathematically as well as computationally valid when we use the following iterative schemes to compute the Moore-Penrose (minimum-norm least squares) inverse of a well-conditioned rectangular (including square ones) matrix A [23, 25].

$$\begin{aligned}
X_{k+1} &= X_k + Y_k \quad \text{(for only square matrix A)} && \text{(linear)}\\
X_{k+1} &= X_k(I + Y_k) && \text{(quadratic)}\\
X_{k+1} &= X_k(I + Y_k(I + Y_k)) && \text{(cubic)}\\
X_{k+1} &= X_k(I + Y_k(I + Y_k(I + Y_k))) && \text{(quartic)}\\
X_{k+1} &= X_k(I + Y_k(I + Y_k(I + Y_k(I + Y_k)))) && \text{(quintic)}\\
&\vdots\\
X_{k+1} &= X_k(I + Y_k(I + Y_k(I + Y_k(I + Y_k(\cdots(I + Y_k)\cdots))))) && (pth \text{ order})
\end{aligned}$$

where $Y_k = I - A\,X_k$ for $k = 0, 1, 2, \ldots$ with $X_0 = A^t/\mathrm{tr}(AA^t)$ and tr being the trace.

X_{k+1} for sufficiently large k will be the required Moore-Penrose inverse. Here if the matrix is well-conditioned (i.e. not having too near linearly dependent rows), the cubic process is computationally more economical than any other process, although sometimes the quadratic scheme also is a competitor. Or, in other words, the second order scheme or the third order scheme is most economical computationally. Comparative study of the amount of computation needed in various schemes is made in Table 1.

Table 1. Amount of computation required in various iterative schemes for specified accuracy (a=one matrix addition, m=one matrix multiplication).

Process	**Number of Iterations**	**Amount of Computation**
Linear	$n_1 = k$	$n_1 = (2a + m)$
Quadratic	$n_2 \approx \log_e k/\log_e 2$	$n_2(2a + 2m) = 2.885(a + m)\log_e k$
Cubic	$n_3 \approx \log_e k/\log_e 3$	$n_3(3a + 3m) = 2.731(a + m)\log_e k$
Quartic	$n_4 \approx \log_e k/\log_e 4$	$n_4(4a + 4m) = 2.885(a + m)\log_e k$
Quintic	$n_5 \approx \log_e k/\log_e 5$	$n_5(5a + 5m) = 3.107(a + m)\log_e k$

We present, in section 2, two iterative schemes one fourth order and the other eighth order without derivations [13, 26]. The fourth order scheme needs three function-evaluations per iteration while the eighth order scheme requires four function-evaluations per iteration. The scope of these schemes for real-world computations specifically with respect to ill-posed nonlinear equations having root-clusters and also with repeated roots is included in section 3. These high order schemes are compared with the widely known (second order) Newton scheme through numerical examples in this section. Section 4 comprises conclusions.

2. Fourth and Eighth Order Iterative Schemes

The fourth order scheme needing only three function-evaluations for obtaining one root of the nonlinear equation $f(x) = 0$ is as follows.

$$\begin{aligned} y_i &= x_i - \frac{f(x_i)}{f'(x_i)}, \quad x_{i+1} = y_i - \frac{f(y_i)}{2\left(\frac{f(y_i)-f(x_i)}{y_i-x_i}\right) - f'(x_i)}, \\ &\quad i = 0, 1, 2, \ldots, \quad \text{till} \quad \frac{\|x_{i+1} - x_i\|}{\|x_{i+1}\|} \le 0.5 \times 10^{-k} \end{aligned} \tag{3}$$

where x_0 is a specified initial approximation and $k =$ the number of significant digits to which the result (root) is required to be correct.

The eighth order scheme needing four function-evaluations per iteration to compute one root of the equation $f(x) = 0$, on the other hand, is as follows.

$$\begin{aligned} y_i &= x_i - \frac{f(x_i)}{f'(x_i)}, \quad z_i = y_i - \frac{f(y_i)}{2\left(\frac{f(y_i)-f(x_i)}{y_i-x_i}\right) - f'(x_i)}, \\ &\quad x_{i+1} = z_i - \frac{f(z_i)(x_i - y_i)^2(y_i - z_i)(x_i - z_i)}{A_i f'(x_i) + B_i f(x_i) + C_i f(y_i) + D_i f(z_i)}, \\ &\quad i = 0, 1, 2, \ldots, \quad \text{till} \quad \frac{\|x_{i+1} - x_i\|}{\|x_{i+1}\|} \le 0.5 \times 10^{-k} \end{aligned} \tag{4}$$

where

$$\begin{aligned} A_i &= (y_i - z_i)^2(x_i - z_i)(x_i - y_i), & B_i &= -(y_i - z_i)^2(3x_i - 2y_i - z_i), \\ C_i &= (x_i - z_i)^3, & D_i &= -(x_i - y_i)^2(x_i + 2y_i - 3z_i), \end{aligned}$$

and x_0 is a specified initial approximation and $k =$ the number of significant digits to which the result (root) is required to be correct.

3. Scope of Proposed Schemes in Real-World Computations

Single root computation and error due to finite precision All real-world computations are performed using a finite precision (finite word-length) machine (computer). So errors are introduced in computations. This is unlike the universal natural computations which use infinite precision and hence no errors exist in natural computations. In other words, the universal natural computer always takes exact real quantities and produces exact real quantities as outputs. The foregoing fourth and eighth order schemes as well as similar high and low order schemes compute only one root of the given equation-usually the one nearest to supplied initial approximation using a digital computer and is usually always erroneous [22].

Complex initial approximation for complex root of real function Our schemes like any other similar schemes including the Newton scheme need to use a complex initial approximation (imaginary part $\neq 0$) for computing a complex root of a real function. However, for a complex function such a restriction is not required. One may have a real initial approximation or an imaginary (real part $= 0$) initial approximation or a complex initial approximation (both real and imaginary parts not equal to zero). To obtain a complex root of a real polynomial, a real approximation will never be able to produce a complex value throughout the computations. The concerned arithmetic will remain always in real field and will not have any scope to shift from real field to complex field. In this context, by real we imply rational only rational numbers and not the irrational numbers. Any computer can represent only a very (negligibly) small fraction of rational numbers out all possible (infinite) rational numbers and cannot represent an irrational number in its numerical computations.

Multiple roots-deflation of function/polynomial In a finite precision computation, for a multiple root real/complex the iterates x_i will go on oscillating around the root in Newton scheme as well as similar higher order schemes including the foregoing ones (Schemes (2)-(4)). This is due to the error in computing the function and its derivative, both of which tend to zero. While the numerator, viz. the function tends to zero faster than the denominator, viz. the derivative of the function, the error due to division by a small number becomes the reason for numerical oscillations. Once an iterate x_i goes too close to a multiple root, the error is more pronounced resulting in a root relatively farther away from the previous root. In subsequent iterations, the iterates x_i will oscillate around the true multiple root without converging. To obviate this oscillation, one can use deflated Newton method [23,24]. The deflated Newton scheme is as follows.

$$\begin{aligned} &x_{i+1} = x_i - \frac{f^q(x_i)}{f^{q+1}(x_i)}, \\ &i = 0, 1, 2, \ldots, \text{till } \frac{|x_{i+1} - x_i|}{x_{i+1}} \leq 0.5 \times 10^{-k} \text{ and} |f^{q+1}(x_i)| \neq 0, \end{aligned} \tag{5}$$

where $q = 0, 1, 2, \ldots, s-1, s$ being the number of repeated roots and x_0 is an initial approximation (numerically supplied). When $s = 1 (q = 0)$, the deflated Newton method reduces to the conventional Newton method. Also, it is not necessary to know beforehand the number of repetitions of a root. Near a repeated root, the iterates x_i oscillate around the root (without converging) due to rounding errors present in finite precision computation; this serves as an indication for the repetition of a root. The deflation starts. Since either x_0 (the initial approximation) or the actual root could be a complex number, complex arithmetic is used for computing a zero, using the foregoing scheme.

Multiple zeros can only be found if the given function is a polynomial (of finite degree) or a function having a polynomial factor and a transcendental factor (polynomial of degree infinity). In fact, a transcendental function (having no polynomial factor which is usually the case) cannot have multiple zeros, it may have zero-clusters though [24].

Root-clusters: Ill-posed problems All the methods including the foregoing schemes (2)-(5) so far developed produce varying degree of accuracy and often the roots computed by high or low order fixed-point iteration schemes are not acceptable. In the present 21st century computing environment, the best approach is a simple exhaustive search after locating/bracketing a root-cluster within a sufficiently small interval/region[24].

Numerical experiment using Matlab

Example 1. Consider the polynomial

$$f(x) = x^4 - 10.002x^3 + 36.015999x^2 - 56.039993998x + 32.031991992$$

having a zero-cluster. Its derivative is

$$f'(x) = 4x^3 - 3 \times 10.002x^2 + 2 \times 36.015999x - 56.039993998.$$

$f(x)$ has zeros 1.999, 2.001, 2.002, 4. The Matlab commands (without stopping condition based on relative error) used for the fourth order scheme (3) are in the script file **fourthorderrootcluster** which is as follows

```
clear all; format long g; x=20;
for n=1:7, y = x - fc(x)/fcp(x);
x=y-(fc(y)/((2*(fc(y)-fc(x))/(y-x))-fcp(x))), end;
```

where the function $fc(x)$ and its derivative $fcp(x)$ subprograms are

```
function [fc] = fc(x)
fc=x^4-10.002*x^3+36.015999*x^2-56.039993998*x+32.031991992;
```

and

```
function [fcp] = fcp(x)
fcp=4*x^3- 3*10.002*x^2 + 2* 36.015999*x-56.039993998;
```

respectively.

On execution of the program **fourthorderrootcluster**, the successive iterates x_i are

11.8827704314505, 7.57084912651869, 5.32653899020177, 4.27485980243537, 4.00512009650793, 4.0000000015191, 4.00000000000001.

We see that the iterates x_i with the initial approximation $x_0 = 20$ converge to the distinct zero 4. Using the factor $x-4$ we may deflate the polynomial to make it a third order polynomial and try to compute the zeros in the zero-cluster. Or, without deflating, we take another appropriate initial approximation and try to obtain a zero in the cluster. The proposed schemes including all existing fixed-point schemes so far available are not likely to perform well unless we increase the precision of computation beyond the standard precision of 15 digits [24]. If we replace, in the script file **fourthorderrootcluster,** $x = 20$ by $x = 2$, then on executing the program, we get the successive iterates x_i as

2.00199207494168, 2.00199999879958, 2.00199999737707, 2.00199999737707,
2.00199999737707, 2.00199999737707, 2.00199999737707,

where the zero (iterate) $x_7 = 2.00199999737707$ (correct up to 9 decimal digits), which is closer to 2.002 than to 2.001 or 1.999 which were closest to the initial approximation $x_0 = 2$. Although the accuracy does not seem to be too bad here for this small polynomial, a deflation could result in unacceptable errors while computing the other zeros in the cluster. Also, for a polynomial of higher degree with two or more zero-clusters, the error will be simply unacceptable [24] for both the high order as well as low order schemes. These errors are due to the fixed precision which is usually in most computation 15 digits only. The best approach, in the current ultra-high speed computing era with over 90% computing power wasted due to non-usage of laptops/desktops would be to use exhaustive search in a significantly narrow region in which the zeros of a cluster are situated [24].

If we now consider the Matlab program **eighthordermethodrootcluster**, viz.

```
clear all; format long g;
x=20, for i=1:7, y=x-fc(x)/fcp(x); n=fc(y);
d=2*(fc(y)-fc(x))/(y-x) -fcp(x);z= y-n/d;
A=(x-y)*(x-z)*(y-z)^2; B=-(y-z)^2*(3*x-2*y-z);C=(x-z)^3;
D=-(x-y)^2*(2*y-3*z+x); n1=fc(z)*(x-y)^2*(y-z)*(x-z);
d1=A*fcp(x)+B*fc(x)+C*fc(y)+D*fc(z); x=z-n1/d1, end;
```

where fc and fcp are as defined above with initial approximation $x_0 = 20$, then we obtain the iterates x_i as

$9.72271212208445, 5.57654778173109, 4.13064288029322, 4.00000027522581, NaN, NaN.$

This numerical results for this small polynomial indicates that the eighth order method has performed worse than the fourth order method even for computing the distinct zero which is 4 here. Further, it did not remain fixed-point as as NaN (not a number implying 0/0 or ∞/∞) in the standard 15 digits computation. If we now replace in the program **eighthordermethodrootcluster**, $x = 20$ by $x = 2$ and execute the program, we get the iterate x_i as

2.00200008338187 (correct up to 8 decimal digits), $NaN, NaN, NaN, NaN, NaN.$

The eighth order method is less accurate than the fourth order method. Furthermore, it violates the character of a fixed-point iterative scheme by virtue of producing NaN. This is undesirable. However, by arranging the arithmetic computation in the iteration scheme, accuracy may be marginally improved.

Example 2. Consider now the 10th degree polynomial $f(x)$ with multiple zero $5, 5, 5, 5, 5, 5, 5, 5, 5, 5$. The function subprogram **fm** and its derivative subprogram **fmp** are

```
function [fm]=fm(x)
fm=x^10-50*x^9+1125*x^8-15000*x^7+131250*x^6-787500*x^5
+3281250*x^4-9375000*x^3+17578125*x^2-19531250*x+9765625;
```

and

```
function [fmp]=fmp(x)
fmp=10*x^9-9*50*x^8+8*1125*x^7-7*15000*x^6+6*131250*x^5-
5*787500*x^4+4*3281250*x^3-3*9375000*x^2+2*17578125*x
-19531250;
```

respectively. The Matlab program **fourthorderrepeatedroots** is as follows.

```
clear all; format long g; x=20;
for n=1:40, y=x-fm(x)/fmp(x);
x=y-(fm(y)/((2*(fm(y)-fm(x))/(y-x))-fmp(x))),end;
```

On execution of this program, we get the successive iterates x_i as

16.7718336352876, 14.2384044757925, 12.2501973696347, 10.6898744838891, 9.46535038866577, 8.5043574599222, 7.75018086771735, 7.15831144320801, 6.69381888317328, 6.32929028116822, 6.04321231118897, 5.81870002175088, 5.64250496585513, 5.50421818160933, 5.39584227120809, 5.31375999964652, 5.21688307830152, 5.2338015566893, 5.31083436696033, 5.19271762687222, 5.24636188076744, 5.19499188820688, 5.21873289540113, 5.18922276358921, 4.99024237143234, 5.29580602486053, 5.24829790650256, 5.24968427013892, 5.18304715429968, 5.22671259095404, 5.17090654760647, 5.14590654760647, 5.20919768684698, $NaN, NaN, \ldots$

The result produced by the **fourthorderrepeatedroots** program is simply unacceptable. The iterates go on oscillating around the exact zero 5. It is even incorrect at the very first decimal place. After some oscillations it produces NaN (not a number, i.e. $0/0$ or ∞/∞) which is an additional drawback of high order iterative methods. The second order deflated Newton scheme (5), on the other hand, will produce all the ten zeros 5,5, 5, 5, 5, 5, 5, 5, 5, 5 very accurately (rather exactly within the precision) by successive deflation (nine times). This deflation can be symbolically performed in Matlab and then the numerical computation can be superimposed. The iterates in the second order Newton scheme (2) too will oscillate around the root 5 due to fixed-precision computation involving rounding errors. In addition, the scheme could produce NaN. As a matter of fact, both low order as well as high order schemes will fare badly as such unless appropriate remedial measures are adopted.

If we now execute the Matlab program **eighthordermethodrepeatedroots**, viz.,

```
clear all; format long g;
x=20, for i=1:40,
y=x-fm(x)/fmp(x);n=fm(y);d=2*(fm(y)-fm(x))/(y-x)-fmp(x);
z=y-n/d;
A=(x-y)*(x-z)*(y-z)^2; B=-(y-z)^2*(3*x-2*y-z); C=(x-z)^3
D=-(x-y)^2*(2*y-3*z+x); n1=fm(z)*(x-y)^2*(y-z)*(x-z);
d1=A*fmp(x)+B*fm(x)+C*fm(y)+D*fm(z); x=z-n1/d1,end;
```

then the successive iterates x_i will oscillate around the zero 5 and will not approach reasonably close to 5. It is even worse than the foregoing fourth order scheme both in terms of number of iterations as well as in terms of accuracy. Although NaN does not appear in this eighth order scheme, the iterates will continue to oscillate for ever in a way which is usually worse than a lower order method. These iterates will be as follows.

15.9979173493121, 13.0636124014867, 10.9121961818964, 9.33478966409772, 8.17824389683044, 7.33027091290792, 6.70854179391171, 6.25269342675679, 5.91846760696389, 5.67341591862772, 5.49379534694353, 5.36253632589853, 5.25709333231426, 5.30288526913316, 4.85109560106865, 4.88907378669703, 4.89404237321551, 4.90409909315816, 5.56960552492844, 5.41770782457898, 5.30335312695354, 5.30824693347822, 5.19180833637212, 5.21561644685535, 5.01809362914905, 5.88932562621055, 5.6520451695298, 5.47809204196954, 5.35021936639331, 5.24984132008913, 5.24283552977236, 5.25193398430819, 5.22383270927142, 5.16744730887884, 5.17658659944416, 4.87606980008085, 4.75738936255274, 4.95554630760618, 5.95245208860072, 5.69833526864787.

Example 3. Consider the real polynomial1 $f(x) = x^{10} - 3^{10}$ having 8 complex (conjugate) zeros and two real zeros $-3, 3$. The following Matlab program **fourthordercomplexrealroots** for the fourth order scheme, viz.

```
clear all; format long g; x=10^10
for n=1:100, y=x-fcr(x)/fcrp(x);
x=y-(fcr(y)/((2*(fcr(y)-fcr(x))/(y-x))-fcrp(x))),end;
```

along with the following subprograms **fcr** and **fcrp**, viz.

```
function [fcr]=fcr(x)
fcr=x^10-3^10;
```

and

```
function [fcrp]=fcrp(x)
fcrp=10*x^9;
```

will need 93 iterations to obtain 3 as a zero (root) when the initial approximation was chosen as 10^{10}. The output from 94th iteration onwards resulted in NaN (i.e. 0/0 form here) and not 3. If we take the initial approximation as $3^2 = 9$ then in 7 iterations of the fourth order scheme, we obtain the root as 3. The output from 8th iteration onwards becomes NaN indicating that the iteration does not remain fixed-point. The eighth order scheme, viz.

```
clear all; format long g;
x=10^10, for i=1:80, y=x-fcr(x)/fcrp(x); n=fcr(y);
d=2*(fcr(y)-fcr(x))/(y-x)-fcrp(x);
z=y-n/d;
A=(x-y)*(x-z)*(y-z)^2; B=-(y-z)^2*(3*x-2*y-z); C=(x-z)^3;
D=-(x-y)^2*(2*y-3*z+x); n1=fcr(z)*(x-y)^2*(y-z)*(x-z);
d1=A*fcrp(x)+B*fcr(x)+C*fcr(y)+D*fcr(z); x=z-n1/d1, end;
```

will need 72 iterations to obtain 3.00000000000029 as a zero when the initial approximation was chosen as 10^{10}. The output from 73rd iteration onwards resulted in NaN (i.e. 0/0 form here) and not 3. If we take the initial approximation as $3^2 = 9$ then in 5 iterations of the eighth order scheme, we obtain the root as 3. The output from 6th iteration onwards becomes NaN indicating that the iteration does not remain fixed-point. The output NaN is due to numerical instability which is due to insufficient precision of computation. In other words, the standard precision of 15 digits, which is employed in almost all real-world computational problems is insufficient for the high order methods. To obviate this precision problem , one needs to use non-standard precision which should be indefinitely large (finite though) in this context. vpa (variable precision arithmetic) in Matlab for sufficiently large precision computation may be employed, although it has its own limitations.

The Newton scheme (2) (used here without stopping condition based on relative error), viz. **newtonsecondorder**

```
clear all; format long g; x=9;
for n=1:18, x=x-f(x)/fp(x), end;
```

along with the subprograms f and fp, viz.

```
 function [f]=f(x)
f=x^10-3^10;
```

and

```
function [fp]=fp(x)
fp=10*x^9;
```

on the other hand produces the root as 3 in 15th iteration and continue to remain 3 afterwards unlike the fourth order scheme. If we take an initial approximation -10^{10} then the Newton scheme produces -3 as the root at 112th iteration. The root remains as -3 in subsequent iterations; no *NaN* is produced. The fourth order scheme with $x = -10^{10}$ as the initial approximation produces -3 as the root at 93rd iteration. The root becomes NaN from 94th iterations onwards.

It may be seen from Table 2 that, for the foregoing equation $f(x) = x^{10} - 3^{10} = 0$, when the initial approximation is chosen as $x_0 = 10^{10}$ under the Matlab standard precision of 15 digits, the (second order) Newton method needs the least number of function-evaluations.

TABLE 2. Overall number of function-evaluations in 2nd, 4th and 8th order schemes to obtain root 3 for the equation $x^{10} - 3^{10} = 0$ under Matlab standard precision of 15 digits. Initial approximation chosen as $x_0 = 10^{10}$.

Scheme	**No. of function evaluations per iteration**	**Overall number of function evaluations**	**Remark**
2nd order	2	$112 \times 2 = 224$	Cheapest
4th order	3	$93 \times 3 = 279$	
8th order	4	$72 \times 4 = 288$	Costliest

Besides the cost, viz., the computational complexity which could be more for a higher order scheme, the numerical instability could also be more pronounced in this scheme as NaN (Not a Number such as 0/0 and ∞/∞ form) is more likely to occur here than in a low order scheme. It may be remarked that infinite precision computation which is only in the jurisdiction of natural computation – ever error-free, ever exact, ever perfect - is completely out of bound for the available computers in the world and will remain so for ever. Thus high order schemes with optimal number of function evaluations per iteration are at best of academic interest.

The Matlab **roots** command, viz.,

```
>> roots([1  0  0  0  0  0  0  0  0  0  -3^10])
```

for the equation (polynomial) $x^{10} - 3^{10} = 0$ produces the roots

```
ans =
-3
-2.42705098312485 + 1.76335575687742i
-2.42705098312485 - 1.76335575687742i
-0.92705098312484 + 2.85316954888546i
-0.92705098312484 - 2.85316954888546i
0.927050983124845 + 2.85316954888546i
0.927050983124845 - 2.85316954888546i
3
2.42705098312484 + 1.76335575687742i
2.42705098312484 - 1.76335575687742i
```

which result in the polynomial when poly(ans) is executed, whose coefficients are

1 (coefficient of x^{10})

$-5.32907051820075e{-}015$	$-1.99840144432528e{-}014$
$1.4210854715202e{-}014$	$6.96331881044898e{-}013$
$-2.8421709430404e{-}012$	$3.97903932025656e{-}012$
$-8.29913915367797e{-}012$	$3.41060513164848e{-}012$
$-1.81898940354586e{-}012$	-59049.0000000001(constant term $\cong -3^{10}$).

The Matlab constructed polynomial is sufficiently good in a real-world situation. While Matlab roots produce all the **roots** of the equation, the schemes here produce only one root at a time.

Example 4. We now consider a complex polynomial $f(x) = x^2 + (3 + 5i)x + 7i$ whose zeros are complex and take an initial approximation 5. Since the polynomial is complex, the computations are carried out using complex arithmetic and we obtain the complex zeros without any problem even though the initial approximation is a real number. The Matlab program for the fourth order scheme (omitting stopping condition based on relative error) **fourthordercomplex-ploy** along with its function and its derivative subprograms **fco, fcop** are

```
clear all; format long g; x=5;
for n=1:10,
y=x-fco(x)/fcop(x);
x=y-(fco(y)/((2*(fco(y)-fco(x))/(y-x))-fcop(x))),
end;

function [fco]=fco(x)
fco=x^2+(3+5i)*x-4+7i;
```

and

```
function [fcop]=fcop(x)
fcop=2*x+(3+5i);
```

On execution of the program **fourthordercomplexploy**, we obtain the successive iterates x_i are

$0.15754392789989 - 1.79189522142255i, -0.939407647732583 - 2.04102205128106i - 1.00000170547744 - 1.99999045181327i, -1 - 2i, -1 - 2i$, NaN +NaN$i$, NaN +NaN$i$, NaN +NaN$i$, NaN +NaN$i$, NaN +NaN$i$.

To obtain the other complex zero we need to take a negative (real) initial approximation, say $x_0 = -5$. A positive initial approximation will always result in the earlier zero $-1-2i$. In the foregoing fourth order program, if we replace $x = 5$ by $x = -5$, then we get the other root $-2-3i$ in the 5th iteration while 8th iteration onwards we get NaN +NaNi. When we use the Newton scheme with initial approximation $x_0 = 5$, we obtain the root in the 7th iteration (correct up to 9 decimal digits) and no NaN +NaNi is produced. The complex initial approximation $-1-i$ will result in the root $-1-2i$ while the complex initial approximation $-3-4i$ will result in the other root $-2-3i$.

4. Conclusions

Matlab **roots** *versus the high order iterative schemes* The high order iterative schemes like other similar existing schemes are definitely of academic interest. The schemes presented here finds one root of an equation at a time while the Matlab **roots** produce all the roots at the same time fairly accurately as long as the roots are not considerably *clustered or highly repeated.* The implementation details of the command **roots** are not known to a Matlab user. The polynomial equation (with a root-cluster) whose roots are $1.999, 2.001, 2.002$ and 4 is, using the Matlab command,

```
>> format long g; poly([1.999 2.001 2.002 4])
```

$x^4 - 10.002x^3 + 36.015999x^2 - 56.039993998x + 32.031991992 = 0$, where the output of the command was

```
ans = 1 -10.002 36.015999 -56.039993998 32.031991992
```

The Matlab command **roots(ans)** then produces all the roots 4.00000000000001, 2.00199999527988, 2.00100000706712, and 1.99899999765298 at the same time, which are reasonably good. The presented schemes find the roots with an appropriate initial approximation one at a time, which are also reasonably good. However, the iterates in higher order schemes may not remain fixed after sufficient number of iterations. See, for instance, the iterates in the eighth order scheme in Example 1. For the foregoing repeated roots in Example 2, we have similarly

```
>> format long g; poly([5 5 5 5 5 5 5 5 5 5])
   ans = 1 −50 1125 −15000 131250 −787500 3281250
        −9375000 17578125 −19531250 9765625
>> roots(ans)
ans =
5.2708860862188
5.21657123682004 + 0.161087508718445i
5.21657123682004 − 0.161087508718445i
5.07796968457484 + 0.255734028629629i
5.07796968457484 − 0.255734028629629i
4.91377060087738 + 0.249729492943409i
4.91377060087738 − 0.249729492943409i
4.78658703847331 + 0.151382226453981i
4.78658703847331 − 0.151382226453981i
4.73931679229006
```

The foregoing 2 real roots and 8 complex roots are not acceptable when the actual roots are all 5. Thus the Matlab **roots** command did not fare well. From Example 2 we see that higher the order of the iterative scheme is, worse are the computed roots. This implies that the numerical instability (due to fixed precision of computation) becomes more pronounced as we go from lower order to higher order schemes. Incidentally, the deflated Newton scheme (second order) is bets suited to tackle the multiple root problem. For this problem, the deflated Newton method produces all the ten roots, viz. 5 very accurately.

Overall number of function-evaluations is more important than that per iteration and numerical instability From Example 3, we see that, to achieve a specified accuracy, the overall number of function-evaluations is more important than that per iteration. Not only the cost, viz., the computational complexity could be more for a higher order scheme, but also the numerical instability could be more pronounced in this scheme as NaN (Not a Number such as $0/0$ and ∞/∞ form) is more likely to occur here than in a low order scheme. It may be remarked that infinite precision computation which is only in the jurisdiction of natural computation – ever error-free, ever exact, ever perfect – is completely out of bound for the available computers in the world and will remain so for ever. In the realm of a variable precision computation, to allow indefinitely large precision for an iterative scheme could be a memory problem unlike for a non-iterative scheme. High order schemes with optimal number of function evaluations per iteration are more of academic interest than of real practical utility.

Impact of computing power on practical usage of a method Computing power available to us has a significant impact on the practical utility of a method. In the current 21st century, we are having at our disposal enormous computing resources. An estimated over 90% of this power remains unutilized and thus a waste. Unlike the main frame days during the mid-twentieth century as well as even late twentieth century when many users used to use one single large computer in batches, this century as well as late 1980s have witnessed internet revolution and rapid change in computing scene. Every 18 months processor speed is doubling, every 12 months band width is doubling, and every 9 months hard disk space is doubling in our silicon technology/architecture. Today we are in teraflops (10^{12} floating point operations per second) speed and have touched petaflops (10^{15} floating-point operations per second) speed and now are heading toward exaflops (10^{18} floating-point operations). Most of the engineers, scientists, and students all over the globe have their personal laptops/desktops which are used hardly 10% time on an average. Under these circumstances, it is not much meaningful to attach too much of importance to computational complexity when it is significantly small. But when one needs to solve millions of problems such as polynomial zero-finding problem in real-time, saving a millisecond for a problem is important. Hence the computationally optimal iterative schemes (which are often not high order) that save time and produce quality roots (more accurate), has a scope in real-time/embedded computations.

Matrix eigenvalue problem versus fixed-point iterative schemes Matrix methods for matrix eigenvalue problems essentially compute all the zeros of the corresponding characteristic polynomial. Most of the matrix iterative methods produce all the zeros (eigenvalues) simultaneously. All the presented iterative schemes, except possibly deflated Newton scheme, could only find one zero at a time. To find other zeros, we need to deflate the polynomial by factoring and then using a scheme. Or we try a different initial approximation for the original (undeflated) polynomial to get another zero this may not always succeed, however. The deflated Newton scheme can find all the repeated zeros by successive differentiation/deflation with an appropriate initial approximation while other non-repeated zeros are obtained in the same way as the Newton method does.

Zero-clusters and multiple zeros: Oscillation around a zero Any polynomial with zero-clusters are considered ill-posed. To meaningfully obtain all the zeros in a cluster so that the zeros are reasonably known with an acceptable accuracy is always a problem to a varying degree with all the methods – all fixed-point iterative methods as well as all matrix eigenvalue finding methods – so far devised. The best approach is, however, the exhaustive search algorithm for both zero-cluster and multiple-zero problems so far as the quality of solution is concerned [24].

Convergence of iterations For polynomials the foregoing iteration schemes starting from Newton scheme upwards (higher order convergence) will always converge for any (complex) initial approximation far or near the zero assuming that (i) the precision is sufficiently large, (ii) zeros are distinct, and not clustered or repeated, and (iii) the number of iterations allowed is sufficiently large.

Computational implication of order of convergence If the order of convergence is k and the zero is correct up to d decimal digits then, roughly speaking, at the $(k+1)^{\text{st}}$ iteration the zero will be correct up to kd decimal digits. This is (more or less) true when the zero is neither in a cluster nor a multiple zero. In zero-cluster and multiple-zero problems, the actual convergence will be too slow and then around the zero there will be oscillations due to computational errors caused by the finite precision of the computer. The higher order schemes will perform usually worse than the Newton scheme for these problems. In fact, numerical instability is more pronounced in higher order schemes.

Optimal order of convergence It is always possible to develop any high-order method. The question then crops up: Is it that the higher the order of method is, better it is for computations? The answer is obviously no. Otherwise, the Newton method which is a second order one and other variations of Newton method would have been possibly forgotten long back. The higher the order of the method is beyond 2 or 3, more is usually the computation. Is the cost of computation or, in other words, computational complexity less for higher order schemes than lower order schemes? Is there an optimal order? The answer to the first question is not necessarily, in general while the answer to our second question is yes, in general. Through numerical experiments or/and through theoretical computational complexity, we have seen that the order 2 or 3 are usually optimal (computationally). A rigorous mathematical proof for any nonlinear equation including those with root-clusters and also those with multiple roots is yet an open problem.

References

1. S. Weerakoon and T.G.I. Fernando, *A variant of Newtons method for accelerated third-order convergence*, Appl. Math. Lett. 13 (8), 2000, 87-93.
2. M. Frontini and E. Sormani, *Some variant of Newtons method with third-order convergence*, Appl. Math. Comput. 140, 2003, 419-426.
3. H.H.H. Homeier, *On Newton-type methods with cubic convergence*, I. Comput. Appl. Math., 176, 2005, 425-432.
4. H.H.H. Homeier, *A modified Newton method for root-finding with cubic convergence*, J. Comput. Appl. Math., 157, 2003, 227-230.
5. H.T. Kung and J.F. Traub, *Optimal order of one-point and multi-point iteration*, J. Assoc. Comput. Math., 21, 1974, 634-651.
6. P. Jarratt, *Some fourth order multi-point iterative methods for solving equations*, Math. Comp., 20 (5), 1966, 434-437.
7. C. Chun, *Some fourth-order iterative methods for solving nonlinear equations*, Appl. Math. Comput., 195 (2), 2008, 454-459.

8. A.K. Maheshwari, *A fourth-order iterative method for solving nonlinear equations*, Appl. Math. Comput., 211 (2), 2009, 383-391.
9. J.F. Traub, *Iterative Methods for the Solution of Equations*, Prentice-Hall, New York, 1964.
10. A.M. Ostrowski, *Solution of Equations and Systems of Equations*, Academic Press, New York, 1966.
11. J. Kou, Y. Li and X. Wang, *A composite fourth-order iterative method*, Appl. Math. Comput., 184, 2007, 471-475.
12. R. King, *A family of fourth order methods for nonlinear equations*, SIAM J. Numer. Anal., 10, 1973, 876-879.
13. M.S. Petkovic, *On a general class of multipoint root-finding methods of high computational efficiency*, SIAM J. Numer. Anal., 47 (6), 2010, 4402-4414.
14. S.K. Khattri, *Altered Jacobian Newton iterative method for nonlinear elliptic problems*, IAENG Int. J. Appl. Math., 38, 2008.
15. V. Kanwar and S.K. Tomar, *Modified families of Newton, Halley and Chebyshev methods*, Appl. Math. Comput. 192 (1), 2007, 20-26.
16. X. Li, C. Mu, J. Ma, and C. Wang, *Sixteenth order method for nonlinear equations*, Appl. Math. Comput. 215 (10), 2009, 3754-3758.
17. H. Ren, Q. Wu, and W. Bi, *New variants of Jarratts method with sixth-order convergence*, Numer. Algorithms, 52, 2009, 585-603.
18. X. Wang, J. Kou, and Y. Li, *A variant of Jarratts method with sixth-order convergence*, Appl. Math. Comput., 190, 2008, 14-19.
19. J.R. Sharma and R.K. Guha, *A family of modified Ostrowski methods with accelerated sixth order convergence*, Appl. Math. Comput., 190, 2007, 111-115.
20. B. Neta, A sixth-order family of methods for nonlinear equations, Int. J. Comput. Math., 7, 1979, 157-161.
21. C. Chun and Y. Ham, *Some sixth-order variants of Ostrowski root-finding methods*, Appl. Math. Comput., 193, 2003, 389-394.
22. V. Lahshmikantham and S.K. Sen, *Computational Error and Complexity in Science and Engineering*, Elsevier, Amsterdam, 2005.
23. E.V. Krishnamurthy and S.K. Sen, *Numerical Algorithms: Computations in Science and Engineering*, Affiliated East-West Press, New Delhi, 2007.
24. S.K. Sen and R.P. Agarwal, *Zero-clusters of polynomials: Best approach in supercomputing era*, Appl. Math. Comput., 215, 2010, 4080-4093.
25. S.K. Sen and S.S. Prabhu, *Optimal iterative schemes for computing Moore-Penrose matrix inverse*, Int. J. Systems Sci., 8, 1976, 748-753.
26. J. Kuo, X. Wang and Y. Li, *Some eighth-order root finding three step methods*, Communications in Nonlinear Science and Numerical Simulation, 15, 2010, 536-544.

Syamal K. Sen First Author received Ph.D. from Indian Institute of Science (I.I.Sc.), Bangalore in 1973. Prior to joining Florida Institute of Technology, Melbourne, Florida as a professor, he was a professor in I.I.Sc. His research activities include applicable computational mathematics and operations research.

Department of Mathematical Sciences, Florida Institute of Technology, 150 W. University Boulevard Melbourne, Florida 32901, USA.
e-mail: sksen@fit.edu

Ravi P. Agarwal received Ph.D. from Indian Institute of Technology, Madras in 1973. He is currently a professor and the chairman of Mathematics at Texas A&M University-Kingsville. His research interest includes computational mathematics besides differential equations.

Department of Mathematics, Texas A & M University-Kingsville, 700 University Boulevard Kingsville, TX 78363-8202, USA.
e-mail: `Agarwal@tamuk.edu`

Sanjay K. Khattri received M.S. from Texas A & M University College Station and Ph.D at the University of Bergen. Since 2006 he has been working as Associate Professor at the Stord-Haugesund University College. His research interests include scientific computing and numerical analysis.

Department of Engineering, Stord-Haugesund University College, Haugesund, 5528, Norway.
e-mail: `sanjay.khattri@hsh.no`