Package 'mize'

July 23, 2025

Type Package

Title Unconstrained Numerical Optimization Algorithms

Version 0.2.4

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

LazyData true

Imports methods

Suggests testthat, knitr, rmarkdown, covr

RoxygenNote 7.1.1

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

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

VignetteBuilder knitr

NeedsCompilation no

Author James Melville [aut, cre]

Maintainer James Melville <jlmelville@gmail.com>

Repository CRAN

Date/Publication 2020-08-30 05:20:02 UTC

Contents

`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 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 only checked here. To set these criteria, use `make_mize` or `mize_init`.

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)
# check convergence by looking at opt$is_terminated
opt <- check_mize_convergence(step_info)
```

---

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`, the approximation to the inverse Hessian is scaled according to the method described by Nocedal and Wright (approximating an eigenvalue). Applies only to the methods `BFGS` (where the scaling is applied only during the first step) and `L-BFGS` (where the scaling is applied during every iteration). 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 — Strong convexity parameter for the "NAG" method's momentum term. Must take a value between 0 (strongly convex) and 1 (results in 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". Can also be used with the "back" line search method, but is optional. Should be a positive value less than 1.

dbd_weight — Weighting parameter used by the "DBD" method only, and only if no momentum scheme is provided. Must be an integer between 0 and 1.

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	Initial momentum value.
mom_final	Final momentum value.
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, the approximation to the inverse Hessian is scaled according to the method described by Nocedal and Wright (approximating an eigenvalue). Applies only to the methods BFGS (where the scaling is applied only during the first step) and L-BFGS (where the scaling is applied during every iteration). 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	Strong convexity parameter for the NAG momentum term. Must take a value between 0 (strongly convex) and 1 (zero momentum). 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` Weighting parameter used by the `"DBD"` method only, and only if no momentum scheme is provided. Must be an integer between 0 and 1.

`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` Initial momentum value.

`mom_final` Final momentum value.

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 number of function or 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 information about the progress of the optimization in a data frame, and include it as part of the return value.

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. Only required to be present if using the "NEWTON" method. If present, it will be used during initialization for the "BFGS" and "SR1" quasi-Newton methods (otherwise, they will use the identity matrix). The quasi-Newton methods are implemented using the inverse of the Hessian, and rather than directly invert the provided Hessian matrix, will use the inverse of the diagonal of the provided Hessian only.

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 Optimized parameters. Normally, this is the best set of parameters seen during optimization, i.e. the set that produced the minimum function value. This requires that convergence checking with is carried out, including function evaluation where necessary. 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 required to calculate the value of f returned in this list (see below). Additionally, if the verbose parameter is TRUE, then function and gradient information for the initial value of par will be logged to the console. These 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 g2 or ginf (see below).
- f Value of the function, evaluated at the returned value of par.
- g2 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.
- ginf 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 items: what, indicating what convergence criterion was met, and val specifying the value at convergence. See the 'Convergence' section for more details.
- progress Optional data frame containing information on the value of the function, gradient, momentum, and step sizes evaluated at each iteration where convergence is checked. Only present if store_progress is set to TRUE. Could get quite large if the optimization is long and the convergence is checked regularly.

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).
- `"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` Strong convexity parameter. Must take a value between 0 (strongly convex) and 1 (zero momentum). 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.
- `"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 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 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.

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.

If the "DBD" is used for the optimization "method", then the line_search parameter is ignored, because this method controls both the direction of the search and the step size simultaneously. The following parameters can be used to control the step size:

- 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

For method "Momentum", momentum schemes can be accessed through the momentum arguments:

- 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 numerical scalar is provided, a constant momentum will be applied throughout.
  - "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) and return a scalar.

  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.

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 following parameters control various stopping criteria:

- max_iter Maximum number of iterations to calculate. Reaching this limit is indicated by terminate$what being "max_iter".
- max_fn Maximum number of function evaluations allowed. Indicated by terminate$what being "max_fn".
- max_gr Maximum number of gradient evaluations allowed. Indicated by terminate$what being "max_gr".
- max_fg Maximum number of gradient evaluations allowed. Indicated by terminate$what being "max_fg".
- abs_tol Absolute tolerance of the function value. If the absolute value of the function falls below this threshold, terminate$what will be "abs_tol". Will only be triggered if the objective function has a minimum value of zero.
- rel_tol Relative tolerance of the function value, comparing consecutive function evaluation results. Indicated by terminate$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.

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.

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 and will contribute to the total reported in the counts list 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.

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 the same information will be returned as a data frame in the return value. 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 is if an iteration generates a non-finite (i.e. Inf or NaN) 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. 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 requires 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. 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`. Note that if `fg` and `par` are available at the time `mize_step` is called, they can be passed to that function and initialization will be carried out automatically, avoiding the need to call `mize_init`.

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, rosebrock_fg)

# Finally, can commence the optimization loop
par <- rb0
for (iter in 1:3) {
  res <- mize_step(opt, par, rosenbrock_fg)
  par <- res$par
  opt <- res$opt
}
```

`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.

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 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. Running total number of function evaluations carried out since iteration 1.
- ng. Running total number of gradient evaluations carried out since iteration 1.
- 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
)
par <- rb0
for (iter in 1:3) {
  res <- mize_step(opt, par, rosenbrock_fg)
  par <- res$par
  opt <- res$opt
}
```

`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)
```

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.

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.

Note that if a function tolerance was specified, but was not calculated for the relevant value of `par`, they will be calculated here and the calculation does contribute to the total function count (and will be cached for potential use in the next iteration). The same applies for gradient tolerances and gradient calculation. Function and gradient calculation can also be forced here by setting the `calc_fn` and `calc_gr` (respectively) parameters to `TRUE`.

Value

A list with the following items:

- `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.
- `step` Size of the step between `par_old` and `par`, if `par_old` is provided.
- `alpha` Step length of the gradient descent part of the step.
- `mu` Momentum coefficient for this iteration

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)
```

Index