

# Package ‘CVXR’

April 4, 2026

**Type** Package

**Title** Disciplined Convex Optimization

**Version** 1.8.2

**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, Narsimhan, 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 (>= 1.12), 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), Rcomplex (>= 0.3), cccp (>= 0.3), piqp (>= 0.6), scip, xpress

**Additional\_repositories** <https://bnaras.r-universe.dev>

**LinkingTo** Rcpp, RcppEigen

**License** Apache License 2.0 | file LICENSE

**LazyData** true

**Encoding** UTF-8

**RoxygenNote** 7.3.3

**VignetteBuilder** knitr

**Config/testthat/edition** 3

**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** 2026-04-04 05:12:04 UTC

## Contents

And	6
as_cvxr_expr	7
available_solvers	8
bmat	9
cdiac	9
ceil_expr	10
condition_number	11
Constant	11
constants	12
constraints	12
conv	13
cummax_expr	13
cumsum_axis	14
curvature	14
cvar	15
cvxr_diff	15
cvxr_mean	16
cvxr_norm	16
cvxr_outer	17
cvxr_std	17
cvxr_var	18
DiagMat	18
DiagVec	19
diff_pos	19
dist_ratio	20
dotsort	20
dspop	21
dssamp	21
dual_value	22
entr	22
Equality	23
ExpCone	23
expr_H	24
expr_sign	24
eye_minus_inv	25
FiniteSet	25

floor_expr . . . . .	26
gen_lambda_max . . . . .	26
geo_mean . . . . .	27
get_problem_data . . . . .	27
gmatmul . . . . .	28
harmonic_mean . . . . .	29
hstack . . . . .	29
huber . . . . .	30
id . . . . .	30
iff . . . . .	31
implies . . . . .	31
indicator . . . . .	32
Inequality . . . . .	33
installed_solvers . . . . .	33
inv_pos . . . . .	34
inv_prod . . . . .	34
is_affine . . . . .	35
is_concave . . . . .	35
is_constant . . . . .	36
is_convex . . . . .	36
is_dcp . . . . .	37
is_dgp . . . . .	37
is_dpp . . . . .	38
is_dqcp . . . . .	38
is_log_log_affine . . . . .	39
is_log_log_concave . . . . .	39
is_log_log_convex . . . . .	40
is_lp . . . . .	40
is_matrix . . . . .	41
is_mixed_integer . . . . .	41
is_nonneg . . . . .	42
is_nonpos . . . . .	42
is_nsd . . . . .	43
is_psd . . . . .	43
is_pwl . . . . .	44
is_qp . . . . .	44
is_quadratic . . . . .	45
is_quasiconcave . . . . .	45
is_quasiconvex . . . . .	46
is_quasilinear . . . . .	46
is_scalar . . . . .	47
is_symmetric . . . . .	47
is_vector . . . . .	48
is_zero . . . . .	48
kl_div . . . . .	49
kron . . . . .	49
lambda_max . . . . .	50
lambda_min . . . . .	50

lambda_sum_largest . . . . .	51
lambda_sum_smallest . . . . .	51
length_expr . . . . .	52
log1p_atom . . . . .	52
loggamma . . . . .	53
logistic . . . . .	53
log_det . . . . .	54
log_normcdf . . . . .	54
log_sum_exp . . . . .	55
make_sparse_diagonal_matrix . . . . .	55
math_atoms . . . . .	56
matrix_frac . . . . .	58
matrix_trace . . . . .	59
Maximize . . . . .	59
max_elemwise . . . . .	60
max_entries . . . . .	60
Minimize . . . . .	61
min_elemwise . . . . .	61
min_entries . . . . .	62
mixed_norm . . . . .	62
multiply . . . . .	63
name . . . . .	63
neg . . . . .	64
NonNeg . . . . .	64
NonPos . . . . .	65
norm1 . . . . .	65
norm2 . . . . .	66
norm_inf . . . . .	66
norm_nuc . . . . .	67
Not . . . . .	67
objective . . . . .	68
one_minus_pos . . . . .	69
Or . . . . .	69
Parameter . . . . .	70
parameters . . . . .	71
partial_trace . . . . .	71
partial_transpose . . . . .	72
perspective . . . . .	72
pf_eigenvalue . . . . .	73
pos . . . . .	74
PowCone3D . . . . .	74
PowConeND . . . . .	75
power . . . . .	76
Problem . . . . .	76
problem_data . . . . .	77
problem_solution . . . . .	78
problem_status . . . . .	78
problem_unpack_results . . . . .	79

prod_entries . . . . .	80
PSD . . . . .	80
psolve . . . . .	81
ptp . . . . .	82
p_norm . . . . .	82
quad_form . . . . .	83
quad_over_lin . . . . .	84
rel_entr . . . . .	84
reshape_expr . . . . .	85
residual . . . . .	85
resolvent . . . . .	86
scalar_product . . . . .	86
scalene . . . . .	87
set_label . . . . .	87
sigma_max . . . . .	88
size . . . . .	88
SOC . . . . .	89
solution . . . . .	90
solver-constants . . . . .	90
solver_default_param . . . . .	91
solver_opts . . . . .	92
solver_stats . . . . .	93
solve_via_data . . . . .	93
square . . . . .	94
status . . . . .	95
status-constants . . . . .	95
sum_entries . . . . .	96
sum_largest . . . . .	97
sum_smallest . . . . .	97
sum_squares . . . . .	98
total_variation . . . . .	98
to_latex . . . . .	99
tr_inv . . . . .	99
tv . . . . .	100
unpack_results . . . . .	100
upper_tri . . . . .	101
value . . . . .	101
value<- . . . . .	102
Variable . . . . .	102
variables . . . . .	103
vdot . . . . .	103
vec . . . . .	104
vec_to_upper_tri . . . . .	104
violation . . . . .	105
visualize . . . . .	105
vstack . . . . .	106
xexp . . . . .	107
Xor . . . . .	107

Zero . . . . .	108
%>>% . . . . .	109
%<<% . . . . .	109

<b>Index</b>	<b>111</b>
--------------	------------

---

And	<i>Logical AND</i>
-----	--------------------

---

### 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 <a href="#">Variables</a> 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)
```

**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\(\)](#)

---

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.

---

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**

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

**Value**

List of [Constant](#) objects.

---

`constraints`*Get Problem Constraints (read-only)*

---

**Description**

Returns a copy of the problem's constraint list.

**Usage**

```
constraints(x)
```

**Arguments**

<code>x</code>	A <a href="#">Problem</a> 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

---

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	<i>Get Expression Curvature</i>
-----------	---------------------------------

---

**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 *Compute kth Order Differences of an Expression*

---

**Description**

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

**Usage**

cvxr\_diff(x, k = 1L, axis = 2L)

**Arguments**

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

**Value**

An Expression representing the kth order differences.

---

cvxr_mean	<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  $x \otimes t(y)$ . Both inputs must be vectors.

**Usage**

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

**Arguments**

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

**Value**

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

---

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.

---

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 <math>x - y</math> with domain <math>x &gt; y &gt; 0</math></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, . . . , 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	<i>Create an entropy atom <math>-x * \log(x)</math></i>
------	---

---

**Description**

Create an entropy atom  $-x * \log(x)$

**Usage**

entr(x)

**Arguments**

x                    An Expression

**Value**

An Entr atom

---

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

---

**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**

<code>lhs</code>	A CVXR expression (left-hand side).
<code>rhs</code>	A CVXR expression (right-hand side).
<code>constr_id</code>	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**

<code>x_expr</code>	A CVXR expression.
<code>y_expr</code>	A CVXR expression.
<code>z_expr</code>	A CVXR expression.
<code>constr_id</code>	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	<i>FiniteSet Constraint</i>
-----------	-----------------------------

---

**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	<i>Elementwise Floor</i>
------------	--------------------------

---

**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](#)

---

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

---

**Description**

**[Deprecated]**

**Usage**

```
get_problem_data(x, solver = NULL, ...)
```

**Arguments**

x	A <a href="#">Problem</a> object.
solver	Character string naming solver, or NULL for automatic selection.
...	Additional arguments.

**Details**

Use [problem\\_data](#) instead.

**Value**

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

**See Also**

[problem\\_data](#)

---

gmatmul

*Geometric matrix multiplication A diamond X*

---

**Description**

Computes the geometric matrix product where  $(A \text{ diamond } X)_{ij} = \prod_k X_{kj}^{A_{ik}}$ . 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)
```

---

harmonic_mean	<i>Harmonic mean: <math>n / \sum(1/x_i)</math></i>
---------------	--

---

**Description**

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

**Usage**

harmonic\_mean(x)

**Arguments**

x                    An Expression (must be positive for DCP)

**Value**

An Expression representing the harmonic mean

---

hstack	<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 *Logical Biconditional*

---

**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 *Logical Implication*

---

**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  $!x \mid y$ .

**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

*Indicator function for constraints*

---

**Description**

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

**Usage**

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

**Arguments**

**constraints**      A list of constraint objects  
**err\_tol**            Numeric tolerance for checking constraint satisfaction (default 1e-3)

**Value**

An Indicator expression

---

Inequality	<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: <math>x^{-1}</math> (for <math>x &gt; 0</math>)</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_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_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, ...)
```

**Arguments**

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

**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\_log\_log\_affine      *Check if Expression is Log-Log Affine*

---

**Description**

Check if Expression is Log-Log Affine

**Usage**

is\_log\_log\_affine(x, ...)

**Arguments**

x	An expression object.
...	Not used.

**Value**

Logical scalar.

---

is\_log\_log\_concave      *Check if Expression is Log-Log Concave*

---

**Description**

Check if Expression is Log-Log Concave

**Usage**

is\_log\_log\_concave(x, ...)

**Arguments**

x	An expression object.
...	Not used.

**Value**

Logical scalar.

---

is\_log\_log\_convex      *Check if Expression is Log-Log Convex*

---

**Description**

Check if Expression is Log-Log Convex

**Usage**

is\_log\_log\_convex(x, ...)

**Arguments**

x	An expression object.
...	Not used.

**Value**

Logical scalar.

---

is\_lp      *Check if a Problem is a Linear Program*

---

**Description**

Check if a Problem is a Linear Program

**Usage**

is\_lp(x, ...)

**Arguments**

x	A <a href="#">Problem</a> 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`*Check if Expression is Piecewise Linear*

---

**Description**

Check if Expression is Piecewise Linear

Is the Expression Piecewise Linear?

**Usage**

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

**Arguments**

`x`                    A CVXR expression.

`...`                 Not used.

**Value**

Logical scalar.

Logical.

---

`is_qp`*Check if a Problem is a Quadratic Program*

---

**Description**

Check if a Problem is a Quadratic Program

**Usage**

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

**Arguments**

`x`                    A [Problem](#) object.

`...`                 Not used.

**Value**

Logical scalar.

---

is_quadratic	<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_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: <math>x \cdot \log(x/y) - x + y</math></i>
--------	--

---

**Description**

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

**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

---

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     *Sum of largest k eigenvalues*

---

**Description**

Sum of largest k eigenvalues

**Usage**

lambda\_sum\_largest(A, k)

**Arguments**

A                    A square matrix expression  
k                    Number of largest eigenvalues to sum (positive integer)

**Value**

An expression representing the sum of the k largest eigenvalues

---

lambda\_sum\_smallest     *Sum of smallest k eigenvalues*

---

**Description**

Sum of smallest k eigenvalues

**Usage**

lambda\_sum\_smallest(A, k)

**Arguments**

A                    A square matrix expression  
k                    Number of smallest eigenvalues to sum (positive integer)

**Value**

An expression representing the sum of the k smallest eigenvalues

---

length_expr	<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**

x                    A CVXR vector expression.

**Value**

A Length expression (scalar).

**See Also**

[ceil\\_expr](#), [floor\\_expr](#)

---

log1p_atom	<i>Log(1 + x) – elementwise</i>
------------	---------------------------------

---

**Description**

Log(1 + x) – 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: <math>\log(1 + \exp(x))</math> – elementwise</i>
----------	--

---

**Description**

Logistic function:  $\log(1 + \exp(x))$  – 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: <math>\log(\text{sum}(\exp(x)))</math></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	<i>Make a CSC sparse diagonal matrix</i>
-----------------------------	--

---

**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

## 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 <code>outer</code> .
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. $k = 0$ (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  $x \geq 0$ )

`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

*Matrix fractional function*

---

**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  $\text{trace}(A \%* \% B) = \text{sum}(A * t(B))$  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 (<math>L_{p,q}</math> 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: <math>-\min(x, 0)</math></i>
-----	--

---

**Description**

Negative part:  $-\min(x, 0)$

**Usage**

neg(x)

**Arguments**

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

**Value**

A negated Minimum atom

---

NonNeg	<i>Create a Non-Negative Constraint</i>
--------	---

---

**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	<i>Create a Non-Positive Constraint</i>
--------	---

---

**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\(\)](#)


---

norm_inf	<i>L-infinity norm of an expression</i>
----------	---

---

**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 <a href="#">Variable</a> 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

*Get Problem Objective (read-only)*

---

## Description

Returns the problem's objective.

## Usage

```
objective(x)
```

## Arguments

x                    A [Problem](#) object.

## Details

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

## Value

A [Minimize](#) or [Maximize](#) object.

## See Also

[Problem\(\)](#), [constraints\(\)](#)

## Examples

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

---

one_minus_pos	<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

---

*Create a Parameter*


---

**Description**

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

**Usage**

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

**Arguments**

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

**Value**

A Parameter object (inherits from Leaf and Expression).

**Examples**

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

---

 parameters

*Get the Parameters in an Expression*


---

**Description**

Get the Parameters in an Expression

**Usage**

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

**Arguments**

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

**Value**

List of [Parameter](#) objects.

---

 partial\_trace

*Partial trace of a tensor product expression*


---

**Description**

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

**Usage**

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

**Arguments**

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

**Value**

An Expression representing the partial trace

---

partial\_trace      *Partial transpose of a tensor product expression*

---

**Description**

Assumes expr is a 2D square matrix representing a Kronecker product of  $\text{length}(\text{dims})$  subsystems. Returns the partial transpose with the transpose applied to 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 transpose

**Value**

An Expression representing the partial transpose

---

perspective      *Perspective Transform*

---

**Description**

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

**Usage**

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

**Arguments**

- f                    A scalar convex or concave Expression.
- s                    A nonneg Variable (scalar).
- f\_recession        Optional recession function for handling  $s = 0$ .

**Value**

A Perspective expression.

---

pf_eigenvalue	<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: <math>\max(x, 0)</math></i>
-----	---

---

**Description**

Positive part:  $\max(x, 0)$

**Usage**

pos(x)

**Arguments**

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

**Value**

A Maximum atom

---

PowCone3D	<i>Create a 3D Power Cone Constraint</i>
-----------	--

---

**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

<code>W</code>	A CVXR expression (vector or matrix).
<code>z</code>	A CVXR expression (scalar or vector).
<code>alpha</code>	A CVXR expression with positive entries summing to 1 along the specified axis.
<code>axis</code>	Integer, 2 (default, column-wise) or 1 (row-wise).
<code>constr_id</code>	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
p	Numeric exponent
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

**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 <a href="#">Minimize</a> or <a href="#">Maximize</a> object.
constraints	A list of Constraint objects (e.g., created by <code>==</code> , <code>&lt;=</code> , <code>&gt;=</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, ...)
```

**Arguments**

x	A <a href="#">Problem</a> object.
solver	Character string naming solver, or NULL for automatic selection.
...	Additional arguments.

**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**

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

**Details**

After calling this function, variable values are available via [value\(\)](#) and constraint duals via [dual\\_value\(\)](#).

**Value**

The problem object (invisibly), with solution unpacked.

**See Also**

[problem\\_data](#), [solve\\_via\\_data](#)

---

prod\_entries                      *Product of entries along an axis*

---

### Description

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

### Usage

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

### Arguments

<code>x</code>	An Expression
<code>axis</code>	NULL (all), 1 (row-wise), or 2 (column-wise)
<code>keepdims</code>	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                                      *Create a Positive Semidefinite Constraint*

---

### Description

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

### Usage

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

### Arguments

<code>expr</code>	A CVXR expression representing a square matrix.
<code>constr_id</code>	Optional integer constraint ID.

### Value

A PSD constraint object.

**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,
    ...
)
```

**Arguments**

<code>problem</code>	A <a href="#">Problem</a> object.
<code>solver</code>	Character string naming the solver to use (e.g., "CLARABEL", "SCS", "OSQP", "HIGHS"), or NULL for automatic selection.
<code>gp</code>	Logical; if TRUE, solve as a geometric program (DGP).
<code>qcp</code>	Logical; if TRUE, solve as a quasiconvex program (DQCP) via bisection. Only needed for non-DCP DQCP problems.
<code>verbose</code>	Logical; if TRUE, print solver output.
<code>warm_start</code>	Logical; if TRUE, use the current variable values as a warm-start point for the solver.
<code>...</code>	Solver options passed to <a href="#">solver_opts()</a> . Includes chain-construction options ( <code>use_quad_obj</code> ), standard tolerances ( <code>feastol</code> , <code>reltol</code> , <code>abstol</code> , <code>num_iter</code> ), and solver-specific parameters (e.g., <code>eps_abs</code> , <code>scip_params</code> ). See <a href="#">solver_opts</a> for details. For DQCP problems ( <code>qcp = TRUE</code> ), additional arguments include <code>low</code> , <code>high</code> , <code>eps</code> , <code>max_iters</code> , and <code>max_iters_interval_search</code> .

**Value**

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

**See Also**

[Problem](#), [status](#), [solver\\_stats](#), [solver\\_default\\_param](#)

**Examples**

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

---

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

---

**Description**

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

**Usage**

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

**Arguments**

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

**Value**

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

---

p\_norm *General p-norm of an expression*

---

**Description**

General p-norm of an expression

**Usage**

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

**Arguments**

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

**Value**

A Pnorm, PnormApprox, Norm1, or NormInf atom

---

quad_form	<i>Quadratic form <math>x^T P x</math></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

---

rel_entr	<i>Relative Entropy: <math>x \cdot \log(x/y)</math></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

---

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

---

**Description**

Scalar product (alias for vdot)

**Usage**

scalar\_product(x, y)

**Arguments**

x	An Expression or numeric value.
y	An Expression or numeric value.

**Value**

A scalar Expression representing  $\text{sum}(x * y)$ .

---

scalene	<i>Scalene penalty: <math>\alpha * \text{pos}(x) + \beta * \text{neg}(x)</math></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>Set a Label on a Constraint</i>
-----------	------------------------------------

---

**Description**

Attaches a human-readable label to a constraint for use in visualizations and pretty-printing. Labels are visualization-only and never affect the solver pipeline.

**Usage**

```
set_label(constraint, label)
```

**Arguments**

constraint	A Constraint object.
label	A character string label.

**Details**

Because R uses copy-on-modify semantics, you must either assign the result back or use `set_label` fluently when building constraint lists.

**Value**

The modified constraint (invisibly).

**Examples**

```
## Not run:
x <- Variable(3, name = "x")

# Assign back
con <- (x >= 0)
con <- set_label(con, "non-negativity")

# Fluent use in constraint lists
constraints <- list(
  set_label(x >= 1, "lower bound"),
  set_label(sum_entries(x) <= 10, "budget")
)

## End(Not run)
```

---

sigma_max	<i>Maximum singular value</i>
-----------	-------------------------------

---

**Description**

Maximum singular value

**Usage**

```
sigma_max(A)
```

**Arguments**

A                    A matrix expression

**Value**

An expression representing the maximum singular value of A

---

size	<i>Get Expression Size</i>
------	----------------------------

---

**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\(\)](#)

---

 SOC

---

*Create a Second-Order Cone Constraint*


---

**Description**

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

**Usage**

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

**Arguments**

`t` A CVXR expression (scalar or vector) representing the upper bound.

`X` A CVXR expression (vector or matrix) whose columns/rows are bounded.

`axis` Integer, 2 (default, column-wise) or 1 (row-wise). Determines whether columns (2) or rows (1) of  $X$  define the individual cones.

`constr_id` Optional integer constraint ID.

**Value**

An SOC constraint object.

---

solution	<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
```

```
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

### Format

An object of class character of length 1.

An object of class character of length 1.

An object of class character of length 1.

An object of class character of length 1.

An object of class character of length 1.

An object of class character of length 1.

An object of class character of length 1.

An object of class character of length 1.

An object of class character of length 1.

An object of class character of length 1.

An object of class character of length 1.

An object of class character of length 1.

An object of class character of length 1.

An object of class character of length 1.

An object of class character of length 1.

### 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 and their default values. Used internally by `psolve` to translate standard parameters into solver-native names.

**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 (solver-native parameter name) and value (default value).

**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 <a href="#">problem_data()</a> .
warm_start	Logical; use warm-start if supported.
verbose	Logical; print solver output.
solver_opts	Named list of solver-specific options.
...	Additional arguments (e.g., problem for SolvingChain dispatch, solver_cache for Solver dispatch).

**Value**

Solver-specific result (a named list).

**See Also**

[problem\\_data](#), [problem\\_unpack\\_results](#)

---

square

*Square of an expression:  $x^2$*

---

**Description**

Square of an expression:  $x^2$

**Usage**

square(x)

**Arguments**

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

**Value**

A Power atom with p=2

---

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

### Format

An object of class character of length 1.

An object of class character of length 1.

An object of class character of length 1.

An object of class character of length 1.

An object of class character of length 1.

An object of class character of length 1.

An object of class character of length 1.

An object of class character of length 1.

An object of class character of length 1.

### 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)

**Arguments**

x	An Expression
k	Number of largest entries to sum

**Value**

A SumLargest atom

---

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

---

**Description**

Sum of k smallest entries

**Usage**

sum\_smallest(x, k)

**Arguments**

x	An Expression
k	Number of smallest entries to sum

**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 <a href="#">Problem</a> object.
solution	The raw solver result from <a href="#">solve_via_data()</a> .
chain	The SolvingChain from <a href="#">problem_data()</a> .
inverse_data	The inverse data list from <a href="#">problem_data()</a> .

**Details**Use [problem\\_unpack\\_results\(\)](#) instead. This alias exists for backward compatibility with older CVXR examples.

**Value**

The problem object (invisibly), with solution unpacked.

**See Also**

[problem\\_unpack\\_results\(\)](#)

---

upper_tri	<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.

---

<code>value&lt;-</code>	<i>Set the Value of a Leaf Expression</i>
-------------------------	---

---

**Description**

Assigns a numeric value to a [Variable](#) or [Parameter](#).

**Usage**

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

**Arguments**

<code>x</code>	A leaf expression object.
<code>value</code>	The value to assign.

**Value**

The modified object (invisibly).

---

<code>Variable</code>	<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, var_id = NULL, latex_name = NULL, ...)
```

**Arguments**

<code>shape</code>	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).
<code>name</code>	Optional character string name for the variable. If NULL, an automatic name "var<id>" is generated.
<code>var_id</code>	Optional integer ID. If NULL, a unique ID is generated.
<code>latex_name</code>	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.
<code>...</code>	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.

---

vdot

*Vector dot product (inner product)*


---

**Description**

Computes the inner product: sum of element-wise products after flattening. Returns a scalar expression.

**Usage**

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

**Arguments**

x	An Expression or numeric value.
y	An Expression or numeric value.

**Value**

A scalar Expression representing  $\text{sum}(x * y)$ .

---

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 <a href="#">Problem</a> 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 <code>psolve()</code> ). 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  $x * \exp(x)$  – *elementwise*

**Description**

$x * \exp(x)$  – *elementwise*

**Usage**

xexp(x)

**Arguments**

x An Expression

**Value**

An Xexp atom

Xor *Logical XOR*

**Description**

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

**Usage**

Xor(..., id = NULL)

**Arguments**

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

**Details**

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

**Value**

A `Xor` expression.

**See Also**

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

**Examples**

```
## 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  $A \succ B$ .

**Usage**

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

**Arguments**

`e1, e2` CVXR expressions or numeric matrices.

**Value**

A PSD constraint object.

**See Also**

[PSD\(\)](#)

**Examples**

```
## Not run:  
X <- Variable(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  $A \ll B$ .

**Usage**

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

**Arguments**

`e1, e2` CVXR expressions or numeric matrices.

**Value**

A PSD constraint object.

**See Also**

[PSD\(\)](#)

**Examples**

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

# Index

- \* **datasets**
  - solver-constants, 90
  - status-constants, 95
- \* **data**
  - cdiac, 9
  - dspop, 21
  - dssamp, 21
- %<<%, 109
- %>>%, 109
  
- And, 6
- And(), 31, 32, 67, 70, 108
- as.matrix(), 7
- as\_cvxr\_expr, 7
- available\_solvers, 8
  
- bmat, 9
  
- cdiac, 9
- ceil\_expr, 10, 26, 52
- CLARABEL\_SOLVER (solver-constants), 90
- condition\_number, 11
- Constant, 7, 11, 12
- constants, 12
- constraints, 12
- constraints(), 68
- conv, 13
- CPLEX\_SOLVER (solver-constants), 90
- cummax\_expr, 13
- cumsum\_axis, 14
- curvature, 14
- cvar, 15
- CVXOPT\_SOLVER (solver-constants), 90
- cvxr\_diff, 15, 57, 58
- cvxr\_mean, 16, 58
- cvxr\_norm, 16, 58
- cvxr\_outer, 17, 58
- cvxr\_std, 17, 58
- cvxr\_var, 18, 58
  
- diag (math\_atoms), 56
  
- DiagMat, 18, 19, 56–58
- DiagVec, 18, 19, 56–58
- diff\_pos, 19
- dist\_ratio, 20
- dotsort, 20
- dspop, 21, 21
- dssamp, 21, 21
- dual\_value, 22, 79, 81
  
- ECOS\_BB\_SOLVER (solver-constants), 90
- ECOS\_SOLVER (solver-constants), 90
- entr, 22
- Equality, 23, 33, 108
- exclude\_solvers (available\_solvers), 8
- exclude\_solvers(), 8
- ExpCone, 23
- expr\_H, 24
- expr\_sign, 24
- eye\_minus\_inv, 25
  
- FiniteSet, 25
- floor\_expr, 10, 26, 52
  
- gen\_lambda\_max, 26
- geo\_mean, 27
- get\_problem\_data, 27
- GLPK\_MI\_SOLVER (solver-constants), 90
- GLPK\_SOLVER (solver-constants), 90
- gmatmul, 28
- GUROBI\_SOLVER (solver-constants), 90
  
- harmonic\_mean, 29
- HIGHS\_SOLVER (solver-constants), 90
- hstack, 29
- huber, 30
  
- id, 30
- iff, 31
- iff(), 6, 32, 67, 70, 108
- implies, 31
- implies(), 6, 31, 67, 70, 108

- include\_solvers (available\_solvers), 8
- include\_solvers(), 8
- indicator, 32
- Inequality, 23, 33, 64, 65
- INFEASIBLE, 95
- INFEASIBLE (status-constants), 95
- INFEASIBLE\_INACCURATE (status-constants), 95
- INFEASIBLE\_OR\_UNBOUNDED (status-constants), 95
- installed\_solvers, 33
- installed\_solvers(), 8
- inv\_pos, 34
- inv\_prod, 34
- is\_affine, 35
- is\_affine(), 14
- is\_concave, 35
- is\_concave(), 14
- is\_constant, 36
- is\_constant(), 14
- is\_convex, 36
- is\_convex(), 14
- is\_dcp, 37
- is\_dgp, 37
- is\_dpp, 38
- is\_dqcp, 38
- is\_log\_log\_affine, 39
- is\_log\_log\_concave, 39
- is\_log\_log\_convex, 40
- is\_lp, 40
- is\_matrix, 41
- is\_matrix(), 47, 48, 89
- is\_mixed\_integer, 41
- is\_nonneg, 42
- is\_nonpos, 42
- is\_nsd, 43
- is\_psd, 43
- is\_pwl, 44
- is\_qp, 44
- is\_quadratic, 45
- is\_quasiconcave, 45
- is\_quasiconvex, 46
- is\_quasilinear, 46
- is\_scalar, 47
- is\_scalar(), 41, 48, 89
- is\_symmetric, 47
- is\_vector, 48
- is\_vector(), 41, 47, 89
- is\_zero, 48
- kl\_div, 49
- kron, 49
- lambda\_max, 50
- lambda\_min, 50
- lambda\_sum\_largest, 51
- lambda\_sum\_smallest, 51
- length\_expr, 52
- log1p\_atom, 52
- log1p\_expr (log1p\_atom), 52
- log\_det, 54
- log\_normcdf, 54
- log\_sum\_exp, 55
- loggamma, 53
- logistic, 53
- make\_sparse\_diagonal\_matrix, 55
- math\_atoms, 56, 76
- Matrix::Matrix, 7
- Matrix::sparseVector, 7
- matrix\_frac, 58
- matrix\_trace, 59
- max\_elemwise, 60
- max\_entries, 58, 60
- Maximize, 59, 61, 68, 76
- min\_elemwise, 61
- min\_entries, 58, 62
- Minimize, 59, 61, 68, 76
- mixed\_norm, 62
- MOSEK\_SOLVER (solver-constants), 90
- multiply, 63
- name, 63
- neg, 64
- NonNeg, 33, 64, 65, 108
- NonPos, 33, 64, 65, 108
- norm (math\_atoms), 56
- norm1, 65
- norm2, 66
- norm\_inf, 66
- norm\_nuc, 67
- Not, 31, 67
- Not(), 6, 31, 32, 70, 108
- objective, 68
- objective(), 13
- one\_minus\_pos, 69

- OPTIMAL, [95](#)
- OPTIMAL (status-constants), [95](#)
- OPTIMAL\_INACCURATE (status-constants), [95](#)
- Or, [32](#), [69](#)
- Or(), [6](#), [31](#), [32](#), [67](#), [108](#)
- OSQP\_SOLVER (solver-constants), [90](#)
- outer, [56](#)
- outer (math\_atoms), [56](#)
- p\_norm, [82](#)
- p\_norm(), [66](#)
- Parameter, [12](#), [70](#), [71](#), [102](#)
- parameters, [71](#)
- partial\_trace, [71](#)
- partial\_transpose, [72](#)
- perspective, [72](#)
- pf\_eigenvalue, [73](#)
- PIQP\_SOLVER (solver-constants), [90](#)
- pos, [74](#)
- PowCone3D, [74](#), [75](#)
- PowConeND, [74](#), [75](#)
- power, [57](#), [58](#), [76](#)
- power(), [108](#)
- Problem, [12](#), [27](#), [40](#), [41](#), [44](#), [59](#), [61](#), [68](#), [76](#), [77–79](#), [81](#), [90](#), [93](#), [95](#), [100](#), [106](#)
- Problem(), [12](#), [13](#), [68](#)
- problem\_data, [27](#), [28](#), [77](#), [79](#), [92](#), [94](#), [100](#)
- problem\_solution, [78](#)
- problem\_status, [78](#), [95](#)
- problem\_unpack\_results, [79](#), [94](#)
- problem\_unpack\_results(), [100](#), [101](#)
- prod\_entries, [80](#)
- PSD, [80](#)
- PSD(), [109](#), [110](#)
- psolve, [76](#), [81](#), [91](#), [92](#), [96](#)
- psolve(), [106](#)
- ptp, [82](#)
- quad\_form, [83](#)
- quad\_over\_lin, [84](#)
- rel\_entr, [84](#)
- reshape\_expr, [85](#)
- residual, [85](#)
- resolvent, [86](#)
- scalar\_product, [86](#)
- scalene, [87](#)
- SCIP\_SOLVER (solver-constants), [90](#)
- SCS\_SOLVER (solver-constants), [90](#)
- sd, [56](#)
- sd (math\_atoms), [56](#)
- set\_excluded\_solvers (available\_solvers), [8](#)
- set\_excluded\_solvers(), [8](#)
- set\_label, [87](#)
- sigma\_max, [88](#)
- size, [88](#)
- size(), [41](#), [47](#), [48](#)
- SOC, [89](#)
- solution, [78](#), [90](#)
- solve\_via\_data, [79](#), [93](#), [100](#)
- solver-constants, [90](#)
- solver\_default\_param, [81](#), [91](#)
- SOLVER\_ERROR (status-constants), [95](#)
- solver\_opts, [81](#), [92](#)
- solver\_stats, [81](#), [93](#)
- square, [94](#)
- status, [78](#), [79](#), [81](#), [95](#), [96](#)
- status-constants, [95](#)
- std (cvxr\_std), [17](#)
- sum\_entries, [58](#), [96](#)
- sum\_largest, [97](#)
- sum\_smallest, [97](#)
- sum\_squares, [98](#)
- to\_latex, [99](#)
- total\_variation, [98](#)
- total\_variation(), [100](#)
- tr\_inv, [99](#)
- tv, [100](#)
- UNBOUNDED, [95](#)
- UNBOUNDED (status-constants), [95](#)
- UNBOUNDED\_INACCURATE (status-constants), [95](#)
- unpack\_results, [100](#)
- upper\_tri, [101](#), [104](#)
- USER\_LIMIT (status-constants), [95](#)
- value, [79](#), [81](#), [101](#)
- value<-, [102](#)
- var, [56](#)
- var (math\_atoms), [56](#)
- Variable, [6](#), [31](#), [67](#), [69](#), [77](#), [102](#), [102](#), [103](#), [107](#)
- variables, [103](#)
- vdot, [103](#)

vec, [104](#)  
vec\_to\_upper\_tri, [104](#)  
violation, [105](#)  
visualize, [105](#)  
vstack, [106](#)

xexp, [107](#)  
Xor, [31](#), [107](#)  
Xor(), [6](#), [31](#), [32](#), [67](#), [70](#)  
XPRESS\_SOLVER (solver-constants), [90](#)

Zero, [23](#), [108](#)