Package 'knotR'

May 26, 2017

Type Package

Title Knot Diagrams using Bezier Curves

Version 1.0-2

Date 2017-05-23

Author Robin K. S. Hankin

Depends R (>= 2.10)

Maintainer Robin K. S. Hankin <hankin.robin@gmail.com>

LazyData TRUE

Description Makes nice pictures of knots using Bezier curves and numerical optimization.

License GPL-2

NeedsCompilation no

Repository CRAN

Date/Publication 2017-05-26 06:07:29 UTC

R topics documented:

`knotR-package` *The knotR package*

Description

This page gives an overview of the package and pointers to more detailed documentation. See the vignette for a more complete exposition.

Details

The basic workflow is to create an `.svg` file in inkscape comprising a single closed path (that is, the first and last node are the same point). Control nodes should all be symmetrical. Many examples of correctly formatted `.svg` files are given in the `inst/` directory.

The best way to reproduce a knot from an image of its projection is to fire up inkscape, then *import* the image into inkscape, resize and rotate as desired, then follow the string with the 'Bezier curves and straight lines' tool (also called the 'pen tool' by Kirsanov; the keyboard shortcut is shift-F6). Use 1-2 nodes per segment, or 3 nodes for longer or more visually prominent segments.

Keep the lines straight at first. Close the path by making a final click on the initial node; now you have a closed polygon, which will self-intersect at the knot's crossing points. To smoothen the path, select the 'edit paths by node' tool (shift-F2), then convert the corner nodes of the path to symmetric Bezier nodes ('make selected nodes symmetric'). You can then tweak the path by moving the control nodes about with the mouse. Be aware that adding or deleting nodes changes the adjacent nodes to asymmetrical Bezier control points; make them symmetric by selecting all nodes ('Ctrl-A'), then hit the 'make selected nodes symmetric' button. Do this frequently to avoid confusion.

An `.svg` file may be imported into R using the package's `reader()` function, which creates an `inkscape` object. This represents the path of the knot: it does not include over and under information. The package assumes that inkscape uses absolute coordinates (as opposed to relative coordinates); see `reader.Rd` for more information.

A path may be specified using one of three file formats which have different uses: `inkscape`, `minobj`, `controlpoints`, and `knotvec`. Objects may be converted from one form to another by using functions such as `make_minobj_from_ink()`, documented at `?utilities`.

A knot requires information on which strands pass over or under which other strands; full documentation at `?overunder`.

Knots sometimes have symmetry constraints such as horizontal or vertical symmetry, or rotational symmetry. Symmetry is imposed by using the `symmetrize()` function: this takes a knot path (coereced to `minobj` form) and a symmetry object. Symmetry objects are created with function `symmetry_object()`, which takes a knot path and a series of matrices and vectors that specify the symmetry of the knot.

Author(s)

Robin K. S. Hankin

Examples

```
a <- reader(system.file("7_6.svg",package="knotR"))
knotplot2(a)  # shows curvature

# Now use text=TRUE to display strand numbers so you can figure out the
# overunder relations:

knotplot2(a,text=TRUE,lwd=1)

ou76 <- matrix(c(
    12,01,
    02,11,
    07,03,
    04,15,
    16,06,
    14,08,
    10,13
    ),byrow=TRUE,ncol=2)

# Now we can do a proper knot plot:

knotplot(a,ou76)


# To symmetrize a knot we use the symmetry functionality of the knot:

a <- reader(system.file("3_1_not_symmetric.svg",package="knotR"))

knotplot2(a,seg=TRUE,text=TRUE,lwd=1,node=TRUE)

# First specify the vertical symmetry:

Mver <- matrix(c(
    08,10,
    07,11,
    02,04,
    01,05,
    12,06
    ),ncol=2,byrow=TRUE)

# Then the rotational symmetry:
Mrot <- matrix(c(
    09,05,01,
    10,06,02,
    08,04,12
    ),byrow=TRUE,ncol=3)
```

```
# Now the overunder information:
ou31 <- matrix(c(
    03,08,
    11,04,
    07,12
    ),byrow=TRUE,ncol=2)


# create a symmetry object:

sym31 <- symmetry_object(a, Mver=Mver,xver=c(9,3),Mrot=Mrot)


knotplot(symmetrize(a,sym31),ou31)

# Symmetric-- but ugly as a burglar's bulldog.

# to beautify, either use the knotoptim() function, or do it by hand:


objective <- function(m) {badness(make_minobj_from_minsymvec(m, sym31))}
startval  <- make_minsymvec_from_minobj(as.minobj(a),sym31)

## Not run:
# nlm() is the best optimization method, I think.  Limit to 1 iteration:
o <- nlm(f=objective, p=startval, iterlim=1)

# extract the evaluate:
oo <- make_minobj_from_minsymvec(o$estimate, sym31)

# create a knot:
k31_marginally_better <-
knot(x = oo, overunderobj = ou31, symobj = sym31)

# then plot it:
knotplot(k31_marginally_better)


## End(Not run)
```

---

as *Conversions between various forms of a knot*

Description

Conversions between various forms of a knot.

Usage

```
as.knotvec(x)
as.minobj(x)
as.inkscape(x)
as.controlpoints(x)
as.minsymvec(x,symobj)
```

Arguments

x	Object to be converted
symobj	A symmetry object

Details

The `as.foo()` functions are meant to be user-friendly; they use low-level functions like `make_knotvec_from_minobj()` (all of which are documented at `utilities.Rd`), which are a bit messy.

Author(s)

Robin K. S. Hankin

See Also

utilities

Examples

```
as.minobj(k6_2)

x <- reader(system.file("6_3.svg",package="knotR"))  # x is class inkscape

as.minsymvec(x,symmetry_object(k6_3)) # as.minsymvec() needs a symmetry object

as.controlpoints(x)

as.knotvec(x)
```

---

badness *Badness of knots*

Description

Various functions that calculate different aspects of the badness of a knot, generally with low values representing pleasing visual representations

Usage

```
badness(b, cpb, weights, prob=0, give=FALSE)
curvature_switching_badness(b)
curvature_consecutive_segment_switching_badness(b, ...)
midpoint_badness(b,cpb)
node_crossing_badness(b,cpb)
total_string_length(b)
total_bending_energy(b,power=2)
total_crossing_potential_energy(b,cpb)
metrics(b,cpb)
always_left_badness(b)
non_crossing_strand_close_approach_badness(b,cpb)
```

Arguments

b	A description of a knot, coerced to a controlpoints object
cpb	Optional argument containing information on crossing points; expensive to calculate, so providing this makes the code run faster
prob	In function badness(), the probability of plotting a knotplot. Nonzero values are useful when optimizing a knot, to keep tabs on the process
give	In function badness(), Boolean with default FALSE meaning to return the sum of the badnesses, and TRUE meaning to return them separately
power	Function total_bending_energy() returns the arc integral of R^{-p}; defaults to 2
weights	A vector of weights specifying the relative importance of the various badness measures
...	In function curvature_consecutive_segment_switching_badness(), extra arguments passed to integrate()

Details

Various functions that calculate different aspects of the badness of a knot, generally with low values representing pleasing visual representations:

- Function badness() returns the sum of the eight individual badnesses.
- Function curvature_switching_badness() provides a penalty for segments with curvatures that switch sign. The magnitude of the penalty is zero if the curvature is of one sign, otherwise proportional to the square of the minimum of the maximum value of the absolute value of the positive and negative curvatures. The source code is easier to look at, honest.
- Function curvature_consecutive_segment_switching_badness() penalizes knots with consecutive segments that switch curvature from positive to negative.
- Function midpoint_badness() penalizes knots with crossing points far from the midpoint of segments.
- Function node_crossing_badness() penalizes knots with nodes too close together (compare function total_crossing_potential_energy()).

- Function `total_string_length()` returns ℓ, the total string length. The badness is proportional to $(\ell - 5000)^2$. A length of 5000 corresponds to knots that look about right on a sheet of A4 paper.
- Function `total_bending_energy()` gives the total bending energy, effectively the arc integral of the reciprocal of the square of the radius of curvature.
- Function `total_crossing_potential_energy()` gives the potential energy of the nodes, under an inverse square force law.
- Function `always_left_badness()` penalizes knots that are *supposed* to curve to the left all the time (eg knot 8_{18}). The penalty is proportional to the greatest rightward curvature over the whole knot.
- Function `metrics()` gives all these.

Value

Returns a scalar badness

Author(s)

Robin K. S. Hankin

Examples

```
# use the k_infinity knot for speed:

system.time(badness(k_infinity))

cc <- crossing_points(k_infinity)

system.time(badness(k_infinity,cc))


metrics(k_infinity,cc)
```

`bezier` *Various functionality for Bezier curves*

Description

Various functionality for Bezier curves including derivatives and radius of curvature.

Usage

```
bezier(P, tee, n=100)
bezier_deriv(P, tee, n=100)
bezier_deriv2(P, tee, n=100)
bezier_radius(P, tee, n=100)
bezier_curvature(P,tee,n=100)
myseg(P, ...)
```

Arguments

P	Control points in the form of a 4 by 2 matrix with rows corresponding to P_0 to P_3
tee	Parametric variable t
n	Integer specifying number of points between 0 and 1 to use. Default value of 100 looks OK
...	Further arguments passed by `myseg()` to `points()`

Details

- Function `bezier()` returns a two column matrix with rows corresponding to the positions of the specified Bezier curve.
- Functions `bezier_deriv()` and `bezier_deriv2()` give the first and second derivatives respectively.
- Function `bezier_radius()` gives the radius of curvature.
- Functions `bezier_length()` and `bezier_bending_energy()` use numerical quadrature to give the arc length and bending energy ($\int R^{-1} ds$).

Author(s)

Robin K. S. Hankin

See Also

bezier_angle

Examples

```
P <- matrix(c(0, 1, 2, 2, 2, 0, 3, 2),4,2)
xy <- bezier(P,n=100)
dx <- bezier_deriv(P,n=100)

plot(xy,asp=1)
myseg(P)

plot(xy,asp=1,cex=sqrt(rowSums(dx^2))/3.2)
```

```
plot(xy,asp=1)
segments(xy[,1],xy[,2],(xy+dx/200)[,1],(xy+dx/200)[,2])


plot(xy, asp=1,cex=bezier_radius(P,n=100)/2)

lapply(as.controlpoints(k8_9),bezier_radius)
lapply(as.controlpoints(k8_9),bezier_arclength)
```

`bezier_angle` *Intersection of two Bezier curves*

Description

Description of the intersection of two Bezier curves including position and angle of the point of intersection.

Usage

```
bezier_angle(P1, P2)
bezier_intersect(P1,P2, type='pos', ...)
```

Arguments

`P1,P2`	Control points for two Bezier curves as per `bezier()`
`type`	In function `bezier_intersect()`, string argument governing what exactly is to be returned; see details.
`...`	In function `bezier_intersect()`, further arguments passed to `constOptim()`

Details

Function `bezier_intersect()` uses `constOptim()` to find the point of closest approach.

Function `bezier_angle()` returns the square of the cosine of the intersection angle (so strands crossing at right angles return zero). If the strands do not intersect, then return 1. This is needed because sometimes, strands which intersect are perturbed by the optimization routine so that they are disjoint.

Note

If the curves intersect in more than one point, the behaviour of these routines is not defined.

Author(s)

Robin K. S. Hankin

See Also

bezier

Examples

```
P1 <- matrix(c(1, 3, 6, 4, 7, 3, 2, 2),ncol=2)
P2 <- matrix(c(4, 5, 5, 3, 7, 2, 5, 1),ncol=2)

x1 <- bezier(P1,n=100)
x2 <- bezier(P2,n=100)

plot(x1,asp=1,xlim=c(0,8),ylim=c(0,8))
points(x2)

myseg(P1)
myseg(P2)

jj <- bezier_intersect(P1,P2)
points(x=jj[1],y=jj[2],pch=16,cex=3,col='blue')

# looks close to orthogonal, actually 82 degrees:
acos(sqrt(bezier_angle(P1,P2)))*180/pi
```

---

bezier_find_length *Solve for arclength*

Description

Finds the value of the Bezier parameter t that corresponds to a given arclength from the start of a Bezier curve

Usage

```
bezier_find_length(P, len, from = 0, increasing = TRUE, give = FALSE, ...)
```

Arguments

P	Control points in the form of a 4 by 2 matrix with rows corresponding to P_0 to P_3
from	Point from which to start measuring arc length
len	Arc length
increasing	Boolean, with default TRUE meaning to measure length towards the end, andFALSE meaning to measure in the opposite direction

give	Boolean, with TRUE meaning to return details from uniroot() and default FALSE meaning to give just the position on the curve
...	Further arguments passed to uniroot()

Details

The function just uses uniroot() to find the appropriate value of tee.

Author(s)

Robin K. S. Hankin

See Also

bezier_integral

Examples

```
P <- matrix(c(1, 3, 6, 4, 7, 3, 2, 2),ncol=2)
bezier_find_length(P,5)
```

---

bezier_integrals *Arcwise integrals over Bezier curves*

Description

Various integrals over Bezier curves such as total arc length and bending energy

Usage

```
bezier_arclength(P, t1=0,t2=1,give=FALSE,...)
bezier_bending_energy(P, t1=0,t2=1, give=FALSE, power=2, ...)
```

Arguments

P	Control points in the form of a 4 by 2 matrix with rows corresponding to P_0 to P_3
give	Boolean, with TRUE meaning to return more information and default FALSE meaning to return just the value of the integration as estimated by the numerical routine
power	Function bezier_bending_energy() returns bending energy is $\int_S \frac{ds}{R^{\text{power}}}$, where $R = R(s)$ is the radius of curvature. If $\text{power} = 2$ this corresponds to the Eulerian bending energy of a flexible beam
t1,t2	In function bezier_arclength(), the values of t to start and end the integration at
...	Further arguments passed to integrate()

Details

These functions use numerical integration, specifically `integrate()`, between two specified points on a Bezier curve.

1. Function `bezier_bending_energy()` gives the and bending energy ($\int R^{-1} ds$).
2. Function `bezier_arclength()` gives the arc length.

Author(s)

Robin K. S. Hankin

See Also

`bezier_angle`

Examples

```
P <- matrix(c(0, 1, 2, 2, 2, 0, 3, 2),4,2)

bezier_arclength(P)
```

`crossing` *Crossing Metrics for knots*

Description

Various descriptions for the crossing points of a knot

Usage

```
crossing_points(b, give_all = TRUE)
crossing_matrix(b)
crossing_strands(b)
```

Arguments

`b`	A list of Bezier control parameters, typically given by `getcontrolpoints()`
`give_all`	In function `crossing_points()`, Boolean, with `TRUE` meaning to give details of the strands that cross and default `FALSE` meaning to give just the coordinates of the crossing points

Author(s)

Robin K. S. Hankin

See Also

as.controlpoints,bezier

Examples

```
crossing_points(k7_2,give_all=TRUE)
```

`getstringpoints` *Returns the coordinates of a knot's path*

Description

Returns the coordinates of a knot's path

Usage

```
getstringpoints(b, give_strand = FALSE, n = 100)
```

Arguments

b	Knot path, object coerced to `controlpoints` form
give_strand	Boolean, with default `FALSE` meaning to return a two-column matrix with rows corresponding to coordinates of the strands, and `TRUE` meaning to return a matrix with an additional column indicating the strand number
n	The number of points to use when constructing the Bezier curve

Value

Returns either a two- or three- column matrix

Note

Function `knotplot()` returns the points of the string too, but with `NA` for understrands.

Author(s)

Robin K. S. Hankin

See Also

knotplot

Examples

```
plot(getstringpoints(k4_1),asp=1)

a <- getstringpoints(k11a179,TRUE)
plot(a,asp=1,col=rainbow(24)[a[,3]])

d <- 1200
plot(rbind(
    sweep(getstringpoints(k7_1),2,c(0,0)),
    sweep(getstringpoints(k7_2),2,c(0,d)),
    sweep(getstringpoints(k7_3),2,c(d,0)),
    sweep(getstringpoints(k7_4),2,c(d,d))
),asp=1,xlab='',ylab='')
```

`head.inkscape` *Head and tail methods for inkscape objects*

Description

Head and tail methods for inkscape objects

Usage

```
## S3 method for class 'inkscape'
head(x, ...)
## S3 method for class 'inkscape'
tail(x, ...)
```

Arguments

`x`	Primary argument, an inkscape object
`...`	Further arguments, passed to `head()` or `tail()`

Author(s)

Robin K. S. Hankin

Examples

```
a <- reader(system.file("7_1.svg",package="knotR"))
head(a)
tail(a)

head(as.inkscape(k8_2))
```

is.sensible *Check to see whether an overunderobject is sensible*

Description

Check to see whether an overunderobject is compatible with a particular knot path

Usage

```
is.sensible(overunderobj, knot)
```

Arguments

overunderobj	A two-column matrix specifying the overs and unders
knot	A knot object, coerced to controlpoints form

Value

Returns TRUE or FALSE

Author(s)

Robin K. S. Hankin

See Also

overunder

Examples

```
is.sensible(overunder(k5_1),k5_1)
```

knotoptim *Optimization of knot appearance*

Description

Optimization of knot appearance using user-definable objective functions

Usage

```
knotoptim(svg, weights=1, symobj=NULL,
  Mver = NULL, xver = NULL, Mhor = NULL, xhor = NULL, Mrot = NULL,
  mcdonalds = FALSE, celtic = FALSE, ou, prob = 0, useNLM=TRUE, ...)
```

Arguments

svg	Name of an svg file to read
Mver,xver,Mhor,xhor,Mrot,mcdonalds,celtic	Arguments passed to symmetry_object(), specifying the symmetry of the knot
symobj	A symmetry object
ou	An overunder object
prob	The probability of plotting a knotplot; this is slow so don't make this too big
weights	A vector of weights, defaulting to all ones, passed to badness()
useNLM	Boolean, with default TRUE meaning to use nlm() and FALSE meaning to use optim()
...	Further arguments passed to nlm()

Details

Function knotoptim() is a generic optimization routine that starts from an svg file and minimizes the knot's badness.

Value

Returns a knot object

Author(s)

Robin K. S. Hankin

See Also

symmetry_object, badness

Examples

```
## Not run:     #takes too long
knotoptim(
      svg = system.file("4_1_first_draft.svg",package="knotR"),
     Mver = rbind(c(2,3),c(9,7),c(10,6),c(1,4),c(5,11)),
     xver = 8,   # node on vertical axis
     ou   = rbind( c(1,5), c(9,2), c(4,8),c(6,11)),
     prob = 0.1,
  iterlim = 100, print.level=2)

## End(Not run)
```

knotplot *Plotting of knots*

Description

Routines to plot projections of knots with a wide range of user-settable options

Usage

```
knotplot(x, ou, gap=20, n=100, lwd=8, setup=TRUE, ...)
knotplot2(x, rainbow=FALSE, seg=FALSE, text=FALSE, cross=FALSE, ink=FALSE,
                node=FALSE, width=TRUE, all=FALSE, n=100, circ=1000,
                lwd=8, setup=TRUE,...)
```

Arguments

x	Description of a knot, coerced to a controlpoints object and a minobj object
rainbow,seg,text,cross,ink,node,all,width,circ	Variables controlling sundry knotplot() features; see details
ou,gap	Variables controlling sundry knotplot2() features; see details
n	Number of points on each Bezier curve
lwd	Width of line to use
setup	Boolean, with default TRUE meaning to set up a new plot, and FALSE meaning to just add points and lines to an existing plot
...	Further arguments, passed to plot() and points()

Details

Function knotplot() is useful for production-quality plotting of knots with crossings indicated; knotplot2() is more useful for development.

For knotplot2():

- rainbow; use rainbow colouring for the segments
- seg; plot the Bezier nodes and handles. The positions of the nodes and handles are obtained from an object of class controlpoints.
- text; include the segment number on the segment
- cross; label the crossings
- ink; label the nodes with their inkscape numbering
- width; show the bending strain energy

For knotplot():

- overunderobj; A two-column matrix indicating the sense of the crossing. Each row corresponds to a crossing; the first entry is the segment number of the overstrand, and the second is the understrand
- gap; the width of the gap
- arclength Gap width measured by arc length or Euclidean distance

Author(s)

Robin K. S. Hankin

Examples

```
knotplot(k5_1)

knotplot2(k6_1,text=TRUE,seg=TRUE,lwd=1)
```

---

knots *Optimized knots*

Description

A variety of knots with optimized forms

Details

A selection of knots that have been optimized for visual appearance. The list makes no claims for completeness; the examples are intended to show the abilities of the package.

Knots with names like `k7_3` use the naming scheme of Rolfsen.

Knots with names like `k11n157` follow the nomenclature of the Hoste-Thistlethwaite table; 'a' means 'alternating' and 'n' means 'nonalternating'.

Knot `k12a_614` is drawn from the "Table of Knot Invariants" by Livingstone and Cha.

Knot `amphichiral15` is the unique amphichiral knot with crossing number 15, due to Hoste, Thistlethwaite, and Weeks.

Knots `k12n_0411` and `k11a203` show that partial symmetry may be enforced.

Knot `k8_18` is an exceptional knot.

Knot `pretzel_p3_p5_p7_m3_m5` is drawn from a knot appearing in Bryant 2016. The notation specifies the sense ('p' for plus and 'm' for minus) of the twists.

Knot `T20` is a "remarkable 20-crossing tangle"; see references

References

- J. C. Cha and C. Livingston. *KnotInfo: Table of Knot Invariants*, http://www.indiana.edu/~knotinfo, July 7, 2016
- K. A. Bryant, 2016. *Slice implies mutant-ribbon for odd, 5-stranted pretzel knots*, arXiv:1511.07009v2
- S. Eliahou and J. Fromentin 2017. "A remarkable 20-crossing tangle". Arxiv, https://arxiv.org/abs/1610.05560v2

Examples

```
knotplot(k3_1)
## maybe str(k3_1) ; plot(k3_1) ...
```

`overunder` *Functionality for specifying overstrands and understrands*

Description

Functionality for specifying overstrands and understrands

Usage

```
overunder(x)
overunder(x) <- value
mirror(x)
```

Arguments

`x`	A knot object
`value`	A two-column integer matrix with rows corresponding to crossings. In each row, the first entry identifies the overstrand, and the second entry specifies the understrand

Details

These functions are not really needed but are here for convenience.

Value

All three functions return a knot object

Author(s)

Robin K. S. Hankin

See Also

`knot`

Examples

```
overunder(k4_1)

par(mfcol=c(1,2))
knotplot(k4_1,gap=80)
knotplot(mirror(k4_1),gap=80)
```

reader *Reading and writing svg files*

Description

Various utilities for reading and creating svg files for use with inkscape

Usage

```
reader(filename)
write_svg(k, oldfile, safe=TRUE,
    regex1 ='sodipodi:docname=',
    regex2=' *d *= *" *M.*C.*[zZ] *"')
```

Arguments

filename	Name of a file to be read by reader(); usually an inkscape .svg file
safe	Boolean, with default TRUE meaning to save file "foo.svg" as "foo_smooth.svg" and FALSE meaning to overwrite foo.svg.
k, oldfile, regex1, regex2	Various arguments sent to write_svg(); see the source code for details. Argument k is a knot, oldfile an .svg file for reference.

Details

Function reader() is the way to get started with a new knot. This takes a filename which is an .svg file created with inkscape. Instructions for creating a suitable inkscape file are given in knotR-package.Rd.

Note

Inkscape's default is to use a mixture of absolute and relative coordinates. Function reader() assumes that the .svg file uses only absolute coordinates.

To ensure that only absolute coordinates are used, open the 'SVG output' menu in 'inkscape preferences' and uncheck the "Allow relative coordinates" option.

The format of .svg file is described in the W3C recommendation (2011) for Scalable Vector Graphics (SVG) 1.1, second edition.

Author(s)

Robin K. S. Hankin

See Also

utilities,knotR-package

Examples

```
## Not run:
a <- reader("6_3.svg")
b <- getcontrolpoints(a)
knotplot(a)


## End(Not run)
```

---

symmetrize *Symmetry and knots*

Description

Various functionality to impose different types of symmetry on knots

Usage

```
force_nodes_mirror_images_LR(x,symobj)
force_nodes_mirror_images_UD(x,symobj)
force_nodes_exactly_horizontal(x,symobj)
force_nodes_exactly_vertical(x,symobj)
force_nodes_on_V_axis(x,xver)
force_nodes_on_H_axis(x,xhor)
force_nodes_rotational(x,symobj)
symmetrize(x,symobj)
tag_notneeded(x, Mver, xver, Mhor, xhor, Mrot,exact_h,exact_v)
make_minsymvec_from_minobj(x,symobj)
minsymvec(vec)
make_minobj_from_minsymvec(minsymvec,symobj)
symmetry_object(x, Mver=NULL, xver=NULL, Mhor=NULL, xhor=NULL,
Mrot=NULL, exact_h=NULL, exact_v=NULL,
mcdonalds=FALSE, celtic=FALSE, reefknot=FALSE,center_crossing=FALSE)
knot(x, overunderobj, symobj, Mver=NULL, xver=NULL, Mhor=NULL,
xhor=NULL, Mrot=NULL, mcdonalds=FALSE, celtic=FALSE,
reefknot=FALSE,center_crossing=FALSE)
```

Arguments

x	Object coerced to class minobj
Mver,Mhor	Matrices specifying vertical (horizontal) symmetry, with two columns. The rows specify pairs of symmetric nodes about a vertical (horizontal) axis. Nodes specified by the first column should be on the left (upper) side
Mrot	A matrix specifying rotational symmetry. Each row corresponds to a set of nodes in a rotational relationship. The number of columns specifies the order of the rotational symmetry

`xver,xhor` Vector specifying nodes to be on the vertical (horizontal) axis of symmetry. The nodes are assumed to flow from left to right

`exact_h,exact_v` Vector specifying nodes to be exactly horizontal or exactly vertical. A node is exactly horizontal (vertical) if the y(x) coordinate of the node is the same as the y(x) coordinate of the handle. Note that the position of an exactly horizontal node is not restricted, and may be anywhere

`symobj` An object representing the symmetry of the knot, usually created by function `symmetry_object()`

`mcdonalds` For vertical symmetry, argument `mcdonalds` is Boolean, defaulting to `FALSE`, with `TRUE` meaning that the symmetric pairs of strands approach the vertical line of symmetry in the same sense (either both moving inward, or both moving outward). It is hard to explain (and named for the gesture one makes when tracing the top two strands a knot with this type of symmetry). The only common knot that needs this is `7_2`

`celtic` Like `mcdonalds` but for horizontal symmetry

`reefknot` Like `mcdonalds` but for the reefknot

`center_crossing` Implements a peculiar type of rotational symmetry in which the strands pass through the geometrical center of the knot projection. The only common knot needing this is `9_29`

`minsymvec` A "minimal symmetric vector". This is a numeric vector containing just the independent degrees of freedom of a knot, after symmetry constraints have been imposed. The idea is that one may optimize a `minsymvec` object using `nlm()`, and then reconstruct a knot using `make_minobj_from_minsymvec()` together with a symmetry object

`vec` A vector, given to function `minsymvec()`

`overunderobj` A matrix specifying the overs and the unders; a two-column matrix with rows corresponding to pairs of strands intersecting. The first element of a row identifies the overstrand and the second element specifies the understrand

Details

- Function `symmetry_object()` creates a symmetry object from `Mver` et seq, but if given a `knot` object, returns the embedded symmetry object.
- Functions `force_nodes_mirror_images_LR()` and `force_nodes_mirror_images_UD()` symmetrize a knot about a vertical (horizontal) axis by taking ordered pairs of nodes, specified by matrix `Mver` (`Mhor`) and forcing the second node to be symmetrically placed with respect to the first
- Functions `force_nodes_exactly_horizontal()` and `force_nodes_exactly_vertical()` force nodes to be exactly horizontal or exactly vertical respectively. Nodes so forced do not need to be on an axis of symmetry; they can be anywhere
- Function `symmetrize_ROT()` symmetrizes a knot around a point of symmetry, producing a rotationally symmetric knot

Value

These functions return a symmetric knot in `minobj` form.

Note

You can achieve up-down symmetry (that is, a horizontal line of symmetry) by making a left-right symmetric knot and rotating by 90 degrees. D'oh.

Author(s)

Robin K.S. Hankin

Examples

```
# each row of M = a pair of symmetrical nodes; each element of v is a
# node on the vertical axis

M <- matrix(c(6,4,13,11,7,3,2,8,9,1,14,10),byrow=TRUE,ncol=2)
v <-  c(5,12) # on vertical axis

sym_7_3 <- symmetry_object(k7_3, M, v)

k <- symmetrize(as.minobj(k7_3), sym_7_3)

knotplot2(k)  #nice and symmetric!


## OK now convert to and from a mimimal vector for a symmetrical knot:

mii <- make_minsymvec_from_minobj(k, sym_7_3)
pii <- make_minobj_from_minsymvec(mii,sym_7_3)
knotplot2(pii)


##   So 'mii' is a minimal vector for a symmetrical knot, and 'pii' is
##   the corresponding minobj object.  Note that you can mess about with
##   mii, but whatever you do the resulting knot is still symmetric:

mii[2] <- 1000
knotplot2(make_minobj_from_minsymvec(mii,sym_7_3))   # still symmetric.

## and, in particular, you can optimize the badness, using nlm():

## Not run:
fun <- function(m){badness(make_minobj_from_minsymvec(m,sym_7_3))}
o <- nlm(fun,mii,iterlim=4,print.level=2)

knotplot2(make_minobj_from_minsymvec(o$estimate,sym_7_3))

## End(Not run)
```

`utilities` *Various utilities for knots*

Description

Various utilities for knots including reading files and creating objects

Usage

```
controlpoints(x)
inkscape(x)
minobj(x)
knotvec(x)
make_controlpoints_from_ink(a)
make_minobj_from_ink(a)
make_minobj_from_vector(vec)
make_ink_from_minobj(x)
make_inkscape_from_controlpoints(b)
make_minobj_from_knot(k)
make_knotvec_from_minobj(x)
```

Arguments

`x`	Suitable object for coercion; see details
`a`	An `inkscape` object: a two column matrix with rows representing the positions of nodes and control points
`b`	A controlpoints object
`k`	An object of class knot
`vec`	A vector of reals

Details

Functions `inkscape()`, `minobj()`, and `knotvec()` are low-level functions; these are the only places that objects have their classes assigned directly. These functions are not user-friendly and require very specific types of object; they perform some checks but are not really intended for the user. Functions `as.foo()` are much more user-friendly, and are documented at `as.Rd`.

Functions `make_foo_from_bar()` coerce `bar` objects into `foo` objects. Functions that involve symmetry are documented at `symmetry.Rd`.

Objects of class `inkscape` are in the form of a two-column matrix, with rows corresponding to 2D positions. The rows correspond to the (x, y) coordinates of points as held in the inkscape file.

There is quite a lot of redundancy in an inkscape object:

- The first row of an inkscape object is equal to the last row (this follows from the fact that the path is closed).

- If $n = 0$ modulo 3, then `a[n+2,]-a[n+1,]==a[n+1,]-a[n]`, corresponding to the fact that the handles are symmetric in inkscape. This is visualised best by `knotplot2(k4_1,ink=TRUE,seg=TRUE)`

Look at functions `make_inkscape_from_minobj()` and `make_minobj_from_ink()` to see this from a symbolic perspective. The vignette also gives some details.

The `minobj` class is a 'MINimal OBJect'; there is no redundancy. Objects of class `minobj` are a list of two elements: `$node` and `$handle_A`. Each element has rows corresponding to 2D positions, the same as `inkscape` objects. Element `$node` shows the positions of the nodes, and element `$handle_A` shows the positions of (one of) the handles; the other handle is symmetrically positioned with respect to its node. Use `knotplot2(k4_1,node=TRUE,seg=TRUE)` to see the meaning of the entries; the nodes are indicated by a square and the handles by circles.

An object of class `controlpoints` is a list of matrices of size 4-by-2. For each matrix, the four rows correspond to the points in 2D Cartesian space needed to specify a Bezier curve; further details and examples are given in `bezier.Rd`.

The `knotvec` class is a named vector of independent reals suitable for use with optimization routines.

None of the functions here deal with symmetry relations. This is documented at `symmetry.Rd`.

Author(s)

Robin K. S. Hankin

See Also

`knotplot`,`symmetrize`,`bezier`

Examples

```
## Not run:
a <- reader("6_3.svg")  # 'a' is an inkscape object.
knotplot(a)



## End(Not run)
```

Index