GALAHAD SHA

USER DOCUMENTATION GALAHAD Optimization Library version 4.1

1 SUMMARY

This package **computes a component-wise secant approximation to the Hessian matrix** $\mathbf{H}(\mathbf{x})$, for which $(\mathbf{H}(\mathbf{x}))_{i,j} = \partial f^2(\mathbf{x})/\partial x_i \partial x_j$, $1 \leq i, j \leq n$, using values of the gradient $\mathbf{g}(\mathbf{x}) = \nabla_x f(\mathbf{x})$ of the function $f(\mathbf{x})$ of n unknowns $\mathbf{x} = (x_1, \ldots, x_n)^T$ at a sequence of given distinct $\{\mathbf{x}^{(k)}\}$, $k \geq 0$. More specifically, given **differences**

$$\mathbf{s}^{(k)} = \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)} \text{ and } \mathbf{y}^{(k)} = \mathbf{g}(\mathbf{x}^{(k+1)}) - \mathbf{g}(\mathbf{x}^{(k)})$$

the package aims to find an approximation $\mathbf{B}$ to $\mathbf{H}(\mathbf{x})$ for which the secant conditions $\mathbf{B}\mathbf{s}^{(k)} \approx \mathbf{y}^{(k)}$ hold for a chosen set of values k. The methods provided take advantage of the entries in the Hessian that are known to be zero.

The package is particularly intended to allow gradient-based optimization methods, that generate iterates $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \mathbf{s}^{(k)}$ based upon the values $\mathbf{g}(\mathbf{x}^{(k)})$ for $k \geq 0$, to build a suitable approximation to the Hessian $\mathbf{H}(\mathbf{x}^{(k+1)})$. This then gives the method an opportunity to accelerate the iteration using the Hessian approximation.

ATTRIBUTES — Versions: GALAHAD_SHA_single, GALAHAD_SHA_double. **Uses:** GALAHAD_SYMBOLS, GALAHAD_SPECFILE and GALAHAD_SPACE. **Date:** August 2023. **Origin:** N. I. M. Gould, STFC-Rutherford Appleton Laboratory, **Language:** Fortran 2003.

2 HOW TO USE THE PACKAGE

Access to the package requires a USE statement such as

Single precision version

```
USE GALAHAD_SHA_single
```

Double precision version

```
USE GALAHAD_SHA_double
```

The user is **strongly advised** to use the double precision version unless single precision corresponds to 8-byte arithmetic.

If it is required to use both modules at the same time, the derived types SHA_control_type, SHA_inform_type, SHA_data_type and NLPT_userdata_type, (Section 2.1) and the subroutines SHA_initialize, SHA_analyse, SHA_estimate, SHA_terminate, (Section 2.2) and SHA_read_specfile (Section 2.4) must be renamed on one of the USE statements.

2.1 The derived data types

Four derived data types are accessible from the package.

2.1.1 The derived data type for holding control parameters

The derived data type SHA_control_type is used to hold controlling data. Default values may be obtained by calling SHA_initialize (see Section 2.2.1), while components may also be changed by calling GALAHAD_SHA_read_spec (see Section 2.4.1). The components of SHA_control_type are:

`error` is a scalar variable of type default `INTEGER`, that holds the stream number for error messages. Printing of error messages in `SHA_analyse`, `SHA_estimate` and `SHA_terminate` is suppressed if `error` ≤ 0. The default is `error = 6`.

`out` is a scalar variable of type default `INTEGER`, that holds the stream number for informational messages. Printing of informational messages in `SHA_analyse` and `SHA_estimate` is suppressed if `out` < 0. The default is `out = 6`.

`print_level` is a scalar variable of type default `INTEGER`, that is used to control the amount of informational output which is required. No informational output will occur if `print_level` ≤ 0. If `print_level` > 01, details of any data errors encountered will be reported. The default is `print_level = 0`.

`approximation_algorithm` is a scalar variable of type default `INTEGER`, that is used to select which approximation algorithm employed. This may be

1. 1. unsymmetric (Algorithm 2.1 in paper)
2. 2. symmetric (Algorithm 2.2 in paper)
3. 3. composite (Algorithm 2.3 in paper)
4. 4. composite 2 (Algorithm 2.4 in paper)
5. 5. cautious (Algorithm 2.5 in paper)

Any value outside this range will be reset to the default; the default is `approximation_algorithm = 4`.

`dense_linear_solver` is a scalar variable of type default `INTEGER`, that specifies which (LAPACK) dense linear equation solver to use when finding the values of entries in each row of **B**. This may be

1. 1. Gaussian elimination
2. 2. QR factorization
3. 3. singular-value decomposition
4. 4. singular-value decomposition with divide-and-conquer

Any value outside this range will be reset to the default; the default is `dense_linear_solver = 3`.

`max_sparse_degree` is a scalar variable of type default `INTEGER`, that is used to specify the maximum sparse degree if a composite algorithm (`%approximation_algorithm = 3, 4` is employed. The default is `max_sparse_degree = 50`.

`extra_differences` is a scalar variable of type default `INTEGER`, that is used to specify how may additional gradients (in addition to the number output in `inform%differences_needed` from `SHA_analyse`) are available when calling `SHA_estimate`. The default is `extra_differences = 0`.

`space_critical` is a scalar variable of type default `LOGICAL`, that must be set `.TRUE.` if space is critical when allocating arrays and `.FALSE.` otherwise. The package may run faster if `space_critical` is `.FALSE.` but at the possible expense of a larger storage requirement. The default is `space_critical = .FALSE.`.

`deallocate_error_fatal` is a scalar variable of type default `LOGICAL`, that must be set `.TRUE.` if the user wishes to terminate execution if a deallocation fails, and `.FALSE.` if an attempt to continue will be made. The default is `deallocate_error_fatal = .FALSE.`.

`prefix` is a scalar variable of type default `CHARACTER` and length 30, that may be used to provide a user-selected character string to preface every line of printed output. Specifically, each line of output will be prefaced by the string `prefix(2:LEN(TRIM( prefix ))-1)`, thus ignoreing the first and last non-null components of the supplied string. If the user does not want to preface lines by such a string, they may use the default `prefix = ""`.

2.1.2 The derived data type for holding informational parameters

The derived data type SHA_inform_type is used to hold parameters that give information about the progress and needs of the algorithm. The components of SHA_inform_type are:

status is a scalar variable of type default INTEGER, that gives the exit status of the algorithm. See Section 2.3 for details.

alloc_status is a scalar variable of type default INTEGER, that gives the status of the last attempted array allocation or deallocation. This will be 0 if status = 0.

bad_alloc is a scalar variable of type default CHARACTER and length 80, that gives the name of the last internal array for which there were allocation or deallocation errors. This will be the null string if status = 0.

max_degree is a scalar variable of type default INTEGER, that holds the maximum degree in the adjacency graph.

differences_needed is a scalar variable of type default INTEGER, that holds the number of differences that will be needed (more may be helpful) by SHA_estimate. This value is computed by SHA_analyse.

max_reduced_degree is a scalar variable of type default INTEGER, that holds the maximum reduced degree in the adjacency graph.

approximation_algorithm_used is a scalar variable of type default INTEGER, that specifies the actual approximation algorithm used (see control%approximation_algorithm).

bad_row is a scalar variable of type default INTEGER, that holds the index of the first row for which a failure occurred when forming its Hessian values (or 0 if the data if no failures occurred).

2.1.3 The derived data type for holding problem data

The derived data type SHA_data_type is used to hold all the data for a particular problem, or sequences of problems with the same structure, between calls of SHA procedures. This data should be preserved, untouched from the initial call to SHA_initialize to the final call to SHA_terminate.

2.2 Argument lists and calling sequences

There are four procedures for user calls (see Section 2.4 for further features):

1. The subroutine SHA_initialize is used to set default values, and initialize private data, before solving one or more problems with the same sparsity and bound structure.

2. The subroutine SHA_analyse is called to analyze the sparsity pattern of the Hessian and to generate information that will be used when estimating its values.

3. The subroutine SHA_estimate is called to estimate the Hessian by component-wise secant approximation. This must be preceded by a call to SHA_analyse.

4. The subroutine SHA_terminate is provided to allow the user to automatically deallocate array components of the private data, allocated by SHA_solve, at the end of the solution process. It is important to do this if the data object is re-used for another problem **with a different structure** since SHA_initialize cannot test for this situation, and any existing associated targets will subsequently become unreachable.

We use square brackets [ ] to indicate OPTIONAL arguments.

2.2.1 The initialization subroutine

Default values are provided as follows:

```
CALL SHA_initialize( data, control, inform )
```

`data` is a scalar `INTENT(INOUT)` argument of type `SHA_data_type` (see Section 2.1.3). It is used to hold data about the problem being solved.

`control` is a scalar `INTENT(OUT)` argument of type `SHA_control_type` (see Section 2.1.1). On exit, `control` contains default values for the components as described in Section 2.1.1. These values should only be changed after calling `SHA_initialize`.

`inform` is a scalar `INTENT(OUT)` argument of type `SHA_inform_type` (see Section 2.1.2). A successful call to `SHA_initialize` is indicated when the component `status` has the value 0. For other return values of `status`, see Section 2.3.

2.2.2 The analysis subroutine

The analysis phase, in which the given sparsity pattern of the Hessian is used to generate information that will be used when estimating its values, is called as follows:

```
CALL SHA_analyse( n, nz, ROW, COL, data, control, inform )
```

`n` is a scalar `INTENT(IN)` scalar argument of type default `INTEGER`, that must be set to n the dimension of the Hessian matrix, i.e. the number of variables in the function f. **Restrictions:** `n` $>$ `0`.

`nz` is a scalar `INTENT(IN)` scalar argument of type default `INTEGER`, that must be set to the number of nonzero entries on and above the diagonal of the Hessian matrix. **Restrictions:** `nz` $\geq$ `0`.

`ROW` and `COL` are a scalar `INTENT(IN)` rank-one array arguments of type default `INTEGER` and dimension `nz`, that are used to describe the sparsity structure of the Hessian matrix, $\mathbf{H}(\mathbf{x})$. They must be set so that `ROW`(i) and `COL`(i), $i = 1, \ldots,$ `nz`, contains the row and column indices of the nonzero elements of the **upper triangular part, including the diagonal,** of the Hessian matrix. The entries may appear in any order. **Restrictions:** $1 \leq$ `ROW`(j) $\leq$ `COL`(j) $\leq n$, $j = 1, \ldots,$ `nz`.

`data` is a scalar `INTENT(INOUT)` argument of type `SHA_data_type` (see Section 2.1.3). It is used to hold data about the problem being solved. It must not have been altered **by the user** since the last call to `SHA_initialize`.

`control` is a scalar `INTENT(IN)` argument of type `SHA_control_type` (see Section 2.1.1). Default values may be assigned by calling `SHA_initialize` prior to the first call to `SHA_analyse`.

`inform` is a scalar `INTENT(INOUT)` argument of type `SHA_inform_type` (see Section 2.1.2). A successful call to `SHA_analyse` is indicated when the component `status` has the value 0. For other return values of `status`, see Section 2.3.

2.2.3 The estimation subroutine

The estimation phase, in which the nonzero entries of the Hessian are estimated by finite differences, is called as follows:

```
CALL SHA_estimate( n, nz, ROW, COL, m_available, S, ls1, ls2, &
                   Y, ly1, ly2, VAL, data, control, inform[, PRECEDENCE] )
```

`n, nz`, `ROW` and `COL` are `INTENT(IN)` arguments exactly as described and input to `SHA_analyse`, and must not have been changed in the interim.

m_available is a scalar INTENT(IN) scalar argument of type default INTEGER, that should be set to the number of differences provided; ideally this will be as large as inform%differences_needed as reported by SHA_analyse, but better still there should be a further control%extra_differences to allow for unlikely singularities.

S is a scalar INTENT(IN) rank-two array argument of type default REAL (double precision in GALAHAD_SHA_double), and dimension (ls1, ls2), that should be set on input so that the i-th entry of the k-th difference $s_i^{(k)}$ lies in S(i,k).

ls1 is a scalar INTENT(IN) scalar argument of type default INTEGER, that must be set to the length of the leading dimension of S, and must be at least n.

ls2 is a scalar INTENT(IN) scalar argument of type default INTEGER, that must be set to the length of the trailing dimension of S, and must be at least m_available.

Y is a scalar INTENT(IN) rank-two array argument of type default REAL (double precision in GALAHAD_SHA_double), and dimension (ly1, ly2), that should be set on input so that the i-th entry of the k-th difference $y_i^{(k)}$ lies in Y(i,k).

ly1 is a scalar INTENT(IN) scalar argument of type default INTEGER, that must be set to the length of the leading dimension of Y, and must be at least n.

ly2 is a scalar INTENT(IN) scalar argument of type default INTEGER, that must be set to the length of the trailing dimension of Y, and must be at least m_available.

VAL is a scalar INTENT(OUT) rank-one array argument of type default REAL (double precision in GALAHAD_SHA_double), and dimension nz, that will be set on output to the non-zeros of the Hessian approximation **B** in the order defined by the list stored in ROW and COL.

data is a scalar INTENT(INOUT) argument of type SHA_data_type (see Section 2.1.3). It is used to hold data about the problem being solved. It must not have been altered **by the user** since the last call to SHA_initialize.

control is a scalar INTENT(IN) argument of type SHA_control_type (see Section 2.1.1). Default values may be assigned by calling SHA_initialize prior to the first call to SHA_analyse.

inform is a scalar INTENT(INOUT) argument of type SHA_inform_type (see Section 2.1.2). A successful call to SHA_analyse is indicated when the component status has the value 0. For other return values of status, see Section 2.3.

PRECEDENCE is an OPTIONAL scalar INTENT(IN) rank-one array argument of type default integer and dimension m_available, that can be set to the preferred order of access of the differences stored in S and Y. The calculation of each row of the Hessian approximation **B** depends on the number of nonzeros in the row, and PRECEDENCE allows the user to specify the order in which the columns of S and Y are accessed to determine these row values. If PRECEDENCE is PRESENT the i-th accessed column will be PRECEDENCE(i). Otherwise the columns will be accessed in their natural order $i, i = 1, \ldots,$ m_available.

2.2.4 The termination subroutine

All previously allocated arrays are deallocated as follows:

```
CALL SHA_terminate( data, control, inform )
```

data is a scalar INTENT(INOUT) argument of type SHA_data_type exactly as for SHA_solve, which must not have been altered **by the user** since the last call to SHA_initialize. On exit, array components will have been deallocated.

control is a scalar INTENT(IN) argument of type SHA_control_type exactly as for SHA_solve.

`inform` is a scalar `INTENT(OUT)` argument of type `SHA_inform_type` exactly as for `SHA_solve`. Only the component `status` will be set on exit, and a successful call to `SHA_terminate` is indicated when this component `status` has the value 0. For other return values of `status`, see Section 2.3.

2.3 Warning and error messages

A negative value of `inform%status` on exit from `SHA_analyse`, `SHA_estimate` or `SHA_terminate` indicates that an error has occurred. No further calls should be made until the error has been corrected. Possible values are:

-1. An allocation error occurred. A message indicating the offending array is written on unit `control%error`, and the returned allocation status and a string containing the name of the offending array are held in `inform%alloc_status` and `inform%bad_alloc`, respectively.

-2. A deallocation error occurred. A message indicating the offending array is written on unit `control%error` and the returned allocation status and a string containing the name of the offending array are held in `inform%alloc_status` and `inform%bad_alloc`, respectively.

-3. One or more of the restrictions $\mathtt{n} > 0$, $\mathtt{nz} \geq 0$, $1 \leq \mathtt{ROW}(j) \leq \mathtt{COL}(j) \leq \mathtt{n}$, $j = 1, \ldots, \mathtt{nz}$, has been violated.

-10. The LAPACK dense linear equation solver used to find the values of the rows of **B** has failed.

-31. `SHA_estimate` has been called before `SHA_analyse`.

2.4 Further features

In this section, we describe an alternative means of setting control parameters, that is components of the variable `control` of type `SHA_control_type` (see Section 2.1.1), by reading an appropriate data specification file using the subroutine `SHA_read_specfile`. This facility is useful as it allows a user to change `SHA` control parameters without editing and recompiling programs that call `SHA`.

A specification file, or specfile, is a data file containing a number of "specification commands". Each command occurs on a separate line, and comprises a "keyword", which is a string (in a close-to-natural language) used to identify a control parameter, and an (optional) "value", which defines the value to be assigned to the given control parameter. All keywords and values are case insensitive, keywords may be preceded by one or more blanks but values must not contain blanks, and each value must be separated from its keyword by at least one blank. Values must not contain more than 30 characters, and each line of the specfile is limited to 80 characters, including the blanks separating keyword and value.

The portion of the specification file used by `SHA_read_specfile` must start with a "`BEGIN SHA`" command and end with an "`END`" command. The syntax of the specfile is thus defined as follows:

```
( .. lines ignored by SHA_read_specfile .. )
  BEGIN SHA
     keyword    value
     .......    .....
     keyword    value
  END
( .. lines ignored by SHA_read_specfile .. )
```

where keyword and value are two strings separated by (at least) one blank. The "`BEGIN SHA`" and "`END`" delimiter command lines may contain additional (trailing) strings so long as such strings are separated by one or more blanks, so that lines such as

```
BEGIN SHA SPECIFICATION
```

and

```
END SHA SPECIFICATION
```

are acceptable. Furthermore, between the “BEGIN SHA” and “END” delimiters, specification commands may occur in any order. Blank lines and lines whose first non-blank character is ! or * are ignored. The content of a line after a ! or * character is also ignored (as is the ! or * character itself). This provides an easy manner to ”comment out” some specification commands, or to comment specific values of certain control parameters.

The value of a control parameters may be of three different types, namely integer, logical or real. Integer and real values may be expressed in any relevant Fortran integer and floating-point formats (respectively). Permitted values for logical parameters are ”ON”, ”TRUE”, ”.TRUE.”, ”T”, ”YES”, ”Y”, or ”OFF”, ”NO”, ”N”, ”FALSE”, ”.FALSE.” and ”F”. Empty values are also allowed for logical control parameters, and are interpreted as ”TRUE”.

The specification file must be open for input when SHA_read_specfile is called, and the associated device number passed to the routine in device (see below). Note that the corresponding file is REWINDed, which makes it possible to combine the specifications for more than one program/routine. For the same reason, the file is not closed by SHA_read_specfile.

2.4.1 To read control parameters from a specification file

Control parameters may be read from a file as follows:

```
CALL SHA_read_specfile( control, device )
```

control is a scalar INTENT(INOUT) argument of type SHA_control_type (see Section 2.1.1). Default values should have already been set, perhaps by calling SHA_initialize. On exit, individual components of control may have been changed according to the commands found in the specfile. Specfile commands and the component (see Section 2.1.1) of control that each affects are given in Table 2.1.

command	component of control	value type
error-printout-device	%error	integer
printout-device	%out	integer
print-level	%print_level	integer
approximation-algorithm	%approximation_algorithm	integer
dense-linear-solver	%dense_linear_solver	integer
maximum-degree-considered-sparse	%max_sparse_degree	integer
extra-differences	%extra_differences	integer
space-critical	%space_critical	logical
deallocate-error-fatal	%deallocate_error_fatal	logical
output-line-prefix	%prefix	character

Table 2.1: Specfile commands and associated components of control.

device is a scalar INTENT(IN) argument of type default INTEGER, that must be set to the unit number on which the specfile has been opened. If device is not open, control will not be altered and execution will continue, but an error message will be printed on unit control%error.

2.5 Information printed

If control%print_level is positive, information about errors encountered will be printed on unit control%out.

3 GENERAL INFORMATION

Use of common: None.

Workspace: Provided automatically by the module.

Other routines called directly: None.

Other modules used directly: `SHA_solve` calls the GALAHAD packages `GALAHAD_SYMBOLS`, `GALAHAD_SPECFILE` and `GALAHAD_NLPT`.

Input/output: Output is under control of the arguments `control%error`, `control%out` and `control%print_level`.

Restrictions: $0 <$ `n`, $0 \leq$ `nz`, $1 \leq$ `ROW`$(j) \leq$ `COL`$(j) \leq$ `n`, $j = 1, \ldots,$ `nz`.

Portability: Fortran 2003. The package is thread-safe.

4 METHOD

The package computes the entries in the each row of $\mathbf{B}$ one at a time. The entries b_{ij} in row i may be chosen to

$$\underset{b_{i,j}}{\text{minimize}} \quad \sum_{k \in I_i} \left[\sum_{\text{nonzeros } j} b_{i,j} s_j^{(k)} - y_i^{(k)} \right]^2, \tag{4.1}$$

where I_i is ideally chosen to be sufficiently large so that (4.1) has a unique minimizer. Since this requires that there are at least as many $(\mathbf{s}^{(k)}, \mathbf{y}^{(k)})$ pairs as the maximum number of nonzeros in any row, this may be prohibitive in some cases. We might then be content with a minimum-norm (under-determined) least-squares solution. Or, we may take advantage of the symmetry of the Hessian, and note that if we have already found the values in row j, then the value $b_{i,j} = b_{j,i}$ in (4.1) is known before we process row i. Thus by ordering the rows and exploiting symmetry we may reduce the numbers of unknowns in future unprocessed rows.

In the analysis phase, we order the rows by constructing the connectivity graph—a graph comprising nodes 1 to n and edges connecting nodes i and j if $h_{i,j}$ is everywhere nonzero—of $\mathbf{H}(\mathbf{x})$. The nodes are ordered by increasing degree (that is, the number of edges emanating from the node) using a bucket sort. The row chosen to be ordered next corresponds to a node of minimum degree, the node is removed from the graph, the degrees updated efficiently, and the process repeated until all rows have been ordered. This often leads to a significant reduction in the numbers of unknown values in each row as it is processed in turn, but numerical rounding can lead to inaccurate values in some cases. A useful remedy is to process all rows for which there are sufficient $(\mathbf{s}^{(k)}, \mathbf{y}^{(k)})$ as before, and then process the remaining rows taking into account the symmetry. That is, the rows and columns are rearranged so that the matrix is in block form

$$\mathbf{B} = \begin{pmatrix} \mathbf{B}_{11} & \mathbf{B}_{12} \\ \mathbf{B}_{12}^T & \mathbf{B}_{22} \end{pmatrix},$$

the $(\mathbf{B}_{11} \;\; \mathbf{B}_{12})$ rows are processed without regard for symmetry but give the $2,1$ block $\mathbf{B}_{12}^T$, and finally the $2,2$ block $\mathbf{B}_{22}$ is processed either with the option of exploiting symmetry. More details of the precise algorithms (Algorithms 2.1–2.5) are given in the reference below. The linear least-squares problems (4.1) themselves are solved by a choice of LAPACK packages.

Reference:

The method is described in detail in

J. M. Fowkes, N. I. M. Gould and J. A. Scott, Approximating large-scale Hessians using secant equations. Technical Report TR-2023, Rutherford Appleton Laboratory.

5 EXAMPLES OF USE

Suppose we wish to estimate the Hessian matrix whose values at a given $\mathbf{x}$ are

$$\mathbf{H}(\mathbf{x}) = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 6 & 0 & 0 & 0 \\ 3 & 0 & 7 & 0 & 0 \\ 4 & 0 & 0 & 8 & 0 \\ 5 & 0 & 0 & 0 & 9 \end{pmatrix}$$

and that we have (artificially) sampled the matrix via $\mathbf{y}^{(k)} = \mathbf{H}(\mathbf{x})\mathbf{s}^{(k)}$ along random vectors $\mathbf{s}^{(k)}$ for $k = 1, \ldots, k_s$; a suitable value for k_s is returned by `SHA_analyse`. Then we may recover $\mathbf{H}(\mathbf{x})$ as follows:

```
! THIS VERSION: GALAHAD 4.1 - 2023-08-19 AT 15:40 GMT.
   PROGRAM GALAHAD_SHA_EXAMPLE
   USE GALAHAD_SHA_double  ! double precision version
   USE GALAHAD_RAND_double
   IMPLICIT NONE
   INTEGER, PARAMETER :: wp = KIND( 1.0D+0 ) ! set precision
   TYPE ( SHA_data_type ) :: data
   TYPE ( SHA_control_type ) :: control
   TYPE ( SHA_inform_type ) :: inform
   INTEGER :: i, j, k, k_s, l
   REAL ( KIND = wp ) ::  v
   INTEGER, ALLOCATABLE, DIMENSION( : ) :: PRECEDENCE
   REAL ( KIND = wp ), ALLOCATABLE, DIMENSION( : , : ) :: S, Y
   TYPE ( RAND_seed ) :: seed
   INTEGER, PARAMETER :: n = 5, nz = 9  ! set problem data
   INTEGER :: ROW( nz ), COL( nz )
   REAL ( KIND = wp ) ::  VAL( nz ), VAL_est( nz )
   ROW = (/ 1, 1, 1, 1, 1, 2, 3, 4, 5 /)  ! N.B. upper triangle only
   COL = (/ 1, 2, 3, 4, 5, 2, 3, 4, 5 /)
   VAL = (/ 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0 /) ! artificial values
   CALL SHA_initialize( data, control, inform )  ! initialize
   control%approximation_algorithm = 2 ! symmetric approximation
   CALL SHA_analyse( n, nz, ROW, COL, data, control, inform ) ! analyse sparsity
   IF ( inform%status /= 0 ) THEN             ! Failure
     WRITE( 6, "( ' return with nonzero status ', I0, ' from SHA_analyse' )" ) &
       inform%status ; STOP
   END IF
   WRITE( 6, "( 1X, I0, ' differences are needed,',                           &
  & ' one or more extra might help' )" ) inform%differences_needed
   control%extra_differences = 1 ! use as many differences as required + 1
   k_s = inform%differences_needed + control%extra_differences
!  artifical setup: compute random s_i and then form y_i = Hessian * s_i
   ALLOCATE( S( n, k_s ), Y( n, k_s ), PRECEDENCE( k_s ) )
   CALL RAND_initialize( seed )
   DO k = 1, k_s
     DO i = 1, n  ! choose random S
       CALL RAND_random_real( seed, .FALSE., S( i, k ) )
       CALL RAND_random_real( seed, .FALSE., Y( i, k ) )
       Y( i, k ) = Y( i, k ) * 0.001
     END DO
     Y( : n, k ) = 0.0_wp  ! form Y = H * S
     DO l = 1, nz
```

```
      i = ROW( l ) ; j = COL( l ) ; v = VAL( l )
      Y( i, k ) = Y( i, k ) + v * S( j, k )
      IF ( i /= j ) Y( j, k ) = Y( j, k ) + v * S( i, k )
    END DO
    PRECEDENCE( k ) = k_s - k + 1  ! pick the (s,y) vectors in reverse order
  END DO
!  approximate the Hessian
  CALL SHA_estimate( n, nz, ROW, COL, k_s, S, n, k_s, Y, n, k_s, VAL_est,     &
                     data, control, inform, PRECEDENCE = PRECEDENCE )
  IF ( inform%status /= 0 ) THEN  ! Failure
    WRITE( 6, "( ' return with nonzero status ', I0, ' from SHA_estimate' )" )&
      inform%status ; STOP
  ELSE
    WRITE( 6, "( /, ' Successful run with ', I0,                              &
   &             ' differences, estimated matrix:' )" ) k_s
    DO l = 1, nz
      WRITE( 6, "( ' (row,col,val) = (', I0, ',', I0, ',', ES9.2, ')' )" )    &
       ROW( l ), COL( l ), VAL_est( l )
    END DO
  END IF
  CALL SHA_terminate( data, control, inform ) ! Delete internal workspace
  END PROGRAM GALAHAD_SHA_EXAMPLE
```

The code produces the following output:

```
 2 differences are needed, one or more extra might help

 Successful run with 3 differences, estimated matrix:
 (row,col,val) = (1,1, 1.00E+00)
 (row,col,val) = (1,2, 2.00E+00)
 (row,col,val) = (1,3, 3.00E+00)
 (row,col,val) = (1,4, 4.00E+00)
 (row,col,val) = (1,5, 5.00E+00)
 (row,col,val) = (2,2, 6.00E+00)
 (row,col,val) = (3,3, 7.00E+00)
 (row,col,val) = (4,4, 8.00E+00)
 (row,col,val) = (5,5, 9.00E+00)
```