Computational Implementation of the Multivariate Halley Method for Solving Nonlinear Systems of Equations

ANNIE A. M. CUYT
University of Antwerp
and
L. B. RALL
University of Wisconsin—Madison

Cubically convergent iterative methods for the solution of nonlinear systems of equations, such as the multivariate Halley method, require first and second partial derivatives of the functions comprising the system. Automatic differentiation is used to automate the Halley method, using the data type HESSIAN and routines for the required operators and functions. A Pascal-SC program is given, which implements this method in a single-step iteration mode. The program is applied to two nonlinear systems, and the results are compared with Newton's method.

Categories and Subject Descriptors: G.1.5 [**Numerical Analysis**]: Roots of Nonlinear Equations—*iterative methods, systems of equations*; G.1.m [**Numerical Analysis**]: Miscellaneous

General Terms: Languages

Additional Key Words and Phrases: Automatic differentiation, cubic convergence, Halley method, Pascal-SC, type HESSIAN

1. NONLINEAR SYSTEMS OF EQUATIONS

One of the central problems of scientific computation is the efficient numerical solution of systems of n equations

$$f_i(x_1, x_2, \ldots, x_n) = 0, \qquad i = 1, 2, \ldots, n, \tag{1.1}$$

in n unknowns $x_1, x_2, \ldots, x_n$. This is a special case of the *operator equation*

$$f(x) = 0, \tag{1.2}$$

in which $f: D \subset R^n \to R^n$, $0 \in R^n$ denotes the zero vector $0 = (0, 0, \ldots, 0)$, and the point $x \in R^n$ is sought. If f is an *affine* operator,

$$f(x) = Ax + b, \tag{1.3}$$

Research was sponsored by the Belgian National fund for Scientific Research (NFWO), and in part by the U.S. Army under Contract No. DAAG29-80-C-0041.
Authors' addresses: A. A. M. Cuyt, Department of Mathematics, University of Antwerp UIA, B-2610 Wilrijk, Belgium; L. B. Rall, Mathematics Research Center, University of Wisconsin—Madison, Madison WI 53706.

with the matrix $A = (a_{ij})$ and the vector $b = (b_1, b_2, \ldots, b_n)$ given, then the system (1.1) is said to be *linear*. This important special case is now fairly well understood in both theory and computational practice. Otherwise, (1.1) is a nonlinear system, and the situation is quite different from the linear case with respect to both theory and practice. Most of the methods for nonlinear systems investigated to date [14, 15] involve some form of iteration, and many also involve approximation of the nonlinear system by a linear system during the various steps of the solution process, such as in the case of Newton's methods and its many variants [14, 15]. It has been observed that some solution procedures work better than others on a given problem, so that in the absence of a clear-cut criterion for choosing the optimal method, it is advisable to have several choices available in the form of computer programs that are easy to use.

It will be assumed that the operator f corresponding to the system (1.1) has first and second Fréchet derivatives f', f'' on its domain $D \subset R^n$ [15]. In this case, the first Fréchet derivative of f at x is represented by the *Jacobian matrix*

$$f'(x) = \left(\frac{\partial f_i(x)}{\partial x_j}\right), \tag{1.4}$$

and the second by the *Hessian operator*

$$f''(x) = \left(\frac{\partial^2 f_i(x)}{\partial x_j \partial x_k}\right) \tag{1.5}$$

[15]. Necessary values of the derivatives appearing in (1.4) and (1.5) will be obtained by automatic differentiation [17], so that the user need only supply expressions or subroutines for the n functions $f_i(x_1, x_2, \ldots, x_n)$ appearing in (1.1). This avoids both the labor of providing code for derivatives and the inaccuracy of numerical differentiation. In [20], it was shown how to automate the calculation of the Jacobian matrix (1.4) needed in Newton's method by the use of type GRADIENT. Here, type HESSIAN [18] will be used to evaluate both (1.4) and (1.5), which are required for the computational implementation of a cubically convergent iterative procedure, the multivariate Halley method due to Cuyt [2, 3, 4].

2. THE MULTIVARIATE HALLEY METHOD

This method is based on the theory of abstract Padé approximants [2, 4], and conditions for its numerical stability have been given by Cuyt [3]. The abstract setting is a Banach algebra [15]: R^n with multiplication and division of vectors defined componentwise forms such a structure for the norm $\| x \| = \max_{(i)} | x_i |$, for example. Halley's method starts from an initial approximation x^0 to a solution $x = x^*$ of (1.2), and then defines the sequence $\{x^\nu\}$ of successive approximations by the following algorithm:

$$x^{\nu+1} = x^\nu + \frac{(a^\nu)^2}{a^\nu + \frac{1}{2}b^\nu}, \tag{2.1}$$

where

$$a^\nu = -f'(x^\nu)^{-1}f(x^\nu) \quad \text{(the Newton correction)},$$

and

$$b^\nu = f'(x^\nu)^{-1}f''(x^\nu)a^\nu a^\nu, \qquad \nu = 0, 1, 2, \ldots.$$

In the actual computation, the Jacobian matrix $f'(x^\nu)$ is not inverted. Rather, the linear system

$$f'(x^\nu)a^\nu = -f(x^\nu) \tag{2.2}$$

is solved for a^ν, following which the linear system

$$f'(x^\nu)b^\nu = f''(x^\nu)a^\nu a^\nu \tag{2.3}$$

is then solved for b^ν. Since the systems (2.2) and (2.3) have the same coefficient matrix, the decomposition of the Jacobian matrix $f'(x^\nu)$ used to solve (2.2) can also be used to solve (2.3), resulting in a saving of effort.

An outline of the computational effort for one step of Halley's method is thus

(1) evaluation of $f(x^\nu), f'(x^\nu), f''(x^\nu)$;
(2) solution of (2.2) for a^ν;
(3) evaluation of $f''(x^\nu)a^\nu a^\nu$;
(4) solution of (2.3) for b^ν;
(5) calculation of the *Halley correction* $c^\nu = (a^\nu)^2/(a^\nu + \frac{1}{2}b^\nu)$;
(6) addition of the Halley correction to x^ν to obtain $x^{\nu+1}$.

This sequence of operations is more elaborate than required for Newton's method [15, 20], which requires only the evaluation of $f(x^\nu), f'(x^\nu)$, the solution of (2.2) for a^ν, and finally the addition of a^ν to x^ν to obtain $x^{\nu+1}$. However, in favorable cases, the rate of convergence of Halley's method is cubic, whereas Newton's method converges quadratically. Thus the greater effort required for each step of Halley's method could be offset if fewer steps are required to obtain the accuracy desired. Two steps of Newton's method can be combined to yield a method with biquadratic convergence. However, this requires the solution of (2.2) with different coefficient matrices $f'(x^\nu)$ and $f'(x^{\nu+1})$ and right sides $-f(x^\nu)$ and $-f(x^{\nu+1})$.

For computational implementation, it is convenient to consider the steps of Halley's method to consist of a procedure for *evaluation* (Step 1), which will depend on the specific system being solved, and a procedure for *iteration* (Steps 2–6), which will have the same form for all systems. The operations of componentwise multiplication and division of vectors will also have to be provided in addition to the standard vector operations. These are simple to define, since for $a = (a_1, a_2, \ldots, a_n)$ and $b = (b_1, b_2, \ldots, b_n) \in R^n$, one has

$$\begin{aligned} ab &= (a_1b_1, a_2b_2, \ldots, a_nb_n), \\ \frac{a}{b} &= \left(\frac{a_1}{b_1}, \frac{a_2}{b_2}, \ldots, \frac{a_n}{b_n}\right), \end{aligned} \tag{2.4}$$

where division is defined in general only if $b_i \neq 0$, $i = 1, 2, \ldots, n$.

3. USE OF AUTOMATIC DIFFERENTIATION

Newton's method and the method of Section 2 are sometimes shunned because it is assumed that code has to be supplied for the derivatives of the functions f_i, or because these functions are defined by subroutines rather than simple expressions. However, since the rules for differentiation are well understood, the computer itself can produce the required code by automatic differentiation of the given expressions or subroutines [6, 17]. In the case of functions defined by expressions, programs capable of obtaining first and second derivatives have been in use for some time [5, 7, 15]. More recently, differentiation methods for subroutines have also been developed [6, 16, 17, 18]. Since the latter case is the most general, it will be examined here.

To illustrate the fundamental idea of automatic differentiation, consider a real function f of n real variables $x = (x_1, x_2, \ldots, x_n)$. The pair $(f, f') = (f(x), \nabla f(x))$ is a datum of type GRADIENT for a given value of x [18, 20], where $\nabla f(x)$ denotes the *gradient vector*

$$\nabla f(x) = \left(\frac{\partial f(x)}{\partial x_1}, \frac{\partial f(x)}{\partial x_2}, \ldots, \frac{\partial f(x)}{\partial x_n}\right). \tag{3.1}$$

Writing $F = (f, f')$ to represent an element of this new type of data, the next step is to define the corresponding arithmetic operations to implement the rules for differentiation in a computable form. For example, for $G = (g, g')$, addition and multiplication are defined by

$$\begin{aligned} F + G &= (f + g, f' + g'), \\ F*G &= (f*g, f*g' + g*f'), \end{aligned} \tag{3.2}$$

respectively. Similarly, functions such as the sine function can be represented in the form

$$\mathrm{GSIN}(F) = (\sin(f), \cos(f)*f'). \tag{3.3}$$

The independent variable x_i is represented by the GRADIENT variable $X[i] = (x_i, e_i)$, where e_i is the ith unit vector, and the evaluation of a GRADIENT expression will automatically yield both the values of the function $f(x)$ and its gradient vector $\nabla f(x)$ at the given value of x. Thus the programmer need only supply code for the evaluation of a function to get also its derivative, once the standard set of GRADIENT operators and functions [20] is available.

For the present purpose, second derivatives are needed, and so type GRADIENT is extended to type HESSIAN, a datum of which is the triple $F = (f, f', f'') = (f(x), \nabla f(x), Hf(x))$ [18], where $Hf(x)$ is the *Hessian matrix*

$$Hf(x) = \left(\frac{\partial^2 f(x)}{\partial x_j \partial x_k}\right). \tag{3.4}$$

Once again, there is no problem in the implementation of arithmetic operations and standard functions, for example,

$$\begin{aligned} F + G &= (f + g, f' + g', f'' + g''), \\ F*G &= (f*g, f*g' + g*f', f*g'' + f'*g'^T + g'*f'^T + g*f''), \end{aligned} \tag{3.5}$$

and

$$\mathrm{SIN}(F) = (\sin(f),\ \cos(f) * f',\ \cos(f) * f'' - \sin(f) * f' * f'^{T}). \tag{3.6}$$

The HESSIAN variables $X[i]$ corresponding to the independent variables x_i are $X[i] = (x_i, e_i, 0)$, where e_i is the ith unit vector, and 0 denotes the $n \times n$ zero matrix. Thus evaluation of expressions of type HESSIAN yields the value of the second derivative $f''(x)$ as well as the values of the function $f(x)$ and its first derivative $f'(x)$. Although the formulations of HESSIAN operators and standard functions are more complicated than those for type GRADIENT [20], programming them is no real challenge, and this needs to be done only one time. Once available, these subroutines shift the burden of differentiation from the programmer to the computer, which is as it should be.

In order to calculate the Jacobian matrix (1.4) and Hessian operator (1.5) of a *vector-valued* function $f(x) = (f_1(x), f_2(x), \ldots, f_n(x))$, each real-valued component function $f_i(x)$ is defined to be of type HESSIAN. In this case, the ith row of the Jacobian matrix $f'(x)$ is given by the gradient vector $\nabla f_i(x)$ of the ith component function, and the Hessian matrix $Hf_i(x)$ of the ith component function will be the ith "panel" of the Hessian operator $f''(x)$.

4. TYPE HESSIAN IN PASCAL-SC

In primitive computing languages such as FORTRAN, automatic differentiation requires interpretation of expressions [5, 7] or precompilation [6, 17]. However, in languages that permit user-defined operators, such as ALGOL 68, Ada, and Pascal-SC [1, 13], statements can be written in ordinary notation, with derivatives evaluated automatically. In order to make full use of the facilities already available in Pascal-SC for vector and matrix arithmetic [22], type HESSIAN is introduced in a way to make it consistent with the definitions of types RVECTOR and RMATRIX for n-dimensional REAL vectors and matrices, respectively. The standard declarations [22] are

```
CONST DIM = n;

  TYPE DIMTYPE = 1..DIM;
       RVECTOR = ARRAY [DIMTYPE] OF REAL;
       RMATRIX = ARRAY [DIMTYPE] OF RVECTOR;
```
(4.1)

Following these, type HESSIAN is declared by

```
TYPE HESSIAN = RECORD F: REAL; DF: RVECTOR; HF: RMATRIX END;
```
(4.2)

[19]. Thus, as the result of a subroutine for computation of $f(x)$ as the HESSIAN variable F, one has

$$\mathrm{F.F} = f(x), \qquad \mathrm{F.DF} = \nabla f(x), \qquad \mathrm{F.HF} = Hf(x). \tag{4.3}$$

A complete package of HESSIAN arithmetic operators and standard functions has been prepared in Pascal-SC [18]. It is efficient to consider variables of types INTEGER and REAL as constants for the purpose of differentiation, so a total of 22 arithmetic operators are required. If K, R, and H denote generic variables

of types INTEGER, REAL, and HESSIAN, respectively, these are

```
+ H, K + H, H + K, R + H, H + R, H + H, - H, K - H, H -
K, R - H, H - R, H - H, K * H, H * K, R * H, H * R, H * H,
K / H, H / K, R / H, H / R, H / H.
```
(4.4)

The power operator ** and various standard functions are also available for type HESSIAN [18]. Typical examples of HESSIAN operators can be found in the evaluation routine given in Appendix C.

In order to represent the vector $x = (x_1, x_2, \ldots, x_n)$ of independent variables and the vector-valued function $f(x) = (f_1(x), f_2(x), \ldots, f_n(x))$, with components of type HESSIAN, it is convenient to introduce the data type HESSVAR, defined by

```
TYPE HESSVAR = ARRAY [DIMTYPE] OF HESSIAN;
```
(4.5)

In this way, it is possible to code systems of equations (1.1) in a form that follows ordinary mathematical notation. For example, the simple system

$$\begin{aligned} e^{-x_1+x_2} - 0.1 &= 0, \\ e^{-x_1-x_2} - 0.1 &= 0, \end{aligned} \tag{4.6}$$

investigated by Cuyt and Van der Cruyssen [2, 4] requires the following HESSIAN operators and functions:

```
OPERATOR - (H: HESSIAN) RES: HESSIAN;
OPERATOR - (HA, HB: HESSIAN) RES: HESSIAN);
OPERATOR - (H: HESSIAN; R: REAL) RES: HESSIAN;
OPERATOR + (HA, HB: HESSIAN) RES: HESSIAN;
FUNCTION HEXP(H: HESSIAN): HESSIAN;
```
(4.7)

The subroutines for these have to appear in the heading of the procedure HESSEVAL(VAR X, F: HESSVAR) for the evaluation of $f(x)$ corresponding to (4.6) (see Appendix C). The evaluation of f and its first and second derivatives is then carried out by the statements

```
F[1] := HEXP(-X[1] + X[2]) - 0.1;
F[2] := HEXP(-X[1] - X[2]) - 0.1;
```
(4.8)

which follow the form of (4.6) exactly.

Similarly, HESSEVAL for the function $f(x)$ corresponding to the system

$$\begin{aligned} 16x_1^4 + 16x_2^4 + x_3^4 - 16 &= 0, \\ x_1^2 + x_2^2 + x_3^2 - 3 &= 0, \\ x_1^3 - x_2 &= 0, \end{aligned} \tag{4.9}$$

considered in [20], requires the following operators:

```
OPERATOR * (K: INTEGER; H: HESSIAN) RES: HESSIAN;
OPERATOR ** (R: REAL; K: INTEGER) RES: REAL;
OPERATOR ** (H: HESSIAN; K: INTEGER) RES: HESSIAN;
OPERATOR + (HA, HB: HESSIAN) RES: HESSIAN;
OPERATOR - (H: HESSIAN; K: INTEGER) RES: HESSIAN;
OPERATOR - (HA, HB: HESSIAN) RES: HESSIAN;
```
(4.10)

after which the evaluation of f and its derivatives takes place by means of the statements

```
F[1] := 16 * (X[1] ** 4) + 16 * (X[2] ** 4)
        + X[3] ** 4 - 16;                                    (4.11)
F[2] := X[1] ** 2 + X[2] ** 2 + X[3] ** 2 - 3;
F[3] := X[1] ** 3 - X[2];
```

which resemble (4.9). Parentheses are necessary in the first statement of (4.11) because * and ** have the same priority in Pascal-SC [13]. The coding for the system (4.9) is analogous to the statements given in Appendix C for the system (4.6).

Step 1 of Halley's method as described in Section 2 is thus carried out simply by the evaluation of the function $f(x)$ as of type HESSVAR, where the independent variable x also has the same type, with current value X[i].F = x_i, $i = 1, \ldots, n$. The value of the transformation F is given by the RVECTOR B with components B[i] = F[i].F, and the Jacobian matrix of F is the RMATRIX JACF with rows JACF [i] = F[i].DF, $i = 1, \ldots, n$. As will be shown below, the panels F[i].HF of the Hessian operator of F can be used directly in the computation, and so it is not necessary to construct the operator itself.

5. SOLUTION OF LINEAR SYSTEMS OF EQUATIONS IN PASCAL-SC

Steps 2 and 4 of Halley's method require the solution of linear systems of equations, an operation that is also required by Newton's method. Pascal-SC provides the basic procedure LGLP for this purpose [22], which is declared by

```
PROCEDURE LGLP(DIM, AKDIM: INTEGER; VAR A: RMATRIX; VAR B:
                RVECTOR;
                VAR Y: IVECTOR);
```

The meaning of DIM is the same as before; if one wishes to solve a smaller system, the parameter AKDIM can be used to set the number of rows and columns of the coefficient matrix A and components of the right side vector B that enter into the computation. More significantly, instead of returning a floating-point RVECTOR x as an approximate solution of the linear system

$$Ax = B, \tag{5.1}$$

LGLP returns an *interval vector* (IVECTOR) Y, which, if proper, is *guaranteed* to contain the *exact* solution x of (5.1) [10, 21, 22]. Furthermore, successful completion of LGLP guarantees that the floating-point matrix A is nonsingular [21, 22]. If A is singular or extremely badly conditioned, LGLP will return an *improper* interval vector Y with all components equal to the improper interval $[+1, -1]$ [22].

In actual practice, LGLP is observed to be highly accurate, even for matrices that are known to be poorly conditioned [21]. In any case, if

$$Y = ([a_1, b_1], [a_2, b_2], \ldots, [a_n, b_n]) \tag{5.2}$$

is proper ($a_i \leq b_i$ for $i = 1, \ldots, n$), one has

$$a_i \leq x_i \leq b_i, \qquad i = 1, 2, \ldots, n, \tag{5.3}$$

for all components x_i of the *exact* solution $x = (x_1, x_2, \ldots, x_n)$ of (5.1), from which an approximate solution with known error bounds can be constructed [19]. This kind of guaranteed accuracy is possible because Pascal-SC completely supports interval arithmetic [8, 11, 12, 22] as well as accurate floating-point arithmetic.

Since Step 4 of Halley's method requires the solution of another linear system of equations with the same coefficient matrix as in Step 2, it is more efficient to use the decomposition of the coefficient matrix and other auxiliary results from the first system to solve the second than starting anew. For this purpose, the Pascal-SC procedure LGLPR is provided [22]:

```
PROCEDURE LGLPR (DIM, AKDIM: INTEGER; VAR A: RMATRIX; VAR B:
                 RVECTOR
                 NRS: BOOLEAN; VAR R: RMATRIX; VAR MB: IMA-
                 TRIX; VAR Y: INVECTOR);
  EXTERNAL 522;
```

For the first system to be solved, one sets NRS = FALSE, and then subsequently NRS = TRUE for each new right side. The results from the first solution needed later are stored as the real matrix R and interval matrix MB.

After solution of the linear systems (2.2) or (2.3), the interval vector Y has to be checked and converted to a real vector, before the computation can be continued. This is done by the function MID given in Appendix B.

6. COMPUTATION WITH BILINEAR OPERATORS

In Step 3 of Halley's method, the right side $f''(x^\nu)a^\nu a^\nu$ of the system (2.3) is constructed by operating with the bilinear operator $f''(x^\nu)$ twice on the vector a^ν. The first operation yields a matrix, and the second a vector [15]. The way in which HESSIAN variables are defined makes it easy to implement these operations in terms of the vector and matrix operators available in Pascal-SC [22]. In general, a bilinear operator

$$B = (b_{ijk}) \tag{6.1}$$

will be considered to be composed of n matrices

$$B_1 = (b_{1jk}), B_2 = (B_{2jk}), \ldots, B_n = (B_{njk}), \tag{6.2}$$

called *i-panels*, or simply *panels* of B. For a vector $x \in R^n$, the matrix

$$A = (a_{ij}) = Bx = \left(\sum_{k=1}^{n} b_{ijk} x_k \right) \tag{6.3}$$

will have rows A_i^{T} given by the matrix–vector product

$$A_i = B_i x, \qquad i = 1, 2, \ldots, n. \tag{6.4}$$

Once the matrix L is formed by computing the vectors (4.4), then the vector

$$y = Ax = Bxx = \left(\sum_{j=1}^{n} \sum_{k=1}^{n} b_{ijk} x_k x_j \right) \tag{6.5}$$

is obtained by a single additional matrix-vector multiplication. In the case $B = f''(x)$, one has $B_i = Hf_i(x^\nu)$, so that

$$A_i = Hf_i(x^\nu)a^\nu, \qquad i = 1, 2, \ldots, n, \tag{6.6}$$

and thus

$$f''(x^\nu)a^\nu a^\nu = Aa^\nu, \tag{6.7}$$

so the required vector is obtained by a total of $(n + 1)$ matrix–vector multiplications. In Pascal-SC, matrices are stored rowwise, and so no transposition is required when forming the matrix A from the vectors A_i in (4.6) [22]. It is also important to note that in Pascal-SC, scalar products of vectors and also matrix–vector products are computed with the minimum possible round-off error; that is, their values are obtained to the closest floating-point numbers [9, 10, 22]. This accuracy is far greater than can be obtained by the usual method of simulation of these operations by sums of products of floating-point numbers [10]. The calculation of the rows of A by (6.6) and the vector (6.7) require only the Pascal-SC operator

```
OPERATOR * (A: RMATRIX; B: RVECTOR) RES: RVECTOR;
```

for matrix-by-vector multiplication [22].

7. AN ITERATION PROCEDURE FOR HALLEY'S METHOD

In order to write a procedure for one step of Halley's method (2.1), all that is needed in addition to the above is the calculation of the Halley correction in Step 4 and the addition of this vector to the initial vector (Step 5). Calculation of the Halley correction requires operators for the componentwise multiplication and division of vectors. Suitable formulations of these are as follows:

```
OPERATOR * (VA, VB: RVECTOR) RES: RVECTOR;
  VAR I: DIMTYPE; U: RVECTOR;
  BEGIN
    FOR I := 1 TO DIM DO
    U[I] := VA[I] * VB[I];
    RES := U
  END;
```

(7.1)

and

```
OPERATOR / (VA, VB: RVECTOR) RES: RVECTOR;
  VAR I: DIMTYPE; U: RVECTOR;
  BEGIN
    FOR I := 1 TO DIM DO
    IF (VA[I] = 0) and (VB[I] = 0) THEN U[I] := 0
    ELSE U[I] := VA[I]/VB[I];
    RES := U
  END;
```

(7.2)

In (6.2), the indeterminant form 0/0 is assigned the value 0, by continuity of the Halley approximation. The calculation of the Halley correction also requires the standard Pascal-SC operators

```
OPERATOR * (A: REAL; B: RVECTOR) RES: RVECTOR;
OPERATOR + (A, B: RVECTOR) RES: RVECTOR;
```

for multiplication of vectors by real numbers, and addition of vectors [22]. With these and the componentwise operators (7.1) and (7.2), the Halley correction can be evaluated by a statement of the form

```
CN := (AN * AN)/(AN + 0.5 * BN);                    (7.3)
```

where, of course, AN = a^{ν}, BN = b^{ν}, CN = c^{ν}. The current value of X is then updated by the statement

```
FOR I := 1 TO DIM DO X[I]·F := X[I]·F + CN[I];      (7.4)
```

The steps required for a Halley iteration are collected in the form of the Pascal-SC procedure given in Appendix B. Together with the procedure

```
HESSEVAL(VAR X, F: HESSVAR);
```

for the evaluation of the function $f(x)$ corresponding to the system of equations (1.1), a program for the iterative solution of (1.1) by Halley's method can be constructed easily. A simple program of this type is given in Appendix A, which presents the results of each iteration to the user, who can then decide whether to iterate further, stop the iteration, or start over with another initial vector.

In the program of Appendix A, the compiler directive

```
$USES LGL, DIM = #;
```

brings in the necessary type declarations, sets the constant DIM to the dimension of the system specified by the user [13], and refers the compiler to the external library LGLLIB containing the linear equation-solving and matrix inversion routines [13]. The $INCLUDE directives bring in the source code for the method being used and for evaluation of the systems, which are in the external files HALLEY.SRC and HESSEVAL.SRC, respectively [13]. In general, the programmer need only supply the file HESSEVAL.SRC for evaluation of the system being solved, and modify the source code of the program ITERATE to set the dimension and give the name of the method being used in the heading of the output. The only place where modifications are necessary is indicated by "#" in the source code file ITERATE.SRC.

8. NUMERICAL RESULTS

The method described in this paper was applied to the systems (4.8) and (4.9), and the results were compared with those obtained by Newton's method [20]. The initial approximations for the system (4.8) were

$$x_1 = 4.3, \qquad x_2 = 2.0, \tag{8.1}$$

[3, 4], and the initial approximations for (4.9) were

$$x_1 = 1.0, \qquad x_2 = 1.0, \qquad x_3 = 1.0, \tag{8.2}$$

[20]. For the system (4.8), Newton's method requires 55 iterations to reduce the residual to 0 to 12 decimal places, whereas Halley's methods required only five iterations. On the other hand, for (4.9), the corresponding numbers were eight iterations for Newton's method, and five for Halley's method, a result that is more favorable to Newton's method. The results are given in detail in Appendix D.

The methodology presented in this paper can also be used to automate other higher order methods for the solution of systems of equations, such as Chebyshev's method and the method of tangent hyperbolas [14, 15].

APPENDIX A. A SIMPLE PROGRAM TO DRIVE HIGHER ORDER ITERATIVE METHODS

```
PROGRAM ITERATE(INPUT,OUTPUT);
$USES LGL,DIM=#;              (* DIMENSION OF SYSTEM *)

    TYPE HESSIAN = RECORD  F: REAL;      (* FUNCTION VALUE *)
                           DF: RVECTOR;  (* GRADIENT VECTOR *)
                           HF: RMATRIX   (* HESSIAN MATRIX *)
                   END;
         HESSVAR = ARRAY [DIMTYPE] OF HESSIAN;

     VAR X,F: HESSVAR;    (* INDEPENDENT AND DEPENDENT VARIABLES *)
         I,J: DIMTYPE;    (* INDEX VARIABLES *)
           C: CHAR;       (* CONTROL CHARACTER *)
           K: INTEGER;    (* ITERATION COUNTER *)

FUNCTION MRNULL: RMATRIX;  (* RETURNS THE ZERO MATRIX *)
 VAR I,J: DIMTYPE;
   C: RMATRIX;
 BEGIN
  FOR I:=1 TO DIM DO
   FOR J:=1 TO DIM DO
    C[I,J] := 0;
  MRNULL := C
 END;             (* FUNCTION MRNULL *)

$INCLUDE HESSEVAL.SRC;    (* SOURCE CODE FOR PROCEDURE HESSEVAL *)
$INCLUDE HALLEY.SRC;      (* SOURCE CODE FOR ITERATION STEP *)

BEGIN  (* PROGRAM ITERATE *)

    WRITELN;  (* SIGN-ON MESSAGE *)
    WRITELN('HALLEY''S METHOD FOR SOLUTION OF SYSTEMS OF EQUATIONS');
    WRITELN;

    FOR I:=1 TO DIM DO    (* INITIALIZATION OF INDEPENDENT VARIABLES *)
        BEGIN

            X[I].HF:=MRNULL;  (* SETS HESSIANS TO ZERO MATRIX *)
            FOR J:=1 TO DIM DO X[I].DF[J]:=0;
            X[I].DF[I]:=1     (* SETS GRADIENTS TO UNIT VECTORS *)

        END;              (* INITIALIZATION OF INDEPENDENT VARIABLES *)

    C:='R';WHILE C = 'R' DO
```

```
    BEGIN  (* SYSTEM SOLUTION *)

        WRITELN;WRITELN('ENTER VALUES OF INDEPENDENT VARIABLES');
        FOR I:=1 TO DIM DO READ(X[I].F);
        WRITELN;WRITELN('INITIAL VALUES ARE');K:=0;
        C:='I';WHILE C = 'I' DO

            BEGIN  (* ITERATION *)

            HESSEVAL(X,F);    (* EVALUATE SYSTEM AT CURRENT VALUE OF X *)
            WRITELN;
            FOR I:=1 TO DIM DO  (* PRINT VALUES OF X,F *)
            WRITELN('X[',I:2,'] = ',X[I].F,'  F[',I:2,'] = ',F[I].F);
            WRITELN;
            WRITELN('ENTER "I" TO ITERATE, "R" TO RESTART, "Q" TO QUIT');
            READ(C,C);        (* ITERATION CONTROL *)
            IF C = 'I' THEN

                BEGIN
                    HALLEY(X,F);    (* ITERATION STEP *)
                    K:=K+1;         (* INCREASE ITERATION COUNTER *)
                    WRITELN;WRITELN('RESULTS OF ITERATION ',K:3);
                END;                (* ITERATION STEP *)

            END;    (* ITERATION *)

    END     (* SYSTEM SOLUTION *)

END.    (* PROGRAM ITERATE *)
```

APPENDIX B. SOURCE CODE FOR THE MULTIVARIATE HALLEY METHOD

```
PROCEDURE HALLEY(VAR X,F: HESSVAR);

    (* HALLEY METHOD *)

   VAR      I: DIMTYPE;
          JACF: RMATRIX;    (* THE JACOBIAN MATRIX *)
            AN: RVECTOR;    (* THE NEWTON CORRECTION *)
            BN: RVECTOR;
            CN: RVECTOR;    (* THE HALLEY CORRECTION *)
             A: RMATRIX;
             B: RVECTOR;    (* USED BY LGLPR *)
             R: RMATRIX;    (*       "       *)
             Y: IVECTOR;    (*       "       *)
            MB: IMATRIX;    (*       "       *)

    (* STANDARD MATRIX AND VECTOR OPERATORS *)

OPERATOR + (A,B: RVECTOR) RES: RVECTOR;
 VAR I: DIMTYPE;
BEGIN FOR I:=1 TO DIM DO A[I] := A[I]+B[I];
  RES := A
END;
```

```
OPERATOR * (A: REAL; B: RVECTOR) RES: RVECTOR;
 VAR I: DIMTYPE;
 BEGIN FOR I:=1 TO DIM DO B[I] := A*B[I];
   RES := B
 END;

OPERATOR * (A: RMATRIX; B: RVECTOR) RES: RVECTOR;
 VAR I: DIMTYPE;
   BVAR: RVECTOR;
 BEGIN
  BVAR := B;
  FOR I:=1 TO DIM DO
  B[I] := SCALP (A[I],BVAR,0);
  RES := B
 END;

    (* END OF STANDARD MATRIX AND VECTOR OPERATORS *)

    (* OPERATORS FOR COMPONENTWISE MULTIPLICATION AND DIVISION OF VECTORS *)

OPERATOR * (A,B: RVECTOR) RES: RVECTOR;
   VAR I: DIMTYPE;C: RVECTOR;
   BEGIN FOR I:=1 TO DIM DO C[I]:=A[I]*B[I];
         RES:=C
   END;
OPERATOR / (A,B: RVECTOR) RES: RVECTOR;
   VAR I: DIMTYPE;C: RVECTOR;
   BEGIN FOR I:= 1 TO DIM DO
         IF (A[I]=0) AND (B[I]=0) THEN C[I]:=0
         ELSE C[I]:=A[I]/B[I];
         RES:=C
   END;
    (* FUNCTION TO CONVERT PROPER INTERVAL VECTOR TO REAL VECTOR *)

FUNCTION MID(VAR Y: IVECTOR): RVECTOR;
   VAR I: DIMTYPE;C: RVECTOR;
   BEGIN
        IF Y[1].INF <= Y[1].SUP THEN  (* Y IS PROPER *)
        FOR I:=1 TO DIM DO C[I]:=Y[I].INF+(Y[I].SUP-Y[I].INF)/2
        ELSE                          (* Y IS IMPROPER *)
        BEGIN  (* SEND ERROR MESSAGE AND RETURN TO OPERATING SYSTEM *)
            WRITELN('JACOBIAN MATRIX IS SINGULAR OR BADLY CONDITIONED');
            FOR I:=1 TO DIM DO Y[I].SUP:=Y[I].INF;  (* RESET Y *)
            SVR(0)       (* RETURN TO O/S *)
        END;
        MID:=C
   END;

BEGIN  (* HALLEY ITERATION *)

    (* CALCULATE JACOBIAN MATRIX AND RIGHT SIDE OF (2.2) *)

    FOR I:=1 TO DIM DO
    BEGIN
        JACF[I] := F[I].DF;     (* JACOBIAN MATRIX *)
        B[I] := -F[I].F         (* RIGHT HAND SIDE *)
    END;
```

```
    (* SOLVE FOR NEWTON CORRECTION *)

    LGLPR(DIM,DIM,JACF,B,FALSE,R,MB,Y);
    AN := MID(Y);                (* NEWTON CORRECTION *)

    (* CALCULATE RIGHT SIDE OF (2.3) AND SOLVE FOR BN *)

    FOR I:=1 TO DIM DO A[I] := F[I].HF*AN;
    B:=A*AN;
    LGLPR(DIM,DIM,JACF,B,TRUE,R,MB,Y);
    BN:=MID(Y);

    (* CALCULATE HALLEY CORRECTION *)

    CN := (AN*AN)/(AN + 0.5*BN);     (* HALLEY CORRECTION *)

    FOR I:=1 TO DIM DO X[I].F := X[I].F + CN[I];     (* UPDATE VALUES OF X *)

END;      (* HALLEY ITERATION *)
```

APPENDIX C. SOURCE CODE FOR HESSIAN EVALUATION OF THE SYSTEM (4.8)

```
   PROCEDURE HESSEVAL(VAR X,F: HESSVAR);

        (* HESSIAN OPERATORS AND FUNCTIONS FOR SYSTEM EVALUATION *)

   OPERATOR + (HA,HB: HESSIAN) RES: HESSIAN;      (* H + H *)
      VAR I,J: DIMTYPE;U: HESSIAN;
      BEGIN U.F:=HA.F+HB.F;FOR I:=1 TO DIM DO
        BEGIN U.DF[I]:=HA.DF[I]+HB.DF[I];
              FOR J:=1 TO DIM DO
              U.HF[I,J]:=HA.HF[I,J]+HB.HF[I,J]
        END;
            RES:=U
      END;

   OPERATOR - (H: HESSIAN) RES: HESSIAN;          (* -H *)
      VAR I,J: DIMTYPE;U: HESSIAN;
      BEGIN U.F:=-H.F;FOR I:=1 TO DIM DO
           BEGIN U.DF[I]:=-H.DF[I];
                 FOR J:=1 TO DIM DO
                 U.HF[I,J]:=-H.HF[I,J]
           END;
        RES:=U
      END;

   OPERATOR - (H: HESSIAN;R: REAL) RES: HESSIAN; (* H - R *)
      VAR U: HESSIAN;
      BEGIN U.F:=H.F-R;U.DF:=H.DF;U.HF:=H.HF;
            RES:=U
      END;
```

```
OPERATOR - (HA,HB: HESSIAN) RES: HESSIAN;      (* H - H *)
   VAR I,J: DIMTYPE;U: HESSIAN;
   BEGIN U.F:=HA.F-HB.F;FOR I:=1 TO DIM DO
        BEGIN U.DF[I]:=HA.DF[I]-HB.DF[I];
              FOR J:=1 TO DIM DO
              U.HF[I,J]:=HA.HF[I,J]-HB.HF[I,J]
        END;
     RES:=U
   END;

FUNCTION HEXP(H: HESSIAN): HESSIAN;            (* HEXP *)
   VAR I,J: DIMTYPE;U: HESSIAN;
   BEGIN U.F:=EXP(H.F);
     FOR I:=1 TO DIM DO
        BEGIN U.DF[I]:=U.F*H.DF[I];  (* I LOOP *)
          FOR J:=1 TO I DO
             BEGIN U.HF[I,J]:=U.F*H.HF[I,J]+U.DF[I]*H.DF[J];
                   IF I<>J THEN U.HF[J,I]:=U.HF[I,J]
             END;
        END;                              (* I LOOP *)
     HEXP:=U
   END;     (* FUNCTION HEXP *)
       (* END OF HESSIAN OPERATORS AND FUNCTIONS *)

   BEGIN    (* HESSEVAL *)

       (* DEFINITIONS OF FUNCTIONS IN SYSTEM *)

           F[1] := HEXP(-X[1] + X[2]) - 0.1;
           F[2] := HEXP(-X[1] - X[2]) - 0.1;

       (* END OF DEFINITIONS OF SYSTEM FUNCTIONS *)

   END;     (* PROCEDURE HESSEVAL *)
```

APPENDIX D. NUMERICAL RESULTS

D.1 Halley's Method for the System (4.8)

```
INITIAL VALUES ARE

X[ 1] =  4.30000000000E+00  F[ 1] =  2.58843723000E-04
X[ 2] =  2.00000000000E+00  F[ 2] = -9.81636952230E-02

RESULTS OF ITERATION   1

X[ 1] =  3.33615528246E+00  F[ 1] =  2.40511813000E-04
X[ 2] =  1.03597241993E+00  F[ 2] = -8.73756488903E-02

RESULTS OF ITERATION   2

X[ 1] =  2.56081800937E+00  F[ 1] =  1.44792584000E-04
X[ 2] =  2.59679794981E-01  F[ 2] = -4.04237220044E-02

RESULTS OF ITERATION   3

X[ 1] =  2.30817563469E+00  F[ 1] =  9.32479500000E-06
X[ 2] =  5.68378530500E-03  F[ 2] = -1.12110099570E-03
```

```
RESULTS OF ITERATION   4

X[ 1] =  2.30258515118E+00  F[ 1] =  3.02000000000E-10
X[ 2] =  6.12055700000E-08  F[ 2] = -1.19396000000E-08

RESULTS OF ITERATION   5

X[ 1] =  2.30258509299E+00  F[ 1] =  0.00000000000E+00
X[ 2] = -2.43356140000E-12  F[ 2] =  0.00000000000E+00
```

D.2. Halley's Method for the System (4.11)

```
INITIAL VALUES ARE

X[ 1] =  1.00000000000E+00  F[ 1] =  1.70000000000E+01
X[ 2] =  1.00000000000E+00  F[ 2] =  0.00000000000E+00
X[ 3] =  1.00000000000E+00  F[ 3] =  0.00000000000E+00

RESULTS OF ITERATION   1

X[ 1] =  8.91118701964E-01  F[ 1] =  9.37521623100E-01
X[ 2] =  7.05429341548E-01  F[ 2] = -9.44922445000E-03
X[ 3] =  1.30339083879E+00  F[ 3] =  2.20137281800E-03

RESULTS OF ITERATION   2

X[ 1] =  8.77982528233E-01  F[ 1] =  1.03685690000E-03
X[ 2] =  6.76786689302E-01  F[ 2] = -9.09324000000E-06
X[ 3] =  1.33082582033E+00  F[ 3] =  9.05738500000E-06

RESULTS OF ITERATION   3

X[ 1] =  8.77965760274E-01  F[ 1] =  0.00000000000E+00
X[ 2] =  6.76756970516E-01  F[ 2] =  0.00000000000E+00
X[ 3] =  1.33085541162E+00  F[ 3] =  2.00000000000E-12

RESULTS OF ITERATION   4

X[ 1] =  8.77965760274E-01  F[ 1] =  0.00000000000E+00
X[ 2] =  6.76756970517E-01  F[ 2] =  0.00000000000E+00
X[ 3] =  1.33085541162E+00  F[ 3] =  1.00000000000E-12

RESULTS OF ITERATION   5

X[ 1] =  8.77965760274E-01  F[ 1] =  0.00000000000E+00
X[ 2] =  6.76756970518E-01  F[ 2] =  0.00000000000E+00
X[ 3] =  1.33085541162E+00  F[ 3] =  0.00000000000E+00
```

REFERENCES

1. BOHLENDER, G., GRUNER, K., KAUCHER, E., KLATTE, R., KRAMER, W., KULISCH, U. W., RUMP, S. M., ULLRICH, C. WOLFF VON GUDENBERG, J., AND MIRANKER, W. L. PASCAL-SC: A PASCAL for contemporary scientific computation. Res. Rep. RC 9009, IBM Thomas J. Watson Research Center, Yorktown Heights, N.Y., 1981.
2. CUYT, A. A. M. Abstract Padé approximants for operators: Theory and applications. Lecture Notes in Mathematics, vol 1065, Springer-Verlag, New York, 1984.
3. CUYT, A. A. M. Numerical stability of the Halley-iteration for the solution of a system of nonlinear equations. *Math. Comput. 38* (1982), 171–179.

4. CUYT, A. A. M., AND VAN DER CRUYSSEN, P. Abstract Padé approximants for the solution of a system of nonlinear equations. Rep. 80-17, University of Antwerp UIA, Antwerp, Belgium, 1980.
5. GRAY, J. H., AND RALL, L. B. NEWTON: A general purpose program for solving nonlinear systems. In *Proceedings of the 1967 Army Numerical Analysis Conference.* U. S. Army Research Office, Durham, N.C., 1967, pp. 11–59.
6. KEDEM, G. Automatic differentiation of computer programs. *ACM Trans. Math. Softw. 6*, 2 (June 1980), 150–165.
7. KUBA, D., AND RALL, L. B. A UNIVAC 1108 program for obtaining rigorous error estimates for approximate solutions of systems of equations. Tech. Summary Rep. 1168, Mathematics Research Center, University of Wisconsin—Madison, 1972.
8. KULISCH, U. A new arithmetic for scientific computation. In *A New Approach to Scientific Computation*, U. Kulisch and W. L. Miranker, Eds. Academic Press, New York, 1983, pp. 1–26.
9. KULISCH, U., AND MIRANKER, W. L. *Computer Arithmetic in Theory and Practice.* Academic Press, New York, 1981.
10. KULISCH, U., AND MIRANKER, W. L., Eds. *A New Approach to Scientific Computation.* Academic Press, New York, 1983.
11. MOORE, R. E. *Interval Analysis.* Prentice-Hall, Englewood Cliffs, N. J., 1966.
12. MOORE, R. E. *Techniques and Applications of Interval Analysis*, vol. 2, SIAM Studies in Applied Mathematics. SIAM, Philadelphia, Pa., 1979.
13. NEAGA, M. Pascal-SC Language Description and Programming Guide (German). Department of Computer Science, University of Kaiserslautern, Kaiserslautern, W. Germany, 1982.
14. ORTEGA, J. M., AND RHEINBOLDT, W. C. *Iterative Solution of Nonlinear Equations in Several Variables.* Academic Press, New York, 1970.
15. RALL, L. B. *Computational Solution of Nonlinear Operator Equations.* Krieger, Huntington, N. Y., 1979.
16. RALL, L. B. Applications of software for automatic differentiation in numerical computation. *Computing, Suppl. 2* (1980), 141–156.
17. RALL, L. B. Automatic Differentiation: *Techniques and Applications*, Lecture Notes in Computer Science, vol. 120. Springer-Verlag, Berlin, Heidelberg, New York, 1981.
18. RALL, L. B. Differentiation and generation of Taylor coefficients in PASCAL-SC. In *A New Approach to Scientific Computation*, U. W. Kulisch and W. L. Miranker, Eds. Academic Press, New York, 1983, pp. 291–309.
19. RALL, L. B. Representations of intervals and optimal error bounds. *Math. Comput. 41*, 163 (1983), 219–227.
20. RALL, L. B. Differentiation in Pascal-SC: Type GRADIENT. *ACM Trans. Math. Softw. 10*, 2 (June 1984), 161–184.
21. RUMP, S. Solving algebraic problems with high accuracy. In *A New Approach to Scientific Computation*, U. W. Kulisch and W. L. Miranker, Eds. Academic Press, New York, 1983, pp. 53–120.
22. WOLFF VON GUDENBERG, J. Complete Arithmetic of the PASCAL-SC Computer: User Handbook (German). Institute for Applied Mathematics, University of Karlsruhe, Karlsruhe, W. Germany, 1981.

Received February 1984; revised August 1984; accepted September 1984