Package ‘dspline’

May 14, 2025

Title Tools for Computations with Discrete Splines

Version 1.0.2

Description Discrete splines are a class of univariate piecewise polynomial functions which are analogous to splines, but whose smoothness is defined via divided differences rather than derivatives. Tools for efficient computations relating to discrete splines are provided here. These tools include discrete differentiation and integration, various matrix computations with discrete derivative or discrete spline bases matrices, and interpolation within discrete spline spaces. These techniques are described in Tibshirani (2020) <doi:10.48550/arXiv.2003.03886>.

License MIT + file LICENSE

URL `https://github.com/glmgen/dspline`,
`https://glmgen.github.io/dspline/`

BugReports `https://github.com/glmgen/dspline/issues`

Imports Matrix, Rcpp, rlang

Suggests testthat (>= 3.0.0)

LinkingTo Rcpp, RcppEigen

Config/testthat/edition 3

Encoding UTF-8

RoxygenNote 7.3.2

NeedsCompilation yes

Author Logan Brooks [ctb],
Addison Hu [aut],
Daniel McDonald [ctb],
Ryan Tibshirani [aut, cre, cph]

Maintainer Ryan Tibshirani <`ryantibs@gmail.com`>

Repository CRAN

Date/Publication 2025-05-14 08:50:04 UTC

Contents

`b_mat` *Construct B matrix*

Description

Constructs the extended discrete derivative matrix of a given order, with respect to given design points.

Usage

```
b_mat(k, xd, tf_weighting = FALSE, row_idx = NULL)
```

Arguments

`k`	Order for the extended discrete derivative matrix. Must be >= 0.
`xd`	Design points. Must be sorted in increasing order, and have length at least `k+1`.
`tf_weighting`	Should "trend filtering weighting" be used? This is a weighting of the discrete derivatives that is implicit in trend filtering; see details for more information. The default is `FALSE`.
`row_idx`	Vector of indices, a subset of `1:n` where `n = length(xd)`, that indicates which rows of the constructed matrix should be returned. The default is `NULL`, which is taken to mean `1:n`.

Details

The extended discrete derivative matrix of order k, with respect to design points $x_1 < \ldots < x_n$, is denoted B^k_n. It has dimension $n \times n$, and is banded with bandwidth $k + 1$. It can be constructed recursively, as follows. For $k \geq 1$, we first define the $n \times n$ extended difference matrix $\bar{B}_{n,k}$:

$$\bar{B}_{n,k} = \left[\begin{array}{ccccccccc} 1 & 0 & \ldots & 0 & & & & & \\ 0 & 1 & \ldots & 0 & & & & & \\ \vdots & & & & & & & & \\ 0 & 0 & \ldots & 1 & & & & & \\ & & & -1 & 1 & 0 & \ldots & 0 & 0 \\ & & & 0 & -1 & 1 & \ldots & 0 & 0 \\ & & & \vdots & & & & & \\ & & & 0 & 0 & 0 & \ldots & -1 & 1 \end{array}\right] \begin{array}{l} \left.\vphantom{\begin{array}{c}1\\0\\\vdots\\0\end{array}}\right\} k \text{ rows} \\ \left.\vphantom{\begin{array}{c}1\\0\\\vdots\\0\end{array}}\right\} n-k \text{ rows} \end{array} .$$

We also define the $n \times n$ extended diagonal weight matrix Z^k_n to have first k diagonal entries equal to 1 and last $n-k$ diagonal entries equal to $(x_{i+k} - x_i)/k$, $i = 1, \ldots, n-k$. The kth order extended discrete derivative matrix B^k_n is then given by the recursion:

$$\begin{aligned} B^1_n &= (Z^1_n)^{-1} \bar{B}_{n,1}, \\ B^k_n &= (Z^k_n)^{-1} \bar{B}_{n,k} \, B^{k-1}_n, \quad \text{for } k \geq 2. \end{aligned}$$

We note that the discrete derivative matrix D^k_n from `d_mat()` is simply given by the last $n - k$ rows of the extended matrix B^k_n.

The option `tf_weighting = TRUE` returns $Z^k_n B^k_n$ where Z^k_n is the $n \times n$ diagonal matrix as described above. This weighting is implicit in trend filtering, as explained in the help file for `d_mat_mult()`. See also Sections 6.1 and 6.2 of Tibshirani (2020) for further discussion.

Note: For multiplication of a given vector by B^k_n, instead of forming B^k_n with the current function and then carrying out the multiplication, one should instead use `b_mat_mult()`, as this will be more efficient (both will be linear time, but the latter saves the cost of forming any matrix in the first place).

Value

Sparse matrix of dimension `length(row_idx)` by `length(xd)`.

References

Tibshirani (2020), "Divided differences, falling factorials, and discrete splines: Another look at trend filtering and related problems", Section 6.2.

See Also

`d_mat()` for constructing the discrete derivative matrix, and `b_mat_mult()` for multiplying by the extended discrete derivative matrix.

Examples

```
b_mat(2, 1:10)
b_mat(2, 1:10 / 10)
b_mat(2, 1:10, row_idx = 4:7)
```

`b_mat_mult` *Multiply by B matrix*

Description

Multiplies a given vector by B, the extended discrete derivative matrix of a given order, with respect to given design points.

Usage

```
b_mat_mult(v, k, xd, tf_weighting = FALSE, transpose = FALSE, inverse = FALSE)
```

Arguments

`v`	Vector to be multiplied by B, the extended discrete derivative matrix.
`k`	Order for the extended discrete derivative matrix. Must be >= 0.
`xd`	Design points. Must be sorted in increasing order, and have length at least `k+1`.
`tf_weighting`	Should "trend filtering weighting" be used? This is a weighting of the discrete derivatives that is implicit in trend filtering; see details for more information. The default is `FALSE`.
`transpose`	Multiply by the transpose of B? The default is `FALSE`.
`inverse`	Multiply by the inverse of B? The default is `FALSE`.

Details

The extended discrete derivative matrix of order k, with respect to design points $x_1 < \ldots < x_n$, is denoted B^k_n. It is square, having dimension $n \times n$. Acting on a vector v of function evaluations at the design points, denoted $v = f(x_{1:n})$, it gives the discrete derivatives of f at the points $x_{1:n}$:

$$B^k_n v = (\Delta^k_n f)(x_{1:n}).$$

The matrix B^k_n can be constructed recursively as the product of a diagonally-weighted first difference matrix and B^{k-1}_n; see the help file for `b_mat()`, or Section 6.2 of Tibshirani (2020). Therefore, multiplication by B^k_n or by its transpose can be performed in $O(nk)$ operations based on iterated weighted differences. See Appendix D of Tibshirani (2020) for details.

The option `tf_weighting = TRUE` performs multiplication by $Z^k_n B^k_n$ where Z^k_n is an $n \times n$ diagonal matrix whose top left $k \times k$ block equals the identity matrix and bottom right $(n-k) \times (n-k)$ block equals W^k_n, the latter being a diagonal weight matrix that is implicit in trend filtering, as explained in the help file for `d_mat_mult()`.

Lastly, the matrix B_n^k has a special **inverse relationship** to the falling factorial basis matrix H_n^{k-1} of degree $k-1$ with knots in $x_{k:(n-1)}$; it satisfies:

$$Z_n^k B_n^k H_n^{k-1} = I_n,$$

where Z_n^k is the $n \times n$ diagonal matrix as described above, and I_n is the $n \times n$ identity matrix. This, combined with the fact that the falling factorial basis matrix has an efficient recursive representation in terms of weighted cumulative sums, means that multiplying by $(B_n^k)^{-1}$ or its transpose can be performed in $O(nk)$ operations. See Section 6.3 and Appendix D of Tibshirani (2020) for details.

Value

Product of the extended discrete derivative matrix B and the input vector `v`.

References

Tibshirani (2020), "Divided differences, falling factorials, and discrete splines: Another look at trend filtering and related problems", Section 6.2.

See Also

`discrete_deriv()` for discrete differentiation at arbitrary query points, `d_mat_mult()` for multiplying by the discrete derivative matrix, and `b_mat()` for constructing the extended discrete derivative matrix.

Examples

```
v = sort(runif(10))
as.vector(b_mat(2, 1:10) %*% v)
b_mat_mult(v, 2, 1:10)
```

`discrete_deriv` *Discrete differentiation*

Description

Computes the discrete derivative of a function (or vector of function evaluations) of a given order, with respect to given design points, and evaluated at a given query point.

Usage

```
discrete_deriv(f, k, xd, x)
```

Arguments

`f`	Function, or vector of function evaluations at `c(xd, x)`, the design points `xd` adjoined with the query point(s) `x`.
`k`	Order for the discrete derivative calculation. Must be >= 0.
`xd`	Design points. Must be sorted in increasing order, and have length at least `k+1`.
`x`	Query point(s).

Details

The discrete derivative operator of order k, with respect design points $x_1 < \ldots < x_n$, is denoted Δ_n^k. Acting on a function f, and evaluated at a query point x, it yields:

$$(\Delta_n^k f)(x) = \begin{cases} k! \cdot f[x_{i-k+1}, \ldots, x_i, x] & \text{if } x \in (x_i, x_{i+1}], i \ge k \\ i! \cdot f[x_1, \ldots, x_i, x] & \text{if } x \in (x_i, x_{i+1}], i < k \\ f(x) & \text{if } x \le x_1, \end{cases}$$

where we take $x_{n+1} = \infty$ for convenience. In other words, for "most" points $x > x_k$, we define $(\Delta_n^k f)(x)$ in terms of a (scaled) divided difference of f of order k, where the centers are the k points immediately to the left of x, and x itself. Meanwhile, for "boundary" points $x \le x_k$, we define $(\Delta_n^k f)(x)$ to be a (scaled) divided difference of f of the highest possible order, where the centers are the points to the left of x, and x itself. For more discussion, including alternative representations for the discrete differentiation, see Section 3.1 of Tibshirani (2020).

Note: for calculating discrete derivatives at the design points themselves, which could be achieved by taking `x = xd` in the current function, one should instead use `b_mat_mult()` or `d_mat_mult()`, as these will be more efficient (both will be linear-time, but the latter functions will be faster).

Value

Discrete derivative of `f` of order `k`, with respect to design points `xd`, evaluated at the query point(s) `x`.

References

Tibshirani (2020), "Divided differences, falling factorials, and discrete splines: Another look at trend filtering and related problems", Section 3.1.

See Also

`b_mat_mult()`, `d_mat_mult()` for multiplication by the extended and non-extended discrete derivative matrices, giving discrete derivatives at design points.

Examples

```
xd = 1:10 / 10
discrete_deriv(function(x) x^2, 1, xd, xd)
```

`discrete_integ` *Discrete integration*

Description

Computes the discrete integral of a function (or vector of function evaluations) of a given order, with respect to given design points, and evaluated at a given query point.

Usage

```
discrete_integ(f, k, xd, x)
```

Arguments

`f`	Function, or vector of function evaluations at `c(xd, x)`, the design points `xd` adjoined with the query point(s) `x`.
`k`	Order for the discrete integral calculation. Must be >= 0.
`xd`	Design points. Must be sorted in increasing order, and have length at least `k+1`.
`x`	Query point(s).

Details

The discrete integral operator of order k, with respect to design points $x_1 < \ldots < x_n$, is denoted S^k_n. It is the inverse operator to the discrete derivative operator Δ^k_n, so that:

$$S^k_n \Delta^k_n = \Delta^k_n S^k_n = \mathrm{Id},$$

where Id denotes the identity operator. It can also be represented in a more explicit form, as follows. Acting on a function f of order k, and evaluated at a query point x, it yields:

$$(S^k_n f)(x) = \begin{cases} \displaystyle\sum_{j=1}^{k} h^{k-1}_j(x) \cdot f(x_j) + \sum_{j=k+1}^{i} h^{k-1}_j(x) \cdot \frac{x_j - x_{j-k}}{k} \cdot f(x_j) + h^{k-1}_{i+1}(x) \cdot \frac{x - x_{i-k+1}}{k} \cdot f(x) & \\ & \text{if } x \in (x_i, x_{i+1}],\ i \geq k \\ \displaystyle\sum_{j=1}^{i} h^{k-1}_j(x) \cdot f(x_j) \,+\, h^{k-1}_{i+1}(x) \cdot f(x) & \text{if } x \in (x_i, x_{i+1}],\ i < k \\ f(x) & \text{if } x \leq x_1, \end{cases}$$

where $h^{k-1}_1, \ldots, h^{k-1}_n$ denote the falling factorial basis functions of degree $k-1$, with knots in $x_{k:(n-1)}$. The help file for `h_mat()` gives a definition of the falling factorial basis. It can be seen (due to the one-sided support of the falling factorial basis functions) that discrete integration at $x = x_i$, $i = 1, \ldots, n$ is equivalent to multiplication by a weighted version of the falling factorial basis matrix. For more details, including an alternative recursive representation for discrete integration (that elucidates its relationship to discrete differentiation), see Section 3.2 of Tibshirani (2020).

Note: for calculating discrete integrals at the design points themselves, which could be achieved by taking `x = xd` in the current function, one should instead use `h_mat_mult()` with `di_weighting = TRUE`, as this will be **much** more efficient (quadratic-time versus linear-time).

Value

Discrete integral of `f` of order `k`, with respect to design points `xd`, evaluated at the query point(s) `x`.

References

Tibshirani (2020), "Divided differences, falling factorials, and discrete splines: Another look at trend filtering and related problems", Section 3.2.

See Also

h_mat_mult() for multiplication by the falling factorial basis matrix, giving a weighted analog of discrete integration at the design points.

Examples

```
xd = 1:10 / 10
discrete_integ(function(x) 1, 1, xd, xd)
```

divided_diff *Divided differencing*

Description

Computes the divided difference of a function (or vector of function evaluations) with respect to given centers.

Usage

```
divided_diff(f, z)
```

Arguments

f	Function, or vector of function evaluations at the centers.
z	Centers for the divided difference calculation.

Details

The divided difference of a function f with respect to centers $z_1, \ldots, z_{k+1}$ is defined recursively as:

$$f[z_1, \ldots, z_{k+1}] = \frac{f[z_2, \ldots, z_{k+1}] - f[z_1, \ldots, z_k]}{z_{k+1} - z_1},$$

with base case $f[z_1] = f(z_1)$ (that is, divided differencing with respect to a single point reduces to function evaluation).

A notable special case is when the centers are evenly-spaced, say, $z_i = z + ih$, $i = 0, \ldots, k$ for some spacing $h > 0$, in which case the divided difference becomes a (scaled) forward difference, or equivalently a (scaled) backward difference,

$$k! \cdot f[z, \ldots, z + kh] = \frac{1}{h^k}(F_h^k f)(z) = \frac{1}{h^k}(B_h^k f)(z + kh),$$

where we use F_h^k and B_v^k to denote the forward and backward difference operators, respectively, of order k and with spacing h.

Value

Divided difference of f with respect to centers z.

Examples

```
f = runif(4)
z = runif(4)
divided_diff(f[1], z[1])
f[1]
divided_diff(f[1:2], z[1:2])
(f[1]-f[2])/(z[1]-z[2])
divided_diff(f[1:3], z[1:3])
((f[1]-f[2])/(z[1]-z[2]) - (f[2]-f[3])/(z[2]-z[3])) / (z[1]-z[3])
divided_diff(f, 1:4)
diff(f, diff = 3) / factorial(3)
```

`dot_functions` *In-place computations*

Description

Each "dot" function accepts arguments as in its "non-dot" counterpart, but peforms computations in place, overwriting the first input argument (which must be a vector of the appropriate length) with the desired output.

Usage

```
.divided_diff(f, z)

.b_mat_mult(v, k, xd, tf_weighting, transpose, inverse)

.h_mat_mult(v, k, xd, di_weighting, transpose, inverse)
```

Arguments

`f`	Function, or vector of function evaluations at the centers.
`z`	Centers for the divided difference calculation.
`v`	Vector to be multiplied by B, the extended discrete derivative matrix.
`k`	Order for the extended discrete derivative matrix. Must be >= 0.
`xd`	Design points. Must be sorted in increasing order, and have length at least `k+1`.
`tf_weighting`	Should "trend filtering weighting" be used? This is a weighting of the discrete derivatives that is implicit in trend filtering; see details for more information. The default is `FALSE`.
`transpose`	Multiply by the transpose of B? The default is `FALSE`.
`inverse`	Multiply by the inverse of B? The default is `FALSE`.
`di_weighting`	Should "discrete integration weighting" be used? Multiplication by such a weighted H gives discrete integrals at the design points; see details for more information. The default is `FALSE`.

Details

These functions should not be used unless you are intentionally doing so for memory considerations and are nonetheless being very careful.

An **important warning:** each "dot" function only works as expected if its first argument is passed in as a vector of numeric type. If the first argument is passed in as an integer vector, then since the output must (in general) be a numeric vector, it cannot be computed in place (Rcpp performs an implicit cast and copy when it converts this to NumericVector type for use in C++).

Also, each "dot" function does not perform any error checking on its input arguments. Use with care. More details on the computations performed by individual functions are provided below.

Value

None. These functions *overwrite* their input.

`.divided_diff()`

Overwrites `f` with all lower-order divided differences: each element `f[i]` becomes the divided difference with respect to centers `z[1:i]`. See also `divided_diff()`.

`.b_mat_mult()`

Overwrites `v` with `B %*% v`, where `B` is the extended discrete derivative matrix as returned by `b_mat()`. See also `b_mat_mult()`.

`.h_mat_mult()`

Overwrites `v` with `H %*% v`, where `H` is the falling factorial basis matrix as returned by `h_mat()`. See also `h_mat_mult()`.

Examples

```
v = as.numeric(1:10) # Note: must be of numeric type
b_mat_mult(v, 1, 1:10)
v
.b_mat_mult(v, 1, 1:10, FALSE, FALSE, FALSE)
v
```

`dspline_interp` *Discrete spline interpolation*

Description

Interpolates a sequence of values within the "canonical" space of discrete splines of a given order, with respect to given design points.

Usage

```
dspline_interp(v, k, xd, x, implicit = TRUE)
```

Arguments

`v`	Vector to be values to be interpolated, one value per design point.
`k`	Order for the discrete spline space. Must be >= 0.
`xd`	Design points. Must be sorted in increasing order, and have length at least `k+1`.
`x`	Query point(s), at which to perform interpolation.
`implicit`	Should implicit form interpolation be used? See details for what this means. The default is `TRUE`.

Details

The "canonical" space of discrete splines of degree k, with respect to design points $x_{1:n}$, is spanned by the falling factorial basis functions $h_1^k, \dots, h_n^k$, which are defined as:

$$h_j^k(x) = \frac{1}{(j-1)!} \prod_{\ell=1}^{j-1} (x - x_\ell), \quad j = 1, \dots, k+1,$$
$$h_j^k(x) = \frac{1}{k!} \prod_{\ell=j-k}^{j-1} (x - x_\ell) \cdot 1\{x > x_{j-1}\}, \quad j = k+2, \dots, n.$$

Their span is a space of piecewise polynomials of degree k with knots in $x_{(k+1):(n-1)}$—in fact, not just any piecewise polynomials, but special ones that have continuous discrete derivatives (defined in terms of divided differences; see the help file for `discrete_deriv()` for the definition) of all orders $0, \dots, k-1$ at the knot points. This is precisely analogous to splines but with derivatives replaced by discrete derivatves, hence the name discrete splines. See Section 4.1 of Tibshirani (2020) for more details.

As the space of discrete splines of degree k with knots in $x_{(k+1):(n-1)}$ has linear dimension n, any sequence of n values (one at each of the design points $x_{1:n}$) has a unique discrete spline interpolant. Evaluating this interpolant at any query point x can be done via its falling factorial basis expansion, where the coefficients in this expansion can be computed efficiently (in $O(nk)$ operations) due to the fact that the inverse falling factorial basis matrix can be represented in terms of extended discrete derivatives (see the help file for `h_mat_mult()` for details).

When `implicit = FALSE`, the interpolation is carried out as described in the above paragraph. It is worth noting this is a strict generalization of **Newton's divided difference interpolation**, which is given by the special case when $n = k + 1$ (in this case the knot set is empty, and the "canonical" space of degree k discrete splines is nothing more than the space of degree k polynomials). See Section 5.3 of Tibshirani (2020) for more details.

When `implicit = TRUE`, an implicit form is used to evaluate the interpolant at an arbitrary query point x, which locates x among the design points $x_{1:n}$ (a $O(\log n)$ computational cost), and solves for the of value of $f(x)$ that results in a local order $k + 1$ discrete derivative being equal to zero (a $O(k)$ computational cost). This is generally a more efficient and stable scheme for interpolation. See Section 5.4 of Tibshirani (2020) for more details.

Value

Value(s) of the unique discrete spline interpolant (defined by the values `v` at design points `xd`) at query point(s) `x`.

References

Tibshirani (2020), "Divided differences, falling factorials, and discrete splines: Another look at trend filtering and related problems", Section 5.

See Also

`dspline_solve()` for the least squares projection onto a "custom" space of discrete splines (defined by a custom knot set $T \subseteq x_{(k+1):(n-1)}$).

Examples

```
xd = 1:100 / 100
knot_idx = 1:9 * 10
y = sin(2 * pi * xd) + rnorm(100, 0, 0.2)
yhat = dspline_solve(y, 2, xd, knot_idx)$fit
x = seq(0, 1, length = 1000)
fhat = dspline_interp(yhat, 2, xd, x)
plot(xd, y, pch = 16, col = "gray65")
lines(x, fhat, col = "firebrick", lwd = 2)
abline(v = xd[knot_idx], lty = 2)
```

`dspline_solve` *Discrete spline projection*

Description

Projects a sequence of values onto the space of discrete splines a given order, with respect to given design points, and given knot points.

Usage

```
dspline_solve(v, k, xd, knot_idx, basis = c("N", "B", "H"), mat)
```

Arguments

`v`	Vector to be values to be projected, one value per design point.
`k`	Order for the discrete spline space. Must be >= 0.
`xd`	Design points. Must be sorted in increasing order, and have length at least `k+1`.
`knot_idx`	Vector of indices, a subset of `(k+1):(n-1)` where `n = length(xd)`, that indicates which design points should be used as knot points for the discrete spline space. Must be sorted in increasing order.
`basis`	String, one of `"N"`, `"B"`, or `"H"`, indicating which basis representation is to be used for the least squares computation. The default is `"N"`, the discrete B-spline basis. See details for more information.
`mat`	Matrix to use for the least squares computation. If missing, the default, then the matrix will be formed according to the `basis` argument. See details for more information.

Details

This function minimizes the least squares criterion

$$\|v - M\beta\|_2^2$$

over coefficient vectors β; that is, it computes

$$\hat{\beta} = (M^T M)^{-1} M^T v$$

for a vector v and basis matrix M. The basis matrix M is specified via the `basis` argument, which allows for three options. The default is `"N"`, in which case the discrete B-spline basis matrix from `n_mat()` is used, with the `knot_idx` argument set accordingly. This is generally the **most stable and efficient** option: it leads to a banded, well-conditioned linear system. Bandedness means that the least squares projection can be computed in $O(nk^2)$ operations. See Section 8.4 of Tibshirani (2020) for numerical experiments investigating conditioning.

The option `"H"` means that the falling factorial basis matrix from `h_mat()` is used, with the `col_idx` argument set appropriately. This option should be **avoided in general** since it leads to a linear system that is neither sparse nor well-conditioned.

The option `"B"` means that the extended discrete derivative matrix from `b_mat()`, with `tf_weighting = TRUE`, is used to compute the least squares solution from projecting onto the falling factorial basis. The fact this is possible stems from a special inverse relationship between the discrete derivative matrix and the falling factorial basis matrix. While this option leads to a banded linear system, this system tends to have worse conditioning than that using the discrete B-spline representation. However, it is essentially always preferable to the `"H"` option, and it produces the same solution (coefficients in the falling factorial basis expansion).

Note 1: the basis matrix to be used in the least squares problem can be passed in directly via the `mat` argument (which saves on the cost of computing it in the first place). Even when `mat` not missing, the `basis` argument must still be used to specify which type of basis matrix is being passed in, as the downstream computations differ depending on the type.

Note 2: when `mat` is not missing and `basis = "B"`, the matrix being passed in must be the **entire** extended discrete derivative matrix, as returned by `b_mat()` with `row_idx = NULL` (the default), and not some row-subsetted version. This is because both the rows corresponding to the knots in the discrete spline space and the complementary set of roles play a role in computing the solution. See Section 8.1 of Tibshirani (2020).

Value

List with components `sol`: the least squares solution; `fit`: the least squares fit; and `mat`: the basis matrix used for the least squares problem (only present if the input argument `mat` is missing).

References

Tibshirani (2020), "Divided differences, falling factorials, and discrete splines: Another look at trend filtering and related problems", Sections 8.1 and 8.4.

See Also

`dspline_interp()` for interpolation within the "canonical" space of discrete splines.

Examples

```
xd = 1:100 / 100
knot_idx = 1:9 * 10
y = sin(2 * pi * xd) + rnorm(100, 0, 0.2)
yhat = res = dspline_solve(y, 2, xd, knot_idx)$fit
plot(xd, y, pch = 16, col = "gray60")
points(xd, yhat, col = "firebrick")
abline(v = xd[knot_idx], lty = 2)
```

---

d_mat *Construct D matrix*

Description

Constructs the discrete derivative matrix of a given order, with respect to given design points.

Usage

```
d_mat(k, xd, tf_weighting = FALSE, row_idx = NULL)
```

Arguments

k	Order for the discrete derivative matrix. Must be >= 0.
xd	Design points. Must be sorted in increasing order, and have length at least k+1.
tf_weighting	Should "trend filtering weighting" be used? This is a weighting of the discrete derivatives that is implicit in trend filtering; see details for more information. The default is FALSE.
row_idx	Vector of indices, a subset of 1:(n-k) where n = length(xd), that indicates which rows of the constructed matrix should be returned. The default is NULL, which is taken to mean 1:(n-k).

Details

The discrete derivative matrix of order k, with respect to design points $x_1 < \ldots < x_n$, is denoted D_n^k. It has dimension $(n-k) \times n$, and is banded with bandwidth $k+1$. It can be constructed recursively, as follows. We first define the $(n-1) \times n$ first difference matrix $\bar{D}_n$:

$$\bar{D}_n = \begin{bmatrix} -1 & 1 & 0 & \ldots & 0 & 0 \\ 0 & -1 & 1 & \ldots & 0 & 0 \\ \vdots & & & & & \\ 0 & 0 & 0 & \ldots & -1 & 1 \end{bmatrix},$$

and for $k \geq 1$, define the $(n-k) \times (n-k)$ diagonal weight matrix W_n^k to have diagonal entries $(x_{i+k} - x_i)/k$, $i = 1, \ldots, n-k$. The kth order discrete derivative matrix D_n^k is then given by the recursion:

$$D_n^1 = (W_n^1)^{-1} \bar{D}_n,$$
$$D_n^k = (W_n^k)^{-1} \bar{D}_{n-k+1} \, D_n^{k-1}, \quad \text{for } k \geq 2.$$

We note that $\bar{D}_{n-k+1}$ above denotes the $(n-k) \times (n-k+1)$ version of the first difference matrix that is defined in the second-to-last display.

The option `tf_weighting = TRUE` returns $W_n^k D_n^k$ where W_n^k is the $(n-k) \times (n-k)$ diagonal matrix as described above. This weighting is implicit in trend filtering, as explained in the help file for `d_mat_mult()`. See also Section 6.1 of Tibshirani (2020) for further discussion.

Note: For multiplication of a given vector by D_n^k, instead of forming D_n^k with the current function and then carrying out the multiplication, one should instead use `d_mat_mult()`, as this will be more efficient (both will be linear time, but the latter saves the cost of forming any matrix in the first place).

Value

Sparse matrix of dimension `length(row_idx)` by `length(xd)`.

References

Tibshirani (2020), "Divided differences, falling factorials, and discrete splines: Another look at trend filtering and related problems", Section 6.1.

See Also

`b_mat()` for constructing the extended discrete derivative matrix, and `d_mat_mult()` for multiplying by the discrete derivative matrix.

Examples

```
d_mat(2, 1:10)
d_mat(2, 1:10 / 10)
d_mat(2, 1:10, row_idx = 2:5)
```

`d_mat_mult` *Multiply by D matrix*

Description

Multiplies a given vector by D, the discrete derivative matrix of a given order, with respect to given design points.

Usage

```
d_mat_mult(v, k, xd, tf_weighting = FALSE, transpose = FALSE)
```

Arguments

`v`	Vector to be multiplied by D, the discrete derivative matrix.
`k`	Order for the discrete derivative matrix. Must be >= 0.
`xd`	Design points. Must be sorted in increasing order, and have length at least `k+1`.
`tf_weighting`	Should "trend filtering weighting" be used? This is a weighting of the discrete derivatives that is implicit in trend filtering; see details for more information. The default is `FALSE`.
`transpose`	Multiply by the transpose of D? The default is `FALSE`.

Details

The discrete derivative matrix of order k, with respect to design points $x_1 < \ldots < x_n$, is denoted D^k_n. It has dimension $(n-k) \times n$. Acting on a vector v of function evaluations at the design points, denoted $v = f(x_{1:n})$, it gives the discrete derivatives of f at the points $x_{(k+1):n}$:

$$D^k_n v = (\Delta^k_n f)(x_{(k+1):n}).$$

The matrix D^k_n can be constructed recursively as the product of a diagonally-weighted first difference matrix and D^{k-1}_n; see the help file for `d_mat()`, or Section 6.1 of Tibshirani (2020). Therefore, multiplication by D^k_n or by its transpose can be performed in $O(nk)$ operations based on iterated weighted differences. See Appendix D of Tibshirani (2020) for details.

The option `tf_weighting = TRUE` performs multiplication by $W^k_n D^k_n$ where W^k_n is a $(n-k) \times (n-k)$ diagonal matrix with entries $(x_{i+k} - x_i)/k$, $i = 1, \ldots, n-k$. This weighting is implicit in trend filtering, as the penalty in the kth order trend filtering optimization problem (with optimization parameter θ) is $\|W^{k+1}_n D^{k+1}_n \theta\|_1$. Moreover, this is precisely the kth order total variation of the kth degree discrete spline interpolant f to θ, with knots in $x_{(k+1):(n-1)}$; that is, such an interpolant satisfies:

$$\mathrm{TV}(D^k f) = \|W^{k+1}_n D^{k+1}_n \theta\|_1,$$

where $D^k f$ is the kth derivative of f. See Section 9.1. of Tibshirani (2020) for more details.

Value

Product of the discrete derivative matrix D and the input vector `v`.

References

Tibshirani (2020), "Divided differences, falling factorials, and discrete splines: Another look at trend filtering and related problems", Section 6.1.

See Also

`discrete_deriv()` for discrete differentiation at arbitrary query points, `b_mat_mult()` for multiplying by the extended discrete derivative matrix, and `d_mat()` for constructing the discrete derivative matrix.

Examples

```
v = sort(runif(10))
as.vector(d_mat(2, 1:10) %*% v)
d_mat_mult(v, 2, 1:10)
```

`h_eval` *Evaluate H basis*

Description

Evaluates the falling factorial basis of a given order, with respect to given design points, at arbitrary query points.

Usage

```
h_eval(k, xd, x, col_idx = NULL)
```

Arguments

`k`	Order for the falling factorial basis. Must be >= 0.
`xd`	Design points. Must be sorted in increasing order, and have length at least `k+1`.
`x`	Query points. Must be sorted in increasing order.
`col_idx`	Vector of indices, a subset of `1:n` where `n = length(xd)`, that indicates which columns of the constructed matrix should be returned. The default is `NULL`, which is taken to mean `1:n`.

Details

The falling factorial basis functions of order k, defined with respect to design points $x_1 < \ldots < x_n$, are denoted $h^k_1, \ldots, h^k_n$. For their precise definition and further references, see the help file for `h_mat()`. The current function produces a matrix of evaluations of the falling factorial basis at an arbitrary sequence of query points. For each query point x, this matrix has a corresponding row with entries:

$$h^k_j(x), \ j = 1, \ldots, n.$$

Value

Sparse matrix of dimension `length(x)` by `length(col_idx)`.

See Also

`h_mat()` for constructing evaluations of the falling factorial basis at the design points.

Examples

```
xd = 1:10 / 10
x = 1:9 / 10 + 0.05
h_mat(2, xd)
h_eval(2, xd, x)
```

`h_mat` *Construct H matrix*

Description

Constructs the falling factorial basis matrix of a given order, with respect to given design points.

Usage

```
h_mat(k, xd, di_weighting = FALSE, col_idx = NULL)
```

Arguments

`k`	Order for the falling factorial basis matrix. Must be >= 0.
`xd`	Design points. Must be sorted in increasing order, and have length at least `k+1`.
`di_weighting`	Should "discrete integration weighting" be used? Multiplication by such a weighted H gives discrete integrals at the design points; see details for more information. The default is `FALSE`.
`col_idx`	Vector of indices, a subset of `1:n` where `n = length(xd)`, that indicates which columns of the constructed matrix should be returned. The default is `NULL`, which is taken to mean `1:n`.

Details

The falling factorial basis matrix of order k, with respect to design points $x_1 < \ldots < x_n$, is denoted H^k_n. It has dimension $n \times n$, and its entries are defined as:

$$(H^k_n)_{ij} = h^k_j(x_i),$$

where $h^k_1, \ldots, h^k_n$ are the falling factorial basis functions, defined as:

$$\begin{aligned} h^k_j(x) &= \frac{1}{(j-1)!} \prod_{\ell=1}^{j-1} (x - x_\ell), \quad j = 1, \ldots, k+1, \\ h^k_j(x) &= \frac{1}{k!} \prod_{\ell=j-k}^{j-1} (x - x_\ell) \cdot 1\{x > x_{j-1}\}, \quad j = k+2, \ldots, n. \end{aligned}$$

The matrix H^k_n can also be constructed recursively, as follows. We first define the $n \times n$ lower triangular matrix of 1s:

$$L_n = \begin{bmatrix} 1 & 0 & \ldots & 0 \\ 1 & 1 & \ldots & 0 \\ \vdots & & & \\ 1 & 1 & \ldots & 1 \end{bmatrix},$$

and for $k \geq 1$, define the $n \times n$ extended diagonal weight matrix Z_n^k to have first k diagonal entries equal to 1 and last $n-k$ diagonal entries equal to $(x_{i+k} - x_i)/k$, $i = 1, \ldots, n-k$. The kth order falling factorial basis matrix is then given by the recursion:

$$\begin{aligned} H_n^0 &= L_n, \\ H_n^k &= H_n^{k-1} Z_n^k \begin{bmatrix} I_k & 0 \\ 0 & L_{n-k} \end{bmatrix}, \quad \text{for } k \geq 1, \end{aligned}$$

where I_k denotes the $k \times k$ identity matrix, and L_{n-k} denotes the $(n-k) \times (n-k)$ lower triangular matrix of 1s. For further details about this recursive representation, see Sections 3.3 and 6.3 of Tibshirani (2020).

The option `di_weighting = TRUE` returns $H_n^k Z_n^{k+1}$ where Z_n^{k+1} is the $n \times n$ diagonal matrix as defined above. This is connected to discrete integration as explained in the help file for `h_mat_mult()`. See also Section 3.3 of Tibshirani (2020) for more details.

Each basis function h_j^k, for $j \geq k+2$, has a single knot at x_{j-1}. The falling factorial basis thus spans kth degree piecewise polynomials—discrete splines, in fact—with knots in $x_{(k+1):(n-1)}$. The dimension of this space is $n-k-1$ (number of knots) $+ k + 1$ (polynomial dimension) $= n$. Setting the argument `col_idx` appropriately allow one to form a basis matrix for a discrete spline space corresponding to an arbitrary knot set $T \subseteq x_{(k+1):(n-1)}$. For more information, see Sections 4.1 and 8 of Tibshirani (2020).

Note 1: For computing the least squares projection onto a discrete spline space defined by an arbitrary knot set $T \subseteq x_{(k+1):(n-1)}$, one should **not** use the falling factorial basis, but instead use the discrete natural spline basis from `n_mat()`, as the latter has **much** better numerical properties in general. The help file for `dspline_solve()` gives more information.

Note 2: For multiplication of a given vector by H_n^k, one should **not** form H_n^k with the current function and then carry out the multiplication, but instead use `h_mat_mult()`, as the latter will be **much** more efficient (quadratic-time versus linear-time).

Value

Sparse matrix of dimension `length(xd)` by `length(col_idx)`.

References

Tibshirani (2020), "Divided differences, falling factorials, and discrete splines: Another look at trend filtering and related problems", Section 6.3.

See Also

`h_mat_mult()` for multiplying by the falling factorial basis matrix and `h_eval()` for constructing evaluations of the falling factorial basis at arbitrary query points.

Examples

```
h_mat(2, 1:10)
h_mat(2, 1:10 / 10)
h_mat(2, 1:10, col_idx = 4:7)
```

h_mat_mult *Multiply by H matrix*

Description

Multiplies a given vector by H, the falling factorial basis matrix of a given order, with respect to given design points.

Usage

```
h_mat_mult(v, k, xd, di_weighting = FALSE, transpose = FALSE, inverse = FALSE)
```

Arguments

`v`	Vector to be multiplied by H, the falling factorial basis matrix.
`k`	Order for the falling factorial basis matrix. Must be >= 0.
`xd`	Design points. Must be sorted in increasing order, and have length at least `k+1`.
`di_weighting`	Should "discrete integration weighting" be used? Multiplication by such a weighted H gives discrete integrals at the design points; see details for more information. The default is `FALSE`.
`transpose`	Multiply by the transpose of H? The default is `FALSE`.
`inverse`	Multiply by the inverse of H? The default is `FALSE`.

Details

The falling factorial basis matrix of order k, with respect to design points $x_1 < \ldots < x_n$, is denoted H^k_n. Its entries are defined as:

$$(H^k_n)_{ij} = h^k_j(x_i),$$

where h^k_j is the jth falling factorial basis function, as defined in the help file for `h_mat()`. The matrix H^k_n can be constructed recursively as the product of H^{k-1}_n and a diagonally-weighted cumulative sum matrix; see the help file for `h_mat()`, or Section 6.3 of Tibshirani (2020). Therefore, multiplication by H^k_n or by its transpose can be performed in $O(nk)$ operations based on iterated weighted cumulative sums. See Appendix D of Tibshirani (2020) for details.

The option `di_weighting = TRUE` performs multiplication by $H^k_n Z^{k+1}_n$ where Z^{k+1}_n is an $n \times n$ diagonal matrix whose first $k+1$ diagonal entries of Z^{k+1}_n are 1 and last $n-k-1$ diagonal entries are $(x_{i+k+1} - x_i)/(k+1)$, $i = 1, \ldots, n-k-1$. The connection to discrete integration is as follows: multiplication of $v = f(x_{1:n})$ by $H^k_n Z^{k+1}_n$ gives order $k+1$ discrete integrals (note the increment in order of integration here) of f at the points $x_{1:n}$:

$$H^k_n Z^{k+1}_n v = (S^{k+1}_n f)(x_{1:n}).$$

Lastly, the matrix H^k_n has a special **inverse relationship** to the extended discrete derivative matrix B^{k+1}_n of degree $k+1$; it satisfies:

$$H^k_n Z^{k+1}_n B^{k+1}_n = I_n,$$

where Z_n^{k+1} is the $n \times n$ diagonal matrix as described above, and I_n is the $n \times n$ identity matrix. This, combined with the fact that the extended discrete derivative matrix has an efficient recursive representation in terms of weighted differences, means that multiplying by $(H_n^k)^{-1}$ or its transpose can be performed in $O(nk)$ operations. See Section 6.2 and Appendix D of Tibshirani (2020) for details.

Value

Product of falling factorial basis matrix H and the input vector `v`.

References

Tibshirani (2020), "Divided differences, falling factorials, and discrete splines: Another look at trend filtering and related problems", Section 6.2.

See Also

`discrete_integ()` for discrete integration at arbitrary query points, and `h_mat()` for constructing the falling factorial basis matrix.

Examples

```
v = sort(runif(10))
as.vector(h_mat(2, 1:10) %*% v)
h_mat_mult(v, 2, 1:10)
```

`n_eval` *Evaluate N basis*

Description

Evaluates the discrete B-spline basis of a given order, with respect to given design points, evaluated at arbitrary query points.

Usage

```
n_eval(k, xd, x, normalized = TRUE, knot_idx = NULL, N = NULL)
```

Arguments

`k`	Order for the discrete B-spline basis. Must be >= 0.
`xd`	Design points. Must be sorted in increasing order, and have length at least `k+1`.
`x`	Query points. Must be sorted in increasing order.
`normalized`	Should the discrete B-spline basis vectors be normalized to attain a maximum value of 1 over the design points? The default is `TRUE`.

knot_idx	Vector of indices, a subset of `(k+1):(n-1)` where `n = length(xd)`, that indicates which design points should be used as knot points for the discrete B-splines. Must be sorted in increasing order. The default is `NULL`, which is taken to mean `(k+1):(n-1)`.
N	Matrix of discrete B-spline evaluations at the design points. The default is `NULL`, which means that this is precomputed before constructing the matrix of discrete B-spline evaluations at the query points. If `N` is non-`NULL`, then the argument `normalized` will be ignored (as this would have only been used to construct N at the design points).

Details

The discrete B-spline basis functions of order k, defined with respect to design points $x_1 < \ldots < x_n$, are denoted $\eta_1^k, \ldots, \eta_n^k$. For a discussion of their properties and further references, see the help file for `n_mat()`. The current function produces a matrix of evaluations of the discrete B-spline basis at an arbitrary sequence of query points. For each query point x, this matrix has a corresponding row with entries:

$$\eta_j^k(x), \; j = 1, \ldots, n.$$

Unlike the falling factorial basis, the discrete B-spline basis is not generally available in closed-form. Therefore, the current function (unlike `h_eval()`) will first check if it should precompute the evaluations of the discrete B-spline basis at the design points. If the argument `N` is non-`NULL`, then it will use this as the matrix of evaluations at the design points; if `N` is `NULL`, then it will call `n_mat()` to produce such a matrix, and will pass to this function the arguments `normalized` and `knot_idx` accordingly.

After obtaining the matrix of discrete B-spline evaluations at the design points, the fast interpolation scheme from `dspline_interp()` is used to produce evaluations at the query points.

Value

Sparse matrix of dimension `length(x)` by `length(knot_idx) + k + 1`.

See Also

`n_mat()` for constructing evaluations of the discrete B-spline basis at the design points.

Examples

```
xd = 1:10 / 10
x = 1:9 / 10 + 0.05
n_mat(2, xd, knot_idx = c(3, 5, 7))
n_eval(2, xd, x, knot_idx = c(3, 5, 7))
```

n_mat *Construct N matrix*

Description

Constructs the discrete B-spline basis matrix of a given order, with respect to given design points and given knot points.

Usage

```
n_mat(k, xd, normalized = TRUE, knot_idx = NULL)
```

Arguments

`k`	Order for the discrete B-spline basis matrix. Must be >= 0.
`xd`	Design points. Must be sorted in increasing order, and have length at least `k+1`.
`normalized`	Should the discrete B-spline basis vectors be normalized to attain a maximum value of 1 over the design points? The default is `TRUE`.
`knot_idx`	Vector of indices, a subset of `(k+1):(n-1)` where `n = length(xd)`, that indicates which design points should be used as knot points for the discrete B-splines. Must be sorted in increasing order. The default is `NULL`, which is taken to mean `(k+1):(n-1)` as in the "canonical" discrete spline space (though in this case the returned N matrix will be trivial: it will be the identity matrix). See details.

Details

The discrete B-spline basis matrix of order k, with respect to design points $x_1 < \ldots < x_n$, and knot set $T \subseteq x_{(k+1):(n-1)}$ is denoted N^k_T. It has dimension $(|T| + k + 1) \times n$, and its entries are given by:

$$(N^k_T)_{ij} = \eta^k_j(x_i),$$

where $\eta^k_1, \ldots, \eta^k_m$ are discrete B-spline (DB-spline) basis functions and $m = |T| + k + 1$. As is suggested by their name, the DB-spline functions are linearly independent and span the space of discrete splines with knots at T. Each DB-spline η^k_j has a key local support property: it is supported on an interval containing at most $k + 2$ adjacent knots.

The functions $\eta^k_1, \ldots, \eta^k_m$ are, in general, not available in closed-form, and are defined by setting up and solving a sequence of locally-defined linear systems. For any knot set T, computation of the evaluations of all DB-splines at the design points can be done in $O(nk^2)$ operations; see Sections 7, 8.2, and 8.3 of Tibshirani (2020) for details. The current function uses a sparse QR decomposition from the `Eigen::SparseQR` module in C++ in order to solve the local linear systems.

When $T = x_{(k+1):(n-1)}$, the knot set corresponding to the "canonical" discrete spline space (spanned by the falling factorial basis functions $h^k_1, \ldots, h^k_n$ whose evaluations make up H^k_n; see the help file for `h_mat()`), the DB-spline basis matrix, which we denote by N^k_n, is trivial: it equals the $n \times n$ identity matrix, $N^k_n = I_n$. Therefore DB-splines are really only useful for knot sets T that are proper subsets of $x_{(k+1):(n-1)}$. Specification of the knot set T is done via the argument `knot_idx`.

Value

Sparse matrix of dimension `length(xd)` by `length(knot_idx) + k + 1`.

References

Tibshirani (2020), "Divided differences, falling factorials, and discrete splines: Another look at trend filtering and related problems", Sections 7, 8.2, and 8.3.

See Also

`h_eval()` for constructing evaluations of the discrete B-spline basis at arbitrary query points.

Examples

```
n_mat(2, 1:10, knot_idx = c(3, 5, 7))
n_mat(2, 1:10, knot_idx = c(4, 6, 8))
```

Index