Package: mize (via r-universe)

October 31, 2026

Type Package

Title Unconstrained Numerical Optimization Algorithms

Version 0.3.0

Description Optimization algorithms implemented in R, including conjugate gradient (CG), Broyden-Fletcher-Goldfarb-Shanno (BFGS) and the limited memory BFGS (L-BFGS) methods. Most internal parameters can be set through the call interface. The solvers hold up quite well for higher-dimensional problems.

License BSD 2-clause License + file LICENSE

Encoding UTF-8

Imports methods

Suggests testthat, knitr, rmarkdown, covr, pkgdown

Config/testthat/edition 3

URL https://jlmelville.github.io/mize/, https://github.com/jlmelville/mize

BugReports https://github.com/jlmelville/mize/issues

VignetteBuilder knitr

Config/Needs/website pkgdown

Config/roxygen2/version 8.1.1

NeedsCompilation no

Author James Melville [aut, cre]

Maintainer James Melville <jlmelville@gmail.com>

Repository https://cran.r-universe.dev

Date/Publication 2026-10-31 12:49:29 UTC

RemoteUrl https://github.com/cran/mize

RemoteRef HEAD

RemoteSha 4eb5cec41ff248db79f951e988d52269dc0ec0b3

Contents

check_mize_convergence 2
check_mize_gradient 3
make_mize 5
mize 10
mize_init 24
mize_step 27
mize_step_summary 29

Index **32**

`check_mize_convergence` *Check Optimization Convergence*

Description

Updates the optimizer with information about convergence or termination, signaling if the optimization process should stop.

Usage

```
check_mize_convergence(mize_step_info)
```

Arguments

`mize_step_info`	Step info for this iteration, created by `mize_step_summary()`

Details

On returning from this function, the updated value of `opt` will contain:

- A boolean value `is_terminated` which is `TRUE` if termination has been indicated, and `FALSE` otherwise.
- A boolean value `converged` which is `TRUE` only if termination was caused by a convergence tolerance.
- A string `status` classifying the termination as `"converged"`, `"budget_exhausted"`, `"failed"`, or `"terminated"`.
- A string `message` describing the termination.
- A list `terminate` if `is_terminated` is `TRUE`. This contains two items: `what`, a short string describing what caused the termination, and `val`, the value of the termination criterion that caused termination. This list will not be present if `is_terminated` is `FALSE`.

Convergence criteria are checked here after `mize_step_summary()` has gathered the observations they require. To set these criteria, use `make_mize()` or `mize_init()`. In a stateful loop, first retain `mize_step_info$opt`; it may already be terminal because a requested observation exhausted a hard callback budget. Call this function only while that optimizer remains active.

Value

opt updated with convergence and termination data. See 'Details'.

Examples

```
rb_fg <- list(
  fn = function(x) {
    100 * (x[2] - x[1] * x[1])^2 + (1 - x[1])^2
  },
  gr = function(x) {
    c(
      -400 * x[1] * (x[2] - x[1] * x[1]) - 2 * (1 - x[1]),
      200 * (x[2] - x[1] * x[1])
    )
  }
)
rb0 <- c(-1.2, 1)

opt <- make_mize(method = "BFGS", par = rb0, fg = rb_fg, max_iter = 30)
mize_res <- mize_step(opt = opt, par = rb0, fg = rb_fg)
step_info <- mize_step_summary(mize_res$opt, mize_res$par, rb_fg, rb0)
opt <- step_info$opt
if (!opt$is_terminated) {
  opt <- check_mize_convergence(step_info)
}
```

---

check_mize_gradient *Check Gradient Consistency*

Description

Compares an analytic gradient with a finite-difference approximation of the objective function at a parameter vector.

Usage

```
check_mize_gradient(
  fg,
  par,
  method = c("central", "forward"),
  rel_eps = NULL,
  abs_eps = 0,
  ...
)
```

Arguments

fg	Function and gradient list. See the documentation of mize().
par	Parameter vector where the gradient should be checked.
method	Finite-difference method: either "central" or "forward".
rel_eps	Relative finite-difference step. If NULL, a method-specific default is used.
abs_eps	Non-negative scalar added to each finite-difference step.
...	Additional arguments passed to fgfn, fggr, and fg$fg when present.

Details

The fg argument uses the same list shape as mize(): it must contain fn and gr functions. If it also contains an fg function, the combined function-and-gradient result is checked for consistency with the separate fn and gr functions at par.

The finite-difference step for each coordinate is abs_eps + rel_eps * pmax(abs(par), 1). Central differences are usually more accurate and use twice as many function evaluations as forward differences. Each required perturbed coordinate must be finite and representably distinct from its original value. An unusable perturbation raises an error identifying the coordinate; the checker does not enlarge the requested step. Callback results must have the same scalar or vector shape as in mize() and must contain only finite values.

Value

A list with components:

- par: Parameter vector checked.
- method: Finite-difference method used.
- step: Per-coordinate finite-difference steps.
- fn: Function value at par.
- gr: Gradient returned by fg$gr(par).
- gr_fd: Finite-difference gradient approximation.
- error: Difference gr - gr_fd.
- abs_error: Absolute coordinate-wise errors.
- rel_error: Relative coordinate-wise errors, scaled by the larger coordinate magnitude of gr and gr_fd.
- max_abs_error: Maximum absolute error.
- max_rel_error: Maximum relative error.
- worst_abs: One-row data frame for the coordinate with the largest absolute error.
- worst_rel: One-row data frame for the coordinate with the largest relative error.
- by_coord: Data frame with per-coordinate gradient comparisons.
- fg_consistency: List describing whether fg$fg() was checked and, when available, its function and gradient consistency with fg$fn() and fg$gr().

See Also

mize()

Examples

```
rb_fg <- list(
  fn = function(x) {
    100 * (x[2] - x[1] * x[1])^2 + (1 - x[1])^2
  },
  gr = function(x) {
    c(
      -400 * x[1] * (x[2] - x[1] * x[1]) - 2 * (1 - x[1]),
      200 * (x[2] - x[1] * x[1])
    )
  }
)

check <- check_mize_gradient(rb_fg, c(-1.2, 1))
check$max_abs_error
```

---

make_mize *Create an Optimizer*

Description

Factory function for creating a (possibly uninitialized) optimizer.

Usage

```
make_mize(
  method = "L-BFGS",
  norm_direction = FALSE,
  scale_hess = TRUE,
  memory = 5,
  cg_update = "PR+",
  preconditioner = "",
  tn_init = 0,
  tn_exit = "curvature",
  nest_q = 0,
  nest_convex_approx = FALSE,
  nest_burn_in = 0,
  step_up = 1.1,
  step_up_fun = c("*", "+"),
  step_down = NULL,
  dbd_weight = 0.1,
  line_search = "More-Thuente",
  c1 = 1e-04,
  c2 = NULL,
```

```
  step0 = NULL,
  step_next_init = NULL,
  try_newton_step = NULL,
  ls_max_fn = 20,
  ls_max_gr = Inf,
  ls_max_fg = Inf,
  ls_max_alpha_mult = Inf,
  ls_max_alpha = Inf,
  ls_safe_cubic = FALSE,
  strong_curvature = NULL,
  approx_armijo = NULL,
  mom_type = NULL,
  mom_schedule = NULL,
  mom_init = NULL,
  mom_final = NULL,
  mom_switch_iter = NULL,
  mom_linear_weight = FALSE,
  use_init_mom = FALSE,
  restart = NULL,
  restart_wait = 10,
  par = NULL,
  fg = NULL,
  max_iter = 100,
  max_fn = Inf,
  max_gr = Inf,
  max_fg = Inf,
  abs_tol = NULL,
  rel_tol = abs_tol,
  grad_tol = NULL,
  ginf_tol = NULL,
  step_tol = NULL
)
```

Arguments

`method`	Optimization method. See 'Details' of `mize()`.
`norm_direction`	If `TRUE`, then the steepest descent direction is normalized to unit length. Useful for adaptive step size methods where the previous step size is used to initialize the next iteration.
`scale_hess`	If `TRUE`, scale the initial inverse-Hessian approximation using the heuristic described by Nocedal and Wright. BFGS and SR1 apply the scaling before the first quasi-Newton update if it has positive curvature; L-BFGS applies it at every iteration with stored curvature history. Ignored otherwise.
`memory`	The number of updates to store if using the `L-BFGS` method. Ignored otherwise. Must be a positive integer.
`cg_update`	Type of update to use for the `"CG"` method. For details see the "CG" subsection of the "Optimization Methods" section. Ignored if `method` is not `"CG"`.

preconditioner Type of preconditioner to use in Truncated Newton. Leave blank or set to "L-BFGS" to use a limited memory BFGS preconditioner. Use the "memory" parameter to control the number of updates to store. Applies only if method = "TN", or "CG", ignored otherwise.

tn_init Type of initialization to use in inner loop of Truncated Newton. Use 0 to use the zero vector (the usual TN initialization), or "previous" to use the final result from the previous iteration, as suggested by Martens (2010). Applies only if method = "TN", ignored otherwise.

tn_exit Type of exit criterion to use when terminating the inner CG loop of Truncated Newton method. Either "curvature" to use the standard negative curvature test, or "strong" to use the modified "strong" curvature test in TNPACK (Xie and Schlick, 1999). Applies only if method = "TN", ignored otherwise.

nest_q NAG momentum parameter. Must take a value between 0 (largest momentum) and 1 (zero momentum, giving steepest descent). Ignored unless the method is "NAG" and nest_convex_approx is FALSE.

nest_convex_approx
If TRUE, then use an approximation due to Sutskever for calculating the momentum parameter in the NAG method. Only applies if method is "NAG".

nest_burn_in Number of iterations to wait before using a non-zero momentum. Only applies if using the "NAG" method or setting the momentum_type to "Nesterov".

step_up Value by which to increase the step size for the "bold" step size method or the "DBD" method.

step_up_fun Operator to use when combining the current step size with step_up. Can be one of "*" (to multiply the current step size with step_up) or "+" (to add).

step_down Multiplier to reduce the step size by if using the "DBD" method or the "bold" line search. Can also be used with the "back" line search method, but is optional. Should be a positive value less than 1.

dbd_weight Numeric weighting value used by the "DBD" method only, and only if no momentum scheme is provided. Must be between 0 and 1, inclusive.

line_search Type of line search to use. See 'Details' of mize().

c1 Sufficient decrease parameter for Wolfe-type line searches. Should be a value between 0 and 1.

c2 Sufficient curvature parameter for line search for Wolfe-type line searches. Should be a value between c1 and 1.

step0 Initial value for the line search on the first step. See 'Details' of mize().

step_next_init For Wolfe-type line searches only, how to initialize the line search on iterations after the first. See 'Details' of mize().

try_newton_step
For Wolfe-type line searches only, try the line step value of 1 as the initial step size whenever step_next_init suggests a step size > 1. Defaults to TRUE for quasi-Newton methods such as BFGS and L-BFGS, FALSE otherwise.

ls_max_fn Maximum number of function evaluations allowed during a line search.

ls_max_gr Maximum number of gradient evaluations allowed during a line search.

ls_max_fg — Maximum number of function or gradient evaluations allowed during a line search.

ls_max_alpha_mult — The maximum value that can be attained by the ratio of the initial guess for alpha for the current line search, to the final value of alpha of the previous line search. Used to stop line searches diverging due to very large initial guesses. Only applies for Wolfe-type line searches.

ls_max_alpha — Maximum value of alpha allowed during line search. Only applies for `line_search = "more-thuente"`.

ls_safe_cubic — (Optional). If `TRUE`, check that cubic interpolation in the Wolfe line search does not produce too small a value. Only applies for `line_search = "more-thuente"`.

strong_curvature — (Optional). If `TRUE` use the strong curvature condition in Wolfe line search. See the 'Line Search' section of `mize()` for details.

approx_armijo — (Optional). If `TRUE` use the approximate Armijo condition in Wolfe line search. See the 'Line Search' section of `mize()` for details.

mom_type — Momentum type, either `"classical"` or `"nesterov"`.

mom_schedule — Momentum schedule. See 'Details' of `mize()`.

mom_init — Numeric initial momentum value. Required for the `"ramp"` and `"switch"` momentum schedules.

mom_final — Numeric final momentum value. Required for the `"ramp"` and `"switch"` momentum schedules.

mom_switch_iter — For `mom_schedule "switch"` only, the iteration when `mom_init` is changed to `mom_final`.

mom_linear_weight — If `TRUE`, the gradient contribution to the update is weighted using momentum contribution.

use_init_mom — If `TRUE`, then the momentum coefficient on the first iteration is non-zero. Otherwise, it's zero. Only applies if using a momentum schedule.

restart — Momentum restart type. Can be one of "fn" or "gr". See 'Details' of `mize()`.

restart_wait — Number of iterations to wait between restarts. Ignored if `restart` is `NULL`.

par — (Optional) Initial values for the function to be optimized over.

fg — (Optional). Function and gradient list. See 'Details' of `mize()`.

max_iter — (Optional). Maximum number of iterations. See the 'Convergence' section of `mize()` for details.

max_fn — (Optional). Maximum number of function evaluations. See the 'Convergence' section of `mize()` for details.

max_gr — (Optional). Maximum number of gradient evaluations. See the 'Convergence' section of `mize()` for details.

max_fg — (Optional). Maximum number of function or gradient evaluations. See the 'Convergence' section of `mize()` for details.

abs_tol — (Optional). Absolute tolerance for comparing two function evaluations. See the 'Convergence' section of `mize()` for details.

rel_tol	(Optional). Relative tolerance for comparing two function evaluations. See the 'Convergence' section of mize() for details.
grad_tol	(Optional). Absolute tolerance for the length (l2-norm) of the gradient vector. See the 'Convergence' section of mize() for details.
ginf_tol	(Optional). Absolute tolerance for the infinity norm (maximum absolute component) of the gradient vector. See the 'Convergence' section of mize() for details.
step_tol	(Optional). Absolute tolerance for the size of the parameter update. See the 'Convergence' section of mize() for details.

Details

If the function to be optimized and starting point are not present at creation time, then the optimizer should be initialized using mize_init() before being used with mize_step().

See the documentation to mize() for an explanation of all the parameters.

Details of the fg list containing the function to be optimized and its gradient can be found in the 'Details' section of mize(). It is optional for this function, but if it is passed to this function, along with the vector of initial values, par, the optimizer will be returned already initialized for this function. Otherwise, mize_init() must be called before optimization begins.

Additionally, optional convergence parameters may also be passed here, for use with check_mize_convergence(). They are optional here if you plan to call mize_init() later, or if you want to do your own convergence checking.

Examples

```
# Function to optimize and starting point
rosenbrock_fg <- list(
  fn = function(x) {
    100 * (x[2] - x[1] * x[1])^2 + (1 - x[1])^2
  },
  gr = function(x) {
    c(
      -400 * x[1] * (x[2] - x[1] * x[1]) - 2 * (1 - x[1]),
      200 * (x[2] - x[1] * x[1])
    )
  }
)
rb0 <- c(-1.2, 1)

# Create an optimizer and initialize it for use with the Rosenbrock function
opt <- make_mize(method = "L-BFGS", par = rb0, fg = rosenbrock_fg)

# Create optimizer without initialization
opt <- make_mize(method = "L-BFGS")

# Need to call mize_init separately:
opt <- mize_init(opt, rb0, rosenbrock_fg)
```

`mize` *Numerical Optimization*

Description

Numerical optimization including conjugate gradient, Broyden-Fletcher-Goldfarb-Shanno (BFGS), and the limited memory BFGS.

Usage

```
mize(
  par,
  fg,
  method = "L-BFGS",
  norm_direction = FALSE,
  memory = 5,
  scale_hess = TRUE,
  cg_update = "PR+",
  preconditioner = "",
  tn_init = 0,
  tn_exit = "curvature",
  nest_q = 0,
  nest_convex_approx = FALSE,
  nest_burn_in = 0,
  step_up = 1.1,
  step_up_fun = "*",
  step_down = NULL,
  dbd_weight = 0.1,
  line_search = "More-Thuente",
  c1 = 1e-04,
  c2 = NULL,
  step0 = NULL,
  step_next_init = NULL,
  try_newton_step = NULL,
  ls_max_fn = 20,
  ls_max_gr = Inf,
  ls_max_fg = Inf,
  ls_max_alpha_mult = Inf,
  ls_max_alpha = Inf,
  ls_safe_cubic = FALSE,
  strong_curvature = NULL,
  approx_armijo = NULL,
  mom_type = NULL,
  mom_schedule = NULL,
  mom_init = NULL,
  mom_final = NULL,
  mom_switch_iter = NULL,
```

```
  mom_linear_weight = FALSE,
  use_init_mom = FALSE,
  restart = NULL,
  restart_wait = 10,
  max_iter = 100,
  max_fn = Inf,
  max_gr = Inf,
  max_fg = Inf,
  abs_tol = sqrt(.Machine$double.eps),
  rel_tol = abs_tol,
  grad_tol = NULL,
  ginf_tol = NULL,
  step_tol = sqrt(.Machine$double.eps),
  check_conv_every = 1,
  log_every = check_conv_every,
  verbose = FALSE,
  store_progress = FALSE
)
```

Arguments

par	Initial values for the function to be optimized over.
fg	Function and gradient list. See 'Details'.
method	Optimization method. See 'Details'.
norm_direction	If TRUE, then the steepest descent direction is normalized to unit length. Useful for adaptive step size methods where the previous step size is used to initialize the next iteration.
memory	The number of updates to store if using the L-BFGS method. Ignored otherwise. Must be a positive integer.
scale_hess	If TRUE, scale the initial inverse-Hessian approximation using the heuristic described by Nocedal and Wright. BFGS and SR1 apply the scaling before the first quasi-Newton update if it has positive curvature; L-BFGS applies it at every iteration with stored curvature history. Ignored otherwise.
cg_update	Type of update to use for the "CG" method. For details see the "CG" subsection of the "Optimization Methods" section. Ignored if method is not "CG".
preconditioner	Type of preconditioner to use in Truncated Newton. Leave blank or set to "L-BFGS" to use a limited memory BFGS preconditioner. Use the "memory" parameter to control the number of updates to store. Applies only if method = "TN" or "CG", ignored otherwise.
tn_init	Type of initialization to use in inner loop of Truncated Newton. Use 0 to use the zero vector (the usual TN initialization), or "previous" to use the final result from the previous iteration, as suggested by Martens (2010). Applies only if method = "TN", ignored otherwise.
tn_exit	Type of exit criterion to use when terminating the inner CG loop of Truncated Newton method. Either "curvature" to use the standard negative curvature test, or "strong" to use the modified "strong" curvature test in TNPACK (Xie and Schlick, 1999). Applies only if method = "TN", ignored otherwise.

`nest_q` NAG momentum parameter. Must take a value between 0 (largest momentum) and 1 (zero momentum, giving steepest descent). Only applies using the NAG method or a momentum method with Nesterov momentum schedule. Also does nothing if `nest_convex_approx` is `TRUE`.

`nest_convex_approx` If `TRUE`, then use an approximation due to Sutskever for calculating the momentum parameter in the NAG method. Only applies using the NAG method or a momentum method with Nesterov momentum schedule.

`nest_burn_in` Number of iterations to wait before using a non-zero momentum. Only applies using the NAG method or a momentum method with Nesterov momentum schedule.

`step_up` Value by which to increase the step size for the `"bold"` step size method or the `"DBD"` method.

`step_up_fun` Operator to use when combining the current step size with `step_up`. Can be one of `"*"` (to multiply the current step size with `step_up`) or `"+"` (to add).

`step_down` Multiplier to reduce the step size by if using the `"DBD"` method or the `"bold"` line search method. Should be a positive value less than 1. Also optional for use with the `"back"` line search method.

`dbd_weight` Numeric weighting value used by the `"DBD"` method only, and only if no momentum scheme is provided. Must be between 0 and 1, inclusive.

`line_search` Type of line search to use. See 'Details'.

`c1` Sufficient decrease parameter for Wolfe-type line searches. Should be a value between 0 and 1.

`c2` Sufficient curvature parameter for line search for Wolfe-type line searches. Should be a value between `c1` and 1.

`step0` Initial value for the line search on the first step. See 'Details'.

`step_next_init` For Wolfe-type line searches only, how to initialize the line search on iterations after the first. See 'Details'.

`try_newton_step` For Wolfe-type line searches only, try the line step value of 1 as the initial step size whenever `step_next_init` suggests a step size > 1. Defaults to `TRUE` for quasi-Newton methods such as BFGS and L-BFGS, `FALSE` otherwise.

`ls_max_fn` Maximum number of function evaluations allowed during a line search.

`ls_max_gr` Maximum number of gradient evaluations allowed during a line search.

`ls_max_fg` Maximum number of function or gradient evaluations allowed during a line search.

`ls_max_alpha_mult` The maximum value that can be attained by the ratio of the initial guess for alpha for the current line search, to the final value of alpha of the previous line search. Used to stop line searches diverging due to very large initial guesses. Only applies for Wolfe-type line searches.

`ls_max_alpha` Maximum value of alpha allowed during line search. Only applies for `line_search = "more-thuente"`.

`ls_safe_cubic` (Optional). If `TRUE`, check that cubic interpolation in the Wolfe line search does not produce too small a value, using method of Xie and Schlick (2002). Only applies for `line_search = "more-thuente"`.

`strong_curvature` (Optional). If `TRUE` use the strong curvature condition in Wolfe line search. See the 'Line Search' section for details.

`approx_armijo` (Optional). If `TRUE` use the approximate Armijo condition in Wolfe line search. See the 'Line Search' section for details.

`mom_type` Momentum type, either `"classical"` or `"nesterov"`. See 'Details'.

`mom_schedule` Momentum schedule. See 'Details'.

`mom_init` Numeric initial momentum value. Required for the `"ramp"` and `"switch"` momentum schedules.

`mom_final` Numeric final momentum value. Required for the `"ramp"` and `"switch"` momentum schedules.

`mom_switch_iter` For `mom_schedule "switch"` only, the iteration when `mom_init` is changed to `mom_final`.

`mom_linear_weight` If `TRUE`, the gradient contribution to the update is weighted using momentum contribution.

`use_init_mom` If `TRUE`, then the momentum coefficient on the first iteration is non-zero. Otherwise, it's zero. Only applies if using a momentum schedule.

`restart` Momentum restart type. Can be one of "fn", "gr" or "speed". See 'Details'. Ignored if no momentum scheme is being used.

`restart_wait` Number of iterations to wait between restarts. Ignored if `restart` is `NULL`.

`max_iter` Maximum number of iterations to optimize for. Defaults to 100. See the 'Convergence' section for details.

`max_fn` Maximum number of function evaluations. See the 'Convergence' section for details.

`max_gr` Maximum number of gradient evaluations. See the 'Convergence' section for details.

`max_fg` Maximum combined number of function and gradient evaluations. See the 'Convergence' section for details.

`abs_tol` Absolute tolerance for comparing two function evaluations. See the 'Convergence' section for details.

`rel_tol` Relative tolerance for comparing two function evaluations. See the 'Convergence' section for details.

`grad_tol` Absolute tolerance for the length (l2-norm) of the gradient vector. See the 'Convergence' section for details.

`ginf_tol` Absolute tolerance for the infinity norm (maximum absolute component) of the gradient vector. See the 'Convergence' section for details.

`step_tol` Absolute tolerance for the size of the parameter update. See the 'Convergence' section for details.

`check_conv_every`
: Positive integer indicating how often to check convergence. Default is 1, i.e. every iteration. See the 'Convergence' section for details.

`log_every`
: Positive integer indicating how often to log convergence results to the console. Ignored if `verbose` is `FALSE`. If not an integer multiple of `check_conv_every`, it will be set to `check_conv_every`.

`verbose`
: If `TRUE`, log information about the progress of the optimization to the console.

`store_progress`
: If `TRUE`, store available information about the optimization in a data frame and include it in the return value. This option does not independently request objective or gradient calculations. See the 'Progress' section.

Details

The function to be optimized should be passed as a list to the `fg` parameter. This should consist of:

- `fn`. The function to be optimized. Takes a vector of parameters and returns a scalar.
- `gr`. The gradient of the function. Takes a vector of parameters and returns a vector with the same length as the input parameter vector.
- `fg`. (Optional) function which calculates the function and gradient in the same routine. Takes a vector of parameters and returns a list containing the function result as `fn` and the gradient result as `gr`.
- `hs`. (Optional) Hessian of the function. Takes a vector of parameters and returns a square matrix with dimensions the same as the length of the input vector, containing the second derivatives. For `"NEWTON"`, it may also return a vector, which is treated as the diagonal of the Hessian. Required by `"PHESS"` and by `"NEWTON"` when no `hi` function is provided.
- `hi`. (Optional) inverse Hessian of the function. Takes a vector of parameters and returns either a square matrix with dimensions the same as the length of the input vector or a vector, which is treated as the diagonal of the inverse Hessian. Used by `"NEWTON"` when `hs` is not provided, by `"BFGS"` and `"SR1"` to initialize their inverse-Hessian approximation, and by `"L-BFGS"` as the initial inverse-Hessian approximation in its two-loop recursion.

Calls to `hs` and `hi` are reported separately as `nh` and `nhi` in results and progress. These counts increase only after the callback result passes mize's Hessian or inverse-Hessian validation. Internal BFGS, SR1, and L-BFGS updates do not contribute. NEWTON requests its supplied curvature on each direction calculation; PHESS normally requests `hs` during initialization and reuses it; BFGS and SR1 request a supplied `hi` during initialization; and L-BFGS requests a supplied `hi` on each direction calculation.

The `fg` function is optional, but for some methods (e.g. line search methods based on the Wolfe criteria), both the function and gradient values are needed for the same parameter value. Calculating them in the same function can save time if there is a lot of shared work.

Value

A list with components:

- `par`: Returned parameters. When objective values are available for best-result tracking, this is an observed optimizer state with the lowest objective. If objective values never establish the comparison, `mize()` may instead use the smallest observed infinity norm of the gradient. See the 'Convergence' section for details.

- nf: Total number of function evaluations carried out. This includes any extra evaluations required for convergence calculations. Also, a function evaluation may be attempted to calculate the value of f returned in this list (see below), subject to the budget behavior in the 'Convergence' section. Additionally, if the verbose parameter is TRUE, then function and gradient information for the initial value of par will be logged to the console when available. Those values are cached for subsequent use by the optimizer.
- ng: Total number of gradient evaluations carried out. This includes any extra evaluations required for convergence calculations using grad_tol. As with nf, additional gradient calculations beyond what you're expecting may have been needed for logging, convergence and calculating the value of g2n or ginfn (see below).
- nh: Total number of accepted Hessian callback evaluations. This is zero when no fg$hs callback was used.
- nhi: Total number of accepted inverse-Hessian callback evaluations. This is zero when no fg$hi callback was used. Internal quasi-Newton updates do not contribute to nh or nhi.
- f: Value of the function at the returned value of par. This component is absent when the value is unavailable under the hard-budget behavior described in the 'Convergence' section.
- best_par: Explicit alias for par, provided so callers can use a best-versus-last naming convention.
- best_f: Value of the function at best_par. This is the same value as f and is absent under the same budget condition.
- last_par: Parameters from the last optimizer state before any final best-result restoration. This is the same as par unless mize() returns an earlier best point.
- last_f: Value of the function at last_par when available. This component is absent if its value is unavailable under the hard-budget behavior described in the 'Convergence' section. If last_par differs from par and the function value was not already available, this is NA_real_.
- g2n: Optional: the length (Euclidean or l2-norm) of the gradient vector, evaluated at the returned value of par. Calculated only if grad_tol is non-null.
- ginfn: Optional: the infinity norm (maximum absolute component) of the gradient vector, evaluated at the returned value of par. Calculated only if ginf_tol is non-null.
- iter: The number of iterations the optimization was carried out for.
- terminate: List containing what, which identifies the termination condition, and val, which records its observed value or diagnostic detail. See the 'Convergence' section for more details.
- converged: Logical value indicating whether terminate$what is one of the tolerance-based convergence criteria.
- status: Short string classifying the termination reason. One of "converged", "budget_exhausted", "failed", or "terminated".
- message: Human-readable summary of the termination reason.
- progress: Optional data frame containing optimization progress and method-specific diagnostics. Only present if store_progress is set to TRUE. See the 'Progress' section. Could get quite large if the optimization is long and progress is stored regularly.

Run vignette("convergence") for a worked example of choosing between the best and last result fields.

Optimization Methods

The `method` specifies the optimization method:

- `"SD"` is plain steepest descent. Not very effective on its own, but can be combined with various momentum approaches.
- `"BFGS"` is the Broyden-Fletcher-Goldfarb-Shanno quasi-Newton method. This stores an approximation to the inverse of the Hessian of the function being minimized, which requires storage proportional to the square of the length of `par`, so is unsuitable for large problems.
- `"SR1"` is the Symmetric Rank 1 quasi-Newton method, incorporating the safeguard given by Nocedal and Wright. Even with the safeguard, the SR1 method is not guaranteed to produce a descent direction. If this happens, the BFGS update is used for that iteration instead. Note that I have not done any research into the theoretical justification for combining BFGS with SR1 like this, but empirically (comparing to BFGS results with the datasets in the funconstrain package https://github.com/jlmelville/funconstrain), it works competitively with BFGS, particularly with a loose line search.
- `"L-BFGS"` is the Limited memory Broyden-Fletcher-Goldfarb-Shanno quasi-Newton method. This does not store the inverse Hessian approximation directly and so can scale to larger-sized problems than `"BFGS"`. The amount of memory used can be controlled with the `memory` parameter.
- `"CG"` is the conjugate gradient method. The `cg_update` parameter allows for different methods for choosing the next direction:
  - `"FR"` The method of Fletcher and Reeves.
  - `"PR"` The method of Polak and Ribiere.
  - `"PR+"` The method of Polak and Ribiere with a restart to steepest descent if conjugacy is lost. The default.
  - `"HS"` The method of Hestenes and Stiefel.
  - `"DY"` The method of Dai and Yuan.
  - `"HZ"` The method of Hager and Zhang.
  - `"HZ+"` The method of Hager and Zhang with restart, as used in CG_DESCENT.
  - `"PRFR"` The modified PR-FR method of Gilbert and Nocedal (1992). The `"PR+"` and `"HZ+"` are likely to be most robust in practice. Other updates are available more for curiosity purposes.
- `"TN"` is the Truncated Newton method, which approximately solves the Newton step without explicitly calculating the Hessian (at the expense of extra gradient calculations).
- `"NEWTON"` is Newton's method using a Hessian or inverse Hessian supplied in `fg`. It is supported, but is less commonly used than the quasi-Newton and truncated Newton methods.
- `"PHESS"` is a partial-Hessian Newton variant that reuses a Hessian factorization. It is accepted by `make_mize`, but should be treated as experimental/internal and may change without the same compatibility guarantees as the other methods.
- `"NAG"` is the Nesterov Accelerated Gradient method. The exact form of the momentum update in this method can be controlled with the following parameters:
  - `nest_q`: NAG momentum parameter. Must take a value between 0 (largest momentum) and 1 (zero momentum, giving steepest descent). Ignored if `nest_convex_approx` is `TRUE`.

  - nest_convex_approx: If TRUE, then use an approximation due to Sutskever for calculating the momentum parameter.
  - nest_burn_in: Number of iterations to wait before using a non-zero momentum.
- "DBD" is the Delta-Bar-Delta method of Jacobs.
- "Momentum" is steepest descent with momentum. See below for momentum options.

For more details on gradient-based optimization in general, and the BFGS, L-BFGS and CG methods, see Nocedal and Wright.

Line Search

The parameter line_search determines the line search to be carried out:

- "Rasmussen" carries out a line search using the strong Wolfe conditions as implemented by Carl Edward Rasmussen's minimize.m routines.
- "More-Thuente" carries out a line search using the strong Wolfe conditions and the method of More-Thuente. Can be abbreviated to "MT".
- "Schmidt" carries out a line search using the strong Wolfe conditions as implemented in Mark Schmidt's minFunc routines.
- "Backtracking" carries out a back tracking line search using the sufficient decrease (Armijo) condition. By default, cubic interpolation using function and gradient values is used to find an acceptable step size. A constant step size reduction can be used by specifying a value for step_down between 0 and 1 (e.g. step size will be halved if step_down is set to 0.5). If a constant step size reduction is used then only function evaluations are carried out and no extra gradient calculations are made.
- "Bold Driver" carries out a back tracking line search until a reduction in the function value is found. For this method, step0 must be a positive finite numeric scalar or NULL (the default), which uses an initial step size of 1.
- "Constant" uses a constant line search, the value of which should be provided with step0. Note that this value will be multiplied by the magnitude of the direction vector used in the gradient descent method. For method "SD" only, setting the norm_direction parameter to TRUE will scale the direction vector so it has unit length.

If using one of the methods: "BFGS", "L-BFGS", "CG" or "NAG", one of the Wolfe line searches: "Rasmussen" or "More-Thuente", "Schmidt" or "Hager-Zhang" should be used, otherwise very poor performance is likely to be encountered. The following parameters can be used to control the line search:

- c1: The sufficient decrease condition. Normally left at its default value of 1e-4.
- c2: The sufficient curvature condition. Defaults to 0.9 if using the methods "BFGS" and "L-BFGS", and to 0.1 for every other method, more or less in line with the recommendations given by Nocedal and Wright. The smaller the value of c2, the stricter the line search, but it should not be set to smaller than c1.
- step0: Initial value for the line search on the first step. If a positive finite numeric value is passed as an argument, that value is used as-is. Otherwise, by passing a string as an argument, a guess is made based on values of the gradient, function or parameters, at the starting point:

  - "rasmussen": As used by Rasmussen in minimize.m:

$$\frac{1}{1 + |g|^2}$$

  - "scipy": As used in optimize.py in the python library Scipy.

$$\frac{1}{|g|}$$

  - "schmidt": As used by Schmidt in minFunc.m (the reciprocal of the l1 norm of g)

$$\frac{1}{|g|_1}$$

  - "hz": The method suggested by Hager and Zhang (2006) for the CG_DESCENT software.

  These arguments can be abbreviated.

- step_next_init: How to initialize alpha value of subsequent line searches after the first, using results from the previous line search:
  - "slope ratio": Slope ratio method.
  - "quadratic": Quadratic interpolation method.
  - "hz": The QuadStep method of Hager and Zhang (2006) for the CG_DESCENT software.
  - scalar numeric: Set to a numeric value (e.g. step_next_init = 1) to explicitly set alpha to this value initially.

  These arguments can be abbreviated. Details on the first two methods are provided by Nocedal and Wright.

- try_newton_step: For quasi-Newton methods (e.g. "TN", "BFGS" and "L-BFGS"), setting this to TRUE will try the "natural" step size of 1, whenever the step_next_init method suggests an initial step size larger than that.
- strong_curvature: If TRUE, then the strong curvature condition will be used to check termination in Wolfe line search methods. If FALSE, then the standard curvature condition will be used. The default is NULL which lets the different Wolfe line searches choose whichever is their default behavior. This option is ignored if not using a Wolfe line search method.
- approx_armijo: If TRUE, then the approximate Armijo sufficient decrease condition (Hager and Zhang, 2005) will be used to check termination in Wolfe line search methods. If FALSE, then the exact Armijo condition will be used. The default is NULL which lets the different Wolfe line searches choose whichever is their default behavior. This option is ignored if not using a Wolfe line search method.

For the Wolfe line searches, the methods of "Rasmussen", "Schmidt" and "More-Thuente" default to using the strong curvature condition and the exact Armijo condition to terminate the line search (i.e. Strong Wolfe conditions). The default step size initialization methods use the Rasmussen method for the first iteration and quadratic interpolation for subsequent iterations.

The "Hager-Zhang" Wolfe line search method defaults to the standard curvature condition and the approximate Armijo condition (i.e. approximate Wolfe conditions). The default step size initialization methods are those used by Hager and Zhang (2006) in the description of CG_DESCENT.

With method = "DBD", line_search is ignored because DBD controls both the search direction and the step size. The following parameters control its coordinatewise step sizes:

- step_up: The amount by which to increase the step size in a direction where the current step size is deemed to be too short. This should be a positive scalar.
- step_down: The amount by which to decrease the step size in a direction where the currents step size is deemed to be too long. This should be a positive scalar smaller than 1. Default is 0.5.
- step_up_fun: How to increase the step size: either the method of Jacobs (addition of step_up) or Janet and co-workers (multiplication by step_up). Note that the step size decrease step_down is always a multiplication.

The "bold driver" line search also uses the step_up and step_down parameters with similar meanings to their use with the "DBD" method: the backtracking portion reduces the step size by a factor of step_down. Once a satisfactory step size has been found, the line search for the next iteration is initialized by multiplying the previously found step size by step_up.

Momentum

Momentum controls apply to method = "Momentum" and to any other method given a mom_schedule:

- mom_type: Momentum type, either "classical" or "nesterov" (case insensitive, can be abbreviated). Using "nesterov" applies the momentum step before the gradient descent as suggested by Sutskever, emulating the behavior of the Nesterov Accelerated Gradient method.
- mom_schedule: How the momentum changes over the course of the optimization:
  - If a finite numerical scalar is provided, a constant momentum will be applied throughout. Values outside [0, 1] are clamped to that range.
  - "nsconvex": Use the momentum schedule from the Nesterov Accelerated Gradient method suggested for non-strongly convex functions. Parameters which control the NAG momentum can also be used in combination with this option.
  - "switch": Switch from one momentum value (specified via mom_init) to another (mom_final) at a a specified iteration (mom_switch_iter).
  - "ramp": Linearly increase from one momentum value (mom_init) to another (mom_final).
  - If a function is provided, this will be invoked to provide a momentum value. It must take one argument (the current iteration number), may optionally take max_iter as a second argument, and must return a finite numeric scalar. Returned values are clamped to [0, 1].

  String arguments are case insensitive and can be abbreviated.

The restart parameter provides a way to restart the momentum if the optimization appears to be not be making progress, inspired by the method of O'Donoghue and Candes (2013) and Su and co-workers (2014). There are three strategies:

- "fn": A restart is applied if the function does not decrease on consecutive iterations.
- "gr": A restart is applied if the direction of the optimization is not a descent direction.
- "speed": A restart is applied if the update vector is not longer (as measured by Euclidean 2-norm) in consecutive iterations.

The effect of the restart is to "forget" any previous momentum update vector, and, for those momentum schemes that change with iteration number, to effectively reset the iteration number back to zero. If the mom_type is "nesterov", the gradient-based restart is not available. The restart_wait parameter controls how many iterations to wait after a restart, before allowing another restart. Must

be a positive integer. Default is 10, as used by Su and co-workers (2014). Setting this too low could cause premature convergence. These methods were developed specifically for the NAG method, but can be employed with any momentum type and schedule.

If `method` type `"momentum"` is specified with no other values, the momentum scheme will default to a constant value of `0.9`.

For the fixed zero-momentum forms of classical Momentum (`mom_schedule = 0`) and NAG's built-in schedule (`nest_q = 1`, `nest_convex_approx = FALSE`), status reporting follows `method = "SD"`, so momentum-specific fields such as `mu` are omitted. Adaptive restart also has no effect. Other momentum configurations retain momentum-specific reporting even if their schedule evaluates to zero.

Progress

If `store_progress = TRUE`, the returned `progress` data frame records the information available at each stored iteration. Common columns are `f`, `g2n`, `ginfn`, cumulative callback counts `nf`, `ng`, `nh`, and `nhi`, the realized outer `step`, the gradient-descent step length `alpha`, and the momentum coefficient `mu`. The four cumulative count columns are always present; `nh` and `nhi` are zero when no Hessian or inverse-Hessian callback has been accepted. Other columns are included only when their owning method or calculation supplies them.

`alpha` describes the selected gradient descent sub-step; `step` measures the complete parameter update, including momentum. For Bold Driver, `alpha` is zero when no gradient step was selected. See `mize_step_summary()` for the diagnostic field definitions.

`store_progress` controls retention of these observations. It does not by itself force objective or gradient calculations; those callbacks run only when the optimization method, convergence criteria, or another explicit calculation request needs them.

Line searches may add these diagnostic columns:

- `alpha_init`: Initial step length used by the line search after initialization and safeguards.
- `slope_init`: Directional derivative at the start of the search.
- `ls_reason`: Why the line search stopped, such as `"wolfe"`, `"armijo"`, `"objective_decrease"`, `"budget_exhausted"`, or `"rounding_stagnation"`.
- `ls_outcome`: Kind of point ultimately selected: `"wolfe"`, `"armijo"`, `"objective_decrease"`, `"improving_fallback"`, or `"no_step"`. This is independent of `ls_reason`; for example, an exhausted search can still return an evaluated improving point.
- `ls_nf` and `ls_ng`: Function and gradient callbacks owned by that outer line search, excluding its cached starting point. An objective-only Hager-Zhang initializer probe is included in `ls_nf`.

Exact Newton (`method = "NEWTON"`) may add `direction_reason`, which records how its direction was produced: `"hessian_solve"`, `"hessian_diagonal"`, `"inverse_hessian_multiply"`, `"inverse_hessian_diagonal"`, `"cholesky_fallback"`, or `"direction_check_fallback"`. The two fallback values mean that steepest descent was used because Cholesky factorization failed, or because the computed direction was non-finite or not a descent direction, respectively.

When a diagnostic column first appears after earlier progress rows, numeric columns use `NA_real_` in those rows and categorical columns use `NA_character_`. A method-specific column is absent if its owning stage never produced it.

Convergence

There are several ways for the optimization to terminate. The type of termination is communicated by a two-item list `terminate` in the return value, consisting of `what`, a short string describing what caused the termination, and `val`, the value of the termination criterion that caused termination.

The `converged`, `status`, and `message` return values summarize `terminate` without replacing it. `converged` is `TRUE` only for tolerance based termination (`"abs_tol"`, `"rel_tol"`, `"grad_tol"`, `"ginf_tol"`, or `"step_tol"`). The `status` value is `"converged"` for those tolerance exits, `"budget_exhausted"` for `"max_iter"`, `"max_fn"`, `"max_gr"`, or `"max_fg"`, `"failed"` for `"par_inf"`, `"fn_inf"`, `"gr_inf"`, or `"line_search_failed"`, and `"terminated"` for any other termination reason.

Point selection is separate from stopping. When objective values are available for best-result tracking, `par` and `best_par` identify an observed optimizer state with the lowest observed objective, while `last_par` identifies the final state before any best-result restoration. If a stopping condition was detected before restoration, `terminate`, `status`, and `converged` describe that stopping event. In contrast, `f`, `g2n`, and `ginfn` describe the returned `par`.

The following controls and runtime conditions determine stopping:

- `max_iter`: Maximum number of iterations to calculate. Reaching this limit is indicated by `terminate$what` being `"max_iter"`.
- `max_fn`: Hard maximum number of function evaluations allowed. Indicated by `terminate$what` being `"max_fn"`.
- `max_gr`: Hard maximum number of gradient evaluations allowed. Indicated by `terminate$what` being `"max_gr"`.
- `max_fg`: Hard maximum combined number of function and gradient evaluations allowed. Indicated by `terminate$what` being `"max_fg"`.
- `abs_tol`: Absolute tolerance for the change in consecutive function values. It is reached when `abs(fn_old - fn_new) < abs_tol`, and is indicated by `terminate$what` being `"abs_tol"`.
- `rel_tol`: Relative tolerance for the change in consecutive function values. It is reached when `abs(fn_old - fn_new) / min(abs(fn_old), abs(fn_new)) < rel_tol`. If the denominator is zero, the relative change is zero when the function values are equal and `Inf` otherwise. It is indicated by `terminate$what` being `"rel_tol"`.
- `grad_tol`: Absolute tolerance of the l2 (Euclidean) norm of the gradient. Indicated by `terminate$what` being `"grad_tol"`. Note that the gradient norm is not a very reliable stopping criterion (see Nocedal and co-workers 2002), but is quite commonly used, so this might be useful for comparison with results from other optimization software.
- `ginf_tol`: Absolute tolerance of the infinity norm (maximum absolute component) of the gradient. Indicated by `terminate$what` being `"ginf_tol"`.
- `step_tol`: Absolute tolerance of the step size, i.e. the Euclidean distance between values of `par` fell below the specified value. Indicated by `terminate$what` being `"step_tol"`. For those optimization methods which allow for abandoning the result of an iteration and restarting using the previous iteration's value of `par` an iteration, `step_tol` will not be triggered.
- A non-finite candidate parameter vector is indicated by `terminate$what` being `"par_inf"`. The update is rejected and `par` is rolled back before a downstream hook or callback can observe the candidate.

Once max_fn, max_gr, or max_fg is the termination reason, no later function or gradient callback is made, even if another callback-specific limit still has capacity. If the final permitted callback instead establishes convergence or a non-finite failure, that observed result takes precedence. Normal final reporting may then evaluate the objective only if doing so violates neither max_fn nor max_fg.

If a line search selects ls_outcome = "no_step" and the complete optimizer iteration also realizes a zero step, optimization terminates with terminate$what = "line_search_failed", the line-search reason in terminate$val, status = "failed", and converged = FALSE. A global callback budget reached by that attempt takes precedence. A nonzero transition from a later optimizer stage is not classified as a no-step failure.

Convergence is checked between specific iterations. How often is determined by the check_conv_every parameter, which specifies the number of iterations between each check. By default, this is set for every iteration. Line-search no-step failures are checked immediately, independently of this cadence.

Be aware that if abs_tol or rel_tol are non-NULL, this requires the function to have been evaluated at the current position at the end of each iteration. If the function at that position has not been calculated, it will be calculated only if no evaluation-budget termination is already active and neither max_fn nor max_fg prevents the callback. It will contribute to nf in the return value. The calculated function value is cached for use by the optimizer in the next iteration, so if the optimizer would have needed to calculate the function anyway (e.g. use of the strong Wolfe line search methods), there is no significant cost accrued by calculating it earlier for convergence calculations. However, for methods that don't use the function value at that location, this could represent a lot of extra function evaluations. On the other hand, not checking convergence could result in a lot of extra unnecessary iterations. Similarly, if grad_tol or ginf_tol is non-NULL, then the gradient will be calculated if needed, no evaluation-budget termination is already active, and neither max_gr nor max_fg prevents the callback.

If extra function or gradient evaluations is an issue, set check_conv_every to a higher value, but be aware that this can cause convergence limits to be exceeded by a greater amount.

Note also that if the verbose parameter is TRUE, then a summary of the results so far will be logged to the console whenever a convergence check is carried out. If the store_progress parameter is TRUE, then progress summaries and available method-specific diagnostics will be returned as a data frame. For a long optimization this could be a lot of data, so by default it is not stored.

Other ways for the optimization to terminate are if an iteration generates a non-finite (i.e. Inf, NaN, or NA) candidate parameter vector, gradient, or function value. Some, but not all, line-searches will try to recover from the latter by reducing the step size, but a non-finite gradient calculation during the gradient descent portion of optimization is considered catastrophic by mize, and it will give up. Termination under non-finite gradient or function conditions will result in terminate$what being "gr_inf" or "fn_inf" respectively; a non-finite candidate parameter vector results in "par_inf" and rollback to the parameters at the start of the iteration. Unlike the convergence criteria, the optimization will detect these error conditions and terminate even if a convergence check would not be carried out for this iteration.

The value of par in the return value should be the parameters which correspond to the lowest value of the function that has been calculated during the optimization. As discussed above however, determining which set of parameters may require a function evaluation at the end of each iteration, which only happens if either the optimization method calculates it as part of its own operation or if a convergence check is being carried out during this iteration, subject to the budget behavior described above. Therefore, if your method does not carry out function evaluations and check_conv_every

is set to be so large that no convergence calculation is carried out before `max_iter` is reached, then the returned value of `par` is the last value encountered.

References

Gilbert, J. C., & Nocedal, J. (1992). Global convergence properties of conjugate gradient methods for optimization. *SIAM Journal on optimization*, *2*(1), 21-42.

Hager, W. W., & Zhang, H. (2005). A new conjugate gradient method with guaranteed descent and an efficient line search. *SIAM Journal on Optimization*, *16*(1), 170-192.

Hager, W. W., & Zhang, H. (2006). Algorithm 851: CG_DESCENT, a conjugate gradient method with guaranteed descent. *ACM Transactions on Mathematical Software (TOMS)*, *32*(1), 113-137.

Jacobs, R. A. (1988). Increased rates of convergence through learning rate adaptation. *Neural networks*, *1*(4), 295-307.

Janet, J. A., Scoggins, S. M., Schultz, S. M., Snyder, W. E., White, M. W., & Sutton, J. C. (1998, May). Shocking: An approach to stabilize backprop training with greedy adaptive learning rates. In *1998 IEEE International Joint Conference on Neural Networks Proceedings.* (Vol. 3, pp. 2218-2223). IEEE.

Martens, J. (2010, June). Deep learning via Hessian-free optimization. In *Proceedings of the International Conference on Machine Learning.* (Vol. 27, pp. 735-742).

More', J. J., & Thuente, D. J. (1994). Line search algorithms with guaranteed sufficient decrease. *ACM Transactions on Mathematical Software (TOMS)*, *20*(3), 286-307.

Nocedal, J., Sartenaer, A., & Zhu, C. (2002). On the behavior of the gradient norm in the steepest descent method. *Computational Optimization and Applications*, *22*(1), 5-35.

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

O'Donoghue, B., & Candes, E. (2013). Adaptive restart for accelerated gradient schemes. *Foundations of computational mathematics*, *15*(3), 715-732.

Schmidt, M. (2005). minFunc: unconstrained differentiable multivariate optimization in Matlab. https://www.cs.ubc.ca/~schmidtm/Software/minFunc.html

Su, W., Boyd, S., & Candes, E. (2014). A differential equation for modeling Nesterov's accelerated gradient method: theory and insights. In *Advances in Neural Information Processing Systems* (pp. 2510-2518).

Sutskever, I. (2013). *Training recurrent neural networks* (Doctoral dissertation, University of Toronto).

Sutskever, I., Martens, J., Dahl, G., & Hinton, G. (2013). On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th international conference on machine learning (ICML-13)* (pp. 1139-1147).

Xie, D., & Schlick, T. (1999). Remark on Algorithm 702 - The updated truncated Newton minimization package. *ACM Transactions on Mathematical Software (TOMS)*, *25*(1), 108-122.

Xie, D., & Schlick, T. (2002). A more lenient stopping rule for line search algorithms. *Optimization Methods and Software*, *17*(4), 683-700.

Examples

```
# Function to optimize and starting point defined after creating optimizer
rosenbrock_fg <- list(
  fn = function(x) {
    100 * (x[2] - x[1] * x[1])^2 + (1 - x[1])^2
  },
  gr = function(x) {
    c(
      -400 * x[1] * (x[2] - x[1] * x[1]) - 2 * (1 - x[1]),
      200 * (x[2] - x[1] * x[1])
    )
  }
)
rb0 <- c(-1.2, 1)

# Minimize using L-BFGS
res <- mize(rb0, rosenbrock_fg, method = "L-BFGS")

# Conjugate gradient with Fletcher-Reeves update, tight Wolfe line search
res <- mize(rb0, rosenbrock_fg, method = "CG", cg_update = "FR", c2 = 0.1)

# Steepest decent with constant momentum = 0.9
res <- mize(rb0, rosenbrock_fg, method = "MOM", mom_schedule = 0.9)

# Steepest descent with constant momentum in the Nesterov style as described
# in papers by Sutskever and Bengio
res <- mize(rb0, rosenbrock_fg,
  method = "MOM", mom_type = "nesterov",
  mom_schedule = 0.9
)

# Nesterov momentum with adaptive restart comparing function values
res <- mize(rb0, rosenbrock_fg,
  method = "MOM", mom_type = "nesterov",
  mom_schedule = 0.9, restart = "fn"
)
```

`mize_init` *Initialize the Optimizer.*

Description

Prepares the optimizer for use with a specific function and starting point.

Usage

```
mize_init(
  opt,
  par,
```

```
  fg,
  max_iter = Inf,
  max_fn = Inf,
  max_gr = Inf,
  max_fg = Inf,
  abs_tol = NULL,
  rel_tol = abs_tol,
  grad_tol = NULL,
  ginf_tol = NULL,
  step_tol = NULL
)
```

Arguments

opt	Optimizer, created by make_mize().
par	Vector of initial values for the function to be optimized over.
fg	Function and gradient list. See the documentation of mize().
max_iter	(Optional). Maximum number of iterations. See the 'Convergence' section of mize() for details.
max_fn	(Optional). Maximum number of function evaluations. See the 'Convergence' section of mize() for details.
max_gr	(Optional). Maximum number of gradient evaluations. See the 'Convergence' section of mize() for details.
max_fg	(Optional). Maximum number of function or gradient evaluations. See the 'Convergence' section of mize() for details.
abs_tol	(Optional). Absolute tolerance for comparing two function evaluations. See the 'Convergence' section of mize() for details.
rel_tol	(Optional). Relative tolerance for comparing two function evaluations. See the 'Convergence' section of mize() for details.
grad_tol	(Optional). Absolute tolerance for the length (l2-norm) of the gradient vector. See the 'Convergence' section of mize() for details.
ginf_tol	(Optional). Absolute tolerance for the infinity norm (maximum absolute component) of the gradient vector. See the 'Convergence' section of mize() for details.
step_tol	(Optional). Absolute tolerance for the size of the parameter update. See the 'Convergence' section of mize() for details.

Details

Should be called after creating an optimizer with make_mize() and before beginning any optimization with mize_step(). Alternatively, if fg and par are available when calling make_mize(), they can be passed to that function and the returned optimizer will already be initialized. mize_step() requires an initialized optimizer and does not carry out initialization itself. Reinitialization resets the iteration counter and transient algorithm state, but preserves accumulated function, gradient, Hessian, and inverse-Hessian evaluation counts and explicitly registered custom hooks. Initialization itself may extend the curvature counts: PHESS requests fg$hs, while BFGS and SR1 request

a supplied `fg$hi` as part of initialization. Initialization does not guarantee that the objective or gradient has been evaluated at `par`. If caller-owned records need a starting observation, request it with `mize_step_summary()` using `calc_fn = TRUE` or `calc_gr = TRUE`, and retain the `opt` returned by that call. Any requested callback still honors the configured hard budgets.

Optional convergence parameters may also be passed here, for use with `check_mize_convergence()`. They are optional if you do your own convergence checking.

Details of the `fg` list can be found in the 'Details' section of `mize()`.

Value

Initialized optimizer.

Examples

```
# Create an optimizer
opt <- make_mize(method = "L-BFGS")

# Function to optimize and starting point defined after creating optimizer
rosenbrock_fg <- list(
  fn = function(x) {
    100 * (x[2] - x[1] * x[1])^2 + (1 - x[1])^2
  },
  gr = function(x) {
    c(
      -400 * x[1] * (x[2] - x[1] * x[1]) - 2 * (1 - x[1]),
      200 * (x[2] - x[1] * x[1])
    )
  }
)
rb0 <- c(-1.2, 1)

# Initialize with function and starting point before commencing optimization
opt <- mize_init(opt, rb0, rosenbrock_fg, max_iter = 3)

# Finally, commence a terminal-aware optimization loop
par <- rb0
while (!opt$is_terminated) {
  par_old <- par
  step_result <- mize_step(opt, par, rosenbrock_fg)
  opt <- step_result$opt
  par <- step_result$par
  if (opt$is_terminated) {
    break
  }

  step_info <- mize_step_summary(opt, par, rosenbrock_fg, par_old)
  opt <- step_info$opt
  if (opt$is_terminated) {
    break
  }
  opt <- check_mize_convergence(step_info)
}
```

`mize_step` *One Step of Optimization*

Description

Performs one iteration of optimization using a specified optimizer.

Usage

```
mize_step(opt, par, fg)
```

Arguments

`opt`	Optimizer, created by `make_mize()`.
`par`	Vector of initial values for the function to be optimized over.
`fg`	Function and gradient list. See the documentation of `mize()`.

Details

This function returns both the (hopefully) optimized vector of parameters, and an updated version of the optimizer itself. This is intended to be used when you want more control over the optimization process compared to the more black box approach of the `mize()` function. In return for having to manually call this function every time you want the next iteration of optimization, you gain the ability to do your own checks for convergence, logging and so on, as well as take other action between iterations, e.g. visualization.

A stateful optimization loop divides stopping work across three calls. `mize_step()` can stop immediately when a hard callback budget is exhausted or an optimization method fails. Ordinary numerical tolerances and `max_iter` are applied by `check_mize_convergence()`. After each active step, call `mize_step_summary()`, retain its returned `opt`, and then call `check_mize_convergence()` if the summary did not terminate the optimizer. The examples below show this sequence. A candidate parameter vector containing `Inf`, `NaN`, or `NA` terminates the optimizer with `opt$terminate$what = "par_inf"`; the returned `par` is rolled back to its value at the start of the step.

Normally calling this function should return a more optimized vector of parameters than the input, or at least leave the parameters unchanged if no improvement was found, although this is determined by how the optimizer was configured by `make_mize()`. It is very possible to create an optimizer that can cause a solution to diverge. It is the responsibility of the caller to check that the result of the optimization step has actually reduced the value returned from the function being optimized.

Details of the `fg` list can be found in the 'Details' section of `mize()`.

Value

Result of the current optimization step, a list with components:

- `opt`: Updated version of the optimizer passed to the `opt` argument. Should be passed as the `opt` argument in the next iteration.

- par: Updated version of the parameters passed to the par argument. Should be passed as the par argument in the next iteration.
- nf: Total number of function evaluations over the optimizer's lifetime. This count persists across mize_init() calls.
- ng: Total number of gradient evaluations over the optimizer's lifetime. This count persists across mize_init() calls.
- nh: Total number of accepted Hessian callback evaluations over the optimizer's lifetime. This count persists across mize_init() calls.
- nhi: Total number of accepted inverse-Hessian callback evaluations over the optimizer's lifetime. This count persists across mize_init() calls.
- f: Optional. The new value of the function, evaluated at the returned value of par. Only present if calculated as part of the optimization step (e.g. during a line search calculation).
- g: Optional. The gradient vector, evaluated at the returned value of par. Only present if the gradient was calculated as part of the optimization step (e.g. during a line search calculation.)

See Also

make_mize() to create a value to pass to opt, mize_init() to initialize opt before passing it to this function for the first time. mize() creates an optimizer and carries out a full optimization with it.

Examples

```
rosenbrock_fg <- list(
  fn = function(x) {
    100 * (x[2] - x[1] * x[1])^2 + (1 - x[1])^2
  },
  gr = function(x) {
    c(
      -400 * x[1] * (x[2] - x[1] * x[1]) - 2 * (1 - x[1]),
      200 * (x[2] - x[1] * x[1])
    )
  }
)
rb0 <- c(-1.2, 1)

opt <- make_mize(
  method = "SD", line_search = "const", step0 = 0.0001,
  par = rb0, fg = rosenbrock_fg, max_iter = 3
)
par <- rb0
while (!opt$is_terminated) {
  par_old <- par
  step_result <- mize_step(opt, par, rosenbrock_fg)
  opt <- step_result$opt
  par <- step_result$par
  if (opt$is_terminated) {
    break
  }
```

```
  step_info <- mize_step_summary(opt, par, rosenbrock_fg, par_old)
  opt <- step_info$opt
  if (opt$is_terminated) {
    break
  }
  opt <- check_mize_convergence(step_info)
}
```

`mize_step_summary` *Mize Step Summary*

Description

Produces a result summary for an optimization iteration. Information such as function value, gradient norm and step size may be returned.

Usage

```
mize_step_summary(opt, par, fg, par_old = NULL, calc_fn = NULL, calc_gr = NULL)
```

Arguments

`opt`	Optimizer to generate summary for, from return value of `mize_step()`.
`par`	Vector of parameters at the end of the iteration, from return value of `mize_step()`.
`fg`	Function and gradient list. See the documentation of `mize()`.
`par_old`	(Optional). Vector of parameters at the end of the previous iteration. Used to calculate step size.
`calc_fn`	(Optional). If `TRUE`, force calculation of function if not already cached in `opt`, even if it would not be needed for convergence checking.
`calc_gr`	(Optional). If `TRUE`, force calculation of gradient if not already cached in `opt`, even if it would not be needed for convergence checking.

Details

By default, convergence tolerance parameters will be used to determine what function and gradient data is returned. The function value will be returned if it was already calculated and cached in the optimization iteration. Otherwise, it will be calculated only if a non-null absolute or relative tolerance value was asked for. A gradient norm will be returned only if a non-null gradient tolerance was specified, even if the gradient is available.

If a function value is required by a tolerance but is not cached for the relevant value of `par`, it will be calculated here. The calculation contributes to the total function count and is cached for potential use in the next iteration. The same rule applies when a gradient tolerance requires a gradient calculation. Function and gradient calculation can also be forced here by setting the `calc_fn` and `calc_gr` parameters, respectively, to `TRUE`.

Requested objective and gradient calculations honor the optimizer's hard callback budgets. If a request would exceed a budget, the corresponding field is omitted and the returned `opt` records the termination. Always retain this updated optimizer before continuing. This summary does not apply ordinary numerical tolerances or `max_iter`; pass it to `check_mize_convergence()` for those checks. Non-finite observations are likewise interpreted there.

Value

A list containing the available items below. Items whose owning method or calculation did not supply a value are omitted.

- `opt`: Optimizer with updated state (e.g. function and gradient counts).
- `iter`: Iteration number.
- `f`: Function value at `par`.
- `g2n`: 2-norm of the gradient at `par`.
- `ginfn`: Infinity-norm of the gradient at `par`.
- `nf`: Number of function evaluations so far.
- `ng`: Number of gradient evaluations so far.
- `nh`: Number of accepted Hessian callback evaluations so far.
- `nhi`: Number of accepted inverse-Hessian callback evaluations so far.
- `step`: Size of the step between `par_old` and `par`, if `par_old` is provided.
- `alpha`: Step length selected for the completed gradient descent sub-step. Zero means that sub-step selected no movement; a later momentum stage can still produce a nonzero complete `step`. Before the first step, this is the initialized step length, if available.
- `mu`: Momentum coefficient for this iteration.
- `alpha_init`: Initial line-search step length after safeguards.
- `slope_init`: Directional derivative at the start of the line search.
- `ls_reason`: Reason the line search stopped.
- `ls_outcome`: Kind of point selected by the line search.
- `ls_nf`, `ls_ng`: Function and gradient callbacks owned by the line search.
- `direction_reason`: Exact-Newton direction provenance.

See the 'Progress' section of `mize()` for diagnostic value meanings.

Examples

```
rb_fg <- list(
  fn = function(x) {
    100 * (x[2] - x[1] * x[1])^2 + (1 - x[1])^2
  },
  gr = function(x) {
    c(
      -400 * x[1] * (x[2] - x[1] * x[1]) - 2 * (1 - x[1]),
      200 * (x[2] - x[1] * x[1])
    )
```

```
  }
)
rb0 <- c(-1.2, 1)

opt <- make_mize(method = "BFGS", par = rb0, fg = rb_fg, max_iter = 30)
mize_res <- mize_step(opt = opt, par = rb0, fg = rb_fg)
# Get info about first step, use rb0 to compare new par with initial value
step_info <- mize_step_summary(mize_res$opt, mize_res$par, rb_fg, rb0)
opt <- step_info$opt
if (!opt$is_terminated) {
  opt <- check_mize_convergence(step_info)
}
```

Index

check_mize_convergence, 2
check_mize_convergence(), *9*, *26, 27*, *30*
check_mize_gradient, 3

make_mize, 5
make_mize(), *2*, *25*, *27, 28*
mize, 10
mize(), *4–9*, *25–30*
mize_init, 24
mize_init(), *2*, *9*, *28*
mize_step, 27
mize_step(), *9*, *25*, *27*, *29*
mize_step_summary, 29
mize_step_summary(), *2*, *20*, *26, 27*