

Package ‘sparsediff’

October 6, 2026

Type Package

Title R Interface to the 'SparseDiffEngine' Sparse Differentiation Backend

Version 0.6.1

Description Bindings for the 'SparseDiffEngine' C library, the sparse Jacobian and Hessian differentiation backend used by 'CVXPY' for its Disciplined Nonlinear Programming (DNLP) extension. Provides low-level routines for building nonlinear expression graphs and evaluating sparse derivatives, intended as a backend for higher-level modeling layers such as 'CVXR'. This is the R analog of the 'sparsediffpy' Python package and wraps the same C library.

License Apache License (== 2.0)

Copyright file inst/COPYRIGHTS

URL <https://bnaras.github.io/sparsediff/>,
<https://github.com/bnaras/sparsediff>

BugReports <https://github.com/bnaras/sparsediff/issues>

Encoding UTF-8

SystemRequirements GNU make

LinkingTo cpp11

Suggests cpp11, knitr, rmarkdown, testthat (>= 3.0.0)

VignetteBuilder knitr

Config/testthat/edition 3

NeedsCompilation yes

Config/roxygen2/version 8.1.0

Author Balasubramanian Narasimhan [aut, cre],
Daniel Cederberg [aut, cph] (Author of the bundled SparseDiffEngine C library),
William Zijie Zhang [aut, cph] (Author of the bundled SparseDiffEngine C library)

Maintainer Balasubramanian Narasimhan <naras@stanford.edu>

Repository CRAN
Date/Publication 2026-10-06 04:10:02 UTC

Contents

engine_version	2
sparsediff-affine	3
sparsediff-bivariate	4
sparsediff-elementwise	5
sparsediff-getters	6
sparsediff-leaves	7
sparsediff-matrix	7
sparsediff-oracle	9
sparsediff-problem	11
sparsediff-reduction	12
Index	13

engine_version	<i>Bundled SparseDiffEngine version</i>
----------------	---

Description

Returns the version string of the SparseDiffEngine C library bundled with this package.

Usage

```
engine_version()
```

Value

A character scalar, e.g. "0.6.1".

Examples

```
engine_version()
```

sparsediff-affine	<i>Affine and shape atoms</i>
-------------------	-------------------------------

Description

Affine combinations and shape manipulations of expressions. These have constant (zero) second derivatives but participate in the Jacobian.

Usage

```
sd_add(left, right)
sd_sum(child, axis)
sd_trace(c)
sd_transpose(c)
sd_diag_vec(c)
sd_diag_mat(c)
sd_upper_tri(c)
sd_promote(c, d1, d2)
sd_reshape(c, d1, d2)
sd_broadcast(c, d1, d2)
sd_index(child, d1, d2, indices)
sd_hstack(args, n_vars)
sd_vstack(args, n_vars)
```

Arguments

left, right, child, c	expression handles.
d1, d2	target row and column dimensions (for sd_promote, sd_reshape, sd_broadcast, sd_index).
indices	0-based column-major flat indices selected by sd_index.
args	a list of expression handles to stack (sd_hstack, sd_vstack).
n_vars	total number of variables in the problem.
axis	reduction axis for sd_sum: -1 (all entries), 0 (down rows) or 1 (across columns).

Details

sd_add elementwise sum of two expressions.
sd_sum sum reduction along axis.
sd_trace matrix trace.
sd_transpose matrix transpose.
sd_diag_vec diagonal matrix from a vector.
sd_diag_mat diagonal vector from a matrix.
sd_upper_tri the strict upper-triangular entries.

`sd_promote` promote a scalar to shape $d1 \times d2$.
`sd_reshape` reshape to $d1 \times d2$ (column-major).
`sd_broadcast` broadcast to $d1 \times d2$.
`sd_index` select entries by 0-based flat indices.
`sd_hstack`, `sd_vstack` horizontal / vertical stacking.

Value

An expression handle.

See Also

[sparsediff-elementwise](#), [sparsediff-matrix](#)

`sparsediff-bivariate` *Bivariate atoms*

Description

Functions of two expression arguments.

Usage

```

sd_elementwise_mult(l, r)
sd_matmul(x, y)
sd_quad_over_lin(l, r)
sd_rel_entr(l, r)
sd_rel_entr_first_scalar(l, r)
sd_rel_entr_second_scalar(l, r)

```

Arguments

`l`, `r`, `x`, `y` expression handles.

Details

`sd_elementwise_mult` elementwise (Hadamard) product.
`sd_matmul` matrix product xy .
`sd_quad_over_lin` the quadratic-over-linear $\|x\|^2/y$.
`sd_rel_entr` relative entropy $x \log(x/y)$. Like 'sparsediffpy' `make_rel_entr`, it dispatches on operand size: a scalar `l` with a non-scalar `r` uses `sd_rel_entr_first_scalar`, the reverse uses `sd_rel_entr_second_scalar`, and otherwise it is elementwise.
`sd_rel_entr_first_scalar`, `sd_rel_entr_second_scalar` relative entropy with a scalar first or second argument broadcast against the other.

Value

An expression handle.

See Also

[sparsediff-elementwise](#), [sparsediff-reduction](#)

sparsediff-elementwise

Elementwise atoms

Description

Smooth elementwise functions of a single expression. Each returns an expression of the same shape as its argument.

Usage

```
sd_exp(child)
sd_log(c)
sd_sin(c)
sd_cos(c)
sd_tan(c)
sd_sinh(c)
sd_tanh(c)
sd_asinh(c)
sd_atanh(c)
sd_logistic(c)
sd_xexp(c)
sd_normal_cdf(c)
sd_entr(c)
sd_power(c, p)
sd_neg(child)
```

Arguments

child, c	an expression handle (the argument).
p	exponent for sd_power.

Details

sd_exp, sd_log exponential and natural logarithm.

sd_sin, sd_cos, sd_tan trigonometric functions.

sd_sinh, sd_tanh, sd_asinh, sd_atanh hyperbolic and inverse-hyperbolic functions.

sd_logistic the logistic $\log(1 + e^x)$.

sd_xexp xe^x .

sd_normal_cdf the standard normal CDF.
 sd_entr the elementwise entropy $-x \log x$.
 sd_power the power x^p .
 sd_neg negation $-x$.

Value

An expression handle.

See Also

[sparsediff-affine](#), [sparsediff-bivariate](#)

sparsediff-getters	<i>Expression shape</i>
--------------------	-------------------------

Description

The dimensions and the number of entries of an expression node (the 'sparsediffpy' get_expr_dimensions and get_expr_size).

Usage

```
sd_get_expr_dimensions(node)
sd_get_expr_size(node)
```

Arguments

node an expression handle.

Value

sd_get_expr_dimensions: an integer vector $c(d1, d2)$, the node's rows and columns. sd_get_expr_size: an integer scalar, the number of entries $d1 * d2$.

See Also

[sparsediff-leaves](#)

sparsediff-leaves *Leaf expressions: variables and parameters*

Description

Create the leaves of an expression graph. A variable is a slice of the differentiation vector; a parameter is fixed data that can be updated between evaluations (see [sd_register_params](#)).

Usage

```
sd_variable(d1, d2, var_id, n_vars)
sd_parameter(d1, d2, param_id, n_vars, values)
```

Arguments

d1, d2	row and column dimensions of the leaf.
var_id	0-based flat (column-major) offset of this variable's first entry within the primal vector u.
param_id	0-based parameter offset, analogous to var_id.
n_vars	total number of variables in the problem.
values	numeric data for the parameter (length d1 * d2, column-major).

Value

An expression handle.

See Also

[sparsediff-elementwise](#), [sd_problem](#)

sparsediff-matrix *Parameter- and constant-matrix atoms*

Description

Operations that combine an expression with fixed data — a registered parameter node or a constant matrix — so the data can flow through the differentiated graph (and, for parameters, be updated between evaluations).

Usage

```
sd_scalar_mult(param, child)
sd_vector_mult(param, child)
sd_convolve(param, child)
sd_quad_form(child, Qp, Qi, Qx)
sd_quad_form_dense(param, child, data)
sd_left_matmul(child, Ap, Ai, Ax, ncol)
sd_right_matmul(child, Ap, Ai, Ax, ncol)
sd_left_matmul_dense(param, child, m, n, data)
sd_right_matmul_dense(param, child, m, n, data)
sd_left_kron(param, child, p, q, r, s, active_blocks)
sd_right_kron(param, child, p, q, r, s, active_blocks)
```

Arguments

param	a parameter expression handle (see sd_parameter).
child	an expression handle (the variable argument).
Qp, Qi, Qx	the row-pointer, column-index and value arrays of a compressed-sparse-row (CSR) matrix Q for sd_quad_form 's $x^\top Qx$. Q must be symmetric, so its CSR arrays equal its compressed-sparse-column arrays and the @p, @i, @x slots of a <code>Matrix::dgCMatrix</code> holding Q can be passed directly.
Ap, Ai, Ax	the row-pointer, column-index and value arrays of a constant matrix A in compressed-sparse-row (CSR) form, for the sparse matrix products. These are the @p, @i, @x slots of a <code>Matrix::dgCMatrix</code> holding $A^\top$, not A .
ncol	number of columns of the sparse constant matrix A .
m, n	row and column dimensions of the dense constant matrix.
p, q, r, s	for the Kronecker products $Z = A \otimes B$: A is $p \times q$ and B is $r \times s$.
active_blocks	for the Kronecker products, an integer vector of 0-based column-major indices of the nonzero entries of the variable-free operand (param); only the output rows they cover are built. For a parametric operand pass every index, $\emptyset$: (length - 1).
data	the dense constant-matrix entries in row-major order (length $m * n$ for the matrix products, $n * n$ for $\text{sd_quad_form_dense}$); for an R matrix <code>M</code> , pass as <code>.vector(t(M))</code> . Pass <code>numeric(0)</code> when the matrix comes from param.

Details

`sd_scalar_mult`, `sd_vector_mult` multiply a child by a scalar / vector parameter.

`sd_convolve` convolution of a parameter kernel with a child.

`sd_quad_form` the quadratic form $x^\top Qx$ with sparse constant Q .

`sd_quad_form_dense` the quadratic form $x^\top Qx$ with a dense symmetric $n \times n$ Q , where n is the length of the vector child. Supply exactly one source: `param = NULL` and `data` for a constant Q (checked for symmetry), or a parameter handle of size n^2 with `data = numeric(0)` for a parametric Q that follows [sd_update_params](#). A parametric Q is not checked: keeping it symmetric is the caller's responsibility.

`sd_left_matmul, sd_right_matmul` left / right product with a sparse constant matrix A .

`sd_left_matmul_dense, sd_right_matmul_dense` left / right product with a dense constant or parametric matrix.

`sd_left_kron, sd_right_kron` the Kronecker product $A \otimes B$ with the variable-free operand `param` on the left (A) or on the right (B) and the variable operand `child` on the other side. `param` may be a parameter or a constant made with `sd_parameter(..., param_id = -1, ...)`.

A parameter cannot be the source of the sparse products `sd_left_matmul / sd_right_matmul` (the engine does not support it; 'sparsediffpy' accepts the argument but the engine then aborts); use the dense products for a parametric matrix.

Value

An expression handle.

See Also

[sd_parameter](#), [sd_register_params](#)

sparsediff-oracle	<i>Sparse derivative oracle</i>
-------------------	---------------------------------

Description

Initialize and evaluate the value, gradient, sparse constraint Jacobian and sparse Lagrangian Hessian of a problem built with [sd_problem](#). Derivatives come in two forms, matching 'sparsediffpy': coordinate (COO) form, with the Hessian as its lower triangle (the form CVXPY uses), and compressed-sparse-row (CSR) form, with the full Hessian.

Usage

```
sd_init_derivatives(prob)
sd_init_jacobian(prob)
sd_init_jacobian_coo(prob)
sd_init_hessian_coo(prob)
sd_objective_forward(prob, u)
sd_constraint_forward(prob, u)
sd_gradient(prob)
sd_jacobian_sparsity(prob)
sd_jacobian_values(prob)
sd_hessian_sparsity(prob)
sd_hessian_values(prob, obj_w, w)
sd_jacobian(prob)
sd_get_jacobian(prob)
sd_init_hessian(prob)
sd_hessian(prob, obj_w, w)
sd_get_hessian(prob)
```

Arguments

<code>prob</code>	a problem handle from sd_problem .
<code>u</code>	a numeric vector: the primal point at which to evaluate, laid out in the engine's column-major flat ordering (the same ordering used by the <code>var_id</code> offsets passed to sd_variable).
<code>obj_w</code>	a scalar weight σ multiplying the objective Hessian.
<code>w</code>	a numeric vector of constraint multipliers (length equal to the total constraint size) weighting the constraint Hessians.

Details

Evaluation is ordered: a forward pass first (`sd_objective_forward` / `sd_constraint_forward`) populates the node values at `u`, after which `sd_gradient`, `sd_jacobian_values` and `sd_hessian_values` read them. Sparsity patterns are structural — fixed once the corresponding `sd_init_*` routine has run — so they are queried once and reused, while the values are recomputed at each new point. Row and column indices are 0-based (the engine convention; a higher-level modeling layer such as **CVXR** translates them to 1-based as needed).

Value

`sd_init_derivatives`, `sd_init_jacobian`, `sd_init_jacobian_coo`, `sd_init_hessian`, `sd_init_hessian_coo` called for their side effect; return NULL invisibly.

`sd_jacobian`, `sd_get_jacobian`, `sd_hessian`, `sd_get_hessian` a list with the CSR arrays of the matrix: `data` (double), `indices` (0-based column indices) and `indptr` (row pointers), plus `shape = c(nrow, ncol)`. `sd_jacobian` evaluates the constraint Jacobian (after `sd_init_jacobian` and a forward pass) and `sd_hessian` the full Lagrangian Hessian $\sigma \nabla^2 f + \sum_i w_i \nabla^2 g_i$ (after `sd_init_hessian`); the `sd_get_*` forms return the matrix as last evaluated, without re-evaluating it.

`sd_objective_forward` the scalar objective value at `u`.

`sd_constraint_forward` the constraint vector at `u`.

`sd_gradient` the objective gradient, length `n_vars`.

`sd_jacobian_sparsity`, `sd_hessian_sparsity` a list with integer rows/cols (0-based COO indices) and `nrow/ncol`.

`sd_jacobian_values` the constraint-Jacobian nonzeros, matching the Jacobian sparsity order.

`sd_hessian_values` the nonzeros of $\sigma \nabla^2 f + \sum_i w_i \nabla^2 g_i$, lower triangle, matching the Hessian sparsity order.

See Also

[sd_problem](#)

sparsediff-problem	<i>Assemble a differentiable problem</i>
--------------------	--

Description

Combine an objective expression and a list of constraint expressions (built with the `sd_*` atom constructors) into a single problem object whose value and sparse derivatives can be evaluated repeatedly.

Usage

```
sd_problem(objective, constraints, verbose)
sd_register_params(prob, params)
sd_update_params(prob, theta)
```

Arguments

objective	an expression handle for the scalar objective.
constraints	a list of expression handles, stacked vertically to form the constraint vector $g(x)$ (may be empty).
verbose	logical; if TRUE, print diagnostic information while the problem is assembled.
prob	a problem handle returned by <code>sd_problem()</code> .
params	a list of parameter expression handles (see sd_parameter) to register for fast re-evaluation under changing data.
theta	a numeric vector of new parameter values, concatenated in the order the parameters were registered.

Details

The typical lifecycle is: build expressions with the `sd_*` constructors, assemble them with `sd_problem()`, initialise the derivative structures once ([sd_init_derivatives](#) and friends), then evaluate the objective/constraints and their sparse derivatives at as many primal points as needed. When the problem has parameters, register them once with `sd_register_params()` and push new values with `sd_update_params()` to re-evaluate without rebuilding the graph (the DPP-style fast path).

Value

`sd_problem()` returns an external-pointer problem handle. The handle owns the underlying expression graph and frees it when garbage collected. `sd_register_params()` and `sd_update_params()` are called for their side effect and return NULL invisibly.

See Also

[sd_init_derivatives](#) for the evaluation oracle, [sd_variable](#) and the atom constructors for building expressions.

sparsediff-reduction *Product-reduction atoms*

Description

Multiplicative reductions of an expression.

Usage

```
sd_prod(c)
sd_prod_axis_zero(c)
sd_prod_axis_one(c)
```

Arguments

c an expression handle.

Details

sd_prod product of all entries.
sd_prod_axis_zero column-wise products (reduce down rows).
sd_prod_axis_one row-wise products (reduce across columns).

Value

An expression handle.

See Also

[sparsediff-affine](#)

Index

engine_version, 2

sd_add (sparsediff-affine), 3

sd_asinh (sparsediff-elementwise), 5

sd_atanh (sparsediff-elementwise), 5

sd_broadcast (sparsediff-affine), 3

sd_constraint_forward
(sparsediff-oracle), 9

sd_convolve (sparsediff-matrix), 7

sd_cos (sparsediff-elementwise), 5

sd_diag_mat (sparsediff-affine), 3

sd_diag_vec (sparsediff-affine), 3

sd_elementwise_mult
(sparsediff-bivariate), 4

sd_entr (sparsediff-elementwise), 5

sd_exp (sparsediff-elementwise), 5

sd_get_expr_dimensions
(sparsediff-getters), 6

sd_get_expr_size (sparsediff-getters), 6

sd_get_hessian (sparsediff-oracle), 9

sd_get_jacobian (sparsediff-oracle), 9

sd_gradient (sparsediff-oracle), 9

sd_hessian (sparsediff-oracle), 9

sd_hessian_sparsity
(sparsediff-oracle), 9

sd_hessian_values (sparsediff-oracle), 9

sd_hstack (sparsediff-affine), 3

sd_index (sparsediff-affine), 3

sd_init_derivatives, 11

sd_init_derivatives
(sparsediff-oracle), 9

sd_init_hessian (sparsediff-oracle), 9

sd_init_hessian_coo
(sparsediff-oracle), 9

sd_init_jacobian (sparsediff-oracle), 9

sd_init_jacobian_coo
(sparsediff-oracle), 9

sd_jacobian (sparsediff-oracle), 9

sd_jacobian_sparsity
(sparsediff-oracle), 9

sd_jacobian_values (sparsediff-oracle),
9

sd_left_kron (sparsediff-matrix), 7

sd_left_matmul (sparsediff-matrix), 7

sd_left_matmul_dense
(sparsediff-matrix), 7

sd_log (sparsediff-elementwise), 5

sd_logistic (sparsediff-elementwise), 5

sd_matmul (sparsediff-bivariate), 4

sd_neg (sparsediff-elementwise), 5

sd_normal_cdf (sparsediff-elementwise),
5

sd_objective_forward
(sparsediff-oracle), 9

sd_parameter, 8, 9, 11

sd_parameter (sparsediff-leaves), 7

sd_power (sparsediff-elementwise), 5

sd_problem, 7, 9, 10

sd_problem (sparsediff-problem), 11

sd_prod (sparsediff-reduction), 12

sd_prod_axis_one
(sparsediff-reduction), 12

sd_prod_axis_zero
(sparsediff-reduction), 12

sd_promote (sparsediff-affine), 3

sd_quad_form (sparsediff-matrix), 7

sd_quad_form_dense (sparsediff-matrix),
7

sd_quad_over_lin
(sparsediff-bivariate), 4

sd_register_params, 7, 9

sd_register_params
(sparsediff-problem), 11

sd_rel_entr (sparsediff-bivariate), 4

sd_rel_entr_first_scalar
(sparsediff-bivariate), 4

sd_rel_entr_second_scalar
(sparsediff-bivariate), 4

sd_reshape (sparsediff-affine), 3

`sd_right_kron` (sparsediff-matrix), 7
`sd_right_matmul` (sparsediff-matrix), 7
`sd_right_matmul_dense`
 (sparsediff-matrix), 7
`sd_scalar_mult` (sparsediff-matrix), 7
`sd_sin` (sparsediff-elementwise), 5
`sd_sinh` (sparsediff-elementwise), 5
`sd_sum` (sparsediff-affine), 3
`sd_tan` (sparsediff-elementwise), 5
`sd_tanh` (sparsediff-elementwise), 5
`sd_trace` (sparsediff-affine), 3
`sd_transpose` (sparsediff-affine), 3
`sd_update_params`, 8
`sd_update_params` (sparsediff-problem),
 11
`sd_upper_tri` (sparsediff-affine), 3
`sd_variable`, 10, 11
`sd_variable` (sparsediff-leaves), 7
`sd_vector_mult` (sparsediff-matrix), 7
`sd_vstack` (sparsediff-affine), 3
`sd_xexp` (sparsediff-elementwise), 5
sparsediff-affine, 3
sparsediff-bivariate, 4
sparsediff-elementwise, 5
sparsediff-getters, 6
sparsediff-leaves, 7
sparsediff-matrix, 7
sparsediff-oracle, 9
sparsediff-problem, 11
sparsediff-reduction, 12