Package: CVXR (via r-universe)

October 20, 2026

Type Package

Title Disciplined Convex Optimization

Version 1.9.3

Date 2026-10-18

URL https://cvxr.rbind.io, https://www.cvxgrp.org/CVXR/

BugReports https://github.com/cvxgrp/CVXR/issues

Description An object-oriented modeling language for disciplined convex programming (DCP) as described in Fu, Narasimhan, and Boyd (2020, <doi:10.18637/jss.v094.i14>). It allows the user to formulate convex optimization problems in a natural way following mathematical convention and DCP rules. The system analyzes the problem, verifies its convexity, converts it into a canonical form, and hands it off to an appropriate solver to obtain the solution. This version uses the S7 object system for improved performance and maintainability.

Depends R (>= 4.3.0)

Imports S7 (>= 0.2), methods, Matrix (>= 1.7), Rcpp (>= 1.1), clarabel (>= 0.11), cli (>= 3.6), gmp (>= 0.7), highs (>= 1.14), osqp (>= 1.0), scs (>= 3.2), slam (>= 0.1)

Suggests jsonlite (>= 1.9), knitr, rmarkdown, testthat (>= 3.3), rlang

Enhances Rmosek, gurobi (>= 13.0), Rglpk (>= 0.6), ECOSolveR (>= 0.6.2), Rcplex (>= 0.3), cccp (>= 0.3), piqp (>= 0.6), scip, xpress, diffcp (>= 0.1.2), ipopt, sparsediff (>= 0.6.1), Uno

Additional_repositories https://bnaras.r-universe.dev, https://bnaras.github.io/drat

LinkingTo Rcpp, RcppEigen

License Apache License 2.0 | file LICENSE

LazyData true

Encoding UTF-8

VignetteBuilder knitr

Config/testthat/edition 3

Config/roxygen2/version 8.1.0

NeedsCompilation yes

Author Anqi Fu [aut, cre], Balasubramanian Narasimhan [aut], Steven Diamond [aut], John Miller [aut], Stephen Boyd [ctb]

Maintainer Anqi Fu <anqif@alumni.stanford.edu>

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

Date/Publication 2026-10-19 15:40:02 UTC

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

RemoteRef HEAD

RemoteSha e2484c78fc7966c98b37b2c387902645f22dd12b

Contents

%>>% . . . 6
%<<% . . . 7
And . . . 8
as_cvxr_expr . . . 8
available_solvers . . . 9
backward . . . 10
bmat . . . 11
CallbackParam . . . 11
cdiac . . . 12
ceil_expr . . . 13
condition_number . . . 14
Constant . . . 14
constants . . . 15
constraints . . . 16
conv . . . 16
convolve . . . 17
cummax_expr . . . 17
cumsum_axis . . . 18
curvature . . . 18
curvatures . . . 19
cvar . . . 20
cvxr_diff . . . 20
cvxr_mean . . . 21
cvxr_norm . . . 21
cvxr_outer . . . 22
cvxr_power . . . 22
cvxr_std . . . 23
cvxr_var . . . 24
deep_flatten . . . 24
delta . . . 25
derivative . . . 25

DiagMat . . . 26
DiagVec . . . 27
diff_pos . . . 27
dist_ratio . . . 28
dotsort . . . 28
dspop . . . 29
dssamp . . . 29
dual_value . . . 30
entr . . . 30
Equality . . . 31
ExpCone . . . 31
expr_H . . . 32
expr_sign . . . 32
eye_minus_inv . . . 33
FiniteSet . . . 33
floor_expr . . . 34
format_labeled . . . 34
gen_lambda_max . . . 35
geo_mean . . . 35
get_bounds . . . 36
get_problem_data . . . 37
gmatmul . . . 38
gradient . . . 38
harmonic_mean . . . 39
hstack . . . 39
huber . . . 40
id . . . 40
iff . . . 41
implies . . . 41
indicator . . . 42
Inequality . . . 43
installed_solvers . . . 43
inv_pos . . . 44
inv_prod . . . 44
is_affine . . . 45
is_atom_smooth . . . 45
is_concave . . . 46
is_constant . . . 46
is_convex . . . 47
is_dcp . . . 47
is_dgp . . . 48
is_diag . . . 48
is_dnlp . . . 49
is_dpp . . . 49
is_dqcp . . . 50
is_linearizable_concave . . . 50
is_linearizable_convex . . . 51
is_log_log_affine . . . 51

is_log_log_concave . . . 52
is_log_log_convex . . . 52
is_lp . . . 53
is_matrix . . . 53
is_mixed_integer . . . 54
is_nonneg . . . 54
is_nonpos . . . 55
is_nsd . . . 55
is_psd . . . 56
is_pwl . . . 56
is_qp . . . 57
is_quadratic . . . 57
is_quasiconcave . . . 58
is_quasiconvex . . . 58
is_quasilinear . . . 59
is_scalar . . . 59
is_smooth . . . 60
is_symmetric . . . 60
is_vector . . . 61
is_zero . . . 61
kl_div . . . 62
kron . . . 62
label . . . 63
label<- . . . 63
lambda_max . . . 64
lambda_min . . . 64
lambda_sum_largest . . . 65
lambda_sum_smallest . . . 65
length_expr . . . 66
log_det . . . 66
log_log_curvature . . . 67
log_normcdf . . . 67
log_sum_exp . . . 68
log1p_atom . . . 68
loggamma . . . 69
logistic . . . 69
make_sparse_diagonal_matrix . . . 70
math_atoms . . . 70
matrix_frac . . . 73
matrix_trace . . . 73
max_elemwise . . . 74
max_entries . . . 74
Maximize . . . 75
min_elemwise . . . 75
min_entries . . . 76
Minimize . . . 76
mixed_norm . . . 77
multiply . . . 77

name . . . 78
neg . . . 78
NonNeg . . . 79
NonPos . . . 79
norm_inf . . . 80
norm_nuc . . . 80
norm1 . . . 81
norm2 . . . 81
normcdf . . . 82
Not . . . 82
objective . . . 83
one_minus_pos . . . 83
Or . . . 84
p_norm . . . 85
param_dict . . . 85
Parameter . . . 86
parameters . . . 87
partial_optimize . . . 87
partial_trace . . . 88
partial_transpose . . . 89
perspective . . . 90
pf_eigenvalue . . . 90
pos . . . 91
PowCone3D . . . 91
PowConeND . . . 92
Problem . . . 93
problem_data . . . 93
problem_solution . . . 94
problem_status . . . 95
problem_unpack_results . . . 95
prod_entries . . . 96
PSD . . . 97
psolve . . . 97
ptp . . . 99
quad_form . . . 99
quad_over_lin . . . 100
reduction-chain-rule . . . 100
reduction-id-map . . . 101
rel_entr . . . 102
reshape_expr . . . 102
residual . . . 103
resolvent . . . 103
sample_bounds . . . 104
scalar_product . . . 104
scalarize . . . 105
scalene . . . 106
set_label . . . 106
sigma_max . . . 107

size . . . 107
size_metrics . . . 108
SizeMetrics . . . 108
SOC . . . 109
solution . . . 110
solve_via_data . . . 110
solver-constants . . . 111
solver_default_param . . . 112
solver_opts . . . 113
solver_stats . . . 114
square . . . 114
status . . . 115
status-constants . . . 115
sum_entries . . . 116
sum_largest . . . 117
sum_smallest . . . 117
sum_squares . . . 118
to_latex . . . 118
total_variation . . . 119
tr_inv . . . 119
tv . . . 120
unpack_results . . . 120
upper_tri . . . 121
value . . . 121
value<- . . . 122
var_dict . . . 122
Variable . . . 123
variables . . . 124
vdot . . . 124
vec . . . 125
vec_to_upper_tri . . . 125
violation . . . 126
visualize . . . 126
vstack . . . 127
xexp . . . 128
Xor . . . 128
Zero . . . 129

Index **130**

`%>>%` *Positive Semidefinite Constraint Operator*

Description

Creates a PSD constraint: `e1 - e2` is positive semidefinite. This is the R equivalent of Python's `A >> B`.

Usage

```
e1 %>>% e2
```

Arguments

e1, e2	CVXR expressions or numeric matrices.

Value

A PSD constraint object.

See Also

PSD()

Examples

```
X <- Variable(c(3, 3), symmetric = TRUE)
constr <- X %>>% diag(3)  # X - I is PSD
```

%<<% *Negative Semidefinite Constraint Operator*

Description

Creates an NSD constraint: e2 - e1 is positive semidefinite, i.e., e1 is NSD relative to e2. This is the R equivalent of Python's A << B.

Usage

```
e1 %<<% e2
```

Arguments

e1, e2	CVXR expressions or numeric matrices.

Value

A PSD constraint object.

See Also

PSD()

Examples

```
X <- Variable(c(3, 3), symmetric = TRUE)
constr <- X %<<% diag(3)  # I - X is PSD (X is NSD relative to I)
```

And *Logical AND*

Description

Returns 1 if and only if all arguments equal 1, and 0 otherwise. For two operands, can also be written with the `&` operator: `x & y`.

Usage

```
And(..., id = NULL)
```

Arguments

`...`	Two or more boolean Variables or logic expressions.
`id`	Optional integer ID (internal use).

Value

An `And` expression.

See Also

`Not()`, `Or()`, `Xor()`, `implies()`, `iff()`

Examples

```
x <- Variable(boolean = TRUE)
y <- Variable(boolean = TRUE)
z <- Variable(boolean = TRUE)
both <- x & y            # operator syntax
both <- And(x, y)        # functional syntax
all3 <- And(x, y, z)     # n-ary
```

as_cvxr_expr *Convert a value to a CVXR Expression*

Description

Wraps numeric vectors, matrices, and Matrix package objects as CVXR Constant objects. Values that are already CVXR expressions are returned unchanged.

Usage

```
as_cvxr_expr(x)
```

Arguments

x	A numeric vector, matrix, Matrix::Matrix object, Matrix::sparseVector object, or CVXR expression.

Value

A CVXR expression (either the input unchanged or wrapped in Constant).

Matrix package interoperability

Objects from the **Matrix** package (`dgCMatrix`, `dgeMatrix`, `ddiMatrix`, `sparseVector`, etc.) are S4 classes. Because S4 dispatch preempts S7/S3 dispatch, **raw Matrix objects cannot be used directly with CVXR operators** (`+`, `-`, `*`, `/`, `%*%`, `>=`, `==`, etc.).

Use `as_cvxr_expr()` to wrap a Matrix object as a CVXR Constant before combining it with CVXR variables or expressions. This preserves sparsity (unlike `as.matrix()`, which densifies).

Base R `matrix` and `numeric` objects work natively with CVXR operators — no wrapping is needed.

Examples

```
x <- Variable(3)

## Sparse Matrix needs as_cvxr_expr() for CVXR operator dispatch:
A <- Matrix::sparseMatrix(i = 1:3, j = 1:3, x = 1.0)
expr <- as_cvxr_expr(A) %*% x

## All operators work with wrapped Matrix objects:
y <- Variable(c(3, 3))
expr2 <- as_cvxr_expr(A) + y
constr <- as_cvxr_expr(A) >= y

## Base R matrix works natively (no wrapping needed):
D <- matrix(1:9, 3, 3)
expr3 <- D %*% x
```

`available_solvers` *List available solvers*

Description

Returns the names of installed solvers that are not currently excluded. Use `exclude_solvers()` to temporarily disable solvers.

Usage

```
available_solvers()

exclude_solvers(solvers)

include_solvers(solvers)

set_excluded_solvers(solvers)
```

Arguments

solvers	A character vector of solver names.

Value

A character vector of solver names.

The current exclusion list (character vector), invisibly.

Functions

- `exclude_solvers()`: Add solvers to the exclusion list
- `include_solvers()`: Remove solvers from the exclusion list
- `set_excluded_solvers()`: Replace the entire exclusion list

See Also

`installed_solvers()`, `exclude_solvers()`, `include_solvers()`, `set_excluded_solvers()`

---

backward *Compute the gradient of a solution with respect to Parameters*

Description

Differentiates through the solution map of `problem`: populates the `gradient` slot of each `Parameter` with the sensitivity of a scalar-valued function of the variables (defaulting to the sum-of-x loss; override per variable by setting `gradient(variable) <-` before calling) with respect to that parameter. Mirrors `cvxpy.Problem.backward()`.

Usage

```
backward(problem)
```

Arguments

problem	A solved `Problem`.

Details

Must be called after `psolve()` with `requires_grad = TRUE`.

Value

The `problem` (for piping); side-effect sets `gradient(param)` on each parameter.

See Also

`derivative()`, `psolve()`, `gradient()`

---

bmat *Construct a Block Matrix*

Description

Takes a list of lists. Each internal list is stacked horizontally. The internal lists are stacked vertically.

Usage

```
bmat(block_lists)
```

Arguments

`block_lists`	A list of lists of Expression objects (or numerics). Each inner list forms one block row.

Value

An Expression representing the block matrix.

CallbackParam *Callback Parameter*

Description

A `Parameter` whose value is computed by a user-supplied callback function rather than stored explicitly. Mirrors CVXPY's `cp.CallbackParam` (`cvxpy/expressions/constants/callback_param.py`).

Usage

```
CallbackParam(
  callback,
  shape = c(1L, 1L),
  name = NULL,
  id = NULL,
  latex_name = NULL,
  ...
)
```

Arguments

callback	A function (no arguments) returning the parameter's numeric value. Re-evaluated on every read of value(param).
shape	Integer vector of length 1 or 2 giving parameter dimensions. Defaults to c(1, 1) (scalar).
name	Optional character string name.
id	Optional integer ID. If NULL, a unique ID is generated.
latex_name	Optional LaTeX name for visualisation.
...	Other Parameter attributes (e.g., nonneg, nonpos).

Details

Each call to value(param) re-evaluates the callback and validates the returned numeric against the parameter's shape and attribute domain. Setting via value(param) <- v is not allowed and signals an error.

Value

A CallbackParam object (subclass of Parameter).

DPP use

If p and q are scalar Parameters, the expression p * q is not DPP. Wrapping it in a CallbackParam,

```
pq <- CallbackParam(callback = function() value(p) * value(q))
```

yields a DPP-compliant Parameter whose value tracks p and q automatically.

Examples

```
p <- Parameter(); value(p) <- 2
q <- Parameter(); value(q) <- 3
pq <- CallbackParam(callback = function() value(p) * value(q))
value(pq)   # evaluates the callback => 6
```

---

cdiac — *Global Monthly and Annual Temperature Anomalies (degrees C), 1850-2015 (Relative to the 1961-1990 Mean) (May 2016)*

Description

Global Monthly and Annual Temperature Anomalies (degrees C), 1850-2015 (Relative to the 1961-1990 Mean) (May 2016)

Usage

```
cdiac
```

Format

A data frame with 166 rows and 14 variables:

year Year

jan Anomaly for month of January

feb Anomaly for month of February

mar Anomaly for month of March

apr Anomaly for month of April

may Anomaly for month of May

jun Anomaly for month of June

jul Anomaly for month of July

aug Anomaly for month of August

sep Anomaly for month of September

oct Anomaly for month of October

nov Anomaly for month of November

dec Anomaly for month of December

annual Annual anomaly for the year

Source

https://ess-dive.lbl.gov/

References

https://ess-dive.lbl.gov/

---

ceil_expr *Elementwise Ceiling*

Description

Returns the ceiling (smallest integer >= x) of each element. This atom is quasiconvex and quasiconcave but NOT convex or concave, so it can only be used in DQCP problems (solved with `qcp = TRUE`). As in CVXPY, its value rounds `x` to 4 decimal places first, so `ceiling(3e-10)` of an expression is 0, not 1.

Usage

```
ceil_expr(x)
```

Arguments

x	A CVXR expression.

Value

A `Ceil` expression.

See Also

`floor_expr`

---

condition_number *Condition number of a PSD matrix*

Description

Computes the condition number lambda_max(A) / lambda_min(A) for a positive semidefinite matrix A. This is a quasiconvex atom.

Usage

```
condition_number(A)
```

Arguments

A	A square matrix expression (must be PSD)

Value

An expression representing the condition number of A

Constant *Create a Constant Expression*

Description

Wraps a numeric value as a CVXR constant for use in optimization expressions. Constants are typically created implicitly when combining numeric values with CVXR expressions via arithmetic operators.

Usage

```
Constant(value, name = NULL)
```

Arguments

value	A numeric scalar, vector, matrix, or sparse matrix.
name	Optional character string name.

Value

A Constant object (inherits from Leaf and Expression).

Examples

```
c1 <- Constant(5)
c2 <- Constant(matrix(1:6, 2, 3))
```

---

constants *Get the Constants in an Expression*

Description

Get the Constants in an Expression

Usage

```
constants(x, ...)
```

Arguments

x	An expression or problem object.
...	Not used.

Value

List of Constant objects.

constraints *Get Problem Constraints (read-only)*

Description

Returns a copy of the problem's constraint list.

Usage

```
constraints(x)
```

Arguments

x	A Problem object.

Details

Problem objects are **immutable**: constraints cannot be modified after construction. To change constraints, create a new `Problem()`. This matches CVXPY's design where problems are immutable except through Parameter value changes.

Value

A list of constraint objects.

See Also

`Problem()`, `objective()`

Examples

```
x <- Variable(2)
prob <- Problem(Minimize(sum_entries(x)), list(x >= 1))
length(constraints(prob))  # 1
```

conv *1D discrete convolution*

Description

1D discrete convolution

Usage

```
conv(a, b)
```

Arguments

a	An Expression (vector, one must be constant)
b	An Expression (vector)

Value

A Convolve atom

---

convolve *1D discrete convolution (numpy-style)*

Description

`convolve()` is CVXPY 1.9's preferred name for `conv` (CVXPY's `conv` class is deprecated in favor of `convolve`). For a CVXR `Expression` argument it builds a `Convolve` atom, the numpy-style convolution (`numpy.convolve(x, y)` == `stats::convolve(x, rev(y), type = "open")`). If neither argument is an Expression the call is forwarded as written to `convolve`, so the result is exactly **stats**' (circular by default).

Usage

```
convolve(x, y, conj = TRUE, type = c("circular", "open", "filter"))
```

Arguments

x, y	Expressions or numeric vectors; at least one must be constant when building an atom.
conj, type	As in `convolve`; only for plain numeric input, an error with an Expression.

Value

A `Convolve` atom (for expressions) or a numeric vector.

---

cummax_expr *Cumulative maximum along an axis*

Description

Cumulative maximum along an axis

Usage

```
cummax_expr(x, axis = 2L)
```

Arguments

x	An Expression
axis	1 (across rows) or 2 (down columns, default)

Value

A Cummax atom

---

cumsum_axis *Cumulative sum along an axis*

Description

Cumulative sum along an axis

Usage

```
cumsum_axis(x, axis = 2L)
```

Arguments

x	An Expression
axis	2 (down columns, the default, as CVXPY's `axis = 0`), 1 (across rows), or NULL (all entries, in column-major order, as a column)

Value

A Cumsum atom

---

curvature *Get Expression Curvature*

Description

Returns the curvature of an expression as a string.

Usage

```
curvature(x)
```

Arguments

x	A CVXR expression.

Value

Character: the first curvature that holds, checked in the order `"CONSTANT"`, `"AFFINE"`, `"CONVEX"`, `"CONCAVE"`, `"LOG-LOG AFFINE"`, `"LOG-LOG CONVEX"`, `"LOG-LOG CONCAVE"`, `"QUASILINEAR"`, `"QUASICONVEX"`, `"QUASICONCAVE"`; `"UNKNOWN"` if none does.

See Also

`curvatures()`, `log_log_curvature()`, `is_convex()`, `is_concave()`, `is_affine()`, `is_constant()`

---

curvatures *All Curvatures of an Expression*

Description

Lists every curvature that holds for an expression, DCP, DGP and DQCP alike, as CVXPY's `Expression.curvatures`. `curvature` reports only the strongest DCP curvature.

Usage

```
curvatures(x)
```

Arguments

`x`	A CVXR expression.

Value

A character vector, in the order `"CONSTANT"`, `"AFFINE"`, `"CONVEX"`, `"CONCAVE"`, `"LOG-LOG CONSTANT"`, `"LOG-LOG AFFINE"`, `"LOG-LOG CONVEX"`, `"LOG-LOG CONCAVE"`, `"QUASILINEAR"`, `"QUASICONVEX"`, `"QUASICONCAVE"`, holding those that apply; `"UNKNOWN"` alone if none does.

See Also

`curvature()`, `log_log_curvature()`

cvar — *Conditional Value at Risk (CVaR)*

Description

CVaR at confidence level beta: average of the (1-beta) largest values.

Usage

```
cvar(x, beta)
```

Arguments

x	An Expression (vector)
beta	Confidence level in [0, 1)

Value

An Expression representing the CVaR

cvxr_diff — *Compute kth Order Differences of an Expression*

Description

Takes in an expression and returns an expression with the kth order differences along the given axis. The output shape is the same as the input except the size along the specified axis is reduced by k.

Usage

```
cvxr_diff(x, k = 1L, axis = 2L)
```

Arguments

x	An Expression or numeric value.
k	Integer. The number of times values are differenced. Default is 1. (Mapped from R's `lag` argument in diff.default; use `differences` for repeated differencing which maps to k here.)
axis	Integer. The axis along which the difference is taken. 2 = along rows/down columns (default), 1 = along columns/across rows.

Value

An Expression representing the kth order differences.

cvxr_mean *Mean of an expression*

Description

Computes the arithmetic mean of an expression along an axis.

Usage

```
cvxr_mean(x, axis = NULL, keepdims = FALSE)
```

Arguments

x	An Expression or numeric value.
axis	NULL (all), 1 (row-wise), or 2 (column-wise).
keepdims	Logical; keep reduced dimension?

Value

An Expression representing the mean.

cvxr_norm *Compute a norm of an expression*

Description

With `axis = NULL`, `p = 1`, `2` and `Inf` are the matrix norms, as in CVXPY and base R `norm()`: the maximum absolute column sum, the largest singular value (`sigma_max`) and the maximum absolute row sum. A row vector is a 1 x n matrix, so its 1-norm is its largest absolute entry and its Inf-norm the sum of absolute entries. For a column vector these equal the entrywise norms. Any other `p` is an error for a matrix or row vector, as in CVXPY ("Unsupported matrix norm."); use `p_norm` for the entrywise p-norm. On a column vector, and with an `axis`, it is the entrywise norm. The masking `norm()` gives the same results under base R's type names.

Usage

```
cvxr_norm(x, p = 2, axis = NULL, keepdims = FALSE, max_denom = 1024L)
```

Arguments

x	An Expression
p	Norm type: 1, 2, Inf, "fro" (Frobenius), or "nuc" (nuclear norm of a matrix). "fro" and "nuc" are whole-matrix norms and take no `axis`.
axis	NULL (all), 0 (columns), or 1 (rows)
keepdims	Logical
max_denom	Integer max denominator

Value

A norm atom

`cvxr_outer` *Outer product of two vectors*

Description

Computes the outer product `x %*% t(y)`. Both inputs must be vectors. Named `cvxr_outer()` because base R already exports `outer()`.

Usage

```
cvxr_outer(x, y)
```

Arguments

`x`	An Expression or numeric value (vector).
`y`	An Expression or numeric value (vector).

Value

An Expression of shape (length(x), length(y)).

See Also

`vdot()`

`cvxr_power` *Create a Power atom*

Description

CVXPY's `power(x, p)`: `x` raised to the power `p`, for any argument (a plain number becomes a constant). `cvxr_power()` is the preferred name. `power()` is the same function, kept because existing code uses it; it masks `power`, the unrelated `glm()` link constructor, so with CVXR attached write `stats::power()` in a `glm()` family. `x^p` is the same atom for an expression `x`.

Usage

```
cvxr_power(x, p, max_denom = 1024L, approx = TRUE)

power(x, p, max_denom = 1024L, approx = TRUE)
```

Arguments

x	An Expression (the base), OR a positive constant if p is a variable (the identity b^x = exp(x * log(b)) is used).
p	Numeric exponent, Parameter, or Expression. If p is a non-constant Expression and x is a positive constant, dispatches to exp(p * log(x)).
max_denom	Maximum denominator for rational approximation
approx	If TRUE (default), use SOC approximation. If FALSE, use exact power cone.

Value

A Power or PowerApprox atom, or an exp expression for the const-base case.

Note

sqrt(x) on a CVXR expression dispatches to Power(x, 0.5) via the Math group generic. See math_atoms for all standard R function dispatch.

---

cvxr_std *Standard deviation of an expression*

Description

Computes the standard deviation of an expression.

Usage

```
cvxr_std(x, axis = NULL, keepdims = FALSE, ddof = 0)

std(x, axis = NULL, keepdims = FALSE, ddof = 0)
```

Arguments

x	An Expression or numeric value.
axis	NULL (all), 1 (row-wise), or 2 (column-wise).
keepdims	Logical; keep reduced dimension?
ddof	Degrees of freedom correction (default 0, population std).

Value

An Expression representing the standard deviation.

cvxr_var *Variance of an expression*

Description

Computes the variance of an expression, over all entries or along an axis.

Usage

```
cvxr_var(x, axis = NULL, keepdims = FALSE, ddof = 0)
```

Arguments

x	An Expression or numeric value.
axis	NULL (all), 1 (row-wise), or 2 (column-wise).
keepdims	Logical; keep reduced dimension?
ddof	Degrees of freedom correction (default 0, population variance).

Value

An Expression representing the variance.

deep_flatten *Recursively flatten a nested list of expressions into one column vector*

Description

Flattens `x` into a single column vector. An Expression or numeric is vectorized column-major; a list is flattened element by element and the pieces stacked in order, recursively. This is what lets `vdot()` accept nested lists: `vdot(list(a, b), c(1, 2))` is `a * 1 + b * 2`.

Usage

```
deep_flatten(x)
```

Arguments

x	An Expression, a numeric value, or a (possibly nested) list of them.

Value

An Expression of shape `c(n, 1)`.

See Also

`vec()`, `vdot()`

Examples

```
a <- Variable(); b <- Variable()
deep_flatten(list(a, b))
```

---

delta *Access the perturbation* delta *of a Variable or Parameter*

Description

Used by psolve() with requires_grad = TRUE and Problem$derivative(). On a Parameter, the user sets delta to a perturbation of the parameter's value; derivative() then reports the predicted change in each Variable's optimal value as delta(variable).

Usage

```
delta(x)

delta(x) <- value
```

Arguments

x	A Variable or Parameter.
value	A numeric array of the same shape as x, or NULL.

Value

The perturbation (numeric array) or NULL.

---

derivative *Apply the derivative of the solution map to perturbations*

Description

Forward-mode counterpart of backward(): reads delta(param) for each parameter, applies the cone-program derivative, and writes the predicted change in each variable's optimum to delta(var). Mirrors cvxpy.Problem.derivative().

Usage

```
derivative(problem)
```

Arguments

problem	A solved Problem.

Details

Must be called after `psolve()` with `requires_grad = TRUE`.

Value

The `problem` (for piping); side-effect sets `delta(variable)` on each variable.

See Also

`backward()`, `psolve()`, `delta()`

DiagMat *Extract Diagonal from a Matrix*

Description

Extracts the k-th diagonal of a square matrix as a column vector.

Usage

```
DiagMat(x, k = 0L, id = NULL)
```

Arguments

`x`	A CVXR expression (square matrix).
`k`	Integer diagonal offset. `k = 0` (default) is the main diagonal, `k > 0` is above, `k < 0` is below.
`id`	Optional integer ID.

Value

A `DiagMat` expression of shape `c(n - abs(k), 1)`.

See Also

`DiagVec`

DiagVec *Vector to Diagonal Matrix*

Description

Constructs a diagonal matrix from a column vector. If `k != 0`, the vector is placed on the `k`-th super- or sub-diagonal.

Usage

```
DiagVec(x, k = 0L, id = NULL)
```

Arguments

`x`	A CVXR expression (column vector).
`k`	Integer diagonal offset. `k = 0` (default) is the main diagonal, `k > 0` is above, `k < 0` is below.
`id`	Optional integer ID.

Value

A `DiagVec` expression of shape `c(n + abs(k), n + abs(k))`.

See Also

`DiagMat`

diff_pos *The difference x - y with domain x > y > 0*

Description

Equivalent to x * one_minus_pos(y / x).

Usage

```
diff_pos(x, y)
```

Arguments

`x`	An Expression (positive)
`y`	An Expression (positive, elementwise less than x)

Value

A product expression

dist_ratio *Distance ratio*

Description

Computes norm(x - a)_2 / norm(x - b)_2, where a and b are constants. This is a quasiconvex atom.

Usage

```
dist_ratio(x, a, b)
```

Arguments

x	A vector expression
a	A numeric constant vector
b	A numeric constant vector

Value

An expression representing the distance ratio

dotsort *Weighted sorted dot product*

Description

Computes <sort(vec(X)), sort(vec(W))> where X is an expression and W is a constant. A generalization of sum_largest and sum_smallest.

Usage

```
dotsort(X, W)
```

Arguments

X	An Expression or numeric value.
W	A constant numeric vector or matrix.

Value

A scalar convex Expression.

---

dspop *Direct Standardization: Population*

Description

Randomly generated data for direct standardization example. Sex was drawn from a Bernoulli distribution, and age was drawn from a uniform distribution on $10, \ldots, 60$. The response was drawn from a normal distribution with a mean that depends on sex and age, and a variance of 1.

Usage

```
dspop
```

Format

A data frame with 1000 rows and 3 variables:

y Response variable

sex Sex of individual, coded male (0) and female (1)

age Age of individual

See Also

dssamp

---

dssamp *Direct Standardization: Sample*

Description

A sample of dspop for direct standardization example. The sample is skewed such that young males are overrepresented in comparison to the population.

Usage

```
dssamp
```

Format

A data frame with 100 rows and 3 variables:

y Response variable

sex Sex of individual, coded male (0) and female (1)

age Age of individual

See Also

dspop

dual_value *Get the Dual Value of a Constraint*

Description

Returns the dual variable value(s) associated with a constraint after solving. Returns `NULL` before the problem is solved.

Usage

```
dual_value(x)
```

Arguments

`x`	A `Constraint` object.

Value

A numeric matrix (single dual variable) or a list of numeric matrices (multiple dual variables), or `NULL`.

entr *Create an entropy atom -x * log(x)*

Description

Create an entropy atom -x * log(x)

Usage

```
entr(x)
```

Arguments

`x`	An Expression

Value

An Entr atom

Equality *Create an Equality Constraint*

Description

Constrains two expressions to be equal elementwise: $lhs = rhs$. Typically created via the == operator on CVXR expressions.

Usage

```
Equality(lhs, rhs, constr_id = NULL)
```

Arguments

lhs	A CVXR expression (left-hand side).
rhs	A CVXR expression (right-hand side).
constr_id	Optional integer constraint ID.

Value

An Equality constraint object.

See Also

Zero, Inequality

ExpCone *Create an Exponential Cone Constraint*

Description

Constrains (x, y, z) to lie in the exponential cone:

$$K = \{(x, y, z) \mid y \exp(x/y) \leq z,\ y > 0\}$$

Usage

```
ExpCone(x_expr, y_expr, z_expr, constr_id = NULL)
```

Arguments

x_expr	A CVXR expression.
y_expr	A CVXR expression.
z_expr	A CVXR expression.
constr_id	Optional integer constraint ID.

Details

All three arguments must be affine, real, and have the same shape.

Value

An ExpCone constraint object.

---

expr_H *Conjugate-Transpose of an Expression*

Description

Equivalent to CVXPY's .H property. For real expressions, returns t(x). For complex expressions, returns Conj(t(x)).

Usage

```
expr_H(x)
```

Arguments

x	A CVXR Expression.

Value

The conjugate-transpose expression.

---

expr_sign *Get the DCP Sign of an Expression*

Description

Returns the sign of an expression under DCP analysis. Use this instead of sign(), which conflicts with the base R function.

Usage

```
expr_sign(x, ...)
```

Arguments

x	An expression object.
...	Not used.

Value

Character string: "POSITIVE", "NEGATIVE", "ZERO", or "UNKNOWN".

eye_minus_inv *Unity resolvent (I - X) inverse for positive square matrix X*

Description

Log-log convex atom for DGP. Solve with `psolve(problem, gp = TRUE)`.

Usage

```
eye_minus_inv(X)
```

Arguments

X	An Expression (positive square matrix with spectral radius < 1)

Value

An EyeMinusInv atom

Examples

```
X <- Variable(c(2, 2), pos = TRUE)
prob <- Problem(Minimize(sum(eye_minus_inv(X))), list(X <= 0.4))
psolve(prob, gp = TRUE, solver = "SCS")
```

FiniteSet *FiniteSet Constraint*

Description

Constrain each entry of an Expression to take a value in a given finite set of real numbers.

Usage

```
FiniteSet(expre, vec, ineq_form = FALSE, constr_id = NULL)
```

Arguments

expre	An affine Expression.
vec	A numeric vector (or set) of allowed values.
ineq_form	Logical; controls MIP canonicalization strategy. If FALSE (default), uses equality formulation (one-hot binary). If TRUE, uses inequality formulation (sorted differences + ordering).
constr_id	Optional integer constraint ID (internal use).

Value

A `FiniteSet` constraint.

floor_expr *Elementwise Floor*

Description

Returns the floor (largest integer <= x) of each element. This atom is quasiconvex and quasiconcave but NOT convex or concave, so it can only be used in DQCP problems (solved with `qcp = TRUE`). As in CVXPY, its value rounds `x` to 4 decimal places first, so `floor(-3e-10)` of an expression is 0, not -1.

Usage

```
floor_expr(x)
```

Arguments

`x`	A CVXR expression.

Value

A `Floor` expression.

See Also

`ceil_expr`

format_labeled *Pretty-print an expression with labels substituted*

Description

Recursive analogue of `expr_name()` that substitutes user-supplied labels (see `set_label()`) for sub-expressions wherever they are set, falling back to the structural name on unlabelled nodes. Mirrors CVXPY's `Expression.format_labeled`.

Usage

```
format_labeled(x)
```

Arguments

`x`	An Expression object.

Value

A character string.

See Also

set_label(), label()

---

gen_lambda_max *Maximum generalized eigenvalue*

Description

Computes the maximum generalized eigenvalue lambda_max(A, B). Requires A symmetric and B positive semidefinite. This is a quasiconvex atom.

Usage

```
gen_lambda_max(A, B)
```

Arguments

A	A square symmetric matrix expression
B	A square PSD matrix expression of the same dimension as A

Value

An expression representing the maximum generalized eigenvalue

---

geo_mean *(Weighted) geometric mean*

Description

The (weighted) geometric mean of the entries of x, of all of them or of each row or column.

Usage

```
geo_mean(
  x,
  p = NULL,
  max_denom = 1024L,
  approx = TRUE,
  axis = NULL,
  keepdims = FALSE
)
```

Arguments

x	An Expression of any shape. With `axis = NULL` its entries are taken in column-major order, so a row vector, a column vector and a matrix holding the same entries give the same result.
p	Numeric weight vector (default: uniform), one weight per entry being reduced: `length(x)` entries with `axis = NULL`, `ncol(x)` with `axis = 1`, `nrow(x)` with `axis = 2`. Entries with zero weight take no part.
max_denom	Maximum denominator for rational approximation
approx	If TRUE (default), use SOC approximation. If FALSE, use exact power cone.
axis	`NULL` (default) for one geometric mean of all entries, `1` for one per row, `2` for one per column.
keepdims	Keep the reduced dimension with length one.

Value

A GeoMean or GeoMeanApprox atom: a scalar when `axis = NULL`, otherwise one geometric mean per row or column.

---

get_bounds *Lower/Upper Bounds of a Leaf*

Description

Returns the effective `(lower, upper)` bounds of a leaf, combining its `bounds` attribute with sign (`nonneg`/`pos`/`nonpos`/`neg`) and `boolean` attributes. Used by the NLP (DNLP) solve path to form variable bounds.

Usage

```
get_bounds(x, ...)
```

Arguments

x	An expression (a Variable/leaf, or a composite expression whose bounds are propagated from its arguments).
...	Passed to methods; currently unused.

Value

A list `list(lower, upper)` of two real matrices matching the expression shape (column-major). For leaves these come from explicit bounds and sign attributes; for atoms they are propagated from argument bounds via interval arithmetic (see `bounds_from_args`).

`get_problem_data` *Get Problem Data for a Solver (deprecated)*

Description

[Deprecated]

Usage

```
get_problem_data(
  x,
  solver = NULL,
  gp = FALSE,
  enforce_dpp = FALSE,
  ignore_dpp = FALSE,
  ...
)
```

Arguments

`x`	A `Problem` object.
`solver`	Character string naming solver, or `NULL` for automatic selection.
`gp`	Logical; if `TRUE`, parse the problem as a geometric program.
`enforce_dpp`	Logical; if `TRUE`, raise an error when a parametrized problem is not DPP instead of compiling it as non-DPP.
`ignore_dpp`	Logical; if `TRUE`, treat a DPP problem as non-DPP (skip the DPP fast path).
`...`	Additional solver options.

Details

Use `problem_data` instead.

Value

A list with components `data`, `chain`, and `inverse_data`.

See Also

`problem_data`

gmatmul *Geometric matrix multiplication A diamond X*

Description

Computes the geometric matrix product where (A diamond X)_ij = prod_k X_kj^A_ik. Log-log affine atom for DGP. Solve with `psolve(problem, gp = TRUE)`.

Usage

```
gmatmul(A, X)
```

Arguments

`A`	A constant matrix
`X`	An Expression (positive matrix)

Value

A Gmatmul atom

Examples

```
x <- Variable(2, pos = TRUE)
A <- matrix(c(1, 0, 0, 1), 2, 2)
prob <- Problem(Minimize(sum(gmatmul(A, x))), list(x >= 0.5))
psolve(prob, gp = TRUE)
```

gradient *Access the gradient of a Variable or Parameter*

Description

Used by `psolve()` with `requires_grad = TRUE` and `Problem$backward()`. Stores a numeric array of the same shape as the leaf, or `NULL` (the default).

Usage

```
gradient(x)

gradient(x) <- value
```

Arguments

`x`	A `Variable` or `Parameter`.
`value`	A numeric array of the same shape as `x`, or `NULL`.

Value

The gradient (numeric array) or NULL.

`harmonic_mean` *Harmonic mean: n / sum(1/x_i)*

Description

Harmonic mean: n / sum(1/x_i)

Usage

```
harmonic_mean(x)
```

Arguments

x	An Expression (must be positive for DCP)

Value

An Expression representing the harmonic mean

`hstack` *Horizontal concatenation of expressions*

Description

Named operands are accepted, as in `cbind()`, and the names are dropped.

Usage

```
hstack(..., deparse.level = 1)
```

Arguments

...	Expressions (same number of rows)
deparse.level	Accepted for compatibility with `cbind()`; expressions have no dimnames, so it has no effect

Value

An HStack atom

huber *Create a Huber loss atom*

Description

Create a Huber loss atom

Usage

```
huber(x, M = 1)
```

Arguments

x	An Expression
M	Numeric threshold (default 1)

Value

A Huber atom

id *Get Expression ID*

Description

Returns the unique integer identifier for a CVXR object.

Usage

```
id(x)
```

Arguments

x	A CVXR expression, variable, parameter, or constraint.

Value

An integer.

iff *Logical Biconditional*

Description

Logical biconditional: x <=> y. Returns 1 if and only if x and y have the same value. Equivalent to `Not(Xor(x, y))`.

Usage

```
iff(x, y)
```

Arguments

x, y	Boolean Variables or logic expressions.

Value

A Not expression wrapping Xor.

See Also

`implies()`, `Not()`, `And()`, `Or()`, `Xor()`

Examples

```
x <- Variable(boolean = TRUE)
y <- Variable(boolean = TRUE)
expr <- iff(x, y)
```

implies *Logical Implication*

Description

Logical implication: x => y. Returns 1 unless x = 1 and y = 0. Equivalent to `Or(Not(x), y)`.

Usage

```
implies(x, y)
```

Arguments

x, y	Boolean Variables or logic expressions.

Value

An Or expression representing `!x | y`.

See Also

`iff()`, `Not()`, `And()`, `Or()`, `Xor()`

Examples

```
x <- Variable(boolean = TRUE)
y <- Variable(boolean = TRUE)
expr <- implies(x, y)
```

`indicator` *Indicator function for constraints*

Description

Creates an expression that equals 0 if all constraints are satisfied and +Inf otherwise. Use this to embed constraints into the objective.

Usage

```
indicator(constraints, err_tol = 0.001)
```

Arguments

`constraints`	A list of constraint objects
`err_tol`	Numeric tolerance for checking constraint satisfaction (default 1e-3)

Value

An Indicator expression

Inequality *Create an Inequality Constraint*

Description

Constrains the left-hand side to be less than or equal to the right-hand side elementwise: $lhs \leq rhs$. Typically created via the <= operator on CVXR expressions.

Usage

```
Inequality(lhs, rhs, constr_id = NULL)
```

Arguments

lhs	A CVXR expression (left-hand side).
rhs	A CVXR expression (right-hand side).
constr_id	Optional integer constraint ID.

Value

An Inequality constraint object.

See Also

Equality, NonPos, NonNeg

installed_solvers *List installed solvers*

Description

Returns the names of solvers whose R packages are available.

Usage

```
installed_solvers()
```

Value

A character vector of solver names.

inv_pos *Inverse position: $x^{\wedge}-1$ (for x > 0)*

Description

Inverse position: x^{-1} (for x > 0)

Usage

```
inv_pos(x)
```

Arguments

x	An Expression

Value

A Power atom with p=-1

inv_prod *Reciprocal of product of entries*

Description

Computes the reciprocal of the product of entries. Equivalent to `geo_mean(x)^(-n)` where n is the number of entries.

Usage

```
inv_prod(x, approx = TRUE)
```

Arguments

x	An Expression or numeric value (must have positive entries).
approx	Logical; if TRUE (default), use SOC approximation.

Value

A convex Expression representing the reciprocal product.

is_affine *Check if an Expression is Affine*

Description

Check if an Expression is Affine

Usage

```
is_affine(x, ...)
```

Arguments

x	An expression object.
...	Not used.

Value

Logical scalar.

is_atom_smooth *Check if an Atom is Smooth*

Description

Atom-level hook (default FALSE); smooth atoms (e.g. trig/hyperbolic) override this to TRUE. Mirrors CVXPY's `Atom.is_atom_smooth()`.

Usage

```
is_atom_smooth(x, ...)
```

Arguments

x	An atom object.
...	Not used.

Value

Logical scalar.

See Also

is_smooth

is_concave *Check if an Expression is Concave*

Description

Check if an Expression is Concave

Usage

```
is_concave(x, ...)
```

Arguments

x	An expression object.
...	Not used.

Value

Logical scalar.

is_constant *Check if an Expression is Constant*

Description

Check if an Expression is Constant

Usage

```
is_constant(x, ...)
```

Arguments

x	An expression object.
...	Not used.

Value

Logical scalar.

is_convex *Check if an Expression is Convex*

Description

Check if an Expression is Convex

Usage

```
is_convex(x, ...)
```

Arguments

x	An expression object.
...	Not used.

Value

Logical scalar.

is_dcp *Check if an Expression is DCP-Compliant*

Description

Tests whether an expression follows the Disciplined Convex Programming (DCP) rules.

Usage

```
is_dcp(x, ...)
```

Arguments

x	An expression object.
...	Not used.

Value

Logical scalar.

is_dgp *Check if a Constraint is DGP-Compliant*

Description

Check if a Constraint is DGP-Compliant

Usage

```
is_dgp(x, ...)
```

Arguments

x	A constraint object.
...	Not used.

Value

Logical scalar.

is_diag *Check if a Leaf is Diagonal*

Description

Reports the `diag` attribute of a Variable or Parameter, as CVXPY's `Leaf.is_diag()`. Defined for leaves only.

Usage

```
is_diag(x, ...)
```

Arguments

x	A Variable or Parameter.
...	Not used.

Value

Logical scalar: `TRUE` if `x` was created with `diag = TRUE`.

is_dnlp *Check if an Expression or Problem is DNLP-Compliant*

Description

Tests whether the object follows the Disciplined Nonlinear Programming (DNLP) rules: smooth representable, i.e. linearizable-convex or linearizable-concave.

Usage

```
is_dnlp(x, ...)
```

Arguments

x	An expression, objective, constraint, or Problem.
...	Not used.

Value

Logical scalar.

is_dpp *Check DPP Compliance*

Description

Determines whether an expression or problem satisfies the rules of Disciplined Parameterized Programming (DPP). A DPP-compliant problem enables caching the compilation across parameter value changes.

Usage

```
is_dpp(x, context = "dcp")
```

Arguments

x	An expression, constraint, or problem object.
context	Either "dcp" (default) or "dgp": which discipline to check parameterization against. Mirrors CVXPY's is_dpp(context=...).

Value

Logical scalar.

is_dqcp *Check if Expression is DQCP-Compliant*

Description

Tests whether an expression follows the Disciplined Quasiconvex Programming (DQCP) rules.

Usage

```
is_dqcp(x, ...)
```

Arguments

x	An expression object.
...	Not used.

Value

Logical scalar.

is_linearizable_concave *Check if an Expression is Linearizable-Concave*

Description

Concave after linearizing all smooth subexpressions (DNLP composition rule). Mirrors CVXPY's `Expression.is_linearizable_concave()`.

Usage

```
is_linearizable_concave(x, ...)
```

Arguments

x	An expression object.
...	Not used.

Value

Logical scalar.

See Also

`is_dnlp`, `is_linearizable_convex`

is_linearizable_convex *Check if an Expression is Linearizable-Convex*

Description

Convex after linearizing all smooth subexpressions (DNLP composition rule). Mirrors CVXPY's `Expression.is_linearizable_convex()`.

Usage

```
is_linearizable_convex(x, ...)
```

Arguments

x	An expression object.
...	Not used.

Value

Logical scalar.

See Also

`is_dnlp`, `is_linearizable_concave`

is_log_log_affine *Check if Expression is Log-Log Affine*

Description

Check if Expression is Log-Log Affine

Usage

```
is_log_log_affine(x, ...)
```

Arguments

x	An expression object.
...	Not used.

Value

Logical scalar.

is_log_log_concave *Check if Expression is Log-Log Concave*

Description

Check if Expression is Log-Log Concave

Usage

```
is_log_log_concave(x, ...)
```

Arguments

x	An expression object.
...	Not used.

Value

Logical scalar.

is_log_log_convex *Check if Expression is Log-Log Convex*

Description

Check if Expression is Log-Log Convex

Usage

```
is_log_log_convex(x, ...)
```

Arguments

x	An expression object.
...	Not used.

Value

Logical scalar.

is_lp *Check if a Problem is a Linear Program*

Description

Check if a Problem is a Linear Program

Usage

```
is_lp(x, ...)
```

Arguments

x	A Problem object.
...	Not used.

Value

Logical scalar.

is_matrix *Is the Expression a Matrix?*

Description

Returns TRUE if the expression has both dimensions greater than 1.

Usage

```
is_matrix(x)
```

Arguments

x	A CVXR expression.

Value

Logical.

See Also

size(), is_scalar(), is_vector()

is_mixed_integer *Check if a Problem is Mixed-Integer*

Description

Returns TRUE if any variable in the problem has a boolean or integer attribute.

Usage

```
is_mixed_integer(problem)
```

Arguments

problem	A Problem object.

Value

Logical scalar.

is_nonneg *Check if Expression is Non-Negative*

Description

Check if Expression is Non-Negative

Usage

```
is_nonneg(x, ...)
```

Arguments

x	An expression object.
...	Not used.

Value

Logical scalar.

is_nonpos *Check if Expression is Non-Positive*

Description

Check if Expression is Non-Positive

Usage

```
is_nonpos(x, ...)
```

Arguments

x	An expression object.
...	Not used.

Value

Logical scalar.

is_nsd *Check if Expression is Negative Semidefinite*

Description

Check if Expression is Negative Semidefinite

Usage

```
is_nsd(x, ...)
```

Arguments

x	An expression object.
...	Not used.

Value

Logical scalar.

is_psd *Check if Expression is Positive Semidefinite*

Description

Check if Expression is Positive Semidefinite

Usage

```
is_psd(x, ...)
```

Arguments

x	An expression object.
...	Not used.

Value

Logical scalar.

is_pwl *Check if Expression is Piecewise Linear*

Description

Check if Expression is Piecewise Linear

Is the Expression Piecewise Linear?

Usage

```
is_pwl(x, ...)
```

Arguments

x	A CVXR expression.
...	Not used.

Value

Logical scalar.

Logical.

is_qp *Check if a Problem is a Quadratic Program*

Description

Check if a Problem is a Quadratic Program

Usage

```
is_qp(x, ...)
```

Arguments

x	A Problem object.
...	Not used.

Value

Logical scalar.

is_quadratic *Check if an Expression is Quadratic*

Description

Check if an Expression is Quadratic

Usage

```
is_quadratic(x, ...)
```

Arguments

x	An expression object.
...	Not used.

Value

Logical scalar.

is_quasiconcave *Check if Expression is Quasiconcave*

Description

Check if Expression is Quasiconcave

Usage

```
is_quasiconcave(x, ...)
```

Arguments

x	An expression object.
...	Not used.

Value

Logical scalar.

is_quasiconvex *Check if Expression is Quasiconvex*

Description

Check if Expression is Quasiconvex

Usage

```
is_quasiconvex(x, ...)
```

Arguments

x	An expression object.
...	Not used.

Value

Logical scalar.

is_quasilinear *Check if Expression is Quasilinear*

Description

Check if Expression is Quasilinear

Usage

```
is_quasilinear(x, ...)
```

Arguments

x	An expression object.
...	Not used.

Value

Logical scalar.

is_scalar *Is the Expression a Scalar?*

Description

Is the Expression a Scalar?

Usage

```
is_scalar(x)
```

Arguments

x	A CVXR expression.

Value

Logical.

See Also

size(), is_vector(), is_matrix()

is_smooth *Check if an Expression is Smooth*

Description

Smooth = constant, or both linearizable-convex and linearizable-concave. Mirrors CVXPY's `Expression.is_smooth()`.

Usage

```
is_smooth(x, ...)
```

Arguments

`x`	An expression object.
`...`	Not used.

Value

Logical scalar.

See Also

`is_dnlp`, `is_linearizable_convex`, `is_linearizable_concave`

is_symmetric *Check if Expression is Symmetric*

Description

Check if Expression is Symmetric

Usage

```
is_symmetric(x, ...)
```

Arguments

`x`	An expression object.
`...`	Not used.

Value

Logical scalar.

is_vector *Is the Expression a Vector?*

Description

Returns TRUE if the expression has at most one dimension greater than 1.

Usage

```
is_vector(x)
```

Arguments

x	A CVXR expression.

Value

Logical.

See Also

size(), is_scalar(), is_matrix()

is_zero *Check if Expression is Zero*

Description

Check if Expression is Zero

Usage

```
is_zero(x, ...)
```

Arguments

x	An expression object.
...	Not used.

Value

Logical scalar.

kl_div — *KL Divergence: x*log(x/y) - x + y*

Description

KL Divergence: x*log(x/y) - x + y

Usage

```
kl_div(x, y)
```

Arguments

x	An Expression
y	An Expression

Value

A KlDiv atom

kron — *Kronecker product of two expressions*

Description

Kronecker product of two expressions

Usage

```
kron(a, b)
```

Arguments

a	An Expression (one must be constant)
b	An Expression

Value

A Kron atom

label *Get the label of an expression*

Description

Returns the human-readable label set via `set_label()` (or `label(x) <- ...`), or `NULL` if no label has been set.

Usage

```
label(x)
```

Arguments

`x`	An Expression object.

Value

A length-1 character string, or `NULL`.

See Also

`set_label()`, `format_labeled()`

label<- *Set the label of an expression*

Description

R replacement form of `set_label()`. `label(x) <- value` stores `value` (coerced to character) in `x`'s internal label slot; setting to `NULL` clears the label. Equivalent to `x <- set_label(x, value)`.

Usage

```
label(x) <- value
```

Arguments

`x`	An Expression object.
`value`	A character string, or `NULL` to clear.

Value

`x`, invisibly, with the label updated.

See Also

`set_label()`, `format_labeled()`

lambda_max *Maximum eigenvalue*

Description

Maximum eigenvalue

Usage

```
lambda_max(A)
```

Arguments

A	A square matrix expression

Value

An expression representing the maximum eigenvalue of A

lambda_min *Minimum eigenvalue*

Description

Minimum eigenvalue

Usage

```
lambda_min(A)
```

Arguments

A	A square matrix expression

Value

An expression representing the minimum eigenvalue of A

lambda_sum_largest *Sum of largest k eigenvalues*

Description

Sum of largest k eigenvalues

Usage

```
lambda_sum_largest(A, k)
```

Arguments

A	A square matrix expression
k	Number of largest eigenvalues to sum (positive number; a fractional part weights the next eigenvalue, as in CVXPY)

Value

An expression representing the sum of the k largest eigenvalues

lambda_sum_smallest *Sum of smallest k eigenvalues*

Description

Sum of smallest k eigenvalues

Usage

```
lambda_sum_smallest(A, k)
```

Arguments

A	A square matrix expression
k	Number of smallest eigenvalues to sum (positive number; a fractional part weights the next eigenvalue, as in CVXPY)

Value

An expression representing the sum of the k smallest eigenvalues

length_expr *Length of a Vector (Last Nonzero Index)*

Description

Returns the index of the last nonzero element of a vector (1-based). This atom is quasiconvex but NOT convex, so it can only be used in DQCP problems (solved with `qcp = TRUE`).

Usage

```
length_expr(x)
```

Arguments

`x`	A CVXR vector expression.

Value

A `Length` expression (scalar).

See Also

`ceil_expr`, `floor_expr`

log_det *Log-determinant*

Description

Computes log(det(A)) for PSD matrix A.

Usage

```
log_det(A)
```

Arguments

`A`	A square PSD matrix expression

Value

An expression representing log(det(A))

log_log_curvature *Log-Log Curvature of an Expression*

Description

The DGP analogue of `curvature`, as CVXPY's `Expression.log_log_curvature`.

Usage

```
log_log_curvature(x)
```

Arguments

x	A CVXR expression.

Value

Character: the first of `"LOG-LOG CONSTANT"`, `"LOG-LOG AFFINE"`, `"LOG-LOG CONVEX"`, `"LOG-LOG CONCAVE"` that holds, else `"UNKNOWN"`.

See Also

`curvatures()`, `is_dgp()`

log_normcdf *Elementwise log of the standard normal CDF*

Description

Quadratic approximation of `log(pnorm(x))` with modest accuracy over the range -4 to 4.

Usage

```
log_normcdf(x)
```

Arguments

x	An Expression or numeric value.

Value

A concave Expression representing log(Phi(x)).

log_sum_exp *Log-sum-exp: log(sum(exp(x)))*

Description

Log-sum-exp: log(sum(exp(x)))

Usage

```
log_sum_exp(x, axis = NULL, keepdims = FALSE)
```

Arguments

x	An Expression
axis	NULL (all), 1 (row-wise), or 2 (column-wise)
keepdims	Logical: keep reduced dimensions?

Value

A LogSumExp atom

log1p_atom *Log(1 + x) – elementwise*

Description

Log(1 + x) – elementwise

Usage

```
log1p_atom(x)

log1p_expr(x)
```

Arguments

x	An Expression

Value

A Log1p atom

loggamma — *Elementwise log of the gamma function*

Description

Piecewise linear approximation of `log(gamma(x))`. Has modest accuracy over the full range, approaching perfect accuracy as x goes to infinity.

Usage

```
loggamma(x)
```

Arguments

x	An Expression or numeric value (must be positive for DCP).

Value

A convex Expression representing log(gamma(x)).

logistic — *Logistic function: log(1 + exp(x)) – elementwise*

Description

Logistic function: log(1 + exp(x)) – elementwise

Usage

```
logistic(x)
```

Arguments

x	An Expression

Value

A Logistic atom

make_sparse_diagonal_matrix *Make a CSC sparse diagonal matrix*

Description

Make a CSC sparse diagonal matrix

Usage

```
make_sparse_diagonal_matrix(size, diagonal = NULL)
```

Arguments

size	number of rows or columns
diagonal	if specified, the diagonal values, in which case size is ignored

Value

a compressed sparse column diagonal matrix

math_atoms *Standard R Functions for CVXR Expressions*

Description

CVXR registers methods so that standard R functions create the appropriate atoms when applied to Expression objects.

For CVXR expressions, computes the norm atom with the same meaning as base R norm for every type: a vector is a one-column or one-row matrix. For any other input the call is forwarded as written to Matrix::norm, so the result is exactly base R's (including its default type "O").

For CVXR expressions, computes the standard deviation atom (ddof=0 by default, matching CVXPY/numpy convention). For numeric inputs, falls through to sd.

For CVXR expressions, computes the variance atom (ddof=0 by default). For numeric inputs, falls through to var.

For CVXR expressions, computes the outer product of two vectors. For other inputs, falls through to outer.

For CVXR expressions, dispatches to DiagVec (vector to diagonal matrix) or DiagMat (extract diagonal from matrix), matching CVXPY's cp.diag() behavior. For other inputs, falls through to Matrix::diag, called exactly as written, which dispatches via S4 for both Matrix and base matrix objects.

Usage

```
norm(x, type = "2", ...)

sd(x, ...)

var(x, ...)

outer(X, Y, ...)

diag(x = 1, nrow, ncol, names = TRUE, k = NULL)
```

Arguments

x	An Expression, matrix, vector, or scalar.
type	Norm type, as in base R: "O"/"o"/"1" (maximum absolute column sum), "I"/"i" (maximum absolute row sum), "F"/"f"/"E"/"e" (Frobenius), "M"/"m" (largest absolute entry), "2" (largest singular value; the default for an expression only). For an expression, any other value is passed to cvxr_norm as p.
...	For non-Expression inputs: passed to outer. For expressions, only FUN = "*" is accepted; any other FUN or argument is an error.
X	An Expression or numeric.
Y	An Expression or numeric.
nrow	For non-Expression: passed to Matrix::diag; an error with an Expression.
ncol	For non-Expression: passed to Matrix::diag; an error with an Expression.
names	For non-Expression: passed to Matrix::diag; an error with an Expression.
k	Integer diagonal offset for Expressions only. NULL (default) means 0, the main diagonal.

Details

The k parameter (off-diagonal offset) is only available for Expression inputs; giving it with a non-Expression x is an error.

Value

An Expression or numeric value.

An Expression or numeric value.

An Expression or numeric value.

An Expression or matrix.

An Expression, matrix, or vector.

Math group (elementwise, via S3 group generic)

`abs(x)` Absolute value (convex, nonneg)

`exp(x)` Exponential (convex, positive)

`log(x)` Natural logarithm (concave, domain x >= 0)

`sqrt(x)` Square root via `power(x, 0.5)` (concave)

`log1p(x)` log(1+x) compound expression (concave)

`log2(x)`, `log10(x)` Base-2/10 logarithm

`cumsum(x)` Cumulative sum (affine)

`cummax(x)` Cumulative max (convex)

`cumprod(x)` Cumulative product

`ceiling(x)`, `floor(x)` Round up/down (quasilinear, DQCP). As in CVXPY, the value rounds `x` to 4 decimal places first, so `floor(-3e-10)` of an expression is 0, not -1.

`sign(x)` Sign (quasilinear for a scalar, DQCP). As in CVXPY, it is -1 for `x <= 0`, so `sign(0)` of an expression is -1, not 0.

On a matrix expression, `cumsum()`, `cummax()` and `cumprod()` run down each column, as in CVXPY; use `cumsum_axis(x, axis = NULL)` for all entries in column-major order.

Summary group (via S3 group generic)

`sum(x)` Sum all entries (affine)

`max(x)` Maximum entry (convex)

`min(x)` Minimum entry (concave)

`prod(x)` Product of entries

Each also takes `axis` and `keepdims`, as in CVXPY: `sum(x, axis = 2)` is `sum_entries(x, axis = 2)`, and likewise `max_entries`, `min_entries` and `prod_entries`.

S3 generic methods

`mean(x)` Arithmetic mean; pass `axis`/`keepdims` via `...`

`diff(x)` First-order differences; also `cvxr_diff`

Masking wrappers

These mask the base/stats versions and dispatch on argument type:

`norm(x)` 2-norm; use `type` for "1", "I" (infinity), "F" (Frobenius)

`sd(x)` Standard deviation (ddof=0 for expressions)

`var(x)` Variance (ddof=0 for expressions)

`outer(X, Y)` Outer product of two vector expressions

Advanced usage

For axis-aware reductions, keepdims, or other options not available through the standard interface, use the explicit functions: `cvxr_norm`, `cvxr_mean`, `cvxr_diff`, `cvxr_std`, `cvxr_var`, `cvxr_outer`.

See Also

power, sum_entries, max_entries, min_entries

cvxr_norm for the full-featured version with axis and keepdims arguments

cvxr_std for the full-featured version

cvxr_var for the full-featured version

cvxr_outer for the CVXR-specific version

DiagVec, DiagMat

---

matrix_frac *Matrix fractional function*

Description

Computes $\mathrm{trace}(X^T P^{-1} X)$. If P is a constant matrix, uses a QuadForm shortcut for efficiency.

Usage

```
matrix_frac(X, P)
```

Arguments

X	A matrix expression (n by m)
P	A square matrix expression (n by n), must be PSD

Value

An expression representing $\mathrm{trace}(X^T P^{-1} X)$

---

matrix_trace *Trace of a square matrix expression*

Description

For `matrix_trace(A %*% B)`, uses the O(n^2) identity `trace(A %*% B) = sum(A * t(B))` instead of forming the full matrix product.

Usage

```
matrix_trace(x)
```

Arguments

x	An Expression (square matrix)

Value

A Trace atom or equivalent expression (scalar)

max_elemwise *Elementwise maximum of expressions*

Description

Named operands are accepted, as in `pmax()`, and the names are dropped.

Usage

```
max_elemwise(..., na.rm = FALSE)
```

Arguments

`...`	Expressions (at least 2)
`na.rm`	Accepted for compatibility with `pmax()`; expressions cannot contain `NA`, so it has no effect

Value

A Maximum atom

max_entries *Maximum entry of an expression*

Description

Maximum entry of an expression

Usage

```
max_entries(x, axis = NULL, keepdims = FALSE)
```

Arguments

`x`	An Expression
`axis`	NULL (all), 1 (row-wise), or 2 (column-wise)
`keepdims`	Logical

Value

A MaxEntries atom

Maximize *Create a Maximization Objective*

Description

Specifies that the objective expression should be maximized. The expression must be concave and scalar for a DCP-compliant problem.

Usage

```
Maximize(expr)
```

Arguments

expr	A CVXR expression or numeric value to maximize.

Value

A Maximize object.

See Also

Minimize, Problem

Examples

```
x <- Variable()
obj <- Maximize(-x^2 + 1)
```

min_elemwise *Elementwise minimum of expressions*

Description

Named operands are accepted, as in pmin(), and the names are dropped.

Usage

```
min_elemwise(..., na.rm = FALSE)
```

Arguments

...	Expressions (at least 2)
na.rm	Accepted for compatibility with pmin(); expressions cannot contain NA, so it has no effect

Value

A Minimum atom

`min_entries` *Minimum entry of an expression*

Description

Minimum entry of an expression

Usage

```
min_entries(x, axis = NULL, keepdims = FALSE)
```

Arguments

`x`	An Expression
`axis`	NULL (all), 1 (row-wise), or 2 (column-wise)
`keepdims`	Logical

Value

A MinEntries atom

`Minimize` *Create a Minimization Objective*

Description

Specifies that the objective expression should be minimized. The expression must be convex and scalar for a DCP-compliant problem.

Usage

```
Minimize(expr)
```

Arguments

`expr`	A CVXR expression or numeric value to minimize.

Value

A `Minimize` object.

See Also

Maximize, Problem

Examples

```
x <- Variable()
obj <- Minimize(x^2 + 1)
```

`mixed_norm` *Mixed norm (L_p, q norm): column-wise p-norm, then q-norm*

Description

Mixed norm ($L_{p,q}$ norm): column-wise p-norm, then q-norm

Usage

```
mixed_norm(X, p = 2, q = 1)
```

Arguments

X	An Expression (matrix)
p	Inner norm parameter (default 2)
q	Outer norm parameter (default 1)

Value

An Expression representing the mixed norm

`multiply` *Elementwise multiplication (deprecated)*

Description

[Deprecated]

Usage

```
multiply(x, y)
```

Arguments

x, y	Expressions or numeric values.

Details

Use the * operator instead: x * y.

Value

An Expression representing the elementwise product.

---

name *Get Expression Name*

Description

Returns a human-readable string representation of a CVXR expression, variable, or constraint.

Usage

```
name(x)
```

Arguments

x	A CVXR expression, variable, parameter, constant, or constraint.

Value

A character string.

Examples

```
x <- Variable(2, name = "x")
name(x)  # "x"
name(x + 1)  # "x + 1"
```

---

neg *Negative part: -min(x, 0)*

Description

Negative part: -min(x, 0)

Usage

```
neg(x)
```

Arguments

x	An Expression

Value

A negated Minimum atom

NonNeg *Create a Non-Negative Constraint*

Description

Constrains an expression to be non-negative elementwise: $x \geq 0$.

Usage

```
NonNeg(expr, constr_id = NULL)
```

Arguments

expr	A CVXR expression.
constr_id	Optional integer constraint ID.

Value

A NonNeg constraint object.

See Also

NonPos, Inequality

NonPos *Create a Non-Positive Constraint*

Description

Constrains an expression to be non-positive elementwise: $x \leq 0$.

Usage

```
NonPos(expr, constr_id = NULL)
```

Arguments

expr	A CVXR expression.
constr_id	Optional integer constraint ID.

Value

A NonPos constraint object.

See Also

NonNeg, Inequality

`norm_inf` *L-infinity norm of an expression*

Description

L-infinity norm of an expression

Usage

```
norm_inf(x, axis = NULL, keepdims = FALSE)
```

Arguments

`x`	An Expression
`axis`	NULL (all), 1 (row-wise), or 2 (column-wise)
`keepdims`	Logical: keep reduced dimensions?

Value

A NormInf atom

`norm_nuc` *Nuclear norm (sum of singular values)*

Description

Nuclear norm (sum of singular values)

Usage

```
norm_nuc(A)
```

Arguments

`A`	A matrix expression

Value

An expression representing the nuclear norm of A

norm1 *L1 norm of an expression*

Description

L1 norm of an expression

Usage

```
norm1(x, axis = NULL, keepdims = FALSE)
```

Arguments

x	An Expression
axis	NULL (all), 1 (row-wise), or 2 (column-wise)
keepdims	Logical: keep reduced dimensions?

Value

A Norm1 atom

norm2 *Euclidean norm (deprecated alias)*

Description

[Deprecated]

Usage

```
norm2(x)
```

Arguments

x	An Expression.

Details

Use `p_norm(x, 2)` instead.

Value

An Expression representing the L2 norm.

See Also

`p_norm()`

normcdf *Standard Normal Cumulative Distribution Function*

Description

Elementwise standard normal CDF $\Phi(x)$. A smooth (DNLP) atom: it is neither convex nor concave, so it is usable only on the disciplined-nonlinear- programming path.

Usage

```
normcdf(x)
```

Arguments

x	An Expression.

Value

A Normcdf atom.

Not *Logical NOT*

Description

Returns `1 - x`, flipping 0 to 1 and 1 to 0. Can also be written with the `!` operator: `!x`.

Usage

```
Not(x, id = NULL)
```

Arguments

x	A boolean Variable or logic expression.
id	Optional integer ID (internal use).

Value

A `Not` expression.

See Also

`And()`, `Or()`, `Xor()`, `implies()`, `iff()`

Examples

```
x <- Variable(boolean = TRUE)
not_x <- !x          # operator syntax
not_x <- Not(x)      # functional syntax
```

objective *Get Problem Objective (read-only)*

Description

Returns the problem's objective.

Usage

```
objective(x)
```

Arguments

x	A Problem object.

Details

Problem objects are **immutable**: the objective cannot be modified after construction. To change the objective, create a new `Problem()`.

Value

A Minimize or Maximize object.

See Also

`Problem()`, `constraints()`

Examples

```
x <- Variable(2)
prob <- Problem(Minimize(sum_entries(x)), list(x >= 1))
objective(prob)
```

one_minus_pos *The difference 1 - x with domain (0, 1)*

Description

Log-log concave atom for DGP. Solve with `psolve(problem, gp = TRUE)`.

Usage

```
one_minus_pos(x)
```

Arguments

x	An Expression (elementwise in (0, 1))

Value

A OneMInusPos atom

Examples

```
x <- Variable(pos = TRUE)
prob <- Problem(Maximize(one_minus_pos(x)), list(x >= 0.1, x <= 0.5))
psolve(prob, gp = TRUE)
```

Or *Logical OR*

Description

Returns 1 if and only if at least one argument equals 1, and 0 otherwise. For two operands, can also be written with the | operator: x | y.

Usage

```
Or(..., id = NULL)
```

Arguments

...	Two or more boolean Variables or logic expressions.
id	Optional integer ID (internal use).

Value

An Or expression.

See Also

Not(), And(), Xor(), implies(), iff()

Examples

```
x <- Variable(boolean = TRUE)
y <- Variable(boolean = TRUE)
z <- Variable(boolean = TRUE)
either <- x | y          # operator syntax
either <- Or(x, y)       # functional syntax
any3 <- Or(x, y, z)      # n-ary
```

p_norm — *General p-norm of an expression*

Description

General p-norm of an expression

Usage

```
p_norm(
  x,
  p = 2,
  axis = NULL,
  keepdims = FALSE,
  max_denom = 1024L,
  approx = TRUE
)
```

Arguments

x	An Expression
p	Numeric exponent (default 2)
axis	NULL (all), 1 (row-wise), or 2 (column-wise)
keepdims	Logical
max_denom	Integer max denominator for rational approx
approx	If TRUE (default), use SOC approximation. If FALSE, use exact power cone.

Value

A Pnorm, PnormApprox, Norm1, or NormInf atom

param_dict — *Get all Parameters of a Problem as a Named List*

Description

Mirrors CVXPY's `Problem.param_dict` property (`cvxpy/problems/problem.py:257-262`): returns a named list keyed by each parameter's name, where the value is the `Parameter` object itself.

Usage

```
param_dict(x, ...)
```

Arguments

x	A Problem object.
...	Not used.

Value

Named list of Parameter objects, keyed by name.

Parameter *Create a Parameter*

Description

Constructs a parameter whose numeric value can be changed without re-canonicalizing the problem. Parameters are treated as constants for DCP purposes but their value can be updated between solves.

Usage

```
Parameter(
  shape = c(1L, 1L),
  name = NULL,
  value = NULL,
  id = NULL,
  latex_name = NULL,
  ...
)
```

Arguments

shape	Integer vector of length 1 or 2 giving the parameter dimensions. A scalar n is interpreted as c(n, 1). Defaults to c(1, 1) (scalar).
name	Optional character string name. If NULL, an automatic name "param<id>" is generated.
value	Optional initial numeric value.
id	Optional integer ID.
latex_name	Optional character string giving a custom LaTeX name for use in visualizations. For example, "\\gamma". If NULL (default), visualizations auto-generate a LaTeX name.
...	Additional attributes: nonneg, nonpos, etc.

Value

A Parameter object (inherits from Leaf and Expression).

Examples

```
p <- Parameter()
value(p) <- 5
p_vec <- Parameter(3, nonneg = TRUE)
gamma <- Parameter(1, name = "gamma", latex_name = "\\gamma")
```

---

parameters *Get the Parameters in an Expression*

Description

Get the Parameters in an Expression

Usage

```
parameters(x, ...)
```

Arguments

x	An expression or problem object.
...	Not used.

Value

List of Parameter objects.

---

partial_optimize *Partial optimization transform*

Description

Builds an Expression representing the optimal value of prob as a function of the variables you choose NOT to optimize over. Useful for two-stage / hierarchical optimization, custom atom definitions, and embedding sub-problems inside larger problems.

Usage

```
partial_optimize(
  prob,
  opt_vars = NULL,
  dont_opt_vars = NULL,
  solver = NULL,
  ...
)
```

Arguments

prob	A Problem to partially optimize.
opt_vars	Optional list of Variables to optimize over.
dont_opt_vars	Optional list of Variables to keep as free arguments of the resulting expression.
solver	Optional solver name (passed to psolve() when the PartialProblem is evaluated via value() or grad()).
...	Additional named arguments forwarded to psolve() when value() / grad() are called.

Details

Exactly one of opt_vars or dont_opt_vars may be NULL; the missing list is taken to be the complement (relative to the full list of variables in prob). If both are supplied, they must together cover every variable in prob.

The returned PartialProblem is an Expression with scalar shape: it is convex when prob is DCP with a Minimize objective and concave when DCP with Maximize. Embed it like any other expression in a larger Problem; the larger problem's canonicalizer will pull the inner objective and constraints into the outer cone form so a single solve handles both layers.

Value

A PartialProblem expression.

See Also

Problem, psolve()

Examples

```
x <- Variable(3)
t <- Variable(3)
abs_x <- partial_optimize(
  Problem(Minimize(sum_entries(t)), list(-t <= x, x <= t)),
  opt_vars = list(t)
)
## abs_x is now an expression of x alone, equivalent to sum(abs(x)).
```

---

partial_trace *Partial trace of a tensor product expression*

Description

Assumes expr is a 2D square matrix representing a Kronecker product of length(dims) subsystems. Returns the partial trace over the subsystem at index axis (1-indexed).

Usage

```
partial_trace(expr, dims, axis = 1L)
```

Arguments

expr	An Expression (2D square matrix)
dims	Integer vector of subsystem dimensions
axis	Integer (1-indexed) subsystem to trace out

Value

An Expression representing the partial trace

---

partial_transpose *Partial transpose of a tensor product expression*

Description

Assumes `expr` is a 2D square matrix representing a Kronecker product of `length(dims)` subsystems. Returns the partial transpose with the transpose applied to the subsystem at index `axis` (1-indexed).

Usage

```
partial_transpose(expr, dims, axis = 1L)
```

Arguments

expr	An Expression (2D square matrix)
dims	Integer vector of subsystem dimensions
axis	Integer (1-indexed) subsystem to transpose

Value

An Expression representing the partial transpose

perspective *Perspective Transform*

Description

Creates the perspective transform of a scalar convex or concave expression. Given a scalar expression `f(x)` and a nonneg variable `s`, the perspective is `s * f(x/s)`.

Usage

```
perspective(f, s, f_recession = NULL)
```

Arguments

`f`	A scalar convex or concave Expression.
`s`	A nonneg Variable (scalar).
`f_recession`	Optional recession function for handling `s = 0`.

Value

A `Perspective` expression.

pf_eigenvalue *Perron-Frobenius eigenvalue of a positive matrix*

Description

Log-log convex atom for DGP. Solve with `psolve(problem, gp = TRUE)`.

Usage

```
pf_eigenvalue(X)
```

Arguments

`X`	An Expression (positive square matrix)

Value

A PfEigenvalue atom (scalar)

Examples

```
X <- Variable(c(2, 2), pos = TRUE)
prob <- Problem(Minimize(pf_eigenvalue(X)),
                list(X[1,1] >= 0.1, X[2,2] >= 0.1))
psolve(prob, gp = TRUE)
```

pos *Positive part: max(x, 0)*

Description

Positive part: max(x, 0)

Usage

```
pos(x)
```

Arguments

x	An Expression

Value

A Maximum atom

PowCone3D *Create a 3D Power Cone Constraint*

Description

Constrains (x, y, z) to lie in the 3D power cone:

$$x^{\alpha} \cdot y^{1-\alpha} \geq |z|, \quad x \geq 0,\ y \geq 0$$

Usage

```
PowCone3D(x_expr, y_expr, z_expr, alpha, constr_id = NULL)
```

Arguments

x_expr	A CVXR expression.
y_expr	A CVXR expression.
z_expr	A CVXR expression.
alpha	A CVXR expression or numeric value in $(0, 1)$.
constr_id	Optional integer constraint ID.

Value

A PowCone3D constraint object.

See Also

PowConeND

PowConeND *Create an N-Dimensional Power Cone Constraint*

Description

Constrains (W, z) to lie in the N-dimensional power cone:

$$\prod W_i^{\alpha_i} \geq |z|, \quad W \geq 0$$

where $\alpha_i > 0$ and $\sum \alpha_i = 1$.

Usage

```
PowConeND(W, z, alpha, axis = 2L, constr_id = NULL)
```

Arguments

`W`	A CVXR expression (vector or matrix).
`z`	A CVXR expression (scalar or vector).
`alpha`	A CVXR expression with positive entries summing to 1 along the specified axis.
`axis`	Integer, 2 (default, column-wise) or 1 (row-wise).
`constr_id`	Optional integer constraint ID.

Value

A `PowConeND` constraint object.

Known limitations

The R **clarabel** solver does not currently support the PowConeND cone specification. Problems involving PowConeND (e.g., exact geometric mean with more than 2 arguments) should use SCS or MOSEK as the solver, or use approximation-based atoms (e.g., `geo_mean(x, approx = TRUE)`).

See Also

`PowCone3D`

Problem *Create an Optimization Problem*

Description

Constructs a convex optimization problem from an objective and a list of constraints. Use psolve to solve the problem.

Usage

```
Problem(objective, constraints = list())
```

Arguments

objective	A Minimize or Maximize object.
constraints	A list of Constraint objects (e.g., created by ==, <=, >= operators on expressions). Defaults to an empty list (unconstrained).

Value

A Problem object.

Known limitations

- Problems must contain at least one Variable. Zero-variable problems (e.g., minimizing a constant) will cause an internal error in the reduction pipeline.

Examples

```
x <- Variable(2)
prob <- Problem(Minimize(sum_entries(x)), list(x >= 1))
```

problem_data *Get Problem Data for a Solver*

Description

Returns the problem data that would be passed to a specific solver, along with the reduction chain and inverse data for solution retrieval.

Usage

```
problem_data(
  x,
  solver = NULL,
  gp = FALSE,
  enforce_dpp = FALSE,
  ignore_dpp = FALSE,
  ...
)
```

Arguments

x	A Problem object.
solver	Character string naming solver, or NULL for automatic selection.
gp	Logical; if TRUE, parse the problem as a geometric program.
enforce_dpp	Logical; if TRUE, raise an error when a parametrized problem is not DPP instead of compiling it as non-DPP.
ignore_dpp	Logical; if TRUE, treat a DPP problem as non-DPP (skip the DPP fast path).
...	Additional solver options.

Value

A list with components data, chain, and inverse_data.

---

problem_solution *Get the Raw Solution Object (deprecated)*

Description

[Deprecated]

Usage

```
problem_solution(x)
```

Arguments

x	A Problem object.

Details

Use solution instead.

Value

A Solution object, or NULL if the problem has not been solved.

See Also

solution

`problem_status` *Get the Solution Status of a Problem (deprecated)*

Description

[Deprecated]

Usage

```
problem_status(x)
```

Arguments

x	A Problem object.

Details

Use status instead.

Value

Character string, or NULL if the problem has not been solved.

See Also

status

`problem_unpack_results` *Unpack Solver Results into a Problem*

Description

Inverts the reduction chain and unpacks the raw solver solution into the original problem's variables and constraints. This is step 3 of the decomposed solve pipeline:

1. problem_data() – compile the problem
2. solve_via_data(chain, data) – call the solver
3. problem_unpack_results() – invert and unpack

Usage

```
problem_unpack_results(problem, solution, chain, inverse_data)
```

Arguments

problem	A Problem object.
solution	The raw solver result from solve_via_data().
chain	The SolvingChain from problem_data().
inverse_data	The inverse data list from problem_data().

Details

After calling this function, variable values are available via value() and constraint duals via dual_value().

Value

The problem object (invisibly), with solution unpacked.

See Also

problem_data, solve_via_data

---

prod_entries *Product of entries along an axis*

Description

Used in DGP (geometric programming) context. Solve with psolve(problem, gp = TRUE).

Usage

```
prod_entries(x, axis = NULL, keepdims = FALSE)
```

Arguments

x	An Expression
axis	NULL (all), 1 (row-wise), or 2 (column-wise)
keepdims	Whether to keep reduced dimensions

Value

A Prod atom

Examples

```
x <- Variable(3, pos = TRUE)
prob <- Problem(Minimize(prod_entries(x)), list(x >= 2))
psolve(prob, gp = TRUE)
```

PSD — *Create a Positive Semidefinite Constraint*

Description

Constrains a square matrix expression to be positive semidefinite (PSD): $X \succeq 0$. The expression must be square.

Usage

```
PSD(expr, constr_id = NULL)
```

Arguments

`expr`	A CVXR expression representing a square matrix.
`constr_id`	Optional integer constraint ID.

Value

A `PSD` constraint object.

psolve — *Solve a Convex Optimization Problem*

Description

Solves the problem and returns the optimal objective value. After solving, variable values can be retrieved with `value`, constraint dual values with `dual_value`, and solver information with `solver_stats`.

Usage

```
psolve(
  problem,
  solver = NULL,
  gp = FALSE,
  qcp = FALSE,
  verbose = FALSE,
  warm_start = FALSE,
  requires_grad = FALSE,
  nlp = FALSE,
  enforce_dpp = FALSE,
  ignore_dpp = FALSE,
  solver_path = NULL,
  ...
)
```

Arguments

problem	A Problem object.
solver	Character string naming the solver to use (e.g., "CLARABEL", "SCS", "OSQP", "HIGHS"), or NULL for automatic selection.
gp	Logical; if TRUE, solve as a geometric program (DGP).
qcp	Logical; if TRUE, solve as a quasiconvex program (DQCP) via bisection. Only needed for non-DCP DQCP problems.
verbose	Logical; if TRUE, print solver output.
warm_start	Logical; if TRUE, use the current variable values as a warm-start point for the solver.
requires_grad	Logical; if TRUE, route the solve through the DIFFCP wrapper so backward() / derivative() can recover gradients.
nlp	Logical; if TRUE, solve the problem as a disciplined nonlinear program (DNLP) using the NLP reduction chain and an NLP solver (e.g. "UNO"). The problem must satisfy is_dnlp().
enforce_dpp	Logical; if TRUE, raise an error when a parametrized problem is not DPP instead of compiling it as non-DPP.
ignore_dpp	Logical; if TRUE, treat a DPP problem as non-DPP (skip the DPP fast path).
solver_path	Optional fallback chain. A character vector of solver names or a list whose entries are either character names or length-2 list(name, opts) pairs. Each solver is tried in sequence; the first that succeeds returns its result. If every solver fails, a SolverError-classed condition is raised with the per-solver error messages. Cannot be combined with solver. Mirrors CVXPY's solver_path argument.
...	Solver options passed to solver_opts(). Includes chain-construction options (use_quad_obj), standard tolerances (feastol, reltol, abstol, num_iter), and solver-specific parameters (e.g., eps_abs, scip_params). See solver_opts for details. For DQCP problems (qcp = TRUE), additional arguments include low, high, eps, max_iters, and max_iters_interval_search.

Value

The optimal objective value (numeric scalar), or Inf / -Inf for infeasible / unbounded problems.

See Also

Problem, status, solver_stats, solver_default_param

Examples

```
x <- Variable()
prob <- Problem(Minimize(x), list(x >= 5))
result <- psolve(prob, solver = "CLARABEL")
```

ptp *Peak-to-peak (range): max(x) - min(x)*

Description

Computes the range of values along an axis: `max(x) - min(x)`. The result is always nonnegative.

Usage

```
ptp(x, axis = NULL, keepdims = FALSE)
```

Arguments

`x`	An Expression or numeric value.
`axis`	NULL (all), 0 (columns), or 1 (rows).
`keepdims`	Logical; keep reduced dimension?

Value

An Expression representing max(x) - min(x).

quad_form *Quadratic form x^T P x*

Description

When `x` is constant, returns `t(Conj(x)) %*% P %*% x` (affine in `P`). When `P` is constant, returns a `QuadForm` atom (quadratic in `x`). At least one of `x` or `P` must be constant.

Usage

```
quad_form(x, P, assume_PSD = FALSE)
```

Arguments

`x`	An Expression (vector)
`P`	An Expression (square matrix, symmetric/Hermitian)
`assume_PSD`	If TRUE, assume P is PSD without checking (only when P is constant).

Value

A QuadForm atom or an affine Expression

quad_over_lin *Sum of squares divided by a scalar*

Description

Sum of squares divided by a scalar

Usage

```
quad_over_lin(x, y, axis = NULL, keepdims = FALSE)
```

Arguments

x	An Expression
y	An Expression (scalar, positive)
axis	NULL (all), 1 (row-wise), or 2 (column-wise)
keepdims	Logical: keep reduced dimensions?

Value

A QuadOverLin atom

reduction-chain-rule *Reduction chain-rule hooks (dict-in / dict-out)*

Description

Walk the solving chain during `Problem$backward()` / `Problem$derivative()`. Each hook takes and returns a named list keyed by `as.character(leaf@id)` whose values are shaped arrays (the leaf's `dim`). The base `Reduction` methods are identity pass-throughs.

Usage

```
var_backward(x, del_vars)

var_forward(x, dvars)

param_backward(x, dparams)

param_forward(x, param_deltas)
```

Arguments

x	A Reduction.
del_vars	Named list var-id -> gradient array (outer representation).
dvars	Named list var-id -> delta array (inner representation).
dparams	Named list param-id -> gradient array (inner representation).
param_deltas	Named list param-id -> delta array (outer representation).

Value

For var_backward: the same map in the inner (reduced) representation.

For var_forward: the same map in the outer (original) representation.

For param_backward: the same map in the outer (original) representation.

For param_forward: the same map in the inner (transformed) representation.

---

reduction-id-map *Reduction leaf-id maps*

Description

Reduction leaf-id maps

Usage

```
var_id_map(x)

param_id_map(x)
```

Arguments

x	A Reduction.

Value

A named list orig-id -> character vector of reduced-id(s); empty by default (the reduction replaces no leaves).

rel_entr *Relative Entropy: x*log(x/y)*

Description

Relative Entropy: x*log(x/y)

Usage

```
rel_entr(x, y)
```

Arguments

x	An Expression
y	An Expression

Value

A RelEntr atom

reshape_expr *Reshape an expression to a new shape*

Description

Reshape an expression to a new shape

Usage

```
reshape_expr(x, dim, order = "F")
```

Arguments

x	An Expression or numeric value.
dim	Integer vector of length 2: the target shape c(nrow, ncol). A single integer is treated as c(dim, 1). Use -1 to infer a dimension.
order	Character: "F" (column-major, default) or "C" (row-major).

Value

A Reshape expression.

residual *Get the Residual of a Constraint*

Description

Returns the residual of a constraint, measuring how much the constraint is violated or satisfied.

Usage

```
residual(x, ...)
```

Arguments

x	A constraint object.
...	Not used.

Value

Numeric array, or NULL if expression has no value.

resolvent *Resolvent inverse(sI - X)*

Description

Equivalent to (1/s) * eye_minus_inv(X / s).

Usage

```
resolvent(X, s)
```

Arguments

X	An Expression (positive square matrix)
s	A positive scalar

Value

An expression for the resolvent

sample_bounds *Sampling Bounds for NLP Random Restarts*

Description

Get or set a Variable's sample_bounds – a (low, high) region used to draw random initial points in best_of NLP solves (psolve(prob, nlp = TRUE, best_of = n)). When set, it overrides the variable's value during random initialization; when NULL (the default) finite variable bounds are used instead. Supply a pair c(low, high) (scalars broadcast to the variable shape) or a list(low, high) of per-entry vectors; set NULL to clear.

Usage

```
sample_bounds(x, ...)

sample_bounds(x) <- value
```

Arguments

x	A Variable.
...	Not used.
value	A (low, high) pair, or NULL to clear.

Value

sample_bounds(x) returns the stored list(low, high) or NULL; the setter returns the modified variable.

scalar_product *Scalar product (alias for vdot)*

Description

Scalar product (alias for vdot)

Usage

```
scalar_product(x, y)
```

Arguments

x	An Expression, numeric value, or nested list thereof. The conjugate-linear argument.
y	An Expression, numeric value, or nested list thereof. The linear argument.

Value

A scalar Expression representing `sum(Conj(x) * y)`.

See Also

`vdot()`

`scalarize` *Scalarize multiple objectives into a single objective*

Description

Transforms for combining several `Minimize`/`Maximize` objectives into one objective for multi-objective optimization. Mirrors CVXPY's `cvxpy.transforms.scalarize` submodule; access members with `$`:

Usage

```
scalarize
```

Format

A named list of four functions.

Details

- `scalarize$weighted_sum(objectives, weights)` – weighted sum of objectives.
- `scalarize$targets_and_priorities(objectives, priorities, targets, limits = NULL, off_target = 1e-5)` – penalize each objective within a `[target, limit]` range; a negative priority flips the objective sense.
- `scalarize$max(objectives, weights)` – minimize the largest weighted objective term.
- `scalarize$log_sum_exp(objectives, weights, gamma = 1.0)` – smooth maximum; `gamma -> 0` approaches `weighted_sum`, `gamma -> Inf` approaches `max`.

Value

`scalarize` itself is a named list of the four functions above. Each function returns a single `Objective` combining the inputs: `max` and `log_sum_exp` always return a `Minimize`; `weighted_sum` returns a `Minimize` when all objectives are `Minimize` and a `Maximize` when all are `Maximize` (mixing the two is an error); `targets_and_priorities` returns a `Minimize` or `Maximize` according to the objective senses (after negative priorities flip them) and the curvature of the combined expression.

Examples

```
x <- Variable()
objs <- list(Minimize(square(x)), Minimize(square(x - 1)))
obj <- scalarize$weighted_sum(objs, c(1, 1))
psolve(Problem(obj))
```

---

scalene *Scalene penalty: alpha * pos(x) + beta * neg(x)*

Description

Scalene penalty: alpha * pos(x) + beta * neg(x)

Usage

```
scalene(x, alpha, beta)
```

Arguments

x	An Expression
alpha	Coefficient for the positive part
beta	Coefficient for the negative part

Value

An Expression representing the scalene penalty

---

set_label *Attach a label to an expression*

Description

CVXPY-parity setter that returns its first argument so calls can be chained (e.g. `sum_squares(x) |> set_label("cost")`). See `format_labeled()` for the pretty-printer that consumes labels.

Usage

```
set_label(x, value)
```

Arguments

x	An Expression object.
value	A label (character; coerced via `as.character`). Pass `NULL` to clear an existing label.

Value

x with the label updated.

See Also

label(), format_labeled()

---

sigma_max *Maximum singular value*

Description

Maximum singular value

Usage

```
sigma_max(A)
```

Arguments

A	A matrix expression

Value

An expression representing the maximum singular value of A

---

size *Get Expression Size*

Description

Returns the total number of elements in the expression.

Usage

```
size(x)
```

Arguments

x	A CVXR expression.

Value

An integer (product of shape dimensions).

See Also

is_scalar(), is_vector(), is_matrix()

size_metrics *Get Size Metrics for a Problem*

Description

Mirrors CVXPY's `Problem.size_metrics` property (`cvxpy/problems/problem.py:515-521`, class at lines 1592-1654): returns a `SizeMetrics` object summarizing the problem's scale.

Usage

```
size_metrics(x, ...)
```

Arguments

`x`	A `Problem` object.
`...`	Not used.

Value

A `SizeMetrics` object with six numeric fields.

SizeMetrics *Problem Size Metrics*

Description

Reports scalar-counts and data-dimension metrics for a `Problem`. Constructed by `size_metrics`; end users normally call `size_metrics(prob)` rather than this constructor directly.

Usage

```
SizeMetrics(
  num_scalar_variables = 0L,
  num_scalar_data = 0L,
  num_scalar_eq_constr = 0L,
  num_scalar_leq_constr = 0L,
  max_data_dimension = 0L,
  max_big_small_squared = 0
)
```

Arguments

`num_scalar_variables`
: Total scalar entries across all variables in the problem.

`num_scalar_data`
: Total scalar entries across all constants and parameters.

`num_scalar_eq_constr`
: Total scalar entries in equality (`Equality`, `Zero`) constraints.

`num_scalar_leq_constr`
: Total scalar entries in inequality (`Inequality`, `NonNeg`, `NonPos`) constraints.

`max_data_dimension`
: Largest single dimension across any data block (constant or parameter).

`max_big_small_squared`
: Maximum of `big * small^2` over all data blocks, where `big`/`small` are the larger/ smaller dimension of the block.

Value

A `SizeMetrics` object.

`SOC` *Create a Second-Order Cone Constraint*

Description

Constrains $\|X_i\|_2 \leq t_i$ for each column or row i, where `t` is a vector and `X` is a matrix.

Usage

```
SOC(t, X, axis = 2L, constr_id = NULL)
```

Arguments

`t`	A CVXR expression (scalar or vector) representing the upper bound.
`X`	A CVXR expression (vector or matrix) whose columns/rows are bounded.
`axis`	Integer, 2 (default, column-wise) or 1 (row-wise). Determines whether columns (2) or rows (1) of `X` define the individual cones.
`constr_id`	Optional integer constraint ID.

Value

An `SOC` constraint object.

solution *Get the Raw Solution Object*

Description

Returns the raw `Solution` object from the most recent solve, containing primal and dual variable values, status, and solver attributes.

Usage

```
solution(x)
```

Arguments

x	A `Problem` object.

Value

A `Solution` object, or `NULL` if the problem has not been solved.

solve_via_data *Solve via Raw Data*

Description

Calls the solver on pre-compiled problem data (step 2 of the decomposed solve pipeline). Dispatches on `x`: when `x` is a `SolvingChain`, delegates to the terminal solver with proper cache management.

Usage

```
solve_via_data(
  x,
  data,
  warm_start = FALSE,
  verbose = FALSE,
  solver_opts = list(),
  ...
)
```

Arguments

x	A SolvingChain (preferred) or Solver object.
data	Named list of solver data from problem_data().
warm_start	Logical; use warm-start if supported.
verbose	Logical; print solver output.
solver_opts	Named list of solver-specific options.
...	Additional arguments forwarded to the method (e.g. problem for the SolvingChain method, solver_cache for the Solver method).

Value

Solver-specific result (a named list).

See Also

problem_data, problem_unpack_results

---

solver-constants *Solver Name Constants*

Description

Character string constants identifying the available solvers.

Usage

SCS_SOLVER

OSQP_SOLVER

CLARABEL_SOLVER

DIFFCP_SOLVER

HIGHS_SOLVER

MOSEK_SOLVER

GUROBI_SOLVER

GLPK_SOLVER

GLPK_MI_SOLVER

ECOS_SOLVER

`ECOS_BB_SOLVER`

`CPLEX_SOLVER`

`CVXOPT_SOLVER`

`PIQP_SOLVER`

`SCIP_SOLVER`

`XPRESS_SOLVER`

`IPOPT_SOLVER`

`KNITRO_SOLVER`

`UNO_SOLVER`

`COPT_SOLVER`

Value

A character string.

`solver_default_param` *Standard Solver Parameter Mappings*

Description

Returns a named list mapping standard CVXR parameter names (`reltol`, `abstol`, `feastol`, `num_iter`) to solver-specific parameter names. Used internally by `psolve` to translate standard parameters into solver-native names. A standard parameter with no entry for the chosen solver does not reach that solver; `psolve` warns when that happens.

Usage

```
solver_default_param()
```

Value

A named list keyed by solver name (e.g. `"CLARABEL"`, `"OSQP"`). Each element is a list of standard parameter mappings, where each mapping has `name` (the solver-native parameter name the standard name translates to) and `value` (the solver's own default for that parameter, recorded for reference when the mapping was written; CVXR does not apply these values – when you do not set a parameter, the solver uses its own current default).

See Also

psolve

---

solver_opts *Create Solver Options*

Description

Constructs a structured list of solver options for use with psolve and problem_data. Known parameters are sorted into named slots; solver-specific parameters are collected in $solver_specific.

Usage

```
solver_opts(
  use_quad_obj = TRUE,
  feastol = NULL,
  reltol = NULL,
  abstol = NULL,
  num_iter = NULL,
  ...
)
```

Arguments

use_quad_obj	Logical. If TRUE (default), quadratic objectives use the QP matrix path. If FALSE, forces conic decomposition via quad_form_canon.
feastol	Feasibility tolerance (solver-agnostic). Translated to solver-native name by internal mapping. NULL uses solver default.
reltol	Relative tolerance. NULL uses solver default.
abstol	Absolute tolerance. NULL uses solver default.
num_iter	Maximum iterations. NULL uses solver default.
...	Solver-specific parameters passed directly to the solver (e.g., eps_abs, scip_params, mosek_params).

Value

A named list with class "solver_opts".

Examples

```
solver_opts(feastol = 1e-6)
solver_opts(use_quad_obj = FALSE, eps_abs = 1e-7)
solver_opts(scip_params = list("limits/time" = 10))
```

solver_stats *Get Solver Statistics*

Description

Returns solver statistics from the most recent solve, including solve time, setup time, and iteration count.

Usage

```
solver_stats(x)
```

Arguments

x	A Problem object.

Value

A SolverStats object, or NULL if the problem has not been solved.

square *Square of an expression: x^2*

Description

Square of an expression: x^2

Usage

```
square(x)
```

Arguments

x	An Expression

Value

A PowerApprox atom with p=2, the same atom power(x, 2) builds

status — *Get the Solution Status of a Problem*

Description

Returns the status string from the most recent solve, such as `"optimal"`, `"infeasible"`, or `"unbounded"`.

Usage

```
status(x)
```

Arguments

x	A `Problem` object.

Value

Character string, or `NULL` if the problem has not been solved.

See Also

`OPTIMAL`, `INFEASIBLE`, `UNBOUNDED`

status-constants — *Solution Status Constants*

Description

Character string constants representing the possible solution statuses returned by `problem_status`.

Usage

```
OPTIMAL

INFEASIBLE

UNBOUNDED

SOLVER_ERROR

OPTIMAL_INACCURATE

INFEASIBLE_INACCURATE

UNBOUNDED_INACCURATE
```

USER_LIMIT

INFEASIBLE_OR_UNBOUNDED

Value

A character string.

See Also

status, psolve

---

sum_entries *Sum the entries of an expression*

Description

Sum the entries of an expression

Usage

```
sum_entries(x, axis = NULL, keepdims = FALSE)
```

Arguments

x	An Expression or numeric value.
axis	NULL (sum all), 1 (row-wise, like apply(X,1,sum)), or 2 (column-wise, like apply(X,2,sum)).
keepdims	Logical: if TRUE, keep the reduced dimension as size 1.

Value

A SumEntries expression.

sum_largest *Sum of k largest entries*

Description

Sum of k largest entries

Usage

```
sum_largest(x, k, axis = NULL, keepdims = FALSE)
```

Arguments

x	An Expression
k	Number of largest entries to sum
axis	NULL (all entries), 1 (row-wise), or 2 (column-wise)
keepdims	Logical; keep the reduced dimension as size 1

Value

A SumLargest atom

sum_smallest *Sum of k smallest entries*

Description

Sum of k smallest entries

Usage

```
sum_smallest(x, k, axis = NULL, keepdims = FALSE)
```

Arguments

x	An Expression
k	Number of smallest entries to sum
axis	NULL (all entries), 1 (row-wise), or 2 (column-wise)
keepdims	Logical; keep the reduced dimension as size 1

Value

An Expression equal to -SumLargest(-x, k)

sum_squares *Sum of squares (= quad_over_lin(x, 1))*

Description

Sum of squares (= quad_over_lin(x, 1))

Usage

```
sum_squares(x, axis = NULL, keepdims = FALSE)
```

Arguments

x	An Expression
axis	NULL (all), 1 (row-wise), or 2 (column-wise)
keepdims	Logical: keep reduced dimensions?

Value

A QuadOverLin atom

to_latex *Convert CVXR Object to LaTeX*

Description

Renders a CVXR `Problem`, `Expression`, or `Constraint` as a LaTeX string. Problem-level output uses the `optidef` package (`mini*`/`maxi*` environments) and atom macros from `dcp.sty` (shipped as `system.file("sty", "dcp.sty", package = "CVXR")`).

Usage

```
to_latex(x, ...)
```

Arguments

x	A `Problem`, `Expression`, `Constraint`, or `Objective`.
...	Reserved for future options.

Value

A character string containing LaTeX code.

Examples

```
x <- Variable(3, name = "x")
cat(to_latex(p_norm(x, 2)))
# \cvxnorm{x}_2
```

---

total_variation *Total variation of a vector or matrix*

Description

Computes total variation using L1 norm of discrete gradients for vectors and L2 norm of discrete gradients for matrices.

Usage

```
total_variation(value, ...)
```

Arguments

value	An Expression or numeric constant (vector or matrix)
...	Additional matrix expressions extending the third dimension

Value

An Expression representing the total variation

---

tr_inv *Trace of matrix inverse*

Description

Computes $\mathrm{tr}(X^{-1})$ for PSD matrix X.

Usage

```
tr_inv(X)
```

Arguments

X	A square PSD matrix expression

Value

An expression representing $\mathrm{tr}(X^{-1})$

tv *Total variation (deprecated alias)*

Description

[Deprecated]

Usage

```
tv(...)
```

Arguments

`...`	Arguments passed to `total_variation()`.

Details

Use `total_variation()` instead.

Value

The expression returned by `total_variation()`: the total variation of the argument(s). A deprecation warning is issued once per session.

See Also

`total_variation()`

unpack_results *Unpack Results (backward-compatible alias)*

Description

[Deprecated]

Usage

```
unpack_results(problem, solution, chain, inverse_data)
```

Arguments

`problem`	A `Problem` object.
`solution`	The raw solver result from `solve_via_data()`.
`chain`	The `SolvingChain` from `problem_data()`.
`inverse_data`	The inverse data list from `problem_data()`.

Details

Use problem_unpack_results() instead. This alias exists for backward compatibility with older CVXR examples.

Value

The problem object (invisibly), with solution unpacked.

See Also

problem_unpack_results()

---

upper_tri *Extract strict upper triangle of a square matrix*

Description

Extract strict upper triangle of a square matrix

Usage

```
upper_tri(x)
```

Arguments

x	An Expression (square matrix)

Value

An UpperTri atom (column vector)

---

value *Get the Numeric Value of an Expression*

Description

Returns the numeric value of a CVXR expression, variable, or constant. For variables, the value is set after solving a problem.

Usage

```
value(x, ...)
```

Arguments

x	An expression object.
...	Not used.

Value

A numeric matrix, or NULL if no value has been set.

---

value<- *Set the Value of a Leaf Expression*

Description

Assigns a numeric value to a Variable or Parameter.

Usage

```
value(x) <- value
```

Arguments

x	A leaf expression object.
value	The value to assign.

Value

The modified object (invisibly).

---

var_dict *Get all Variables of a Problem as a Named List*

Description

Mirrors CVXPY's Problem.var_dict property (cvxpy/problems/problem.py:264-269): returns a named list keyed by each variable's name, where the value is the Variable object itself.

Usage

```
var_dict(x, ...)
```

Arguments

x	A Problem object.
...	Not used.

Value

Named list of Variable objects, keyed by name.

`Variable` *Create an Optimization Variable*

Description

Constructs a variable to be used in a CVXR optimization problem. Variables are decision variables that the solver optimizes over.

Usage

```
Variable(
  shape = c(1L, 1L),
  name = NULL,
  value = NULL,
  var_id = NULL,
  latex_name = NULL,
  ...
)
```

Arguments

`shape`	Integer vector of length 1 or 2 giving the variable dimensions. A scalar `n` is interpreted as `c(n, 1)`. Defaults to `c(1, 1)` (scalar).
`name`	Optional character string name for the variable. If `NULL`, an automatic name `"var<id>"` is generated.
`value`	Optional numeric initial value (scalar, vector, or matrix matching `shape`). Validated and projected onto the attribute domain via the same path as `value(var) <- val`.
`var_id`	Optional integer ID. If `NULL`, a unique ID is generated.
`latex_name`	Optional character string giving a custom LaTeX name for use in visualizations. For example, `"\\mathbf{x}"`. If `NULL` (default), visualizations auto-generate a LaTeX name.
`...`	Additional attributes: `nonneg`, `nonpos`, `PSD`, `NSD`, `symmetric`, `boolean`, `integer`, etc.

Value

A `Variable` object (inherits from `Leaf` and `Expression`).

Examples

```
x <- Variable(3)        # 3x1 column vector
X <- Variable(c(2, 3))  # 2x3 matrix
y <- Variable(2, nonneg = TRUE)  # non-negative variable
z <- Variable(3, name = "z", latex_name = "\\mathbf{z}")  # custom LaTeX
```

variables *Get the Variables in an Expression*

Description

Get the Variables in an Expression

Usage

```
variables(x, ...)
```

Arguments

x	An expression or problem object.
...	Not used.

Value

List of `Variable` objects.

vdot *Vector dot product (inner product)*

Description

The standard inner product of `x` and `y`: both are flattened, multiplied elementwise, and summed. **Conjugate-linear in** `x` — `x` is conjugated before multiplying, matching `numpy.vdot()` and CVXPY's `vdot()`. For real arguments conjugation is the identity, so this is the ordinary dot product.

Either argument may also be a (possibly nested) list, which is flattened before the product; the two are flattened independently, so their nesting need not match.

Usage

```
vdot(x, y)
```

Arguments

x	An Expression, numeric value, or nested list thereof. The conjugate-linear argument.
y	An Expression, numeric value, or nested list thereof. The linear argument.

Value

A scalar Expression representing `sum(Conj(x) * y)`.

See Also

scalar_product(), cvxr_outer(), deep_flatten()

Examples

```
x <- Variable(3)
vdot(x, c(1, 2, 3))

a <- Variable(); b <- Variable()
vdot(list(a, b), c(1, 2))
```

---

vec *Vectorize an expression (column vector)*

Description

Reshapes an expression into a column vector of shape (n*m, 1).

Usage

```
vec(x)
```

Arguments

x	An Expression or numeric value.

Value

A Reshape atom (column vector).

---

vec_to_upper_tri *Reshape a vector into an upper triangular matrix*

Description

Inverts upper_tri. Takes a flat vector and returns an upper triangular matrix (row-major order, matching CVXPY convention).

Usage

```
vec_to_upper_tri(expr, strict = FALSE)
```

Arguments

expr	An Expression (vector).
strict	Logical. If TRUE, returns a strictly upper triangular matrix (diagonal is zero). If FALSE, includes the diagonal. Default is FALSE.

Value

An Expression representing the upper triangular matrix.

`violation` *Get the Violation of a Constraint*

Description

Returns the scalar violation (distance to feasibility) of a constraint.

Usage

```
violation(x, ...)
```

Arguments

`x`	A constraint object.
`...`	Not used.

Value

Numeric scalar.

`visualize` *Visualize the Canonicalization Pipeline of a CVXR Problem*

Description

Displays the Smith form decomposition of a convex optimization problem, showing each stage of the DCP canonicalization pipeline: expression tree, Smith form, relaxed Smith form, conic form, and (optionally) standard cone form and solver data.

Usage

```
visualize(
  problem,
  output = c("text", "json", "html", "latex", "tikz"),
  solver = NULL,
  digits = 4L,
  file = NULL,
  open = interactive(),
  doc_base = "https://cvxr.rbind.io/reference/"
)
```

Arguments

problem	A Problem object.
output	Character: output format. "text" Console display (default). "json" JSON data model (for interop with HTML/Python). "html" Interactive D3+KaTeX HTML (Phase 2). "latex" LaTeX align* environments (Phase 3). "tikz" TikZ forest tree diagrams (Phase 3).
solver	Solver specification for matrix stuffing stages (4-5). NULL (default) shows only Stages 0-3 with zero overhead. TRUE uses the default solver (same as psolve()). A character string (e.g., "Clarabel") uses that specific solver.
digits	Integer: significant digits for displaying scalar constants. Integer-valued constants (0, 1, -3) always display without decimals regardless of this setting. Defaults to 4.
file	Character: path for HTML output file. If NULL (default), a temporary file is used.
open	Logical: whether to open the HTML file in a browser. Defaults to TRUE in interactive sessions.
doc_base	Character: base URL for atom documentation links. Defaults to the CVXR pkgdown site.

Value

For "text": invisible model list. For "json": a JSON string (or list if jsonlite not available). For "html": the file path (invisibly). For other formats: the rendered output (Phase 2+).

Examples

```
x <- Variable(3, name = "x")
prob <- Problem(Minimize(p_norm(x, 2)), list(x >= 1))
visualize(prob)                          # Stages 0-3 only
visualize(prob, solver = TRUE)           # Stages 0-5, default solver
visualize(prob, solver = "CLARABEL")     # Stages 0-5, specific solver
## Not run:
visualize(prob, output = "html", solver = TRUE)

## End(Not run)
```

---

vstack *Vertical concatenation of expressions*

Description

Named operands are accepted, as in rbind(), and the names are dropped.

Usage

```
vstack(..., deparse.level = 1)
```

Arguments

`...`	Expressions (same number of columns)
`deparse.level`	Accepted for compatibility with `rbind()`; expressions have no dimnames, so it has no effect

Value

A VStack atom

---

xexp *x * exp(x) – elementwise*

Description

x * exp(x) – elementwise

Usage

```
xexp(x)
```

Arguments

`x`	An Expression

Value

An Xexp atom

Xor *Logical XOR*

Description

For two arguments: result is 1 iff exactly one is 1. For n arguments: result is 1 iff an odd number are 1 (parity).

Usage

```
Xor(..., id = NULL)
```

Arguments

`...`	Two or more boolean Variables or logic expressions.
`id`	Optional integer ID (internal use).

Details

Note: R's `^` operator is used for `power()`, so `Xor` is functional syntax only.

Value

A `Xor` expression.

See Also

`Not()`, `And()`, `Or()`, `implies()`, `iff()`

Examples

```
x <- Variable(boolean = TRUE)
y <- Variable(boolean = TRUE)
exclusive <- Xor(x, y)
```

`Zero` *Create a Zero Constraint*

Description

Constrains an expression to equal zero elementwise: $x = 0$.

Usage

```
Zero(expr, constr_id = NULL)
```

Arguments

`expr`	A CVXR expression.
`constr_id`	Optional integer constraint ID.

Value

A `Zero` constraint object.

See Also

`Equality`, `NonPos`, `NonNeg`

Index

∗ **data**
    cdiac, 12
    dspop, 29
    dssamp, 29
%<<%, 7
%>>%, 6

And, 8
And(), *41*, *42*, *82*, *84*, *129*
as.matrix(), *9*
as_cvxr_expr, 8
available_solvers, 9

backward, 10, *98*
backward(), *25*, *26*
bmat, 11

CallbackParam, 11
cdiac, 12
ceil_expr, 13, *34*, *66*
CLARABEL_SOLVER *(*solver-constants*)*, 111
condition_number, 14
Constant, *8*, *9*, 14, *15*
constants, 15
constraints, 16
constraints(), *83*
conv, 16, *17*
convolve, 17, *17*
COPT_SOLVER *(*solver-constants*)*, 111
CPLEX_SOLVER *(*solver-constants*)*, 111
cummax_expr, 17
cumsum_axis, 18, *72*
curvature, 18, *19*, *67*
curvature(), *19*
curvatures, 19
curvatures(), *19*, *67*
cvar, 20
CVXOPT_SOLVER *(*solver-constants*)*, 111
cvxr_diff, 20, *72*
cvxr_mean, 21, *72*
cvxr_norm, 21, *71–73*
cvxr_outer, 22, *72*, *73*
cvxr_outer(), *125*
cvxr_power, 22
cvxr_std, 23, *72*, *73*
cvxr_var, 24, *72*, *73*

deep_flatten, 24
deep_flatten(), *125*
delta, 25
delta(), *26*
delta<- *(*delta*)*, 25
derivative, 25, *98*
derivative(), *11*
diag *(*math_atoms*)*, 70
DiagMat, 26, *27*, *70*, *73*
DiagVec, *26*, 27, *70*, *73*
diff_pos, 27
DIFFCP_SOLVER *(*solver-constants*)*, 111
dist_ratio, 28
dotsort, 28
dspop, 29, *29*
dssamp, 29, *29*
dual_value, 30, *96*, *97*

ECOS_BB_SOLVER *(*solver-constants*)*, 111
ECOS_SOLVER *(*solver-constants*)*, 111
entr, 30
Equality, 31, *43*, *129*
exclude_solvers *(*available_solvers*)*, 9
exclude_solvers(), *9*, *10*
ExpCone, 31
expr_H, 32
expr_name(), *34*
expr_sign, 32
eye_minus_inv, 33

FiniteSet, 33
floor_expr, *14*, 34, *66*
format_labeled, 34

format_labeled(), *63*, *106*, *107*

gen_lambda_max, 35
geo_mean, 35
get_bounds, 36
get_problem_data, 37
GLPK_MI_SOLVER *(*solver-constants*)*, 111
GLPK_SOLVER *(*solver-constants*)*, 111
gmatmul, 38
grad(), *88*
gradient, 38
gradient(), *11*
gradient<- *(*gradient*)*, 38
GUROBI_SOLVER *(*solver-constants*)*, 111

harmonic_mean, 39
HIGHS_SOLVER *(*solver-constants*)*, 111
hstack, 39
huber, 40

id, 40
iff, 41
iff(), *8*, *42*, *82*, *84*, *129*
implies, 41
implies(), *8*, *41*, *82*, *84*, *129*
include_solvers *(*available_solvers*)*, 9
include_solvers(), *10*
indicator, 42
Inequality, *31*, 43, *79*, *80*
INFEASIBLE, *115*
INFEASIBLE *(*status-constants*)*, 115
INFEASIBLE_INACCURATE
        *(*status-constants*)*, 115
INFEASIBLE_OR_UNBOUNDED
        *(*status-constants*)*, 115
installed_solvers, 43
installed_solvers(), *10*
inv_pos, 44
inv_prod, 44
IPOPT_SOLVER *(*solver-constants*)*, 111
is_affine, 45
is_affine(), *19*
is_atom_smooth, 45
is_concave, 46
is_concave(), *19*
is_constant, 46
is_constant(), *19*
is_convex, 47
is_convex(), *19*
is_dcp, 47
is_dgp, 48
is_dgp(), *67*
is_diag, 48
is_dnlp, 49, *50*, *51*, *60*, *98*
is_dpp, 49
is_dqcp, 50
is_linearizable_concave, 50, *51*, *60*
is_linearizable_convex, *50*, 51, *60*
is_log_log_affine, 51
is_log_log_concave, 52
is_log_log_convex, 52
is_lp, 53
is_matrix, 53
is_matrix(), *59*, *61*, *107*
is_mixed_integer, 54
is_nonneg, 54
is_nonpos, 55
is_nsd, 55
is_psd, 56
is_pwl, 56
is_qp, 57
is_quadratic, 57
is_quasiconcave, 58
is_quasiconvex, 58
is_quasilinear, 59
is_scalar, 59
is_scalar(), *53*, *61*, *107*
is_smooth, *45*, 60
is_symmetric, 60
is_vector, 61
is_vector(), *53*, *59*, *107*
is_zero, 61

kl_div, 62
KNITRO_SOLVER *(*solver-constants*)*, 111
kron, 62

label, 63
label(), *35*, *107*
label<-, 63
lambda_max, 64
lambda_min, 64
lambda_sum_largest, 65
lambda_sum_smallest, 65
length_expr, 66
log1p_atom, 68
log1p_expr *(*log1p_atom*)*, 68
log_det, 66

log_log_curvature, 67
log_log_curvature(), *19*
log_normcdf, 67
log_sum_exp, 68
loggamma, 69
logistic, 69

make_sparse_diagonal_matrix, 70
math_atoms, *23*, 70
Matrix::Matrix, *9*
Matrix::sparseVector, *9*
matrix_frac, 73
matrix_trace, 73
max_elemwise, 74
max_entries, *72, 73*, 74
Maximize, 75, *77*, *83*, *93*
min_elemwise, 75
min_entries, *72, 73*, 76
Minimize, *75*, 76, *83*, *93*
mixed_norm, 77
MOSEK_SOLVER *(*solver-constants*)*, 111
multiply, 77

name, 78
neg, 78
NonNeg, *43*, 79, *80*, *129*
NonPos, *43*, 79, *79*, *129*
norm, *70*
norm *(*math_atoms*)*, 70
norm1, 81
norm2, 81
norm_inf, 80
norm_nuc, 80
normcdf, 82
Not, *41*, 82
Not(), *8*, *41, 42*, *84*, *129*

objective, 83
objective(), *16*
one_minus_pos, 83
OPTIMAL, *115*
OPTIMAL *(*status-constants*)*, 115
OPTIMAL_INACCURATE *(*status-constants*)*, 115
Or, *42*, 84
Or(), *8*, *41, 42*, *82*, *129*
OSQP_SOLVER *(*solver-constants*)*, 111
outer, *70, 71*
outer *(*math_atoms*)*, 70

p_norm, *21*, 85
p_norm(), *81*
param_backward *(*reduction-chain-rule*)*, 100
param_dict, 85
param_forward *(*reduction-chain-rule*)*, 100
param_id_map *(*reduction-id-map*)*, 101
Parameter, *16*, *85*, 86, *86, 87*, *122*
parameters, 87
partial_optimize, 87
partial_trace, 88
partial_transpose, 89
perspective, 90
pf_eigenvalue, 90
PIQP_SOLVER *(*solver-constants*)*, 111
pos, 91
PowCone3D, 91, *92*
PowConeND, *91*, 92
power, *22*, *72, 73*
power *(*cvxr_power*)*, 22
power(), *129*
Problem, *16*, *37*, *49*, *53, 54*, *57*, *75*, *77*, *83*, *86*, *88*, 93, *94–96*, *98*, *108*, *110*, *114, 115*, *120*, *122*, *127*
Problem(), *16*, *83*
problem_data, *37*, 93, *95, 96*, *111*, *113*, *120*
problem_solution, 94
problem_status, 95, *115*
problem_unpack_results, 95, *111*
problem_unpack_results(), *121*
prod_entries, *72*, 96
PSD, 97
PSD(), *7*
psolve, *93*, 97, *104*, *112, 113*, *116*
psolve(), *11*, *25, 26*, *38*, *88*, *127*
ptp, 99

quad_form, 99
quad_over_lin, 100

reduction-chain-rule, 100
reduction-id-map, 101
rel_entr, 102
reshape_expr, 102
residual, 103
resolvent, 103

sample_bounds, 104

sample_bounds<- (sample_bounds), 104
scalar_product, 104
scalar_product(), *125*
scalarize, 105
scalene, 106
SCIP_SOLVER (solver-constants), 111
SCS_SOLVER (solver-constants), 111
sd, *70*
sd (math_atoms), 70
set_excluded_solvers (available_solvers), 9
set_excluded_solvers(), *10*
set_label, 106
set_label(), *34, 35*, *63*
sigma_max, *21*, 107
size, 107
size(), *53*, *59*, *61*
size_metrics, 108, *108*
SizeMetrics, 108
SOC, 109
solution, *94, 95*, 110
solve_via_data, *95, 96*, 110, *120*
solver-constants, 111
solver_default_param, *98*, 112
SOLVER_ERROR (status-constants), 115
solver_opts, *98*, 113
solver_stats, *97, 98*, 114
square, 114
status, *95*, *98*, 115, *116*
status-constants, 115
std (cvxr_std), 23
sum_entries, *72, 73*, 116
sum_largest, 117
sum_smallest, 117
sum_squares, 118

to_latex, 118
total_variation, 119
total_variation(), *120*
tr_inv, 119
tv, 120

UNBOUNDED, *115*
UNBOUNDED (status-constants), 115
UNBOUNDED_INACCURATE (status-constants), 115
UNO_SOLVER (solver-constants), 111
unpack_results, 120
upper_tri, 121, *125*

USER_LIMIT (status-constants), 115

value, *96, 97*, 121
value(), *88*
value<-, 122
var, *70*
var (math_atoms), 70
var_backward (reduction-chain-rule), 100
var_dict, 122
var_forward (reduction-chain-rule), 100
var_id_map (reduction-id-map), 101
Variable, *8*, *36*, *41*, *82*, *84*, *88*, *93*, *104*, *122*, 123, *124*, *129*
variables, 124
vdot, 124
vdot(), *22*, *24*, *105*
vec, 125
vec(), *24*
vec_to_upper_tri, 125
violation, 126
visualize, 126
vstack, 127

xexp, 128
Xor, *41*, 128
Xor(), *8*, *41, 42*, *82*, *84*
XPRESS_SOLVER (solver-constants), 111

Zero, *31*, 129