C interfaces to GALAHAD NLS

Generated by Doxygen 1.8.17

Chapter 1

GALAHAD C package nls

1.1 Introduction

1.1.1 Purpose

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

$$\mathbf{f(x) := \frac{1}{2}\sum_{i=1}^{m} w_i c_i^2(x) \equiv \frac{1}{2}\|c(x)\|_W^2}$$

of many variables $\mathbf{x}$ involving positive weights $\mathbf{w_i}$, $\mathbf{i = 1, \ldots, m}$. The method offers the choice of direct and iterative solution of the key regularization subproblems, and is most suitable for large problems. First derivatives of the *residual function* $c(x)$ are required, and if second derivatives of the $c_i(x)$ can be calculated, they may be exploited—if suitable products of the first or second derivatives with a vector may be found but not the derivatives themselves, that can also be used to advantage.

1.1.2 Authors

N. I. M. Gould, STFC-Rutherford Appleton Laboratory, England.

C interface, additionally J. Fowkes, STFC-Rutherford Appleton Laboratory.

1.1.3 Originally released

October 2016, C interface August 2021.

1.1.4 Terminology

The *gradient* $\nabla_x f(x)$ of a function $f(x)$ is the vector whose i-th component is $\partial f(x)/\partial x_i$. The *Hessian* $\nabla_{xx} f(x)$ of $f(x)$ is the symmetric matrix whose i,j-th entry is $\partial^2 f(x)/\partial x_i \partial x_j$. The Hessian is *sparse* if a significant and useful proportion of the entries are universally zero.

The algorithm used by the package is iterative. From the current best estimate of the minimizer x_k, a trial improved point $x_k + s_k$ is sought. The correction s_k is chosen to improve a model $m_k(s)$ of the stabilised objective function $f_{\rho,p}(x_k + s)$ built around the objective function $f(x_k + s)$ built around x_k. The model is the sum of two basic components, a suitable approximation $t_k(s)$ of $f(x_k + s)$, another approximation of $(\rho/r)\|x_k + s\|_r^r$ (if $\rho > 0$), and a regularization term $(\sigma_k/p)\|s\|_{S_k}^p$ involving a weight σ_k, power p and a norm $\|s\|_{S_k} := \sqrt{s^T S_k s}$ for a given positive definite scaling matrix S_k that is included to prevent large corrections. The weight σ_k is adjusted as the algorithm progresses to ensure convergence.

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

$$J(x)_{i,j} := \partial c_i(x)/\partial x_j \text{ for } i = 1, \ldots, m \text{ and } j = 1, \ldots, n.$$

For a given m-vector y, the *weighted-residual Hessian* is the sum

$$H(x,y) := \sum_{\ell=1}^{m} y_\ell H_\ell(x), \text{ where } H_\ell(x)_{i,j} := \partial^2 c_\ell(x)/\partial x_i \partial x_j \text{ for } i,j = 1, \ldots, n$$

is the Hessian of $c_\ell(x)$. Finally, for a given vector v, we define the *residual-Hessians-vector product matrix*

$$P(x,v) := (H_1(x)v, \ldots, H_m(x)v).$$

The models $t_k(s)$ provided are,

1. the first-order Taylor approximation $f(x_k) + g(x_k)^T s$, where $g(x) = J^T(x) W c(x)$,
2. a barely second-order approximation $f(x_k) + g(x_k)^T s + \frac{1}{2} s^T W s$,
3. the Gauss-Newton approximation $\frac{1}{2}\|c(x_k) + J(x_k)s\|_W^2$,
4. the Newton (second-order Taylor) approximation $f(x_k)+g(x_k)^T s+\frac{1}{2}s^T[J^T(x_k)WJ(x_k)+H(x_k, Wc(x_k))]s$, and
5. the tensor Gauss-Newton approximation $\frac{1}{2}\|c(x_k) + J(x_k)s + \frac{1}{2}s^T \cdot P(x_k, s)\|_W^2$, where the i-th component of $s^T \cdot P(x_k, s)$ is shorthand for the scalar $s^T H_i(x_k) s$, where W is the diagonal matrix of weights w_i, $i = 1, \ldots m$.

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*").

1.1.5 Method

An adaptive regularization method is used. In this, an improvement to a current estimate of the required minimizer, x_k is sought by computing a step s_k. The step is chosen to approximately minimize a model $t_k(s)$ of $f_{\rho,r}(x_k + s)$ that includes a weighted regularization term $(\sigma_k/p)\|s\|_{S_k}^p$ for some specified positive weight σ_k. The quality of the resulting step s_k is assessed by computing the "ratio" % $(f_{\rho,p}(x_k) - f_{\rho,p}(x_k + s_k))/(t_k(0) - t_k(s_k))$. $(f(x_k) - f(x_k + s_k))/(t_k(0) - t_k(s_k))$. The step is deemed to have succeeded if the ratio exceeds a given $\eta_s > 0$, and in this case $x_{k+1} = x_k + s_k$. Otherwise $x_{k+1} = x_k$, and the weight is increased by powers of a given increase factor up to a given limit. If the ratio is larger than $\eta_v \geq \eta_d$, 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 $f(x_k)$ or $\|\nabla_x f(x_k)\|$ is smaller than a specified value.

A choice of linear, quadratic or quartic models $t_k(s)$ is available (see the Terminology section), and normally a two-norm regularization will be used, but this may change if preconditioning is employed.

If linear or quadratic models are employed, an appropriate, approximate model minimizer is found using either a direct approach involving factorization of a shift of the model Hessian B_k or an iterative (conjugate-gradient/Lanczos) approach based on approximations to the required solution from a so-called Krlov subspace. The direct approach is based on the knowledge that the required solution satisfies the linear system of equations $(B_k + \lambda_k I)s_k = -\nabla_x f(x_k)$ involving a scalar Lagrange multiplier λ_k. This multiplier is found by uni-variate root finding, using a safeguarded Newton-like process, by the GALAHAD packages RQS. The iterative approach uses the GALA↩ HAD packag GLRT, and is best accelerated by preconditioning with good approximations to the Hessian of the model using GALAHAD's PSLS. The iterative approach has the advantage that only Hessian matrix-vector products are required, and thus the Hessian B_k is not required explicitly. However when factorizations of the Hessian are possible, the direct approach is often more efficient.

When a quartic model is used, the model is itself of least-squares form, and the package calls itself recursively to approximately minimize its model. The quartic model often gives a better approximation, but at the cost of more involved derivative requirements.

1.1.6 Reference

The generic adaptive cubic regularization method is described in detail in

C. Cartis, N. I. M. Gould and Ph. L. Toint, ``Adaptive cubic regularisation methods for unconstrained optimization. Part I: motivation, convergence and numerical results'', Mathematical Programming 127(2) (2011) 245-295,

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, J. A. Scott and T. Rees, ``Convergence and evaluation-complexity analysis of a regularized tensor-Newton method for solving nonlinear least-squares problems''. Computational Optimization and Applications 73(1) (2019) 1–35.

1.1.7 Call order

To solve a given problem, functions from the nls package must be called in the following order:

- nls_initialize - provide default control parameters and set up initial data structures
- nls_read_specfile (optional) - override control values by reading replacement values from a file
- nls_import - set up problem data structures and fixed values
- arc_reset_control (optional) - possibly change control parameters if a sequence of problems are being solved
- solve the problem by calling one of
  - nls_solve_with_mat - solve using function calls to evaluate function, gradient and Hessian values
  - nls_solve_without_mat - solve using function calls to evaluate function and gradient values and Hessian-vector products

  - nls_solve_reverse_with_mat - solve returning to the calling program to obtain function, gradient and Hessian values, or
  - nls_solve_reverse_without_mat - solve returning to the calling prorgram to obtain function and gradient values and Hessian-vector products
- nls_information (optional) - recover information about the solution and solution process
- nls_terminate - deallocate data structures

See Section 6.1 for examples of use.

1.1.8 Unsymmetric matrix storage formats

The unsymmetric m by n Jacobian matrix $J \equiv \nabla_x c(x)$ and the residual-Hessians-vector product matrix $P(x,v)$ may be presented and stored in a variety of convenient input formats. Let A be J or P as appropriate.

Both C-style (0 based) and fortran-style (1-based) indexing is allowed. Choose `control.f_indexing` as `false` for C style and `true` for fortran style; the discussion below presumes C style, but add 1 to indices for the corresponding fortran version.

Wrappers will automatically convert between 0-based (C) and 1-based (fortran) array indexing, so may be used transparently from C. This conversion involves both time and memory overheads that may be avoided by supplying data that is already stored using 1-based indexing.

1.1.8.1 Dense storage format

The matrix A 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. In this case, component $n * i + j$ of the storage array A_val will hold the value A_{ij} for $0 \leq i \leq m-1$, $0 \leq j \leq n-1$.

1.1.8.2 Dense by columns storage format

The matrix A 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. In this case, component $m * j + i$ of the storage array A_val will hold the value A_{ij} for $0 \leq i \leq m-1$, $0 \leq j \leq n-1$.

1.1.8.3 Sparse co-ordinate storage format

Only the nonzero entries of the matrices are stored. For the l-th entry, $0 \leq l \leq ne-1$, of A, its row index i, column index j and value A_{ij}, $0 \leq i \leq m-1$, $0 \leq j \leq n-1$, are stored as the l-th components of the integer arrays A_row and A_col and real array A_val, respectively, while the number of nonzeros is recorded as A_ne = ne.

1.1.8.4 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 A the i-th component of the integer array A_ptr holds the position of the first entry in this row, while A_ptr(m) holds the total number of entries plus one. The column indices j, $0 \leq j \leq n-1$, and values A_{ij} of the nonzero entries in the i-th row are stored in components l = A_ptr(i), ..., A_ptr(i+1)-1, $0 \leq i \leq m-1$, of the integer array A_col, and real array A_val, respectively. For sparse matrices, this scheme almost always requires less storage than its predecessor.

1.1.8.5 Sparse column-wise storage format

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 A the j-th component of the integer array A_ptr holds the position of the first entry in this column, while A_ptr(n) holds the total number of entries plus one. The row indices i, $0 \leq i \leq m-1$, and values A_{ij} of the nonzero entries in the j-th columnsare stored in components l = A_ptr(j), . . ., A_ptr(j+1)-1, $0 \leq j \leq n-1$, of the integer array A_row, and real array A_val, respectively. As before, for sparse matrices, this scheme almost always requires less storage than the co-ordinate format.

1.1.9 Symmetric matrix storage formats

Likewise, the symmetric n by n weighted-residual Hessian matrix $H = H(x, y)$ may be presented and stored in a variety of formats. But crucially symmetry is exploited by only storing values from the lower triangular part (i.e, those entries that lie on or below the leading diagonal).

1.1.9.1 Dense storage format

The matrix H 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. Since H is symmetric, only the lower triangular part (that is the part h_{ij} for $0 \leq j \leq i \leq n-1$) need be held. In this case the lower triangle should be stored by rows, that is component $i * i/2 + j$ of the storage array H_val will hold the value h_{ij} (and, by symmetry, h_{ji}) for $0 \leq j \leq i \leq n-1$.

1.1.9.2 Sparse co-ordinate storage format

Only the nonzero entries of the matrices are stored. For the l-th entry, $0 \leq l \leq ne-1$, of H, its row index i, column index j and value h_{ij}, $0 \leq j \leq i \leq n-1$, are stored as the l-th components of the integer arrays H_row and H_col and real array H_val, respectively, while the number of nonzeros is recorded as H_ne = ne. Note that only the entries in the lower triangle should be stored.

1.1.9.3 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 H the i-th component of the integer array H_ptr holds the position of the first entry in this row, while H_ptr(n) holds the total number of entries plus one. The column indices j, $0 \leq j \leq i$, and values h_{ij} of the entries in the i-th row are stored in components l = H_ptr(i), . . ., H_ptr(i+1)-1 of the integer array H_col, and real array H_val, respectively. Note that as before only the entries in the lower triangle should be stored. For sparse matrices, this scheme almost always requires less storage than its predecessor.

1.1.9.4 Diagonal storage format

If H is diagonal (i.e., $H_{ij} = 0$ for all $0 \leq i \neq j \leq n-1$) only the diagonals entries H_{ii}, $0 \leq i \leq n-1$ need be stored, and the first n components of the array H_val may be used for the purpose.

1.1.9.5 Multiples of the identity storage format

If H is a multiple of the identity matrix, (i.e., $H = \alpha I$ where I is the n by n identity matrix and α is a scalar), it suffices to store α as the first component of H_val.

1.1.9.6 The identity matrix format

If H is the identity matrix, no values need be stored.

1.1.9.7 The zero matrix format

The same is true if H is the zero matrix.

Chapter 2

Data Structure Index

2.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 4

Data Structure Documentation

4.1 nls_control_type Struct Reference

`#include <nls.h>`

Data Fields

- bool f_indexing

  use C or Fortran sparse matrix indexing
- int error

  error and warning diagnostics occur on stream error
- int out

  general output occurs on stream out
- int print_level

  the level of output required.
- int start_print

  any printing will start on this iteration
- int stop_print

  any printing will stop on this iteration
- int print_gap

  the number of iterations between printing
- int maxit

  the maximum number of iterations performed
- int alive_unit

  removal of the file alive_file from unit alive_unit terminates execution
- char alive_file [31]

  see alive_unit
- int jacobian_available

  is the Jacobian matrix of first derivatives available ($\geq$ 2), is access only via matrix-vector products (=1) or is it not available ($\leq$ 0) ?
- int hessian_available

  is the Hessian matrix of second derivatives available ($\geq$ 2), is access only via matrix-vector products (=1) or is it not available ($\leq$ 0) ?
- int model

  the model used.

- int norm

  the regularization norm used.

- int non_monotone

  non-monotone $\leq$ 0 monotone strategy used, anything else non-monotone strategy with this history length used

- int weight_update_strategy

  define the weight-update strategy: 1 (basic), 2 (reset to zero when very successful), 3 (imitate TR), 4 (increase lower bound), 5 (GPT)

- real_wp_ stop_c_absolute

  overall convergence tolerances. The iteration will terminate when $||c(x)||_2 \leq$ MAX(.stop_c_absolute, .stop_c_relative $*||c(x_{initial})||_2$ or when the norm of the gradient, $g = J^T(x)c(x)/||c(x)||_2$, of ||c(x)||_2 satisfies $||g||_2 \leq$ MAX(.stop_g_absolute, .stop_g_relative $*||g_{initial}||_2$, or if the step is less than .stop_s

- real_wp_ stop_c_relative

  see stop_c_absolute

- real_wp_ stop_g_absolute

  see stop_c_absolute

- real_wp_ stop_g_relative

  see stop_c_absolute

- real_wp_ stop_s

  see stop_c_absolute

- real_wp_ power

  the regularization power (<2 => chosen according to the model)

- real_wp_ initial_weight

  initial value for the regularization weight (-ve => $1/\|g_0\|$))

- real_wp_ minimum_weight

  minimum permitted regularization weight

- real_wp_ initial_inner_weight

  initial value for the inner regularization weight for tensor GN (-ve => 0)

- real_wp_ eta_successful

  REAL (KIND = wp) :: initial_inner_weight = 0.0001_wp a potential iterate will only be accepted if the actual decrease f - f(x_new) is larger than .eta_successful times that predicted by a quadratic model of the decrease. The regularization weight will be decreaed if this relative decrease is greater than .eta_very_successful but smaller than .eta_too_successful.

- real_wp_ eta_very_successful

  see eta_successful

- real_wp_ eta_too_successful

  see eta_successful

- real_wp_ weight_decrease_min

  on very successful iterations, the regularization weight will be reduced by the factor .weight_decrease but no more than .weight_decrease_min while if the iteration is unsucceful, the weight will be increased by a factor .weight_increase but no more than .weight_increase_max (these are delta_1, delta_2, delta3 and delta_max in Gould, Porcelli and Toint, 2011)

- real_wp_ weight_decrease

  REAL (KIND = wp) :: weight_decrease = half.

- real_wp_ weight_increase

  REAL (KIND = wp) :: weight_increase = two.

- real_wp_ weight_increase_max

  see weight_increase

- real_wp_ reduce_gap

  expert parameters as suggested in Gould, Porcelli and Toint, "Updating t regularization parameter in the adaptive cubic regularization algorithm" RAL-TR-2011-007, Rutherford Appleton Laboratory, England (2011), http://epubs.stfc.ac.uk/bitstream/6181/RAL-TR-2011-007.pdf (these are denoted beta, epsilon_chi and alpha_max in the paper)

- real_wp_ tiny_gap

  see reduce_gap
- real_wp_ large_root

  see reduce_gap
- real_wp_ switch_to_newton

  if the Gauss-Newto to Newton model is specified, switch to Newton as soon as the norm of the gradient g is smaller than switch_to_newton
- real_wp_ cpu_time_limit

  the maximum CPU time allowed (-ve means infinite)
- real_wp_ clock_time_limit

  the maximum elapsed clock time allowed (-ve means infinite)
- bool subproblem_direct

  use a direct (factorization) or (preconditioned) iterative method to find the search direction
- bool renormalize_weight

  should the weight be renormalized to account for a change in scaling?
- bool magic_step

  allow the user to perform a "magic" step to improve the objective
- bool print_obj

  print values of the objective/gradient rather than $||c||$ and its gradien
- bool space_critical

  if .space_critical true, every effort will be made to use as little space as possible. This may result in longer computation time
- bool deallocate_error_fatal

  if .deallocate_error_fatal is true, any array/pointer deallocation error will terminate execution. Otherwise, computation will continue
- char prefix [31]

  all output lines will be prefixed by .prefix(2:LEN(TRIM(.prefix))-1) where .prefix contains the required string enclosed in quotes, e.g. "string" or 'string'
- struct nls_subproblem_control_type subproblem_control

  control parameters for the step-finding subproblem

4.1.1 Detailed Description

control derived type as a C struct

Examples

nlst.c, and nlstf.c.

4.1.2 Field Documentation

4.1.2.1 model

`int model`

the model used.

Possible values are

- 0 dynamic (*not yet implemented*)
- 1 first-order (no Hessian)
- 2 barely second-order (identity Hessian)
- 3 Gauss-Newton ($J^T J$ Hessian)
- 4 second-order (exact Hessian)
- 5 Gauss-Newton to Newton transition
- 6 tensor Gauss-Newton treated as a least-squares model
- 7 tensor Gauss-Newton treated as a general model
- 8 tensor Gauss-Newton transition from a least-squares to a general mode

4.1.2.2 norm

`int norm`

the regularization norm used.

The norm is defined via $\|v\|^2 = v^T S v$, and will define the preconditioner used for iterative methods. Possible values for S are

- -3 user's own regularization norm
- -2 S = limited-memory BFGS matrix (with .PSLS_control.lbfgs_vectors history) (*not yet implemented*)
- -1 identity (= Euclidan two-norm)
- 0 automatic (*not yet implemented*)
- 1 diagonal, S = diag(max($J^T J$ Hessian, .PSLS_contro.min_diagonal))
- 2 diagonal, S = diag(max(Hessian, .PSLS_contro.min_diagonal))
- 3 banded, S = band(Hessian) with semi-bandwidth .PSLS_control.semi_bandwidth
- 4 re-ordered band, P=band(order(A)) with semi-bandwidth .PSLS_control.semi_bandwidth
- 5 full factorization, S = Hessian, Schnabel-Eskow modification
- 6 full factorization, S = Hessian, GMPS modification (*not yet implemented*)
- 7 incomplete factorization of Hessian, Lin-More'
- 8 incomplete factorization of Hessian, HSL_MI28
- 9 incomplete factorization of Hessian, Munskgaard (*not yet implemented*)
- 10 expanding band of Hessian (*not yet implemented*)

4.1.2.3 print_level

```
int print_level
```

the level of output required.

- $\leq$ 0 gives no output,
- = 1 gives a one-line summary for every iteration,
- = 2 gives a summary of the inner iteration for each iteration,
- $\geq$ 3 gives increasingly verbose (debugging) output

The documentation for this struct was generated from the following file:

- nls/nls.h

4.2 nls_inform_type Struct Reference

```
#include <nls.h>
```

Data Fields

- int status
  return status. See NLS_solve for details
- int alloc_status
  the status of the last attempted allocation/deallocation
- char bad_alloc [81]
  the name of the array for which an allocation/deallocation error ocurred
- char bad_eval [13]
  the name of the user-supplied evaluation routine for which an error ocur
- int iter
  the total number of iterations performed
- int cg_iter
  the total number of CG iterations performed
- int c_eval
  the total number of evaluations of the residual function c(x)
- int j_eval
  the total number of evaluations of the Jacobian J(x) of c(x)
- int h_eval
  the total number of evaluations of the scaled Hessian H(x,y) of c(x)
- int factorization_max
  the maximum number of factorizations in a sub-problem solve
- int factorization_status
  the return status from the factorization
- int max_entries_factors
  the maximum number of entries in the factors
- int factorization_integer

the total integer workspace required for the factorization

- int factorization_real

  the total real workspace required for the factorization

- real_wp_ factorization_average

  the average number of factorizations per sub-problem solve

- real_wp_ obj

  the value of the objective function $\frac{1}{2}\|c(x)\|_W^2$ at the best estimate the solution, x, determined by NLS_solve

- real_wp_ norm_c

  the norm of the residual $\|c(x)\|_W$ at the best estimate of the solution x, determined by NLS_solve

- real_wp_ norm_g

  the norm of the gradient of $\|c(x)\|_W$ of the objective function at the best estimate, x, of the solution determined by NLS_solve

- real_wp_ weight

  the final regularization weight used

- struct nls_time_type time

  timings (see above)

- struct nls_subproblem_inform_type subproblem_inform

  inform parameters for subproblem

4.2.1 Detailed Description

inform derived type as a C struct

Examples

nlst.c, and nlstf.c.

The documentation for this struct was generated from the following file:

- nls/nls.h

4.3 nls_subproblem_control_type Struct Reference

`#include <nls.h>`

Data Fields

- int error

  error and warning diagnostics occur on stream error

- int out

  general output occurs on stream out

- int print_level

  the level of output required.

- int start_print

  any printing will start on this iteration

- int stop_print

  any printing will stop on this iteration

- int print_gap

  the number of iterations between printing
- int maxit

  the maximum number of iterations performed
- int alive_unit

  removal of the file alive_file from unit alive_unit terminates execution
- char alive_file [31]

  see alive_unit
- int jacobian_available

  is the Jacobian matrix of first derivatives available ($\geq$ 2), is access only via matrix-vector products (=1) or is it not available ($\leq$ 0) ?
- int hessian_available

  is the Hessian matrix of second derivatives available ($\geq$ 2), is access only via matrix-vector products (=1) or is it not available ($\leq$ 0) ?
- int model

  the model used.
- int norm

  the regularization norm used.
- int non_monotone

  non-monotone $\leq$ 0 monotone strategy used, anything else non-monotone strategy with this history length used
- int weight_update_strategy

  define the weight-update strategy: 1 (basic), 2 (reset to zero when very successful), 3 (imitate TR), 4 (increase lower bound), 5 (GPT)
- real_wp_ stop_c_absolute

  overall convergence tolerances. The iteration will terminate when $||c(x)||_2 \leq$ MAX(.stop_c_absolute, .stop_c_relative $*\|c(x_{initial})\|_2$, or when the norm of the gradient, $g = J^T(x)c(x)/\|c(x)\|_2$, of ||c||_2, satisfies $\|g\|_2 \leq$ MAX(.stop_g_absolute, .stop_g_relative $*\|g_{initial}\|_2$, or if the step is less than .stop_s
- real_wp_ stop_c_relative

  see stop_c_absolute
- real_wp_ stop_g_absolute

  see stop_c_absolute
- real_wp_ stop_g_relative

  see stop_c_absolute
- real_wp_ stop_s

  see stop_c_absolute
- real_wp_ power

  the regularization power (<2 => chosen according to the model)
- real_wp_ initial_weight

  initial value for the regularization weight (-ve => $1/\|g_0\|$))
- real_wp_ minimum_weight

  minimum permitted regularization weight
- real_wp_ initial_inner_weight

  initial value for the inner regularization weight for tensor GN (-ve => 0)
- real_wp_ eta_successful

  REAL (KIND = wp) :: initial_inner_weight = 0.0001_wp a potential iterate will only be accepted if the actual decrease f - f(x_new) is larger than .eta_successful times that predicted by a quadratic model of the decrease. The regularization weight will be decreaed if this relative decrease is greater than .eta_very_successful but smaller than .eta_too_successful.
- real_wp_ eta_very_successful

  see eta_successful
- real_wp_ eta_too_successful

see eta_successful

- real_wp_ weight_decrease_min

  on very successful iterations, the regularization weight will be reduced by the factor .weight_decrease but no more than .weight_decrease_min while if the iteration is unsucceful, the weight will be increased by a factor .weight_↩ increase but no more than .weight_increase_max (these are delta_1, delta_2, delta3 and delta_max in Gould, Porcelli and Toint, 2011)

- real_wp_ weight_decrease

  REAL (KIND = wp) :: weight_decrease = half.

- real_wp_ weight_increase

  REAL (KIND = wp) :: weight_increase = two.

- real_wp_ weight_increase_max

  see weight_increase

- real_wp_ reduce_gap

  expert parameters as suggested in Gould, Porcelli and Toint, "Updating t regularization parameter in the adaptive cubic regularization algorithm" RAL-TR-2011-007, Rutherford Appleton Laboratory, England (2011), http://epubs.stfc.ac.uk/bitstream/6181/RAL-TR-2011-007.pdf (these are denoted beta, epsilon_chi and alpha_max in the paper)

- real_wp_ tiny_gap

  see reduce_gap

- real_wp_ large_root

  see reduce_gap

- real_wp_ switch_to_newton

  if the Gauss-Newto to Newton model is specified, switch to Newton as soon as the norm of the gradient g is smaller than switch_to_newton

- real_wp_ cpu_time_limit

  the maximum CPU time allowed (-ve means infinite)

- real_wp_ clock_time_limit

  the maximum elapsed clock time allowed (-ve means infinite)

- bool subproblem_direct

  use a direct (factorization) or (preconditioned) iterative method to find the search direction

- bool renormalize_weight

  should the weight be renormalized to account for a change in scaling?

- bool magic_step

  allow the user to perform a "magic" step to improve the objective

- bool print_obj

  print values of the objective/gradient rather than $||c||$ and its gradien

- bool space_critical

  if .space_critical true, every effort will be made to use as little space as possible. This may result in longer computation time

- bool deallocate_error_fatal

  if .deallocate_error_fatal is true, any array/pointer deallocation error will terminate execution. Otherwise, computation will continue

- char prefix [31]

  all output lines will be prefixed by .prefix(2:LEN(TRIM(.prefix))-1) where .prefix contains the required string enclosed in quotes, e.g. "string" or 'string'

4.3.1 Detailed Description

control derived type as a C struct

4.3.2 Field Documentation

4.3.2.1 model

`int model`

the model used.

Possible values are

- 0 dynamic (*not yet implemented*)
- 1 first-order (no Hessian)
- 2 barely second-order (identity Hessian)
- 3 Gauss-Newton ($J^T J$ Hessian)
- 4 second-order (exact Hessian)
- 5 Gauss-Newton to Newton transition
- 6 tensor Gauss-Newton treated as a least-squares model
- 7 tensor Gauss-Newton treated as a general model
- 8 tensor Gauss-Newton transition from a least-squares to a general mode

4.3.2.2 norm

`int norm`

the regularization norm used.

The norm is defined via $\|v\|^2 = v^T S v$, and will define the preconditioner used for iterative methods. Possible values for S are

- -3 user's own regularization norm
- -2 S = limited-memory BFGS matrix (with .PSLS_control.lbfgs_vectors history) (*not yet implemented*)
- -1 identity (= Euclidan two-norm)
- 0 automatic (*not yet implemented*)
- 1 diagonal, S = diag(max($J^T J$ Hessian, .PSLS_contro.min_diagonal))
- 2 diagonal, S = diag(max(Hessian, .PSLS_contro.min_diagonal))
- 3 banded, S = band(Hessian) with semi-bandwidth .PSLS_control.semi_bandwidth
- 4 re-ordered band, P=band(order(A)) with semi-bandwidth .PSLS_control.semi_bandwidth
- 5 full factorization, S = Hessian, Schnabel-Eskow modification
- 6 full factorization, S = Hessian, GMPS modification (*not yet implemented*)
- 7 incomplete factorization of Hessian, Lin-More'
- 8 incomplete factorization of Hessian, HSL_MI28
- 9 incomplete factorization of Hessian, Munskgaard (*not yet implemented*)
- 10 expanding band of Hessian (*not yet implemented*)

4.3.2.3 print_level

```
int print_level
```

the level of output required.

- $\leq$ 0 gives no output,
- = 1 gives a one-line summary for every iteration,
- = 2 gives a summary of the inner iteration for each iteration,
- $\geq$ 3 gives increasingly verbose (debugging) output

The documentation for this struct was generated from the following file:

- nls/nls.h

4.4 nls_subproblem_inform_type Struct Reference

```
#include <nls.h>
```

Data Fields

- int status
  return status. See NLS_solve for details
- int alloc_status
  the status of the last attempted allocation/deallocation
- char bad_alloc [81]
  the name of the array for which an allocation/deallocation error ocurred
- char bad_eval [13]
  the name of the user-supplied evaluation routine for which an error ocur
- int iter
  the total number of iterations performed
- int cg_iter
  the total number of CG iterations performed
- int c_eval
  the total number of evaluations of the residual function c(x)
- int j_eval
  the total number of evaluations of the Jacobian J(x) of c(x)
- int h_eval
  the total number of evaluations of the scaled Hessian H(x,y) of c(x)
- int factorization_max
  the maximum number of factorizations in a sub-problem solve
- int factorization_status
  the return status from the factorization
- int max_entries_factors
  the maximum number of entries in the factors
- int factorization_integer

the total integer workspace required for the factorization

- int factorization_real

  the total real workspace required for the factorization
- real_wp_ factorization_average

  the average number of factorizations per sub-problem solve
- real_wp_ obj

  the value of the objective function $\frac{1}{2}\|c(x)\|_W^2$ at the best estimate the solution, x, determined by NLS_solve
- real_wp_ norm_c

  the norm of the residual $\|c(x)\|_W$ at the best estimate of the solution x, determined by NLS_solve
- real_wp_ norm_g

  the norm of the gradient of $\|c(x)\|_W$ of the objective function at the best estimate, x, of the solution determined by NLS_solve
- real_wp_ weight

  the final regularization weight used
- struct nls_time_type time

  timings (see above)

4.4.1 Detailed Description

inform derived type as a C struct

The documentation for this struct was generated from the following file:

- nls/nls.h

4.5 nls_time_type Struct Reference

`#include <nls.h>`

Data Fields

- real_sp_ total

  the total CPU time spent in the package
- real_sp_ preprocess

  the CPU time spent preprocessing the problem
- real_sp_ analyse

  the CPU time spent analysing the required matrices prior to factorizatio
- real_sp_ factorize

  the CPU time spent factorizing the required matrices
- real_sp_ solve

  the CPU time spent computing the search direction
- real_wp_ clock_total

  the total clock time spent in the package
- real_wp_ clock_preprocess

  the clock time spent preprocessing the problem
- real_wp_ clock_analyse

  the clock time spent analysing the required matrices prior to factorizat
- real_wp_ clock_factorize

  the clock time spent factorizing the required matrices
- real_wp_ clock_solve

  the clock time spent computing the search direction

4.5.1 Detailed Description

time derived type as a C struct

The documentation for this struct was generated from the following file:

- nls/nls.h

Chapter 5

File Documentation

5.1 nls/nls.h File Reference

```
#include <stdbool.h>
#include "galahad_precision.h"
```

Data Structures

- struct nls_subproblem_control_type
- struct nls_control_type
- struct nls_time_type
- struct nls_subproblem_inform_type
- struct nls_inform_type

Functions

- void nls_initialize (void **data, struct nls_control_type *control, struct nls_inform_type *inform)
- void nls_read_specfile (struct nls_control_type *control, const char specfile[])
- void nls_import (struct nls_control_type *control, void **data, int *status, int n, int m, const char J_type[], int J_ne, const int J_row[], const int J_col[], const int J_ptr[], const char H_type[], int H_ne, const int H_row[], const int H_col[], const int H_ptr[], const char P_type[], int P_ne, const int P_row[], const int P_col[], const int P_ptr[], const real_wp_ w[])
- void nls_reset_control (struct nls_control_type *control, void **data, int *status,)
- void nls_solve_with_mat (void **data, void *userdata, int *status, int n, int m, real_wp_ x[], real_wp_ c[], real_wp_ g[], int(*eval_c)(int, int, const real_wp_[], real_wp_[], const void *), int j_ne, int(*eval_j)(int, int, int, const real_wp_[], real_wp_[], const void *), int h_ne, int(*eval_h)(int, int, int, const real_wp_[], const real_↩ wp_[], real_wp_[], const void *), int p_ne, int(*eval_hprods)(int, int, int, const real_wp_[], const real_wp_[], real_wp_[], bool, const void *))
- void nls_solve_without_mat (void **data, void *userdata, int *status, int n, int m, real_wp_ x[], real_wp_ c[], real_wp_ g[], int(*eval_c)(int, int, const real_wp_[], real_wp_[], const void *), int(*eval_jprod)(int, int, const real_wp_[], const bool, real_wp_[], const real_wp_[], bool, const void *), int(*eval_hprod)(int, int, const real↩ _wp_[], const real_wp_[], real_wp_[], const real_wp_[], bool, const void *), int p_ne, int(*eval_hprods)(int, int, int, const real_wp_[], const real_wp_[], real_wp_[], bool, const void *))
- void nls_solve_reverse_with_mat (void **data, int *status, int *eval_status, int n, int m, real_wp_ x[], real↩ _wp_ c[], real_wp_ g[], int j_ne, real_wp_ J_val[], const real_wp_ y[], int h_ne, real_wp_ H_val[], real_wp_ v[], int p_ne, real_wp_ P_val[])
- void nls_solve_reverse_without_mat (void **data, int *status, int *eval_status, int n, int m, real_wp_ x[], real_wp_ c[], real_wp_ g[], bool *transpose, real_wp_ u[], real_wp_ v[], real_wp_ y[], int p_ne, real_wp_ P_val[])
- void nls_information (void **data, struct nls_inform_type *inform, int *status)
- void nls_terminate (void **data, struct nls_control_type *control, struct nls_inform_type *inform)

5.1.1 Function Documentation

5.1.1.1 nls_import()

```
void nls_import (
            struct nls_control_type * control,
            void ** data,
            int * status,
            int n,
            int m,
            const char J_type[],
            int J_ne,
            const int J_row[],
            const int J_col[],
            const int J_ptr[],
            const char H_type[],
            int H_ne,
            const int H_row[],
            const int H_col[],
            const int H_ptr[],
            const char P_type[],
            int P_ne,
            const int P_row[],
            const int P_col[],
            const int P_ptr[],
            const real_wp_ w[] )
```

Import problem data into internal storage prior to solution.

Parameters

in	*control*	is a struct whose members provide control paramters for the remaining prcedures (see nls_control_type)
in,out	*data*	holds private internal data
in,out	*status*	is a scalar variable of type int, that gives the exit status from the package. Possible values are: • 1. The import was succesful, and the package is ready for the solve phase • -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 restrictions n $>$ 0, m $>$ 0 or requirement that J/H/P_type contains its relevant string 'dense', 'dense_by_columns', 'coordinate', 'sparse_by_rows', 'sparse_by_columns', 'diagonal' or 'absent' has been violated.
in	*n*	is a scalar variable of type int, that holds the number of variables.
in	*m*	is a scalar variable of type int, that holds the number of residuals.

Parameters

in	*J_type*	is a one-dimensional array of type char that specifies the symmetric storage scheme used for the Jacobian, \bf$J\bf$. It should be one of 'coordinate', 'sparse_by_rows', 'dense' or 'absent', the latter if access to the Jacobian is via matrix-vector products; lower or upper case variants are allowed.
in	*J_ne*	is a scalar variable of type int, that holds the number of entries in \bf$J\bf$ in the sparse co-ordinate storage scheme. It need not be set for any of the other schemes.
in	*J_row*	is a one-dimensional array of size J_ne and type int, that holds the row indices of \bf$J\bf$ in the sparse co-ordinate storage scheme. It need not be set for any of the other schemes, and in this case can be NULL.
in	*J_col*	is a one-dimensional array of size J_ne and type int, that holds the column indices of \bf$J\bf$ in either the sparse co-ordinate, or the sparse row-wise storage scheme. It need not be set when the dense or diagonal storage schemes are used, and in this case can be NULL.
in	*J_ptr*	is a one-dimensional array of size n+1 and type int, that holds the starting position of each row of J, as well as the total number of entries plus one, in the sparse row-wise storage scheme. It need not be set when the other schemes are used, and in this case can be NULL.
in	*H_type*	is a one-dimensional array of type char that specifies the symmetric storage scheme used for the Hessian, H. It should be one of 'coordinate', 'sparse_by_rows', 'dense', 'diagonal' or 'absent', the latter if access to H is via matrix-vector products; lower or upper case variants are allowed.
in	*H_ne*	is a scalar variable of type int, that holds the number of entries in the lower triangular part of H in the sparse co-ordinate storage scheme. It need not be set for any of the other three schemes.
in	*H_row*	is a one-dimensional array of size H_ne and type int, that holds the row indices of the lower triangular part of H in the sparse co-ordinate storage scheme. It need not be set for any of the other three schemes, and in this case can be NULL.
in	*H_col*	is a one-dimensional array of size H_ne and type int, that holds the column indices of the lower triangular part of H in either the sparse co-ordinate, or the sparse row-wise storage scheme. It need not be set when the dense or diagonal storage schemes are used, and in this case can be NULL.
in	*H_ptr*	is a one-dimensional array of size n+1 and type int, that holds the starting position of each row of the lower triangular part of H, as well as the total number of entries plus one, in the sparse row-wise storage scheme. It need not be set when the other schemes are used, and in this case can be NULL.
in	*P_type*	is a one-dimensional array of type char that specifies the symmetric storage scheme used for the residual-Hessians-vector product matrix, P. It should be one of 'coordinate', 'sparse_by_columns', 'dense_by_columns' or 'absent', the latter if access to P is via matrix-vector products; lower or upper case variants are allowed.
in	*P_ne*	is a scalar variable of type int, that holds the number of entries in P in the sparse co-ordinate storage scheme. It need not be set for any of the other schemes.
in	*P_row*	is a one-dimensional array of size P_ne and type int, that holds the row indices of P in either the sparse co-ordinate, or the sparse column-wise storage scheme. It need not be set when the dense storage scheme is used, and in this case can be NULL.
in	*P_col*	is a one-dimensional array of size P_ne and type int, that holds the row indices of P in the sparse co-ordinate storage scheme. It need not be set for any of the other schemes, and in this case can be NULL.
in	*P_ptr*	is a one-dimensional array of size m+1 and type int, that holds the starting position of each row of P, as well as the total number of entries plus one, in the sparse row-wise storage scheme. It need not be set when the other schemes are used, and in this case can be NULL.
in	*w*	is a one-dimensional array of size m and type double, that holds the values w of the weights on the residuals in the least-squares objective function. It need not be set if the weights are all ones, and in this case can be NULL

Examples

nlst.c, and nlstf.c.

5.1.1.2 nls_information()

```
void nls_information (
            void ** data,
            struct nls_inform_type * inform,
            int * status )
```

Provides output information

Parameters

in,out	*data*	holds private internal data
out	*inform*	is a struct containing output information (see nls_inform_type)
out	*status*	is a scalar variable of type int, that gives the exit status from the package. Possible values are (currently): • 0. The values were recorded succesfully

Examples

nlst.c, and nlstf.c.

5.1.1.3 nls_initialize()

```
void nls_initialize (
            void ** data,
            struct nls_control_type * control,
            struct nls_inform_type * inform )
```

Set default control values and initialize private data

Parameters

in,out	*data*	holds private internal data
out	*control*	is a struct containing control information (see nls_control_type)
out	*inform*	is a struct containing output information (see nls_inform_type)

Examples

nlst.c, and nlstf.c.

5.1.1.4 nls_read_specfile()

```
void nls_read_specfile (
            struct nls_control_type * control,
            const char specfile[] )
```

Read the content of a specification file, and assign values associated with given keywords to the corresponding control parameters

Parameters

in,out	*control*	is a struct containing control information (see nls_control_type)
in	*specfile*	is a character string containing the name of the specification file

5.1.1.5 nls_reset_control()

```
void nls_reset_control (
            struct nls_control_type * control,
            void ** data,
            int * status )
```

Reset control parameters after import if required.

Parameters

in	*control*	is a struct whose members provide control paramters for the remaining prcedures (see nls_control_type)
in,out	*data*	holds private internal data
in,out	*status*	is a scalar variable of type int, that gives the exit status from the package. Possible values are: • 1. The import was succesful, and the package is ready for the solve phase

5.1.1.6 nls_solve_reverse_with_mat()

```
void nls_solve_reverse_with_mat (
            void ** data,
            int * status,
            int * eval_status,
            int n,
            int m,
            real_wp_ x[],
            real_wp_ c[],
            real_wp_ g[],
            int j_ne,
            real_wp_ J_val[],
```

```
const real_wp_ y[],
int h_ne,
real_wp_ H_val[],
real_wp_ v[],
int p_ne,
real_wp_ P_val[] )
```

Find a local minimizer of a given function using a trust-region method.

This call is for the case where $H = \nabla_{xx} f(x)$ is provided specifically, but function/derivative information is only available by returning to the calling procedure

Parameters

in,out	*data*	holds private internal data

Parameters

in,out	*status*	is a scalar variable of type int, that gives the entry and exit status from the package. On initial entry, status must be set to 1. Possible exit are: • 0. The run was succesful • -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 n $>$ 0 or requirement that type contains its relevant string 'dense', 'coordinate', 'sparse_by_rows', 'diagonal' or 'absent' 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 CPU time limit has been reached. This may happen if control.cpu_time_limit is too small, but may also be symptomatic of a badly scaled problem. • -82. The user has forced termination of solver by removing the file named control.alive_file from unit unit control.alive_unit. • 2. The user should compute the vector of residuals $c(x)$ at the point x indicated in x and then re-enter the function. The required value should be set in c, and eval_status should be set to 0. If the user is unable to evaluate $c(x)$— for instance, if the function is undefined at x— the user need not set c, but should then set eval_status to a non-zero value. • 3. The user should compute the Jacobian of the vector of residual functions, $\nabla_x c(x)$, at the point x indicated in x and then re-enter the function. The l-th component of the Jacobian stored according to the scheme specified for the remainder of J in the earlier call to nls_import should be set in J_val[l], for l = 0, ..., J_ne-1 and eval_status should be set to 0. If the user is unable to evaluate a component of J — for instance, if a component of the matrix is undefined at x — the user need not set J_val, but should then set eval_status to a non-zero value.

Parameters

	status	(continued) • 4. The user should compute the matrix $H = \sum_{i=1}^{m} v_i \nabla_{xx} c_i(x)$ of weighted residual Hessian evaluated at x= x and v= v and then re-enter the function. The l-th component of the matrix stored according to the scheme specified for the remainder of H in the earlier call to nls_import should be set in H_val[l], for l = 0, ..., H_ne-1 and eval_status should be set to 0. If the user is unable to evaluate a component of H — for instance, if a component of the matrix is undefined at x — the user need not set H_val, but should then set eval_status to a non-zero value. **Note** that this return will not happen if the Gauss-Newton model is selected. • 7. The user should compute the entries of the matrix P, whose i-th column is the product $\nabla_{xx} c_i(x) v$ between $\nabla_{xx} c_i(x)$, the Hessian of the i-th component of the residual $c(x)$ at x= x, and v= v and then re-enter the function. The l-th component of the matrix stored according to the scheme specified for the remainder of P in the earlier call to nls_import should be set in P_val[l], for l = 0, ..., P_ne-1 and eval_status should be set to 0. If the user is unable to evaluate a component of P — for instance, if a component of the matrix is undefined at x — the user need not set P_val, but should then set eval_status to a non-zero value. **Note** that this return will not happen if either the Gauss-Newton or Newton models is selected.
in,out	*eval_status*	is a scalar variable of type int, that is used to indicate if objective function/gradient/Hessian values can be provided (see above)
in	*n*	is a scalar variable of type int, that holds the number of variables
in	*m*	is a scalar variable of type int, that holds the number of residuals.
in,out	*x*	is a one-dimensional array of size n and type double, that holds the values x of the optimization variables. The j-th component of x, j = 0, ... , n-1, contains x_j.
in,out	*c*	is a one-dimensional array of size m and type double, that holds the residual $c(x)$. The i-th component of c, j = 0, ... , n-1, contains $c_j(x)$. See status = 2, above, for more details.
in,out	*g*	is a one-dimensional array of size n and type double, that holds the gradient $g = \nabla_x f(x)$ of the objective function. The j-th component of g, j = 0, ... , n-1, contains g_j.
in	*j_ne*	is a scalar variable of type int, that holds the number of entries in the Jacobian matrix J.
in	*J_val*	is a one-dimensional array of size ne and type double, that holds the values of the entries of the Jacobian matrix J in any of the available storage schemes. See status = 3, above, for more details.
in,out	*y*	is a one-dimensional array of size m and type double, that is used for reverse communication. See status = 4 above for more details.
in	*h_ne*	is a scalar variable of type int, that holds the number of entries in the lower triangular part of the Hessian matrix H.
in	*H_val*	is a one-dimensional array of size ne and type double, that holds the values of the entries of the lower triangular part of the Hessian matrix H in any of the available storage schemes. See status = 4, above, for more details.
in,out	*v*	is a one-dimensional array of size n and type double, that is used for reverse communication. See status = 7, above, for more details.
in	*p_ne*	is a scalar variable of type int, that holds the number of entries in the residual-Hessians-vector product matrix, P.
in	*P_val*	is a one-dimensional array of size ne and type double, that holds the values of the entries of the residual-Hessians-vector product matrix, P. See status = 7, above, for more details.

Examples

nlst.c, and nlstf.c.

5.1.1.7 nls_solve_reverse_without_mat()

```
void nls_solve_reverse_without_mat (
            void ** data,
            int * status,
            int * eval_status,
            int n,
            int m,
            real_wp_ x[],
            real_wp_ c[],
            real_wp_ g[],
            bool * transpose,
            real_wp_ u[],
            real_wp_ v[],
            real_wp_ y[],
            int p_ne,
            real_wp_ P_val[] )
```

Find a local minimizer of a given function using a trust-region method.

This call is for the case where access to $H = \nabla_{xx} f(x)$ is provided by Hessian-vector products, but function/derivative information is only available by returning to the calling procedure.

Parameters

in,out	*data*	holds private internal data
in,out	*status*	is a scalar variable of type int, that gives the entry and exit status from the package. On initial entry, status must be set to 1. Possible exit are: • 0. The run was succesful • -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 n $>$ 0 or requirement that type contains its relevant string 'dense', 'coordinate', 'sparse_by_rows', 'diagonal' or 'absent' 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

Parameters

	status	(continued) • -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 CPU time limit has been reached. This may happen if control.cpu_time_limit is too small, but may also be symptomatic of a badly scaled problem. • -82. The user has forced termination of solver by removing the file named control.alive_file from unit unit control.alive_unit. • 2. The user should compute the vector of residuals $c(x)$ at the point x indicated in x and then re-enter the function. The required value should be set in c, and eval_status should be set to 0. If the user is unable to evaluate $c(x)$— for instance, if the function is undefined at x— the user need not set c, but should then set eval_status to a non-zero value. • 5. The user should compute the sum $u + \nabla_x c_(x)v$ (if tranpose is false) or $u + (\nabla_x c_(x))^T v$ (if tranpose is true) between the product of the Jacobian $\nabla_x c_(x)$ or its tranpose with the vector v= v and the vector u = $ u, and then re-enter the function. The result should be set in u, and eval_status should be set to 0. If the user is unable to evaluate the sum — for instance, if the Jacobian is undefined at x — the user need not set u, but should then set eval_status to a non-zero value. • 6. The user should compute the sum $u + \sum_{i=1}^{m} y_i \nabla_{xx} c_i(x)v$ between the product of the weighted residual Hessian $H = \sum_{i=1}^{m} y_i \nabla_{xx} c_i(x)$ evaluated at x= x and y= y with the vector v= v and the the vector u = $ u, and then re-enter the function. The result should be set in u, and eval_status should be set to 0. If the user is unable to evaluate the sum — for instance, if the weifghted residual Hessian is undefined at x — the user need not set u, but should then set eval_status to a non-zero value. • 7. The user should compute the entries of the matrix P, whose i-th column is the product $\nabla_{xx} c_i(x)v$ between $\nabla_{xx} c_i(x)$, the Hessian of the i-th component of the residual $c(x)$ at x= x, and v= v and then re-enter the function. The l-th component of the matrix stored according to the scheme specified for the remainder of P in the earlier call to nls_import should be set in P_val[l], for l = 0, ..., P_ne-1 and eval_status should be set to 0. If the user is unable to evaluate a component of P — for instance, if a component of the matrix is undefined at x — the user need not set P_val, but should then set eval_status to a non-zero value. **Note** that this return will not happen if either the Gauss-Newton or Newton models is selected.
`in,out`	*eval_status*	is a scalar variable of type int, that is used to indicate if objective function/gradient/Hessian values can be provided (see above)

Parameters

in	*n*	is a scalar variable of type int, that holds the number of variables
in	*m*	is a scalar variable of type int, that holds the number of residuals.
in,out	*x*	is a one-dimensional array of size n and type double, that holds the values x of the optimization variables. The j-th component of x, j = 0, ... , n-1, contains x_j.
in,out	*c*	is a one-dimensional array of size m and type double, that holds the residual $c(x)$. The i-th component of c, j = 0, ... , n-1, contains $c_j(x)$. See status = 2, above, for more details.
in,out	*g*	is a one-dimensional array of size n and type double, that holds the gradient $g = \nabla_x f(x)$ of the objective function. The j-th component of g, j = 0, ... , n-1, contains g_j.
out	*transpose*	is a scalar variable of type bool, that indicates whether the product with Jacobian or its transpose should be obtained when status=5.
in,out	*u*	is a one-dimensional array of size max(n,m) and type double, that is used for reverse communication. See status = 5,6 above for more details.
in,out	*v*	is a one-dimensional array of size max(n,m) and type double, that is used for reverse communication. See status = 5,6,7 above for more details.
in,out	*y*	is a one-dimensional array of size m and type double, that is used for reverse communication. See status = 6 above for more details.
in	*p_ne*	is a scalar variable of type int, that holds the number of entries in the residual-Hessians-vector product matrix, P.
in	*P_val*	is a one-dimensional array of size P_ne and type double, that holds the values of the entries of the residual-Hessians-vector product matrix, P. See status = 7, above, for more details.

Examples

nlst.c, and nlstf.c.

5.1.1.8 nls_solve_with_mat()

```
void nls_solve_with_mat (
             void ** data,
             void * userdata,
             int * status,
             int n,
             int m,
             real_wp_ x[],
             real_wp_ c[],
             real_wp_ g[],
             int(*)(int, int, const real_wp_[], real_wp_[], const void *) eval_c,
             int j_ne,
             int(*)(int, int, int, const real_wp_[], real_wp_[], const void *) eval_j,
             int h_ne,
             int(*)(int, int, int, const real_wp_[], const real_wp_[], real_wp_[], const void
*) eval_h,
             int p_ne,
             int(*)(int, int, int, const real_wp_[], const real_wp_[], real_wp_[], bool, const
void *) eval_hprods )
```

Find a local minimizer of a given function using a trust-region method.

This call is for the case where $H = \nabla_{xx} f(x)$ is provided specifically, and all function/derivative information is available by function calls.

Parameters

in,out	*data*	holds private internal data
in	*userdata*	is a structure that allows data to be passed into the function and derivative evaluation programs.
in,out	*status*	is a scalar variable of type int, that gives the entry and exit status from the package. On initial entry, status must be set to 1. Possible exit are: • 0. The run was succesful • -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 n $>$ 0 or requirement that type contains its relevant string 'dense', 'coordinate', 'sparse_by_rows', 'diagonal' or 'absent' 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 CPU time limit has been reached. This may happen if control.cpu_time_limit is too small, but may also be symptomatic of a badly scaled problem. • -82. The user has forced termination of solver by removing the file named control.alive_file from unit unit control.alive_unit.
in	*n*	is a scalar variable of type int, that holds the number of variables.
in	*m*	is a scalar variable of type int, that holds the number of residuals.
in,out	*x*	is a one-dimensional array of size n and type double, that holds the values x of the optimization variables. The j-th component of x, j = 0, ... , n-1, contains x_j.
out	*c*	is a one-dimensional array of size m and type double, that holds the residual $c(x)$. The i-th component of c, j = 0, ... , n-1, contains $c_j(x)$.
out	*g*	is a one-dimensional array of size n and type double, that holds the gradient $g = \nabla_x f(x)$ of the objective function. The j-th component of g, j = 0, ... , n-1, contains g_j.

Parameters

	eval_c	is a user-supplied function that must have the following signature: `int eval_c( int n, const double x[], double c[], const void *userdata )` The componnts of the residual function $c(x)$ evaluated at x= x must be assigned to c, and the function return value set to 0. If the evaluation is impossible at x, return should be set to a nonzero value. Data may be passed into `eval_c` via the structure `userdata`.
in	*j_ne*	is a scalar variable of type int, that holds the number of entries in the Jacobian matrix J.
	eval_j	is a user-supplied function that must have the following signature: `int eval_j( int n, int m, int jne, const double x[], double j[], const void *userdata )` The components of the Jacobian $J = \nabla_x c(x)$ of the residuals must be assigned to j in the same order as presented to nls_import, and the function return value set to 0. If the evaluation is impossible at x, return should be set to a nonzero value. Data may be passed into `eval_j` via the structure `userdata`.
in	*h_ne*	is a scalar variable of type int, that holds the number of entries in the lower triangular part of the Hessian matrix H if it is used.
	eval_h	is a user-supplied function that must have the following signature: `int eval_h( int n, int m, int hne, const double x[], const double y[], double h[], const void *userdata )` The nonzeros of the matrix $H = \sum_{i=1}^{m} y_i \nabla_{xx} c_i(x)$ of the weighted residual Hessian evaluated at x= x and y= y must be assigned to h in the same order as presented to nls_import, and the function return value set to 0. If the evaluation is impossible at x, return should be set to a nonzero value. Data may be passed into `eval_h` via the structure `userdata`.
in	*p_ne*	is a scalar variable of type int, that holds the number of entries in the residual-Hessians-vector product matrix P if it is used.
	eval_hprods	is an optional user-supplied function that may be NULL. If non-NULL, it must have the following signature: `int eval_hprods( int n, int m, int pne, const double x[], const double v[], double p[], bool got_h, const void *userdata ) );` The entries of the matrix P, whose i-th column is the product $\nabla_{xx} c_i(x) v$ between $\nabla_{xx} c_i(x)$, the Hessian of the i-th component of the residual $c(x)$ at x= x, and v= v must be returned in p and the function return value set to 0. If the evaluation is impossible at x, return should be set to a nonzero value. Data may be passed into `eval_hprods` via the structure `userdata`.

Examples

nlst.c, and nlstf.c.

5.1.1.9 nls_solve_without_mat()

```
void nls_solve_without_mat (
            void ** data,
            void * userdata,
            int * status,
            int n,
            int m,
            real_wp_ x[],
            real_wp_ c[],
```

```
            real_wp_ g[],
            int(*)(int, int, const real_wp_[], real_wp_[], const void *) eval_c,
            int(*)(int, int, const real_wp_[], const bool, real_wp_[], const real_wp_[],
bool, const void *) eval_jprod,
            int(*)(int, int, const real_wp_[], const real_wp_[], real_wp_[], const real_wp_↩
[], bool, const void *) eval_hprod,
            int p_ne,
            int(*)(int, int, int, const real_wp_[], const real_wp_[], real_wp_[], bool, const
void *) eval_hprods )
```

Find a local minimizer of a given function using a trust-region method.

This call is for the case where access to $H = \nabla_{xx} f(x)$ is provided by Hessian-vector products, and all function/derivative information is available by function calls.

Parameters

in,out	*data*	holds private internal data
in	*userdata*	is a structure that allows data to be passed into the function and derivative evaluation programs.

Parameters

in,out	*status*	is a scalar variable of type int, that gives the entry and exit status from the package. On initial entry, status must be set to 1. Possible exit are: • 0. The run was succesful • -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 n $>$ 0 or requirement that type contains its relevant string 'dense', 'coordinate', 'sparse_by_rows', 'diagonal' or 'absent' 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 CPU time limit has been reached. This may happen if control.cpu_time_limit is too small, but may also be symptomatic of a badly scaled problem. • -82. The user has forced termination of solver by removing the file named control.alive_file from unit unit control.alive_unit.
in	*n*	is a scalar variable of type int, that holds the number of variables
in	*m*	is a scalar variable of type int, that holds the number of residuals.
in,out	*x*	is a one-dimensional array of size n and type double, that holds the values x of the optimization variables. The j-th component of x, j = 0, ... , n-1, contains x_j.
out	*c*	is a one-dimensional array of size m and type double, that holds the residual $c(x)$. The i-th component of c, j = 0, ... , n-1, contains $c_j(x)$.
out	*g*	is a one-dimensional array of size n and type double, that holds the gradient $g = \nabla_x f(x)$ of the objective function. The j-th component of g, j = 0, ... , n-1, contains g_j.

Parameters

	eval_c	is a user-supplied function that must have the following signature: `int eval_c( int n, const double x[], double c[], const void *userdata )` The componnts of the residual function $c(x)$ evaluated at x= x must be assigned to c, and the function return value set to 0. If the evaluation is impossible at x, return should be set to a nonzero value. Data may be passed into `eval_c` via the structure `userdata`.
	eval_jprod	is a user-supplied function that must have the following signature: `int eval_jprod( int n, int m, const double x[], bool transpose, double u[], const double v[], bool got_j, const void *userdata )` The sum $u + \nabla_x c_(x)v$ (if tranpose is false) or The sum $u + (\nabla_x c_(x))^T v$ (if tranpose is true) bewteen the product of the Jacobian $\nabla_x c_(x)$ or its tranpose with the vector v= v and the vector $ u must be returned in u, and the function return value set to 0. If the evaluation is impossible at x, return should be set to a nonzero value. Data may be passed into `eval_jprod` via the structure `userdata`.
	eval_hprod	is a user-supplied function that must have the following signature: `int eval_hprod( int n, int m, const double x[], const double y[], double u[], const double v[], bool got_h, const void *userdata )` The sum $u + \sum_{i=1}^{m} y_i \nabla_{xx} c_i(x)v$ of the product of the weighted residual Hessian $H = \sum_{i=1}^{m} y_i \nabla_{xx} c_i(x)$ evaluated at x= x and y= y with the vector v= v and the vector $ u must be returned in u, and the function return value set to 0. If the evaluation is impossible at x, return should be set to a nonzero value. The Hessians have already been evaluated or used at x if got_h is true. Data may be passed into `eval_hprod` via the structure `userdata`.
`in`	*p_ne*	is a scalar variable of type int, that holds the number of entries in the residual-Hessians-vector product matrix P if it is used.
	eval_hprods	is an optional user-supplied function that may be NULL. If non-NULL, it must have the following signature: `int eval_hprods( int n, int m, int p_ne, const double x[], const double v[], double pval[], bool got_h, const void *userdata )` The entries of the matrix P, whose i-th column is the product $\nabla_{xx} c_i(x)v$ between $\nabla_{xx} c_i(x)$, the Hessian of the i-th component of the residual $c(x)$ at x= x, and v= v must be returned in pval and the function return value set to 0. If the evaluation is impossible at x, return should be set to a nonzero value. Data may be passed into `eval_hprods` via the structure `userdata`.

Examples

nlst.c, and nlstf.c.

5.1.1.10 nls_terminate()

```
void nls_terminate (
            void ** data,
            struct nls_control_type * control,
            struct nls_inform_type * inform )
```

Deallocate all internal private storage

Parameters

`in,out`	*data*	holds private internal data
`out`	*control*	is a struct containing control information (see nls_control_type)
`out`	*inform*	is a struct containing output information (see nls_inform_type)

Examples

nlst.c, and nlstf.c.

Chapter 6

Example Documentation

6.1 nlst.c

This is an example of how to use the package both when the matrices (Jacobian, Hessian, residual-Hessians-vector product) are directly available and when their product with vectors may be found. Both function call evaluations and returns to the calling program to find the required values are illustrated. A variety of supported Hessian storage formats are shown.

Notice that C-style indexing is used, and that this is flaggeed by setting `control.f_indexing` to `false`. In addition, see how parameters may be passed into the evaluation functions via `userdata`.

```
/* nlst.c */
/* Full test for the NLS C interface using C sparse matrix indexing */
/* Jari Fowkes & Nick Gould, STFC-Rutherford Appleton Laboratory, 2021 */
#include <stdio.h>
#include <math.h>
#include "nls.h"
#define max(a,b)  \
({  __typeof__ (a) _a = (a); \
    __typeof__ (b) _b = (b); \
    _a > _b ? _a : _b; })
// Custom userdata struct
struct userdata_type {
   double p;
};
// Function prototypes
int res( int n, int m, const double x[], double c[], const void * );
int jac( int n, int m, int jne, const double x[], double jval[], const void * );
int hess( int n, int m, int hne, const double x[], const double y[],
          double hval[], const void * );
int jacprod( int n, int m, const double x[], const bool transpose, double u[],
             const double v[], bool got_j, const void * );
int hessprod( int n, int m, const double x[], const double y[], double u[],
              const double v[], bool got_h, const void * );
int rhessprods( int n, int m, int pne, const double x[], const double v[],
                double pval[], bool got_h, const void * );
int scale( int n, int m, const double x[], double u[],
           const double v[], const void * );
int jac_dense( int n, int m, int jne, const double x[], double jval[],
               const void * );
int hess_dense( int n, int m, int hne, const double x[], const double y[],
                double hval[], const void * );
int rhessprods_dense( int n, int m, int pne, const double x[],
                      const double v[], double pval[], bool got_h,
                      const void * );
int main(void) {
    // Derived types
    void *data;
    struct nls_control_type control;
    struct nls_inform_type inform;
    // Set user data
    struct userdata_type userdata;
```

```
    userdata.p = 1.0;
    // Set problem data
    int n = 2; // # variables
    int m = 3; // # residuals
    int j_ne = 5; // Jacobian elements
    int h_ne = 2; // Hesssian elements
    int p_ne = 2; // residual-Hessians-vector products elements
    int J_row[] = {0, 1, 1, 2, 2}; // Jacobian J
    int J_col[] = {0, 0, 1, 0, 1}; //
    int J_ptr[] = {0, 1, 3, 5};    // row pointers
    int H_row[] = {0, 1};          // Hessian H
    int H_col[] = {0, 1};          // NB lower triangle
    int H_ptr[] = {0, 1, 2};       // row pointers
    int P_row[] = {0, 1};          // residual-Hessians-vector product matrix
    int P_ptr[] = {0, 1, 2, 2};    // column pointers
    // Set storage
    double g[n]; // gradient
    double c[m]; // residual
    double y[m]; // multipliers
    double W[] = {1.0, 1.0, 1.0}; // weights
    char st;
    int status;
    printf(" C sparse matrix indexing\n\n");
    printf(" tests options for all-in-one storage format\n\n");
    for( int d=1; d <= 5; d++){
//  for( int d=5; d <= 5; d++){
        // Initialize NLS
        nls_initialize( &data, &control, &inform );
        // Set user-defined control options
        control.f_indexing = false; // C sparse matrix indexing
        //control.print_level = 1;
        control.jacobian_available = 2;
        control.hessian_available = 2;
        control.model = 6;
        double x[] = {1.5,1.5}; // starting point
        double W[] = {1.0, 1.0, 1.0}; // weights
        switch(d){
            case 1: // sparse co-ordinate storage
                st = 'C';
                nls_import( &control, &data, &status, n, m,
                            "coordinate", j_ne, J_row, J_col, NULL,
                            "coordinate", h_ne, H_row, H_col, NULL,
                            "sparse_by_columns", p_ne, P_row, NULL, P_ptr, W );
                nls_solve_with_mat( &data, &userdata, &status,
                                    n, m, x, c, g, res, j_ne, jac,
                                    h_ne, hess, p_ne, rhessprods );
                break;
            case 2: // sparse by rows
                st = 'R';
                nls_import( &control, &data, &status, n, m,
                            "sparse_by_rows", j_ne, NULL, J_col, J_ptr,
                            "sparse_by_rows", h_ne, NULL, H_col, H_ptr,
                            "sparse_by_columns", p_ne, P_row, NULL, P_ptr, W );
                nls_solve_with_mat( &data, &userdata, &status,
                                    n, m, x, c, g, res, j_ne, jac,
                                    h_ne, hess, p_ne, rhessprods );
                break;
            case 3: // dense
                st = 'D';
                nls_import( &control, &data, &status, n, m,
                            "dense", j_ne, NULL, NULL, NULL,
                            "dense", h_ne, NULL, NULL, NULL,
                            "dense", p_ne, NULL, NULL, NULL, W );
                nls_solve_with_mat( &data, &userdata, &status,
                                    n, m, x, c, g, res, j_ne, jac_dense,
                                    h_ne, hess_dense, p_ne, rhessprods_dense );
                break;
            case 4: // diagonal
                st = 'I';
                nls_import( &control, &data, &status, n, m,
                            "sparse_by_rows", j_ne, NULL, J_col, J_ptr,
                            "diagonal", h_ne, NULL, NULL, NULL,
                            "sparse_by_columns", p_ne, P_row, NULL, P_ptr, W );
                nls_solve_with_mat( &data, &userdata, &status,
                                    n, m, x, c, g, res, j_ne, jac,
                                    h_ne, hess, p_ne, rhessprods );
                break;
            case 5: // access by products
                st = 'P';
                nls_import( &control, &data, &status, n, m,
                            "absent", j_ne, NULL, NULL, NULL,
                            "absent", h_ne, NULL, NULL, NULL,
                            "sparse_by_columns", p_ne, P_row, NULL, P_ptr, W );
                nls_solve_without_mat( &data, &userdata, &status,
                                       n, m, x, c, g, res, jacprod,
                                       hessprod, p_ne, rhessprods );
                break;
```

```
        }
        nls_information( &data, &inform, &status );
        if(inform.status == 0){
            printf("%c:%6i iterations. Optimal objective value = %5.2f"
                   " status = %1i\n",
                   st, inform.iter, inform.obj, inform.status);
        }else{
            printf("%c: NLS_solve exit status = %1i\n", st, inform.status);
        }
        // Delete internal workspace
        nls_terminate( &data, &control, &inform );
    }
    printf("\n tests reverse-communication options\n\n");
    // reverse-communication input/output
    int eval_status;
    double u[max(m,n)], v[max(m,n)];
    double J_val[j_ne], J_dense[m*n];
    double H_val[h_ne], H_dense[n*(n+1)/2], H_diag[n];
    double P_val[p_ne], P_dense[m*n];
    bool transpose;
    bool got_j = false;
    bool got_h = false;
    for( int d=1; d <= 5; d++){
//  for( int d=1; d <= 4; d++){
        // Initialize NLS
        nls_initialize( &data, &control, &inform );
        // Set user-defined control options
        control.f_indexing = false; // C sparse matrix indexing
        //control.print_level = 1;
        control.jacobian_available = 2;
        control.hessian_available = 2;
        control.model = 6;
        double x[] = {1.5,1.5}; // starting point
        double W[] = {1.0, 1.0, 1.0}; // weights
        switch(d){
            case 1: // sparse co-ordinate storage
                st = 'C';
                nls_import( &control, &data, &status, n, m,
                            "coordinate", j_ne, J_row, J_col, NULL,
                            "coordinate", h_ne, H_row, H_col, NULL,
                            "sparse_by_columns", p_ne, P_row, NULL, P_ptr, W );
                while(true){ // reverse-communication loop
                  nls_solve_reverse_with_mat( &data, &status, &eval_status,
                                              n, m, x, c, g, j_ne, J_val, y,
                                              h_ne, H_val, v, p_ne, P_val );
                  if(status == 0){ // successful termination
                        break;
                  }else if(status < 0){ // error exit
                      break;
                  }else if(status == 2){ // evaluate c
                      eval_status = res( n, m, x, c, &userdata );
                  }else if(status == 3){ // evaluate J
                      eval_status = jac( n, m, j_ne, x, J_val, &userdata );
                  }else if(status == 4){ // evaluate H
                      eval_status = hess( n, m, h_ne, x, y, H_val, &userdata );
                  }else if(status == 7){ // evaluate P
                      eval_status = rhessprods( n, m, p_ne, x, v, P_val,
                                                got_h, &userdata );
                  }else{
                      printf(" the value %1i of status should not occur\n",
                        status);
                      break;
                  }
                }
                break;
            case 2: // sparse by rows
                st = 'R';
                nls_import( &control, &data, &status, n, m,
                            "sparse_by_rows", j_ne, NULL, J_col, J_ptr,
                            "sparse_by_rows", h_ne, NULL, H_col, H_ptr,
                            "sparse_by_columns", p_ne, P_row, NULL, P_ptr, W );
                while(true){ // reverse-communication loop
                  nls_solve_reverse_with_mat( &data, &status, &eval_status,
                                              n, m, x, c, g, j_ne, J_val, y,
                                              h_ne, H_val, v, p_ne, P_val );
                  if(status == 0){ // successful termination
                        break;
                  }else if(status < 0){ // error exit
                      break;
                  }else if(status == 2){ // evaluate c
                      eval_status = res( n, m, x, c, &userdata );
                  }else if(status == 3){ // evaluate J
                      eval_status = jac( n, m, j_ne, x, J_val, &userdata );
                  }else if(status == 4){ // evaluate H
                      eval_status = hess( n, m, h_ne, x, y, H_val, &userdata );
                  }else if(status == 7){ // evaluate P
                      eval_status = rhessprods( n, m, p_ne, x, v, P_val,
```

```
                                        got_h, &userdata );
            }else{
                printf(" the value %1i of status should not occur\n",
                  status);
                break;
            }
          }
          break;
        case 3: // dense
          st = 'D';
          nls_import( &control, &data, &status, n, m,
                      "dense", j_ne, NULL, NULL, NULL,
                      "dense", h_ne, NULL, NULL, NULL,
                      "dense", p_ne, NULL, NULL, NULL, W );
          while(true){ // reverse-communication loop
            nls_solve_reverse_with_mat( &data, &status, &eval_status,
                                        n, m, x, c, g, j_ne, J_dense, y,
                                        h_ne, H_dense, v, p_ne, P_dense );
            if(status == 0){ // successful termination
                  break;
            }else if(status < 0){ // error exit
                break;
            }else if(status == 2){ // evaluate c
                eval_status = res( n, m, x, c, &userdata );
            }else if(status == 3){ // evaluate J
                eval_status = jac_dense( n, m, j_ne, x, J_dense,
                                         &userdata );
            }else if(status == 4){ // evaluate H
                eval_status = hess_dense( n, m, h_ne, x, y, H_dense,
                                          &userdata );
            }else if(status == 7){ // evaluate P
                eval_status = rhessprods_dense( n, m, p_ne, x, v, P_dense,
                                                got_h, &userdata );
            }else{
                printf(" the value %1i of status should not occur\n",
                  status);
                break;
            }
          }
          break;
        case 4: // diagonal
          st = 'I';
          nls_import( &control, &data, &status, n, m,
                      "sparse_by_rows", j_ne, NULL, J_col, J_ptr,
                      "diagonal", h_ne, NULL, NULL, NULL,
                      "sparse_by_columns", p_ne, P_row, NULL, P_ptr, W );
          while(true){ // reverse-communication loop
            nls_solve_reverse_with_mat( &data, &status, &eval_status,
                                        n, m, x, c, g, j_ne, J_val, y,
                                        h_ne, H_diag, v, p_ne, P_val );
            if(status == 0){ // successful termination
                  break;
            }else if(status < 0){ // error exit
                break;
            }else if(status == 2){ // evaluate c
                eval_status = res( n, m, x, c, &userdata );
            }else if(status == 3){ // evaluate J
                eval_status = jac( n, m, j_ne, x, J_val, &userdata );
            }else if(status == 4){ // evaluate H
                eval_status = hess( n, m, h_ne, x, y, H_diag, &userdata );
            }else if(status == 7){ // evaluate P
                eval_status = rhessprods( n, m, p_ne, x, v, P_val,
                                          got_h, &userdata );
            }else{
                printf(" the value %1i of status should not occur\n",
                  status);
                break;
            }
          }
          break;
        case 5: // access by products
          st = 'P';
//        control.print_level = 1;
          nls_import( &control, &data, &status, n, m,
                      "absent", j_ne, NULL, NULL, NULL,
                      "absent", h_ne, NULL, NULL, NULL,
                      "sparse_by_columns", p_ne, P_row, NULL, P_ptr, W );
          while(true){ // reverse-communication loop
            nls_solve_reverse_without_mat( &data, &status, &eval_status,
                                           n, m, x, c, g, &transpose,
                                           u, v, y, p_ne, P_val );
            if(status == 0){ // successful termination
                  break;
            }else if(status < 0){ // error exit
                break;
            }else if(status == 2){ // evaluate c
                eval_status = res( n, m, x, c, &userdata );
```

```
            }else if(status == 5){ // evaluate u + J v or u + J'v
                eval_status = jacprod( n, m, x, transpose, u, v, got_j,
                                       &userdata );
            }else if(status == 6){ // evaluate u + H v
                eval_status = hessprod( n, m, x, y, u, v, got_h,
                                        &userdata );
            }else if(status == 7){ // evaluate P
                eval_status = rhessprods( n, m, p_ne, x, v, P_val,
                                          got_h, &userdata );
            }else{
                printf(" the value %1i of status should not occur\n",
                  status);
                break;
            }
          }
          break;
    }
    nls_information( &data, &inform, &status );
    if(inform.status == 0){
        printf("%c:%6i iterations. Optimal objective value = %5.2f"
               " status = %1i\n",
               st, inform.iter, inform.obj, inform.status);
    }else{
        printf("%c: NLS_solve exit status = %1i\n", st, inform.status);
    }
    // Delete internal workspace
    nls_terminate( &data, &control, &inform );
}
printf("\n basic tests of models used, direct access\n\n");
for( int model=3; model <= 8; model++){
    // Initialize NLS
    nls_initialize( &data, &control, &inform );
    // Set user-defined control options
    control.f_indexing = false; // C sparse matrix indexing
    //control.print_level = 1;
    control.jacobian_available = 2;
    control.hessian_available = 2;
    control.model = model;
    double x[] = {1.5,1.5}; // starting point
    double W[] = {1.0, 1.0, 1.0}; // weights
    nls_import( &control, &data, &status, n, m,
                "sparse_by_rows", j_ne, NULL, J_col, J_ptr,
                "sparse_by_rows", h_ne, NULL, H_col, H_ptr,
                "sparse_by_columns", p_ne, P_row, NULL, P_ptr, W );
    nls_solve_with_mat( &data, &userdata, &status,
                        n, m, x, c, g, res, j_ne, jac,
                        h_ne, hess, p_ne, rhessprods );
    nls_information( &data, &inform, &status );
    if(inform.status == 0){
        printf(" %1i:%6i iterations. Optimal objective value = %5.2f"
               " status = %1i\n",
               model, inform.iter, inform.obj, inform.status);
    }else{
        printf(" %i: NLS_solve exit status = %1i\n", model, inform.status);
    }
    // Delete internal workspace
    nls_terminate( &data, &control, &inform );
}
printf("\n basic tests of models used, access by products\n\n");
for( int model=3; model <= 8; model++){
    // Initialize NLS
    nls_initialize( &data, &control, &inform );
    // Set user-defined control options
    control.f_indexing = false; // C sparse matrix indexing
    //control.print_level = 1;
    control.jacobian_available = 2;
    control.hessian_available = 2;
    control.model = model;
    double x[] = {1.5,1.5}; // starting point
    double W[] = {1.0, 1.0, 1.0}; // weights
    nls_import( &control, &data, &status, n, m,
                "absent", j_ne, NULL, NULL, NULL,
                "absent", h_ne, NULL, NULL, NULL,
                "sparse_by_columns", p_ne, P_row, NULL, P_ptr, W );
    nls_solve_without_mat( &data, &userdata, &status,
                           n, m, x, c, g, res, jacprod,
                           hessprod, p_ne, rhessprods );
    nls_information( &data, &inform, &status );
    if(inform.status == 0){
        printf("P%1i:%6i iterations. Optimal objective value = %5.2f"
               " status = %1i\n",
               model, inform.iter, inform.obj, inform.status);
    }else{
        printf("P%i: NLS_solve exit status = %1i\n", model, inform.status);
    }
    // Delete internal workspace
    nls_terminate( &data, &control, &inform );
```

```
}
printf("\n basic tests of models used, reverse access\n\n");
for( int model=3; model <= 8; model++){
    // Initialize NLS
    nls_initialize( &data, &control, &inform );
    // Set user-defined control options
    control.f_indexing = false; // C sparse matrix indexing
    //control.print_level = 1;
    control.jacobian_available = 2;
    control.hessian_available = 2;
    control.model = model;
    double x[] = {1.5,1.5}; // starting point
    double W[] = {1.0, 1.0, 1.0}; // weights
    nls_import( &control, &data, &status, n, m,
                "sparse_by_rows", j_ne, NULL, J_col, J_ptr,
                "sparse_by_rows", h_ne, NULL, H_col, H_ptr,
                "sparse_by_columns", p_ne, P_row, NULL, P_ptr, W );
    while(true){ // reverse-communication loop
      nls_solve_reverse_with_mat( &data, &status, &eval_status,
                                  n, m, x, c, g, j_ne, J_val, y,
                                  h_ne, H_val, v, p_ne, P_val );
      if(status == 0){ // successful termination
            break;
      }else if(status < 0){ // error exit
          break;
      }else if(status == 2){ // evaluate c
          eval_status = res( n, m, x, c, &userdata );
      }else if(status == 3){ // evaluate J
          eval_status = jac( n, m, j_ne, x, J_val, &userdata );
      }else if(status == 4){ // evaluate H
          eval_status = hess( n, m, h_ne, x, y, H_val, &userdata );
      }else if(status == 7){ // evaluate P
          eval_status = rhessprods( n, m, p_ne, x, v, P_val,
                                    got_h, &userdata );
      }else{
          printf(" the value %1i of status should not occur\n",
            status);
          break;
      }
    }
    nls_information( &data, &inform, &status );
    if(inform.status == 0){
        printf("P%1i:%6i iterations. Optimal objective value = %5.2f"
               " status = %1i\n",
               model, inform.iter, inform.obj, inform.status);
    }else{
        printf(" %i: NLS_solve exit status = %1i\n", model, inform.status);
    }
    // Delete internal workspace
    nls_terminate( &data, &control, &inform );
}
printf("\n basic tests of models used, reverse access by products\n\n");
for( int model=3; model <= 8; model++){
    // Initialize NLS
    nls_initialize( &data, &control, &inform );
    // Set user-defined control options
    control.f_indexing = false; // C sparse matrix indexing
    //control.print_level = 1;
    control.jacobian_available = 2;
    control.hessian_available = 2;
    control.model = model;
    double x[] = {1.5,1.5}; // starting point
    double W[] = {1.0, 1.0, 1.0}; // weights
    nls_import( &control, &data, &status, n, m,
                "absent", j_ne, NULL, NULL, NULL,
                "absent", h_ne, NULL, NULL, NULL,
                "sparse_by_columns", p_ne, P_row, NULL, P_ptr, W );
    while(true){ // reverse-communication loop
      nls_solve_reverse_without_mat( &data, &status, &eval_status,
                                     n, m, x, c, g, &transpose,
                                     u, v, y, p_ne, P_val );
      if(status == 0){ // successful termination
            break;
      }else if(status < 0){ // error exit
          break;
      }else if(status == 2){ // evaluate c
          eval_status = res( n, m, x, c, &userdata );
      }else if(status == 5){ // evaluate u + J v or u + J'v
          eval_status = jacprod( n, m, x, transpose, u, v, got_j,
                                 &userdata );
      }else if(status == 6){ // evaluate u + H v
          eval_status = hessprod( n, m, x, y, u, v, got_h,
                                  &userdata );
      }else if(status == 7){ // evaluate P
          eval_status = rhessprods( n, m, p_ne, x, v, P_val,
                                    got_h, &userdata );
      }else{
```

```
              printf(" the value %1i of status should not occur\n",
                status);
              break;
          }
        }
        nls_information( &data, &inform, &status );
        if(inform.status == 0){
            printf("P%1i:%6i iterations. Optimal objective value = %5.2f"
                   " status = %1i\n",
                   model, inform.iter, inform.obj, inform.status);
        }else{
            printf("P%i: NLS_solve exit status = %1i\n", model, inform.status);
        }
        // Delete internal workspace
        nls_terminate( &data, &control, &inform );
    }
}
// compute the residuals
int res( int n, int m, const double x[], double c[], const void *userdata ){
    struct userdata_type *myuserdata = ( struct userdata_type * ) userdata;
    double p = myuserdata->p;
    c[0] = pow(x[0],2.0) + p;
    c[1] = x[0] + pow(x[1],2.0);
    c[2] = x[0] - x[1];
    return 0;
}
// compute the Jacobian
int jac( int n, int m, int jne, const double x[], double jval[],
         const void *userdata ){
    struct userdata_type *myuserdata = ( struct userdata_type * ) userdata;
    jval[0] = 2.0 * x[0];
    jval[1] = 1.0;
    jval[2] = 2.0 * x[1];
    jval[3] = 1.0;
    jval[4] = - 1.0;
    return 0;
}
// compute the Hessian
int hess( int n, int m, int hne, const double x[], const double y[],
          double hval[], const void *userdata ){
    struct userdata_type *myuserdata = ( struct userdata_type * ) userdata;
    hval[0] = 2.0 * y[0];
    hval[1] = 2.0 * y[1];
    return 0;
}
// compute Jacobian-vector products
int jacprod( int n, int m, const double x[], const bool transpose, double u[],
             const double v[], bool got_j, const void *userdata ){
    struct userdata_type *myuserdata = ( struct userdata_type * ) userdata;
    if (transpose) {
      u[0] = u[0] + 2.0 * x[0] * v[0] + v[1] + v[2];
      u[1] = u[1] + 2.0 * x[1] * v[1] - v[2];
    }else{
      u[0] = u[0] + 2.0 * x[0] * v[0];
      u[1] = u[1] + v[0]  + 2.0 * x[1] * v[1];
      u[2] = u[2] + v[0] - v[1];
    }
    return 0;
}
// compute Hessian-vector products
int hessprod( int n, int m, const double x[], const double y[], double u[],
              const double v[], bool got_h, const void *userdata ){
    struct userdata_type *myuserdata = ( struct userdata_type * ) userdata;
    u[0] = u[0] + 2.0 * y[0] * v[0];
    u[1] = u[1] + 2.0 * y[1] * v[1];
    return 0;
}
// compute residual-Hessians-vector products
int rhessprods( int n, int m, int pne, const double x[], const double v[],
                double pval[], bool got_h, const void *userdata ){
    struct userdata_type *myuserdata = ( struct userdata_type * ) userdata;
    pval[0] = 2.0 * v[0];
    pval[1] = 2.0 * v[1];
    return 0;
}
// scale v
int scale( int n, int m, const double x[], double u[],
           const double v[], const void *userdata ){
    struct userdata_type *myuserdata = ( struct userdata_type * ) userdata;
    u[0] = v[0];
    u[1] = v[1];
    return 0;
}
// compute the dense Jacobian
int jac_dense( int n, int m, int jne, const double x[], double jval[],
               const void *userdata ){
    struct userdata_type *myuserdata = ( struct userdata_type * ) userdata;
```

```
    jval[0] = 2.0 * x[0];
    jval[1] = 0.0;
    jval[2] = 1.0;
    jval[3] = 2.0 * x[1];
    jval[4] = 1.0;
    jval[5] = - 1.0;
    return 0;
}
// compute the dense Hessian
int hess_dense( int n, int m, int hne, const double x[], const double y[],
                double hval[], const void *userdata ){
    struct userdata_type *myuserdata = ( struct userdata_type * ) userdata;
    hval[0] = 2.0 * y[0];
    hval[1] = 0.0;
    hval[2] = 2.0 * y[1];
    return 0;
}
// compute dense residual-Hessians-vector products
int rhessprods_dense( int n, int m, int pne, const double x[],
                      const double v[], double pval[], bool got_h,
                      const void *userdata ){
    struct userdata_type *myuserdata = ( struct userdata_type * ) userdata;
    pval[0] = 2.0 * v[0];
    pval[1] = 0.0;
    pval[2] = 0.0;
    pval[3] = 2.0 * v[1];
    pval[4] = 0.0;
    pval[5] = 0.0;
    return 0;
}
```

6.2 nlstf.c

This is the same example, but now fortran-style indexing is used.

```
/* nlstf.c */
/* Full test for the NLS interface using Fortran sparse matrix indexing */
/* Jari Fowkes & Nick Gould, STFC-Rutherford Appleton Laboratory, 2021 */
#include <stdio.h>
#include <math.h>
#include "nls.h"
#define max(a,b)  \
({  __typeof__ (a) _a = (a); \
    __typeof__ (b) _b = (b); \
    _a > _b ? _a : _b; })
// Custom userdata struct
struct userdata_type {
   double p;
};
// Function prototypes
int res( int n, int m, const double x[], double c[], const void * );
int jac( int n, int m, int jne, const double x[], double jval[], const void * );
int hess( int n, int m, int hne, const double x[], const double y[],
          double hval[], const void * );
int jacprod( int n, int m, const double x[], const bool transpose, double u[],
             const double v[], bool got_j, const void * );
int hessprod( int n, int m, const double x[], const double y[], double u[],
              const double v[], bool got_h, const void * );
int rhessprods( int n, int m, int pne, const double x[], const double v[],
                double pval[], bool got_h, const void * );
int scale( int n, int m, const double x[], double u[],
           const double v[], const void * );
int jac_dense( int n, int m, int jne, const double x[], double jval[],
               const void * );
int hess_dense( int n, int m, int hne, const double x[], const double y[],
                double hval[], const void * );
int rhessprods_dense( int n, int m, int pne, const double x[],
                      const double v[], double pval[], bool got_h,
                      const void * );
int main(void) {
    // Derived types
    void *data;
    struct nls_control_type control;
    struct nls_inform_type inform;
    // Set user data
    struct userdata_type userdata;
    userdata.p = 1.0;
    // Set problem data
    int n = 2; // # variables
    int m = 3; // # residuals
    int j_ne = 5; // Jacobian elements
```

```
    int h_ne = 2; // Hesssian elements
    int p_ne = 2; // residual-Hessians-vector products elements
    int J_row[] = {1, 2, 2, 3, 3}; // Jacobian J
    int J_col[] = {1, 1, 2, 1, 2}; //
    int J_ptr[] = {1, 2, 4, 6};    // row pointers
    int H_row[] = {1, 2};          // Hessian H
    int H_col[] = {1, 2};          // NB lower triangle
    int H_ptr[] = {1, 2, 3};       // row pointers
    int P_row[] = {1, 2};          // residual-Hessians-vector product matrix
    int P_ptr[] = {1, 2, 3, 3};    // column pointers
    // Set storage
    double g[n]; // gradient
    double c[m]; // residual
    double y[m]; // multipliers
    double W[] = {1.0, 1.0, 1.0}; // weights
    char st;
    int status;
    printf(" Fortran sparse matrix indexing\n\n");
    printf(" tests options for all-in-one storage format\n\n");
    for( int d=1; d <= 5; d++){
//  for( int d=5; d <= 5; d++){
        // Initialize NLS
        nls_initialize( &data, &control, &inform );
        // Set user-defined control options
        control.f_indexing = true; // Fortran sparse matrix indexing
        //control.print_level = 1;
        control.jacobian_available = 2;
        control.hessian_available = 2;
        control.model = 6;
        double x[] = {1.5,1.5}; // starting point
        double W[] = {1.0, 1.0, 1.0}; // weights
        switch(d){
            case 1: // sparse co-ordinate storage
                st = 'C';
                nls_import( &control, &data, &status, n, m,
                            "coordinate", j_ne, J_row, J_col, NULL,
                            "coordinate", h_ne, H_row, H_col, NULL,
                            "sparse_by_columns", p_ne, P_row, NULL, P_ptr, W );
                nls_solve_with_mat( &data, &userdata, &status,
                                    n, m, x, c, g, res, j_ne, jac,
                                    h_ne, hess, p_ne, rhessprods );
                break;
            case 2: // sparse by rows
                st = 'R';
                nls_import( &control, &data, &status, n, m,
                            "sparse_by_rows", j_ne, NULL, J_col, J_ptr,
                            "sparse_by_rows", h_ne, NULL, H_col, H_ptr,
                            "sparse_by_columns", p_ne, P_row, NULL, P_ptr, W );
                nls_solve_with_mat( &data, &userdata, &status,
                                    n, m, x, c, g, res, j_ne, jac,
                                    h_ne, hess, p_ne, rhessprods );
                break;
            case 3: // dense
                st = 'D';
                nls_import( &control, &data, &status, n, m,
                            "dense", j_ne, NULL, NULL, NULL,
                            "dense", h_ne, NULL, NULL, NULL,
                            "dense", p_ne, NULL, NULL, NULL, W );
                nls_solve_with_mat( &data, &userdata, &status,
                                    n, m, x, c, g, res, j_ne, jac_dense,
                                    h_ne, hess_dense, p_ne, rhessprods_dense );
                break;
            case 4: // diagonal
                st = 'I';
                nls_import( &control, &data, &status, n, m,
                            "sparse_by_rows", j_ne, NULL, J_col, J_ptr,
                            "diagonal", h_ne, NULL, NULL, NULL,
                            "sparse_by_columns", p_ne, P_row, NULL, P_ptr, W );
                nls_solve_with_mat( &data, &userdata, &status,
                                    n, m, x, c, g, res, j_ne, jac,
                                    h_ne, hess, p_ne, rhessprods );
                break;
            case 5: // access by products
                st = 'P';
                nls_import( &control, &data, &status, n, m,
                            "absent", j_ne, NULL, NULL, NULL,
                            "absent", h_ne, NULL, NULL, NULL,
                            "sparse_by_columns", p_ne, P_row, NULL, P_ptr, W );
                nls_solve_without_mat( &data, &userdata, &status,
                                       n, m, x, c, g, res, jacprod,
                                       hessprod, p_ne, rhessprods );
                break;
        }
        nls_information( &data, &inform, &status );
        if(inform.status == 0){
            printf("%c:%6i iterations. Optimal objective value = %5.2f"
                   " status = %1i\n",
```

```
                  st, inform.iter, inform.obj, inform.status);
        }else{
            printf("%c: NLS_solve exit status = %1i\n", st, inform.status);
        }
        // Delete internal workspace
        nls_terminate( &data, &control, &inform );
    }
    printf("\n tests reverse-communication options\n\n");
    // reverse-communication input/output
    int eval_status;
    double u[max(m,n)], v[max(m,n)];
    double J_val[j_ne], J_dense[m*n];
    double H_val[h_ne], H_dense[n*(n+1)/2], H_diag[n];
    double P_val[p_ne], P_dense[m*n];
    bool transpose;
    bool got_j = false;
    bool got_h = false;
    for( int d=1; d <= 5; d++){
//  for( int d=1; d <= 4; d++){
        // Initialize NLS
        nls_initialize( &data, &control, &inform );
        // Set user-defined control options
        control.f_indexing = true; // Fortran sparse matrix indexing
        //control.print_level = 1;
        control.jacobian_available = 2;
        control.hessian_available = 2;
        control.model = 6;
        double x[] = {1.5,1.5}; // starting point
        double W[] = {1.0, 1.0, 1.0}; // weights
        switch(d){
            case 1: // sparse co-ordinate storage
                st = 'C';
                nls_import( &control, &data, &status, n, m,
                            "coordinate", j_ne, J_row, J_col, NULL,
                            "coordinate", h_ne, H_row, H_col, NULL,
                            "sparse_by_columns", p_ne, P_row, NULL, P_ptr, W );
                while(true){ // reverse-communication loop
                  nls_solve_reverse_with_mat( &data, &status, &eval_status,
                                              n, m, x, c, g, j_ne, J_val, y,
                                              h_ne, H_val, v, p_ne, P_val );
                  if(status == 0){ // successful termination
                        break;
                  }else if(status < 0){ // error exit
                      break;
                  }else if(status == 2){ // evaluate c
                      eval_status = res( n, m, x, c, &userdata );
                  }else if(status == 3){ // evaluate J
                      eval_status = jac( n, m, j_ne, x, J_val, &userdata );
                  }else if(status == 4){ // evaluate H
                      eval_status = hess( n, m, h_ne, x, y, H_val, &userdata );
                  }else if(status == 7){ // evaluate P
                      eval_status = rhessprods( n, m, p_ne, x, v, P_val,
                                                got_h, &userdata );
                  }else{
                      printf(" the value %1i of status should not occur\n",
                        status);
                      break;
                  }
                }
                break;
            case 2: // sparse by rows
                st = 'R';
                nls_import( &control, &data, &status, n, m,
                            "sparse_by_rows", j_ne, NULL, J_col, J_ptr,
                            "sparse_by_rows", h_ne, NULL, H_col, H_ptr,
                            "sparse_by_columns", p_ne, P_row, NULL, P_ptr, W );
                while(true){ // reverse-communication loop
                  nls_solve_reverse_with_mat( &data, &status, &eval_status,
                                              n, m, x, c, g, j_ne, J_val, y,
                                              h_ne, H_val, v, p_ne, P_val );
                  if(status == 0){ // successful termination
                        break;
                  }else if(status < 0){ // error exit
                      break;
                  }else if(status == 2){ // evaluate c
                      eval_status = res( n, m, x, c, &userdata );
                  }else if(status == 3){ // evaluate J
                      eval_status = jac( n, m, j_ne, x, J_val, &userdata );
                  }else if(status == 4){ // evaluate H
                      eval_status = hess( n, m, h_ne, x, y, H_val, &userdata );
                  }else if(status == 7){ // evaluate P
                      eval_status = rhessprods( n, m, p_ne, x, v, P_val,
                                                got_h, &userdata );
                  }else{
                      printf(" the value %1i of status should not occur\n",
                        status);
                      break;
```

```
            }
          }
          break;
        case 3: // dense
          st = 'D';
          nls_import( &control, &data, &status, n, m,
                      "dense", j_ne, NULL, NULL, NULL,
                      "dense", h_ne, NULL, NULL, NULL,
                      "dense", p_ne, NULL, NULL, NULL, W );
          while(true){ // reverse-communication loop
            nls_solve_reverse_with_mat( &data, &status, &eval_status,
                                        n, m, x, c, g, j_ne, J_dense, y,
                                        h_ne, H_dense, v, p_ne, P_dense );
            if(status == 0){ // successful termination
                  break;
            }else if(status < 0){ // error exit
                break;
            }else if(status == 2){ // evaluate c
                eval_status = res( n, m, x, c, &userdata );
            }else if(status == 3){ // evaluate J
                eval_status = jac_dense( n, m, j_ne, x, J_dense,
                                         &userdata );
            }else if(status == 4){ // evaluate H
                eval_status = hess_dense( n, m, h_ne, x, y, H_dense,
                                          &userdata );
            }else if(status == 7){ // evaluate P
                eval_status = rhessprods_dense( n, m, p_ne, x, v, P_dense,
                                                got_h, &userdata );
            }else{
                printf(" the value %1i of status should not occur\n",
                  status);
                break;
            }
          }
          break;
        case 4: // diagonal
          st = 'I';
          nls_import( &control, &data, &status, n, m,
                      "sparse_by_rows", j_ne, NULL, J_col, J_ptr,
                      "diagonal", h_ne, NULL, NULL, NULL,
                      "sparse_by_columns", p_ne, P_row, NULL, P_ptr, W );
          while(true){ // reverse-communication loop
            nls_solve_reverse_with_mat( &data, &status, &eval_status,
                                        n, m, x, c, g, j_ne, J_val, y,
                                        h_ne, H_diag, v, p_ne, P_val );
            if(status == 0){ // successful termination
                  break;
            }else if(status < 0){ // error exit
                break;
            }else if(status == 2){ // evaluate c
                eval_status = res( n, m, x, c, &userdata );
            }else if(status == 3){ // evaluate J
                eval_status = jac( n, m, j_ne, x, J_val, &userdata );
            }else if(status == 4){ // evaluate H
                eval_status = hess( n, m, h_ne, x, y, H_diag, &userdata );
            }else if(status == 7){ // evaluate P
                eval_status = rhessprods( n, m, p_ne, x, v, P_val,
                                          got_h, &userdata );
            }else{
                printf(" the value %1i of status should not occur\n",
                  status);
                break;
            }
          }
          break;
        case 5: // access by products
          st = 'P';
//        control.print_level = 1;
          nls_import( &control, &data, &status, n, m,
                      "absent", j_ne, NULL, NULL, NULL,
                      "absent", h_ne, NULL, NULL, NULL,
                      "sparse_by_columns", p_ne, P_row, NULL, P_ptr, W );
          while(true){ // reverse-communication loop
            nls_solve_reverse_without_mat( &data, &status, &eval_status,
                                           n, m, x, c, g, &transpose,
                                           u, v, y, p_ne, P_val );
            if(status == 0){ // successful termination
                  break;
            }else if(status < 0){ // error exit
                break;
            }else if(status == 2){ // evaluate c
                eval_status = res( n, m, x, c, &userdata );
            }else if(status == 5){ // evaluate u + J v or u + J'v
                eval_status = jacprod( n, m, x, transpose, u, v, got_j,
                                       &userdata );
            }else if(status == 6){ // evaluate u + H v
                eval_status = hessprod( n, m, x, y, u, v, got_h,
```

```
                                        &userdata );
                }else if(status == 7){ // evaluate P
                    eval_status = rhessprods( n, m, p_ne, x, v, P_val,
                                              got_h, &userdata );
                }else{
                    printf(" the value %1i of status should not occur\n",
                      status);
                    break;
                }
              }
              break;
    }
    nls_information( &data, &inform, &status );
    if(inform.status == 0){
        printf("%c:%6i iterations. Optimal objective value = %5.2f"
               " status = %1i\n",
               st, inform.iter, inform.obj, inform.status);
    }else{
        printf("%c: NLS_solve exit status = %1i\n", st, inform.status);
    }
    // Delete internal workspace
    nls_terminate( &data, &control, &inform );
}
printf("\n basic tests of models used, direct access\n\n");
for( int model=3; model <= 8; model++){
    // Initialize NLS
    nls_initialize( &data, &control, &inform );
    // Set user-defined control options
    control.f_indexing = true; // Fortran sparse matrix indexing
    //control.print_level = 1;
    control.jacobian_available = 2;
    control.hessian_available = 2;
    control.model = model;
    double x[] = {1.5,1.5}; // starting point
    double W[] = {1.0, 1.0, 1.0}; // weights
    nls_import( &control, &data, &status, n, m,
                "sparse_by_rows", j_ne, NULL, J_col, J_ptr,
                "sparse_by_rows", h_ne, NULL, H_col, H_ptr,
                "sparse_by_columns", p_ne, P_row, NULL, P_ptr, W );
    nls_solve_with_mat( &data, &userdata, &status,
                        n, m, x, c, g, res, j_ne, jac,
                        h_ne, hess, p_ne, rhessprods );
    nls_information( &data, &inform, &status );
    if(inform.status == 0){
        printf(" %1i:%6i iterations. Optimal objective value = %5.2f"
               " status = %1i\n",
               model, inform.iter, inform.obj, inform.status);
    }else{
        printf(" %i: NLS_solve exit status = %1i\n", model, inform.status);
    }
    // Delete internal workspace
    nls_terminate( &data, &control, &inform );
}
printf("\n basic tests of models used, access by products\n\n");
for( int model=3; model <= 8; model++){
    // Initialize NLS
    nls_initialize( &data, &control, &inform );
    // Set user-defined control options
    control.f_indexing = true; // Fortran sparse matrix indexing
    //control.print_level = 1;
    control.jacobian_available = 2;
    control.hessian_available = 2;
    control.model = model;
    double x[] = {1.5,1.5}; // starting point
    double W[] = {1.0, 1.0, 1.0}; // weights
    nls_import( &control, &data, &status, n, m,
                "absent", j_ne, NULL, NULL, NULL,
                "absent", h_ne, NULL, NULL, NULL,
                "sparse_by_columns", p_ne, P_row, NULL, P_ptr, W );
    nls_solve_without_mat( &data, &userdata, &status,
                           n, m, x, c, g, res, jacprod,
                           hessprod, p_ne, rhessprods );
    nls_information( &data, &inform, &status );
    if(inform.status == 0){
        printf("P%1i:%6i iterations. Optimal objective value = %5.2f"
               " status = %1i\n",
               model, inform.iter, inform.obj, inform.status);
    }else{
        printf("P%i: NLS_solve exit status = %1i\n", model, inform.status);
    }
    // Delete internal workspace
    nls_terminate( &data, &control, &inform );
}
printf("\n basic tests of models used, reverse access\n\n");
for( int model=3; model <= 8; model++){
    // Initialize NLS
    nls_initialize( &data, &control, &inform );
```

```
    // Set user-defined control options
    control.f_indexing = true; // Fortran sparse matrix indexing
    //control.print_level = 1;
    control.jacobian_available = 2;
    control.hessian_available = 2;
    control.model = model;
    double x[] = {1.5,1.5}; // starting point
    double W[] = {1.0, 1.0, 1.0}; // weights
    nls_import( &control, &data, &status, n, m,
                "sparse_by_rows", j_ne, NULL, J_col, J_ptr,
                "sparse_by_rows", h_ne, NULL, H_col, H_ptr,
                "sparse_by_columns", p_ne, P_row, NULL, P_ptr, W );
    while(true){ // reverse-communication loop
      nls_solve_reverse_with_mat( &data, &status, &eval_status,
                                  n, m, x, c, g, j_ne, J_val, y,
                                  h_ne, H_val, v, p_ne, P_val );
      if(status == 0){ // successful termination
            break;
      }else if(status < 0){ // error exit
          break;
      }else if(status == 2){ // evaluate c
          eval_status = res( n, m, x, c, &userdata );
      }else if(status == 3){ // evaluate J
          eval_status = jac( n, m, j_ne, x, J_val, &userdata );
      }else if(status == 4){ // evaluate H
          eval_status = hess( n, m, h_ne, x, y, H_val, &userdata );
      }else if(status == 7){ // evaluate P
          eval_status = rhessprods( n, m, p_ne, x, v, P_val,
                                    got_h, &userdata );
      }else{
          printf(" the value %1i of status should not occur\n",
            status);
          break;
      }
    }
    nls_information( &data, &inform, &status );
    if(inform.status == 0){
        printf("P%1i:%6i iterations. Optimal objective value = %5.2f"
               " status = %1i\n",
               model, inform.iter, inform.obj, inform.status);
    }else{
        printf(" %i: NLS_solve exit status = %1i\n", model, inform.status);
    }
    // Delete internal workspace
    nls_terminate( &data, &control, &inform );
}
printf("\n basic tests of models used, reverse access by products\n\n");
for( int model=3; model <= 8; model++){
    // Initialize NLS
    nls_initialize( &data, &control, &inform );
    // Set user-defined control options
    control.f_indexing = true; // Fortran sparse matrix indexing
    //control.print_level = 1;
    control.jacobian_available = 2;
    control.hessian_available = 2;
    control.model = model;
    double x[] = {1.5,1.5}; // starting point
    double W[] = {1.0, 1.0, 1.0}; // weights
    nls_import( &control, &data, &status, n, m,
                "absent", j_ne, NULL, NULL, NULL,
                "absent", h_ne, NULL, NULL, NULL,
                "sparse_by_columns", p_ne, P_row, NULL, P_ptr, W );
    while(true){ // reverse-communication loop
      nls_solve_reverse_without_mat( &data, &status, &eval_status,
                                     n, m, x, c, g, &transpose,
                                     u, v, y, p_ne, P_val );
      if(status == 0){ // successful termination
            break;
      }else if(status < 0){ // error exit
          break;
      }else if(status == 2){ // evaluate c
          eval_status = res( n, m, x, c, &userdata );
      }else if(status == 5){ // evaluate u + J v or u + J'v
          eval_status = jacprod( n, m, x, transpose, u, v, got_j,
                                 &userdata );
      }else if(status == 6){ // evaluate u + H v
          eval_status = hessprod( n, m, x, y, u, v, got_h,
                                  &userdata );
      }else if(status == 7){ // evaluate P
          eval_status = rhessprods( n, m, p_ne, x, v, P_val,
                                    got_h, &userdata );
      }else{
          printf(" the value %1i of status should not occur\n",
            status);
          break;
      }
    }
```

```
        nls_information( &data, &inform, &status );
        if(inform.status == 0){
            printf("P%1i:%6i iterations. Optimal objective value = %5.2f"
                   " status = %1i\n",
                   model, inform.iter, inform.obj, inform.status);
        }else{
            printf("P%i: NLS_solve exit status = %1i\n", model, inform.status);
        }
        // Delete internal workspace
        nls_terminate( &data, &control, &inform );
    }
}
// compute the residuals
int res( int n, int m, const double x[], double c[], const void *userdata ){
    struct userdata_type *myuserdata = ( struct userdata_type * ) userdata;
    double p = myuserdata->p;
    c[0] = pow(x[0],2.0) + p;
    c[1] = x[0] + pow(x[1],2.0);
    c[2] = x[0] - x[1];
    return 0;
}
// compute the Jacobian
int jac( int n, int m, int jne, const double x[], double jval[],
         const void *userdata ){
    struct userdata_type *myuserdata = ( struct userdata_type * ) userdata;
    jval[0] = 2.0 * x[0];
    jval[1] = 1.0;
    jval[2] = 2.0 * x[1];
    jval[3] = 1.0;
    jval[4] = - 1.0;
    return 0;
}
// compute the Hessian
int hess( int n, int m, int hne, const double x[], const double y[],
          double hval[], const void *userdata ){
    struct userdata_type *myuserdata = ( struct userdata_type * ) userdata;
    hval[0] = 2.0 * y[0];
    hval[1] = 2.0 * y[1];
    return 0;
}
// compute Jacobian-vector products
int jacprod( int n, int m, const double x[], const bool transpose, double u[],
             const double v[], bool got_j, const void *userdata ){
    struct userdata_type *myuserdata = ( struct userdata_type * ) userdata;
    if (transpose) {
      u[0] = u[0] + 2.0 * x[0] * v[0] + v[1] + v[2];
      u[1] = u[1] + 2.0 * x[1] * v[1] - v[2];
    }else{
      u[0] = u[0] + 2.0 * x[0] * v[0];
      u[1] = u[1] + v[0]  + 2.0 * x[1] * v[1];
      u[2] = u[2] + v[0] - v[1];
    }
    return 0;
}
// compute Hessian-vector products
int hessprod( int n, int m, const double x[], const double y[], double u[],
              const double v[], bool got_h, const void *userdata ){
    struct userdata_type *myuserdata = ( struct userdata_type * ) userdata;
    u[0] = u[0] + 2.0 * y[0] * v[0];
    u[1] = u[1] + 2.0 * y[1] * v[1];
    return 0;
}
// compute residual-Hessians-vector products
int rhessprods( int n, int m, int pne, const double x[], const double v[],
                double pval[], bool got_h, const void *userdata ){
    struct userdata_type *myuserdata = ( struct userdata_type * ) userdata;
    pval[0] = 2.0 * v[0];
    pval[1] = 2.0 * v[1];
    return 0;
}
// scale v
int scale( int n, int m, const double x[], double u[],
           const double v[], const void *userdata ){
    struct userdata_type *myuserdata = ( struct userdata_type * ) userdata;
    u[0] = v[0];
    u[1] = v[1];
    return 0;
}
// compute the dense Jacobian
int jac_dense( int n, int m, int jne, const double x[], double jval[],
               const void *userdata ){
    struct userdata_type *myuserdata = ( struct userdata_type * ) userdata;
    jval[0] = 2.0 * x[0];
    jval[1] = 0.0;
    jval[2] = 1.0;
    jval[3] = 2.0 * x[1];
    jval[4] = 1.0;
```

```
    jval[5] = - 1.0;
    return 0;
}
// compute the dense Hessian
int hess_dense( int n, int m, int hne, const double x[], const double y[],
                double hval[], const void *userdata ){
    struct userdata_type *myuserdata = ( struct userdata_type * ) userdata;
    hval[0] = 2.0 * y[0];
    hval[1] = 0.0;
    hval[2] = 2.0 * y[1];
    return 0;
}
// compute dense residual-Hessians-vector products
int rhessprods_dense( int n, int m, int pne, const double x[],
                      const double v[], double pval[], bool got_h,
                      const void *userdata ){
    struct userdata_type *myuserdata = ( struct userdata_type * ) userdata;
    pval[0] = 2.0 * v[0];
    pval[1] = 0.0;
    pval[2] = 0.0;
    pval[3] = 2.0 * v[1];
    pval[4] = 0.0;
    pval[5] = 0.0;
    return 0;
}
```

Index