ALGORITHM 630
BBVSCG—A Variable-Storage Algorithm for Function Minimization

A. BUCKLEY and A. LENIR
Concordia University

Categories and Subject Descriptors: [**Mathematics of Computing**]: Numerical Analysis—*optimization*; G.1.6 [**Numerical Analysis**]: Optimization—*gradient methods*

General Terms: Algorithms

Additional Key Words and Phrases: Conjugate gradient, quasi-Newton method, Unconstrained minimization

1. PURPOSE

This routine is designed to find a close approximation to a local minimum of a nonlinear function $f(x)$. Here x is a vector of n variables, that is, $x = (x_1, x_2, \ldots, x_n)$, and f is assumed to be smooth, that is, to have at least continuous second derivatives. As with almost all minimization algorithms, there is no attempt made to ensure that the minimum obtained is global.

2. METHOD

The algorithm is based on an earlier algorithm, namely, CONMIN, due to Shanno and Phua [6], but offers a fundamental facility which was not available in CONMIN. In the CONMIN code, one could either use a conjugate gradient code if little storage was available, or a quasi-Newton code if there was sufficient storage. BBVSCG offers the user the opportunity to specify the amount of available storage; the code then chooses an appropriate algorithm. What is significant is that, if there is not enough space to run the quasi-Newton part of the algorithm, then this new algorithm will use *all* of the space that has been allocated to it. If this is more than what is needed to run the conjugate gradient code of CONMIN, as one might often expect to be the case, then the algorithm BBVSCG is such that one can expect a more efficient and rapid convergence to the minimum. This

Received February 1982; revised October 1984, accepted February 1985.
Authors' addresses: A. Buckley, Department of Mathematics, Statistics, and Computing Science, Dalhousie University, Halifax, Nova Scotia B3H 4H8, Canada. A. LeNir, Mathematics Department, Concordia University, Montreal, Quebec, H3B 1M8, Canada.

routine is also more adaptable to the needs of the user and offers such features as the ability to use reverse communication if necessary. BBVSCG can also run using less storage than CONMIN and will do automatic computation of finite-difference approximations for derivatives if requested.

The method can be briefly described by saying that, if there is insufficient space to use a quasi-Newton method, then a conjugate gradient method is picked. The conjugate gradient method uses a preconditioner defined by m quasi-Newton update terms, where m is as large as possible, subject only to the availability of storage. The number of updates used may even be 0, which amounts to using an ordinary conjugate gradient algorithm, but implemented in the manner described in [4]. A complete description of the approach and the motivation for its development is given in that paper. In two cases, where storage is sufficient to use a quasi-Newton code, or where there is just storage for one update ($m = 1$), BBVSCG is logically equivalent to CONMIN, and the comments of Shanno and Phua [6] apply. How these comments must be modified in the intermediate cases is given in [4] and is too lengthy to repeat here. Therefore, we refer the reader to [4] for more specific information on the background of the method. Evidence that the algorithm works well is also given in [4].

3. DESCRIPTION

3.1 Installation Guideline

We begin with this section because one cannot use this routine without knowing how to install it, even though it is a rather mundane exercise. The following steps are necessary to use BBVSCG.

(1) Read Section 3.2, decide whether the single or double-precision version is wanted, and read the correct version from the distribution tape. A complete description of the contents of that tape is not given here, but will be supplied from the ACM Distribution Service. It will, however, contain all of the routines from Section 3.9 and the test routines from Section 3.17. Both single and double precision versions will be on the tape.
(2) Read Section 3.3 on machine dependence and make the few but necessary modifications to the very short routines ZZMPAR and ZZSECS to tailor the routines to the system used.
(3) Compile the routines using the 1977 FORTRAN standard [2]. The compiled subroutines may then be used and, if desired, may be saved in a library.
(4) Read Section 3.4 for a description of the calling sequence.

3.2 Single/Double Precision

Both single and double precision versions are contained on the distribution tape, and it is left to users to select the desired version. It is advised, however, that users of 32- or 36-bit machines use double precision. As stated in Section 3.3, both versions have been tested. Note also that, regardless of which version is chosen, it is necessary to set a machine parameter that defines the relative

accuracy of the machine in use. This is described in detail in Section 3.3. Also note that, whichever version is selected, the statements needed for the alternate precision will appear in the code as comments, with columns 1–5 containing the string C!!!!.

3.3 Machine Dependence

The routines have been written to be as portable as possible, and there are very few instances of machine dependence. All routines are in American National Standard FORTRAN 1977 [2]. There is (as yet) no verifier (such as PFORT [5]) available for FORTRAN 77 to check for non-ANSI usage, but we have carefully tried to avoid nonstandard usage. We have run it on three quite different systems (VAX/VMS, VAX/UNIX[1] and CYBER/NOS) and there is no undocumented nonstandard code we are aware of. There are some comments about the code that some readers may find interesting; these appear in the Implementation Notes in the internal documentation. (Also, users of the 1966 FORTRAN standard [1] should see the Appendix.)

There are three specific instances of machine dependence. First, there is a call to a subroutine ZZMPAR for obtaining the value of the relative machine accuracy. ZZMPAR contains constants that define that value for many machines; these values were supplied by Dr. Jorge Moré of the Argonne National Laboratory. All statements defining these constants are comment statements, since they contain a C in column 1. Following the instructions in ZZMPAR, select the appropriate constants for the machine and precision being used and remove the C from column 1 for just those statements. Note that, if the double precision version is being implemented on a VAX 11/780, no action is needed, since the correct statements have already been selected. The same is true for single precision on any of the 60-bit CYBER series, for instance, the 7600 or the 170s. This means that those users on other systems must also ensure that these statements are commented out by putting a C in column 1 for the appropriate VAX or CYBER statements.

Second, a routine ZZSECS is included to provide timing information. ZZSECS contains a call to a system routine for obtaining CPU usage. That call must be replaced by a call that is appropriate for the hardware and operating system being used. The double precision version contains the correct call for a VAX 11/780 running under VMS; the single precision version contains the call for a CYBER running NOS. A simple and quick way to modify ZZSECS is to set SECS = 0.0 and then RETURN; this allows quick implementation of the routines, but no timing information will be obtained.

Third, the user may need to check the opening of the output file. The file (default unit 6) is not explicitly opened in any of the routines; it is assumed that the operating system will automatically open it with an appropriate name. If that is not the case, then it may be necessary to open it or to declare it on the PROGRAM statement before calling the minimization routine. If no output is requested from BBVSCG, this paragraph may be safely ignored.

[1] UNIX is a trademark of AT&T Bell Laboratories.

3.4 Calling Sequence

There are in fact two distinct ways of calling the minimization algorithm. In this section we describe the simplest of these; this should suffice for many or most users of BBVSCG. If more detailed information is desired, which may be the case for certain applications, particularly those in which the minimization problem is imbedded as a subproblem of a more general application, consult Section 3.10.

The simple call is

```
CALL BBVSCG (FUNCTN, N, X, F, G, ACC, STATUS, PFREQ, DERV,
             MAX, WORK, LWORK).
```

The arguments have the following meanings. Note that the column I/O specifies whether (in normal use) the argument must have a value defined on entry to BBVSCG, whether it is a value which will be returned, or both. When reference is made to a variable or array as being REAL, it may be taken to mean REAL or DOUBLE PRECISION, as appropriate.

Argument	I/O	Significance
FUNCTN	input	A subroutine must be provided to evaluate the function to be minimized. FUNCTN is the name of that subroutine, and it must be declared as EXTERNAL in the calling routine. The subroutine must have a fixed calling sequence, which is defined and illustrated in Section 3.6.
N	input	This is the dimension of the problem. The arrays X and G must have dimension N (at least). Of course, N must be an INTEGER.
X	input/output	On input, the array X specifies the initial guess at the minimum. There is no restriction on the values of the components of X, but the closer the guess is to a local minimum, the more rapid the convergence will usually be. On output, X contains the solution; that is, X contains the approximation to the minimum that the routine has determined is within the specified accuracy. If STATUS is not zero, this result should be taken with caution (see Section 3.7). Note that X must be a REAL array of dimension N (at least).
F	output	Provided STATUS = 0 on exit from the routine, F will contain the value of the function at the solution X. F must be a REAL variable.
G	output	Provided STATUS = 0 on exit from the routine, G will contain the value of the gradient vector at the solution X. G must be a REAL array of dimension N (at least).

ACC	input	ACC specifies the accuracy with which the user would like to determine the solution. The routine terminates when $\|G\| \leq ACC$ and $\|X - X^-\| \leq ACC \times \max(1, \|X\|)$. Here X is the current point and X^- denotes the previous iterate. It is possible to vary the termination criterion (see Sections 3.10 and 3.14). This argument must be a REAL variable.
STATUS	input/output	On input, STATUS is normally 0 (for the meaning of other values of STATUS on input, see Sections 3.10 and 3.15). On return from the routine, STATUS indicates whether or not the minimization was successful. If STATUS is 0, it is quite likely that the vector X does indeed represent a good approximation to a local minimum of the function. For the meaning of the nonzero return values of STATUS, see Section 3.7. STATUS must be an INTEGER.
PFREQ	input	PFREQ specifes the frequency at which to print the intermediate approximations to the solution. For the details see Section 3.12. PFREQ must be an INTEGER.
DERV	input	DERV specifies the manner in which the derivatives are to be computed. (Some comments on using these routines in a derivative-free mode appear in Section 3.13.) DERV must be an INTEGER. If DERV = 1 then derivatives are computed using analytic formulas. In this case, as discussed in Section 3.6, the subroutine FUNCTN must compute values of the derivatives when IFG ≤ 0. DERV = 2 then derivative values are obtained with finite difference approximations. Then FUNCTN only needs to be able to evaluate the function, and calls with IFG ≤ 0 will not occur and may be ignored. The user is not responsible for computing the difference approximations—these are done within a routine ZZEVAL. DERV = 3 then both analytic and finite-difference values are computed for the derivatives, and they are compared to ensure that they agree to within a

		reasonable amount. This is useful for newly coded functions (see Section 3.13).
		For details on the method used for computing the finite-difference approximations and for estimating the accuracy of these approximations against the analytically computed values, see the descriptive section of the routine ZZEVAL.
MAX	input	MAX puts an upper bound on the number of times the subroutine FUNCTN an be called to obtain a function value. (The count does not include the extra function evaluations needed to compute finite-difference approximations.) Once an attempt is made to exceed this upper bound, the routine terminates with an error code in STATUS (see Section 3.7). If MAX $\leq$ 0 on entry, there is no limit to the number of function evaluations allowed; this is not advised. MAX must be an INTEGER.
WORK		WORK is provided as temporary working storage for BBVSCG. Its size is important (see LWORK and Section 3.8). This argument must be a REAL array of dimension at least LWORK + 1.
LWORK	input	LWORK specifies the length of the working array WORK, that is, the amount of space available to BBVSCG. It is often the size (less 1) appearing in the dimension statement in which WORK was declared. LWORK must be an INTEGER (For more information, see Section 3.8).

3.5 Example

In Section 3.17 we describe some test problems that are provided with the package and that can be used to test whether it is working correctly. Here, however, we just wish to give a straightforward example of how to use the routines to minimize a simple nonlinear function. Suppose we want to minimize the function

$$f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2.$$

The following program does this is a very easy fashion; note that the evaluation of the function must be coded as in Section 3.6. This starts at $x_0 = (-1.2, 1)$ as an initial guess.

```
      PROGRAM MINF
C     SET THE SIZE OF THE WORKING ARRAY.
      INTEGER     LWORK
      PARAMETER ( LWORK = 50 )
C     DECLARE THE FUNCTION ROUTINE.
      EXTERNAL ROSENB
C     STATE THE VARIABLE DECLARATIONS.
      INTEGER DERV, N, PFREQ, MAX, STATUS
      REAL ACC, F, X(2), G(2), WORK(LWORK + 1)
C     SET THE INITIAL GUESS.
      X(1) = -1.2
      X(2) =  1.0
C     SET THE DIMENSION, ACCURACY DESIRED, ANALYTIC DERIVATIVES,
C     PRINT FREQUENCY (FIRST AND LAST POINTS) AND BOUND ON FUNC-
C     TION EVALUATIONS.
      N     = 2
      ACC   = .0001

      DERV  = 1
      PFREQ = 1000
      MAX   = 200
C     SET STATUS TO INDICATE NORMAL MODE AND MINIMIZE.
      STATUS = 0
      CALL BBVSCG ( ROSENB,
     -              N, X, F, G, ACC, STATUS,
     -              PFREQ, DERV, MAX, WORK, LWORK )
      STOP
      END
```

3.6 Subroutine FUNCTN

A subroutine must be provided for evaluating the function to be minimized. (If this is difficult for certain applications, consult Section 3.15.) The calling sequence of this subroutine is fixed (although the name may vary):

```
CALL FUNCTN ( N, X, F, G, WORK, IFG )
```

Here N is the dimension of the problem and X is a REAL vector of N components that specifies the point at which to evaluate the function. IFG is a flag with the following meaning:

IFG = −1 evaluate the gradient of the function (but *not* $f(x)$);
IFG = 0 evaluate both the gradient and the function;
IFG = 1 evaluate the function itself (but *not* its gradient).

After evaluating the function and/or gradient, as appropriate according to IFG, the function value should be returned in F, and/or the gradient value should be returned in the REAL vector G of length N. (Of course, no damage is done if either of the arrays X or G is declared with more than N components.) If the user wishes to specify analytic derivatives (see Sections 3.4 and 3.13), then this subroutine

must contain code for gradient evaluations when IFG ≤ 0. If only finite-difference computations are desired, then the code may assume that all calls will have IFG > 0. The array WORK is a working array which may be used as desired by the routine FUNCTN. It is passed from BBVSCG as described in Section 3.8.

If it happens to be awkward to code the test function according to this format, then one possible solution is the use of reverse communication, but another is to use a subroutine such as ZZFNS, as discussed in Section 3.15 and illustrated in Section 3.17.

For the function in the example of Section 3.5, the following subroutine is required:

```
SUBROUTINE ROSENB ( N, X, F, G, WORK, IFG )
```

To see how it can be coded, see the test routines documented in Section 3.17.

3.7 Return Codes

On return from BBVSCG, the argument STATUS has one of the following meanings:

0 This is normal termination. In all likelihood, the returned values of X, F and G are accurate representations of a local minimum, as requested.

1 In this case execution halted because an attempt was made to evaluate the function more than the allowed number of times (see MAX in Section 3.4).

2 In this case execution never began because insufficient storage was allocated (see Section 3.8).

3 In this case again execution never began, this time because an invalid method was specified. This should not occur unless the user has bypassed BBVSCG and directly called BBLNIR (see Section 3.10).

4 This indicates that the line search failed. This is probably due to a request for too high an accuracy in the solution (which varies from machine to machine) or to incorrect coding of the function and/or gradient evaluation routine FUNCTN. In the latter case, see Sections 3.6 and 3.13 and consider rerunning the problem with test mode for derivatives. Note that, when using finite-difference approximations for derivatives, one must be careful not to request too high an accuracy; otherwise this (i.e., STATUS = 4) is the likely result since differences will only be evaluated to a precision roughly half that of the machine in most cases.

5 This occurs when a nondownhill search direction has been generated. The normal cause for this is roundoff, and it can be triggered by the same errors as mentioned for STATUS = 4.

−1 See Section 3.15.

3.8 Size of WORK

The REAL array WORK referred to in Section 3.4 must have at least $3n + 1$ elements; thus LWORK > $3n$. It is desirable, however, that more work space be provided in this array. As described in [4], that is in fact the entire point of this routine.

If there is room available, then one can make WORK as large as possible. However, the most room that can be utilized by BBVSCG is given by LWORK =

$n(n + 7)/2$, so there is never any cause to allocate more space than that. In many cases, allocation of this full amount will result in the most rapid computation of the approximate minimum. However there are some situations in which this is not true. Two points might be mentioned. First, the algorithm determines the method to use on the basis of the available storage, as we explain shortly. It attempts first to use a quasi-Newton method, and then, failing that, to choose m as large as possible. The result is that a value m will never be chosen larger than $n/4$. Second, if one is using a virtual memory system, one must resist the temptation to always make LWORK large enough to use the quasi-Newton algorithm. The conjugate gradient part of the code requires very few iterations beyond that required for the quasi-Newton part, so the advantages of the quasi-Newton algorithm may be lost to the paging overhead. Indeed, for some functions, particularly those whose evaluation is not particularly complex, the execution time needed for a conjugate gradient minimization may be less than that using the quasi-Newton code. For more information, see [4].

The routine uses the storage allocated in WORK in the following way. If the full allocation of $n(n + 7)/2$ is given, then a true quasi-Newton method, which stores matrices as such and which is identical to that given by [6], is used. If not, then it computes

$$m = \frac{(\text{LWORK} - 3n)}{2n + 2},$$

and uses an m-update conjugate gradient algorithm as described in [4]. The user may override this choice of method, as described in Section 3.16, by specifying an appropriate value for the variable METH.

One final note needs to be made. It is often convenient in the subroutine FUNCTN, discussed in Section 3.6, to have some working storage available. One way of providing such an array is the following. Decide how much space is needed in FUNCTN, say EXTRA elements of a REAL array. Increase the size of WORK in the calling program by EXTRA, so that the size of WORK is LWORK + EXTRA. Do *not* change the value of LWORK passed to BBVSCG. Then BBVSCG will use the first LWORK entires of WORK itself, but it will pass the remainder to the user routine FUNCTN as an array of length EXTRA. This may be used in FUNCTN as desired. Note that FORTRAN does not allow arrays of length 0 (even if the array is not needed), so that EXTRA must be ≥ 1, which is why the size of WORK is required to be at least LWORK + 1.

3.9 Subroutines Used

We now summarize the subroutines that are part of this minimization package. Note that all names begin either BB or ZZ; this is intended as a convenience to avoid conflict with other user defined external names. The names in brackets are entry points in each routine; these are used primarily for initialization. For convenience, we group them according to their approximate purpose.

(a) The principal minimization package:

BBDFLT This routine is used to initialize the other routines, namely, ZZEVAL, ZZPRNT, ZZTERM and BBLNIR.

BBDIAG [BBSDAG] This is used to preset the quasi-Newton matrix H in order to scale the gradient components for the initial step.

BBLINS [BBSLNS] The purpose of this routine is to perform one iteration of the line search, that is, to compute a new estimate of the trial step length α.

BBLNIR [BBLSET, BBVGET] This is the heart of the minimization process. The logic of [4] is incorporated in this routine.

BBMULT [BBSMLT] This routine performs the job of multiplying $H \times v$ when the matrix H is stored in sum form, as described in [4]. Here v is a vector.

BBUPDT [BBSUPD] This routine updates the quasi-Newton matrix H at the end of a step, whether H is in sum form or is stored as a matrix.

BBVSCG This routine is provided to make the job of calling BBLNIR a bit simpler, as in Section 3.4.

(b) Auxiliary routines. These perform tasks that are not really involved with minimizing $f(x)$ but which nevertheless must be done. These are

ZZEVAL [ZZECHK, ZZEDEF, ZZEGET, ZZESET and ZZESRT] Controls evaluation of the function and/or gradient (see Section 3.13).

ZZIRD Performs real-to-integer conversion when the value is expected to be exactly integral.

ZZNRM2 Computes the Euclidean norm of a vector with due regard to avoiding overflow or underflow.

ZZPRNT [ZZPGET and ZZPSET] Controls the printing of intermediate results (see Section 3.12).

ZZTERM [ZZTDEF, ZZTGET, and ZZTSET] Controls termination of the routine (see Section 3.14).

(c) Routines to isolate machine dependence: These are described in Section 3.3.

ZZMPAR

ZZSECS

3.10 Internal Structure

Here we describe in more detail the manner in which this set of routines is organized. If it is found that the basic call to BBVSCG is inadequate for some reason, then this section should be read. It is expected that there will be little need to read this or subsequent sections for most users of BBVSCG.

First, recall that there are some auxiliary routines that are used. These include, in particular, ZZEVAL, ZZPRNT, and ZZTERM. (These routines actually come from a package TP [3] of more general use, which is why they may seem to have more features than might be considered necessary here.) We can now summarize what BBVSCG does:

1. It initializes the three routines ZZEVAL, ZZPRNT and ZZTERM before beginning the minimization. It does this by calling a routine BBDFLT which contains the default values needed for the initialization. BBDFLT then calls entry points in these routines to actually set the initial values. It also calls an entry point

in BBLNIR to set a number of its parameters to default values. Note that extensive use is being made here of SAVE variables in FORTRAN 77.

2. BBVSCG then subdivides the array WORK into 4 subarrays.
3. Finally, BBVSCG calls a routine BBLNIR which is responsible for the actual minimization of the function. The call is very similar to that of BBVSCG:

```
CALL BBLNIR(FUNCNM,N,X,F,G,ACC,STATUS,D,XX,GG,H,HDIM).
```

The arguments have the same meaning as those BBVSCG, except that WORK has been subdivided into 3 arrays D, XX, and GG of length N, and a fourth array H of length HDIM = LWORK − 3 × N.

If one wishes to exercise more control, BBLNIR can be called directly. In this case, it is necessary to perform the tasks above before doing the call for minimization. In doing so, we are no longer bound to use the default values that have been chosen; this is in fact probably the reason one would choose this approach. There are then two obvious ways one can proceed.

(a) Perform steps 1, 2, 3 as above, but insert a new step 1a after step 1 to explicitly override the defaults set by BBDFLT. One then needs to call only those entry points that redefine the specific values to be changed.
(b) Replace step 1 above with direct calls to all of the entry points in the four routines; then do the call to BBLNIR, supplying the appropriate arrays.

In either case, the entry points are explained in subsequent sections, although for complete details regarding any of the material in Sections 3.12–3.16, the reader should consult the internal documentation of the appropriate routine. Because of the number of values which must be set in five separate entry point calls to the four routines, we recommend (a), unless a lot of the values are being changed.

As an example, to change the norm for the termination test to the maximum norm, add the following declarations to the program of Section 3.5:

```
      INTEGER NORM, TYPE
```

and then replace the call to BBVSCG in Section 3.5 with

```
C     SET DEFAULTS
      CALL BBDFLT ( PFREQ, DERV, MAX )
C     OVERRIDE THE DEFAULT FOR THE NORM.
      NORM = 3
      TYPE = 4
      CALL BBTSET ( NORM, TYPE )
C     SUBDIVIDE THE WORK ARRAY.
      I1 = 1
      I2 = I1 + N
      I3 = I2 + N
      I4 = I3 + N
C     NOW DO THE MINIMIZATION.
      CALL BBLNIR(ROSENB,
     -            N, X, F, G, ACC, STATUS, WORK(I1), WORK(I2),
     -            WORK(I3),   WORK(I4),    LWORK-3*N         )
```

3.11 Changing Defaults

Now that the structure of this minimization packages is known, it is clear how permanent changes can be made if, for example, one wishes to put these routines in a library. As mentioned, one could call BBDFLT and BBLNIR directly to affect the change, but it may be the case that one would prefer to alter the defaults that we have chosen for certain parameters. This may then be done by directly changing the values in the PARAMETER or DATA statements in BBDFLT. The meanings of all parameters can be ascertained from Sections 3.12–3.14 and 3.16, or by examining the listings of the routines ZZEVAL, ZZPRNT, ZZTERM, and BBLNIR.

3.12 Print Control

*Part I: (Even if 3.10 has **not** been read)*

One can change the print frequency PFREQ. If it is set to 0, no printing occurs at all. If it is nonzero, then printing will occur at the initial point, at regular intervals thereafter, and at the final point. The interval, that is, the number of iterations between printing, is given by the magnitude of PFREQ. Furthermore, if PFREQ < 0, then just the iteration number and function value are printed, whereas if PFREQ > 0, the current point and gradient value are printed as well. To reiterate, the first and last points will always be printed when PFREQ $\neq 0$.

*Part II: (If 3.10 **has** been read)*

The print routine ZZPRNT must be initialized before using BBLNIR by calling the entry point ZZPSET (UNIT, GRAD, PFREQ). PFREQ is the argument described in Part I, UNIT is an integer specifying the unit number on which output (if any) is to be put, and GRAD controls printing of the gradient: if it is TRUE, then gradients will be printed if requested; otherwise they will never appear. This is convenient if one wishes to print the current point but not the gradient.

If one wishes to remove ZZPRNT from the package (say to replace it with a more personalized output routine), there are only two calls in BBLNIR. The first is at the top of Phase III and the other is in the EXIT section. Of course, the call to ZZPSET in BBDFLT should also be removed. It would be appropriate to remove ZZPRNT if no printing was ever to be done, or as stated, if another print routine was to be used.

3.13 Function/Gradient Evaluation

*Part I: (Even if 3.10 has **not** been read)*

The routine BBVSCG allows the user to exercise control over the evaluation of the function and gradient. First, a choice of analytic or finite-difference gradient evaluations is allowed, as specified by DERV (see Section 3.4). If finite differences are used, then it is very important that the machine precision be properly defined in ZZMPAR, as stated in Section 3.3. Differences are computed using forward differences and a step length h roughly equal to the root of the machine precision; details are in ZZEVAL.

We should emphasize that this algorithm is based on principles of minimization which assume that gradient information is available. Finite-difference

approximations to those gradients should therefore be considered simply as a user convenience. If a truly "derivative free" algorithm is required, then either the line search module BBLINS should be replaced with an actual derivative free line search, or else a method based on different techniques should be sought.

When test mode is selected, the analytic gradient value is computed and used, but in addition, a finite-difference approximation to each and every gradient component is determined and then an estimate that gives an indication of the level of agreement between the analytic and difference values is provided. Note that test mode involves computation of function values using both h and $\sqrt{h}$, so it can be fairly expensive. This estimate can be obtained by doing a CALL ZZECHK (ERROR, DECIML, INDEX, ITERAT) after BBVSCG is finished. ERROR and DECIML are REAL, and INDEX and ITERAT are INTEGERs.

DECIML is an average taken over all of the gradients computed since ZZEVAL was last initialized. Its value is the number of significant figures of agreement between the analytic and difference gradient values. If the machine in use has roughly t decimal figures of accuracy, then difference approximations will normally be expected to have about $\frac{1}{2}t$ figures of agreement with the analytic gradients. However, as the minimum is approached and the gradient values become small, the agreement will often not be that good, so that the average value given by DECIML can be expected to be somewhat less than $\frac{1}{2}t$. What is important is that, in the event of gross blunders in the coding of function and/or gradient values, DECIML will probably be something like $\frac{1}{2}$ to 1.

ERROR, ITERAT and INDEX give a record of the worst agreement found between the analytic and difference computations. ERROR is given as the actual relative agreement. ITERAT is the value of the gradient evaluation counter taken when ERROR was recorded; INDEX is the component of the gradient for which the worst error was found. This should help isolate errors in the function/gradient evaluation routine. Note that these estimates are unlikely to be reliable for very small gradient values. Thus, when the gradient or its estimate becomes too small, ERROR is flagged by making it negative (without changing its magnitude). More details are in ZZEVAL.

Some simple statistics are also available from ZZEVAL after minimization is complete. If ZZEGET (FNCT, GRCT, TIME) is called, the following will be obtained:

FNCT the number of times the function was evaluated (not including calls needed for finite-difference calculations);

GRCT the number of times the gradient was evaluated, whether by analytic or finite-difference formulas;

TIME the total number of CPU seconds used while in ZZEVAL and the user's function evaluation subroutine.

FNCT and GRCT are returned as INTEGERs and TIME is a REAL.

Part II: (If 3.10 ***has*** *been read)*

The function/gradient evaluation is controlled by a routine ZZEVAL which is called by BBLNIR whenever these values are required. If reverse communication is being used (see Section 3.15), then ZZEVAL is no longer called, and the facilities offered therein become unavailable unless coded directly by the user.

There are two entry points to ZZEVAL that control its execution; normally, these are called automatically. The first is ZZESET (TRF, TRG, UNIT). Since each of the three values has a default value (given in brackets) within ZZEVAL, this call is not mandatory. The arguments have the following meanings:

TRF If true, print the function value when it is computed [False].

TRG If true, print the gradient value when it is computed [False].

(If either TRF or TRG is true, then print an error message if too many function evaluations are attempted.)

UNIT The output unit number on which to print the function and gradient values, when requested, or the error message, when appropriate [6].

The second entry point is ZZESRT (FSCALE, DERV, MAX). This *must* be called before each function is minimized, for, in addition to initializing the arguments, the counting of function evaluations and timing are initialized during this call. DERV and MAX have the meanings defined in Section 3.4; FSCALE specifies that a nonlinear scaling is to be applied to the function. The details are given in the listing of ZZEVAL.

3.14 Termination Control

Part I: (Even if 3.10 has ***not*** *been read)*

There are three ways the user can control the criteria used for deciding when to terminate a minimization run. First, one may prescribe the desired accuracy as explained in Section 3.4. Second, one may pick the norm used for calculating the lengths of the vectors. And third, the actual test on which the decision is based may be changed. Changing the accuracy ACC can be done simply by specifying the desired precision in the calling sequence to BBVSCG (or BBLNIR). To change the others requires reading Section 3.10.

Part II: (If 3.10 ***has*** *been read)*

If so, then note that BBDFLT contains a call to ZZTSET, which is an entry point in the routine ZZTERM of the form CALL ZZTSET (NORM, TYPE). The default value for NORM is 2, which means that the Euclidean norm is used in evaluating the test described under ACC in Section 3.4. Setting NORM to 1 means the absolute sum (l_1) norm will be used instead; setting it to 3 means the maximum value (l_∞) norm will be selected.

The default value for TYPE is 4, which means that the termination test described in Section 3.4 will be used. If TYPE is set to 1, termination occurs when $\| G \| <$ ACC; the norm used is determined by the setting of NORM. If TYPE = 2, the same step size test as for TYPE = 4 is applied, but this time alone, without the gradient test. If TYPE = 3, termination occurs when $\| G \| <$ ACC $\times$ max(1, $\| X \|$); this is a criterion used by Shanno and Phua in CONMIN [6]. For more details, see the descriptive section of ZZTERM. Note that the setting of TYPE only affects normal termination; exists due to errors (see STATUS in Section 3.7) are not affected.

If it is desired to code either one of the given or some other termination criterion directly into BBLNIR, then ZZTERM can be removed. In this case the call to ZZTSET in BBDFLT must also be removed, and there are two calls to

ZZTERM in BBLNIR which must be taken out. One is in Phase III and the other in Phase VII; both occur immediately after the function and gradient have been evaluated.

3.15 Reverse Communication

*Part I: (Even if 3.10 has **not** been read)*

In some circumstances, calling a routine with a fixed calling sequence may just not be appropriate. This routine allows the user the option of using reverse communication.

Before explaining, however, we should observe that this routine may not be necessary. One may set up a dummy routine with the correct calling sequence for BBVSCG, and then use it to call the real evaluation routine with whatever calling sequence is appropriate. Additional parameters can be passed into the dummy routine either through COMMON or by using entry points. The use of the routine ZZFNS, provided with the test routines discussed in Section 3.17, shows how this can be done. The advantage to this approach is that all of the facilities of ZZEVAL, such as the automatic computation of finite differences or testing of the evaluation routine, are still available.

If reverse communication is to be used, the following adjustments are necessary in the use of BBVSCG.

(1) On the initial call to BBVSCG, STATUS must be set to 1 and the function value and gradient value at the initial guess X must be provided in F and G.
(2) Whenever BBVSCG requires function and gradient values, it will return to the calling program with STATUS = −1 and with X containing the coordinates of the point at which to evaluate the function and gradient.
(3) The calling routine is then responsible for obtaining the function and gradient values at X, after which it must call BBVSCG again, but with STATUS = 2.

The following points should be noted. First, on return from BBVSCG to the calling routine, any value for STATUS other than −1 is to be interpreted in the normal way, as in Section 3.7, and the run is then complete. Second, ZZEVAL no longer controls the computation of the function and gradient, so the facilities described in Section 3.13 are no longer available. In particular, if the user wishes to compute finite-difference approximations to the derivatives, it is now the user's responsibility. Also, it now is up to the user to count function calls and ensure that no more than MAX are done, if that is what is desired. In other words, the two arguments DERV and MAX no longer have any effect. Observe that in certain circumstances it may still be appropriate to use ZEEVAL to get function/gradient values, even when reverse communication is being used. If so, then the user must call BBDFLT before the initial call to BBVSCG described in (1) above; this ensures that ZZEVAL is properly initialized.

*Part II: (If 3.10 **has** been read)*

If BBLNIR is being called directly, the comments of Part I apply to BBLNIR in the same way, but there are other points to be aware of. First, as stated in Section 3.10, the user must ensure that the routines are properly initialized through BBDFLT or the proper entry points. In particular, when still using

ZZEVAL as discussed at the end of Part I, it is necessary to initialize ZZEVAL by calling either BBDFLT or ZZESRT before the first call to BBLNIR. On the other hand, since the facilities of ZZEVAL are probably no longer being used, this routine may be deleted. However, if that is done, the calls in BBDFLT to initialize ZZEVAL must be removed. Also, ZZPRNT contains a call to an entry point ZZVGET in ZZEVAL to get the counts of function and gradient evaluations; this call should be removed and it should be remembered that the main program may no longer call ZZVGET to get final function/gradient evaluation counts. Finally, the scaling of the function is no longer available.

3.16 Control Parameters for BBLNIR

The routine BBLNIR contains a number of variables which have very specific effects on the execution of the algorithm. These are all set by calling the entry point BBLSET; in addition, all have default values defined in BBLNIR. These values have been determined to be those for which the routine is both efficient and reliable, but for those who are very familiar with the method, it may be desired to try the routine with various settings of these control variables. The description of these variables and their effect on the routine is given within the descriptive section of BBLNIR and is not repeated here. Note that it is also possible to trigger an internal trace of the execution of BBLNIR by setting a number of trace flags.

3.17 Test Program and Storage Requirements

Included with the routines described in Section 3.9 (which constitute the actual minimization algorithm) are several routines to facilitate implementing these routines on a specific system. These routines are

1. TESTBB This is a main program that tests the minimization algorithm on a variety of problems.
2. ZZFNS The original of this routine, which is part of TP [3], contains a large collection of nonlinear functions, ten of which have been selected here to test the minimization algorithm. A simple integer selects one of the test functions; the ones used are

```
BARD70, BIGGS6, BOX663, CRGLVY, ENGVL2,
PENAL1, PENAL2, PWSING, ROSENB, SCHMVT.
```

There are several points to note. The test function actually called by BBVSCG is ZZFNS; its calling sequence is precisely as described in Section 3.6. With respect to Section 3.5, observe that the sample function ROSENB is one of the ten functions if one wishes to examine its coding. Note that ROSENB does not actually need the working array WORK, so it is simply not used. On the other hand, PENAL2 does require additional array storage; this is available from the array WORK, the size of which is set in a call in the main routine to the entry point ZZFSET. Notice as well that BIGGS6 and BOX663 both require an integer argument which is passed in (as a REAL) to ZZFNS as the first element in the array FARG in the main routine. Similarly, PENAL1 and PENAL2 need two real arguments which are passed in a similar fashion.

Second, note that TESTBB minimizes each of the ten test functions a number of times. In each case, slightly different parameter settings are used in the calls

to BBVSCG and BBLNIR. These are fully described in the descriptive section of TESTBB. Also included is a copy of the output obtained by running TESTBB on a VAX 11/780 running under Unix. Executing the test on a different machine will generally result in slightly different output, but the key points are the following. At the very end of the output, the number of errors listed should be zero. Also, the column AV. DECIMALS should be fairly consistently about one half of the number of decimals of accuracy of the machine in use. On the VAX for example, that means that AV.DECIMALS should be about 7 to 8 (assuming double precision calculations). Note that the value of ACC in TESTBB can not be as small if a machine is being used with much less than 15 figures of relative accuracy. Or, to put it another way, on machines such as the IBM 32-bit machines, double precision should be used.

APPENDIX

If it were essential, the code could be converted to 1966 Standard FORTRAN; the code contains no character variables and we believe that the following steps would be sufficient for the conversion: (a) replace PARAMETER statements with DATA statements; (b) check DO loops for null cases; (c) simulate the IF THEN (ELSE) structure with GOTO statements; (d) replace generic function names with appropriate single or double precision names; (e) replace array declarations of the form H(*) in certain subroutines with declarations H(1); and (f) replace any "'string'" in FORMAT statements with the Hollerith "nHstring".

ACKNOWLEDGMENTS

The authors would like to express their appreciation for the support of this work by the Natural Sciences and Engineering Research Council of Canada under operating grant A8962 (Buckley) and by the National Research Council of Canada through a Postgraduate Scholarship (LeNir). Also, we would like to thank the Computing Center of Concordia University and the MIS Department of the University of Arizona for providing the computer facilities needed for the development of this software.

REFERENCES

1. ANSI. Programming Language FORTRAN, ANSI X3.9-1966. American National Standards Institute, New York, 1966.
2. ANSI. Programming Language FORTRAN, ANSI X3.9-1978. American National Standards Institute, New York, 1978; International Standard ISO 1539-1980(E).
3. BUCKLEY, A. A portable package for testing minimization algorithms. In *Evaluating Mathematical Programming Techniques*. John M. Mulvey, ed. Springer-Verlag, New York, 1982.
4. BUCKLEY, A. AND LENIR, A. QN-like variable storage conjugate gradients. *Math Prog. 27* (1983), 155–175.
5. RYDER, B. G. The PFORT Verifier, *Softw. Prac. and Exper. 4* (1974), 359–377.
6. SHANNO, D. F. AND PHUA, K. H. Remark on Algorithm 500. *ACM Trans. Math. Softw. 6*, 4 (1980), 618–622.