

Package ‘CVXR’

August 24, 2026

Type Package

Title Disciplined Convex Optimization

Version 1.9.2

Date 2026-08-23

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](https://doi.org/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), Rcplex (>= 0.3), cccp (>= 0.3), piqp (>= 0.6), scip, xpress, diffcp (>= 0.1.2), ipopt, sparsediff, 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 CRAN

Date/Publication 2026-08-24 05:21:00 UTC

Contents

And	6
as_cvxr_expr	7
available_solvers	8
backward	9
bmat	10
CallbackParam	10
cdiac	11
ceil_expr	12
condition_number	13
Constant	13
constants	14
constraints	15
conv	15
convolve	16
cummax_expr	17
cumsum_axis	17
curvature	18
cvar	18
cvxr_diff	19
cvxr_mean	19
cvxr_norm	20
cvxr_outer	20
cvxr_std	21
cvxr_var	21
deep_flatten	22
delta	22
derivative	23
DiagMat	24
DiagVec	24
diff_pos	25
dist_ratio	25
dotsort	26

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

is_nonpos	51
is_nsd	52
is_psd	52
is_pwl	53
is_qp	53
is_quadratic	54
is_quasiconcave	54
is_quasiconvex	55
is_quasilinear	55
is_scalar	56
is_smooth	56
is_symmetric	57
is_vector	57
is_zero	58
kl_div	58
kron	59
label	59
label<-	60
lambda_max	60
lambda_min	61
lambda_sum_largest	61
lambda_sum_smallest	62
length_expr	62
log1p_atom	63
loggamma	63
logistic	64
log_det	64
log_normcdf	65
log_sum_exp	65
make_sparse_diagonal_matrix	66
math_atoms	66
matrix_frac	69
matrix_trace	69
Maximize	70
max_elemwise	70
max_entries	71
Minimize	71
min_elemwise	72
min_entries	72
mixed_norm	73
multiply	73
name	74
neg	74
NonNeg	75
NonPos	75
norm1	76
norm2	76
normcdf	77

norm_inf	77
norm_nuc	78
Not	78
objective	79
one_minus_pos	80
Or	80
Parameter	81
parameters	82
param_dict	82
partial_optimize	83
partial_trace	84
partial_transpose	85
perspective	85
pf_eigenvalue	86
pos	86
PowCone3D	87
PowConeND	87
power	88
Problem	89
problem_data	90
problem_solution	90
problem_status	91
problem_unpack_results	92
prod_entries	93
PSD	93
psolve	94
ptp	95
p_norm	96
quad_form	96
quad_over_lin	97
reduction-chain-rule	97
reduction-id-map	98
rel_entr	99
reshape_expr	99
residual	100
resolvent	100
sample_bounds	101
scalarize	101
scalar_product	102
scalene	103
set_label	103
sigma_max	104
size	104
SizeMetrics	105
size_metrics	106
SOC	106
solution	107
solver-constants	107

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

Index**129**

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

```
## Not run:
x <- Variable(boolean = TRUE)
y <- Variable(boolean = TRUE)
both <- x & y           # operator syntax
both <- And(x, y)       # functional syntax
all3 <- And(x, y, z)    # n-ary

## End(Not run)
```

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	<i>List available solvers</i>
-------------------	-------------------------------

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	<i>Construct a Block Matrix</i>
------	---------------------------------

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	<i>Callback Parameter</i>
---------------	---------------------------

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 <code>value(param)</code> .
shape	Integer vector of length 1 or 2 giving parameter dimensions. Defaults to <code>c(1, 1)</code> (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., <code>nonneg</code> , <code>nonpos</code>).

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

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 $\geq x$) of each element. This atom is quasiconvex and quasi-concave but NOT convex or concave, so it can only be used in DQCP problems (solved with `qcp = TRUE`).

Usage

```
ceil_expr(x)
```

Arguments

x A CVXR expression.

Value

A Ceil expression.

See Also

[floor_expr](#)

condition_number	<i>Condition number of a PSD matrix</i>
------------------	---

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	<i>Create a Constant Expression</i>
----------	-------------------------------------

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	<i>Get Problem Constraints (read-only)</i>
-------------	--

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	<i>1D discrete convolution</i>
------	--------------------------------

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	<i>1D discrete convolution (numpy-style)</i>
----------	--

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; for plain numeric input it computes the same numpy-style convolution the atom does (`numpy.convolve(a, b) == stats::convolve(a, rev(b), type = "open")`), so the numeric and Expression paths agree.

Usage

```
convolve(a, b)
```

Arguments

a, b	Expressions or numeric vectors; at least one must be constant when building an atom.
------	--

Details

CVXR masks `stats::convolve` when attached (R reports this on load). If you specifically want `stats'` circular cross-correlation or its other options, call [convolve](#) directly.

Value

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

cummax_expr	<i>Cumulative maximum along an axis</i>
-------------	---

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	<i>Cumulative sum along an axis</i>
-------------	-------------------------------------

Description

Cumulative sum along an axis

Usage

cumsum_axis(x, axis = NULL)

Arguments

- x An Expression
- axis NULL (all), 1 (across rows), or 2 (down columns)

Value

A Cumsum atom

curvature

Get Expression Curvature

Description

Returns the DCP curvature of an expression as a string.

Usage

```
curvature(x)
```

Arguments

x A CVXR expression.

Value

Character: "CONSTANT", "AFFINE", "CONVEX", "CONCAVE", or "UNKNOWN".

See Also

[is_convex\(\)](#), [is_concave\(\)](#), [is_affine\(\)](#), [is_constant\(\)](#)

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	<i>Compute kth Order Differences of an Expression</i>
-----------	---

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	<i>Mean of an expression</i>
-----------	------------------------------

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	<i>Compute a norm of an expression</i>
-----------	--

Description

Compute a norm of an expression

Usage

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

Arguments

x	An Expression
p	Norm type: 1, 2, Inf, or "fro" (Frobenius)
axis	NULL (all), 0 (columns), or 1 (rows)
keepdims	Logical
max_denom	Integer max denominator

Value

A norm atom

cvxr_outer	<i>Outer product of two vectors</i>
------------	-------------------------------------

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 $(\text{length}(x), \text{length}(y))$.

See Also

[vdot\(\)](#)

cvxr_std	<i>Standard deviation of an expression</i>
----------	--

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	<i>Variance of an expression</i>
----------	----------------------------------

Description

Computes the variance. Only supports full reduction (axis = NULL).

Usage

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

Arguments

x	An Expression or numeric value.
axis	NULL only (axis reduction not yet supported).
keepdims	Logical; keep reduced dimension?
ddof	Degrees of freedom correction (default 0, population variance).

Value

An Expression representing the variance.

deep_flatten	<i>Recursively flatten a nested list of expressions into one column vector</i>
--------------	--

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	<i>Access the perturbation delta of a Variable or Parameter</i>
-------	---

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	<i>Apply the derivative of the solution map to perturbations</i>
------------	--

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	<i>Extract Diagonal from a Matrix</i>
---------	---------------------------------------

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 $c(n - \text{abs}(k), 1)$.

See Also

[DiagVec](#)

DiagVec	<i>Vector to Diagonal Matrix</i>
---------	----------------------------------

Description

Constructs a diagonal matrix from a column vector. If $k \neq 0$, 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 $c(n + \text{abs}(k), n + \text{abs}(k))$.

See Also

[DiagMat](#)

diff_pos	<i>The difference $x - y$ with domain $x > y > 0$</i>
----------	---

Description

Equivalent to $x * \text{one_minus_pos}(y / x)$.

Usage

diff_pos(x, y)

Arguments

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

Value

A product expression

dist_ratio	<i>Distance ratio</i>
------------	-----------------------

Description

Computes $\text{norm}(x - a)_2 / \text{norm}(x - b)_2$, 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	<i>Weighted sorted dot product</i>
---------	------------------------------------

Description

Computes $\langle \text{sort}(\text{vec}(X)), \text{sort}(\text{vec}(W)) \rangle$ 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	<i>Direct Standardization: Population</i>
-------	---

Description

Randomly generated data for direct standardization example. Sex was drawn from a Bernoulli distribution, and age was drawn from a uniform distribution on $10, \dots, 60$. 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	<i>Direct Standardization: Sample</i>
--------	---------------------------------------

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	<i>Get the Dual Value of a Constraint</i>
------------	---

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	<i>Create an Exponential Cone Constraint</i>
---------	--

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	<i>Conjugate-Transpose of an Expression</i>
--------	---

Description

Equivalent to CVXPY's `.H` property. For real expressions, returns $t(x)$. For complex expressions, returns $\text{Conj}(t(x))$.

Usage

expr_H(x)

Arguments

x	A CVXR Expression.
---	--------------------

Value

The conjugate-transpose expression.

expr_sign	<i>Get the DCP Sign of an Expression</i>
-----------	--

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	<i>Unity resolvent (I - X) inverse for positive square matrix X</i>
---------------	---

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))
## Not run: 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 $\leq x$) 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`).

Usage

```
floor_expr(x)
```

Arguments

x	A CVXR expression.
---	--------------------

Value

A Floor expression.

See Also

[ceil_expr](#)

format_labeled	<i>Pretty-print an expression with labels substituted</i>
----------------	---

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	<i>Maximum generalized eigenvalue</i>
----------------	---------------------------------------

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	<i>(Weighted) geometric mean of a vector</i>
----------	--

Description

(Weighted) geometric mean of a vector

Usage

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

Arguments

x	An Expression (vector)
p	Numeric weight vector (default: uniform)
max_denom	Maximum denominator for rational approximation
approx	If TRUE (default), use SOC approximation. If FALSE, use exact power cone.

Value

A GeoMean or GeoMeanApprox atom

get_bounds	<i>Lower/Upper Bounds of a Leaf</i>
------------	-------------------------------------

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	<i>Get Problem Data for a Solver (deprecated)</i>
------------------	---

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	<i>Geometric matrix multiplication A diamond X</i>
---------	--

Description

Computes the geometric matrix product where $(A \text{ diamond } X)_{ij} = \prod_k X_{kj}^A_{ik}$. 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))
## Not run: psolve(prob, gp = TRUE)
```

gradient	<i>Access the gradient of a Variable or Parameter</i>
----------	---

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	<i>Harmonic mean: $n / \text{sum}(1/x_i)$</i>
---------------	--

Description

Harmonic mean: $n / \text{sum}(1/x_i)$

Usage

harmonic_mean(x)

Arguments

x An Expression (must be positive for DCP)

Value

An Expression representing the harmonic mean

hstack	<i>Horizontal concatenation of expressions</i>
--------	--

Description

Horizontal concatenation of expressions

Usage

hstack(...)

Arguments

... Expressions (same number of rows)

Value

An HStack atom

huber	<i>Create a Huber loss atom</i>
-------	---------------------------------

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	<i>Get Expression ID</i>
----	--------------------------

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	<i>Logical Biconditional</i>
-----	------------------------------

Description

Logical biconditional: $x \Leftrightarrow y$. 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

```
## Not run:
x <- Variable(boolean = TRUE)
y <- Variable(boolean = TRUE)
expr <- iff(x, y)

## End(Not run)
```

implies	<i>Logical Implication</i>
---------	----------------------------

Description

Logical implication: $x \Rightarrow y$. 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 $\neg x \mid y$.

See Also

[iff\(\)](#), [Not\(\)](#), [And\(\)](#), [Or\(\)](#), [Xor\(\)](#)

Examples

```
## Not run:
x <- Variable(boolean = TRUE)
y <- Variable(boolean = TRUE)
expr <- implies(x, y)

## End(Not run)
```

indicator	<i>Indicator function for constraints</i>
-----------	---

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	<i>Create an Inequality Constraint</i>
------------	--

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	<i>List installed solvers</i>
-------------------	-------------------------------

Description

Returns the names of solvers whose R packages are available.

Usage

```
installed_solvers()
```

Value

A character vector of solver names.

inv_pos	<i>Inverse position: x^{-1} (for $x > 0$)</i>
---------	--

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	<i>Reciprocal of product of entries</i>
----------	---

Description

Computes the reciprocal of the product of entries. Equivalent to $\text{geo_mean}(x)^{-n}$ 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	<i>Check if an Expression is Affine</i>
-----------	---

Description

Check if an Expression is Affine

Usage

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

Arguments

x	An expression object.
...	Not used.

Value

Logical scalar.

is_atom_smooth	<i>Check if an Atom is Smooth</i>
----------------	-----------------------------------

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	<i>Check if an Expression is Concave</i>
------------	--

Description

Check if an Expression is Concave

Usage

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

Arguments

x	An expression object.
...	Not used.

Value

Logical scalar.

is_constant	<i>Check if an Expression is Constant</i>
-------------	---

Description

Check if an Expression is Constant

Usage

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

Arguments

x	An expression object.
...	Not used.

Value

Logical scalar.

is_convex	<i>Check if an Expression is Convex</i>
-----------	---

Description

Check if an Expression is Convex

Usage

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

Arguments

x	An expression object.
...	Not used.

Value

Logical scalar.

is_dcp	<i>Check if an Expression is DCP-Compliant</i>
--------	--

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	<i>Check if a Constraint is DGP-Compliant</i>
--------	---

Description

Check if a Constraint is DGP-Compliant

Usage

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

Arguments

x	A constraint object.
...	Not used.

Value

Logical scalar.

is_dnlp	<i>Check if an Expression or Problem is DNLP-Compliant</i>
---------	--

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	<i>Check DPP Compliance</i>
--------	-----------------------------

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 <code>is_dpp(context=...)</code> .

Value

Logical scalar.

is_dqcp	<i>Check if Expression is DQCP-Compliant</i>
---------	--

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

<code>x</code>	An expression object.
<code>...</code>	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

<code>x</code>	An expression object.
<code>...</code>	Not used.

Value

Logical scalar.

See Also

[is_dnlp](#), [is_linearizable_concave](#)

is_log_log_affine	<i>Check if Expression is Log-Log Affine</i>
-------------------	--

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	<i>Check if Expression is Log-Log Concave</i>
--------------------	---

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	<i>Check if Expression is Log-Log Convex</i>
-------------------	--

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	<i>Check if a Problem is a Linear Program</i>
-------	---

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	<i>Is the Expression a Matrix?</i>
-----------	------------------------------------

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	<i>Check if a Problem is Mixed-Integer</i>
------------------	--

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	<i>Check if Expression is Non-Negative</i>
-----------	--

Description

Check if Expression is Non-Negative

Usage

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

Arguments

x	An expression object.
...	Not used.

Value

Logical scalar.

is_nonpos	<i>Check if Expression is Non-Positive</i>
-----------	--

Description

Check if Expression is Non-Positive

Usage

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

Arguments

x	An expression object.
...	Not used.

Value

Logical scalar.

is_nsd	<i>Check if Expression is Negative Semidefinite</i>
--------	---

Description

Check if Expression is Negative Semidefinite

Usage

is_nsd(x, ...)

Arguments

- | | |
|-----|-----------------------|
| x | An expression object. |
| ... | Not used. |

Value

Logical scalar.

is_psd	<i>Check if Expression is Positive Semidefinite</i>
--------	---

Description

Check if Expression is Positive Semidefinite

Usage

is_psd(x, ...)

Arguments

- | | |
|-----|-----------------------|
| x | An expression object. |
| ... | Not used. |

Value

Logical scalar.

is_pwl	<i>Check if Expression is Piecewise Linear</i>
--------	--

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	<i>Check if a Problem is a Quadratic Program</i>
-------	--

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	<i>Check if an Expression is Quadratic</i>
--------------	--

Description

Check if an Expression is Quadratic

Usage

is_quadratic(x, ...)

Arguments

- | | |
|-----|-----------------------|
| x | An expression object. |
| ... | Not used. |

Value

Logical scalar.

is_quasiconcave	<i>Check if Expression is Quasiconcave</i>
-----------------	--

Description

Check if Expression is Quasiconcave

Usage

is_quasiconcave(x, ...)

Arguments

- | | |
|-----|-----------------------|
| x | An expression object. |
| ... | Not used. |

Value

Logical scalar.

is_quasiconvex	<i>Check if Expression is Quasiconvex</i>
----------------	---

Description

Check if Expression is Quasiconvex

Usage

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

Arguments

x	An expression object.
...	Not used.

Value

Logical scalar.

is_quasilinear	<i>Check if Expression is Quasilinear</i>
----------------	---

Description

Check if Expression is Quasilinear

Usage

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

Arguments

x	An expression object.
...	Not used.

Value

Logical scalar.

is_scalar	<i>Is the Expression a Scalar?</i>
-----------	------------------------------------

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	<i>Check if an Expression is Smooth</i>
-----------	---

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	<i>Check if Expression is Symmetric</i>
--------------	---

Description

Check if Expression is Symmetric

Usage

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

Arguments

x	An expression object.
...	Not used.

Value

Logical scalar.

is_vector	<i>Is the Expression a Vector?</i>
-----------	------------------------------------

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	<i>Check if Expression is Zero</i>
---------	------------------------------------

Description

Check if Expression is Zero

Usage

is_zero(x, ...)

Arguments

- | | |
|-----|-----------------------|
| x | An expression object. |
| ... | Not used. |

Value

Logical scalar.

kl_div	<i>KL Divergence: $x*\log(x/y) - x + y$</i>
--------	--

Description

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

Usage

kl_div(x, y)

Arguments

- | | |
|---|---------------|
| x | An Expression |
| y | An Expression |

Value

A KIDiv atom

kron	<i>Kronecker product of two expressions</i>
------	---

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	<i>Get the label of an expression</i>
-------	---------------------------------------

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<-	<i>Set the label of an expression</i>
---------	---------------------------------------

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	<i>Maximum eigenvalue</i>
------------	---------------------------

Description

Maximum eigenvalue

Usage

```
lambda_max(A)
```

Arguments

A	A square matrix expression
---	----------------------------

Value

An expression representing the maximum eigenvalue of A

lambda_min	<i>Minimum eigenvalue</i>
------------	---------------------------

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	<i>Sum of largest k eigenvalues</i>
--------------------	-------------------------------------

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 integer)

Value

An expression representing the sum of the k largest eigenvalues

<code>lambda_sum_smallest</code>	<i>Sum of smallest k eigenvalues</i>
----------------------------------	---

Description

Sum of smallest k eigenvalues

Usage

`lambda_sum_smallest(A, k)`

Arguments

- | | |
|----------------|--|
| <code>A</code> | A square matrix expression |
| <code>k</code> | Number of smallest eigenvalues to sum (positive integer) |

Value

An expression representing the sum of the k smallest eigenvalues

<code>length_expr</code>	<i>Length of a Vector (Last Nonzero Index)</i>
--------------------------	--

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

- | | |
|----------------|---------------------------|
| <code>x</code> | A CVXR vector expression. |
|----------------|---------------------------|

Value

A Length expression (scalar).

See Also

[ceil_expr](#), [floor_expr](#)

log1p_atom	$\text{Log}(1 + x) - \text{elementwise}$
------------	--

Description

$\text{Log}(1 + x) - \text{elementwise}$

Usage

log1p_atom(x)

log1p_expr(x)

Arguments

x	An Expression
---	---------------

Value

A Log1p atom

loggamma	<i>Elementwise log of the gamma function</i>
----------	--

Description

Piecewise linear approximation of $\log(\text{gamma}(x))$. 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 $\log(\text{gamma}(x))$.

logistic	<i>Logistic function: $\log(1 + \exp(x)) - \text{elementwise}$</i>
----------	---

Description

Logistic function: $\log(1 + \exp(x)) - \text{elementwise}$

Usage

logistic(x)

Arguments

x An Expression

Value

A Logistic atom

log_det	<i>Log-determinant</i>
---------	------------------------

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_normcdf	<i>Elementwise log of the standard normal CDF</i>
-------------	---

Description

Quadratic approximation of $\log(\text{pnorm}(x))$ 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	<i>Log-sum-exp: $\log(\text{sum}(\exp(x)))$</i>
-------------	--

Description

Log-sum-exp: $\log(\text{sum}(\exp(x)))$

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

```
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	<i>Standard R Functions for CVXR Expressions</i>
------------	--

Description

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

For CVXR expressions, computes the matrix/vector norm atom. For other inputs, falls through to `Matrix::norm` which dispatches via S4 for both Matrix and base matrix objects.

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` 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, nrow, ncol, names = TRUE, k = 0L)

```

Arguments

x	An Expression, matrix, vector, or scalar.
type	Norm type: "1", "2" (default), "I"/"i" (infinity), "F"/"f" (Frobenius).
...	For non-Expression inputs: passed to outer .
X	An Expression or numeric.
Y	An Expression or numeric.
nrow	For non-Expression: passed to <code>Matrix::diag</code> .
ncol	For non-Expression: passed to <code>Matrix::diag</code> .
names	For non-Expression: passed to <code>Matrix::diag</code> .
k	Integer diagonal offset for Expressions only. <code>k = 0</code> (default) is the main diagonal.

Details

The `k` parameter (off-diagonal offset) is only available for Expression inputs. For the full-featured version with `k` on non-Expression inputs, use [DiagVec](#) or [DiagMat](#) directly.

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 (MIP)

Summary group (via S3 group generic)

`sum(x)` Sum all entries (affine)
`max(x)` Maximum entry (convex)
`min(x)` Minimum entry (concave)

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	<i>Matrix fractional function</i>
-------------	-----------------------------------

Description

Computes $\text{trace}(X^T P^{-1} X)$. 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 $\text{trace}(X^T P^{-1} X)$

matrix_trace	<i>Trace of a square matrix expression</i>
--------------	--

Description

For matrix_trace(A ** B), uses the $O(n^2)$ identity $\text{trace}(A ** B) = \text{sum}(A * \text{t}(B))$ 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)

Maximize	<i>Create a Maximization Objective</i>
----------	--

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)
```

max_elemwise	<i>Elementwise maximum of expressions</i>
--------------	---

Description

Elementwise maximum of expressions

Usage

```
max_elemwise(...)
```

Arguments

... Expressions (at least 2)

Value

A Maximum atom

max_entries	<i>Maximum entry of an expression</i>
-------------	---------------------------------------

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

Minimize	<i>Create a Minimization Objective</i>
----------	--

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)
```

min_elemwise	<i>Elementwise minimum of expressions</i>
--------------	---

Description

Elementwise minimum of expressions

Usage

```
min_elemwise(...)
```

Arguments

... Expressions (at least 2)

Value

A Minimum atom

min_entries	<i>Minimum entry of an expression</i>
-------------	---------------------------------------

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

mixed_norm	<i>Mixed norm (L_p, q norm): column-wise p-norm, then q-norm</i>
------------	---

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	<i>Elementwise multiplication (deprecated)</i>
----------	--

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	<i>Get Expression Name</i>
------	----------------------------

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	<i>Negative part: $-\min(x, 0)$</i>
-----	--

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](#)

norm1	<i>L1 norm of an expression</i>
-------	---------------------------------

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	<i>Euclidean norm (deprecated alias)</i>
-------	--

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.

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	<i>Nuclear norm (sum of singular values)</i>
----------	--

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

Not	<i>Logical NOT</i>
-----	--------------------

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

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

## End(Not run)
```

objective	<i>Get Problem Objective (read-only)</i>
-----------	--

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	<i>The difference 1 - x with domain (0, 1)</i>
---------------	--

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))
## Not run: psolve(prob, gp = TRUE)
```

Or	<i>Logical OR</i>
----	-------------------

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

```
## Not run:
x <- Variable(boolean = TRUE)
y <- Variable(boolean = TRUE)
either <- x | y      # operator syntax
either <- Or(x, y)    # functional syntax
any3 <- Or(x, y, z)   # n-ary

## End(Not run)
```

Parameter	<i>Create a Parameter</i>
-----------	---------------------------

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, " γ ". 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	<i>Get the Parameters in an Expression</i>
------------	--

Description

Get the Parameters in an Expression

Usage

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

Arguments

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

Value

List of [Parameter](#) objects.

param_dict	<i>Get all Parameters of a Problem as a Named List</i>
------------	--

Description

Mirrors CVXPY's Problem.param_dict property (cvxpy/problems/problem.py:260-264): 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.

partial_optimize	<i>Partial optimization transform</i>
------------------	---------------------------------------

Description

Builds an Expression representing the optimal value of prob as a function of the variables you choose NOT to optimise over. Useful for two-stage / hierarchical optimisation, 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 optimise.
opt_vars	Optional list of Variables to optimise 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

```
## Not run:
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)).

## End(Not run)
```

partial_trace	<i>Partial trace of a tensor product expression</i>
---------------	---

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	<i>Partial transpose of a tensor product expression</i>
-------------------	---

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

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

Value

An Expression representing the partial transpose

perspective	<i>Perspective Transform</i>
-------------	------------------------------

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

<code>f</code>	A scalar convex or concave Expression.
<code>s</code>	A nonneg Variable (scalar).
<code>f_recession</code>	Optional recession function for handling $s = 0$.

Value

A Perspective expression.

pf_eigenvalue	<i>Perron-Frobenius eigenvalue of a positive matrix</i>
---------------	---

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))
## Not run: psolve(prob, gp = TRUE)
```

pos	<i>Positive part: $\max(x, 0)$</i>
-----	---

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](#)

power	<i>Create a Power atom</i>
-------	----------------------------

Description

Create a Power atom

Usage

`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 <code>exp(p * log(x))</code> .
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.

Problem	<i>Create an Optimization Problem</i>
---------	---------------------------------------

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 <code>==</code> , <code><=</code> , <code>>=</code> 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	<i>Get Problem Data for a Solver</i>
--------------	--------------------------------------

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	<i>Get the Raw Solution Object (deprecated)</i>
------------------	---

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	<i>Get the Solution Status of a Problem (deprecated)</i>
----------------	--

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

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

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	<i>Product of entries along an axis</i>
--------------	---

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))
## Not run: psolve(prob, gp = TRUE)
```

PSD	<i>Create a Positive Semidefinite Constraint</i>
-----	--

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	<i>Peak-to-peak (range): $\max(x) - \min(x)$</i>
-----	---

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 $\max(x) - \min(x)$.

p_norm	<i>General p-norm of an expression</i>
--------	--

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

quad_form	<i>Quadratic form $x^T P x$</i>
-----------	--

Description

When x is constant, returns $t(\text{Conj}(x)) \%P\% x$ (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	<i>Sum of squares divided by a scalar</i>
---------------	---

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	<i>Reduction chain-rule hooks (dict-in / dict-out)</i>
----------------------	--

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	<i>Reduction leaf-id maps</i>
------------------	-------------------------------

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	<i>Relative Entropy: $x \cdot \log(x/y)$</i>
----------	---

Description

Relative Entropy: $x \cdot \log(x/y)$

Usage

rel_entr(x, y)

Arguments

x	An Expression
y	An Expression

Value

A RelEntr atom

reshape_expr	<i>Reshape an expression to a new shape</i>
--------------	---

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	<i>Get the Residual of a Constraint</i>
----------	---

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	<i>Resolvent inverse($sI - X$)</i>
-----------	---

Description

Equivalent to $(1/s) * \text{eye_minus_inv}(X / s)$.

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.

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`.

Examples

```
x <- Variable()
objs <- list(Minimize(square(x)), Minimize(square(x - 1)))
obj <- scalarize$weighted_sum(objs, c(1, 1))
## Not run: psolve(Problem(obj))
```

scalar_product	<i>Scalar product (alias for vdot)</i>
----------------	--

Description

Scalar product (alias for `vdot`)

Usage

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

Arguments

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

Value

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

See Also

[vdot\(\)](#)

scalene	<i>Scalene penalty: $\alpha * \text{pos}(x) + \beta * \text{neg}(x)$</i>
---------	---

Description

Scalene penalty: $\alpha * \text{pos}(x) + \beta * \text{neg}(x)$

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	<i>Attach a label to an expression</i>
-----------	--

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 <code>as.character</code>). 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\(\)](#)

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 $\text{big} * \text{small}^2$ over all data blocks, where big/small are the larger/ smaller dimension of the block.

Value

A SizeMetrics object.

size_metrics	<i>Get Size Metrics for a Problem</i>
--------------	---------------------------------------

Description

Mirrors CVXPY's Problem.size_metrics property (cvxpy/problems/problem.py:486-490, class at lines 1690-1752): returns a SizeMetrics object summarising the problem's scale.

Usage

size_metrics(x, ...)

Arguments

x A [Problem](#) object.
... Not used.

Value

A SizeMetrics object with seven numeric fields.

SOC	<i>Create a Second-Order Cone Constraint</i>
-----	--

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	<i>Get the Raw Solution Object</i>
----------	------------------------------------

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.

solver-constants	<i>Solver Name Constants</i>
------------------	------------------------------

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

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

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.

 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](#)

square	<i>Square of an expression: x^2</i>
--------	--

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	<i>Get the Solution Status of a Problem</i>
--------	---

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	<i>Solution Status Constants</i>
------------------	----------------------------------

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	<i>Sum the entries of an expression</i>
-------------	---

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 <code>apply(X,1,sum)</code>), or 2 (column-wise, like <code>apply(X,2,sum)</code>). |
| keepdims | Logical: if TRUE, keep the reduced dimension as size 1. |

Value

A SumEntries expression.

sum_largest	<i>Sum of k largest entries</i>
-------------	---------------------------------

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	<i>Sum of k smallest entries</i>
--------------	----------------------------------

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	<i>Sum of squares (= quad_over_lin(x, 1))</i>
-------------	---

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

total_variation	<i>Total variation of a vector or matrix</i>
-----------------	--

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

to_latex	<i>Convert CVXR Object to LaTeX</i>
----------	-------------------------------------

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
```

tr_inv	<i>Trace of matrix inverse</i>
--------	--------------------------------

Description

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

Usage

```
tr_inv(X)
```

Arguments

X	A square PSD matrix expression
---	--------------------------------

Value

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

tv	<i>Total variation (deprecated alias)</i>
----	---

Description**[Deprecated]****Usage**

tv(...)

Arguments... Arguments passed to `total_variation()`.**Details**Use `total_variation()` instead.**See Also**`total_variation()`

unpack_results	<i>Unpack Results (backward-compatible alias)</i>
----------------	---

Description**[Deprecated]****Usage**

unpack_results(problem, solution, chain, inverse_data)

Arguments

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

DetailsUse `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	<i>Extract strict upper triangle of a square matrix</i>
-----------	---

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	<i>Get the Numeric Value of an Expression</i>
-------	---

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<-	<i>Set the Value of a Leaf Expression</i>
---------	---

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).

Variable	<i>Create an Optimization Variable</i>
----------	--

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 <code>value(var) <- val</code> .
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: <code>nonneg</code> , <code>nonpos</code> , <code>PSD</code> , <code>NSD</code> , <code>symmetric</code> , <code>boolean</code> , <code>integer</code> , 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.

var_dict

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

Mirrors CVXPY's `Problem.var_dict` property (`cvxpy/problems/problem.py:267-271`): 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.

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	<i>Vectorize an expression (column vector)</i>
-----	--

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	<i>Reshape a vector into an upper triangular matrix</i>
------------------	---

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	<i>Get the Violation of a Constraint</i>
-----------	--

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	<i>Visualize the Canonicalization Pipeline of a CVXR Problem</i>
-----------	--

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

```
## Not run:
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
visualize(prob, output = "html", solver = TRUE)

## End(Not run)
```

vstack

Vertical concatenation of expressions

Description

Vertical concatenation of expressions

Usage

vstack(...)

Arguments

... Expressions (same number of columns)

Value

A VStack atom

xexp	$x * \exp(x) - \text{elementwise}$
------	------------------------------------

Description

$x * \exp(x) - \text{elementwise}$

Usage

xexp(x)

Arguments

x An Expression

Value

An Xexp atom

Xor	<i>Logical XOR</i>
-----	--------------------

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 $\wedge$ operator is used for `power()`, so Xor is functional syntax only.

Value

A Xor expression.

See Also

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

Examples

```
## Not run:
x <- Variable(boolean = TRUE)
y <- Variable(boolean = TRUE)
exclusive <- Xor(x, y)

## End(Not run)
```

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`

%>>%*Positive Semidefinite Constraint Operator*

Description

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

Usage

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

Arguments

`e1, e2` CVXR expressions or numeric matrices.

Value

A PSD constraint object.

See Also

[PSD\(\)](#)

Examples

```
## Not run:
X <- Variable(c(3, 3), symmetric = TRUE)
constr <- X %>>% diag(3) # X - I is PSD

## End(Not run)
```

%<<%*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 $A \preccurlyeq B$.

Usage

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

Arguments

`e1, e2` CVXR expressions or numeric matrices.

Value

A PSD constraint object.

See Also

[PSD\(\)](#)

Examples

```
## Not run:  
X <- Variable(c(3, 3), symmetric = TRUE)  
constr <- X %<<% diag(3) # I - X is PSD (X is NSD relative to I)  
  
## End(Not run)
```

Index

- * **data**
 - cdiac, 11
 - dspop, 26
 - dssamp, 27
- %<<%, 127
- %>>%, 127
- And, 6
- And(), 38, 39, 78, 81, 126
- as.matrix(), 8
- as_cvxr_expr, 7
- available_solvers, 8
- backward, 9, 94
- backward(), 23
- bmat, 10
- CallbackParam, 10
- cdiac, 11
- ceil_expr, 12, 31, 62
- CLARABEL_SOLVER (solver-constants), 107
- condition_number, 13
- Constant, 7, 8, 13, 14
- constants, 14
- constraints, 15
- constraints(), 79
- conv, 15, 16
- convolve, 16, 16
- COPT_SOLVER (solver-constants), 107
- CPLEX_SOLVER (solver-constants), 107
- cummax_expr, 17
- cumsum_axis, 17
- curvature, 18
- cvar, 18
- CVXOPT_SOLVER (solver-constants), 107
- cvxr_diff, 19, 68
- cvxr_mean, 19, 68
- cvxr_norm, 20, 68
- cvxr_outer, 20, 68
- cvxr_outer(), 122
- cvxr_std, 21, 68
- cvxr_var, 21, 68
- deep_flatten, 22
- deep_flatten(), 122
- delta, 22
- delta(), 23
- delta<-(delta), 22
- derivative, 23, 94
- derivative(), 9
- diag (math_atoms), 66
- DiagMat, 24, 25, 66–68
- DiagVec, 24, 24, 66–68
- diff_pos, 25
- DIFFCP_SOLVER (solver-constants), 107
- dist_ratio, 25
- dotsort, 26
- dspop, 26, 27
- dssamp, 26, 27
- dual_value, 27, 92, 94
- ECOS_BB_SOLVER (solver-constants), 107
- ECOS_SOLVER (solver-constants), 107
- entr, 28
- Equality, 28, 40, 126
- exclude_solvers (available_solvers), 8
- exclude_solvers(), 8, 9
- ExpCone, 29
- expr_H, 29
- expr_name(), 32
- expr_sign, 30
- eye_minus_inv, 30
- FiniteSet, 31
- floor_expr, 13, 31, 62
- format_labeled, 32
- format_labeled(), 59, 60, 103
- gen_lambda_max, 32
- geo_mean, 33

- get_bounds, 33
- get_problem_data, 34
- GLPK_MI_SOLVER (solver-constants), 107
- GLPK_SOLVER (solver-constants), 107
- gmatmul, 35
- grad(), 83
- gradient, 35
- gradient(), 9
- gradient<- (gradient), 35
- GUROBI_SOLVER (solver-constants), 107
- harmonic_mean, 36
- HIGHS_SOLVER (solver-constants), 107
- hstack, 36
- huber, 37
- id, 37
- iff, 38
- iff(), 7, 39, 78, 81, 126
- implies, 38
- implies(), 7, 38, 78, 81, 126
- include_solvers (available_solvers), 8
- include_solvers(), 9
- indicator, 39
- Inequality, 28, 40, 75
- INFEASIBLE, 112
- INFEASIBLE (status-constants), 112
- INFEASIBLE_INACCURATE (status-constants), 112
- INFEASIBLE_OR_UNBOUNDED (status-constants), 112
- installed_solvers, 40
- installed_solvers(), 9
- inv_pos, 41
- inv_prod, 41
- IPOPT_SOLVER (solver-constants), 107
- is_affine, 42
- is_affine(), 18
- is_atom_smooth, 42
- is_concave, 43
- is_concave(), 18
- is_constant, 43
- is_constant(), 18
- is_convex, 44
- is_convex(), 18
- is_dcp, 44
- is_dgp, 45
- is_dnlp, 45, 47, 48, 56, 94
- is_dpp, 46
- is_dqcp, 46
- is_linearizable_concave, 47, 48, 56
- is_linearizable_convex, 47, 47, 56
- is_log_log_affine, 48
- is_log_log_concave, 48
- is_log_log_convex, 49
- is_lp, 49
- is_matrix, 50
- is_matrix(), 56, 57, 104
- is_mixed_integer, 50
- is_nonneg, 51
- is_nonpos, 51
- is_nsd, 52
- is_psd, 52
- is_pwl, 53
- is_qp, 53
- is_quadratic, 54
- is_quasiconcave, 54
- is_quasiconvex, 55
- is_quasilinear, 55
- is_scalar, 56
- is_scalar(), 50, 57, 104
- is_smooth, 42, 56
- is_symmetric, 57
- is_vector, 57
- is_vector(), 50, 56, 104
- is_zero, 58
- kl_div, 58
- KNITRO_SOLVER (solver-constants), 107
- kron, 59
- label, 59
- label(), 32, 103
- label<-, 60
- lambda_max, 60
- lambda_min, 61
- lambda_sum_largest, 61
- lambda_sum_smallest, 62
- length_expr, 62
- log1p_atom, 63
- log1p_expr (log1p_atom), 63
- log_det, 64
- log_normcdf, 65
- log_sum_exp, 65
- loggamma, 63
- logistic, 64
- make_sparse_diagonal_matrix, 66

- math_atoms, 66, 89
- Matrix::Matrix, 7
- Matrix::sparseVector, 7
- matrix_frac, 69
- matrix_trace, 69
- max_elemwise, 70
- max_entries, 68, 71
- Maximize, 70, 71, 79, 89
- min_elemwise, 72
- min_entries, 68, 72
- Minimize, 70, 71, 79, 89
- mixed_norm, 73
- MOSEK_SOLVER (solver-constants), 107
- multiply, 73

- name, 74
- neg, 74
- NonNeg, 40, 75, 75, 126
- NonPos, 40, 75, 75, 126
- norm (math_atoms), 66
- norm1, 76
- norm2, 76
- norm_inf, 77
- norm_nuc, 78
- normcdf, 77
- Not, 38, 78
- Not(), 7, 38, 39, 81, 126

- objective, 79
- objective(), 15
- one_minus_pos, 80
- OPTIMAL, 112
- OPTIMAL (status-constants), 112
- OPTIMAL_INACCURATE (status-constants), 112
- Or, 39, 80
- Or(), 7, 38, 39, 78, 126
- OSQP_SOLVER (solver-constants), 107
- outer, 66, 67
- outer (math_atoms), 66

- p_norm, 96
- p_norm(), 76
- param_backward (reduction-chain-rule), 97
- param_dict, 82
- param_forward (reduction-chain-rule), 97
- param_id_map (reduction-id-map), 98
- Parameter, 15, 81, 82, 83, 119
- parameters, 82
- partial_optimize, 83
- partial_trace, 84
- partial_transpose, 85
- perspective, 85
- pf_eigenvalue, 86
- PIQP_SOLVER (solver-constants), 107
- pos, 86
- PowCone3D, 87, 88
- PowConeND, 87, 87
- power, 67, 68, 88
- power(), 126
- Problem, 15, 34, 45, 49, 50, 53, 70, 71, 79, 82–84, 89, 90–92, 94, 95, 105–107, 110, 112, 117, 121, 124
- Problem(), 15, 79
- problem_data, 34, 90, 92, 109, 111, 117
- problem_solution, 90
- problem_status, 91, 112
- problem_unpack_results, 92, 111
- problem_unpack_results(), 117, 118
- prod_entries, 93
- PSD, 93
- PSD(), 127, 128
- psolve, 89, 94, 101, 108, 109, 113
- psolve(), 9, 22, 23, 35, 83, 84, 124
- ptp, 95

- quad_form, 96
- quad_over_lin, 97

- reduction-chain-rule, 97
- reduction-id-map, 98
- rel_entr, 99
- reshape_expr, 99
- residual, 100
- resolvent, 100

- sample_bounds, 101
- sample_bounds<- (sample_bounds), 101
- scalar_product, 102
- scalar_product(), 122
- scalarize, 101
- scalene, 103
- SCIP_SOLVER (solver-constants), 107
- SCS_SOLVER (solver-constants), 107
- sd, 66
- sd (math_atoms), 66

set_excluded_solvers
 (available_solvers), 8
 set_excluded_solvers(), 9
 set_label, 103
 set_label(), 32, 59, 60
 sigma_max, 104
 size, 104
 size(), 50, 56, 57
 size_metrics, 105, 106
 SizeMetrics, 105
 SOC, 106
 solution, 91, 107
 solve_via_data, 92, 110, 117
 solver-constants, 107
 solver_default_param, 95, 108
 SOLVER_ERROR (status-constants), 112
 solver_opts, 95, 109
 solver_stats, 94, 95, 110
 square, 111
 status, 91, 95, 112, 113
 status-constants, 112
 std (cvxr_std), 21
 sum_entries, 68, 113
 sum_largest, 114
 sum_smallest, 114
 sum_squares, 115

 to_latex, 116
 total_variation, 115
 total_variation(), 117
 tr_inv, 116
 tv, 117

 UNBOUNDED, 112
 UNBOUNDED (status-constants), 112
 UNBOUNDED_INACCURATE
 (status-constants), 112
 UNO_SOLVER (solver-constants), 107
 unpack_results, 117
 upper_tri, 118, 122
 USER_LIMIT (status-constants), 112

 value, 92, 94, 118
 value(), 83
 value<-, 119
 var, 66
 var (math_atoms), 66
 var_backward (reduction-chain-rule), 97
 var_dict, 121
 var_forward (reduction-chain-rule), 97
 var_id_map (reduction-id-map), 98
 Variable, 7, 33, 38, 78, 80, 83, 89, 101, 119,
 119, 120, 121, 125
 variables, 120
 vdot, 121
 vdot(), 20, 22, 102
 vec, 122
 vec(), 22
 vec_to_upper_tri, 122
 violation, 123
 visualize, 123
 vstack, 124

 xexp, 125
 Xor, 38, 125
 Xor(), 7, 38, 39, 78, 81
 XPRESS_SOLVER (solver-constants), 107

 Zero, 28, 126