Package: psqn (via r-universe)

October 28, 2026

Type Package

Title Partially Separable Quasi-Newton

Version 0.3.2

Maintainer Benjamin Christoffersen <boennecd@gmail.com>

Description Provides quasi-Newton methods to minimize partially separable functions. The methods are largely described by Nocedal and Wright (2006) <doi:10.1007/978-0-387-40065-5>.

License Apache License (>= 2)

Encoding UTF-8

RoxygenNote 7.1.2

Depends R (>= 3.5.0), Matrix

URL https://github.com/boennecd/psqn

BugReports https://github.com/boennecd/psqn/issues

LinkingTo Rcpp, RcppEigen, testthat

Imports Rcpp

Suggests R.rsp, rmarkdown, RcppArmadillo, RcppEigen, bench, testthat, numDeriv, lbfgsb3c, lbfgs, alabama

VignetteBuilder R.rsp

NeedsCompilation yes

Author Benjamin Christoffersen [cre, aut] (<https://orcid.org/0000-0002-7182-1346>)

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

Date/Publication 2024-09-23 03:00:52 UTC

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

RemoteRef HEAD

RemoteSha 4e07bdab948cf0e5149f6a6e86f7383d2f89d03e

Contents

psqn-package 2
psqn 2
psqn_bfgs 8
psqn_generic 10
psqn_hess 16

Index **20**

`psqn-package` *psqn: Partially Separable Quasi-Newton*

Description

The main methods in the psqn package are the `psqn` and `psqn_generic` function. Notice that it is also possible to use the package from C++. This may yield a large reduction in the computation time. See the vignette for details e.g. by calling `vignette("psqn", package = "psqn")`. A brief introduction is provided in the "quick-intro" vignette (see `vignette("quick-intro", package = "psqn")`).

This package is fairly new. Thus, results may change and contributions and feedback is much appreciated.

Author(s)

Maintainer: Benjamin Christoffersen <boennecd@gmail.com> (ORCID)

See Also

Useful links:

- https://github.com/boennecd/psqn
- Report bugs at https://github.com/boennecd/psqn/issues

`psqn` *Partially Separable Function Optimization*

Description

Optimization method for specially structured partially separable functions. The `psqn_aug_Lagrang` function supports non-linear equality constraints using an augmented Lagrangian method.

Usage

```
psqn(
  par,
  fn,
  n_ele_func,
  rel_eps = 1e-08,
  max_it = 100L,
  n_threads = 1L,
  c1 = 1e-04,
  c2 = 0.9,
  use_bfgs = TRUE,
  trace = 0L,
  cg_tol = 0.5,
  strong_wolfe = TRUE,
  env = NULL,
  max_cg = 0L,
  pre_method = 1L,
  mask = as.integer(c()),
  gr_tol = -1
)

psqn_aug_Lagrang(
  par,
  fn,
  n_ele_func,
  consts,
  n_constraints,
  multipliers = as.numeric(c()),
  penalty_start = 1L,
  rel_eps = 1e-08,
  max_it = 100L,
  max_it_outer = 100L,
  violations_norm_thresh = 1e-06,
  n_threads = 1L,
  c1 = 1e-04,
  c2 = 0.9,
  tau = 1.5,
  use_bfgs = TRUE,
  trace = 0L,
  cg_tol = 0.5,
  strong_wolfe = TRUE,
  env = NULL,
  max_cg = 0L,
  pre_method = 1L,
  mask = as.integer(c()),
  gr_tol = -1
)
```

Arguments

par	Initial values for the parameters. It is a concatenated vector of the global parameters and all the private parameters.
fn	Function to compute the element functions and their derivatives. Each call computes an element function. See the examples section.
n_ele_func	Number of element functions.
rel_eps	Relative convergence threshold.
max_it	Maximum number of iterations.
n_threads	Number of threads to use.
c1, c2	Thresholds for the Wolfe condition.
use_bfgs	Logical for whether to use BFGS updates or SR1 updates.
trace	Integer where larger values gives more information during the optimization.
cg_tol	Threshold for the conjugate gradient method.
strong_wolfe	TRUE if the strong Wolfe condition should be used.
env	Environment to evaluate fn in. NULL yields the global environment.
max_cg	Maximum number of conjugate gradient iterations in each iteration. Use zero if there should not be a limit.
pre_method	Preconditioning method in the conjugate gradient method. Zero yields no preconditioning, one yields diagonal preconditioning, two yields the incomplete Cholesky factorization from Eigen, and three yields a block diagonal preconditioning. One and three are fast options with three seeming to work well for some poorly conditioned problems.
mask	zero based indices for parameters to mask (i.e. fix).
gr_tol	convergence tolerance for the Euclidean norm of the gradient. A negative value yields no check.
consts	Function to compute the constraints which must be equal to zero. See the example Section.
n_constraints	The number of constraints.
multipliers	Staring values for the multipliers in the augmented Lagrangian method. There needs to be the same number of multipliers as the number of constraints. An empty vector, numeric(), yields zero as the starting value for all multipliers.
penalty_start	Starting value for the penalty parameterin the augmented Lagrangian method.
max_it_outer	Maximum number of augmented Lagrangian steps.
violations_norm_thresh	Threshold for the norm of the constraint violations.
tau	Multiplier used for the penalty parameter between each outer iterations.

Details

The function follows the method described by Nocedal and Wright (2006) and mainly what is described in Section 7.4. Details are provided in the psqn vignette. See `vignette("psqn", package = "psqn")`.

The partially separable function we consider are special in that the function to be minimized is a sum of so-called element functions which only depend on few shared (global) parameters and some private parameters which are particular to each element function. A generic method for other partially separable functions is available through the `psqn_generic` function.

The optimization function is also available in C++ as a header-only library. Using C++ may reduce the computation time substantially. See the vignette in the package for examples.

You have to define the `PSQN_USE_EIGEN` macro variable in C++ if you want to use the incomplete Cholesky factorization from Eigen. You will also have to include Eigen or RcppEigen. This is not needed when you use the R functions documented here. The incomplete Cholesky factorization comes with some additional overhead because of the allocations of the factorization, forming the factorization, and the assignment of the sparse version of the Hessian approximation. However, it may substantially reduce the required number of conjugate gradient iterations.

Value

`pqne`: An object with the following elements:

`par`	the estimated global and private parameters.
`value`	function value at `par`.
`info`	information code. 0 implies convergence. -1 implies that the maximum number iterations is reached. -2 implies that the conjugate gradient method failed. -3 implies that the line search failed. -4 implies that the user interrupted the optimization.
`counts`	An integer vector with the number of function evaluations, gradient evaluations, and the number of conjugate gradient iterations.
`convergence`	`TRUE` if `info == 0`.

`psqn_aug_Lagrang`: Like `psqn` with a few exceptions:

`multipliers`	final multipliers from the augmented Lagrangian method.
`counts`	has an additional element called `n_aug_Lagrang` with the number of augmented Lagrangian iterations.
`penalty`	the final penalty parameter from the augmented Lagrangian method.

References

Nocedal, J. and Wright, S. J. (2006). *Numerical Optimization* (2nd ed.). Springer.

Lin, C. and Moré, J. J. (1999). *Incomplete Cholesky factorizations with limited memory*. SIAM Journal on Scientific Computing.

Examples

```
# example with inner problem in a Taylor approximation for a GLMM as in the
# vignette

# assign model parameters, number of random effects, and fixed effects
q <- 2 # number of private parameters per cluster
p <- 1 # number of global parameters
beta <- sqrt((1:p) / sum(1:p))
Sigma <- diag(q)

# simulate a data set
set.seed(66608927)
n_clusters <- 20L # number of clusters
sim_dat <- replicate(n_clusters, {
  n_members <- sample.int(8L, 1L) + 2L
  X <- matrix(runif(p * n_members, -sqrt(6 / 2), sqrt(6 / 2)),
              p)
  u <- drop(rnorm(q) %*% chol(Sigma))
  Z <- matrix(runif(q * n_members, -sqrt(6 / 2 / q), sqrt(6 / 2 / q)),
              q)
  eta <- drop(beta %*% X + u %*% Z)
  y <- as.numeric((1 + exp(-eta))^(-1) > runif(n_members))

  list(X = X, Z = Z, y = y, u = u, Sigma_inv = solve(Sigma))
}, simplify = FALSE)

# evaluates the negative log integrand.
#
# Args:
#   i cluster/element function index.
#   par the global and private parameter for this cluster. It has length
#       zero if the number of parameters is requested. That is, a 2D integer
#       vector the number of global parameters as the first element and the
#       number of private parameters as the second element.
#   comp_grad logical for whether to compute the gradient.
r_func <- function(i, par, comp_grad){
  dat <- sim_dat[[i]]
  X <- dat$X
  Z <- dat$Z

  if(length(par) < 1)
    # requested the dimension of the parameter
    return(c(global_dim = NROW(dat$X), private_dim = NROW(dat$Z)))

  y <- dat$y
  Sigma_inv <- dat$Sigma_inv

  beta <- par[1:p]
  uhat <- par[1:q + p]
  eta <- drop(beta %*% X + uhat %*% Z)
  exp_eta <- exp(eta)
```

```
  out <- -sum(y * eta) + sum(log(1 + exp_eta)) +
    sum(uhat * (Sigma_inv %*% uhat)) / 2
  if(comp_grad){
    d_eta <- -y + exp_eta / (1 + exp_eta)
    grad <- c(X %*% d_eta,
              Z %*% d_eta + dat$Sigma_inv %*% uhat)
    attr(out, "grad") <- grad
  }

  out
}

# optimize the log integrand
res <- psqn(par = rep(0, p + q * n_clusters), fn = r_func,
            n_ele_func = n_clusters)
head(res$par, p)              # the estimated global parameters
tail(res$par, n_clusters * q) # the estimated private parameters

# compare with
beta
c(sapply(sim_dat, "[[", "u"))

# add equality constraints
idx_constrained <- list(c(2L, 19L), c(1L, 5L, 8L))

# evaluates the c(x) in equalities c(x) = 0.
#
# Args:
#   i constrain index.
#   par the constrained parameters. It has length zero if we need to pass the
#       one-based indices of the parameters that the i'th constrain depends on.
#   what integer which is zero if the function should be returned and one if the
#        gradient should be computed.
consts <- function(i, par, what){
  if(length(par) == 0)
    # need to return the indices
    return(idx_constrained[[i]])

  if(i == 1){
    # a linear equality constrain. It is implemented as a non-linear constrain
    # though
    out <- sum(par) - 3
    if(what == 1)
      attr(out, "grad") <- rep(1, length(par))

  } else if(i == 2){
    # the parameters need to be on a circle
    out <- sum(par^2) - 1
    if(what == 1)
      attr(out, "grad") <- 2 * par
  }

  out
```

```
}

# optimize with the constraints
res_consts <- psqn_aug_Lagrang(
  par = rep(0, p + q * n_clusters), fn = r_func, consts = consts,
  n_ele_func = n_clusters, n_constraints = length(idx_constrained))

res_consts
res_consts$multipliers # the estimated multipliers
res_consts$penalty # the penalty parameter

# the function value is higher (worse) as expected
res$value - res_consts$value

# the two constraints are satisfied
sum(res_consts$par[idx_constrained[[1]]]) - 3   # ~ 0
sum(res_consts$par[idx_constrained[[2]]]^2) - 1 # ~ 0

# we can also use another pre conditioner
res_consts_chol <- psqn_aug_Lagrang(
  par = rep(0, p + q * n_clusters), fn = r_func, consts = consts,
  n_ele_func = n_clusters, n_constraints = length(idx_constrained),
  pre_method = 2L)

res_consts_chol
```

---

psqn_bfgs *BFGS Implementation Used Internally in the psqn Package*

Description

The method seems to mainly differ from optim by the line search method. This version uses the interpolation method with a zoom phase using cubic interpolation as described by Nocedal and Wright (2006).

Usage

```
psqn_bfgs(
  par,
  fn,
  gr,
  rel_eps = 1e-08,
  max_it = 100L,
  c1 = 1e-04,
  c2 = 0.9,
  trace = 0L,
  env = NULL,
  gr_tol = -1,
```

```
  abs_eps = -1
)
```

Arguments

par	Initial values for the parameters.
fn	Function to evaluate the function to be minimized.
gr	Gradient of fn. Should return the function value as an attribute called "value".
rel_eps	Relative convergence threshold.
max_it	Maximum number of iterations.
c1	Thresholds for the Wolfe condition.
c2	Thresholds for the Wolfe condition.
trace	Integer where larger values gives more information during the optimization.
env	Environment to evaluate fn and gr in. NULL yields the global environment.
gr_tol	Convergence tolerance for the Euclidean norm of the gradient. A negative value yields no check.
abs_eps	Absolute convergence threshold. A negative values yields no check.

Value

An object like the object returned by psqn.

References

Nocedal, J. and Wright, S. J. (2006). *Numerical Optimization* (2nd ed.). Springer.

Examples

```
# declare function and gradient from the example from help(optim)
fn <- function(x) {
  x1 <- x[1]
  x2 <- x[2]
  100 * (x2 - x1 * x1)^2 + (1 - x1)^2
}
gr <- function(x) {
  x1 <- x[1]
  x2 <- x[2]
  c(-400 * x1 * (x2 - x1 * x1) - 2 * (1 - x1),
     200 *      (x2 - x1 * x1))
}

# we need a different function for the method in this package
gr_psqn <- function(x) {
  x1 <- x[1]
  x2 <- x[2]
  out <- c(-400 * x1 * (x2 - x1 * x1) - 2 * (1 - x1),
           200 *      (x2 - x1 * x1))
  attr(out, "value") <- 100 * (x2 - x1 * x1)^2 + (1 - x1)^2
```

```
  out
}

# we get the same
optim    (c(-1.2, 1), fn, gr, method = "BFGS")
psqn_bfgs(c(-1.2, 1), fn, gr_psqn)

# compare the computation time
system.time(replicate(1000,
                      optim    (c(-1.2, 1), fn, gr, method = "BFGS")))
system.time(replicate(1000,
                      psqn_bfgs(c(-1.2, 1), fn, gr_psqn)))

# we can use an alternative convergence criterion
org <- psqn_bfgs(c(-1.2, 1), fn, gr_psqn, rel_eps = 1e-4)
sqrt(sum(gr_psqn(org$par)^2))

new_res <- psqn_bfgs(c(-1.2, 1), fn, gr_psqn, rel_eps = 1e-4, gr_tol = 1e-8)
sqrt(sum(gr_psqn(new_res$par)^2))

new_res <- psqn_bfgs(c(-1.2, 1), fn, gr_psqn, rel_eps = 1, abs_eps = 1e-2)
new_res$value - org$value # ~ there (but this is not guaranteed)
```

---

psqn_generic *Generic Partially Separable Function Optimization*

Description

Optimization method for generic partially separable functions.

Usage

```
psqn_generic(
  par,
  fn,
  n_ele_func,
  rel_eps = 1e-08,
  max_it = 100L,
  n_threads = 1L,
  c1 = 1e-04,
  c2 = 0.9,
  use_bfgs = TRUE,
  trace = 0L,
  cg_tol = 0.5,
  strong_wolfe = TRUE,
  env = NULL,
  max_cg = 0L,
  pre_method = 1L,
  mask = as.integer(c()),
```

```
  gr_tol = -1
)

psqn_aug_Lagrang_generic(
  par,
  fn,
  n_ele_func,
  consts,
  n_constraints,
  multipliers = as.numeric(c()),
  penalty_start = 1L,
  rel_eps = 1e-08,
  max_it = 100L,
  max_it_outer = 100L,
  violations_norm_thresh = 1e-06,
  n_threads = 1L,
  c1 = 1e-04,
  c2 = 0.9,
  tau = 1.5,
  use_bfgs = TRUE,
  trace = 0L,
  cg_tol = 0.5,
  strong_wolfe = TRUE,
  env = NULL,
  max_cg = 0L,
  pre_method = 1L,
  mask = as.integer(c()),
  gr_tol = -1
)
```

Arguments

par	Initial values for the parameters.
fn	Function to compute the element functions and their derivatives. Each call computes an element function. See the examples section.
n_ele_func	Number of element functions.
rel_eps	Relative convergence threshold.
max_it	Maximum number of iterations.
n_threads	Number of threads to use.
c1	Thresholds for the Wolfe condition.
c2	Thresholds for the Wolfe condition.
use_bfgs	Logical for whether to use BFGS updates or SR1 updates.
trace	Integer where larger values gives more information during the optimization.
cg_tol	Threshold for the conjugate gradient method.
strong_wolfe	TRUE if the strong Wolfe condition should be used.

env	Environment to evaluate fn in. NULL yields the global environment.
max_cg	Maximum number of conjugate gradient iterations in each iteration. Use zero if there should not be a limit.
pre_method	Preconditioning method in the conjugate gradient method. Zero yields no preconditioning, one yields diagonal preconditioning, two yields the incomplete Cholesky factorization from Eigen, and three yields a block diagonal preconditioning. One and three are fast options with three seeming to work well for some poorly conditioned problems.
mask	zero based indices for parameters to mask (i.e. fix).
gr_tol	convergence tolerance for the Euclidean norm of the gradient. A negative value yields no check.
consts	Function to compute the constraints which must be equal to zero. See the example Section.
n_constraints	The number of constraints.
multipliers	Staring values for the multipliers in the augmented Lagrangian method. There needs to be the same number of multipliers as the number of constraints. An empty vector, numeric(), yields zero as the starting value for all multipliers.
penalty_start	Starting value for the penalty parameterin the augmented Lagrangian method.
max_it_outer	Maximum number of augmented Lagrangian steps.
violations_norm_thresh	Threshold for the norm of the constraint violations.
tau	Multiplier used for the penalty parameter between each outer iterations.

Details

The function follows the method described by Nocedal and Wright (2006) and mainly what is described in Section 7.4. Details are provided in the psqn vignette. See `vignette("psqn", package = "psqn")`.

The partially separable function we consider can be quite general and the only restriction is that we can write the function to be minimized as a sum of so-called element functions each of which only depends on a small number of the parameters. A more restricted version is available through the psqn function.

The optimization function is also available in C++ as a header-only library. Using C++ may reduce the computation time substantially. See the vignette in the package for examples.

Value

A list like psqn and psqn_aug_Lagrang.

References

Nocedal, J. and Wright, S. J. (2006). *Numerical Optimization* (2nd ed.). Springer.

Lin, C. and Moré, J. J. (1999). *Incomplete Cholesky factorizations with limited memory*. SIAM Journal on Scientific Computing.

Examples

```
# example with a GLM as in the vignette

# assign the number of parameters and number of observations
set.seed(1)
K <- 20L
n <- 5L * K

# simulate the data
truth_limit <- runif(K, -1, 1)
dat <- replicate(
  n, {
    # sample the indices
    n_samp <- sample.int(5L, 1L) + 1L
    indices <- sort(sample.int(K, n_samp))

    # sample the outcome, y, and return
    list(y = rpois(1, exp(sum(truth_limit[indices]))),
         indices = indices)
  }, simplify = FALSE)

# we need each parameter to be present at least once
stopifnot(length(unique(unlist(
  lapply(dat, `[`, "indices")
))) == K) # otherwise we need to change the code

# assign the function we need to pass to psqn_generic
#
# Args:
#   i cluster/element function index.
#   par the parameters that this element function depends on. It has length zero
#       if we need to pass the one-based indices of the parameters that the i'th
#       element function depends on.
#   comp_grad TRUE of the gradient should be computed.
r_func <- function(i, par, comp_grad){
  z <- dat[[i]]
  if(length(par) == 0L)
    # return the indices
    return(z$indices)

  eta <- sum(par)
  exp_eta <- exp(eta)
  out <- -z$y * eta + exp_eta
  if(comp_grad)
    attr(out, "grad") <- rep(-z$y + exp_eta, length(z$indices))
  out
}

# minimize the function
R_res <- psqn_generic(
  par = numeric(K), fn = r_func, n_ele_func = length(dat), c1 = 1e-4, c2 = .1,
  trace = 0L, rel_eps = 1e-9, max_it = 1000L, env = environment())
```

```
# get the same as if we had used optim
R_func <- function(x){
  out <- vapply(dat, function(z){
    eta <- sum(x[z$indices])
    -z$y * eta + exp(eta)
  }, 0.)
  sum(out)
}
R_func_gr <- function(x){
  out <- numeric(length(x))
  for(z in dat){
    idx_i <- z$indices
    eta <- sum(x[idx_i])
    out[idx_i] <- out[idx_i] -z$y + exp(eta)
  }
  out
}

opt <- optim(numeric(K), R_func, R_func_gr, method = "BFGS",
             control = list(maxit = 1000L))

# we got the same
all.equal(opt$value, R_res$value)

# also works if we fix some parameters
to_fix <- c(7L, 1L, 18L)
par_fix <- numeric(K)
par_fix[to_fix] <- c(-1, -.5, 0)

R_res <- psqn_generic(
  par = par_fix, fn = r_func, n_ele_func = length(dat), c1 = 1e-4, c2 = .1,
  trace = 0L, rel_eps = 1e-9, max_it = 1000L, env = environment(),
  mask = to_fix - 1L) # notice the -1L because of the zero based indices

# the equivalent optim version is
opt <- optim(
  numeric(K - length(to_fix)),
  function(par) { par_fix[-to_fix] <- par; R_func   (par_fix) },
  function(par) { par_fix[-to_fix] <- par; R_func_gr(par_fix)[-to_fix] },
  method = "BFGS", control = list(maxit = 1000L))

res_optim <- par_fix
res_optim[-to_fix] <- opt$par

# we got the same
all.equal(res_optim, R_res$par, tolerance = 1e-5)
all.equal(R_res$par[to_fix], par_fix[to_fix]) # the parameters are fixed

# add equality constraints
idx_constrained <- list(c(2L, 19L, 11L, 7L), c(3L, 5L, 8L), 9:7)

# evaluates the c(x) in equalities c(x) = 0.
```

```
#
# Args:
#   i constrain index.
#   par the constrained parameters. It has length zero if we need to pass the
#       one-based indices of the parameters that the i'th constrain depends on.
#   what integer which is zero if the function should be returned and one if the
#        gradient should be computed.
consts <- function(i, par, what){
  if(length(par) == 0)
    # need to return the indices
    return(idx_constrained[[i]])

  if(i == 1){
    out <- exp(sum(par[1:2])) + exp(sum(par[3:4])) - 1
    if(what == 1)
      attr(out, "grad") <- c(rep(exp(sum(par[1:2])), 2),
                             rep(exp(sum(par[3:4])), 2))

  } else if(i == 2){
    # the parameters need to be on a circle
    out <- sum(par^2) - 1
    if(what == 1)
      attr(out, "grad") <- 2 * par
  } else if(i == 3){
    out <- sum(par) - .5
    if(what == 1)
      attr(out, "grad") <- rep(1, length(par))
  }

  out
}

# optimize with the constraints and masking
res_consts <- psqn_aug_Lagrang_generic(
  par = par_fix, fn = r_func, n_ele_func = length(dat), c1 = 1e-4, c2 = .1,
  trace = 0L, rel_eps = 1e-8, max_it = 1000L, env = environment(),
  consts = consts, n_constraints = length(idx_constrained),
  mask = to_fix - 1L)

res_consts

# the constraints are satisfied
consts(1, res_consts$par[idx_constrained[[1]]], 0) # ~ 0
consts(2, res_consts$par[idx_constrained[[2]]], 0) # ~ 0
consts(3, res_consts$par[idx_constrained[[3]]], 0) # ~ 0

# compare with the alabama package
if(require(alabama)){
    ala_fit <- auglag(
      par_fix, R_func, R_func_gr,
      heq = function(x){
        c(x[to_fix] - par_fix[to_fix],
          consts(1, x[idx_constrained[[1]]], 0),
```

```
          consts(2, x[idx_constrained[[2]]], 0),
          consts(3, x[idx_constrained[[3]]], 0))
    }, control.outer = list(trace = 0L))

  cat(sprintf("Difference in objective value is %.6f. Parametes are\n",
              ala_fit$value - res_consts$value))
  print(rbind(alabama = ala_fit$par,
              psqn = res_consts$par))

  cat("\nOutput from all.equal\n")
  print(all.equal(ala_fit$par, res_consts$par))
}

# the overhead here is though quite large with the R interface from the psqn
# package. A C++ implementation is much faster as shown in
# vignette("psqn", package = "psqn"). The reason it is that it is very fast
# to evaluate the element functions in this case
```

---

psqn_hess *Computes the Hessian.*

Description

Computes the Hessian using numerical differentiation with Richardson extrapolation.

Usage

```
psqn_hess(
  val,
  fn,
  n_ele_func,
  n_threads = 1L,
  env = NULL,
  eps = 0.001,
  scale = 2,
  tol = 1e-09,
  order = 6L
)

psqn_generic_hess(
  val,
  fn,
  n_ele_func,
  n_threads = 1L,
  env = NULL,
  eps = 0.001,
  scale = 2,
```

```
  tol = 1e-09,
  order = 6L
)
```

Arguments

val	Where to evaluate the function at.
fn	Function to compute the element functions and their derivatives. See psqn and psqn_generic.
n_ele_func	Number of element functions.
n_threads	Number of threads to use.
env	Environment to evaluate fn in. NULL yields the global environment.
eps	Determines the step size. See the details.
scale	Scaling factor in the Richardson extrapolation. See the details.
tol	Relative convergence criteria. See the details.
order	Maximum number of iteration of the Richardson extrapolation.

Details

The function computes the Hessian using numerical differentiation with centered differences and subsequent use of Richardson extrapolation to refine the estimate.

The additional arguments are as follows: The numerical differentiation is applied for each argument with a step size of `s = max(eps, |x| * eps)`. The Richardson extrapolation at iteration `i` uses a step size of `s * scale^(-i)`. The convergence threshold for each comportment of the gradient is `max(tol, |gr(x)[j]| * tol)`.

The numerical differentiation is done on each element function and thus much more efficient then doing it on the whole gradient.

Examples

```
# assign model parameters, number of random effects, and fixed effects
q <- 2 # number of private parameters per cluster
p <- 1 # number of global parameters
beta <- sqrt((1:p) / sum(1:p))
Sigma <- diag(q)

# simulate a data set
set.seed(66608927)
n_clusters <- 20L # number of clusters
sim_dat <- replicate(n_clusters, {
  n_members <- sample.int(8L, 1L) + 2L
  X <- matrix(runif(p * n_members, -sqrt(6 / 2), sqrt(6 / 2)),
              p)
  u <- drop(rnorm(q) %*% chol(Sigma))
  Z <- matrix(runif(q * n_members, -sqrt(6 / 2 / q), sqrt(6 / 2 / q)),
              q)
  eta <- drop(beta %*% X + u %*% Z)
```

```
  y <- as.numeric((1 + exp(-eta))^(-1) > runif(n_members))

  list(X = X, Z = Z, y = y, u = u, Sigma_inv = solve(Sigma))
}, simplify = FALSE)

# evaluates the negative log integrand.
#
# Args:
#   i cluster/element function index.
#   par the global and private parameter for this cluster. It has length
#       zero if the number of parameters is requested. That is, a 2D integer
#       vector the number of global parameters as the first element and the
#       number of private parameters as the second element.
#   comp_grad logical for whether to compute the gradient.
r_func <- function(i, par, comp_grad){
  dat <- sim_dat[[i]]
  X <- dat$X
  Z <- dat$Z

  if(length(par) < 1)
    # requested the dimension of the parameter
    return(c(global_dim = NROW(dat$X), private_dim = NROW(dat$Z)))

  y <- dat$y
  Sigma_inv <- dat$Sigma_inv

  beta <- par[1:p]
  uhat <- par[1:q + p]
  eta <- drop(beta %*% X + uhat %*% Z)
  exp_eta <- exp(eta)

  out <- -sum(y * eta) + sum(log(1 + exp_eta)) +
    sum(uhat * (Sigma_inv %*% uhat)) / 2
  if(comp_grad){
    d_eta <- -y + exp_eta / (1 + exp_eta)
    grad <- c(X %*% d_eta,
              Z %*% d_eta + dat$Sigma_inv %*% uhat)
    attr(out, "grad") <- grad
  }

  out
}

# compute the hessian
set.seed(1)
par <- runif(p + q * n_clusters, -1)

hess <- psqn_hess(val = par, fn = r_func, n_ele_func = n_clusters)

# compare with numerical differentiation from R
if(require(numDeriv)){
    hess_num <- jacobian(function(x){
        out <- numeric(length(x))
```

```
            for(i in seq_len(n_clusters)){
                out_i <- r_func(i, x[c(1:p, 1:q + (i - 1L) * q + p)], TRUE)
                out[1:p] <- out[1:p] + attr(out_i, "grad")[1:p]
                out[1:q + (i - 1L) * q + p] <- attr(out_i, "grad")[1:q + p]
            }
            out
        }, par)

        cat("Output of all.equal\n")
        print(all.equal(Matrix(hess_num, sparse = TRUE), hess))
    }
```

Index

_PACKAGE *(*psqn-package*)*, 2

optim, *8*

psqn, 2, *2*, *9*, *12*, *17*
psqn-package, 2
psqn_aug_Lagrang, *12*
psqn_aug_Lagrang *(*psqn*)*, 2
psqn_aug_Lagrang_generic
*(*psqn_generic*)*, 10
psqn_bfgs, 8
psqn_generic, *2*, *5*, 10, *17*
psqn_generic_hess *(*psqn_hess*)*, 16
psqn_hess, 16