



Science and
Technology
Facilities Council



GALAHAD

SNLS

USER DOCUMENTATION

GALAHAD Optimization Library version 5.5

1 SUMMARY

This package uses a **regularization method to find a (local) minimizer of a differentiable weighted sum-of-squares objective function**

$$f(\mathbf{x}) \stackrel{\text{def}}{=} \frac{1}{2} \sum_{i=1}^{m_r} w_i r_i^2(\mathbf{x}) \equiv \frac{1}{2} \|\mathbf{r}(\mathbf{x})\|_W^2, \quad (1.1)$$

where the variables $\mathbf{x}$ are required to lie within the **regular simplex**

$$\mathbf{e}^T \mathbf{x} = 1 \text{ and } \mathbf{x} \geq 0,$$

or an intersection of **multiple non-overlapping regular simplices**

$$\mathbf{e}_{C_i}^T \mathbf{x}_{C_i} = 1 \text{ and } \mathbf{x}_{C_i} \geq 0 \text{ for } i = 1, \dots, m_c, \quad (1.2)$$

of many variables $\mathbf{x}$, the positive weights $w_i, i = 1, \dots, m_c$ are given, and the vector $\mathbf{v}_C$ is made up of those entries of $\mathbf{v}$ indexed by the set C , the index sets of **cohorts** $C_i \subseteq \{1, \dots, n\}$ for which $C_i \cap C_j = \emptyset$ for $1 \leq i, j \leq m_c, i \neq j$. The method offers the choice of projected-gradient and interior-point iteration to solve the key regularization subproblems, and is most suitable for large problems. First derivatives of the **residual function** $\mathbf{r}(\mathbf{x})$ must be available, either directly as the Jacobian matrix $\mathbf{J}_r(\mathbf{x}) = \nabla \mathbf{r}(\mathbf{x})$ or via the action of this matrix (and its transpose) on specified vectors.

ATTRIBUTES — Versions: GALAHAD_SNLS_single, GALAHAD_SNLS_double. **Uses:** GALAHAD_CLOCK, GALAHAD_SYMBOLS, GALAHAD_NLPT, GALAHAD_USERDATA, GALAHAD_REVERSE, GALAHAD_SPECFILE, GALAHAD_QPT, GALAHAD_SLLS, GALAHAD_SLLSB, GALAHAD_SPACE, GALAHAD_MOP, GALAHAD_NORMS and GALAHAD_STRING. **Date:** March 2026. **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 with single, double and (if available) quadruple 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_SNLS_single
```

with the obvious substitution GALAHAD_SNLS_double, GALAHAD_SNLS_quadruple, GALAHAD_SNLS_single_64, GALAHAD_SNLS_double_64 and GALAHAD_SNLS_quadruple_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, USERDATA_type, REVERSE_type, SNLS_time_type, SNLS_control_type, SNLS_inform_type, SNLS_data_type and NLPT_problem_type, (Section 2.4) and the subroutines SNLS_initialize, SNLS_solve, SNLS_terminate, (Section 2.5) and SNLS_read_specfile (Section 2.9) must be renamed on one of the USE statements.

2.1 Basic terminology

The algorithm used is iterative. From the current best estimate of the minimizer $\mathbf{x}_k$, a trial improved point $\mathbf{x}_k + \mathbf{s}_k$ is sought. The correction $\mathbf{s}_k$ is chosen to improve a model $m_k(\mathbf{s})$ of the objective function $f(\mathbf{x}_k + \mathbf{s})$, built around $\mathbf{x}_k$, within the simplex constraint set $C(\mathbf{x}_k + \mathbf{s})$ for which

$$C(\mathbf{x}) \stackrel{\text{def}}{=} \{\mathbf{x} \mid (1.2) \text{ holds}\}.$$

All use is subject to the conditions of a BSD-3-Clause License.
See <https://www.galahad.rl.ac.uk/download/> for full details.

The model is the sum of two basic components, a suitable approximation $t_k(\mathbf{s})$ of $f(\mathbf{x}_k + \mathbf{s})$, and a regularization term $\frac{\sigma_k}{p} \|\mathbf{s}\|^p$ involving a weight σ_k and a power p that is included to prevent large corrections. The weight σ_k is adjusted as the algorithm progresses to ensure convergence.

The model $t_k(\mathbf{s})$ is a truncated Taylor-series approximation, and this relies on being able to compute or estimate derivatives of $\mathbf{r}(\mathbf{x})$. Various models are provided, and each has different derivative requirements. We denote the m_r by n **residual/observation Jacobian** $\mathbf{J}_r(\mathbf{x})$ as the matrix whose i, j -th component

$$\mathbf{J}_r(\mathbf{x})_{i,j} \stackrel{\text{def}}{=} \partial r_i(\mathbf{x}) / \partial x_j \text{ for } i = 1, \dots, m_r \text{ and } j = 1, \dots, n.$$

For a given m_r -vector $\mathbf{y}$, the **weighted residual Hessian** is the sum

$$\mathbf{H}(\mathbf{x}, \mathbf{y}) \stackrel{\text{def}}{=} \sum_{\ell=1}^m y_\ell \mathbf{H}_\ell(\mathbf{x}), \text{ where } \mathbf{H}_\ell(\mathbf{x})_{i,j} \stackrel{\text{def}}{=} \partial^2 r_\ell(\mathbf{x}) / \partial x_i \partial x_j \text{ for } i, j = 1, \dots, n$$

is the Hessian of $r_\ell(\mathbf{x})$. The models $t_k(\mathbf{s})$ provided are,

1. the Gauss-Newton approximation $\frac{1}{2} \|\mathbf{r}(\mathbf{x}_k) + \mathbf{J}_r(\mathbf{x}_k)\mathbf{s}\|_{\mathbf{W}}^2$,
2. the Newton (second-order Taylor) approximation $f(\mathbf{x}_k) + \mathbf{g}(\mathbf{x}_k)^T \mathbf{s} + \frac{1}{2} \mathbf{s}^T [\mathbf{J}_r^T(\mathbf{x}_k) \mathbf{W} \mathbf{J}_r(\mathbf{x}_k) + \mathbf{H}(\mathbf{x}_k, \mathbf{W} \mathbf{r}(\mathbf{x}_k))] \mathbf{s}$,

where $\mathbf{W}$ is the diagonal matrix of weights w_i , $i = 1, \dots, m_r$.

Access to a particular model requires that the user is either able to provide the derivatives needed (“**matrix available**”) or that the products of these derivatives (and their transposes) with specified vectors are possible (“**matrix free**”).

The method makes considerable use of efficient available methods for computing the projection $P[\mathbf{x}]$, that is defined to be the closest (two-norm) point of a given $\mathbf{x}$ to the simplex constraint set $\mathcal{C}(\mathbf{x})$.

2.2 Matrix storage formats

The m_r by n unsymmetric matrix $\mathbf{J}_r(\mathbf{x})$ and the symmetric n by n matrix $\mathbf{H}(\mathbf{x}, \mathbf{y})$ (as required and when available) may be stored in a variety of input formats.

2.2.1 Dense storage format

The matrix $\mathbf{J}_r$ is stored as a compact dense matrix by rows, that is, the values of the entries of each row in turn are stored in order within an appropriate real one-dimensional array. Component $n * (i - 1) + j$ of the storage array `Jr%val` will hold the value $\mathbf{J}_{ri,j}$ for $i = 1, \dots, m_r$, $j = 1, \dots, n$. Since $\mathbf{H}$ is symmetric, only the lower triangular part (that is the part $\mathbf{H}_{ij}$ for $1 \leq j \leq i \leq n$) should be stored. In this case the lower triangle will be stored by rows, that is component $i * (i - 1) / 2 + j$ of the storage array `H%val` will hold the value $\mathbf{H}_{ij}$ (and, by symmetry, $\mathbf{H}_{ji}$) for $1 \leq j \leq i \leq n$.

2.2.2 Dense column-wise storage format

The matrix $\mathbf{J}_r$ is stored as a compact dense matrix by columns, that is, the values of the entries of each column in turn are stored in order within an appropriate real one-dimensional array. Component $m_r * (j - 1) + i$ of the storage array `Jr%val` will hold the value $\mathbf{J}_{ri,j}$ for $i = 1, \dots, m_r$, $j = 1, \dots, n$.

2.2.3 Sparse co-ordinate storage format

Only the nonzero entries of the matrices are stored. For the l -th entry of $\mathbf{J}_r$, its row index i , column index j and value $\mathbf{J}_{ri,j}$ are stored in the l -th components of the integer arrays `Jr%row`, `Jr%col` and real array `Jr%val`. The order is unimportant, but the total number of entries `Jr%ne` is required. The same scheme is applicable to $\mathbf{H}$ (thus requiring integer arrays `H%row`, `H%col`, a real array `H%val`, and an integer value `H%ne`), except that only the entries in the lower triangle should be stored.

All use is subject to the conditions of a BSD-3-Clause License.
See <https://www.galahad.rl.ac.uk/download/> for full details.

2.2.4 Sparse row-wise storage format

Again only the nonzero entries are stored, but this time they are ordered so that those in row i appear directly before those in row $i + 1$. For the i -th row of $\mathbf{J}_r$, the i -th component of the integer array `Jr%ptr` holds the position of the first entry in this row, while `Jr%ptr (mr + 1)` holds the total number of entries plus one. The column indices j and values $\mathbf{J}_{r,j}$ of the entries in the i -th row are stored in components $l = \text{Jr\%ptr}(i), \dots, \text{Jr\%ptr}(i + 1) - 1$ of the integer array `Jr%col`, and real array `Jr%val`, respectively. The same scheme is applicable to $\mathbf{H}$ (thus requiring integer arrays `H%ptr`, `H%col`, and a real array `H%val`), except that only the entries in the lower triangle should be stored.

For sparse matrices, this scheme almost always requires less storage than its predecessor.

2.2.5 Sparse column-wise storage format

For the matrix $\mathbf{J}_r$, once again only the nonzero entries are stored, but this time they are ordered so that those in column j appear directly before those in column $j + 1$. For the j -th column of $\mathbf{J}_r$, the j -th component of the integer array `Jr%ptr` holds the position of the first entry in this column, while `Jr%ptr (n + 1)` holds the total number of entries plus one. The row indices i and values $\mathbf{J}_{r,i}$ of the entries in the j -th column are stored in components $l = \text{Jr\%ptr}(j), \dots, \text{Jr\%ptr}(j + 1) - 1$ of the integer array `Jr%row`, and real array `Jr%val`, respectively.

2.2.6 Diagonal storage format

If $\mathbf{H}$ is diagonal (i.e., $\mathbf{H}_{ij} = 0$ for all $1 \leq i \neq j \leq n$) only the diagonal entries $\mathbf{H}_{ii}$ for $1 \leq i \leq n$ need be stored, and the first n components of the array `H%val` may be used for the purpose. There is no sensible equivalent for the non-square matrix $\mathbf{J}_r$.

2.3 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, `DOUBLE PRECISION` for the double precision cases and quadruple-precision if 128-bit reals are available, and correspond to `rp_ = real32`, `rp_ = real64` and `rp_ = real128` respectively as defined 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.

2.4 The derived data types

Eight derived data types are accessible from the package.

2.4.1 The derived data type for holding matrices

The derived data type `SMT_TYPE` is used to hold any unsymmetric matrix $\mathbf{A}$ or symmetric one $\mathbf{H}$ if these are available. The components of `SMT_TYPE` used here are:

`m` is a scalar component of type `INTEGER(ip_)`, that holds the row dimension of the matrix.

`n` is a scalar component of type `INTEGER(ip_)`, that holds the column dimension of the matrix.

`ne` is a scalar variable of type `INTEGER(ip_)`, that holds the number of matrix entries.

`type` is a rank-one allocatable array of type default `CHARACTER`, that is used to indicate the matrix storage scheme used. Its precise length and content depends on the type of matrix to be stored (see §2.4.2).

All use is subject to the conditions of a BSD-3-Clause License.
See <https://www.galahad.rl.ac.uk/download/> for full details.

- `val` is a rank-one allocatable array of type `REAL(rp_)` and dimension at least `ne`, that holds the values of the entries. If the matrix is symmetric, each pair of off-diagonal entries $h_{ij} = h_{ji}$ is represented as a single entry (see §2.2.1–2.2.6). Any duplicated entries that appear in the sparse co-ordinate or row-wise schemes will be summed.
- `row` is a rank-one allocatable array of type `INTEGER(ip_)`, and dimension at least `ne`, that may hold the row indices of the entries. (see §2.2.3 and §2.2.5).
- `col` is a rank-one allocatable array of type `INTEGER(ip_)`, and dimension at least `ne`, that may hold the column indices of the entries (see §2.2.3–2.2.4).
- `ptr` is a rank-one allocatable array of type `INTEGER(ip_)`, and dimension at least `m + 1`, that may hold the pointers to the first entry in each row, or dimension at least `n + 1`, that may hold the pointers to the first entry in each column, as required (see §2.2.4–2.2.5).

2.4.2 The derived data type for holding the problem

The derived data type `NLPT_problem_type` is used to hold the problem. The relevant components of `NLPT_problem_type` are:

- `n` is a scalar variable of type `INTEGER(ip_)`, that holds the number of optimization variables, n .
- `m_r` is a scalar variable of type `INTEGER(ip_)`, that holds the number of residual functions, m_r .
- `m_c` is a scalar variable of type `INTEGER(ip_)`, that holds the number of cohorts, m_c .
- `Jr` is scalar variable of type `SMT_TYPE` that holds the Jacobian matrix $\mathbf{J}_r(\mathbf{x})$ (if it is available). The following components are used here:

`Jr%type` is an allocatable array of rank one and type default `CHARACTER`, that is used to indicate the storage scheme used. If the dense-by-row storage scheme (see Section 2.2.1) is used, the first five components of `Jr%type` must contain the string `DENSE` or (or the first thirteen contain `DENSE_BY_ROWS`), while if a dense-by-column storage scheme (see Section 2.2.2) is desired, the first sixteen components must contain the string `DENSE_BY_COLUMNS`. For the sparse co-ordinate scheme (see Section 2.2.3), the first ten components of `Jr%type` must contain the string `COORDINATE`, for the sparse row-wise storage scheme (see Section 2.2.4), its first fourteen components must contain the string `SPARSE_BY_ROWS`. and for the sparse column-wise storage scheme (see Section 2.2.5), the first seventeen components of `Jr%type` must contain the string `SPARSE_BY_COLUMNS`.

For convenience, the procedure `SMT_put` may be used to allocate sufficient space and insert the required keyword into `Jr%type`. For example, if `nlp` is of derived type `SNLS_problem_type` and involves a Jacobian we wish to store using the co-ordinate scheme, we may simply

```
CALL SMT_put( nlp%J%type, 'COORDINATE' )
```

See the documentation for the GALAHAD package `SMT` for further details on the use of `SMT_put`.

`Jr%ne` is a scalar variable of type `INTEGER(ip_)`, that holds the number of entries in $\mathbf{J}_r$ in the sparse co-ordinate storage scheme (see Section 2.2.3). It need not be set for any of the other four schemes.

`Jr%val` is a rank-one allocatable array of type `REAL(rp_)`, that holds the values of the entries of the Jacobian matrix $\mathbf{J}_r$ in any of the storage schemes discussed in Section 2.2.

`Jr%row` is a rank-one allocatable array of type `INTEGER(ip_)`, that holds the row indices of $\mathbf{J}_r$ in the sparse co-ordinate storage scheme (see Section 2.2.3), or the sparse column-wise (see Section 2.2.5) storage scheme. It need not be allocated for the other dense sparse schemes.

All use is subject to the conditions of a BSD-3-Clause License.
 See <https://www.galahad.rl.ac.uk/download/> for full details.

`Jr%col` is a rank-one allocatable array variable of type `INTEGER(ip_)`, that holds the column indices of $\mathbf{J}_r$ in either the sparse co-ordinate (see Section 2.2.3), or the sparse row-wise (see Section 2.2.4) storage scheme. It need not be allocated for the other dense sparse schemes.

`Jr%ptr` is a rank-one allocatable array of dimension `m_r+1` and type `INTEGER(ip_)`, that holds the starting position of each row of $\mathbf{J}_r$, as well as the total number of entries plus one, in the sparse row-wise storage scheme (see Section 2.2.4). It should be of dimension `n+1` and type `INTEGER(ip_)` to hold the starting position of each column of $\mathbf{J}_r$, as well as the total number of entries plus one, in the sparse column-wise scheme (see Section 2.2.5). It need not be allocated when the other schemes are used.

`H` is scalar variable of type `SMT_TYPE` that may hold the weighted Hessian matrix $\mathbf{H}(\mathbf{x}, \mathbf{y})$ (if it is available). The following components are used here:

`H%type` is an allocatable array of rank one and type default `CHARACTER`, that is used to indicate the storage scheme used. If the dense storage scheme (see Section 2.2.1) is used, the first five components of `H%type` must contain the string `DENSE`. For the sparse co-ordinate scheme (see Section 2.2.3), the first ten components of `H%type` must contain the string `COORDINATE`, for the sparse row-wise storage scheme (see Section 2.2.4), the first fourteen components of `H%type` must contain the string `SPARSE_BY_ROWS`, and for the diagonal storage scheme (see Section 2.2.6), the first eight components of `H%type` must contain the string `DIAGONAL`.

For convenience, the procedure `SMT_put` may be used to allocate sufficient space and insert the required keyword into `H%type`. For example, if `nlp` is of derived type `SNLS_problem_type` and involves a Hessian we wish to store using the co-ordinate scheme, we may simply

```
CALL SMT_put( nlp%H%type, 'COORDINATE' )
```

See the documentation for the GALAHAD package `SMT` for further details on the use of `SMT_put`.

`H%ne` is a scalar variable of type `INTEGER(ip_)`, that holds the number of entries in the **lower triangular** part of $\mathbf{H}$ in the sparse co-ordinate storage scheme (see Section 2.2.3). It need not be set for any of the other three schemes.

`H%val` is a rank-one allocatable array of type `REAL(rp_)`, that holds the values of the entries of the **lower triangular** part of the Hessian matrix $\mathbf{H}$ in any of the storage schemes discussed in Section 2.2.

`H%row` is a rank-one allocatable array of type `INTEGER(ip_)`, that holds the row indices of the **lower triangular** part of $\mathbf{H}$ in the sparse co-ordinate storage scheme (see Section 2.2.3). It need not be allocated for any of the other three schemes.

`H%col` is a rank-one allocatable array variable of type `INTEGER(ip_)`, that holds the column indices of the **lower triangular** part of $\mathbf{H}$ in either the sparse co-ordinate (see Section 2.2.3), or the sparse row-wise (see Section 2.2.4) storage scheme. It need not be allocated when the dense or diagonal storage schemes are used.

`H%ptr` is a rank-one allocatable array of dimension `n+1` and type `INTEGER(ip_)`, that holds the starting position of each row of the **lower triangular** part of $\mathbf{H}$, as well as the total number of entries plus one, in the sparse row-wise storage scheme (see Section 2.2.4). It need not be allocated when the other schemes are used.

`COHORT` is a rank-one allocatable array of dimension `n` and type `INTEGER(ip_)`, that specifies with which cohort variable x_j , $j = 1, \dots, n$, is associated. If variable x_j is associated with cohort C_i , $1 \leq i \leq m$, `COHORT(j)` should be set to `i`, while if x_j is unconstrained `COHORT(j) = 0` should be assigned. At least one value `COHORT(j)` for $j = 1, \dots, n$ is expected to take the value `i` for every $1 \leq i \leq m$, that is no empty cohorts are allowed. If `COHORT` is not allocated, a single cohort containing all of the variables will be used instead, i.e., a single unit simplex is employed.

`W` is a rank-one allocatable array of dimension `m_r` and type default `REAL(rp_)`, that holds the values of the weights, $\mathbf{w}$. The i -th component of `W`, $i = 1, \dots, m_r$, contains w_i . If `W` is not allocated, weights of one will be assumed.

All use is subject to the conditions of a BSD-3-Clause License.
See <https://www.galahad.rl.ac.uk/download/> for full details.

- X** is a rank-one allocatable array of dimension n and type `REAL(rp_)`, that holds the values $\mathbf{x}$ of the optimization variables. The j -th component of $\mathbf{X}$, $j = 1, \dots, n$, contains x_j .
- Y** is a rank-one allocatable array of dimension `m_c` (or 1 if a single unit simplex is implied) and type default `REAL(rp_)`, that holds the values $\mathbf{y}$ of estimates of the Lagrange multipliers corresponding to the equality constraint for each cohort (see Section 4). The i -th component of $\mathbf{Y}$, $i = 1, \dots, m_c$, contains y_i .
- Z** is a rank-one allocatable array of dimension n and type default `REAL(rp_)`, that holds the values $\mathbf{z}$ of estimates of the dual variables corresponding to the non-negativity constraints (see Section 4). The j -th component of $\mathbf{Z}$, $j = 1, \dots, n$, contains z_j .
- R** is a rank-one allocatable array of dimension `o` and type `REAL(rp_)`, that holds the residual values $\mathbf{r}(\mathbf{x})$ at the point $\mathbf{x}$. The i -th component of $\mathbf{R}$, $i = 1, \dots, o$, contains $r_i(\mathbf{x})$.
- G** is a rank-one allocatable array of dimension m and type `REAL(rp_)`, that holds the gradient $g(\mathbf{x}) = \nabla_{\mathbf{x}} f(\mathbf{x})$ of the objective function at the point $\mathbf{x}$. The j -th component of $\mathbf{G}$, $i = 1, \dots, n$, contains $\partial f(\mathbf{x}) / \partial x_j$.
- X_status** is a rank-one allocatable array argument of dimension n and type `INTEGER(ip_)`, that indicates which of the non-negativity constraints are in the current active set. Possible values for `X_status(j)`, $j = 1, \dots, n$, and their meanings are
- `< 0` the j -th non-negativity constraint is in the active set, at the value zero, and
 - `0` the j -th non-negativity constraint is not in the active set, or the associated variable is unconstrained.

2.4.3 The derived data type for holding control parameters

The derived data type `SNLS_control_type` is used to hold controlling data. Default values may be obtained by calling `SNLS_initialize` (see Section 2.5.1), while components may also be changed by calling `GALAHAD.SNLS_read_spec` (see Section 2.9.1). The derived type `SNLS_subproblem_control_type` comprises all of the components of `SNLS_control_type` except for the last (i.e., `subproblem_control`). The components of `SNLS_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 `SNLS_solve` and `SNLS_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 `SNLS_solve` 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 iteration of the process. If `print_level ≥ 2`, this output will be increased to provide significant detail of each iteration. The default is `print_level = 0`.
- start_print** is a scalar variable of type `INTEGER(ip_)`, that specifies the first iteration for which printing will occur in `SNLS_solve`. If `start_print` is negative, printing will occur from the outset. The default is `start_print = -1`.
- stop_print** is a scalar variable of type `INTEGER(ip_)`, that specifies the last iteration for which printing will occur in `SNLS_solve`. If `stop_print` is negative, printing will occur once it has been started by `start_print`. The default is `stop_print = -1`.
- print_gap** is a scalar variable of type `INTEGER(ip_)`. Once printing has been started, output will occur once every `print_gap` iterations. If `print_gap` is no larger than 1, printing will be permitted on every iteration. The default is `print_gap = 1`.

All use is subject to the conditions of a BSD-3-Clause License.
See <https://www.galahad.rl.ac.uk/download/> for full details.

`maxit` is a scalar variable of type `INTEGER(ip_)`, that holds the maximum number of iterations which will be allowed in `SNLS_solve`. The default is `maxit = 1000`.

`alive_unit` is a scalar variable of type `INTEGER(ip_)`. If `alive_unit > 0`, a temporary file named `alive_file` (see below) will be created on stream number `alive_unit` on initial entry to `GALAHAD_SNLS_solve`, and execution of `GALAHAD_SNLS_solve` will continue so long as this file continues to exist. Thus, a user may terminate execution simply by removing the temporary file from this unit. If `alive_unit ≤ 0`, no temporary file will be created, and execution cannot be terminated in this way. The default is `alive_unit = 40`.

`jacobian_available` is a scalar variable of type `INTEGER(ip_)`, that specifies the availability of the residual Jacobian. Possible values are:

≥ 2 The Jacobian is available.

1 The Jacobian is not available explicitly but its effect may be accessed by matrix-vector products, i.e., it is “matrix-free”.

≤ 0 The Jacobian is not available either explicitly or via matrix-vector products.

The default is `jacobian_available = 1`.

`subproblem_solver` is a scalar variable of type `INTEGER(ip_)`, that specifies the method used to solve the crucial step-determination subproblem. Possible values are:

≤ 1 A projected-gradient method using `GALAHAD_SLLS` will be used.

2 An interior-point method using `GALAHAD_SLLSB` will be used.

≥ 3 An interior-point method will initially be used, but a switch to a projected-gradient method will occur when sufficient progress has occurred (see `%stop_pg_switch`).

The default is `subproblem_solver = 1`.

`non_monotone` is a scalar variable of type `INTEGER(ip_)`, that specifies the history-length for non-monotone descent strategy. Any non-positive value results in standard monotone descent, for which merit function improvement occurs at each iteration. There are often definite advantages in using a non-monotone strategy with a modest history, since the occasional local increase in the merit function may enable the algorithm to move across (gentle) “ripples” in the merit function surface. However, we do not usually recommend large values of `non_monotone`. The default is `non_monotone = 1`.

`weight_update_strategy` is a scalar variable of type `INTEGER(ip_)`, that specifies the way in which the regularization weight will be adjusted at the end of each iteration. Possible values are:

1 the traditional acceptance and rejection strategy as described below.

2 the traditional strategy, except that a zero weight will be tried first after a very successful step.

3 a more sophisticated strategy that mimics that proposed for trust-region methods by Gould, Porcelli and Toint.

Any other value will be considered as if `weight_update_strategy = 1`, and this is the default.

`stop_r_absolute` is a scalar variable of type `REAL(rp_)`, that is used to specify the maximum permitted norm of the residual, $\|\mathbf{r}(\mathbf{x})\|_{\mathbf{w}}$, (see Section 4) at the estimate of the solution sought. Any negative value will be recorded as zero. The default is `stop_r_absolute = 10-6`.

`stop_r_relative` is a scalar variable of type `REAL(rp_)`, that is used to specify the largest relative reduction in the two-norm of the residual, $\|\mathbf{r}(\mathbf{x})\|_{\mathbf{w}}$, that will be permitted (see Section 4) at the estimate of the solution sought compared to that at the initial point. Any negative value will be recorded as zero. The default is `stop_r_relative = 0.0`.

All use is subject to the conditions of a BSD-3-Clause License.
See <https://www.galahad.rl.ac.uk/download/> for full details.

`stop_pg_absolute` is a scalar variable of type `REAL(rp_)`, that is used to specify the maximum permitted norm of the projected gradient, $\|P[\mathbf{x} - \mathbf{J}_r^T(\mathbf{x})\mathbf{W}\mathbf{r}(\mathbf{x})] - \mathbf{x}\|_2$, of the residual $\|\mathbf{r}(\mathbf{x})\|_{\mathbf{W}}$, (see Section 4) at the estimate of the solution sought. Any negative value will be recorded as zero. The default is `stop_pg_absolute` = 10^{-6} .

`stop_pg_relative` is a scalar variable of type `REAL(rp_)`, that is used to specify the largest relative reduction in the norm of the projected gradient of the residual that will be permitted (see Section 4) at the estimate of the solution sought compared to that at the initial point. Any negative value will be recorded as zero. The default is `stop_pg_relative` = 0.0.

`stop_s` is a scalar variable of type `REAL(rp_)`, that is used to specify the minimum acceptable correction step $\mathbf{s}$ relative to the current estimate of the solution $\mathbf{x}$. The algorithm will be deemed to have converged if $|s_i| \leq \text{stop_s} * \max(1, |x_i|)$ for all $i = 1, \dots, n$. The default is `stop_s` = u , where u is `EPSILON(1.0)` (`EPSILON(1.0D0)` in `GALAHAD_SNLS_double`).

`stop_pg_switch` is a scalar variable of type `REAL(rp_)`, that is used to specify the value of the norm of the projected gradient below which the subproblem solver will change from an interior-point to a projection method when `%subproblem_solver` = 3. The default is `stop_pg_switch` = 0.001.

`initial_weight` is a scalar variable of type `REAL(rp_)`, that holds the required initial value of the regularization weight. If `initial_weight` ≤ 0 , the weight will be chosen automatically by `GALAHAD_SNLS_solve`. The default is `initial_weight` = 100.0.

`minimum_weight` is a scalar variable of type `REAL(rp_)`, that holds the largest permitted value of the regularization weight as the algorithm proceeds. The default is `minimum_weight` = 10^{-8} .

`eta_successful`, `eta_very_successful` and `eta_too_successful` are scalar variables of type default `REAL(rp_)`, that control the acceptance and rejection of the trial step and the updates to the regularization weight. At every iteration, the ratio of the actual reduction in the merit function following the trial step to that predicted by the model is computed. The step is accepted whenever this ratio exceeds `eta_successful`; otherwise the regularization weight will be reduced. If, in addition, the ratio exceeds `eta_very_successful` but not `eta_too_successful`, the regularization weight may be increased. The defaults are `eta_successful` = 10^{-8} , `eta_very_successful` = 0.9 and `eta_too_successful` = 2.0.

`weight_increase`, `weight_decrease`, `weight_increase_max` and `weight_decrease_min` are scalar variables of type `REAL(rp_)`, that control the maximum amounts by which the regularization weight can contract or expand during an iteration. The weight will be decreased by powers of `weight_decrease`, but not in total more than `weight_decrease_min`, until it is smaller than the norm of the current step. It can be increased by at most a factor `weight_increase`, but not in total less than `weight_increase_max`. The defaults are `weight_increase` = 10.0, `weight_decrease` = 0.1, `weight_increase_max` = 100.0 and `weight_decrease_min` = 0.1.

`switch_to_newton` is a scalar variable of type `REAL(rp_)`, that is used to specify the value of the two-norm of the projected gradient, required before a switch is made from the Gauss-Newton model to the Newton one when `%newton_acceleration` is `.TRUE.` **Not yet implemented.**

`cpu_time_limit` is a scalar variable of type `REAL(rp_)`, that is used to specify the maximum permitted CPU time. Any negative value indicates no limit will be imposed. The default is `cpu_time_limit` = - 1.0.

`clock_time_limit` is a scalar variable of type `REAL(rp_)`, that is used to specify the maximum permitted elapsed system clock time. Any negative value indicates no limit will be imposed. The default is `clock_time_limit` = - 1.0.

`newton_acceleration` is a scalar variable of type default `LOGICAL`, that should be set `.TRUE.` if available second derivatives should be used to accelerate the convergence of the algorithm. The default is `newton_acceleration` = `.FALSE.` **Not yet implemented.**

All use is subject to the conditions of a BSD-3-Clause License.
See <https://www.galahad.rl.ac.uk/download/> for full details.

`magic_step` is a scalar variable of type default LOGICAL, that should be set `.TRUE.` if additional “magic” steps are to be used in order to improve the objective as the iteration proceeds, and `.FALSE.` if no “magic” steps are used. The default is `magic_step = .FALSE..`

`print_obj` is a scalar variable of type default LOGICAL, that should be set `.TRUE.` if output values relate to $f(\mathbf{x})$, and `.FALSE.` if they relate to $\|\mathbf{r}(\mathbf{x})\|_{\mathbf{W}}$. The default is `print_obj = .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..`

`alive_file` is a scalar variable of type default CHARACTER and length 30, that gives the name of the temporary file whose removal from stream number `alive_unit` terminates execution of `GALAHAD_SNLS_solve`. The default is `alive_unit = ALIVE.d`.

`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 = ""`.

`SLLS_control` is a scalar variable of type `SLLS_control_type` whose components are used to control the projected-gradient step calculation using `GALAHAD_SLLS` has been selected (see `subproblem_solver`. See the specification sheet for the package `GALAHAD_SLLS` for details, and appropriate default values.

`SLLSB_control` is a scalar variable of type `SLLSB_control_type` whose components are used to control the interior-point step calculation using `GALAHAD_SLLB` has been selected (see `%subproblem_solver`. See the specification sheet for the package `GALAHAD_SLLB` for details, and appropriate default values.

2.4.4 The derived data type for holding timing information

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

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

`slls` is a scalar variable of type `REAL(rp_)`, that gives the total CPU time spent solving the step-determination subproblem using the projected-gradient method from the `GALAHAD_SLLS` package.

`sllsb` is a scalar variable of type `REAL(rp_)`, that gives the total CPU time spent solving the step-determination subproblem using the interior-point method from the `GALAHAD_SLLSB` package.

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

`clock_slls` is a scalar variable of type `REAL(rp_)`, that gives the total elapsed system clock time spent solving the step-determination subproblem using the projected-gradient method from the `GALAHAD_SLLS` package.

`clock_sllsb` is a scalar variable of type `REAL(rp_)`, that gives the total elapsed system clock time spent solving the step-determination subproblem using the interior-point method from the `GALAHAD_SLLSB` package.

All use is subject to the conditions of a BSD-3-Clause License.
See <https://www.galahad.rl.ac.uk/download/> for full details.

2.4.5 The derived data type for holding informational parameters

The derived data type `SNLS_inform_type` is used to hold parameters that give information about the progress and needs of the algorithm. The derived type `SNLS_subproblem_inform_type` comprises all of the components of `SNLS_inform_type` except for the last (i.e., `subproblem_inform`). The components of `SNLS_inform_type` are:

`status` is a scalar variable of type `INTEGER(ip_)`, that gives the exit status of the algorithm. See Sections 2.7 and 2.8 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`.

`iter` is a scalar variable of type `INTEGER(ip_)`, that holds the number of iterations performed.

`inner_iter` is a scalar variable of type `INTEGER(ip_)`, that gives the total number of inner (projected gradient and/or interior-point) iterations required.

`c_eval` is a scalar variable of type `INTEGER(ip_)`, that gives the total number of residual function evaluations performed.

`Jr_eval` is a scalar variable of type `INTEGER(ip_)`, that gives the total number of residual Jacobian evaluations performed.

`obj` is a scalar variable of type `REAL(rp_)`, that holds the value of the objective function $f(\mathbf{x})$ at the best estimate of the solution found.

`norm_r` is a scalar variable of type `REAL(rp_)`, that holds the value of the two-norm of the residual function, $\|\mathbf{r}(\mathbf{x})\|_{\mathbf{W}}$, at the best estimate of the solution found.

`norm_g` is a scalar variable of type `REAL(rp_)`, that holds the value of the norm of the gradient of the two-norm of the residual function, $\|\mathbf{J}_r^T(\mathbf{x})\mathbf{W}\mathbf{r}(\mathbf{x})\|_2/\|\mathbf{r}(\mathbf{x})\|_{\mathbf{W}}$, at the best estimate of the solution found.

`norm_pg` is a scalar variable of type `REAL(rp_)`, that holds the value of the norm of the projected gradient of the residual function, $\|P[\mathbf{x} - \mathbf{J}_r^T(\mathbf{x})\mathbf{W}\mathbf{r}(\mathbf{x})] - \mathbf{x}\|_2$, at the best estimate of the solution found.

`weight` is a scalar variable of type `REAL(rp_)`, that holds the final value of the regularization weight used.

`time` is a scalar variable of type `SNLS_time_type` whose components are used to hold elapsed CPU and system clock times for the various parts of the calculation (see Section 2.4.4).

`SLLS_inform` is a scalar variable of type `SLLS_inform_type` whose components give information about the progress and needs of the projected-gradient subproblem package `GALAHAD_SLLS`. See the specification sheet for the package `GALAHAD_SLLS` for details.

`SLLSB_inform` is a scalar variable of type `SLLSB_inform_type` whose components give information about the progress and needs of the integer-point subproblem package `GALAHAD_SLLSB`. See the specification sheet for the package `GALAHAD_SLLSB` for details.

2.4.6 The derived data type for holding problem data

The derived data type `SNLS_data_type` is used to hold all the data for a particular problem, or sequences of problems with the same structure, between calls of SNLS procedures. This data should be preserved, untouched (except as directed on return from `GALAHAD_SNLS_solve` with positive values of `inform%status`, see Section 2.7), from the initial call to `SNLS_initialize` to the final call to `SNLS_terminate`.

All use is subject to the conditions of a BSD-3-Clause License.
See <https://www.galahad.rl.ac.uk/download/> for full details.

2.4.7 The derived data type for holding user data

The derived data type `USERDATA_type` is available to allow the user to pass data to and from user-supplied subroutines for function and derivative calculations (see Section 2.6). Components of variables of type `USERDATA_type` may be allocated as necessary. The following components are available:

`integer` is a rank-one allocatable array of type `INTEGER(ip_)`.

`real` is a rank-one allocatable array of type default `REAL(rp_)`

`complex` is a rank-one allocatable array of type default `COMPLEX` (double precision complex in `GALAHAD_SNLS_`–`double`).

`character` is a rank-one allocatable array of type default `CHARACTER`.

`logical` is a rank-one allocatable array of type default `LOGICAL`.

`integer_pointer` is a rank-one pointer array of type `INTEGER(ip_)`.

`real_pointer` is a rank-one pointer array of type default `REAL(rp_)`

`complex_pointer` is a rank-one pointer array of type default `COMPLEX` (double precision complex in `GALAHAD_SNLS_`–`double`).

`character_pointer` is a rank-one pointer array of type default `CHARACTER`.

`logical_pointer` is a rank-one pointer array of type default `LOGICAL`.

2.4.8 The derived data type for holding reverse-communication data

The derived data type `REVERSE_type` is used to hold data needed for reverse communication when this is required. The components of `REVERSE_type` are:

`lv1` is a scalar variable of type `INTEGER(ip_)`, that may be used to hold the starting position in `V` (see below) of the list of indices of nonzero components of $\mathbf{v}$.

`lvu` is a scalar variable of type `INTEGER(ip_)`, that may be used to hold the finishing position in `V` (see below) of the list of indices of nonzero components of $\mathbf{v}$.

`IV` is a rank-one allocatable array of dimension n and type `INTEGER(ip_)`, that may be used to hold the indices of the nonzero components of $\mathbf{v}$. If used, components `IV(lv1:lvu)` of `V` (see below) will be nonzero.

`V` is a rank-one allocatable array of dimension n and type `REAL(rp_)`, that is used to hold the components of an input vector $\mathbf{v}$.

`index` is a scalar variable of type `INTEGER(ip_)`, that may be used to hold a column index of $\mathbf{J}_r$ if required.

`lp` is a scalar variable of type `INTEGER(ip_)`, that is used to record the finishing position in `P` and `IP` (see below) of the list of indices of nonzero components a requested operation on $\mathbf{v}$.

`P` is a rank-one allocatable array of dimension m_r and type `REAL(rp_)`, that is used to record the components of the vector that results from a requested operation on $\mathbf{v}$.

`IP` is a rank-one allocatable array of dimension n and type `INTEGER(ip_)`, that is used to record the list of indices of nonzero components of a requested operation on $\mathbf{v}$.

All use is subject to the conditions of a BSD-3-Clause License.
See <https://www.galahad.rl.ac.uk/download/> for full details.

2.5 Argument lists and calling sequences

There are three procedures for user calls (see Section 2.9 for further features):

1. The subroutine `SNLS_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 `SNLS_solve` is called to solve the problem.
3. The subroutine `SNLS_terminate` is provided to allow the user to automatically deallocate array components of the private data, allocated by `SNLS_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 `SNLS_initialize` cannot test for this situation, and any existing associated targets will subsequently become unreachable.

We use square brackets [] to indicate OPTIONAL arguments.

2.5.1 The initialization subroutine

Default values are provided as follows:

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

`data` is a scalar INTENT(INOUT) argument of type `SNLS_data_type` (see Section 2.4.6). It is used to hold data about the problem being solved.

`control` is a scalar INTENT(OUT) argument of type `SNLS_control_type` (see Section 2.4.3). On exit, `control` contains default values for the components as described in Section 2.4.3, while `control%subproblem_control` contains default values that are required for the model minimization if `control%model ≥ 6`). These values should only be changed after calling `SNLS_initialize`.

`inform` is a scalar INTENT(OUT) argument of type `SNLS_inform_type` (see Section 2.4.5). A successful call to `SNLS_initialize` is indicated when the component `inform%status` has the value 0. For other return values of `inform%status`, see Section 2.8. The components of `inform` correspond to the main algorithm, while those in `inform%subproblem_inform` refer to the model minimization if `control%model ≥ 6`).

2.5.2 The minimization subroutine

The minimization algorithm is called as follows:

```
CALL SNLS_solve( nlp, control, inform, data, userdata[, reverse, eval_R, eval_Jr, &
                 eval_Jr_prod, eval_Jr_scol, eval_Jr_sprod ] )
```

`nlp` is a scalar INTENT(INOUT) argument of type `NLPT_problem_type` (see Section 2.4.2). It is used to hold data about the problem being solved. For a new problem, the user must allocate all the array components, and set values for `nlp%n` and the required non-real components of `nlp%Jr` depending on what level of derivatives are required by the model requested. Specifically, if `control%jacobian_available = 2`, all appropriate components of `nlp%Jr` should be allocated and, with the exception of `nlp%Jr%val`, filled with data. Users are free to choose whichever of the matrix formats described in Section 2.2 is appropriate for J_r for their application.

The component `nlp%X` must be set to an initial estimate, $\mathbf{x}^0$, of the minimization variables. A good choice will increase the speed of the package, but the underlying method is designed to converge (at least to a local solution) from an arbitrary initial guess. The component `nlp%COHORT` needs only be allocated and set if there are multiple non-overlapping regular simplex constraints (1.2), while the same is true for `nlp%W` when non-unit weights are required.

On exit, the component `nlp%X` will contain the best estimates of the minimization variables $\mathbf{x}$.

All use is subject to the conditions of a BSD-3-Clause License.
 See <https://www.galahad.rl.ac.uk/download/> for full details.

W is an OPTIONAL rank-one array of type REAL(rp_) whose components specifies the diagonal weighting matrix W that defines the objective function $f(\mathbf{x})$. If W is present, it must be of length at least n , and $W(i)$ should contain $w_i > 0, i = 1, \dots, n$. If W is absent, the weights w_i will all be taken to be 1.0.

Restrictions: $nlp\%n > 0, nlp\%m_r > 0$ and $nlp\%Jr\%type \in \{'DENSE', 'DENSE_BY_ROWS', 'DENSE_BY_COLUMNS', 'COORDINATE', 'SPARSE_BY_ROWS', 'SPARSE_BY_COLUMNS'\}$

`control` is a scalar INTENT(IN) argument of type SNLS_control_type (see Section 2.4.3). Default values may be assigned by calling SNLS_initialize prior to the first call to SNLS_solve.

`inform` is a scalar INTENT(INOUT) argument of type SNLS_inform_type (see Section 2.4.5). On initial entry, the component `inform%status` must be set to the value 1. Other entries need not be set. A successful call to SNLS_solve is indicated when the component `inform%status` has the value 0. For other return values of `inform%status`, see Sections 2.7 and 2.8.

`data` is a scalar INTENT(INOUT) argument of type SNLS_data_type (see Section 2.4.6). It is used to hold data about the problem being solved. With the possible exceptions of the components `data%eval_status` and `data%U` (see Section 2.7), it must not have been altered by the user since the last call to SNLS_initialize.

`userdata` is a scalar INTENT(INOUT) argument of type USERDATA_type whose components may be used to communicate user-supplied data to and from the OPTIONAL subroutines `eval_R`, `eval_Jr`, `eval_H`, `eval_JPROD`, `eval_HPROD`, `eval_HPRODS` and `eval_SCALE` (see Section 2.4.7).

`userdata` is a scalar INTENT(INOUT) argument of type USERDATA_type whose components may be used to communicate user-supplied data (see Section 2.4.7) to and from the OPTIONAL subroutines `eval_R`, `eval_Jr`, `eval_Jr_prod`, `eval_Jr_scol` and `eval_Jr_sprod` (see below).

`reverse` is an OPTIONAL scalar INTENT(INOUT) argument of type REVERSE_type (see Section 2.4.8). It is used to communicate reverse-communication data between the subroutine and calling program. It is required if either (i) `control%jacobian_available = 2` and `eval_Jr` (see below) is absent or (ii) `control%jacobian_available = 1` and `eval_Jr_prod` or `eval_Jr_scol` are absent. In these cases, the user should monitor `inform%status` on exit (see Section 2.7).

`eval_R` is an OPTIONAL user-supplied subroutine whose purpose is to evaluate the value of the residual function $\mathbf{r}(\mathbf{x})$ at a given vector $\mathbf{x}$. See Section 2.6.1 for details. If `eval_R` is present, it must be declared EXTERNAL in the calling program. If `eval_R` is absent, GALAHAD_SNLS_solve will use reverse communication to obtain objective function values (see Section 2.7).

`eval_Jr` is an OPTIONAL user-supplied subroutine whose purpose is to evaluate the value of the Jacobian of the residual function $\mathbf{J}_r(\mathbf{x})$ at a given vector $\mathbf{x}$. See Section 2.6.2 for details. If `eval_J` is present, it must be declared EXTERNAL in the calling program. If `eval_J` is absent, GALAHAD_SNLS_solve will use reverse communication to obtain Jacobian values (see Section 2.7).

`eval_Jr_prod` is an OPTIONAL user-supplied subroutine whose purpose is to evaluate the value of the product $\mathbf{p} = \mathbf{J}_r(\mathbf{x})\mathbf{v}$ or $\mathbf{p} = \mathbf{J}_r^T(\mathbf{x})\mathbf{v}$ involving the Jacobian of the residual function $\mathbf{J}_r(\mathbf{x})$, or its transpose, and a given vector $\mathbf{v}$. See Section 2.6.3 for details. If `eval_Jr_prod` is present, it must be declared EXTERNAL in the calling program. If `eval_Jr_prod` is absent, GALAHAD_SNLS_solve will use reverse communication to obtain Jacobian-vector products if required (see Section 2.7).

`eval_Jr_scol` is an OPTIONAL user-supplied subroutine whose purpose is to evaluate the nonzero components of a specified column of $\mathbf{J}_r(\mathbf{x})$. See Section 2.6.4 for details. If `eval_Jr_scol` is present, it must be declared EXTERNAL in the calling program. If `eval_Jr_scol` is absent, GALAHAD_SNLS_solve will use reverse communication to obtain the required column (see Section 2.7).

All use is subject to the conditions of a BSD-3-Clause License.
See <https://www.galahad.rl.ac.uk/download/> for full details.

`eval_Jr_sprod` is an OPTIONAL user-supplied subroutine whose purpose is to evaluate the value of the product $\mathbf{p} = \mathbf{J}_r(\mathbf{x})\mathbf{v}$ for a given sparse vector $\mathbf{v}$, or specified components of the product $\mathbf{p} = \mathbf{J}_r^T(\mathbf{x})\mathbf{v}$, involving the Jacobian of the residual function $\mathbf{J}_r(\mathbf{x})$. See Section 2.6.5 for details. If `eval_Jr_sprod` is present, it must be declared EXTERNAL in the calling program. If `eval_Jr_sprod` is absent, GALAHAD_SNLS_solve will use reverse communication to obtain sparse Jacobian-vector products if required (see Section 2.7).

2.5.3 The termination subroutine

All previously allocated arrays are deallocated as follows:

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

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

`control` is a scalar INTENT(IN) argument of type SNLS_control_type exactly as for SNLS_solve.

`inform` is a scalar INTENT(OUT) argument of type SNLS_inform_type exactly as for SNLS_solve. The component `inform%status` will be set on exit, and a successful call to SNLS_terminate is indicated when this component has the value 0. For other return values of `inform%status`, see Section 2.8.

2.6 Function and derivative values

2.6.1 The residual value via internal evaluation

If the argument `eval_R` is present when calling GALAHAD_SNLS_solve, the user is expected to provide a subroutine of that name to evaluate the value of the residual functions $\mathbf{r}(\mathbf{x})$. The routine must be specified as

```
SUBROUTINE eval_R( status, X, userdata, R )
```

whose arguments are as follows:

`status` is a scalar INTENT(OUT) argument of type INTEGER(ip_), that should be set to 0 if the routine has been able to evaluate the residual functions and to a non-zero value if the evaluation has not been possible.

`X` is a rank-one INTENT(IN) array argument of type REAL(rp_) whose components contain the vector $\mathbf{x}$.

`userdata` is a scalar INTENT(INOUT) argument of type USERDATA_type whose components may be used to communicate user-supplied data to and from the subroutine (see Section 2.4.7).

`R` is a rank-one INTENT(OUT) array argument of type REAL(rp_) that should be set to the value of the residuals $\mathbf{r}(\mathbf{x})$ evaluated at the vector $\mathbf{x}$ input in `X`.

2.6.2 Jacobian values via internal evaluation

If the argument `eval_Jr` is present when calling GALAHAD_SNLS_solve, the user is expected to provide a subroutine of that name to evaluate the values of the residual Jacobian $\mathbf{J}_r(\mathbf{x})$. The routine must be specified as

```
SUBROUTINE eval_Jr( status, X, userdata, Jr_val )
```

whose arguments are as follows:

`status` is a scalar INTENT(OUT) argument of type INTEGER(ip_), that should be set to 0 if the routine has been able to evaluate the residual Jacobian, and to a non-zero value if the evaluation has not been possible.

All use is subject to the conditions of a BSD-3-Clause License.
See <https://www.galahad.rl.ac.uk/download/> for full details.

X is a rank-one `INTENT (IN)` array argument of type `REAL (rp_)` whose components contain the vector $\mathbf{x}$.

`userdata` is a scalar `INTENT (INOUT)` argument of type `USERDATA_type` whose components may be used to communicate user-supplied data to and from the subroutine (see Section 2.4.7).

`Jr_val` is a scalar `INTENT (OUT)` argument of type `REAL (rp_)`, whose components should be set to the values of the Jacobian $\mathbf{J}_r(\mathbf{x})$ evaluated at the vector $\mathbf{x}$ input in X . The values should be input in the same order as that in which the array indices were given in `nlp%J`.

2.6.3 Jacobian-vector products via internal evaluation

If the argument `eval_Jr_prod` is present when calling `GALAHAD_SNLS_solve`, the user is expected to provide a subroutine of that name to evaluate the product $\mathbf{p} = \mathbf{J}_r(\mathbf{x})\mathbf{v}$ or $\mathbf{p} = \mathbf{J}_r^T(\mathbf{x})\mathbf{v}$ involving the product of the residual Jacobian $\mathbf{J}_r(\mathbf{x})$ or its transpose $\mathbf{J}_r^T(\mathbf{x})$. The routine must be specified as

```
SUBROUTINE eval_Jr_prod( status, X, userdata, transpose, V, P, got_Jr )
```

whose arguments are as follows:

`status` is a scalar `INTENT (OUT)` argument of type `INTEGER (ip_)`, that should be set to 0 if the routine has been able to evaluate the product $\mathbf{p} = \mathbf{J}_r(\mathbf{x})\mathbf{v}$ or $\mathbf{p} = \mathbf{J}_r^T(\mathbf{x})\mathbf{v}$ and to a non-zero value if the evaluation has not been possible.

X is a rank-one `INTENT (IN)` array argument of type `REAL (rp_)` whose components contain the vector $\mathbf{x}$.

`userdata` is a scalar `INTENT (INOUT)` argument of type `USERDATA_type` whose components may be used to communicate user-supplied data to and from the subroutine (see Section 2.4.7).

`transpose` is a scalar `INTENT (IN)` array argument of type default that will be set `.TRUE.` if the product involves the transpose of the Jacobian $\mathbf{J}_r^T(\mathbf{x})$ and `.FALSE.` if the product involves the Jacobian $\mathbf{J}_r(\mathbf{x})$ itself.

V is a rank-one `INTENT (IN)` array argument of type `REAL (rp_)` whose components contain the vector $\mathbf{v}$.

P is a rank-one `INTENT (INOUT)` array argument of type `REAL (rp_)` whose components on input contain the vector $\mathbf{p}$ and on output the product $\mathbf{p} = \mathbf{J}_r(\mathbf{x})\mathbf{v}$ when `%transpose` is `.FALSE.` or $\mathbf{p} = \mathbf{J}_r^T(\mathbf{x})\mathbf{v}$ when `%transpose` is `.TRUE.`.

`got_Jr` is a scalar `INTENT (IN)` array argument of type default that will be set `.TRUE.` if the user has already computed some information about $\mathbf{J}_r(\mathbf{x})$ at the given X , and thus may avoid unnecessary re-computation. It will be `.FALSE.` otherwise.

2.6.4 Columns of the Jacobian via internal evaluation

If the argument `eval_Jr_scol` is present when calling `GALAHAD_SNLS_solve`, the user is expected to provide a subroutine of that name to evaluate a specified column of the Jacobian $\mathbf{J}_r(\mathbf{x})$. The routine must be specified as

```
SUBROUTINE eval_Jr_scol( status, X, userdata, index, VAL, ROW, NZ, got_Jr )
```

whose arguments are as follows:

`status` is a scalar `INTENT (OUT)` argument of type `INTEGER (ip_)`, that should be set to 0 if the routine has been able to evaluate the desired columns of $\mathbf{J}_r(\mathbf{x})$ and to a non-zero value if the evaluation has not been possible.

X is a rank-one `INTENT (IN)` array argument of type `REAL (rp_)` whose components contain the vector $\mathbf{x}$.

All use is subject to the conditions of a BSD-3-Clause License.
See <https://www.galahad.rl.ac.uk/download/> for full details.

`userdata` is a scalar `INTENT (INOUT)` argument of type `USERDATA_type` whose components may be used to communicate user-supplied data to and from the subroutine (see Section 2.4.7).

`index` is a scalar `INTENT (OUT)` argument of type `INTEGER(ip_)`, that should be set to the index of the required column of $\mathbf{J}_r(\mathbf{x})$.

`VAL` is a rank-one `INTENT (OUT)` array argument of type `REAL(rp_)` whose first `nz` components on output contain the nonzeros of the `index`-th column of $\mathbf{J}_r(\mathbf{x})$.

`ROW` is an `INTENT (INOUT)` rank-one array of dimension at least `nz` and type `INTEGER(ip_)`, that must be set to record the list of indices of nonzero components of the `index`-th column of $\mathbf{J}_r(\mathbf{x})$.

`nz` is an `INTENT (INOUT)` scalar variable of type `INTEGER(ip_)`, that must be set to record the number of non-zeros in the `col`-th column of $\mathbf{J}_r(\mathbf{x})$.

`got_Jr` is a scalar `INTENT (IN)` array argument of type default that will be set `.TRUE.` if the user has already computed some information about $\mathbf{J}_r(\mathbf{x})$ at the given $\mathbf{x}$, and thus may avoid unnecessary re-computation. It will be `.FALSE.` otherwise.

2.6.5 Jacobian-vector sub-products via internal evaluation

If the argument `eval_Jr_sprod` is present when calling `GALAHAD_SNLS_solve`, the user is expected to provide a subroutine of that name to evaluate the product of the Jacobian $\mathbf{J}_r(\mathbf{x})$, or its transpose, with a given vector $\mathbf{v}$. Here, either only a subset of the components of the vector $\mathbf{v}$ are nonzero, or only a subset of the components product $\mathbf{J}_r^T(\mathbf{x})\mathbf{v}$ are required. The routine must be specified as

```
SUBROUTINE eval_Jr_sprod( status, X, userdata, transpose, V, P, FREE, n_free, got_Jr )
```

whose arguments are as follows:

`status` is a scalar `INTENT (OUT)` argument of type `INTEGER(ip_)`, that should be set to 0 if the routine has been able to evaluate the sparse product $\mathbf{J}_r\mathbf{v}$ or $\mathbf{J}_r^T\mathbf{v}$ and to a non-zero value if the evaluation has not been possible.

`X` is a rank-one `INTENT (IN)` array argument of type `REAL(rp_)` whose components contain the vector $\mathbf{x}$.

`userdata` is a scalar `INTENT (INOUT)` argument of type `USERDATA_type` whose components may be used to communicate user-supplied data to and from the subroutine (see Section 2.4.7).

`transpose` is a scalar `INTENT (IN)` array argument of type default that will be set `.TRUE.` if the product involves the transpose of the Jacobian $\mathbf{J}_r^T(\mathbf{x})$ and `.FALSE.` if the product involves the Jacobian $\mathbf{J}_r(\mathbf{x})$ itself.

`V` is a rank-one `INTENT (IN)` array argument of type `REAL(rp_)` whose components contain the vector $\mathbf{v}$. If `transpose` is `.FALSE.` only those components whose indices `FREE(:n_free)` (see below) will be set, and the remainder will be presumed to be zero.

`P` is a rank-one `INTENT (OUT)` array argument of type `REAL(rp_)` whose components on output must be set to the product $\mathbf{J}_r(\mathbf{x})\mathbf{v}$ when `%transpose` is `.FALSE.`. If `%transpose` is `.TRUE.`, components with indices `FREE(:n_free)` (see below) should be set to the corresponding components of the product $\mathbf{J}_r^T\mathbf{v}$, and the remaining components ignored.

`FREE` is a rank-one `INTENT (IN)` array argument of type `INTEGER(ip_)` that flags the input components of $\mathbf{v}$ that are set (when `transpose` is `.FALSE.`) or output components of $\mathbf{J}_r^T(\mathbf{x})\mathbf{v}$ that are required (when `transpose` is `.TRUE.`). Specifically, only indices `FREE(:n_free)` of the relevant vector is set or required, and the remainder should be treated as zero (if `transpose` is `.FALSE.`) or ignored (if `transpose` is `.TRUE.`).

All use is subject to the conditions of a BSD-3-Clause License.
 See <https://www.galahad.rl.ac.uk/download/> for full details.

`n_free` is a scalar `INTENT(IN)` argument of type `INTEGER(ip_)`, that specifies the number of components of `FREE` that need be considered.

`got_Jr` is a scalar `INTENT(IN)` array argument of type default that will be set `.TRUE.` if the user has already computed some information about $\mathbf{J}_r(\mathbf{x})$ at the given $\mathbf{x}$, and thus may avoid unnecessary re-computation. It will be `.FALSE.` otherwise.

2.7 Reverse Communication Information

A positive value of `inform%status` on exit from `SNLS_solve` indicates that `GALAHAD_SNLS_solve` is seeking further information—this will happen if the user has chosen not to evaluate function or derivative values internally (see Section 2.6). The user should compute the required information and re-enter `GALAHAD_SNLS_solve` with `inform%status` and all other arguments (except those specifically mentioned below) unchanged.

Possible values of `inform%status` and the information required are

2. The user should compute the vector of residual functions $\mathbf{r}(\mathbf{x})$ at the point $\mathbf{x}$ indicated in `nlp%X`. The required value should be set in `nlp%R`, and `reverse%eval_status` should be set to 0. If the user is unable to evaluate $\mathbf{r}(\mathbf{x})$ —for instance, if one or more of the residual functions is undefined at $\mathbf{x}$ —the user need not set `nlp%R`, but should then set `reverse%eval_status` to a non-zero value.
3. The user should compute the Jacobian matrix $\mathbf{J}_r(\mathbf{x})$ of the residuals $\mathbf{r}(\mathbf{x})$ at the point $\mathbf{x}$ indicated in `nlp%X`. The l -th component of the Jacobian stored according to the scheme input in the remainder of `nlp%Jr` (see Section 2.4.2) should be set in `nlp%Jr%val(l)`, for $l = 1, \dots, \text{nlp}\%Jr\%ne$ and `reverse%eval_status` should be set to 0. If the user is unable to evaluate a component of $\mathbf{J}_r(\mathbf{x})$ —for instance, if a component of the Jacobian is undefined at $\mathbf{x}$ —the user need not set `nlp%Jr%val`, but should then set `reverse%eval_status` to a non-zero value.
4. The user should compute the product $\mathbf{p} = \mathbf{J}_r(\mathbf{x})\mathbf{v}$ involving the residual Jacobian $\mathbf{J}_r(\mathbf{x})$ at the point $\mathbf{x}$, given in `nlp%X`, and a given vector $\mathbf{v}$. The vector $\mathbf{v}$ is provided in `reverse%V`, the resulting $\mathbf{p}$ should be placed in `reverse%P`, and `reverse%eval_status` should be set to 0. If the user is unable to evaluate the product—for instance, if a component of the Jacobian is undefined at $\mathbf{x}$ —the user need not set `reverse%P`, but should then set `reverse%eval_status` to a non-zero value.
5. The user should compute the product $\mathbf{p} = \mathbf{J}_r^T(\mathbf{x})\mathbf{v}$ involving the transpose of the residual Jacobian $\mathbf{J}_r(\mathbf{x})$ at the point $\mathbf{x}$, given in `nlp%X`, and a given vector $\mathbf{v}$. The vector $\mathbf{v}$ is provided in `reverse%V`, the resulting $\mathbf{p}$ should be placed in `reverse%P`, and `reverse%eval_status` should be set to 0. If the user is unable to evaluate the product—for instance, if a component of the Jacobian is undefined at $\mathbf{x}$ —the user need not set `reverse%P`, but should then set `reverse%eval_status` to a non-zero value.
6. The user should compute the j -th column of $\mathbf{J}_r(\mathbf{x})$, with j provided in `reverse%index`, at the point $\mathbf{x}$ given in `nlp%X`. The resulting *nonzeros* and their corresponding row indices of the j -th column of $\mathbf{J}_r(\mathbf{x})$ must be placed in `reverse%P(1 : reverse%lp)` and `reverse%IP(1 : reverse%lp)` with `reverse%lp` set accordingly, and `reverse%eval_status` should be set to 0. If the user is unable to evaluate the required column—for instance, if a component of the Jacobian is undefined at $\mathbf{x}$ —the user need not set `reverse%P`, `reverse%IP` and `reverse%nz`, but should then set `reverse%eval_status` to a non-zero value.
7. The user should compute the product $\mathbf{p} = \mathbf{J}_r(\mathbf{x})\mathbf{v}$ involving the residual Jacobian $\mathbf{J}_r(\mathbf{x})$ at the point $\mathbf{x}$, given in `nlp%X`, and a given sparse vector $\mathbf{v}$, whose nonzeros are in positions `reverse%iv(reverse%lvl : reverse%lvu)` of `reverse%V`. The resulting $\mathbf{p}$ should be placed in `reverse%P` and `reverse%eval_status` should be set to 0. If the user is unable to evaluate the product—for instance, if a component of the Jacobian is undefined at $\mathbf{x}$ —the user need not set `reverse%P`, but should then set `reverse%eval_status` to a non-zero value.

All use is subject to the conditions of a BSD-3-Clause License.
See <https://www.galahad.rl.ac.uk/download/> for full details.

8. The user should compute selected components of the product $\mathbf{p} = \mathbf{J}_r^T(\mathbf{x})\mathbf{v}$ involving the transpose of the residual Jacobian $\mathbf{J}_r(\mathbf{x})$ at the point $\mathbf{x}$, given in `nlp%X`, and a given vector $\mathbf{v}$. Only components `reverse%IV(reverse%lvl:reverse%lvu)` of $\mathbf{p}$ should be computed, and recorded in `reverse%P(reverse%IV(reverse%lvl:reverse%lvu))`, and `reverse%eval_status` should be set to 0. If the user is unable to evaluate the product—for instance, if a component of the Jacobian is undefined at $\mathbf{x}$ —the user need not set `reverse%P`, but should then set `reverse%eval_status` to a non-zero value.
9. The user has the opportunity to replace the estimate $\mathbf{x}$ in `nlp%X` by an improved value $\mathbf{x}^+$ for which $f(\mathbf{x}^+) \leq f(\mathbf{x})$; in that case `nlp%R` must also be reset to hold $\mathbf{r}(\mathbf{x}^+)$.

2.8 Warning and error messages

A negative value of `inform%status` on exit from `SNLS_solve` or `SNLS_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. The restriction `nlp%n > 0` and `nlp%m_r > 0`, or requirements that `nlp%Jr%type` contains its relevant string 'DENSE', 'DENSE_BY_ROWS', 'DENSE_BY_COLUMNS', 'SPARSE_BY_ROWS', 'SPARSE_BY_COLUMNS', 'COORDINATE' has been violated.
- 9. The analysis phase of the factorization failed; the return status from the factorization package is given in the component `inform%factor_status`.
- 10. The factorization failed; the return status from the factorization package is given in the component `inform%factor_status`.
- 11. The solution of a set of linear equations using factors from the factorization package failed; the return status from the factorization package is given in the component `inform%factor_status`.
- 16. The problem is so ill-conditioned that further progress is impossible.
- 17. The step is too small to make further impact.
- 18. Too many iterations have been performed. This may happen if `control%maxit` is too small, but may also be symptomatic of a badly scaled problem.
- 19. The elapsed CPU or system clock time limit has been reached. This may happen if either `control%cpu_time_limit` or `control%clock_time_limit` is too small, but may also be symptomatic of a badly scaled problem.
- 82. The user has forced termination of `GALAHAD_SNLS_solve` by removing the file named `control%alive_file` from unit `unit control%alive_unit`.

2.9 Further features

In this section, we describe an alternative means of setting control parameters, that is components of the variable `control` of type `SNLS_control_type` (see Section 2.4.3), by reading an appropriate data specification file using the subroutine `SNLS_read_specfile`. This facility is useful as it allows a user to change SNLS control parameters without editing and recompiling programs that call SNLS.

All use is subject to the conditions of a BSD-3-Clause License.
See <https://www.galahad.rl.ac.uk/download/> for full details.

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

```
( .. lines ignored by SNLS_read_specfile .. )
BEGIN SNLS
    keyword      value
    .....
    keyword      value
END
( .. lines ignored by SNLS_read_specfile .. )
```

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

and

```
END SNLS SPECIFICATION
```

are acceptable. Furthermore, between the "BEGIN SNLS" 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 `SNLS_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 `SNLS_read_specfile`.

2.9.1 To read control parameters from a specification file

Control parameters may be read from a file as follows:

```
CALL SNLS_read_specfile( control, device )
```

`control` is a scalar `INTENT(INOUT)` argument of type `SNLS_control_type` (see Section 2.4.3). Default values should have already been set, perhaps by calling `SNLS_initialize`. On exit, individual components of `control` and `control%subproblem_control` may have been changed according to the commands found in the specfile. Specfile commands and the component (see Section 2.4.3) of `control` and `control%subproblem_control` that each affects are given in Table 2.1.

All use is subject to the conditions of a BSD-3-Clause License.
See <https://www.galahad.rl.ac.uk/download/> for full details.

command	component of control	value type
error-printout-device	%error	integer
printout-device	%out	integer
print-level	%print_level	integer
start-print	%start_print	integer
stop-print	%stop_print	integer
iterations-between-printing	%print_gap	integer
maximum-number-of-iterations	%maxit	integer
alive-device	%alive_unit	integer
jacobian-available	%jacobian_available	integer
subproblem-solver	%subproblem_solver	integer
history-length-for-non-monotone-descent	%non_monotone	integer
weight-update-strategy	weight_update_strategy	integer
absolute-residual-accuracy-required	%stop_c_absolute	real
relative-residual-reduction-required	%stop_c_relative	real
absolute-gradient-accuracy-required	%stop_pg_absolute	real
relative-gradient-reduction-required	%stop_pg_relative	real
minimum-relative-step-allowed	%stop_s	real
relative-gradient-switch-tolerance	%stop_pg_switch	real
initial-regularization-weight	%initial_weight	real
minimum-regularization-weight	%minimum_weight	real
successful-iteration-tolerance	%eta_successful	real
very-successful-iteration-tolerance	%eta_very_successful	real
too-successful-iteration-tolerance	%eta_too_successful	real
regularization-weight-minimum-decrease-factor	%weight_decrease_min	real
regularization-weight-decrease-factor	%weight_decrease	real
regularization-weight-increase-factor	%weight_increase	real
regularization-weight-maximum-increase-factor	%weight_increase_max	real
maximum-cpu-time-limit	%cpu_time_limit	real
maximum-clock-time-limit	%clock_time_limit	real
try-newton-acceleration	%newton_acceleration	logical
choose-magic-step	%magic_step	logical
print-objective	%print_obj	logical
space-critical	%space_critical	logical
deallocate-error-fatal	%deallocate_error_fatal	logical
alive-filename	%alive_file	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`.

2.10 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`, a single line of output will be produced for each iteration of the process. This will include the values of the objective function and the norm of its projected gradient, the ratio of actual to predicted decrease following the step, the value of the regularization weight and the time taken so far.

All use is subject to the conditions of a BSD-3-Clause License.
See <https://www.galahad.rl.ac.uk/download/> for full details.

If `control%print_level` ≥ 2 this output will be increased to provide significant detail of each iteration. This extra output includes residuals of the linear systems solved, and, for larger values of `control%print_level`, values of the variables and gradients. Further details concerning the attempted solution the model may be obtained by increasing `control%SLLS_control%print_level` and `control%SLLSB_control%print_level`. See the specification sheets for the packages GALAHAD_SLLS and GALAHAD_SLLSB for details.

3 GENERAL INFORMATION

Use of common: None.

Workspace: Provided automatically by the module.

Other routines called directly: None.

Other modules used directly: SNLS_solve calls the GALAHAD packages GALAHAD_CLOCK, GALAHAD_SYMBOLS, GALAHAD_NLPT, GALAHAD_USERDATA, GALAHAD_REVERSE, GALAHAD_SPECFILE, GALAHAD_QPT, GALAHAD_SLLS, GALAHAD_SLLSB, GALAHAD_SPACE, GALAHAD_MOP, GALAHAD_NORMS and GALAHAD_STRING.

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

Restrictions: `nlp%n` > 0 , `nlp%m_r` > 0 , `nlp%Jr%type` $\in \{ 'DENSE', 'COORDINATE', 'SPARSE_BY_ROWS' \}$.

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

4 METHOD

The primal optimality conditions (1.2) and dual optimality conditions

$$J_r^T(\mathbf{x})\mathbf{W}\mathbf{r}(\mathbf{x}) = \sum_{i=1}^{m_c} \mathbf{e}_{C_i} y_i + \mathbf{z}$$

necessarily hold at an optimal point $\mathbf{x}$ for some Lagrange multipliers $\mathbf{y}$ and dual variables $\mathbf{z} \geq 0$, where additionally $\mathbf{x}$ and $\mathbf{z}$ satisfy the complementarity conditions $x_i z_i = 0$ for $i = 1, \dots, n$.

An adaptive regularization method is used. In this, an improvement to a current estimate of the required minimizer, $\mathbf{x}_k$ is sought by computing a step $\mathbf{s}_k$. The step is chosen to approximately minimize a model $t_k(\mathbf{s})$ of $f_{p,r}(\mathbf{x}_k + \mathbf{s})$ that includes a weighted regularization term $\frac{1}{2}\sigma_k \|\mathbf{s}\|_2^2$ for some specified positive weight σ_k . The quality of the resulting step $\mathbf{s}_k$ is assessed by computing the "ratio" $(f(\mathbf{x}_k) - f(\mathbf{x}_k + \mathbf{s}_k)) / (t_k(\mathbf{0}) - t_k(\mathbf{s}_k))$. The step is deemed to have succeeded if the ratio exceeds a given $\eta_s > 0$, and in this case $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{s}_k$. Otherwise $\mathbf{x}_{k+1} = \mathbf{x}_k$, and the weight is increased by powers of a given increase factor up to a given limit. If the ratio is larger than $\eta_r \geq \eta_d$, the weight will be decreased by powers of a given decrease factor again up to a given limit. The method will terminate as soon as $f(\mathbf{x}_k)$ or $\|P[\mathbf{x}_k - \nabla_x f(\mathbf{x}_k)] - \mathbf{x}_k\|$ is smaller than a specified value.

The step $\mathbf{s}_k$ may be computed either by employing a projected-gradient method to minimize the model within the simplex constraint set $C(\mathbf{x}_k + \mathbf{s})$ using the GALAHAD module SLLS, or by applying the interior-point method available in the module SLLSB to the same subproblem. Experience has shown that it can be beneficial to use the latter method during early iterations, but to switch to the former as the iterates approach convergence.

References:

Generic regularization methods are described in detail in

C. Cartis, N. I. M. Gould and Ph. L. Toint, "Evaluation complexity of algorithms for nonconvex optimization" SIAM-

All use is subject to the conditions of a BSD-3-Clause License.

See <https://www.galahad.rl.ac.uk/download/> for full details.

MOS Series on Optimization (2022),

and uses “tricks” as suggested in

N. I. M. Gould, M. Porcelli and Ph. L. Toint, “Updating the regularization parameter in the adaptive cubic regularization algorithm”. *Computational Optimization and Applications* **53(1)** (2012) 1–22.

The specific methods employed here are discussed in

N. I. M. Gould et. al. “Nonlinear least-squares over unit simplices”. Rutherford Appleton Laboratory, Oxfordshire, England (2026) in preparation.

5 EXAMPLES OF USE

Suppose we have the parametric residual

$$\mathbf{r}(\mathbf{x}) = \begin{pmatrix} x_1x_2 - p \\ x_2x_3 - 1 \\ x_3x_4 - 1 \\ x_4x_5 - 1 \end{pmatrix}$$

and wish to minimize the least-squares objective (1.1), with unit weights $\mathbf{w} = 1$ and parameter value $p = 4$, over the intersection of the two regular simplices

$$x_1 + x_4 = 1, x_1, x_4 \geq 0 \text{ and } x_2 + x_5 = 1, x_2, x_5 \geq 0.$$

Noting that

$$\mathbf{J}_r(\mathbf{x}) = \begin{pmatrix} x_2 & x_1 & & & \\ & x_3 & x_2 & & \\ & & x_4 & x_3 & \\ & & & x_5 & x_4 \end{pmatrix},$$

assigning variables x_1 and x_4 to cohort 1, x_2 and x_5 to cohort 2 (and recording that x_3 is unconstrained), and starting from the initial guess $\mathbf{x} = (0.5, 0.5, 0.5, 0.5, 0.5)$, we may use the following code:

```
PROGRAM GALAHAD_SNLS_EXAMPLE ! GALAHAD 5.5 - 2026-05-04 AT 11:10 GMT.
USE GALAHAD_SNLS_double      ! double precision version
IMPLICIT NONE
INTEGER, PARAMETER :: wp = KIND( 1.0D+0 ) ! set precision
TYPE ( NLPT_problem_type ) :: nlp
TYPE ( SNLS_control_type ) :: control
TYPE ( SNLS_inform_type ) :: inform
TYPE ( SNLS_data_type ) :: data
TYPE ( USERDATA_type ) :: userdata
EXTERNAL :: EVALR, EVALJr
INTEGER :: s
INTEGER, PARAMETER :: n = 5, m_r = 4, m = 2, jr_ne = 8
REAL ( KIND = wp ), PARAMETER :: p = 4.0_wp ! parameter p
! start problem data
nlp%n = n ; nlp%m_r = m_r ; nlp%m_c = m ; nlp%jr_ne = jr_ne ! dimensions
ALLOCATE( nlp%COHORT( n ), nlp%X( n ) )
nlp%COHORT = [ 1, 2, 0, 1, 2 ]
nlp%X = [ 0.5_wp, 0.5_wp, 0.5_wp, 0.5_wp, 0.5_wp ]
! sparse co-ordinate storage format
CALL SMT_put( nlp%jr_type, 'COORDINATE', s ) ! specify co-ordinate storage
ALLOCATE( nlp%jr_val( jr_ne ), nlp%jr_row( jr_ne ), nlp%jr_col( jr_ne ) )
nlp%jr_row = (/ 1, 1, 2, 2, 3, 3, 4, 4 /) ! Jacobian Jr(x)
nlp%jr_col = (/ 1, 2, 2, 3, 3, 4, 4, 5 /)
```

All use is subject to the conditions of a BSD-3-Clause License.
See <https://www.galahad.rl.ac.uk/download/> for full details.

```

      ALLOCATE( userdata%real( 1 ) )           ! allocate space for parameter
      userdata%real( 1 ) = p                   ! record parameter, p
! problem data complete ; solve using a Gauss-Newton model
      CALL SNLS_initialize( data, control, inform ) ! initialize control params
      control%jacobian_available = 2           ! jacobian is available
      control%print_level = 1
      control%print_obj = .TRUE.
      control%subproblem_solver = 1 ! use internal slls (2 for sllsb)
! control%SLLS_control%print_level = 1
      control%SLLS_control%SBLs_control%definite_linear_solver = 'potr '
      control%SLLS_control%SBLs_control%symmetric_linear_solver = 'sytr '
! control%SLLSB_control%print_level = 1
      control%SLLSB_control%symmetric_linear_solver = 'sytr '
      control%SLLSB_control%FDC_control%symmetric_linear_solver = 'sytr '
      inform%status = 1                       ! set for initial entry
      CALL SNLS_solve( nlp, control, inform, data, userdata,
                      eval_R = EVALR, eval_Jr = EVALJr )
      IF ( inform%status == 0 ) THEN           ! successful return
        WRITE( 6, "( ' SNLS: ', I0, ' iterations -',
          &      ' optimal objective value =',
          &      ES12.4, '/', ' Optimal solution = ', ( 5ES12.4 ) )" )
          inform%iter, inform%obj, nlp%X
      ELSE
        ! Error returns
        WRITE( 6, "( ' SNLS_solve exit status = ', I6 ) " ) inform%status
      END IF
      CALL SNLS_terminate( data, control, inform ) ! delete internal workspace
      DEALLOCATE( nlp%X, nlp%Y, nlp%Z, nlp%G, nlp%R, nlp%X_status )
      DEALLOCATE( nlp%COHORT, userdata%real )
      DEALLOCATE( nlp%Jr%type, nlp%Jr%val, nlp%Jr%row, nlp%Jr%col )
      END PROGRAM GALAHAD_SNLS_EXAMPLE

      SUBROUTINE EVALR( status, X, userdata, R ) ! residual
      USE GALAHAD_USERDATA_double
      INTEGER, PARAMETER :: wp = KIND( 1.0D+0 )
      INTEGER, INTENT( OUT ) :: status
      REAL ( KIND = wp ), DIMENSION( : ), INTENT( IN ) :: X
      REAL ( KIND = wp ), DIMENSION( : ), INTENT( OUT ) :: R
      TYPE ( USERDATA_type ), INTENT( INOUT ) :: userdata
      REAL ( KIND = wp ) :: p
      p = userdata%real( 1 )
      R( 1 ) = X( 1 ) * X( 2 ) - p
      R( 2 ) = X( 2 ) * X( 3 ) - 1.0_wp
      R( 3 ) = X( 3 ) * X( 4 ) - 1.0_wp
      R( 4 ) = X( 4 ) * X( 5 ) - 1.0_wp
      status = 0
      RETURN
      END SUBROUTINE EVALR

      SUBROUTINE EVALJr( status, X, userdata, Jr_val ) ! Jacobian
      USE GALAHAD_USERDATA_double
      INTEGER, PARAMETER :: wp = KIND( 1.0D+0 )
      INTEGER, INTENT( OUT ) :: status
      REAL ( KIND = wp ), DIMENSION( : ), INTENT( IN ) :: X
      REAL ( KIND = wp ), DIMENSION( : ), INTENT( OUT ) :: Jr_val
      TYPE ( USERDATA_type ), INTENT( INOUT ) :: userdata

```

All use is subject to the conditions of a BSD-3-Clause License.
See <https://www.galahad.rl.ac.uk/download/> for full details.

```

REAL ( KIND = wp ) :: p
p = userdata%real( 1 )
Jr_val( 1 ) = X( 2 )
Jr_val( 2 ) = X( 1 )
Jr_val( 3 ) = X( 3 )
Jr_val( 4 ) = X( 2 )
Jr_val( 5 ) = X( 4 )
Jr_val( 6 ) = X( 3 )
Jr_val( 7 ) = X( 5 )
Jr_val( 8 ) = X( 4 )
status = 0
RETURN
END SUBROUTINE EVALJr

```

Notice how the parameter p is passed to the function evaluation routines via the real component of the derived type `USERDATA_type`. The code produces the following output:

```

Problem:  (n = 5, m_r = 4, m_c = 2)
SNLS stopping tolerances (r,P[-J'r]) = 1.00E-06 1.00E-06

```

```

      (a=accept r=reject)
It      f      ||pg||      ratio  weight  step it slls #free  time
0      3.9686E+00 2.9881E-01
1 a     7.8455E+00 2.9881E-01 1.0E+00 1.0E+02 1.7E-02      3      5    0.00
2 a     7.5469E+00 2.9916E-01 1.1E+00 1.0E+01 1.6E-01      3      5    0.00
3 a     5.5003E+00 5.5680E-03 1.4E+00 1.0E+00 9.5E-01      3      3    0.00
4 a     5.5000E+00 5.0498E-04 1.0E+00 1.0E-01 2.2E-02      1      3    0.00
5 a     5.5000E+00 4.9987E-06 1.0E+00 1.0E-02 2.2E-03      1      3    0.00
6 a     5.5000E+00 4.9937E-09 1.0E+00 1.0E-03 2.2E-05      1      3    0.00

```

```

Problem:  (n = 5, m = 4)
SNLS stopping tolerances (r,P[-J'r]) = 1.00E-06 1.00E-06
Total time = .00 seconds

```

```

SNLS: 6 iterations - optimal objective value = 5.5000E+00
Optimal solution = 1.0000E+00 1.0000E+00 1.0000E+00 0.0000E+00 0.0000E+00

```

If the Jacobian is unavailable, but products of the form $\mathbf{J}_r(\mathbf{x})\mathbf{v}$ and $\mathbf{J}_r^T(\mathbf{x})\mathbf{v}$ are, the same problem may be solved as follows:

```

PROGRAM GALAHAD_SNLS_EXAMPLE2 ! GALAHAD 5.5 - 2026-05-04 AT 11:10 GMT.
USE GALAHAD_SNLS_double      ! double precision version
IMPLICIT NONE
INTEGER, PARAMETER :: wp = KIND( 1.0D+0 ) ! set precision
TYPE ( NLPT_problem_type ) :: nlp
TYPE ( SNLS_control_type ) :: control
TYPE ( SNLS_inform_type ) :: inform
TYPE ( SNLS_data_type ) :: data
TYPE ( USERDATA_type ) :: userdata
TYPE ( REVERSE_type ) :: reverse
INTEGER :: s
INTEGER, PARAMETER :: n = 5, m_r = 4, m_c = 2, jr_ne = 8
REAL ( KIND = wp ), PARAMETER :: p = 4.0_wp ! parameter p
! start problem data
nlp%n = n ; nlp%m_r = m_r ; nlp%m_c = m_c ; nlp%jr_ne = jr_ne ! dimensions
ALLOCATE( nlp%COHORT( n ), nlp%X( n ) )

```

All use is subject to the conditions of a BSD-3-Clause License.
See <https://www.galahad.rl.ac.uk/download/> for full details.

```

nlp%COHORT = [ 1, 2, 0, 1, 2 ]
! nlp%X = [ 1.0_wp, 1.0_wp, 1.0_wp, 0.0_wp, 0.0_wp ]
nlp%X = [ 0.5_wp, 0.5_wp, 0.5_wp, 0.5_wp, 0.5_wp ]
! sparse co-ordinate storage format
CALL SMT_put( nlp%Jr%type, 'COORDINATE', s ) ! specify co-ordinate storage
ALLOCATE( nlp%Jr%val( jr_ne ), nlp%Jr%row( jr_ne ), nlp%Jr%col( jr_ne ) )
nlp%Jr%row = (/ 1, 1, 2, 2, 3, 3, 4, 4 /) ! Jacobian Jr(x)
nlp%Jr%col = (/ 1, 2, 2, 3, 3, 4, 4, 5 /)
! problem data complete ; solve using a Gauss-Newton model
CALL SNLS_initialize( data, control, inform ) ! initialize control params
control%jacobian_available = 2 ! jacobian is available
control%print_level = 1
control%print_obj = .TRUE.
control%subproblem_solver = 1 ! use internal slls (2 for sllsb)
! control%SLLS_control%print_level = 1
control%SLLS_control%SBLs_control%definite_linear_solver = 'potr '
control%SLLS_control%SBLs_control%symmetric_linear_solver = 'sytr '
! control%SLLSB_control%print_level = 1
control%SLLSB_control%symmetric_linear_solver = 'sytr '
control%SLLSB_control%FDC_control%symmetric_linear_solver = 'sytr '
inform%status = 1 ! set for initial entry
DO
CALL SNLS_solve( nlp, control, inform, data, userdata, reverse = reverse )
SELECT CASE( inform%status )
CASE ( 0 ) ! successful return
WRITE( 6, "( ' SNLS: ', I0, ' iterations -', &
& ' optimal objective value =', &
& ES12.4, /, ' Optimal solution = ', ( 5ES12.4 ) )" ) &
inform%iter, inform%obj, nlp%X
EXIT
CASE( 2 ) ! evaluate residual
nlp%R( 1 ) = nlp%X( 1 ) * nlp%X( 2 ) - p
nlp%R( 2 ) = nlp%X( 2 ) * nlp%X( 3 ) - 1.0_wp
nlp%R( 3 ) = nlp%X( 3 ) * nlp%X( 4 ) - 1.0_wp
nlp%R( 4 ) = nlp%X( 4 ) * nlp%X( 5 ) - 1.0_wp
reverse%eval_status = 0
CASE( 3 ) ! evaluate Jacobian
nlp%Jr%val( 1 ) = nlp%X( 2 )
nlp%Jr%val( 2 ) = nlp%X( 1 )
nlp%Jr%val( 3 ) = nlp%X( 3 )
nlp%Jr%val( 4 ) = nlp%X( 2 )
nlp%Jr%val( 5 ) = nlp%X( 4 )
nlp%Jr%val( 6 ) = nlp%X( 3 )
nlp%Jr%val( 7 ) = nlp%X( 5 )
nlp%Jr%val( 8 ) = nlp%X( 4 )
reverse%eval_status = 0
CASE DEFAULT ! error returns
WRITE( 6, "( ' SNLS_solve exit status = ', I6 ) " ) inform%status
EXIT
END SELECT
END DO
! delete internal workspace
CALL SNLS_terminate( data, control, inform, reverse = reverse )
DEALLOCATE( nlp%X, nlp%Y, nlp%Z, nlp%G, nlp%R, nlp%X_status, nlp%COHORT )
DEALLOCATE( nlp%Jr%type, nlp%Jr%val, nlp%Jr%row, nlp%Jr%col )

```

All use is subject to the conditions of a BSD-3-Clause License.
See <https://www.galahad.rl.ac.uk/download/> for full details.

```
END PROGRAM GALAHAD_SNLS_EXAMPLE2
```

This produces the same output.

If the user prefers to provide function and Jacobian information without calls to specified routines, the following code is appropriate:

```
PROGRAM GALAHAD_SNLS_EXAMPLE3 ! GALAHAD 5.5 - 2026-05-04 AT 11:20 GMT
USE GALAHAD_SNLS_double ! double precision version
IMPLICIT NONE
INTEGER, PARAMETER :: wp = KIND( 1.0D+0 ) ! set precision
TYPE ( NLPT_problem_type ) :: nlp
TYPE ( SNLS_control_type ) :: control
TYPE ( SNLS_inform_type ) :: inform
TYPE ( SNLS_data_type ) :: data
TYPE ( USERDATA_type ) :: userdata
EXTERNAL :: EVALR, EVALJr_prod, EVALJr_scol, EVALJr_sprod
INTEGER, PARAMETER :: n = 5, m_r = 4, m_c = 2
REAL ( KIND = wp ), PARAMETER :: p = 4.0_wp ! parameter p
! start problem data
nlp%n = n ; nlp%m_r = m_r ; nlp%m_c = m_c ! dimensions
ALLOCATE( nlp%COHORT( n ), nlp%X( n ) )
nlp%COHORT = [ 1, 2, 0, 1, 2 ]
nlp%X = [ 0.5_wp, 0.5_wp, 0.5_wp, 0.5_wp, 0.5_wp ]
ALLOCATE( userdata%real( 1 ), userdata%integer( 2 ) ) ! space for parameters
userdata%real( 1 ) = p ! record parameter, p
userdata%integer( 1 ) = n ! record parameter, n
userdata%integer( 2 ) = m_r ! record parameter, m_r
! problem data complete ; solve using a Gauss-Newton model
CALL SNLS_initialize( data, control, inform ) ! initialize control params
control%jacobian_available = 1 ! jacobian by products
control%print_level = 1
control%print_obj = .TRUE.
control%subproblem_solver = 2 ! use internal slls (1 for sllsb)
! control%SLLS_control%print_level = 1
control%SLLS_control%SBLs_control%definite_linear_solver = 'potr '
control%SLLS_control%SBLs_control%symmetric_linear_solver = 'sytr '
inform%status = 1 ! set for initial entry
CALL SNLS_solve( nlp, control, inform, data, userdata, &
eval_R = EVALR, eval_Jr_prod = EVALJr_prod, &
eval_Jr_scol = EVALJr_scol, eval_Jr_sprod = EVALJr_sprod )
IF ( inform%status == 0 ) THEN ! successful return
WRITE( 6, "( ' SNLS: ', I0, ' iterations -', &
& ' optimal objective value =', &
& ES12.4, /, ' Optimal solution = ', ( 5ES12.4 ) )" ) &
inform%iter, inform%obj, nlp%X
ELSE ! Error returns
WRITE( 6, "( ' SNLS_solve exit status = ', I6 ) " ) inform%status
END IF
CALL SNLS_terminate( data, control, inform ) ! delete internal workspace
DEALLOCATE( nlp%X, nlp%Y, nlp%Z, nlp%G, nlp%R, nlp%X_status, nlp%COHORT )
DEALLOCATE( userdata%real, userdata%integer )
END PROGRAM GALAHAD_SNLS_EXAMPLE3

SUBROUTINE EVALR( status, X, userdata, R ) ! residual
USE GALAHAD_USERDATA_double
INTEGER, PARAMETER :: wp = KIND( 1.0D+0 )
```

All use is subject to the conditions of a BSD-3-Clause License.
See <https://www.galahad.rl.ac.uk/download/> for full details.

```
INTEGER, INTENT( OUT ) :: status
REAL ( KIND = wp ), DIMENSION( : ), INTENT( IN ) :: X
REAL ( KIND = wp ), DIMENSION( : ), INTENT( OUT ) :: R
TYPE ( USERDATA_type ), INTENT( INOUT ) :: userdata
REAL ( KIND = wp ) :: p
p = userdata%real( 1 )
R( 1 ) = X( 1 ) * X( 2 ) - p
R( 2 ) = X( 2 ) * X( 3 ) - 1.0_wp
R( 3 ) = X( 3 ) * X( 4 ) - 1.0_wp
R( 4 ) = X( 4 ) * X( 5 ) - 1.0_wp
status = 0
RETURN
END SUBROUTINE EVALR

SUBROUTINE EVALJr_prod( status, X, userdata, transpose, V, P, got_jr )
USE GALAHAD_USERDATA_double
INTEGER, PARAMETER :: wp = KIND( 1.0D+0 )
INTEGER, INTENT( OUT ) :: status
REAL ( KIND = wp ), DIMENSION( : ), INTENT( IN ) :: X
TYPE ( USERDATA_type ), INTENT( INOUT ) :: userdata
LOGICAL, INTENT( IN ) :: transpose
REAL ( KIND = wp ), DIMENSION( : ), INTENT( IN ) :: V
REAL ( KIND = wp ), DIMENSION( : ), INTENT( OUT ) :: P
LOGICAL, OPTIONAL, INTENT( IN ) :: got_jr
IF ( transpose ) THEN
  P( 1 ) = X( 2 ) * V( 1 )
  P( 2 ) = X( 3 ) * V( 2 ) + X( 1 ) * V( 1 )
  P( 3 ) = X( 4 ) * V( 3 ) + X( 2 ) * V( 2 )
  P( 4 ) = X( 5 ) * V( 4 ) + X( 3 ) * V( 3 )
  P( 5 ) = X( 4 ) * V( 4 )
ELSE
  P( 1 ) = X( 2 ) * V( 1 ) + X( 1 ) * V( 2 )
  P( 2 ) = X( 3 ) * V( 2 ) + X( 2 ) * V( 3 )
  P( 3 ) = X( 4 ) * V( 3 ) + X( 3 ) * V( 4 )
  P( 4 ) = X( 5 ) * V( 4 ) + X( 4 ) * V( 5 )
END IF
status = 0
RETURN
END SUBROUTINE EVALJr_prod

SUBROUTINE EVALJr_scol( status, X, userdata, index, VAL, ROW, nz, got_jr )
USE GALAHAD_USERDATA_double
INTEGER, PARAMETER :: wp = KIND( 1.0D+0 )
INTEGER, INTENT( OUT ) :: status
REAL ( KIND = wp ), DIMENSION( : ), INTENT( IN ) :: X
TYPE ( USERDATA_type ), INTENT( INOUT ) :: userdata
INTEGER, INTENT( IN ) :: index
REAL ( KIND = wp ), DIMENSION( : ), INTENT( OUT ) :: VAL
INTEGER, DIMENSION( : ), INTENT( INOUT ) :: ROW
INTEGER, INTENT( INOUT ) :: nz
LOGICAL, OPTIONAL, INTENT( IN ) :: got_jr
INTEGER :: n
n = userdata%integer( 1 )
IF ( index == 1 ) THEN
  VAL( 1 ) = X( 2 )
```

```

    ROW( 1 ) = 1
    nz = 1
ELSE IF ( index == n ) THEN
    VAL( 1 ) = X( n - 1 )
    ROW( 1 ) = n - 1
    nz = 1
ELSE
    VAL( 1 ) = X( index - 1 )
    ROW( 1 ) = index - 1
    VAL( 2 ) = X( index + 1 )
    ROW( 2 ) = index
    nz = 2
END IF
status = 0
RETURN
END SUBROUTINE EVALJr_scol

SUBROUTINE EVALJr_sprod( status, X, userdata, transpose, V, P, FREE,      &
                        n_free, got_jr )
USE GALAHAD_USERDATA_double
INTEGER, PARAMETER :: wp = KIND( 1.0D+0 )
INTEGER, INTENT( OUT ) :: status
REAL ( KIND = wp ), DIMENSION( : ), INTENT( IN ) :: X
TYPE ( USERDATA_type ), INTENT( INOUT ) :: userdata
LOGICAL, INTENT( IN ) :: transpose
REAL ( KIND = wp ), DIMENSION( : ), INTENT( IN ) :: V
REAL ( KIND = wp ), DIMENSION( : ), INTENT( OUT ) :: P
INTEGER, INTENT( IN ), DIMENSION( : ) :: FREE
INTEGER, INTENT( IN ) :: n_free
LOGICAL, OPTIONAL, INTENT( IN ) :: got_jr
INTEGER :: i, j, n, m_r
REAL ( KIND = wp ) :: val
n = userdata%integer( 1 )
m_r = userdata%integer( 2 )
IF ( transpose ) THEN
    DO i = 1, n_free
        j = FREE( i )
        IF ( j == 1 ) THEN
            P( 1 ) = X( 2 ) * V( 1 )
        ELSE IF ( j == n ) THEN
            P( n ) = X( m_r ) * V( m_r )
        ELSE
            P( j ) = X( j - 1 ) * V( j - 1 ) + X( j + 1 ) * V( j )
        END IF
    END DO
ELSE
    P( : m_r ) = 0.0_wp
    DO i = 1, n_free
        j = FREE( i )
        val = V( j )
        IF ( j == 1 ) THEN
            P( 1 ) = P( 1 ) + X( 2 ) * val
        ELSE IF ( j == n ) THEN
            P( m_r ) = P( m_r ) + X( m_r ) * val
        ELSE

```

All use is subject to the conditions of a BSD-3-Clause License.
See <https://www.galahad.rl.ac.uk/download/> for full details.

```
      P( j - 1 ) = P( j - 1 ) + X( j - 1 ) * val
      P( j ) = P( j ) + X( j + 1 ) * val
    END IF
  END DO
END IF
status = 0
RETURN
END SUBROUTINE EVALJr_sprod
```

This produces the same output as in the previous case.