GALAHAD LMS

USER DOCUMENTATION GALAHAD Optimization Library version 5.0

1 SUMMARY

Given a sequence of vectors $\{\mathbf{s}_k\}$ and $\{\mathbf{y}_k\}$ and scale factors δ_k, **obtain the product of a limited-memory secant approximation $\mathbf{H}_k$ (or its inverse) with a given vector**, using one of a variety of well-established formulae.

ATTRIBUTES — Versions: GALAHAD_LMS_single, GALAHAD_LMS_double. **Uses:** GALAHAD_CLOCK, GALAHAD_SYMBOLS, GALAHAD_SPACE, GALAHAD_LAPACK_interface, GALAHAD_BLAS_interface, GALAHAD_SPECFILE. **Date:** July 2014. **Origin:** N. I. M. Gould, Rutherford Appleton Laboratory. **Language:** Fortran 95 + TR 15581 or Fortran 2003.

2 HOW TO USE THE PACKAGE

The package is available using both single and double precision reals, and either 32-bit or 64-bit integers. Access to the 32-bit integer, single precision version requires the USE statement

```
USE GALAHAD_LMS_single
```

with the obvious substitution GALAHAD_LMS_double, GALAHAD_LMS_single_64 and GALAHAD_LMS_double_64 for the other variants.

If it is required to use more than one of the modules at the same time, the derived types SMT_type, LMS_time_type, LMS_control_type, LMS_inform_type and LMS_data_type (§2.2) and the subroutines LMS_initialize, LMS_setup, LMS_form, LMS_form_shift, LMS_apply, LMS_terminate, (§2.3) and LMS_read_specfile (§2.5) must be renamed on one of the USE statements.

2.1 Real and integer kinds

We use the terms integer and real to refer to the fortran keywords REAL(rp_) and INTEGER(ip_), where rp_ and ip_ are the relevant kind values for the real and integer types employed by the particular module in use. The former are equivalent to default REAL for the single precision versions and DOUBLE PRECISION for the double precision cases, and correspond to rp_ = real32 and rp_ = real64, respectively, as supplied by the fortran iso_fortran_env module. The latter are default (32-bit) and long (64-bit) integers, and correspond to ip_ = int32 and ip_ = int64, respectively, again from the iso_fortran_env module.

2.2 The derived data types

Four derived data types are accessible from the package.

2.2.1 The derived data type for holding control parameters

The derived data type LMS_control_type is used to hold controlling data. Default values may be obtained by calling LMS_initialize (see §2.3.1), while components may also be changed by calling GALAHAD_LMS_read_spec (see §2.5.1). The components of LMS_control_type are:

error is a scalar variable of type INTEGER(ip_), that holds the stream number for error messages. Printing of error messages in LMS_setup, LMS_apply and LMS_terminate is suppressed if error ≤ 0. The default is error = 6.

`out` is a scalar variable of type `INTEGER(ip_)`, that holds the stream number for informational messages. Printing of informational messages in `LMS_setup`, `LMS_apply` is suppressed if `out` < 0. The default is `out = 6`.

`print_level` is a scalar variable of type `INTEGER(ip_)`, 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` $= 1$, a single line of output will be produced for each level of the process. If `print_level` ≥ 2, this output will be increased to provide significant detail of the factorization. The default is `print_level = 0`.

`memory_length` is a scalar variable of type `INTEGER(ip_)`, that is used to specify the maximum number of vectors $\{\mathbf{s}_k\}$ and $\{\mathbf{y}_k\}$ that will be used when building the secant approximation. Any non-positive value will be interpreted as `1`. The default is `memory_length = 10`.

`method` is a scalar variable of type `INTEGER(ip_)`, that is used to specify the limited-memory formula that will be applied. Possible values are

1. A limited-memory BFGS formula will be applied.
2. A limited-memory symmetric rank-one formula will be applied.
3. The inverse of the limited-memory BFGS formula will be applied.
4. The inverse of the shifted limited-memory BFGS formula will be applied. This should be used instead of `%method = 3` whenever a shift is planned.

Any value outside this range will be interpreted as 1. The default is `method = 1`.

`any_method` is a scalar variable of type default `LOGICAL`, that must be set `.TRUE.` if more than one method (see `%method` above) is to be used and `.FALSE.` otherwise. The package will require more storage and may run slower if `any_method` is `.TRUE.`. The default is `any_method = .FALSE.`.

`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 ignoring 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 = ""`.

2.2.2 The derived data type for holding timing information

The derived data type `LMS_time_type` is used to hold elapsed CPU and system clock times for the various parts of the calculation. The components of `LMS_time_type` are:

`total` is a scalar variable of type `REAL(rp_)`, that gives the total CPU time spent in the package.

`setup` is a scalar variable of type `REAL(rp_)`, that gives the CPU time spent setting up the data structures to represent the limited-memory matrix.

`form` is a scalar variable of type `REAL(rp_)`, that gives the CPU time spent forming and updating the limited-memory matrix as new data arrives.

`apply` is a scalar variable of type `REAL(rp_)`, that gives the CPU time spent applying the matrix to given vectors.

`clock_total` is a scalar variable of type `REAL(rp_)`, that gives the total elapsed system clock time spent in the package.

`clock_setup` is a scalar variable of type `REAL(rp_)`, that gives the elapsed system clock time spent setting up the data structures to represent the limited-memory matrix.

`clock_form` is a scalar variable of type `REAL(rp_)`, that gives the elapsed system clock time spent forming and updating the limited-memory matrix as new data arrives.

`clock_apply` is a scalar variable of type `REAL(rp_)`, that gives the elapsed system clock time spent applying the matrix to given vectors. factor **R**.

2.2.3 The derived data type for holding informational parameters

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

`status` is a scalar variable of type `INTEGER(ip_)`, that gives the exit status of the algorithm. See §2.4 for details.

`alloc_status` is a scalar variable of type `INTEGER(ip_)`, 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`.

`length` is a scalar variable of type `INTEGER(ip_)`, that gives the number of pairs $\{\mathbf{s}_k, \mathbf{y}_k\}$ currently used to represent the limited-memory matrix

`updates_skipped` is a scalar variable of type default `LOGICAL`, that will be `.TRUE.` if one or more of the current pairs $\{\mathbf{s}_k, \mathbf{y}_k\}$ has been ignored for stability reasons when building the current limited-memory matrix, and `.FALSE.` otherwise.

`time` is a scalar variable of type `LMS_time_type` whose components are used to hold elapsed CPU and system clock times for the various parts of the calculation (see Section 2.2.2).

2.2.4 The derived data type for holding problem data

The derived data type `LMS_data_type` is used to hold all the data for the problem and the workspace arrays used to construct the multi-level incomplete factorization between calls of `LMS` procedures. This data should be preserved, untouched, from the initial call to `LMS_initialize` to the final call to `LMS_terminate`.

2.3 Argument lists and calling sequences

There are seven procedures for user calls (see §2.5 for further features):

1. The subroutine `LMS_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 `LMS_setup` is called to set up the data structures needed to represent the limited-memory matrix $\mathbf{H}_k$ or its inverse.

3. The subroutine `LMS_form` is called to form the limited-memory matrix $\mathbf{H}_k$ or its inverse as new data $(\mathbf{s}_k, \mathbf{y}_k, \delta_k)$ arrives. The matrix $\mathbf{H}_k + \lambda_k \mathbf{I}$ or its inverse for a specified shift λ_k may be formed instead.

4. The subroutine `LMS_form_shift` is called to update the inverse of the limited-memory matrix $\mathbf{H}_k + \lambda_k \mathbf{I}$ when a new shift λ_k is required.

5. The subroutine `LMS_change_method` is called to build the limited-memory matrices $\mathbf{H}_k$, $\mathbf{H}_k + \lambda_k \mathbf{I}$ or their inverse for a new method from the current data.

6. The subroutine `LMS_apply` is called to form the product $\mathbf{u} = \mathbf{H}_k \mathbf{v}$, $\mathbf{u} = (\mathbf{H}_k + \lambda_k \mathbf{I})\mathbf{v}$, $\mathbf{u} = \mathbf{H}_k^{-1} \mathbf{v}$ or $\mathbf{u} = (\mathbf{H}_k + \lambda_k \mathbf{I})^{-1} \mathbf{v}$ for a given vector $\mathbf{v}$.

7. The subroutine `LMS_terminate` is provided to allow the user to automatically deallocate array components of the private data, allocated by `LMS_setup` at the end of the solution process.

We use square brackets `[ ]` to indicate `OPTIONAL` arguments.

2.3.1 The initialization subroutine

Default values are provided as follows:

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

`data` is a scalar `INTENT(INOUT)` argument of type `LMS_data_type` (see §2.2.4). It is used to hold data about the problem being solved.

`control` is a scalar `INTENT(OUT)` argument of type `LMS_control_type` (see §2.2.1). On exit, `control` contains default values for the components as described in §2.2.1. These values should only be changed after calling `LMS_initialize`.

`inform` is a scalar `INTENT(OUT)` argument of type `LMS_inform_type` (see Section 2.2.3). A successful call to `LMS_initialize` is indicated when the component `status` has the value 0. For other return values of `status`, see Section 2.4.

2.3.2 The subroutine for setting up the required data structures

The data structures needed to represent the limited-memory matrix $\mathbf{H}_k$ or its inverse are set up as follows:

```
CALL LMS_setup( n, data, control, inform )
```

`n` is a scalar `INTENT(IN)` argument of type `INTEGER(ip_)`, that must be set to the dimension of the limited-memory matrix required. **Restriction:** $n \geq 1$.

`data` is a scalar `INTENT(INOUT)` argument of type `LMS_data_type` (see §2.2.4). It is used to hold data about the factors obtained. It must not have been altered **by the user** since the last call to `LMS_initialize`.

`control` is a scalar `INTENT(IN)` argument of type `LMS_control_type` (see §2.2.1). Default values may be assigned by calling `LMS_initialize` prior to the first call to `LMS_setup`.

`inform` is a scalar `INTENT(OUT)` argument of type `LMS_inform_type` (see §2.2.3). A successful call to `LMS_setup` is indicated when the component `status` has the value 0. For other return values of `status`, see §2.4.

2.3.3 The subroutine for updating the limited memory matrix

The required limited memory matrix is updated to accommodate the incoming triple $(\mathbf{s}_k, \mathbf{y}_k, \delta_k)$ as follows:

```
CALL LMS_form( S, Y, delta, data, control, inform[, lambda] )
```

`S` is an `INTENT(IN)` rank-1 array of type `REAL(rp_)` and length at least as large as the value `n` as set on input to `LMS_setup`, whose first `n` components must hold the incoming vector $\mathbf{s}_k$.

`Y` is an `INTENT(IN)` rank-1 array of type `REAL(rp_)` and length at least as large as the value `n` as set on input to `LMS_setup`, whose first `n` components must hold the incoming vector $\mathbf{y}_k$. **Restriction:** the update will be skipped for for limited-memory BFGS methods if the inner product $\mathbf{s}_k^T \mathbf{y}_k <= 0$.

`delta` is an `INTENT(IN) REAL(rp_)` scalar that must hold the value δ_k. **Restriction:** the update will be skipped if `delta` ≤ 0.

`data` is a scalar `INTENT(INOUT)` argument of type `LMS_data_type` (see §2.2.4). It is used to hold data about the factors obtained. It must not have been altered **by the user** since the last call to `LMS_setup`.

`control` is a scalar `INTENT(IN)` argument of type `LMS_control_type` (see §2.2.1). Default values may be assigned by calling `LMS_initialize` prior to the first call to `LMS_setup`.

`inform` is a scalar `INTENT(OUT)` argument of type `LMS_inform_type` (see §2.2.3). A successful call to `LMS_setup` is indicated when the component `status` has the value 0. For other return values of `status`, see §2.4.

`lambda` is an `OPTIONAL`, `INTENT(IN) REAL(rp_)` scalar that if present will be used to specify the shift λ_k that is used by the limited memory methods defined by `control%method = 1, 2` or `4`. **Restriction:** the update will be skipped if `lambda` < 0 for these methods.

2.3.4 The subroutine for shifting the limited-memory matrix

The required limited memory matrix is updated to accommodate the shift λ_k as follows—this call is mandatory when `control%method = 4` if λ_k was not set during the call to `LMS_form`:

```
CALL LMS_form_shift( lambda, data, control, inform )
```

`lambda` is an `INTENT(IN) REAL(rp_)` scalar that must hold the value λ_k. **Restriction:** the update will be skipped if `lambda` < 0 or if `control%method = 3`.

`data` is a scalar `INTENT(INOUT)` argument of type `LMS_data_type` (see §2.2.4). It is used to hold data about the factors obtained. It must not have been altered **by the user** since the last call to `LMS_form`.

`control` is a scalar `INTENT(IN)` argument of type `LMS_control_type` (see §2.2.1). Default values may be assigned by calling `LMS_initialize` prior to the first call to `LMS_setup`.

`inform` is a scalar `INTENT(OUT)` argument of type `LMS_inform_type` (see §2.2.3). A successful call to `LMS_setup` is indicated when the component `status` has the value 0. For other return values of `status`, see §2.4.

2.3.5 The subroutine for changing the method defining the limited-memory matrix

The required limited memory matrix is updated to accommodate the shift λ_k as follows—this call is only permitted if `control%any_method = .TRUE.` was set when `LMS_setup` was originally called:

```
CALL LMS_change_method( data, control, inform, lambda )
```

`data` is a scalar `INTENT(INOUT)` argument of type `LMS_data_type` (see §2.2.4). It is used to hold data about the factors obtained. It must not have been altered **by the user** since the last call to `LMS_form`.

`control` is a scalar `INTENT(IN)` argument of type `LMS_control_type` (see §2.2.1). Default values may be assigned by calling `LMS_initialize` prior to the first call to `LMS_setup`.

`inform` is a scalar `INTENT(OUT)` argument of type `LMS_inform_type` (see §2.2.3). A successful call to `LMS_setup` is indicated when the component `status` has the value 0. For other return values of `status`, see §2.4.

`lambda` is an `OPTIONAL`, `INTENT(IN) REAL(rp_)` scalar that if present will be used to specify the shift λ_k that is used by the limited memory methods defined by `control%method = 1, 2` or `4`. **Restriction:** the update will be skipped if `lambda` < 0 for these methods.

2.3.6 The subroutine for applying the limited-memory formula to a vector

Given the vector **v**, the required limited-memory formula, as specified in the most recent call to LMS_form, LMS_form_shift or LMS_change_method, is applied to **v** as follows:

```
CALL LMS_apply( V, U, data, control, inform )
```

V is a rank-one INTENT(IN) array of type default REAL that must be set on entry to hold the components of the vector **v**.

U is a rank-one INTENT(OUT) array of type default REAL that will be set on exit to the result of applying the required limited-memory formula to **v**.

data is a scalar INTENT(INOUT) argument of type LMS_data_type (see §2.2.4). It is used to hold data about the factors obtained. It must not have been altered **by the user** since the last call to LMS_setup.

control is a scalar INTENT(IN) argument of type LMS_control_type (see §2.2.1). Default values may be assigned by calling LMS_initialize prior to the first call to LMS_setup.

inform is a scalar INTENT(OUT) argument of type LMS_inform_type (see §2.2.3). A successful call to LMS_apply is indicated when the component status has the value 0. For other return values of status, see §2.4.

2.3.7 The termination subroutine

All previously allocated arrays are deallocated as follows:

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

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

control is a scalar INTENT(IN) argument of type LMS_control_type exactly as for LMS_setup.

inform is a scalar INTENT(OUT) argument of type LMS_inform_type exactly as for LMS_setup. Only the component status will be set on exit, and a successful call to LMS_terminate is indicated when this component status has the value 0. For other return values of status, see §2.4.

2.4 Warning and error messages

A negative value of inform%status on exit from LMS_setup, LMS_apply or LMS_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 of the restrictions n > 0, delta > 0, lambda ≥ 0 or $\mathbf{s}^T \mathbf{y} > 0$ has been violated and the update has been skipped.

-10. The matrix cannot be built from the current vectors $\{\mathbf{s}_k\}$ and $\{\mathbf{y}_k\}$ and values δ_k and λ_k and the update has been skipped.

-31. A call to subroutine LMS_apply has been made without a prior call to LMS_form_shift or LMS_form with lambda specified when control%method = 4, or LMS_form_shift has been called when control%method = 3, or LMS_change_method has been called after control%any_method = .FALSE. was specified when calling LMS_setup.

2.5 Further features

In this section, we describe an alternative means of setting control parameters, that is components of the variable control of type LMS_control_type (see §2.2.1), by reading an appropriate data specification file using the subroutine LMS_read_specfile. This facility is useful as it allows a user to change LMS control parameters without editing and recompiling programs that call LMS.

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 LMS_read_specfile must start with a "BEGIN LMS" command and end with an "END" command. The syntax of the specfile is thus defined as follows:

```
( .. lines ignored by LMS_read_specfile .. )
  BEGIN LMS
     keyword    value
     .......    .....
     keyword    value
  END
( .. lines ignored by LMS_read_specfile .. )
```

where keyword and value are two strings separated by (at least) one blank. The "BEGIN LMS" 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 LMS SPECIFICATION
```

and

```
END LMS SPECIFICATION
```

are acceptable. Furthermore, between the "BEGIN LMS" 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 LMS_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 LMS_read_specfile.

2.5.1 To read control parameters from a specification file

Control parameters may be read from a file as follows:

```
CALL LMS_read_specfile( control, device )
```

`control` is a scalar `INTENT(INOUT)` argument of type `LMS_control_type` (see §2.2.1). Default values should have already been set, perhaps by calling `LMS_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 §2.2.1) 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
`limited-memory-length`	`%memory_length`	integer
`limited-memory-method`	`%method`	integer
`allow-any-method`	`%any_method`	logical
`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 `INTEGER(ip_)`, 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`.

2.6 Information printed

If `control%print_level` is positive, information about the progress of the algorithm will be printed on unit `control-%out`. If `control%print_level` ≥ 1, statistics concerning the formation of **R** as well as warning and error messages will be reported.

3 GENERAL INFORMATION

Use of common: None.

Workspace: Provided automatically by the module.

Other routines called directly: None.

Other modules used directly: `LMS` calls the GALAHAD packages `GALAHAD_CLOCK`, `GALAHAD_SYMBOLS`, `GALAHAD_SPACE`, `GALAHAD_LAPACK_interface`, `GALAHAD_BLAS_interface` and `GALAHAD_SPECFILE`,

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

Restrictions: `n` > 0, `delta` > 0, `lambda` ≥ 0.

Portability: ISO Fortran 95 + TR 15581 or Fortran 2003. The package is thread-safe.

4 METHOD

Given a sequence of vectors $\{\mathbf{s}_k\}$ and $\{\mathbf{y}_k\}$ and scale factors δ_k, a limited-memory secant approximation $\mathbf{H}_k$ is chosen so that $\mathbf{H}_{\max(k-m,0)} = \delta_k \mathbf{I}$, $\mathbf{H}_{k-j}\mathbf{s}_{k-j} = \mathbf{y}_{k-j}$ and $\|\mathbf{H}_{k-j+1} - \mathbf{H}_{k-j}\|$ is "small" for $j = \min(k-1, m-1), \ldots, 0$. Different ways of quantifying "small" distinguish different methods, but the crucial observation is that it is possible to construct $\mathbf{H}_k$ quickly from $\{\mathbf{s}_k\}$, $\{\mathbf{y}_k\}$ and δ_k, and to apply it and its inverse to a given vector $\mathbf{v}$. It is also possible to apply similar formulae to the "shifted" matrix $\mathbf{H}_k + \lambda_k \mathbf{I}$ that occurs in trust-region methods.

References:

The basic methods are those given by

R. H. Byrd, J. Nocedal and R. B. Schnabel (1994) "Representations of quasi-Newton matrices and their use in limited memory methods". Mathematical Programming **63**(2) 129–156,

with obvious extensions.

5 EXAMPLE OF USE

Suppose that we generate random vectors $\{\mathbf{s}_k\}$ and $\{\mathbf{y}_k\}$ and scale factors δ_k, that we build the limited-memory BFGS matrix $\mathbf{H}_k$ and its inverse $\mathbf{H}_k^{-1}$ and that we apply $\mathbf{H}_k$ and then $\mathbf{H}_k^{-1}$ to a given vector $\mathbf{v}$. Suppose further, that at some stage, we instead apply the inverse $(\mathbf{H}_k + \lambda_k \mathbf{I})^{-1}$ with $\lambda_k = 0$. Then we may use the following code; of course since we have the identities $\mathbf{v} = \mathbf{H}_k^{-1}(\mathbf{H}_k \mathbf{v})$ and $\mathbf{v} = (\mathbf{H}_k + \lambda_k \mathbf{I})^{-1}(\mathbf{H}_k \mathbf{v})$ when $\lambda_k = 0$, we expect to recover the original $\mathbf{v}$ after every step:

```
! THIS VERSION: GALAHAD 2.6 - 12/06/2014 AT 15:30 GMT.
   PROGRAM GALAHAD_LMS_example
   USE GALAHAD_LMS_double                    ! double precision version
   USE GALAHAD_rand_double
   IMPLICIT NONE
   INTEGER, PARAMETER :: wp = KIND( 1.0D+0 ) ! set precision
   INTEGER, PARAMETER :: n = 5, m = 3
   TYPE ( LMS_data_type ) :: data, data2
   TYPE ( LMS_control_type ) :: control, control2
   TYPE ( LMS_inform_type ) :: inform, inform2
   REAL ( KIND = wp ), DIMENSION( n ) :: S, Y, U, V
   INTEGER :: iter, fail
   REAL ( KIND = wp ) :: delta, lambda
   TYPE ( RAND_seed ) :: seed
   CALL RAND_initialize( seed ) ! Initialize the random generator word
   CALL LMS_initialize( data, control, inform ) !  initialize data
   control%memory_length = m ! set the memory length
   control2 = control
   control%method = 1 ! start with L-BFGS
   CALL LMS_setup( n, data, control, inform )
   control2%method = 3 ! then inverse L-BFGS
   control2%any_method =.TRUE. ! allow the 2nd update to change method
   CALL LMS_setup( n, data2, control2, inform2 )
   fail = 0 ! count the failures
   DO iter = 1, 5 * n
     IF ( iter == 3 * n ) THEN ! switch to inverse shifted L-BFGS
       CALL LMS_setup( n, data, control, inform )
       control2%method = 4
       CALL LMS_setup( n, data2, control2, inform2 )
```

```
    END IF
    CALL RAND_random_real( seed, .FALSE., S )  ! pick random S, Y and delta
    CALL RAND_random_real( seed, .FALSE., Y )
    IF ( DOT_PRODUCT( S, Y ) < 0.0_wp ) Y = - Y ! ensure that S^T Y is positive
    CALL RAND_random_real( seed, .TRUE., delta )
    CALL LMS_form( S, Y, delta, data, control, inform ) ! update the model
    IF ( inform%status /= 0 ) THEN
      WRITE( 6, "( ' update error, status = ', I0 )" ) inform%status
      fail = fail + 1 ; CYCLE
    END IF
    V = 1.0_wp ! form the first product with the vector ones
    CALL LMS_apply( V, U, data, control, inform ) ! form the required product
    IF ( inform%status /= 0 ) THEN
      WRITE( 6, "( ' apply error, status = ', I0 )" ) inform%status
      fail = fail + 1 ; CYCLE
    END IF
    CALL LMS_form( S, Y, delta, data2, control2, inform2 ) ! update model 2
    IF ( inform2%status /= 0 ) THEN
      WRITE( 6, "( ' update error, status = ', I0 )" ) inform2%status
      fail = fail + 1 ; CYCLE
    END IF
    IF ( control2%method == 4 ) THEN
      lambda = 0.0_wp ! apply the shifted L_BFGS (inverse) with zero shift
      CALL LMS_form_shift( lambda, data2, control2, inform2 )
      IF ( inform2%status /= 0 ) THEN
        WRITE( 6, "( ' update error, status = ', I0 )" ) inform2%status
        fail = fail + 1 ; CYCLE
      END IF
    END IF
! note, the preceeding two calls could have been condensed as
!   CALL LMS_form( S, Y, delta, data2, control2, inform2, lambda = 0.0_wp )
    CALL LMS_apply( U, V, data2, control2, inform2 ) ! form the new product
    IF ( inform2%status /= 0 ) THEN
      WRITE( 6, "( ' apply error, status = ', I0 )" ) inform2%status
      fail = fail + 1 ; CYCLE
    END IF
    IF ( MAXVAL( ABS( V - 1.0_wp ) ) > 0.00001_wp ) fail = fail + 1
  END DO
  IF ( fail == 0 ) THEN  ! check for overall success
    WRITE( 6, "( ' no failures ' )" )
  ELSE
    WRITE( 6, "( I0, ' failures ' )" ) fail
  END IF
  CALL LMS_terminate( data, control, inform )  !  delete internal workspace
  CALL LMS_terminate( data2, control2, inform2 )
  END PROGRAM GALAHAD_LMS_example
```

This produces the following output:

```
 no failures
```