

# Package ‘psqn’

May 4, 2021

**Type** Package

**Title** Partially Separable Quasi-Newton

**Version** 0.2.1

**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** GPL-2

**Encoding** UTF-8

**RoxygenNote** 7.1.1

**Depends** R (>= 3.5.0)

**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, Matrix

**VignetteBuilder** R.rsp

**SystemRequirements** C++11

**NeedsCompilation** yes

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

**Repository** CRAN

**Date/Publication** 2021-05-04 19:50:02 UTC

## R topics documented:

psqn-package	2
psqn	2
psqn_bfgs	5
psqn_generic	7

---

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.

### 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
  )

```

## Arguments

<code>par</code>	Initial values for the parameters. It is a concatenated vector of the global parameters and all the private parameters.
<code>fn</code>	Function to compute the element functions and their derivatives. Each call computes an element function. See the examples section.
<code>n_ele_func</code>	Number of element functions.
<code>rel_eps</code>	Relative convergence threshold.
<code>max_it</code>	Maximum number of iterations.
<code>n_threads</code>	Number of threads to use.
<code>c1, c2</code>	Thresholds for the Wolfe condition.
<code>use_bfgs</code>	Logical for whether to use BFGS updates or SR1 updates.
<code>trace</code>	Integer where larger values gives more information during the optimization.
<code>cg_tol</code>	Threshold for the conjugate gradient method.
<code>strong_wolfe</code>	TRUE if the strong Wolfe condition should be used.
<code>env</code>	Environment to evaluate <code>fn</code> in. NULL yields the global environment.
<code>max_cg</code>	Maximum number of conjugate gradient iterations in each iteration. Use zero if there should not be a limit.
<code>pre_method</code>	Preconditioning method in the conjugate gradient method. Zero yields no preconditioning, one yields diagonal preconditioning, and two yields the incomplete Cholesky factorization from Eigen.

## 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**

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.

**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"))

```

**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
)
```

### 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.

### 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)))

```

---

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
)

```

## 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, and two yields the incomplete Cholesky factorization from Eigen.

## 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](#).

## 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 this 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)

# 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
```

# Index

`_PACKAGE` (psqn-package), [2](#)

`optim`, [5](#)

`psqn`, [2](#), [2](#), [6](#), [8](#)

`psqn-package`, [2](#)

`psqn_bfgs`, [5](#)

`psqn_generic`, [2](#), [3](#), [7](#)