Package 'kubik'

March 23, 2020

Title Cubic Hermite Splines and Related Optimization Methods

Version 0.2.0

Date 2020-03-23

License GPL (>= 2)

Maintainer Abby Spurdle <spurdle.a@gmail.com>

Author Abby Spurdle

URL https://sites.google.com/site/spurdlea/r

Description

Constructs, plots and evaluates constrained cubic Hermite splines, which can be used to construct monotonic and bounded splines. Computes their first derivatives, indefinite integrals and smooth approximations of their first, second and higher derivatives. Also, computes their roots/solutions, along with their minima/maxima and inflection points.

Suggests intoo, vectools

NeedsCompilation no

Repository CRAN

Date/Publication 2020-03-23 07:20:11 UTC

R topics documented:

11_cubic_hermite_splines *Cubic Hermite Splines*

Description

Functions for constructing cubic Hermite splines (as function objects), and their derivatives and indefinite integrals.

Usage

```
chs (cx, cy, cb, ..., constraints = chs.constraints (, ...),
    transform=FALSE, outside = c (NA, NA), init.method)

chs.derivative (cx, cy, cb, ..., constraints = chs.constraints (, ...),
    transform=FALSE, outside = c (NA, NA), init.method)

chs.integral (cx, cy, cb, ..., constraints = chs.constraints (, ...),
    transform=FALSE, outside = c (NA, NA), init.method,
    constant=0)

approx.chs.derivative (cx, cy, cb, ...,
    constraints = chs.constraints (, ...),
    transform=FALSE, outside = c (NA, NA), init.method,
    apply.constraints.to=0, nth=1, trim=TRUE)
```

Arguments

cx	Vector of unique ascending control point x values. (Preferably, equally-spaced).
cy	Vector of control point y values.
cb	Vector of control slopes. Optional, unless transform is true.
constraints	A chs.constraints object. Can be NULL or NA, for no constraints. By default (if transform is false), ignored if cb supplied. (Except for approximate derivatives). Refer to the help file for chs.constraints for more information.
transform	Logical value, if true, transform cb (the slopes), subject to constraints.
outside	A vector of length two, giving the value of the spline outside the control points.
init.method	Initialization method for slopes. Possible values are "SL" and "SQ". Ignored if cb supplied. (Except for approximate derivatives).

Refer to the help file for chs.slopes for more information.

`constant` Constant term.

`apply.constraints.to`
Integer, between 0 and nth, which derivative to apply constraints to.

`nth` Which derivative to approximate.

`trim` Logical value, if true, remove nth control points, from both the beginning and end of the spline.
For example, if nth=2, two control points are removed from each end.

This should be true (the default), if non-default constraints are used.

`...` Alternative (possibly simpler) way of specifying constraints.
Ignored if a chs.constraints object supplied.

This (specifying constraints in dots) should not be used inside packages, because the use of dots may change.

Details

These functions are constructors for function objects representing cubic Hermite splines, and their derivatives and indefinite integrals.

The resulting function objects can be printed, plotted and evaluated (for x).
Alternatively, you can use the spline eval functions (with a .eval suffix), without using function objects, which may be more efficient, in some cases.

chs
Construct a chs object, for cubic Hermite splines.

chs.derivative
Construct a chs.derivative object, for (exact) first derivatives of cubic Hermite splines.

chs.integral
Construct a chs.integral object, for indefinite integrals of cubic Hermite splines.

approx.chs.derivative
Construct an approx.chs.derivative object (which is also a chs object), for (new) cubic Hermite splines representing smooth approximations of first, second or higher derivatives of (initial) cubic Hermite splines.

Each of these objects requires two or more control points, and optionally their control slopes.

Note that currently, some functions may fail to work, for very small or very large input values.

Expanding on the constraints and transform arguments, there are three valid cases:
(1) The slopes are omitted, in which case, default slopes are used subject to any constraints.
(2) Slopes are provided and transform=FALSE (the default), in which case, user-supplied slopes are used.
(3) Slopes are provided and transform=TRUE, in which case, the user-supplied slopes are transformed subject to any constraints, and the transformed slopes are used.

The approximate derivative is computed by constructing an initial cubic Hermite spline, then for each derivative, the cy values are set equal to the slopes, and new slopes are computed. This is unlikely to give a good result unless a large number of equally-spaced control points are used.

Also note that:
(1) Exact derivatives aren't necessarily smooth.
(2) In the chs.derivative and chs.integral objects, constraints and slopes apply to the original function, not the derivatives or integrals of that function.
(3) In contrast, the outside values apply to the resulting derivatives or integrals.
(4) If it's not possible to compute slopes that satisfy constraints, an error is generated.
(5) In approximate derivatives, user-supplied constraints only apply to one iteration (determined by apply.constraints.to), with default constraints used for other iterations.
(6) Approximate derivatives may give poor approximations in the tail regions, and applying non-default constraints to approximate derivatives may be problematic if tail regions are included.
(7) Currently, constraints don't apply to outside values, however, this may be changed in the future.
(8) If the control slopes are omitted, then they're computed using the same method as chs.slopes.

Refer to the help pages for chs.constraints and chs.slopes for more information on constraints and slopes.

Note that the help page on constraints, also discusses the relationship between the spline's shape, and its slopes. And the help page on slopes also discusses the initialization method.

Value

Self-referencing functions of the form:

function (x) {...}

Where x is a numeric vector.

References

Help pages from the stats package, for the splinefun, splinefunH and spline functions.

Wikipedia pages "Cubic Hermite spline" and "Monotone cubic interpolation".

Fritsch, F.N. & Carlson, R.E. (1980). Monotone Piecewise Cubic Interpolation. SIAM Journal on Numerical Analysis, 17 (2). doi:10.1137/0717021.

See Also

chs.constraints, apply.chs.constraints, chs.slopes, chs.eval

Examples

```
############
#example (1)
############
#control points
cx <- 1:4
cy <- c (-4, -1, 1, 4)

#cubic hermite spline
#(with function object, and default slopes)
f <- chs (cx, cy)

#plot
#(with control points)
```

```
plot (f, control.points=TRUE)

#evaluate
f (3.5)

#add point to plot
points (3.5, f (3.5), pch=16, cex=2, col="blue")

############
#example (2)
############
#control points
#(sine wave)
cx <- seq (-2 * pi, 2 * pi, length.out=200)
cy <- sin (cx)

#cubic hermite spline, and approximate 4th derivative
f0 <- chs (cx, cy)
f4 <- approx.chs.derivative (cx, cy, nth=4)

#plot
#(approx 4th derivative in blue)
plot (f0)
lines (f4, col="blue")

#evaluate
#(results should be close to one)
f0 (pi / 2)
f4 (pi / 2)
```

12_constraints *Constraints*

Description

Constraints for cubic Hermite splines.

Usage

```
chs.constraints (correction=TRUE, ...,
    bounds,
    increasing=FALSE, decreasing=FALSE)
```

Arguments

correction	Logical value, if true, apply the main correction method.
bounds	Optional length-two vector giving bound-constraints. They need to be unique ascending values. And they give the lowest and highest values that the spline can take.

	Assuming that one of the two values is finite, setting the lower bound-constraint to -Inf or the upper bound-constraint to +Inf, results in an upper bounded spline or a lower bounded spline, respectively.
`increasing`	Logical value, if true, a non-decreasing function.
`decreasing`	Logical value, if true, a non-increasing function.
`...`	Ignored.

Details

A chs.constraints object defines constraints for cubic Hermite splines.

Such constraints, only apply to computation of control slopes.
Not the control points.

So, the control point positions need to be suitable.
(i.e. You can't apply an increasing constraint to decreasing control points).

The main correction method is derived from the Fritsch-Carlson algorithm, and prevents local optima from appearing within each spline segment, except where either of the following applies:
(1) The sign of the secant's slope is zero, and the surrounding control slopes have opposite signs.
(2) The secant's slope has the opposite sign to either of the surrounding control slopes.

Where the secant refers to the secant line between the surrounding control points.

In general, if the control points are equally-spaced (in x), are ascending or descending (in y), and the default slopes are used, then the main correction method should be sufficient to construct a monotonically increasing or decreasing function. Even so, I recommend you apply increasing or decreasing constraints, if you need an increasing or decreasing function.

Currently, both monotonicity and bound constraints, will set some slopes to zero, if the initial slopes cause the spline to break the constraints. This is not an optimal transformation, and hopefully it will be improved in the future.

Note that (re: bounded splines) this package allows control points to be exactly equal to bound-constraints, and I can not guarantee that very small floating point errors won't occur, when evaluating such splines.

If very small floating point errors are problematic, then users can either:
(1) Ensure there's a relatively small margin between the control points and the bound-constraints.
(2) Apply a correction method after evaluating splines.

Note that signs can be zero, hence not having the same sign, doesn't necessarily mean having opposite signs.

Also note that, given a single spline segment:
(1) If the surrounding control slopes have the same sign, a pair of local optima may or may not be present.
(2) If the surrounding control slopes have opposite signs, a single local optimum should be present.
(3) If the surrounding control tangent lines are on the same side of the secant line, a single local inflection point may or may not be present.
(4) If the surrounding control tangent lines are on opposite sides of the secant line, a single local inflection point should be present.

Value

A chs.constraints object.

References

Please refer to the help page for chs for background information and references.

See Also

chs, apply.chs.constraints, chs.slopes

Examples

```
#constraints
#f (x) bounded within [0, 1]
constraints <- chs.constraints (bounds = c (0, 1) )

#control points
#(control points close to bound-constraints)
cx <- 1:6
cy <- c (0.001, 0.999, 0.999, 0.001, 0.001, 0.999)

#default and bounded splines
f.def <- chs (cx, cy)
f.bnd <- chs (cx, cy, constraints=constraints)

#plot
#(bounded spline in blue)
plot (f.def)
abline (h = c (0, 1), lty=2, col="grey")
lines (f.bnd, col="blue")
```

13_print_and_plot_methods

Print and Plot Methods

Description

Print and plot methods, for chs-related objects.

Usage

```
## S3 method for class 'kspline'
print(x, ...)

## S3 method for class 'kspline'
plot(x, ..., control.points=FALSE)
## S3 method for class 'kspline'
lines(x, ..., control.points=FALSE)
## S3 method for class 'kspline'
points(x, ...)
```

Arguments

x	A kspline object, which is the superclass of chs, chs.derivative and chs.integral objects.
control.points	If true, plot the control points (for chs objects), or their corresponding points (for chs.derivative and chs.integral objects).
...	Other arguments.

Details

The print method calls intoo::object.summary, if the intoo package is on the search path.

Otherwise, it calls the default print method.

References

Please refer to the help page for chs for background information and references.

See Also

chs

Examples

```
#control points
cx <- 1:4
cy <- c (-4, -1, 1, 4)

#cubic hermite spline
#(with function object, and default slopes)
f <- chs (cx, cy)

#plot the object
#(with control points)
plot (f, control.points=TRUE)
```

14_roots *Roots*

Description

Compute roots of cubic Hermite splines, using function objects.

Usage

```
root.chs (f, ..., include.implied.roots=TRUE, warning=TRUE)
argmin.chs (f, ..., include.implied.roots=TRUE, warning=TRUE)
argmax.chs (f, ..., include.implied.roots=TRUE, warning=TRUE)
argflex.chs (f, ..., include.implied.roots=TRUE, warning=TRUE)

roots.chs (f, ..., include.implied.roots=TRUE, warning=TRUE)
argmins.chs (f, ..., include.implied.roots=TRUE, warning=TRUE)
argmaxs.chs (f, ..., include.implied.roots=TRUE, warning=TRUE)
argflexs.chs (f, ..., include.implied.roots=TRUE, warning=TRUE)

chs.roots.derivative (f, ..., include.implied.roots=TRUE, warning=TRUE)
```

Arguments

f	A chs (or approx.chs.derivative) object.
include.implied.roots	If true, include implied roots.
warning	If true, give a warning if there's relevant constant/level (or linear) sections.
...	Ignored.

Details

These functions compute roots of cubic Hermite splines, using function objects.
Alternatively, you can you can use the spline root eval functions (with a .eval suffix), without using function objects, which may be more efficient, in some cases.

By default, the functions include implied roots (defined below) and remove spurious inflection points (also, defined below).

Here, an implied root refers to a root that coincides with a constant/level section.
(Or coincides with a linear section, for inflection points).

Here, a constant/level section refers to one or more consecutive spline segments, with a constant value.

Here, a spurious inflection point refers to an inflection point within a quadratic-suggestive section.

And here, a quadratic-suggestive section refers to one or more consecutive spline segments, where all the control tangents within that section, are on the same side of the secant lines between consecutive control points.

If implied roots are included, then their value is equal to the midpoint of their combined interval.

Notes:
(1) Optima and inflection points exclude the endpoints.
(2) Currently, these functions may fail to find roots, if they're extremely close to the control points. Except, the functions should find optima in well-defined splines (with no more than one optimum per spline segment), regardless of where the optima are located.
(3) The chs.roots.derivative function is likely to be deprecated in the future, and replaced by a better function.

Value

root.chs returns a single real root, if a single real root exists, otherwise, it returns an error.
roots.chs returns all the real roots, which will be a zero-length numeric vector, if there are no real roots.

argmin.chs returns a single global argmin, if a single global argmin exists, otherwise, it returns an error.
argmins.chs returns all argmins, which will be a zero-length numeric vector, if there are no minima.

argmax.chs and argmaxs.chs are the same as argmin.chs and argmins.chs, except they're for argmax(s) rather than argmin(s).
argflex.chs() and argflexs.chs are similar to the functions above, except that they're for inflection points.

Unlike the other functions here, chs.roots.derivative returns a two-column matrix.
The first column gives the roots of the derivative.
And the second column gives a numeric value equal to the sign of the second derivative, so:
1 (for minima), -1 (for maxima) and 0 (for stationary inflection points).

References

Please refer to the help page for chs for background information and references.

See Also

chs, solve.chs, roots.chs.eval

Examples

```
#control points
cx <- 1:4
cy <- c (1, -1, -1, 1)

#cubic hermite spline
#(with function object, and default slopes)
f <- chs (cx, cy)

#roots
roots.chs (f)
argmins.chs (f)
```

`15_solve_method` *Solve Method*

Description

Solve method, for chs objects.

Usage

```
## S3 method for class 'chs'
solve(a, b, ..., to.list=FALSE)
```

Arguments

a	A chs object.
b	A vector of y values.
to.list	If true, return list. Ignored, if b has length two or greater.
...	Other arguments, for the roots.chs.eval function.

Details

This function is a wrapper for roots.chs.eval.
(It calls roots.chs.eval with modified cy values).

It computes x values, where the spline's y value is equal to the values in b.

Value

If to.list is false (the default), and length (b) is one:
It returns a single numeric vector.

Otherwise:
It returns a list of numeric vectors, one for each value in b.

References

Please refer to the help page for chs for background information and references.

See Also

chs, roots.chs

Examples

```
#control points
cx <- 1:4
cy <- c (-4, -1, 1, 4)

#cubic hermite spline
#(with function object, and default slopes)
f <- chs (cx, cy)

#solve for x, given y=2
solve (f, 2)
```

16_slopes_and_tangents *Slopes and Tangents*

Description

Functions for computing (possibly) suitable slopes and tangents.

Usage

```
chs.slopes (cx, cy, ..., constraints = chs.constraints (, ...),
    init.method)
chs.tangents (cx, cy, ..., constraints = chs.constraints (, ...),
    init.method)

apply.chs.constraints (cx, cy, cb, ...,
    constraints = chs.constraints (, ...) )
```

Arguments

cx	Vector of unique ascending control point x values. (Preferably, equally-spaced).
cy	Vector of control point y values.
cb	Vector of control slopes.
constraints	A chs.constraints object. Can be NULL or NA, for no constraints. Refer to the help file for chs.constraints for more information.
init.method	Initialization method for slopes. Possible values are "SL" and "SQ".
...	Alternative (possibly simpler) way of specifying constraints. Ignored if a chs.constraints object supplied. This (specifying constraints in dots) should not be used inside packages, because the use of dots may change.

Details

These functions work by computing a initial vector of slopes, and then trying to transform them to satisfy constraints.

Currently, there are two methods that may be used to initialize the slopes:
SL, Simple secant-based method with near-linear tails.
SQ, Modified secant based-method with quadratic tails.

If the initialization method isn't specified by the user, it defaults to "SQ", if the control points are equally-spaced (or near-equally-spaced), and it defaults to "SL", otherwise.

If there's only one spline segment:
(Regardless of of what initialization method is used).
The initial slopes are set equal to the secant's slope.

Where the secant refers to the secant line between the surrounding control points.

And if there's two more spline segments:
The second slope is computed from the (double-interval) secant line between the first and third control points, the third slope is computed from the (double-interval) secant line between the second and fourth control points, and so on, up to the second to last slope, which is computed from the (double-interval) secant line between the third-to-last and last control points.

For the SL method, the the outermost slopes are set equal to the adjacent secants' slopes, resulting in near-linear tails. For the SQ method, the first slope is computed by fitting a quadratic polynomial to the first three control points and evaluating the derivative at the first point, and the last slope is computed by fitting a quadratic polynomial to the last three control points and evaluating the derivative at the last point.

The SQ method should only be used for equally-spaced control points.

Note that while secants may sound linear, computing slopes from secants gives similar results to computing all the slopes via quadratic polynomials.
(And precedence is given to the SQ method, because the level of resulting curvature, is relatively consistent over the entire spline, including the tails).

The apply.chs.constraints function tries to transform a vector of slopes to satisfy a set of constraints.

If a transformation isn't possible, an error is generated.
(This applies to all three functions).

Note that it's possible to use chs.slopes to compute slopes, and then modify them as desired.

Value

chs.slopes returns a (possibly) suitable vector of slopes.

chs.tangents returns a (possibly) suitable two-column matrix, giving the intercepts and slopes.

appl.chs.constraints returns a transformed vector of slopes.

References

Please refer to the help page for chs for background information and references.

See Also

chs, chs.constraints

Examples

```
#control points
cx <- 1:4
cy <- c (-4, -1, 1, 4)

#control slopes
chs.slopes (cx, cy)
```

```
#control tangents
#(intercepts and slopes)
chs.tangents (cx, cy)
```

21_direct_chs_evaluation
Direct CHS Evaluation

Description

Functions for evaluating cubic Hermite splines, and their derivatives and indefinite integrals, directly.

Usage

```
chs.eval (cx, cy, cb, x, ...,
    outside = c (NA, NA) )

chs.derivative.eval (cx, cy, cb, x, ...,
    outside = c (NA, NA) )

chs.integral.eval (cx, cy, cb, x, ...,
    outside = c (NA, NA), constant=0)
```

Arguments

cx	Vector of unique ascending control point x values. (Preferably, equally-spaced).
cy	Vector of control point y values.
cb	Vector of control slopes.
x	Vector of x values, where the spline is evaluated at.
outside	A vector of length two, giving the value of the spline outside the control points.
constant	Constant term.
...	.

Details

Refer to the help page for chs, for more information.
(The functions described in that help page are similar to these functions).

These functions (with a .eval suffix) evaluate cubic Hermite splines, and their derivatives and indefinite integrals, without using function objects, and with minimal error checking.
Alternatively, you can use function objects, which are likely to be more convenient, in most cases.

chs.eval
Evaluate cubic Hermite splines.

chs.deritivative.eval
Evaluate (exact) derivatives of cubic Hermite splines.

chs.integral.eval
Evaluate indefinite integrals of cubic Hermite splines.

References

Please refer to the help page for chs for background information and references.

See Also

chs

Examples

```
#control points
cx <- 1:4
cy <- c (-4, -1, 1, 4)

#control slopes
cb <- chs.slopes (cx, cy)

#evaluate
#(without function object)
chs.eval (cx, cy, cb, 3.5)
```

22_direct_chs_root_evaluation

Direct CHS Root Evaluation

Description

Compute roots of cubic Hermite splines, directly.

Usage

```
root.chs.eval (cx, cy, cb, ...,
    include.implied.roots=TRUE, warning=TRUE)
argmin.chs.eval (cx, cy, cb, ...,
    include.implied.roots=TRUE, warning=TRUE)
argmax.chs.eval (cx, cy, cb, ...,
    include.implied.roots=TRUE, warning=TRUE)
argflex.chs.eval (cx, cy, cb, ...,
    include.implied.roots=TRUE, warning=TRUE)

roots.chs.eval (cx, cy, cb, ...,
    include.implied.roots=TRUE, warning=TRUE)
argmins.chs.eval (cx, cy, cb, ...,
```

```
    include.implied.roots=TRUE, warning=TRUE)
argmaxs.chs.eval (cx, cy, cb, ...,
    include.implied.roots=TRUE, warning=TRUE)
argflexs.chs.eval (cx, cy, cb, ...,
    include.implied.roots=TRUE, warning=TRUE)

chs.roots.derivative.eval (cx, cy, cb, ...,
    include.implied.roots=TRUE, warning=TRUE)
```

Arguments

cx	Vector of unique ascending control point x values. (Preferably, equally-spaced).
cy	Vector of control point y values.
cb	Vector of control slopes.
include.implied.roots	If true, include implied roots.
warning	If true, give a warning if there's relevant constant/level (or linear) sections.
...	Ignored.

Details

Refer to the help page for roots.chs, for more information.
(The functions described in that help page are similar to these functions).

These functions (with a .eval suffix) compute roots of cubic Hermite splines, without using function objects, and with minimal error checking.
Alternatively, you can use function objects, which are likely to be more convenient, in most cases.

References

Please refer to the help page for chs for background information and references.

See Also

chs, roots.chs

Examples

```
#control points
cx <- 1:4
cy <- c (1, -1, -1, 1)

#control slopes
cb <- chs.slopes (cx, cy)

#roots
#(without function object)
roots.chs.eval (cx, cy, cb)
argmins.chs.eval (cx, cy, cb)
```

31_deprecated *Deprecated Functions*

Description

Deprecated functions, please do not use.

Usage

```
chs.argmins (...)
chs.argmaxs (...)
```

Arguments

...	.

Index