Package ‘cirls’

September 12, 2025

Title Constrained Iteratively Reweighted Least Squares

Version 0.4.0

Description
Fitting and inference functions for generalized linear models with constrained coefficients.

License GPL (>= 3)

Encoding UTF-8

RoxygenNote 7.3.2

Imports quadprog, osqp, coneproj, TruncatedNormal, stats, limSolve

Depends R (>= 4.1.0)

Suggests dlnm, splines, testthat (>= 3.0.0)

Config/testthat/edition 3

URL `https://github.com/PierreMasselot/cirls`

NeedsCompilation no

Author Pierre Masselot [aut, cre, cph] (ORCID:
<`https://orcid.org/0000-0002-7326-1290`>),
Antonio Gasparrini [aut] (ORCID:
<`https://orcid.org/0000-0002-2271-3568`>)

Maintainer Pierre Masselot <`pierre.masselot@lshtm.ac.uk`>

Repository CRAN

Date/Publication 2025-09-12 16:50:02 UTC

Contents

`buildCmat` *Build a constraint matrix*

Description

Function building a full constraint matrix from a list of constraint matrices and/or a formula providing specific constraints. Mainly used internally by cirls.fit.

Usage

```
buildCmat(mf, constr = NULL, Cmat = NULL, lb = 0, ub = Inf)
```

Arguments

`mf`	A model.frame or a list of variables.
`constr`	A formula specifying constraints.
`Cmat`	A named list of constraint matrices where names should be found among the terms in `mf`.
`lb, ub`	Vector or list of vectors containing constraint bounds. If a vector, is used as default bounds for terms with no specified bounds. If a named list, is matched to `Cmat` to provide corresponding bounds.

Details

This function is called internally by `cirls.fit` whenever `Cmat` is not a matrix, and provides a way to specify constraints without having to build a full constraint matrix beforehand. It uses the model frame in `mf` to match specific constraints to the right columns in the design matrix.

The argument `constr` provides a simple way to specify potentially complex constraints. It is a formula of the form `~ shape(x, ...)` where `shape` specifies the constraint and `x` the term in `mf` to which it applies. Internally, the formula will look for a function named `shapeConstr` to be called on variable `x` (which allows for several columns). The `...` represent potential additional arguments for the `shapeConstr` function. For the list of available constraints and how to create new ones, see **upcoming**.

The argument `Cmat` is used to provide a named list of constraint matrices, where names should correspond to terms in `mf`. This allows providing custom constraint matrices to specific terms that wouldn't be available through `constr`. Names in `Cmat` can include several terms, which should be separated by a `;`, for instance `x1;x2`. Although not mandatory, elements in `Cmat` can have attributes `lb`, `ub` and `vars` to provide lower and upper bounds, and term names, respectively.

`lb` and `ub` are meant to be used in conjunction with `Cmat`. If a simple value or vector, they will be used as default values for elements in `Cmat` for which no bounds is specified in its attributes. If lists, they provide bounds for constraint matrices in `Cmat`. In this case, all the names in `Cmat` should be found in `lb` and `ub`.

Note that both `constr` and `Cmat` can be used at the same time, and neither is mandatory. If both are `NULL`, an empty constraint matrix will be returned.

Value

A list with containing elements `Cmat`, `lb` and `ub` containing the full constraint matrix, lower and upper bounds for the model specified in argument `mf`. `Cmat` additionally include an attribute called `terms` which maps constraints represented in the matrix to individual terms in the model.

Examples

```
####### Upcoming
```

`checkCmat` *Check constraint matrix irreducibility*

Description

Checks a constraint matrix does not contains redundant rows

Usage

```
checkCmat(Cmat)
```

Arguments

`Cmat`	A constraint matrix as passed to `cirls.fit()`

Details

The user typically doesn't need to use `checkCmat` as it is internally called by `cirls.control()`. However, it might be useful to undertsand if `Cmat` can be reduced for inference purpose. See the note in `confint.cirls()`.

A constraint matrix is irreducible if no row can be expressed as a *positive* linear combination of the other rows. When it happens, it means the constraint is actually implicitly included in other constraints in the matrix and can be dropped. Note that this a less restrictive condition than the constraint matrix having full row rank (see some examples).

The function starts by checking if some constraints are redundant and, if so, checks if they underline equality constraints. In the latter case, the constraint matrix can be reduced by expressing these constraints as a single equality constraint with identical lower and upper bounds (see `cirls.fit()`).

The function also checks whether there are "zero constraints" i.e. constraints with only zeros in `Cmat` in which case they will be labelled as redundant.

Value

A list with three elements:

`redundant`	Logical vector of indicating redundant constraints
`equality`	Logical vector indicating which constraints are part of an underlying equality constraint

References

Meyer, M.C., 1999. An extension of the mixed primal–dual bases algorithm to the case of more constraints than dimensions. *Journal of Statistical Planning and Inference* **81**, 13–31. doi:10.1016/S03783758(99)000257

See Also

`confint.cirls()`

Examples

```
####################################################
# Example of reducible matrix

# Constraints: successive coefficients should increase and be convex
p <- 5
cmatic <- rbind(diff(diag(p)), diff(diag(p), diff = 2))

# Checking indicates that constraints 2 to 4 are redundant.
# Intuitively, if the first two coefficients increase,
# then convexity forces the rest to increase
checkCmat(cmatic)

# Check without contraints
checkCmat(cmatic[-(2:4),])

####################################################
# Example of irreducible matrix

# Constraints: coefficients form an S-shape
p <- 4
cmats <- rbind(
  diag(p)[1,], # positive
  diff(diag(p))[c(1, p - 1),], # Increasing at both end
  diff(diag(p), diff = 2)[1:(p/2 - 1),], # First half convex
  -diff(diag(p), diff = 2)[(p/2):(p-2),] # second half concave
)

# Note, this matrix is not of full row rank
qr(t(cmats))$rank
all.equal(cmats[2,] + cmats[4,] - cmats[5,], cmats[3,])

# However, it is irreducible: all constraints are necessary
```

```
checkCmat(cmats)

####################################################
# Example of underlying equality constraint

# Contraint: Parameters sum is >= 0 and sum is <= 0
cmateq <- rbind(rep(1, 3), rep(-1, 3))

# Checking indicates that both constraints imply equality constraint (sum == 0)
checkCmat(cmateq)
```

`cirls.control` *Parameters controlling CIRLS fitting*

Description

Internal function controlling the glm fit with linear constraints. Typically only used internally by cirls.fit, but may be used to construct a control argument.

Usage

```
cirls.control(constr = NULL, Cmat = NULL, lb = 0L, ub = Inf,
  epsilon = 1e-08, maxit = 25, trace = FALSE, qp_solver = "quadprog",
  qp_pars = list())
```

Arguments

`constr`	A formula specifying constraints to be applied to specific terms in the model.
`Cmat`	Constraint matrix specifying the linear constraints applied to coefficients. Can also be provided as a list of matrices for specific terms.
`lb, ub`	Lower and upper bound vectors for the linear constraints. Identical values in `lb` and `ub` identify equality constraints. As for `Cmat` can be provided as a list of terms. If some terms are provided in `Cmat` but not in `lb` or `ub`, default values of 0 and Inf will be used, respectively.
`epsilon`	Positive convergence tolerance. The algorithm converges when the relative change in deviance is smaller than `epsilon`.
`maxit`	Integer giving the maximal number of CIRLS iterations.
`trace`	Logical indicating if output should be produced for each iteration.
`qp_solver`	The quadratic programming solver. One of `"quadprog"` (the default), `"osqp"` or `"coneproj"`.
`qp_pars`	List of parameters specific to the quadratic programming solver. See respective packages help.

Details

The `control` argument of glm is by default passed to the `control` argument of cirls.fit, which uses its elements as arguments for cirls.control: the latter provides defaults and sanity checking. The control parameters can alternatively be passed through the `...` argument of glm. See glm.control for details on general GLM fitting control, and cirls.fit for details on arguments specific to constrained GLMs.

Value

A named list containing arguments to be used in cirls.fit.

See Also

the main function cirls.fit, and glm.control.

Examples

```
# Simulate predictors and response with some negative coefficients
set.seed(111)
n <- 100
p <- 10
betas <- rep_len(c(1, -1), p)
x <- matrix(rnorm(n * p), nrow = n)
y <- x %*% betas + rnorm(n)

# Define constraint matrix (includes intercept)
# By default, bounds are 0 and +Inf
Cmat <- cbind(0, diag(p))

# Fit GLM by CIRLS
res1 <- glm(y ~ x, method = cirls.fit, Cmat = Cmat)
coef(res1)

# Same as passing Cmat through the control argument
res2 <- glm(y ~ x, method = cirls.fit, control = list(Cmat = Cmat))
identical(coef(res1), coef(res2))
```

`cirls.fit` *Constrained Iteratively Reweighted Least-Squares*

Description

Fits a generalized linear model with linear constraints on the coefficients through a Constrained Iteratively Reweighted Least-Squares (CIRLS) algorithm. This function is the constrained counterpart to glm.fit and is meant to be called by glm through its `method` argument. See details for the main differences.

Usage

```
cirls.fit(x, y, weights = rep.int(1, nobs), start = NULL,
  etastart = NULL, mustart = NULL, offset = rep.int(0, nobs),
  family = stats::gaussian(), control = list(), intercept = TRUE,
  singular.ok = TRUE)
```

Arguments

`x, y`	`x` is a design matrix and `y` is a vector of response observations. Usually internally computed by glm.
`weights`	An optional vector of observation weights.
`start`	Starting values for the parameters in the linear predictor.
`etastart`	Starting values for the linear predictor.
`mustart`	Starting values for the vector or means.
`offset`	An optional vector specifying a known component in the model. See model.offset.
`family`	The result of a call to a family function, describing the error distribution and link function of the model. See family for details of available family functions.
`control`	A list of parameters controlling the fitting process. See details and cirls.control.
`intercept`	Logical. Should an intercept be included in the null model?
`singular.ok`	Logical. If `FALSE`, the function returns an error for singular fits.

Details

This function is a plug-in for glm and works similarly to glm.fit. In addition to the parameters already available in glm.fit, `cirls.fit` allows the specification of a constraint matrix `Cmat` with bound vectors `lb` and `ub` on the regression coefficients. These additional parameters can be passed through the `control` list or through `...` in glm *but not both*. If any parameter is passed through `control`, then `...` will be ignored.

The CIRLS algorithm is a modification of the classical IRLS algorithm in which each update of the regression coefficients is performed by a quadratic program (QP), ensuring the update stays within the feasible region defined by `Cmat`, `lb` and `ub`. More specifically, this feasible region is defined as `lb <= Cmat %*% coefficients <= ub`

where `coefficients` is the coefficient vector returned by the model. This specification allows for any linear constraint, including equality ones.

Specifying constraints:

The package includes several mechanisms to specify constraints. The most straightforward is to pass a full matrix to `Cmat` with associated bound vectors in `lb` and `ub`. In this case, the number of columns in `Cmat` must match the number of coefficients estimated by glm. This includes all variables that are not involved in any constraint, potential expansion such as factors or splines for instance, as well as the intercept. By default `lb` and `ub` are set to `0` and `Inf`, respectively, but any bounds are possible. When some elements of `lb` and `ub` are identical, they define equality constraints. Setting `lb = -Inf` and `ub = Inf` disable the constraints.

To avoid pre-constructing potentially large and complex `Cmat` objects, the arguments `Cmat` and `constr` can be combined to conveniently specify constraints for the coefficients. More specifically, `Cmat` can alternatively take a named list of matrices to constrain only specific terms in the model. The argument `constr` provides a formula interface to specify built-in common constraints. The documentation of buildCmat provides full details on how to specify constraints along with examples.

Quadratic programming solvers:

The function cirls.fit relies on a quadratic programming solver. Several solver are currently available.

- `"quadprog"` (the default) performs a dual algorithm to solve the quadratic program. It relies on the function solve.QP.
- `"osqp"` solves the quadratic program via the Alternating Direction Method of Multipliers (ADMM). Internally it calls the function solve_osqp.
- `"coneproj"` solves the quadratic program by a cone projection method. It relies on the function qprog.

Each solver has specific parameters that can be controlled through the argument `qp_pars`. Sensible defaults are set within cirls.control and the user typically doesn't need to provide custom parameters. `"quadprog"` is set as the default being generally more reliable than the other solvers. `"osqp"` is faster but can be less accurate, in which case it is recommended to increase convergence tolerance at the cost of speed.

Value

A `cirls` object inheriting from the class `glm`. At the moment, two non-standard methods specific to `cirls` objects are available: vcov.cirls to obtain the coefficients variance-covariance matrix and confint.cirls to obtain confidence intervals. These custom methods account for the reduced degrees of freedom resulting from the constraints, see vcov.cirls and confint.cirls.

An object of class `cirls` includes all components from glm objects, with the addition of:

`Cmat`, `lb`, `ub`	the constraint matrix, and lower and upper bound vectors. If provided as lists, the full expanded matrix and vectors are returned.
`active.cons`	vector of indices of the active constraints in the fitted model.
`inner.iter`	number of iterations performed by the last call to the QP solver.
`etastart`	the initialisation of the linear predictor `eta`. The same as `etastart` when provided.
`singular.ok`	the value of the `singular.ok` argument.

Any method for `glm` objects can be used on `cirls` objects. Several methods specific to `cirls` are available: vcov.cirls to obtain the coefficients variance-covariance matrix, confint.cirls to obtain confidence intervals, and logLik.cirls to extract the log-likelihood with appropriate degrees of freedom.

References

Goldfarb, D., Idnani, A., 1983. A numerically stable dual method for solving strictly convex quadratic programs. *Mathematical Programming* **27**, 1–33. doi:10.1007/BF02591962

Meyer, M.C., 2013. A Simple New Algorithm for Quadratic Programming with Applications in Statistics. *Communications in Statistics - Simulation and Computation* **42**, 1126–1139. doi:10.1080/03610918.2012.659820

Stellato, B., Banjac, G., Goulart, P., Bemporad, A., Boyd, S., 2020. OSQP: an operator splitting solver for quadratic programs. *Math. Prog. Comp.* **12**, 637–672. doi:10.1007/s12532020001792

See Also

vcov.cirls, confint.cirls, logLik.cirls and edf for methods specific to `cirls` objects. cirls.control for fitting parameters specific to cirls.fit. glm for details on `glm` objects.

Examples

```
#####################################################
# Simple non-negative least squares

# Simulate predictors and response with some negative coefficients
set.seed(111)
n <- 100
p <- 10
betas <- rep_len(c(1, -1), p)
x <- matrix(rnorm(n * p), nrow = n)
y <- x %*% betas + rnorm(n)

# Define constraint matrix (includes intercept)
# By default, bounds are 0 and +Inf
Cmat <- cbind(0, diag(p))

# Fit GLM by CIRLS
res1 <- glm(y ~ x, method = cirls.fit, Cmat = Cmat)
coef(res1)

# Same as passing Cmat through the control argument
res2 <- glm(y ~ x, method = cirls.fit, control = list(Cmat = Cmat))
identical(coef(res1), coef(res2))

#####################################################
# Increasing coefficients

# Generate two group of variables: an isotonic one and an unconstrained one
set.seed(222)
p1 <- 5; p2 <- 3
x1 <- matrix(rnorm(100 * p1), 100, p1)
x2 <- matrix(rnorm(100 * p2), 100, p2)

# Generate coefficients: those in b1 should be increasing
b1 <- runif(p1) |> sort()
b2 <- runif(p2)

# Generate full data
y <- x1 %*% b1 + x2 %*% b2 + rnorm(100, sd = 2)

#----- Fit model

# Create constraint matrix and expand for intercept and unconstrained variables
```

```
Ciso <- diff(diag(p1))
Cmat <- cbind(0, Ciso, matrix(0, nrow(Ciso), p2))

# Fit model
resiso <- glm(y ~ x1 + x2, method = cirls.fit, Cmat = Cmat)
coef(resiso)

# Compare with unconstrained
plot(c(0, b1, b2), pch = 16)
points(coef(resiso), pch = 16, col = 3)
points(coef(glm(y ~ x1 + x2)), col = 2)

#----- More convenient specification

# Cmat can be provided as a list
resiso2 <- glm(y ~ x1 + x2, method = cirls.fit, Cmat = list(x1 = Ciso))

# Internally Cmat is expanded and we obtain the same result
identical(resiso$Cmat, resiso2$Cmat)
identical(coef(resiso), coef(resiso2))

#----- Adding bounds to the constraints
# Difference between coefficients must be above a lower bound and below 1
lb <- 1 / (p1 * 2)
ub <- 1

# Re-fit the model
resiso3 <- glm(y ~ x1 + x2, method = cirls.fit, Cmat = list(x1 = Ciso),
  lb = lb, ub = ub)

# Compare the fit
plot(c(0, b1, b2), pch = 16)
points(coef(resiso), pch = 16, col = 3)
points(coef(glm(y ~ x1 + x2)), col = 2)
points(coef(resiso3), pch = 16, col = 4)
```

dmat *Spline derivative matrix*

Description

Computes a derivative matrix for B-splines that can then be used for shape-constraints. It is internally called by shapeConstr and should not be used directly.

Usage

```
dmat(d, s, knots, ord)
```

Arguments

`d`	Non-negative integer giving the order of derivation. Should be between 0 and `ord - 2`.
`s`	Sign of the derivative.
`knots`	Vector of ordered knots from the spline bases.
`ord`	Non-negative integer giving the order of the spline.

Details

Does the heavy lifting in shapeConstr to create a constraint matrix for shape-constrained B-splines. Only useful for advanced users to create constraint matrices without passing an object to one of the shapeConstr methods.

Value

A matrix of weighted differences that can be used to constrain B-spline bases.

Note

`dmat` doesn't perform any checks of the parameters so use carefully. In normal usage, checks are done by shapeConstr methods.

Examples

```
# A second derivative matrix for cubic B-Splines with regularly spaced knots
# Can be used to enforce convexity
cirls:::dmat(2, 1, 1:15, 4)
```

`edf` *Expected degrees of freedom*

Description

Estimate expected degrees of freedom of a cirls object through simulations.

Usage

```
edf(object, nsim = 10000, seed = NULL)
```

Arguments

`object`	A `cirls` object or any object inheriting from `lm`, see details.
`nsim`	The number of simulations.
`seed`	An optional seed for the random number generator. See set.seed.

Details

Simulates coefficient vectors from their unconstrained distribution, which is the non-truncated multivariate normal distribution. For each simulated vector, counts the number of violated constraints as the number of active constraints under the constrained distribution. The expected degrees of freedom is then the number of parameters minus the average number of active constraints.

This procedure allows to account for the randomness of degrees of freedom for the constrained model. Indeed, the observed degrees of freedom is the number of parameters minus the number of active constraints. However, the number of active constraints is random as, some constraints can be active or not depending on the observed data. For instance, in a model for which the constraints are binding, the expected degrees of freedom will be close to the observed one, while in a model in which the constraints are irrelevant, the expected degrees of freedom will be closer to the unconstrained (usual) ones.

Value

A vector of length three with components:

`udf`	The *unconstrained* degrees of freedom, i.e. the rank plus any dispersion parameter for `glm` objects.
`odf`	The *observed* degrees of freedom, that is `udf` minus the number of active constraints.
`edf`	The *expected* degrees of freedom estimated by simulation as described in the details section. For any other object inheriting from `lm`, attempts to retrieve the *effective* degrees of freedom.

For `cirls` objects, the vector includes the simulated distribution of the number of active constraints as an `actfreq` attribute.

Note

This function is implemented mainly for cirls objects and can return idiosyncratic results for other objects inheriting from `lm`. In this case, it will attempt to retrieve an 'edf' value, but simply return the rank of the model if this fails. For `glm` models for instance, it will return thrice the same value.

References

Meyer, M.C., 2013. Semi-parametric additive constrained regression. *Journal of Nonparametric Statistics* **25**, **715–730**. doi:10.1080/10485252.2013.797577

See Also

logLik.cirls which internally calls `edf` to compute degrees of freedom.

Examples

```
# Simulate a simple dataset
set.seed(5)
x <- rnorm(100)
y <- x + rnorm(100)
```

```
#### Model with a sensible constraint
# Reduces edf compared to udf as the constraint is sometimes active
mod1 <- glm(y ~ x, method = "cirls.fit", Cmat = list(x = 1), lb = 1)
edf(mod1)

#### Model with an almost surely binding constraint
# In this case edf is very close to odf as the constraint is often active
mod2 <- glm(y ~ x, method = "cirls.fit", Cmat = list(x = 1), lb = 1.5)
edf(mod2)

#### Model with an irrelevant constraint
# Here the constraint is useless and edf is equal to unconstrained df
mod3 <- glm(y ~ x, method = "cirls.fit", Cmat = list(x = 1), lb = -5)
edf(mod3)
```

`logLik.cirls` *Log-Likelihood for a fitted* `cirls` *object*

Description

Extracts the log-likelihood for a fitted `cirls` object to be typically used by AIC.

Usage

```
## S3 method for class 'cirls'
logLik(object, df = "edf", ...)
```

Arguments

`object`	A `cirls` object.
`df`	The type of degrees of freedom to assign to the log-Likelihood. Default to expected degrees of freedom. See `edf()`.
`...`	Arguments to be passed to edf to compute degrees of freedom.

Details

The argument `df` provide the type of degrees of freedom attributed to the returned log-likelihood value. This is typically used in the computation of AIC and BIC and changing the degrees of freedom can ultimately change the values of the information criteria. By default, the expected number of freedom given the constraints is used. See edf for details on the computation and for the returned types of degrees of freedom.

Value

A numeric value of class `logLik` with attributes `df` (degrees of freedom, see details) and `nobs` (number of observations used in the estimation).

See Also

edf to compute expected degrees of freedom.

`shapeConstr` *Create shape constraints*

Description

Creates a constraint matrix to shape-constrain a set of coefficients. Mainly intended for splines but can constrain various bases or set of variables. Will typically be called from within cirls.fit but can be used to generate constraint matrices.

Usage

```
shapeConstr(x, shape, ...)

## Default S3 method:
shapeConstr(x, shape, intercept = FALSE, ...)

## S3 method for class 'factor'
shapeConstr(x, shape, intercept = FALSE, ...)
```

Arguments

`x`	An object representing a design matrix of predictor variables, typically basis functions. See details for supported objects.
`shape`	A character vector indicating one or several shape-constraints. See details for supported shapes.
`...`	Additional parameters passed to or from other methods.
`intercept`	For the default method, a logical value indicating if the design matrix includes an intercept. In most cases will be automatically extracted from `x`.

Details

The recommended usage is to directly specify the shape constraint through the `shape` argument in the call to glm with cirls.fit. This method is then called internally to create the constraint matrix. However, `shapeConstr` can nonetheless be called directly to manually build or inspect the constraint matrix for a given shape and design matrix.

The parameters necessary to build the constraint matrix (e.g. `knots` and `ord` for splines) are typically extracted from the `x` object. This is also true for the `intercept` for most of the object, except for the default method for which it can be useful to explicitly provide it. In a typical usage in which `shapeConstr` would only be called within cirls.fit, `intercept` is automatically determined from the glm formula.

Allowed shapes:

The `shape` argument allows to define a specific shape for the association between the expanded term in `x` and the response of the regression model. This shape can describe the relation between coefficients for the default method, or the shape of the smooth term for spline bases. At the moment, six different shapes are supported, with up to three allowed simultaneously (one from each category):

- `"pos"` or `"neg"`: Positive/Negative. Applies to the full association.
- `"inc"` or `"dec"`: Monotonically Increasing/Decreasing.
- `"cvx"` or `"ccv"`: Convex/Concave.

Available methods:

In addition to the default method, `shapeConstr` currently supports several objects, creating an appropriate shape-constraint matrix depending on the object. The full list can be obtained by `methods(shapeConstr)`.

General:

- `factor()`: for categorical variables. Extract the contrasts to define the constraint matrix. here the `intercept` argument has the same interpretation as in the default method, i.e. if set to `TRUE` it means the `glm` model doesn't include an intercept externally to the factor. Note that, in this case, a simple dummy coding is done in R.

From the splines package:

- bs: B-splines.
- ns: Natural splines.

From the dlnm package:

- onebasis: General method for basis functions generated in the package.
- ps: Penalised splines (P-Splines).

Value

A constraint matrix to be passed to `Cmat` in cirls.fit.

References

Zhou, S. & Wolfe, D. A., 2000, On derivative estimation in spline regression. *Statistica Sinica* **10**, **93–108**.

See Also

`cirls.fit()` which typically calls `shapeConstr` internally.

Examples

```
# example code
```

simulCoef — *Simulate coefficients, calculate Confidence Intervals and Variance-Covariance Matrix for a `cirls` object.*

Description

Simulates coefficients for a fitted `cirls` object. `confint` and `vcov` compute confidence intervals and the Variance-Covariance matrix for coefficients from a fitted `cirls` object. These methods supersede the default methods for `cirls` objects.

Usage

```
simulCoef(object, nsim = 1, seed = NULL, complete = TRUE,
  constrained = TRUE)

## S3 method for class 'cirls'
confint(object, parm, level = 0.95, nsim = 1000,
  complete = TRUE, ...)

## S3 method for class 'cirls'
vcov(object, complete = TRUE, nsim = 1000, trunc = TRUE,
  ...)
```

Arguments

`object`	A fitted `cirls` object.
`nsim`	The number of simulations to perform.
`seed`	Either NULL or an integer that will be used in a call to `set.seed()` before simulating the coefficients.
`complete`	If FALSE, doesn't return inference for undetermined coefficients in case of an over-determined model.
`constrained`	A logical switch indicating Whether to simulate from the constrained (the default) or unconstrained coefficients distribution.
`parm`	A specification of which parameters to compute the confidence intervals for. Either a vector of numbers or a vector of names. If missing, all parameters are considered.
`level`	The confidence level required.
`...`	Further arguments passed to or from other methods. For `vcov` and `confint` can be used to provide a `seed` for the internal coefficient simulation.
`trunc`	If set to `FALSE` the unmodified GLM variance-covariance computed within `summary.glm()` is returned.

Details

confint and vcov are custom methods for cirls objects to supersede the default methods used for glm objects. Internally, they both call simulCoef to generate coefficient vectors from a Truncated Multivariate Normal Distribution using the TruncatedNormal::rtmvnorm() function. This distribution accounts for truncation by constraints, ensuring all coefficients are feasible with respect to the constraint matrix. simulCoef typically doesn't need to be used directly for confidence intervals and variance-covariance matrices, but it can be used to check other summaries of the coefficients distribution.

These methods only work when Cmat is of full row rank, i.e. if there are less constraints than variables in object.

Value

For simulCoef, a matrix with nsim rows containing simulated coefficients.

For confint, a two-column matrix with columns giving lower and upper confidence limits for each parameter.

For vcov, a matrix of the estimated covariances between the parameter estimates of the model.

Note

By default, the Variance-Covariance matrix generated by vcov is different than the one returned by summary(obj)$cov.scaled. The former accounts for the reduction in degrees of freedom resulting from the constraints, while the latter is the unconstrained GLM Variance-Covariance. Note that the unconstrained one can be obtained from vcov by setting constrained = FALSE.

References

Geweke, J.F., 1996. Bayesian Inference for Linear Models Subject to Linear Inequality Constraints, in: Lee, J.C., Johnson, W.O., Zellner, A. (Eds.), Modelling and Prediction Honoring Seymour Geisser. *Springer, New York, NY*, pp. 248–263. doi:10.1007/9781461224143_15

Botev, Z.I., 2017, The normal law under linear restrictions: simulation and estimation via minimax tilting, *Journal of the Royal Statistical Society, Series B*, **79** (**1**), pp. 1–24.

See Also

rtmvnorm for the underlying routine to simulate from a TMVN. checkCmat() to check if the contraint matrix can be reduced.

Examples

```
####################################################
# Isotonic regression

#----- Perform isotonic regression

# Generate data
set.seed(222)
p1 <- 5; p2 <- 3
```

```
x1 <- matrix(rnorm(100 * p1), 100, p1)
x2 <- matrix(rnorm(100 * p2), 100, p2)
b1 <- runif(p1) |> sort()
b2 <- runif(p2)
y <- x1 %*% b1 + x2 %*% b2 + rnorm(100, sd = 2)

# Fit model
Ciso <- diff(diag(p1))
resiso <- glm(y ~ x1 + x2, method = cirls.fit, Cmat = list(x1 = Ciso))

#----- Extract uncertainty

# Extract variance covariance
vcov(resiso)

# Extract confidence intervals
confint(resiso)

# We can extract the usual unconstrained matrix
vcov(resiso, constrained = FALSE)
all.equal(vcov(resiso, constrained = FALSE), summary(resiso)$cov.scaled)

# Simulate from the distribution of coefficients
sims <- simulCoef(resiso, nsim = 10)

# Check that all simulated coefficient vectors are feasible
apply(resiso$Cmat %*% t(sims) >= resiso$lb, 2, all)
```

`uncons` *Unconstrained model*

Description

Takes a fitted `cirls` object and returns the corresponding unconstrained model.

Usage

```
uncons(object)
```

Arguments

`object`	Fitted 'cirls' object.

Details

Note on starting values:

If any starting values were provided to fit the `cirls` object, they are not transferred to the fitting of the unconstrained model.

Value

A `cirls` object.

Examples

```
# Generate some data
n <- 1000
betas <- c(0, 1, 2, -1, 1)
p <- length(betas)
x <- matrix(rnorm(n * p), n, p)
eta <- 5 + x %*% betas
y <- eta + rnorm(n, 0, .2)

# Fit two cirls models, passing Cmat through the two different pathways
cinc <- diff(diag(p))
res1 <- glm(y ~ x, method = cirls.fit, Cmat = list(x = cinc))
res2 <- glm(y ~ x, method = cirls.fit, control = list(Cmat = list(x = cinc)))

# 'Unconstrain' the models
uc1 <- uncons(res1)
uc2 <- uncons(res2)
```

`zerosumConstr` *Zero-sum constraint matrix*

Description

Build constraint matrix and bounds for coefficients summing to zero.

Usage

```
zerosumConstr(..., group = FALSE)
```

Arguments

`...`	Variables to be included in the constraint.
`group`	If set to TRUE, the constraint is build independently for each variable in `...`

Index