RALFit Documentation

Release 0.0.1

STFC Numerical Analysis Group

Mar 28, 2019

Contents

1 Contents **3**
- 1.1 Installation . . . 3
- 1.2 How to use the package . . . 3
- 1.3 Description of the method used . . . 14
- 1.4 Examples . . . 20
- 1.5 Indices and tables . . . 23

RALFit computes a solution $\boldsymbol{x}$ to the non-linear least-squares problem

$$\min_{\boldsymbol{x}} \; F(\boldsymbol{x}) := \frac{1}{2}\|\boldsymbol{r}(\boldsymbol{x})\|^2_{\boldsymbol{W}} + \frac{\sigma}{p}\|\boldsymbol{x}\|^p_2,$$

where $\boldsymbol{W} \in \mathbb{R}^{m\times m}$ is a diagonal, non-negative, weighting matrix, and $\boldsymbol{r}(\boldsymbol{x}) = (r_1(\boldsymbol{x}), r_2(\boldsymbol{x}), ..., r_m(\boldsymbol{x}))^T$ is a non-linear function.

A typical use may be to fit a function $f(\boldsymbol{x})$ to the data $y_i,\ t_i$, weighted by the uncertainty of the data, σ_i, so that

$$r_i(\boldsymbol{x}) := y_i - f(\boldsymbol{x}; t_i),$$

and $\boldsymbol{W}$ is the diagonal matrix such that $\boldsymbol{W}_{ii} = (1/\sqrt{\sigma_i})$. For this reason we refer to the function $\boldsymbol{r}$ as the *residual* function.

Various algorithms for solving this problem are implemented – see *Description of the method used*.

CHAPTER 1

Contents

1.1 Installation

1.1.1 Obtaining the code

The latest version of the code can be downloaded from GitHub by issuing the command

```
git clone https://github.com/ralna/RALFit.git
```

1.1.2 Building the library

From the `RALFit/libRALFit/` directory, issue the commands:

```
mkdir build
cd build
cmake ..
make
```

1.2 How to use the package

1.2.1 Overview

Calling sequences

Access to the package requires a `USE` statement

```
use ral_nlls_double
```

The user can then call one of the procedures:

`nlls_solve()` solves the non-linear least squares problem.

`nlls_iterate()` performs one iteration of the non-linear least squares solver.

The calling sequences of these subroutines are outlined in *Argument lists and calling sequences*.

The derived data types

For each problem, the user must employ the derived types defined by the module to declare scalars of the types `nlls_inform` and `nlls_options`. If `nlls_iterate()` is to be used, then a scalar of the type `nlls_workspace` must also be defined. The following pseudocode illustrates this.

```
use nlls_module
!...
type (NLLS_inform) :: inform
type (NLLS_options) :: options
type (NLLS_workspace) :: work ! needed if nlls_iterate to be called
!...
```

The components of `nlls_options` and `nlls_inform` are explained below in *Data types*.

1.2.2 Argument lists and calling sequences

We use square brackets to indicate arguments. In each call, optional arguments follow the argument inform. Since we reserve the right to add additional optional arguments in future releases of the code, **we strongly recommend that all optional arguments be called by keyword, not by position**.

The term **package type** is used to mean double precision.

To solve the non-linear least squares problem

subroutine nlls_solve(*n*, *m*, *X*, *eval_r*, *eval_J*, *eval_Hf*, *params*, *options*, *inform*[, *weights*, *eval_HP*])

Solves the non-linear least squares problem.

Parameters

- **n** *[integer,in]* :: holds the number n of variables to be fitted; i.e., n is the length of the unknown vector $\boldsymbol{x}$. **Restriction: n** > 0.
- **m** *[integer,in]* :: holds the number m of data points available; i.e., m is the number of residuals r_i. **Restriction: m** ≥ 0.
- **X** (n) *[real,inout]* :: on entry, it must hold the initial guess for $\boldsymbol{x}$, and on successful exit it holds the solution to the non-linear least squares problem.
- **eval_r** *[procedure]* :: given a point $\boldsymbol{x}_k$, returns the vector $\boldsymbol{r}(\boldsymbol{x}_k)$. Further details of the format required are given in `eval_r()` in *User-supplied function evaluation routines*.
- **eval_J** *[procedure]* :: given a point $\boldsymbol{x}_k$, returns the $m \times n$ Jacobian matrix, $\boldsymbol{J}_k$, of $\boldsymbol{r}$ evaluated at $\boldsymbol{x}_k$. Further details of the format required are given in `eval_J()` in *User-supplied function evaluation routines*.
- **eval_Hf** *[procedure]* :: given vectors $\boldsymbol{x} \in \mathbb{R}^n$ and $\boldsymbol{r} \in \mathbb{R}^m$, returns the quantity $\sum_{i=1}^{m}(\boldsymbol{r})_i \nabla^2 \boldsymbol{r}_i(\boldsymbol{x})$. Further details of the format required are given in `eval_Hf()`

in *User-supplied function evaluation routines*. If `exact_second_derivative = .false.` in `nlls_options`, then this is not referenced.

- **params** *[params_base_type,in]* :: holds parameters to be passed to the user-defined routines `eval_r()`, `eval_J()`, and `eval_Hf()`. Further details of its use are given in *User-supplied function evaluation routines*.
- **options** *[*nlls_options*,in]* :: controls execution of algorithm.
- **inform** *[*nlls_inform*,out]* :: components provide information about the execution of the subroutine.

Options

- **weights** (n) *[real]* :: If present, this holds the square-roots of the diagonal entries of the weighting matrix, $\boldsymbol{W}$. If absent, then the norm in the least squares problem is taken to be the 2-norm, that is, $\boldsymbol{W} = I$.
- **eval_HP** *[procedure]* :: If present, this is a routine that, given vectors $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^m$, returns the matrix $P(\boldsymbol{x}, \boldsymbol{y}) := (H_1(\boldsymbol{x})\boldsymbol{y} \ldots H_m(\boldsymbol{x})\boldsymbol{y})$. Further details of the format required are given in `eval_HP()` in *User-supplied function evaluation routines*. This is only referenced if `model = 4` in `nlls_options`.

To iterate once

subroutine nlls_iterate(*n*, *m*, *X*, *eval_r*, *eval_J*, *eval_Hf*, *params*, *options*, *inform*[, *weights*])

A call of this form allows the user to step through the solution process one iteration at a time.

n, **m**, **eval_F**, **eval_J**, **eval_HF**, **params**, **info**, **options** and **weights** are as in the desciption of `nlls_solve()`.

Parameters

- **X** (n) *[real,inout]* :: on the first call it must hold the initial guess for $\boldsymbol{x}$. On return it holds the value of $\boldsymbol{x}$ at the current iterate, and must be passed unaltered to any subsequent call to `nlls_iterate()`.
- **w** *[*nlls_workspace*,inout]* :: is used to store the current state of the iteration and should not be altered by the user.

The user may use the components `convergence_normf` and `convergence_normg` and `converge_norms` in `nlls_inform` to determine whether the iteration has converged.

1.2.3 User-supplied function evaluation routines

The user must supply routines to evaluate the residual, Jacobian and Hessian at a point. **RALFit** will call these routines internally.

In order to pass user-defined data into the evaluation calls, `params_base_type` is extended to a `user_type`, as follows:

```
type, extends( params_base_type ) :: user_type
   ! code declaring components of user_type
end type user_type
```

We recommend this type is wrapped in a module with the user-defined routines for evaluating the function, Jacobian, and Hessian.

The components of the extended type are accessed through a `select type` construct:

```
select type(params)
type is(user_type)
  ! code that accesses components of params that were defined within user_type
end select
```

For evaluating the function $r(x)$

A subroutine must be supplied to calculate $\boldsymbol{r}(\boldsymbol{x})$ for a given vector $\boldsymbol{x}$. It must implement the following interface:

```
abstract interface
   subroutine eval_r(status, n, m, x, r, params)
      integer, intent(inout) :: status
      integer, intent(in) :: n
      integer, intent(in) :: m
      double precision, dimension(n), intent(in) :: x
      double precision, dimension(m), intent(out) :: r
      class(params_base_type), intent(in) :: params
   end subroutine eval_r
end interface
```

subroutine eval_r (*status*, *n*, *m*, *x*, *r*, *params*)

Parameters

- **status** *[integer,inout]* :: is initialised to 0 before the routine is called. If it is set to a non-zero value by the routine, then *nlls_solve()* / *nlls_iterate()* will exit with error.
- **n** *[integer,in]* :: is passed unchanged as provided in the call to *nlls_solve()*/*nlls_iterate()*.
- **m** *[integer,in]* :: is passed unchanged as provided in the call to *nlls_solve()*/*nlls_iterate()*.
- **X** (n) *[real,in]* :: holds the current point $\boldsymbol{x}_k$ at which to evaluate $\boldsymbol{r}(\boldsymbol{x}_k)$.
- **r** (m) *[real,out]* :: must be set by the routine to hold the residual function evaluated at the current point $\boldsymbol{x}_k$, $\boldsymbol{r}(\boldsymbol{x}_k)$.
- **params** *[params_base_type,in]* :: is passed unchanged as provided in the call to *nlls_solve()*/*nlls_iterate()*.

For evaluating the function $J = \nabla r(x)$

A subroutine must be supplied to calculate $\boldsymbol{J} = \nabla \boldsymbol{r}(\boldsymbol{x})$ for a given vector $\boldsymbol{x}$. It must implement the following interface:

```
abstract interface
   subroutine eval_J(status, n, m, x, J, params)
      integer, intent(inout) :: status
      integer, intent(in) :: n
      integer, intent(in) :: m
      double precision, dimension(n), intent(in)  :: x
      double precision, dimension(n*m), intent(out) :: J
      class(params_base_type), intent(in) :: params
  end subroutine eval_J
end interface
```

subroutine eval_J (*status*, *n*, *m*, *x*, *J*, *params*)

Parameters

- **status** *[integer,inout]* :: is initialised to `0` before the routine is called. If it is set to a non-zero value by the routine, then `nlls_solve()` / `nlls_iterate()` will exit with error.
- **n** *[integer,in]* :: is passed unchanged as provided in the call to `nlls_solve()`/`nlls_iterate()`.
- **m** *[integer,in]* :: is passed unchanged as provided in the call to `nlls_solve()`/`nlls_iterate()`.
- **X** (n) *[real,in]* :: holds the current point $\boldsymbol{x}_k$ at which to evaluate $\boldsymbol{J}(\boldsymbol{x}_k)$.
- **J** (m*n) *[real,out]* :: must be set by the routine to hold the Jacobian of the residual function evaluated at the current point $\boldsymbol{x}_k$, $\boldsymbol{r}(\boldsymbol{x}_k)$. `J(i*m+j)` must be set to hold $\nabla_{x_j} r_i(\boldsymbol{x}_k)$.
- **params** *[params_base_type,in]* :: is passed unchanged as provided in the call to `nlls_solve()`/`nlls_iterate()`.

For evaluating the function $Hf = \sum_{i=1}^{m} r_i(\boldsymbol{x}) \boldsymbol{W} \nabla^2 r_i(\boldsymbol{x})$

A subroutine must be supplied to calculate $Hf = \sum_{i=1}^{m} (\boldsymbol{r})_i \nabla^2 r_i(\boldsymbol{x})$ for given vectors $\boldsymbol{x} \in \mathbb{R}^n$ and $\boldsymbol{r} \in \mathbb{R}^m$; here $(\boldsymbol{r})_i$ denotes the ith component of the vector $\boldsymbol{r}$. The subroutine must implement the following interface:

```
abstract interface
   subroutine eval_Hf_type(status, n, m, x, r, Hf, params)
       integer, intent(inout) :: status
       integer, intent(in) :: n
       integer, intent(in) :: m
       double precision, dimension(n), intent(in)  :: x
       double precision, dimension(m), intent(in)  :: r
       double precision, dimension(n*n), intent(out) :: Hf
       class(params_base_type), intent(in) :: params
     end subroutine eval_Hf_type
end interface
:language: fortran
```

subroutine eval_Hf (*status*, *n*, *m*, *x*, *r*, *Hf*, *params*)

Parameters

- **status** *[integer,inout]* :: is initialised to `0` before the routine is called. If it is set to a non-zero value by the routine, then `nlls_solve()` / `nlls_iterate()` will exit with error.
- **n** *[integer,in]* :: is passed unchanged as provided in the call to `nlls_solve()`/`nlls_iterate()`.
- **m** *[integer,in]* :: is passed unchanged as provided in the call to `nlls_solve()`/`nlls_iterate()`.
- **X** (n) *[real,in]* :: holds the current point $\boldsymbol{x}_k$ at which to evaluate $\sum_{i=1}^{m} (\boldsymbol{r})_i \nabla^2 r_i(\boldsymbol{x})$.
- **r** (m) *[real,in]* :: holds $\boldsymbol{W}\boldsymbol{r}(\boldsymbol{x})$, the (weighted) residual, as computed from vector returned by the last call to `eval_r()`.
- **Hf** (n*n) *[real,out]* :: must be set by the routine to hold the matrix $\sum_{i=1}^{m} (\boldsymbol{r})_i \nabla^2 r_i(\boldsymbol{x}_k)$, held by columns as a vector, where $(\boldsymbol{r})_i$ denotes the ith component of `r`, the vector passed to the routine.
- **params** *[params_base_type,in]* :: is passed unchanged as provided in the call to `nlls_solve()`/`nlls_iterate()`.

For evaluating the function $P(x, y) := (H_1(x)y \dots H_m(x)y)$

A subroutine may be supplied to calculate $P(x, y) := (H_1(x)y \dots H_m(x)y)$ for given vectors $x, y \in \mathbb{R}^n$. The subroutine must implement the following interface:

```
abstract interface
   subroutine eval_HP_type(status, n, m, x, y, HP, params)
       integer, intent(inout) :: status
       integer, intent(in) :: n
       integer, intent(in) :: m
       double precision, dimension(n), intent(in)  :: x
       double precision, dimension(n), intent(in)  :: y
       double precision, dimension(n*m), intent(out) :: HP
       class(params_base_type), intent(in) :: params
     end subroutine eval_HP_type
end interface
:language: fortran
```

subroutine eval_HP (*status*, *n*, *m*, *x*, *y*, *HP*, *params*)

Parameters

- **status** *[integer,inout]* :: is initialised to 0 before the routine is called. If it is set to a non-zero value by the routine, then *nlls_solve()* / *nlls_iterate()* will exit with error.
- **n** *[integer,in]* :: is passed unchanged as provided in the call to *nlls_solve()*/*nlls_iterate()*.
- **m** *[integer,in]* :: is passed unchanged as provided in the call to *nlls_solve()*/*nlls_iterate()*.
- **x** (n) *[real,in]* :: holds the current point x_k at which to evaluate the Hessians $\nabla^2 r_i(x_k)$.
- **y** (n) *[real,in]* :: holds y, the vector which multiplies each Hessian.
- **HP** (n*m) *[real,out]* :: must be set by the routine to hold the matrix $P(x, y) := (H_1(x)y \dots H_m(x)y)$, held by columns as a vector.
- **params** *[params_base_type,in]* :: is passed unchanged as provided in the call to *nlls_solve()*/*nlls_iterate()*.

1.2.4 Data types

The derived data type for holding options

type nlls_options

This is used to hold controlling data. The components are automatically given default values in the definition of the type.

Printing Controls

Type fields

- **% error** *[integer,default=6]* :: the Fortran unit number for error messages. If it is negative, these messages will be suppressed.
- **% out** *[integer,default=6]* :: the Fortran unit number for general messages. If it is negative, these messages will be suppressed.

- **`% print_level`** *[integer,default=0]* :: controls the level of output required. Options are:

< 1	No informational output will occur.
1	Gives a one-line summary for each iteration.
2	As 1, plus gives a summary of the inner iteration for each iteration.
3	As 2, plus gives more verbose (debugging) output.

Choice of Algorithm

Type fields

- **`% model`** *[integer,default=3]* :: specifies the model, $m_k(\cdot)$, used. Possible values are:

1	Gauss-Newton (no Hessian).
2	(Quasi-)Newton (uses exact Hessian if `exact_second_derivatives` is true, otherwise builds an approximation to the Hessian).
3	Hybrid method (mixture of Gauss-Newton/(Quasi-)Newton, as determined by the package).
4	Newton-tensor method.

  See *The models* for details.

- **`% type_of_method`** *[integer,default=1]* :: specifies the type of globalization method used. Possible values are:

1	Trust-region method.
2	Regularization.

  See *Subproblem solves* for details.

- **`% nlls_method`** *[integer,default=4]* :: specifies the method used to solve (or approximate the solution to) the trust-region sub problem. Possible values are:

1	Powell's dogleg method (approximates the solution).
2	The Adachi-Iwata-Nakatsukasa-Takeda (AINT) method.
3	The More-Sorensen method.
4	`Galahad`'s `DTRS` method if `type_of_method=1`, or `Galahad`'s `DRQS` method if `type_of_method=2`.

  See *Subproblem solves* for details.

- **`% exact_second_derivatives`** *[logical,default=false]* :: if `true`, signifies that the exact second derivatives are available (and, if `false`, approximates them using a secant method).

Stopping rules

Type fields

- **`% maxit`** *[integer,default=100]* :: gives the number of iterations the algorithm is allowed to take before being stopped. This is not accessed if *`nlls_iterate()`* is used.

- **`% stop_g_absolute`** *[real,default=1e-5]* :: specifies the absolute tolerance used in the convergence test on $\|\boldsymbol{J}_k{}^T\boldsymbol{r}(\boldsymbol{x}_k))\|/\|\boldsymbol{r}(\boldsymbol{x}_k)\|$.

- **`% stop_g_relative`** *[real,default=1e-8]* :: specifies the relative tolerance used in the convergence test on $\|\boldsymbol{J}_k{}^T\boldsymbol{r}(\boldsymbol{x}_k)\|/\|\boldsymbol{r}(\boldsymbol{x}_k)\|$.

- **% stop_f_absolute** *[real,default=1e-5]* :: specifies the absolute tolerance used in the convergence test on $\|\boldsymbol{r}(\boldsymbol{x}_k)\|$.
- **% stop_f_relative** *[real,default=1e-8]* :: specifies the relative tolerance used in the convergence test on $\|\boldsymbol{r}(\boldsymbol{x}_k)\|$.
- **% stop_s** *[real,default=eps]* :: specifies the tolerance used in the convergence test on $\|\boldsymbol{s}_k\|$.

Trust region radius/regularization behaviour

Type fields

- **% relative_tr_radius** *[integer,default=0]* :: specifies whether the initial trust region radius should be scaled.
- **% initial_radius_scale** *[real,default=1.0]* :: specifies the scaling parameter for the initial trust region radius, which is only used if `relative_tr_radius = 1`.
- **% initial_radius** *[real,default=100.0]* :: specifies the initial trust-region radius, Δ.
- **% maximum_radius** *[real,default=1e8]* :: specifies the maximum size permitted for the trust-region radius.
- **% eta_successful** *[real,default=1e-8]* :: specifies the smallest value of ρ such that the step is accepted. .. success_but_reduce is also available, but not documented
- **% eta_very_successful** *[real,default=0.9]* :: specifies the value of ρ after which the trust-region radius is increased.
- **% eta_too_successful** *[real,default=2.0]* :: specifies that value of ρ after which the step is accepted, but keep the trust-region radius unchanged.
- **% radius_increase** *[real,default=2.0]* :: specifies the factor to increase the trust-region radius by.
- **% radius_reduce** *[real,default=0.5]* :: specifies the factor to decrease the trust-region radius by.
- **% tr_update_strategy** *[integer,default=1]* :: specifies the strategy used to update Δ_k. Possible values are:

1	use the usual step function.
2	use a the continuous method.

- **% reg_order** *[real,default=0.0]* :: if `type_of_method = 2`, the order of the regularization used (p in ((1.6))). If `reg_order = 0.0`, then the algorithm chooses an appropriate value of p.

Scaling options

Type fields

- **% scale** *[integer,default=1]* :: specifies how, if at all, we scale the Jacobian. We calculate a diagonal scaling matrix, `D`, as follows:
- **% scale_trim_max** *[logical,default=true]* :: specifies whether or not to trim large values of the scaling matrix, D. If `true`, $\texttt{D}_{i,i} \leftarrow min(\texttt{D}_{i,i}, \texttt{scale\_max})$.
- **% scale_max** *[real,default=1e11]* :: specifies the maximum value allowed if `scale_trim_max = true`.

- **% scale_trim_min** *[logical,default=true]* :: specifies whether or not to trim small values of the scaling matrix, D. If true, $\mathtt{D}_{i,i} \leftarrow max(\mathtt{D}_{i,i}, \mathtt{scale_max})$.
- **% scale_min** *[real,default=1e-11]* :: specifies the minimum value allowed if scale_trim_max = true.
- **% scale_require_increase** *[logical,default=false]* :: specifies whether or not to require $\mathtt{D}_{i,i}$ to increase before updating it.

Hybrid method options These options are used if model=3

Type fields

- **% hybrid_switch** *[real,default=0.1]* :: specifies the value, if model=3, at which second derivatives are used.
- **% hybrid_tol** *[real,default=2.0]* :: if model=3, specifies the value such that if $\|\boldsymbol{J}_k{}^T \boldsymbol{W} \boldsymbol{r}(\boldsymbol{x}_k)\|_2 < \mathtt{hybrid\_tol} * 0.5 \|\boldsymbol{r}(\boldsymbol{x}_k)\|^2_{\boldsymbol{W}}$ the method switches to a quasi-Newton method.
- **% hybrid_switch_its** *[integer,default=1]* :: if model=3, sets how many iterates in a row must the condition in the definition of hybrid_tol hold before a switch.

Newton-Tensor options These options are used if model=4

Type fields

- **% inner_method** *[integer,default=2]* :: if nlls_method = 4, specifies the method used to pass in the regularization parameter to the inner non-linear least squares solver. Possible values are:

1	The current regularization parameter is passed in as a base regularization parameter.
2	A larger non-linear least squares problem is explicitly formed to be solved as the subproblem.
3	The regularization is handled implicitly by calling **RALFit** recursively.

More-Sorensen options These options are used if nlls_method=3

Type fields

- **% more_sorensen_maxits** *[integer,default=3]* :: if nlls_method = 3, specifies the maximum number of iterations allowed in the More-Sorensen method.
- **% more_sorensen_maxits** :: if nlls_method = 3, specifies the maximum number of iterations allowed in the More-Sorensen method.
- **% more_sorensen_shift** *[real,default=1e-13]* :: if nlls_method = 3, specifies the shift to be used in the More-Sorensen method.
- **% more_sorensen_tiny** *[real,default=10.0*eps]* :: if nlls_method = 3, specifies the value below which numbers are considered to be essentially zero.
- **% more_sorensen_tol** *[real,default=1e-3]* :: if nlls_method = 3, specifies the tolerance to be used in the More-Sorensen method.

Other options

Type fields

- **% calculate_svd_J** *[logical,default=false]* :: specifies whether or not to calculate the singular value decomposition of J at each iteration.

- **% output_progress_vectors** *[logical,default=false]* :: if true, outputs the progress vectors `nlls_inform%resvec` and `nlls_inform%gradvec` at the end of the routine.

Internal options to help solving a regularized problem implicitly

Type fields

- **% regularization** *[integer,default=0]* :: specifies the method by which a regularized non-linear least squares problem is solved implicitly. Is designed to be used when solving the nonlinear least-squares problem recursively. Possible values are:

0	$\sigma = 0$, and an unchanged problem is solved
1	a non-linear least-squares problem of size $n + m$ is solved implicitly. Can only be used if $p = 2$.
2	a non-linear least-squares problem of size $n + 1$ is solved implicitly.

  See *Incorporating the regularization term* for details.

- **% regularization_term** *[real,default=0.0]* :: specifies the regularization weight, σ, used when implicitly solving the least-squares problem.
- **% regularization_power** *[real,default=0.0]* :: specifies the regularization index, p, used when implicitly solving the least-squares problem.

The derived data type for holding information

type nlls_inform

This is used to hold information about the progress of the algorithm.

Type fields

- **% status** *[integer]* :: gives the exit status of the subroutine. See *Warning and error messages* for details.
- **% error_message** (80) *[character]* :: holds the error message corresponding to the exit status.
- **% alloc_status** *[integer]* :: gives the status of the last attempted allocation/deallocation.
- **% bad_alloc** (80) *[character]* :: holds the name of the array that was being allocated when an error was flagged.
- **% iter** *[integer]* :: gives the total number of iterations performed.
- **% f_eval** *[integer]* :: gives the total number of evaluations of the objective function.
- **% g_eval** *[integer]* :: gives the total number of evaluations of the gradient of the objective function.
- **% h_eval** *[integer]* :: gives the total number of evaluations of the Hessian of the objective function.
- **% convergence_normf** *[integer]* :: tells us if the test on the size of $\boldsymbol{r}$ is satisfied.
- **% convergence_normf** :: that tells us if the test on the size of the gradient is satisfied.
- **% convergence_normf** :: that tells us if the test on the step length is satisfied.
- **% resvec** (iter+1) *[real]* :: if `output_progress_vectors=true` in *nlls_options*, holds the vector of residuals.

- **% resvec** :: if `output_progress_vectors=true` in *nlls_options*, holds the vector of gradients.
- **% obj** *[real]* :: holds the value of the objective function at the best estimate of the solution determined by the algorithm.
- **% norm_g** *[real]* :: holds the gradient of the objective function at the best estimate of the solution determined by the package.
- **% scaled_g** *[real]* :: holds the gradient of the objective function at the best estimate of the solution determined by the package.
- **% external_return** *[integer]* :: gives the error code that was returned by a call to an external routine.
- **% external_name** (80) *[character]* :: holds the name of the external code that flagged an error.

The workspace derived data type

type nlls_workspace
This is used to save the state of the algorithm in between calls to *nlls_iterate()*, and must be used if that subroutine is required. It's components are not designed to be accessed by the user.

1.2.5 Warning and error messages

A successful return from a subroutine in the package is indicated by `status` in *nlls_inform* having the value zero. A non-zero value is asscociated with an error message, which will be output on `error` in *nlls_inform*.

Possible values are:

-1	Maximum number of iterations reached without convergence.
-2	Error from evaluating a function/Jacobian/Hessian.
-3	Unsupported choice of model.
-4	Error return from an external routine.
-5	Unsupported choice of method.
-6	Allocation error.
-7	Maximum number of reductions of the trust radius reached.
-8	No progress being made in the solution.
-9	$\mathbf{n} > \mathbf{m}$.
-10	Unsupported trust region update strategy.
-11	Unable to valid step when solving trust region subproblem.
-12	Unsupported scaling method.
-13	Error accessing pre-allocated workspace.
-14	Unsupported value in type_of_method
-101	Unsupported model in dogleg (`nlls_method = 1`).
-201	All eigenvalues are imaginary (`nlls_method=2`).
-202	Matrix with odd number of columns sent to `max_eig` subroutine (`nlls_method=2`).
-301	`more_sorensen_max_its` is exceeded in `more_sorensen` subroutine (`nlls_method=3`).
-302	Too many shifts taken in `more_sorensen` subroutine (`nlls_method=3`).
-303	No progress being made in `more_sorensen` subroutine (`nlls_method=3`).
-401	`model = 4` selected, but `exact_second_derivatives` is set to `false`.

1.3 Description of the method used

RALFit computes a solution $\boldsymbol{x}$ to the non-linear least-squares problem

$$\min_{\boldsymbol{x}} \; F(\boldsymbol{x}) := \frac{1}{2}\|\boldsymbol{r}(\boldsymbol{x})\|_{\boldsymbol{W}}^2 + \frac{\sigma}{p}\|\boldsymbol{x}\|_2^p, \tag{1.1}$$

Here we describe the method used to solve (1.1). **RALFit** implements an iterative method that, at each iteration, calculates and returns a step $\boldsymbol{s}$ that reduces the model by an acceptable amount by solving (or approximating a solution to) a subproblem, as detailed in *Subproblem solves*.

The algorithm is iterative. At each point, $\boldsymbol{x}_k$, the algorithm builds a model of the function at the next step, $F(\boldsymbol{x}_k + \boldsymbol{s}_k)$, which we refer to as $m_k(\cdot)$. We allow either a Gauss-Newton model, a (quasi-)Newton model, or a Newton-tensor model; see *The models* for more details.

Once the model has been formed we find a candidate for the next step by solving a subitable subproblem. The quantity

$$\rho = \frac{F(\boldsymbol{x}_k) - F(\boldsymbol{x}_k + \boldsymbol{s}_k)}{m_k(\boldsymbol{x}_k) - m_k(\boldsymbol{x}_k + \boldsymbol{s}_k)} \tag{1.2}$$

is then calculated. If this is sufficiently large we accept the step, and $\boldsymbol{x}_{k+1}$ is set to $\boldsymbol{x}_k + \boldsymbol{s}_k$; if not, the parameter Δ_k is reduced and the resulting new trust-region sub-problem is solved. If the step is very successful – in that ρ is close to one – Δ_k is increased. Details are explained in *Accepting the step and updating the parameter*.

This process continues until either the residual, $\|\boldsymbol{r}(\boldsymbol{x}_k)\|_{\boldsymbol{W}}$, or a measure of the gradient, $\|\boldsymbol{J}_k{}^T\boldsymbol{W}\boldsymbol{r}(\boldsymbol{x}_k)\|_2/\|\boldsymbol{r}(\boldsymbol{x}_k)\|_{\boldsymbol{W}}$, is sufficiently small.

1.3.1 The models

A vital component of the algorithm is the choice of model employed. There are four choices available, controlled by the parameter `model` of `nlls_options`.

`model = 1` this implements the **Gauss-Newton** model. Here we replace $\boldsymbol{r}(\boldsymbol{x}_k + \boldsymbol{s})$ by its first-order Taylor approximation, $\boldsymbol{r}(\boldsymbol{x}_k) + \boldsymbol{J}_k\boldsymbol{s}$. The model is therefore given by

$$m_k^{GN}(\boldsymbol{s}) = \frac{1}{2}\|\boldsymbol{r}(\boldsymbol{x}_k) + \boldsymbol{J}_k\boldsymbol{s}\|_{\boldsymbol{W}}^2. \tag{1.3}$$

`model = 2` this implements the **Newton** model. Here, instead of approximating the residual, $\boldsymbol{r}(\cdot)$, we take as our model the second-order Taylor approximation of the function, $F(\boldsymbol{x}_{k+1})$. Namely, we use

where $\boldsymbol{g}_k = \boldsymbol{J}_k{}^T\boldsymbol{W}\boldsymbol{r}(\boldsymbol{x}_k)$ and $\boldsymbol{H}_k = \sum_{i=1}^m r_i(\boldsymbol{x}_k)\boldsymbol{W}\nabla^2 r_i(\boldsymbol{x}_k)$. Note that $m_k^N(\boldsymbol{s}) = m_k^{GN}(\boldsymbol{s}) + \frac{1}{2}\boldsymbol{s}^T\boldsymbol{H}_k\boldsymbol{s}$.

If the second derivatives of $\boldsymbol{r}(\cdot)$ are not available (i.e., the option `exact_second_derivatives` is set to `false`, then the method approximates the matrix $\boldsymbol{H}_k$; see *Approximating the Hessian*.

`model = 3` This implements a **hybrid** model. In practice the Gauss-Newton model tends to work well far away from the solution, whereas Newton performs better once we are near to the minimum (particularly if the residual is large at the solution). This option will try to switch between these two models, picking the model that is most appropriate for the step. In particular, we start using $m_k^{GN}(\cdot)$, and switch to $m_k^N(\cdot)$ if $\|\boldsymbol{g}_k\|_2 \leq$ `hybrid_tol`$\frac{1}{2}\|\boldsymbol{r}(\boldsymbol{x}_k)\|_{\boldsymbol{W}}^2$ for more than `hybrid_switch_its` iterations in a row. If, in subsequent iterations, we fail to get a decrease in the function value, then the algorithm interprets this as being not sufficiently close to the solution, and thus switches back to using the Gauss-Newton model.

The exact method used is described below:

if `use_second_derivatives` *// previous step used Newton model*
  if $\|\mathtt{g}_{k+1}\| > \|\mathtt{g}_k\|$
    `use_second_derivatives = false` *// Switch back to Gauss-Newton*
    $\mathtt{Hf}_{temp} = \mathtt{Hf}_k$, $\mathtt{Hf}_k = 0$ *// Copy Hessian back to temp array*
  endif
else
  if $\|\mathtt{g}_{k+1}\|/\mathtt{normF}_{k+1} <$ `hybrid_tol`
    `hybrid_count = hybrid_count + 1` *// Update the no of successive failures*
    if `hybrid_count = hybrid_count_switch_its`
      `use_second_derivatives = true`
      `hybrid_count = 0`
      $\mathtt{Hf}_{temp} = \mathtt{Hf}_k$*// Copy approximate Hessian back*
    end if
  end if
end if

`model = 4` this implements a **Newton-tensor** model. This uses a second order Taylor approximation to the residual, namely

$$r_i(\boldsymbol{x}_k + \boldsymbol{s}) \approx (\boldsymbol{t}_k(\boldsymbol{s}))_i := r_i(\boldsymbol{x}_k) + (\boldsymbol{J}_k)_i\boldsymbol{s} + \frac{1}{2}\boldsymbol{s}^T B_{ik}\boldsymbol{s},$$

where $(\boldsymbol{J}_k)_i$ is the ith row of $\boldsymbol{J}_k$, and B_{ik} is $\nabla^2 r_i(\boldsymbol{x}_k)$. We use this to define our model

$$m_k^{NT}(\boldsymbol{s}) = \frac{1}{2}\|\boldsymbol{t}_k(\boldsymbol{s})\|^2_{\boldsymbol{W}}. \tag{1.4}$$

1.3.2 Approximating the Hessian

If the exact Hessian is not available, we approximate it using the method of Dennis, Gay, and Welsch[4]. The method used is given as follows:

function $\mathtt{Hf}_{k+1} = \mathtt{rank\_one\_update}(\mathtt{d}, \mathtt{g}_k, \mathtt{g}_{k+1}, \mathtt{r}_{k+1}, \mathtt{J}_k, \mathtt{Hf}_k)$

$\mathtt{y} = \mathtt{g}_k - \mathtt{g}_{k+1}$

$\widehat{\mathtt{y}} = {\mathtt{J}_k}^T\mathtt{r}_{k+1} - \mathtt{g}_{k+1}$

$\widehat{\mathtt{Hf}_k} = \min\left(1, \frac{|\mathtt{d}^T\widehat{\mathtt{y}}|}{|\mathtt{d}^T\mathtt{Hf}_k\mathtt{d}|}\right)\mathtt{Hf}_k$

$\mathtt{Hf}_{k+1} = \widehat{\mathtt{Hf}_k} + \left((\widehat{\mathtt{y}}_{k+1} - \mathtt{Hf}_k\mathtt{d})^T\mathtt{d}\right)/\mathtt{y}^T\mathtt{d}$

It is sometimes the case that this approximation becomes corrupted, and the algorithm may not recover from this. To guard against this, if `model = 3` in *`nlls_options`* and we are using this approximation to the Hessian in our (quasi-Newton) model, we test against the Gauss-Newton model if the first step is unsuccessful. If the Gauss-Newton step would have been successful, we discard the approximate Hessian information, and recompute the step using Gauss-Newton.

In the case where `model=3`, the approximation to the Hessian is updated at each step whether or not it is needed for the current calcuation.

[4] Nocedal, J., & Wright, S. (2006). Numerical optimization. Springer Science & Business Media.

1.3.3 Subproblem solves

The main algorithm calls a number of subroutines. The most vital is the subroutine `calculate_step`, which finds a step that minimizes the model chosen, subject to a globalization strategy. The algorithm supports the use of two such strategies: using a trust-region, and regularization. If Gauss-Newton, (quasi-)Newton, or a hybrid method is used (`model = 1,2,3` in *`nlls_options`*), then the model function is quadratic, and the methods available to solve the subproblem are described in *The trust region method* and *Regularization*. If the Newton-Tensor model is selected (`model = 4` in *`nlls_options`*), then this model is not quadratic, and the methods available are described in *Newton-Tensor subproblem*.

Note that, when calculating the step, if the initial regularization parameter σ in (1.1) is non-zero, then we must modify ${\mathtt{J}_k}^T\mathtt{J}_k$ to take into account the Jacobian of the modified least squares problem being solved. Practically, this amounts to making the change

$$ {\mathtt{J}_k}^T\mathtt{J}_k = {\mathtt{J}_k}^T\mathtt{J}_k + \begin{cases} \sigma I & \text{if } p = 2 \\ \frac{\sigma p}{2}\|\boldsymbol{x}_k\|^{p-4}\boldsymbol{x}_k{\boldsymbol{x}_k}^T & \text{otherwise} \end{cases}. $$

The trust region method

If `model = 1, 2,` or `3`, and `type_of_method=1`, then we solve the subproblem

$$ \boldsymbol{s}_k = \arg\min_{\boldsymbol{s}} \; m_k(\boldsymbol{s}) \quad \text{s.t.} \quad \|\boldsymbol{s}\|_B \leq \Delta_k, \tag{1.5} $$

and we take as our next step the minimum of the model within some radius of the current point. The method used to solve this is dependent on the control parameter optionsnlls_method. The algorithms called for each of the options are listed below:

`nlls_method = 1` approximates the solution to (1.5) by using Powell's dogleg method. This takes as the step a linear combination of the Gauss-Newton step and the steepest descent step, and the method used is described here:

function $\mathtt{dogleg}(\mathtt{J},\mathtt{r},\mathtt{Hf},\mathtt{g},\Delta)$
$\alpha = \|\mathtt{g}\|^2/\|\mathtt{J}*\mathtt{g}\|^2$
$\mathtt{d}_{\mathrm{sd}} = \alpha\,\mathtt{g}$
solve $\mathtt{d}_{\mathrm{gn}} = \arg\min_{\mathtt{x}} \|\mathtt{J}\mathtt{x}-\mathtt{r}\|_2$
if $\|\mathtt{d}_{\mathrm{gn}}\| \leq \Delta$**then**
    $\mathtt{d} = \mathtt{d}_{\mathrm{gn}}$
else if $\|\alpha\,\mathtt{d}_{\mathrm{sd}}\| \geq \Delta$
    $\mathtt{d} = (\Delta/\|\mathtt{d}_{\mathrm{sd}}\|)\mathtt{d}_{\mathrm{sd}}$
else
    $\mathtt{d} = \alpha\,\mathtt{d}_{\mathrm{sd}} + \beta\,(\mathtt{d}_{\mathrm{gn}} - \alpha\mathtt{d}_{\mathrm{sd}})$, where β is chosen such that $\|\mathtt{d}\| = \Delta$
end if

`nlls_method = 2` solves the trust region subproblem using the trust region solver of Adachi, Iwata, Nakatsukasa, and Takeda. This reformulates the problem (1.5) as a generalized eigenvalue problem, and solves that. See[1] for more details.

[1] Adachi, Satoru and Iwata, Satoru and Nakatsukasa, Yuji and Takeda, Akiko (2015). Solving the trust region subproblem by a generalized eigenvalue problem. Technical report, Mathematical Engineering, The University of Tokyo.

nlls_method = 3 this solves (1.5) using a variant of the More-Sorensen method. In particular, we implement Algorithm 7.3.6 in Trust Region Methods by Conn, Gould and Toint[2].

nlls_method = 4 this solves (1.5) by first converting the problem into the form

$$\min_{\boldsymbol{p}} \boldsymbol{w}^T\boldsymbol{p} + \frac{1}{2}\boldsymbol{p}^T\boldsymbol{D}\boldsymbol{p} \quad \text{s.t.} \quad \|\boldsymbol{p}\| \leq \Delta,$$

where $\boldsymbol{D}$ is a diagonal matrix. We do this by performing an eigen-decomposition of the Hessian in the model. Then, we call the Galahad routine DTRS; see the Galahad[3] documentation for further details.

Regularization

If `model = 1, 2,` or `3`, and `type_of_method=2`, then the next step is taken to be the minimum of the model with a regularization term added:

$$\boldsymbol{s}_k = \arg\min_{\boldsymbol{s}} \; m_k(\boldsymbol{s}) + \frac{1}{\Delta_k} \cdot \frac{1}{p}\|\boldsymbol{s}\|_B^p, \tag{1.6}$$

At present, only one method of solving this subproblem is supported:

nlls_method = 4: this solves (1.6) by first converting the problem into the form

$$\min_{\boldsymbol{p}} \boldsymbol{w}^T\boldsymbol{p} + \frac{1}{2}\boldsymbol{p}^T\boldsymbol{D}\boldsymbol{p} + \frac{1}{p}\|\boldsymbol{p}\|_2^p,$$

where $\boldsymbol{D}$ is a diagonal matrix. We do this by performing an eigen-decomposition of the Hessian in the model. Then, we call the Galahad routine DRQS; see the Galahad[3] documentation for further details.

Newton-Tensor subproblem

If `model=4`, then the non-quadratic Newton-Tensor model is used. As such, none of the established subproblem solvers described in *The trust region method* or *Regularization* can be used.

If we use regularization (with $p = 2$), then the subproblem we need to solve is of the form

$$\min_{\boldsymbol{s}} \frac{1}{2}\sum_{i=1}^{m} W_{ii}(\boldsymbol{t}_k(\boldsymbol{s}))_i{}^2 + \frac{1}{2\Delta_k}\|\boldsymbol{s}\|_2^2 \tag{1.7}$$

Note that (1.7) is a sum-of-squares, and as such can be solved by calling `nlls_solve()` recursively. We support two options:

inner_method = 1 if this option is selected, then `nlls_solve()` is called to solve (1.4) directly. The current regularization parameter of the 'outer' method is used as a base regularization in the 'inner' method, so that the (quadratic) subproblem being solved in the 'inner' call is of the form

$$\min_{\boldsymbol{s}} \; m_k(\boldsymbol{s}) + \frac{1}{2}\left(\frac{1}{\Delta_k} + \frac{1}{\delta_k}\right)\|\boldsymbol{s}\|_B^2,$$

where $m_k(\boldsymbol{s})$ is a quadratic model of (1.4), Δ_k is the (fixed) regularization parameter of the outer iteration, and δ_k the regularization parameter of the inner iteration, which is free to be updated as required by the method.

[2] Conn, A. R., Gould, N. I., & Toint, P. L. (2000). Trust region methods. SIAM.

[3] Gould, N. I., Orban, D., & Toint, P. L. (2003). GALAHAD, a library of thread-safe Fortran 90 packages for large-scale nonlinear optimization. ACM Transactions on Mathematical Software (TOMS), 29(4), 353-372.

inner_method = 2 in this case we use *nlls_solve()* to solve the regularized model (1.7) directly. The number of parameters for this subproblem is $n + m$. Specifically, we have a problem of the form

$$\min_{\boldsymbol{s}} \frac{1}{2}\|\widehat{\boldsymbol{r}}(\boldsymbol{s})\|^2_{\boldsymbol{W}}, \quad \text{where } (\widehat{\boldsymbol{r}}(\boldsymbol{s}))_i = \begin{cases} (\boldsymbol{t}_k(\boldsymbol{s}))_i & 1 \leq i \leq m \\ \frac{1}{\sqrt{\Delta_k}} s_i & m+1 \leq i \leq n+m \end{cases}.$$

This subproblem can then be solved using any of the methods described in *The trust region method* or *Regularization*.

`inner_method = 3`

In this case, *nlls_solve()* is called recursively with the inbuilt feature of solving a regularized problem, as described in *Incorporating the regularization term*

1.3.4 Accepting the step and updating the parameter

Once a step has been suggested, we must decide whether or not to accept the step, and whether the trust region radius or regularization parameter, as appropriate, should grow, shrink, or remain the same.

These decisions are made with reference to the parameter, ρ (1.2), which measures the ratio of the actual reduction in the model to the predicted reduction in the model. If this is larger than `eta_successful` in *nlls_options*, then the step is accepted.

The value of Δ_k then needs to be updated, if appropriate. The package supports two options:

tr_update_strategy = 1 a step-function is used to decide whether or not to increase or decrease Δ_k, as described here:

if $\rho \leq$ `eta_success_but_reduce` **then**
  $\Delta =$ `radius_reduce` $* \Delta$ *// reduce Δ*
else if $\rho \leq$ `eta_very_successful`
  $\Delta = \Delta$ *// Δ stays unchanged*
else if $\rho \leq$ `eta_too_successful`
  $\Delta =$ `radius_increase` $* \Delta$ *// increase Δ*
else if $\rho >$ `eta_too_successful`
  $\Delta = \Delta$ *// too successful: accept step, but don't change Δ*
end if

tr_update_strategy = 2 a continuous function is used to make the decision[5], as described below. On the first call, the parameter ν is set to 2.0.

if $\rho \geq$ `eta_too_successful`
  $\Delta = \Delta$ *// Δ stays unchanged*
else if $\rho >$ `eta_successful`
  $\Delta = \Delta * \min\left(\texttt{radius\_increase}, 1 - \left((\texttt{radius\_increase} - 1) * ((1 - 2 * \rho)^3)\right)\right)$
  $\nu =$ `radius_reduce`
else if $\rho \leq$ `eta_successful`
  $\Delta = \nu * \Delta$
  $\nu = 0.5 * \nu$
end if

[5] Nielsen, Hans Bruun (1999). Damping parameter in Marquadt's MEthod. Technical report TR IMM-REP-1999-05, Department of Mathematical Modelling, Technical University of Denmark (http://www2.imm.dtu.dk/documents/ftp/tr99/tr05_99.pdf)

1.3.5 Incorporating the regularization term

If a non-zero regularization term is required in (1.1), then this is handled by transforming the problem internally into a new non-linear least-squares problem. The formulation used will depend on the value of `regularization` in *nlls_options*.

`regularization = 1` This is only supported if $\mathbf{p} = \mathbf{2}$. We solve a least squares problem with n additional degrees of freedom. The new function, $\widehat{\boldsymbol{r}} : \mathbb{R}^n \to \mathbb{R}^{m+n}$, is defined as

$$\widehat{\boldsymbol{r}}_i(\boldsymbol{x}) = \begin{cases} \boldsymbol{r}_i(\boldsymbol{x}) & \text{for } i = 1, \ldots, m \\ \sqrt{\sigma}[\boldsymbol{x}]_j & \text{for } i = m + j,\ j = 1, \ldots, n \end{cases}$$

where $[\boldsymbol{x}]_j$ denotes the jth component of $\boldsymbol{x}$.

This problem is now in the format of a standard non-linear least-squares problem. In addition to the function values, the we also need a Jacobian and some more information about the Hessian. For our modified function, the Jacobian is

$$\widehat{\boldsymbol{J}}(\boldsymbol{x}) = \begin{bmatrix} \boldsymbol{J}(\boldsymbol{x}) \\ \sqrt{\sigma} I \end{bmatrix},$$

and the other function that needs to be supplied is given by

$$\widehat{\boldsymbol{H}}_k(\boldsymbol{x}) = \sum_{i=1}^{n+m} \widehat{\boldsymbol{r}}_i(\boldsymbol{x}) \nabla^2 \widehat{\boldsymbol{r}}_i(\boldsymbol{x}) = \sum_{i=1}^{n} \boldsymbol{r}_i(\boldsymbol{x}) \nabla^2 \boldsymbol{r}_i(\boldsymbol{x}) = \boldsymbol{H}_k(\boldsymbol{x}).$$

We solve these problems implicitly by modifing the code so that the user does not need do any additional work. We can simply note that

$$\|\widehat{\boldsymbol{r}}(\boldsymbol{x})\|^2 = \|\boldsymbol{r}(\boldsymbol{x})\|^2 + \sigma \|\boldsymbol{x}\|^2,$$

$$\widehat{\boldsymbol{J}}^T \widehat{\boldsymbol{r}} = \boldsymbol{J}^T \boldsymbol{r} + \sigma \boldsymbol{x},$$

and that

$$\widehat{\boldsymbol{J}}^T \widehat{\boldsymbol{J}} = \boldsymbol{J}^T \boldsymbol{J} + \sigma I.$$

We also need to update the value of the model. Since the Hessian vanishes, we only need to be concerned with the Gauss-Newton model. We have that

$$\begin{aligned} \widehat{m}_k^{GN}(\boldsymbol{s}) &= \frac{1}{2} \|\widehat{\boldsymbol{r}}(\boldsymbol{x}_k) + \widehat{\boldsymbol{J}}_k \boldsymbol{s}\|^2 \\ &= \frac{1}{2} \left(\widehat{\boldsymbol{r}}^T \widehat{\boldsymbol{r}} + 2\boldsymbol{s}^T \widehat{\boldsymbol{J}}_k^T \widehat{\boldsymbol{r}} + \boldsymbol{s}^T \widehat{\boldsymbol{J}}_k^T \widehat{\boldsymbol{J}}_k \boldsymbol{s} \right) \\ &= \frac{1}{2} \left(\boldsymbol{r}^T \boldsymbol{r} + \sigma \boldsymbol{x}^T \boldsymbol{x} + 2(\boldsymbol{s}^T {\boldsymbol{J}_k}^T \boldsymbol{r} + \sigma \boldsymbol{s}^T \boldsymbol{x}) + \boldsymbol{s}^T {\boldsymbol{J}_k}^T \boldsymbol{J}_k \boldsymbol{s} + \sigma \boldsymbol{s}^T \boldsymbol{s} \right) \\ &= m_k^{GN}(\boldsymbol{s}) + \frac{1}{2} \sigma (\boldsymbol{x}^T \boldsymbol{x} + 2\boldsymbol{s}^T \boldsymbol{x} + \boldsymbol{s}^T \boldsymbol{s}) \\ &= m_k^{GN}(\boldsymbol{s}) + \frac{\sigma}{2} \|\boldsymbol{x} + \boldsymbol{s}\|^2 \end{aligned}$$

`regularization=2`

We solve a non-linear least-squares problem with one additional degree of freedom.

Since the term $\frac{\sigma}{p} \|\boldsymbol{x}\|_2^p$ is non-negative, we can write

$$F_\sigma(\boldsymbol{x}) = \frac{1}{2} \left(\|\boldsymbol{r}(\boldsymbol{x})\|^2 + \left(\left(\frac{2\sigma}{p} \right)^{1/2} \|\boldsymbol{x}\|^{p/2} \right)^2 \right),$$

thereby defining a new non-linear least squares problem involving the function $\boldsymbol{r} : \mathbb{R}^n \to \mathbb{R}^{m+1}$ such that

$$\bar{r}_i(\boldsymbol{x}) = \begin{cases} r_i(\boldsymbol{x}) & 1 \leq i \leq m \\ \frac{2\sigma}{p}\|\boldsymbol{x}\|^{p/2} & i = m+1 \end{cases}.$$

The Jacobian for this new function is given by

$$\bar{\boldsymbol{J}}(\boldsymbol{x}) = \begin{bmatrix} \boldsymbol{J}(\boldsymbol{x}) \\ \left(\frac{\sigma p}{2}\right)^{1/2} \|\boldsymbol{x}\|^{(p-4)/2}\boldsymbol{x}^T \end{bmatrix},$$

and we get that

$$\nabla^2 \bar{r}_{m+1} = \left(\frac{\sigma p}{2}\right)^{1/2} \|\boldsymbol{x}\|^{(p-4)/2}\left(I + \frac{\boldsymbol{x}\boldsymbol{x}^T}{\|\boldsymbol{x}\|^2}\right).$$

As for the case where `regularization=1`, we simply need to update quantities in our non-linear least squares code to solve this problem, and the changes needed in this case are

$$\|\bar{\boldsymbol{r}}(\boldsymbol{x})\|^2 = \|\boldsymbol{r}(\boldsymbol{x})\|^2 + \frac{2\sigma}{p}\|\boldsymbol{x}\|^p,$$

$$\bar{\boldsymbol{J}}^T\bar{\boldsymbol{r}} = \boldsymbol{J}^T\boldsymbol{r} + \sigma\|\boldsymbol{x}\|^{p-2}\boldsymbol{x},$$

$$\bar{\boldsymbol{J}}^T\bar{\boldsymbol{J}} = \boldsymbol{J}^T\boldsymbol{J} + \frac{\sigma p}{2}\|\boldsymbol{x}\|^{p-4}\boldsymbol{x}\boldsymbol{x}^T,$$

$$\sum_{i=1}^{m+1} \bar{r}_i(\boldsymbol{x})\bar{\boldsymbol{H}}_i(\boldsymbol{x}) = \sigma\|\boldsymbol{x}\|^{p-4}\left(\|\boldsymbol{x}\|^2 I + \boldsymbol{x}\boldsymbol{x}^T\right) + \sum_{i=1}^{m} r_i(\boldsymbol{x})\boldsymbol{H}_i(\boldsymbol{x})$$

We also need to update the model. Here we must consider the Gauss-Newton and Newton models separately.

$$\begin{aligned}
\bar{m}_k^{GN}(\boldsymbol{s}) &= \frac{1}{2}\|\bar{\boldsymbol{r}}(\boldsymbol{x}_k) + \bar{\boldsymbol{J}}_k\boldsymbol{s}\|^2 \\
&= \frac{1}{2}\left(\bar{\boldsymbol{r}}^T\bar{\boldsymbol{r}} + 2\boldsymbol{s}^T\bar{\boldsymbol{J}}_k^T\bar{\boldsymbol{r}} + \boldsymbol{s}^T\bar{\boldsymbol{J}}_k^T\bar{\boldsymbol{J}}_k\boldsymbol{s}\right) \\
&= \frac{1}{2}\left(\boldsymbol{r}^T\boldsymbol{r} + \frac{2\sigma}{p}\|\boldsymbol{x}\|^p + 2(\boldsymbol{s}^T{\boldsymbol{J}_k}^T\boldsymbol{r} + \sigma\|\boldsymbol{x}\|^{p-2}\boldsymbol{s}^T\boldsymbol{x}) + \boldsymbol{s}^T{\boldsymbol{J}_k}^T\boldsymbol{J}_k\boldsymbol{s} + \frac{\sigma p}{2}\|\boldsymbol{x}\|^{p-4}(\boldsymbol{s}^T\boldsymbol{x})^2\right) \\
&= m_k^{GN}(\boldsymbol{s}) + \sigma\left(\frac{1}{p}\|\boldsymbol{x}\|^p + \|\boldsymbol{x}\|^{p-2}\boldsymbol{s}^T\boldsymbol{x} + \frac{p}{4}\|\boldsymbol{x}\|^{p-4}(\boldsymbol{s}^T\boldsymbol{x})^2\right).
\end{aligned}$$

If we use a Newton model then

$$\begin{aligned}
\bar{m}_k^{N}(\boldsymbol{s}) &= \bar{m}_k^{GN}(\boldsymbol{s}) + \frac{1}{2}\boldsymbol{s}^T\bar{\boldsymbol{H}}_k\boldsymbol{s} \\
&= \bar{m}_k^{GN}(\boldsymbol{s}) + \frac{1}{2}\boldsymbol{s}^T\left(\boldsymbol{H}_k + \sigma\|\boldsymbol{x}\|^{p-2}\left(I + \frac{\boldsymbol{x}\boldsymbol{x}^T}{\|\boldsymbol{x}\|^2}\right)\right)\boldsymbol{s} \\
&= \bar{m}_k^{GN}(\boldsymbol{s}) + \frac{1}{2}\boldsymbol{s}^T\boldsymbol{H}_k\boldsymbol{s} + \frac{\sigma}{2}\|\boldsymbol{x}\|^{p-4}\boldsymbol{s}^T\left(\boldsymbol{x}^T\boldsymbol{x}I + \boldsymbol{x}\boldsymbol{x}^T\right)\boldsymbol{s} \\
&= \bar{m}_k^{GN}(\boldsymbol{s}) + \frac{1}{2}\boldsymbol{s}^T\boldsymbol{H}_k\boldsymbol{s} + \frac{\sigma}{2}\|\boldsymbol{x}\|^{p-4}\left((\boldsymbol{x}^T\boldsymbol{x})(\boldsymbol{s}^T\boldsymbol{s}) + (\boldsymbol{x}^T\boldsymbol{s})^2\right)
\end{aligned}$$

1.4 Examples

Consider fitting the function $y(t) = x_1 e^{x_2 t}$ to data $(\boldsymbol{t}, \boldsymbol{y})$ using a non-linear least squares fit.

The residual function is given by

$$r_i(\boldsymbol{x}) = x_1 e^{x_2 t_i} - y_i.$$

We can calculate the Jacobian and Hessian of the residual as

$$\nabla r_i(\boldsymbol{x}) = \begin{pmatrix} e^{x_2 t_i} & t_i x_1 e^{x_2 t_i} \end{pmatrix},$$

$$\nabla^2 r_i(\boldsymbol{x}) = \begin{pmatrix} 0 & t_i e^{x_2 t_i} \\ t_i e^{x_2 t_i} & x_1 t_i^2 e^{x_2 t_i} \end{pmatrix}.$$

For some data points, y_i, t_i, $(i = 1, \ldots, m)$ the user must return

$$\boldsymbol{r}(\boldsymbol{x}) = \begin{bmatrix} r_1(\boldsymbol{x}) \\ \vdots \\ r_m(\boldsymbol{x}) \end{bmatrix}, \quad \boldsymbol{J}(\boldsymbol{x}) = \begin{bmatrix} \nabla r_1(\boldsymbol{x}) \\ \vdots \\ \nabla r_m(\boldsymbol{x}) \end{bmatrix}, \quad Hf(\boldsymbol{x}) = \sum_{i=1}^{m} (\boldsymbol{r})_i \nabla^2 r_i(\boldsymbol{x}),$$

where, in the case of the Hessian, $(\boldsymbol{r})_i$ is the ith component of a residual vector passed to the user.

i	1	2	3	4	5
t_i	1	2	4	5	8
y_i	3	4	6	11	20

and initial guess $\boldsymbol{x} = (2.5, 0.25)$, the following code performs the fit (with no weightings, i.e., $\boldsymbol{W} = \boldsymbol{I}$).

```
! examples/Fortran/nlls_example.f90
!
! Attempts to fit the model y_i = x_1 e^(x_2 t_i)
! For parameters x_1 and x_2, and input data (t_i, y_i)
module fndef_example
   use ral_nlls_double, only : params_base_type
   implicit none

   integer, parameter :: wp = kind(0d0)

   type, extends(params_base_type) :: params_type
      real(wp), dimension(:), allocatable :: t ! The m data points t_i
      real(wp), dimension(:), allocatable :: y ! The m data points y_i
   end type

contains
   ! Calculate r_i(x; t_i, y_i) = x_1 e^(x_2 * t_i) - y_i
   subroutine eval_r(status, n, m, x, r, params)
      integer, intent(out) :: status
      integer, intent(in) :: n
      integer, intent(in) :: m
      real(wp), dimension(*), intent(in) :: x
      real(wp), dimension(*), intent(out) :: r
      class(params_base_type), intent(inout) :: params

      real(wp) :: x1, x2

      x1 = x(1)
      x2 = x(2)
      select type(params)
      type is(params_type)
         r(1:m) = x1 * exp(x2*params%t(:)) - params%y(:)
```

```
      end select

      status = 0 ! Success
   end subroutine eval_r
   ! Calculate:
   ! J_i1 = e^(x_2 * t_i)
   ! J_i2 = t_i x_1 e^(x_2 * t_i)
   subroutine eval_J(status, n, m, x, J, params)
      integer, intent(out) :: status
      integer, intent(in) :: n
      integer, intent(in) :: m
      real(wp), dimension(*), intent(in) :: x
      real(wp), dimension(*), intent(out) :: J
      class(params_base_type), intent(inout) :: params

      real(wp) :: x1, x2

      x1 = x(1)
      x2 = x(2)
      select type(params)
      type is(params_type)
         J(  1:  m) = exp(x2*params%t(1:m))                    ! J_i1
         J(m+1:2*m) = params%t(1:m) * x1 * exp(x2*params%t(1:m))! J_i2
      end select

      status = 0 ! Success
   end subroutine eval_J
   ! Calculate
   ! HF = sum_i r_i H_i
   ! Where H_i = [ 0                t_i e^(x_2 t_i)    ]
   !             [ t_i e^(x_2 t_i)  t_i^2 x_1 e^(x_2 t_i)  ]
   subroutine eval_HF(status, n, m, x, r, HF, params)
      integer, intent(out) :: status
      integer, intent(in) :: n
      integer, intent(in) :: m
      real(wp), dimension(*), intent(in) :: x
      real(wp), dimension(*), intent(in) :: r
      real(wp), dimension(*), intent(out) :: HF
      class(params_base_type), intent(inout) :: params

      real(wp) :: x1, x2

      x1 = x(1)
      x2 = x(2)
      select type(params)
      type is(params_type)
         HF(    1) = sum(r(1:m) * 0)                                      ! H_11
         HF(    2) = sum(r(1:m) * params%t(1:m) * exp(x2*params%t(1:m)))  ! H_21
         HF(1*n+1) = HF(2)                                                ! H_12
         HF(1*n+2) = sum(r(1:m) * (params%t(1:m)**2) * x1 * exp(x2*params%t(1:m)))! H_
↪22
      end select

      status = 0 ! Success
   end subroutine eval_HF
end module fndef_example

program nlls_example
```

```
   use ral_nlls_double
   use fndef_example
   implicit none

   type(nlls_options) :: options
   type(nlls_inform) :: inform

   integer :: m,n
   real(wp), allocatable :: x(:)
   type(params_type) :: params

   ! Data to be fitted
   m = 5
   allocate(params%t(m), params%y(m))
   params%t(:) = (/ 1.0, 2.0, 4.0,  5.0,  8.0 /)
   params%y(:) = (/ 3.0, 4.0, 6.0, 11.0, 20.0 /)

   ! Call fitting routine
   n = 2
   allocate(x(n))
   x = (/ 2.5, 0.25 /) ! Initial guess
   call nlls_solve(n, m, x, eval_r, eval_J, eval_HF, params, options, inform)
   if(inform%status.ne.0) then
      print *, "ral_nlls() returned with error flag ", inform%status
      stop
   endif

   ! Print result
   print *, "Found a local optimum at x = ", x
   print *, "Took ", inform%iter, " iterations"
   print *, "     ", inform%f_eval, " function evaluations"
   print *, "     ", inform%g_eval, " gradient evaluations"
   print *, "     ", inform%h_eval, " hessian evaluations"
end program nlls_example
```

This returns the following output:

1.5 Indices and tables

- genindex
- search

Index

E

eval_Hf() (fortran subroutine), **7**
eval_HP() (fortran subroutine), **8**
eval_J() (fortran subroutine), **6**
eval_r() (fortran subroutine), **6**

N

nlls_inform (fortran type), **12**
nlls_iterate() (fortran subroutine), **5**
nlls_options (fortran type), **8**
nlls_solve() (fortran subroutine), **4**
nlls_workspace (fortran type), **13**