Package 'sqp'

March 31, 2020

Type Package

Title (Sequential) Quadratic Programming

Version 0.5

Date 2020-03-25

Author Simon Lenau

Maintainer Simon Lenau <lenau@uni-trier.de>

Description Solving procedures for quadratic programming with optional equality and inequality constraints, which can be used for by sequential quadratic programming (SQP). Similar to Newton-Raphson methods in the unconstrained case, sequential quadratic programming solves nonlinear constrained optimization problems by iteratively solving linear approximations of the optimality conditions of such a problem (cf. Powell (1978) <doi:10.1007/BFb0067703>; Nocedal and Wright (1999, ISBN: 978-0-387-98793-4)). The Hessian matrix in this strategy is commonly approximated by the BFGS method in its damped modification proposed by Powell (1978) <doi:10.1007/BFb0067703>. All methods are implemented in C++ as header-only library, such that it is easy to use in other packages.

License GPL-3

Imports Rcpp (>= 1.0.0), Matrix, Rdpack

LinkingTo Rcpp, RcppArmadillo, RcppEigen

SystemRequirements C++11, GNU Make

NeedsCompilation yes

RdMacros Rdpack

Encoding UTF-8

RoxygenNote 7.1.0

Repository CRAN

Date/Publication 2020-03-31 13:20:02 UTC

R topics documented:

`bfgs_update` *(Damped) BFGS Hessian approximation*

Description

`BFGS` update for appromation of the Hessian matrix (cf. Broyden 1970; Fletcher 1970; Goldfarb 1970; Shanno 1970) in its damped version proposed by Powell (1978). The approximation is based on first-order information (parameter values & gradients) only.

Usage

```
bfgs_update(
  hessian,
  old_y,
  new_y,
  old_gradient,
  new_gradient,
  constraint_adjustment = TRUE
)
```

Arguments

`hessian` **Dense matrix** of size $N \times N$:
Current approximation of the Hessian matrix, which is updated by reference.
Needs to be symmetric positive definite.
A common starting point for the `BFGS` algorithm is the identity matrix.

`old_y, new_y, old_gradient, new_gradient` **Numeric vectors** of size `N`:
parameters **old_y**,**new_y** and
corresponding gradients **old_gradient**,**new_gradient** from previous and current iteration.

`constraint_adjustment` **Boolean**:
Whether to enforce positive definiteness
(mainly for constrained optimization).

Value

Nothing. Argument 'hessian' is updated by reference.

References

Broyden CG (1970). "The convergence of a class of double-rank minimization algorithms: 2. The new algorithm." *IMA journal of applied mathematics*, **6**(3), 222–231. doi: 10.1093/imamat/6.3.222.

Fletcher R (1970). "A new approach to variable metric algorithms." *The computer journal*, **13**(3), 317–322. doi: 10.1093/comjnl/13.3.317.

Goldfarb D (1970). "A family of variable-metric methods derived by variational means." *Mathematics of computation*, **24**(109), 23–26. doi: 10.1090/S0025571819700258249 6.

Powell MJ (1978). "A fast algorithm for nonlinearly constrained optimization calculations." In *Numerical analysis*, 144–157. Springer. doi: 10.1007/BFb0067703.

Shanno DF (1970). "Conditioning of quasi-Newton methods for function minimization." *Mathematics of computation*, **24**(111), 647–656. doi: 10.1090/S002557181970027 4029X.

`qp_solver` *Quadratic optimization solver*

Description

Dense & Sparse solvers for linearly constrained quadratic optimization problems (cf. Fletcher 1971; Nocedal and Wright 1999; Powell 1978; Wilson 1963).

Usage

```
qp_solver(
  Q,
  C_eq = NULL,
  C_ineq = NULL,
  l = NULL,
  t_eq = NULL,
  t_ineq = NULL,
  x = NULL,
  penalty = 1e+10,
  tol = 1e-07,
  max_iter = 500,
  fast = FALSE,
  all_slack = FALSE,
  debug = FALSE,
  solver = 0
)
```

Arguments

`Q, C_eq, C_ineq` **Dense or sparse numeric matrices**:

Q $N \times N$**-matrix**:
Quadratic distance (loss) multiplier for the optimization problem.

C_eq $N_{eq} \times N$**-matrix**:
Equality constraint multiplier for the N_{eq} equality constraints.

C_ineq $N_{ineq} \times N$**-matrix**:
Inequality constraint multiplier for the N_{ineq} inequality constraints.

`l, t_eq, t_ineq` **Numeric vectors**:

l **Vector** of size N:
Linear distance (loss) multiplier for the optimization problem.

t_eq **Vector** of size N_{eq}:
Targets for equality constraints.

t_ineq **Vector** of size N_{ineq}: **upper** bound for inequality constraints.

`x` **Numeric vector** of size `N`:
Initial values for optimization parameters. Slack variables are only used for constraints violated by this `x`, unless `all_slack` is `TRUE`.

`penalty` **Numeric value**:
Penalty multiplier for slack variables in distance function.

`tol` **Numeric value**:
Tolerance for assessing convgergence criteria & constraints.

`max_iter` **Integer value**:
Tolerance for assessing convgergence criteria & constraints.

`fast` **Boolean**:
Whether to use faster (but lower quality) solver (cf. Armadillo documentation: fast mode: disable determining solution quality via rcond, disable iterative refinement, disable equilibration.

`all_slack` **Boolean**:
Whether to use slack variables for all constraints instead of
only for the ones violated by the initial values

`debug` **Boolean**:
Whether to print debugging status messages.

`solver` Solver identification used for optimization in the **dense** matrix case. Not yet used.

Details

Sequential quadratic programming relies on iteratively solving linear approximations of the optimality conditions (cf. Kjeldsen 2000; Kuhn and Tucker 1951).

This is equivalent to minimizing a quadratic approximation of the distance function under linearised constraint functions. `qp_solver` can be used to solve this quadratic sub-problem. Solving a quadratic problem under linear equalits constraints is equivalent to solving a system of linear equations. The inequality constraints are handeled by an active set strategy, where the binding ones are treated as equalities, and the active set is found iteratively (cf. Fletcher 1971; Nocedal and Wright 1999; Powell 1978; Wilson 1963).

Value

A **named list** with values

x Final values for optimization parameters

lagrange_eq, lagrange_ineq Lagrange multipliers for equality and inequality constraints

slack_eq_positive, slack_eq_negative Positive and negative slack variables for equality constraints

slack_ineq Slack variables for inequalits constraints

lagrange_slack_eq_positive, lagrange_slack_eq_negative, lagrange_slack_ineq Lagrange multipliers for positivity of slack variables

Note

Although there is already an implementation for using the SuperLU sparse solver within this package, it is currently disabled due to licensing considerations.

Sparse matrices are converted to dense ones in the solving procedure.

Hopefully, this can be updated in the near future.

References

Fletcher R (1971). "A general quadratic programming algorithm." *IMA Journal of Applied Mathematics*, **7**(1), 76–91. doi: 10.1093/imamat/7.1.76.

Kjeldsen TH (2000). "A contextualized historical analysis of the Kuhn-Tucker theorem in nonlinear programming: the impact of World War II." *Historia mathematica*, **27**(4), 331-361. doi: 10.1006/hmat.2000.2289.

Kuhn HW, Tucker AW (1951). "Nonlinear programming." In Neyman J (ed.), *Proceedings of the Second Berkeley Symposium on Mathematical Statistics and Probability*. http://web.math.ku.dk/~moller/undervisning/MASO2010/kuhntucker1950.pdf.

Nocedal J, Wright SJ (1999). *Numerical optimization*. Springer, New York. ISBN 387987932.

Powell MJ (1978). "A fast algorithm for nonlinearly constrained optimization calculations." In *Numerical analysis*, 144–157. Springer. doi: 10.1007/BFb0067703.

Wilson RB (1963). *A simplicial algorithm for concave programming*. Ph.D. thesis, Harvard University. http://faculty-gsb.stanford.edu/wilson/documents/Asimplicialalgorithmforconcaveprogramming.pdf.

Examples

```
set.seed(1)
n <- 5

x_init <- cbind(runif(n))

w <- runif(n)

Q <- 3*diag(n) # minimize sum(3*x^2 + 3*x)
l <- cbind(rep(3,n)) # minimize sum(3*x^2 + 3*x)

C_eq <- rbind(1,w) # constraints: sum(x) == 1, sum(w*x) == 5
C_ineq <- rbind(diag(n),-diag(n)) # constraints: all(x >= -4) & all(x <= 4)

t_eq <- rbind(1,5) # constraints: sum(x) == 1, sum(w*x) == 5
t_ineq <- cbind(rep(c(4,4),each=n)) # constraints: all(x >= -4) & all(x <= 4)

output <- qp_solver(Q = Q,
               C_eq = C_eq,
               C_ineq = C_ineq,
               l=l,
               t_eq = t_eq,
               t_ineq = t_ineq,
```

```
                x = x_init,
                tol = 1e-15)


sum(output$x) # constraints: sum(x) == 1
sum(w*output$x) # constraints: sum(w*x) == 5

all(output$x >= -4) # constraints: all(x >= -4)
all(output$x <= 4) # constraints: all(x <= 4)
```

Index