Package ‘knotR’

January 24, 2024

Type Package

Title Knot Diagrams using Bezier Curves

Version 1.0-4

Depends R (>= 2.10)

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

LazyData TRUE

Description Makes visually pleasing diagrams of knot projections using optimized Bezier curves.

License GPL-2

NeedsCompilation no

Author Robin K. S. Hankin [aut, cre] (<https://orcid.org/0000-0001-5982-0415>)

Repository CRAN

Date/Publication 2024-01-24 08:50:02 UTC

R topics documented:

---

knotR-package *Knot Diagrams using Bezier Curves*

Description

Makes visually pleasing diagrams of knot projections using optimized Bezier curves.

Details

The DESCRIPTION file:

Package:	knotR
Type:	Package
Title:	Knot Diagrams using Bezier Curves
Version:	1.0-4
Authors@R:	person(given=c("Robin", "K. S."), family="Hankin", role = c("aut","cre"), email="hankin.robin@gmail.com",
Depends:	R (>= 2.10)
Maintainer:	Robin K. S. Hankin <hankin.robin@gmail.com>
LazyData:	TRUE
Description:	Makes visually pleasing diagrams of knot projections using optimized Bezier curves.
License:	GPL-2
Author:	Robin K. S. Hankin [aut, cre] (<https://orcid.org/0000-0001-5982-0415>)

Index of help topics:

```
as                      Conversions between various forms of a knot
badness                 Badness of knots
bezier                  Various functionality for Bezier curves
bezier_angle            Intersection of two Bezier curves
bezier_find_length      Solve for arclength
bezier_integrals        Arcwise integrals over Bezier curves
crossing                Crossing Metrics for knots
getstringpoints         Returns the coordinates of a knot's path
head.inkscape           Head and tail methods for inkscape objects
knotoptim               Optimization of knot appearance
knotplot                Plotting of knots
knotR-package           Knot Diagrams using Bezier Curves
knots                   Optimized knots
overunder               Functionality for specifying overstrands and
                        understrands
reader                  Reading and writing svg files
symmetrize              Symmetry and knots
utilities               Various utilities for knots
```

The package contains a large number of knots, optimized for visual appearance in the sense that the knot path is nice and smooth, and strands cross at close to right angles. These can be displayed by typing

```
knotplot(k9_23)
```

at the R prompt. The package includes all prime knots up to and including 9 crossings, and a number of other interesting and attractive knots.

The package facilitates the creation and optimization of new knots. 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 path 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 `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.

The package provides four classes of objects that specify the path of a knot: `inkscape`, `minobj`, `controlpoints`, and `knotvec`. These four classes have different uses, and objects may be converted from one form to another by using functions such as `as.minobj()`, documented at `as.Rd` and `utilities.Rd`.

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)

NA

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

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:

# Following examples take a long time to 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)
total_crossing_angle_badness(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 (it is short for '`crossing_points(b)`'). It is time-consuming to calculate, so providing a pre-calculated value makes the code run faster
`prob`	In function `badness()`, the probability of plotting a knotplot. I used nonzero values in the early stages of developing the package: when optimizing a knot it was useful 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. See details
`...`	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 a weighted sum of nine individual badnesses.

The list below details the values returned by `metrics()`; the description of each item is the name of corresponding weight assigned by the `weights` argument of `badness()`.

- **pot** Function `total_crossing_potential_energy()` gives the potential energy of the nodes, under an inverse square force law
- **ang** Function `total_crossing_angle_badness()` returns a high value if strands cross at angles far from 90 degrees. It returns the sum, over all crossings, of `bezier_angle()`
- **ben** Function `total_bending_energy()` gives the total bending energy, effectively the arc integral of the reciprocal of the square of the radius of curvature
- **len** 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
- **mid** Function `midpoint_badness()` penalizes knots with crossing points far from the midpoint of segments
- **clo** Function `node_crossing_badness()` penalizes knots with nodes too close together (compare function `total_crossing_potential_energy()`)

swi Function `curvature_switching_badness()` provides a penalty for consecutive segments with curvatures that switch sign. The magnitude of the penalty is zero if both curvatures are of the same 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

con Function `curvature_consecutive_segment_switching_badness()` penalizes knots with consecutive segments that switch curvature from positive to negative

ncn 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

The `weights` argument is nominally a vector of length 9 which is used to assign weights to different aspects of the badness of a knot.

Value

Returns a scalar badness

Author(s)

Robin K. S. Hankin

See Also

crossing

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)



## default:
badness(k_infinity, weights=c(1,1,1,1,1,1,1,1,1))


## downweight the importance of strands crossing at 90 degrees:
badness(k_infinity, weights=c(1,0.1,1,1,1,1,1,1,1))
```

`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.

In function `bezier_intersect()`, argument `type` may take the following values:

pos Position of intersection point

cons Boolean, indicating whether the strands abut; the 'intersection' point is the end of one curve and the beginning of the other

bool Boolean, indicating whether or not the strands actually intersect

para Bezier parameter t for the intersection point; actually return two parameters, one for each curve

opt Details of the optimization output

all Everything

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 $\int_S\frac{ds}{R^{\mathrm{power}}}$, where $R=R(s)$ is the radius of curvature. If $\mathrm{power}=2$ 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	The knot path (coerced to `controlpoints` form)
give_strand	Boolean, with default `FALSE` meaning to return a two-column matrix with rows corresponding to coordinates of the knot path, 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))
```

---

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().

The weights argument is documented more fully at badness.Rd.

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, gapwidth=1, n=100, lwd=8, add=FALSE, ...)
knotplot_old(x, ou, gap=20, n=100, lwd=8, add=FALSE, ...)
knotplot2(x, rainbow=FALSE, seg=FALSE, text=FALSE, cross=FALSE, ink=FALSE,
                node=FALSE, width=TRUE, all=FALSE, n=100, circ=1000,
                lwd=8, add=FALSE, ...)
```

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 `knotplot2()` features; see details
`ou`	An overunder object, useful if overunder information not included in argument `x`
`gap,gapwidth`	Variables controlling visual representation of strand crossings; see details
`n`	Number of points on each Bezier curve
`lwd`	Width of line to use
`add`	Boolean, with default `FALSE` meaning to set up a new plot, and `TRUE` 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 by the understrand having a gap; function `knotplot2()` is more useful for development. Function `knotplot_old()` is included for backward compatibility and is possibly more robust than `knotplot()`.

Function `knotplot()` works by setting a suitable length of the understrand to `NA` which results in it not being plotted.

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
- `gapwidth`; the width of the gap, measured in units of width of the string

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

The `gap` argument of `knotplot_old()` is a the same as the `gapwidth` argument of `knotplot()` but `gap` is measured in the same units as the `plot()`.

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

Knots `k12a1202` and `k12n838` are named following Lamm.

As of version 1.0-4, the complete list of knots is:

k10_1, k10_123, k10_47, k10_61, k12a1202, k12n838, k3_1, k3_1a, k4_1, k4_1a, k5_1, k5_2, k6_1, k6_2, k6_3, k7_1, k7_2, k7_3, k7_4, k7_5, k7_6, k7_7, k7_7a, k8_1, k8_10, k8_11, k8_12, k8_13, k8_14, k8_15, k8_16, k8_17, k8_18, k8_19, k8_19a, k8_19b, k8_2, k8_20, k8_21, k8_3, k8_3_90deg_crossing, k8_4, k8_4a, k8_5, k8_6, k8_7, k8_8, k8_9, k9_1, k9_10, k9_11, k9_12, k9_13, k9_14, k9_15, k9_16, k9_17, k9_18, k9_19, k9_2, k9_20, k9_21, k9_22, k9_23, k9_23a, k9_24, k9_25, k9_26, k9_27, k9_28, k9_29, k9_3, k9_30, k9_31, k9_32, k9_33, k9_34, k9_35, k9_36, k9_37, k9_38, k9_39, k9_4, k9_40, k9_41, k9_42, k9_43, k9_44, k9_45, k9_46, k9_47, k9_48, k9_49, k9_5, k9_6, k9_7, k9_8, k9_9, D16, T20, amphichiral15, celtic3, fiveloops, flower, fourloops, hexknot, hexknot2, hexknot3, k_infinity, k11a1, k11a179, k11a361, k11n157, k11n157_morenodes, k11n22, k12n_0242, k12n_0411, longthin, ochiai, ornamental20, perko_A, perko_B, pretzel_2_3_7, pretzel_7_3_7, pretzel_p3_p5_p7_m3_m5, reefknot, satellite, sum_31_41, three_figure_eights, trefoil_of_trefoils, triloop, unknot

References

- K. A. Bryant, 2016. *Slice implies mutant-ribbon for odd, 5-stranded 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)
is.sensible(overunderobj,x)
```

Arguments

`x`	A knot object
`value,overunderobj`	A two-column integer matrix specifying the senses of the crossings in a knot

Details

Function overunder() returns a two-column integer matrix with rows corresponding to crossing points. The first element of each row corresponds to the strand number of the overstrand and the second element corresponds to the understrand.

Function is.sensible() checks to see whether the overunder matrix is compatible with the knot path. For example, it checks to see whether each crossing has exactly one row, and that each row corresponds to a pair of strands that actually cross.

Function mirror() takes a knot and returns the knot with the senses of each crossing reversed; it is as though the knot is reflected in the plane of the projection.

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)

is.sensible(overunder(k6_1),k6_1)
```

---

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.

Sometimes, `reader()` will fail with a valid `.svg` file if a node is sufficiently close to the x or y axis to require exponential notation (this typically happens with complicated rotational symmetry). If the file contains text like

`...35.3635879230533 -1.323423734554e-15 , 10.3538368384142...`

the second value is zero to numerical precision, but the text form of the number interferes with the operation of `reader()`. To deal with this we need to edit the file in a text editor and replace the offending number with an exact zero:

`...35.3635879230533 0 , 10.3538368384142...`

(I guess the ideal would be to incorporate some clever regexp technique into `reader()` but this turned out to be harder than I thought).

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 (resp. horizontal) symmetry, with two columns. The rows specify pairs of symmetric nodes about a vertical (resp. horizontal) axis. Nodes specified by the first column should be on the left (resp. upper) side; these are fixed. Used by functions `force_nodes_mirror_images_LR()` and `force_nodes_mirror_images_UD()` which move the right (resp. lower) nodes and their associated handles to the positions required for exact vertical (resp. horizontal) symmetry
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. The first column corresponds to nodes whose position is fixed. Used by `force_nodes_rotational()`, which also moves handles appropriately

xver,xhor — Vector specifying nodes to be on the vertical (resp. horizontal) axis of symmetry. The nodes are assumed to flow from left to right. Used by functions force_nodes_on_V_axis() and force_nodes_on_H_axis() respectively, which also move the handles

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

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.

There are seven types of symmetry that may be imposed on a knot. These are imposed by the following seven force_nodes_foo() functions:

- Functions force_nodes_mirror_images_LR() and force_nodes_mirror_images_UD() symmetrize a knot about a vertical (resp. horizontal) axis by taking ordered pairs of nodes, specified by matrix Mver (resp. Mhor) and forcing the second node to be symmetrically placed with respect to the first. It does the same thing to the handles too.

- Functions `force_nodes_exactly_horizontal()` and `force_nodes_exactly_vertical()` force nodes to be exactly horizontal (resp. vertical) by restricting the position of their handles. Nodes so forced do not need to be on an axis of symmetry; they can be anywhere
- Functions `force_nodes_on_V_axis()` and `force_nodes_on_H_axis()` force nodes specified by `xver` (resp. `xhor`) to be on the vertical (resp. horizontal) axis, and to have appropriately placed handles
- Function `force_nodes_rotational()` imposes the rotational symmetry specified by `Mrot`

Function `symmetrize()` imposes the seven kinds of symmetry by calling each of the `force_nodes_foo()` functions in turn.

Function `tag_notneeded()` is an internal function, not really intended for the end-user. It takes a `minobj` object and marks a maximal set of dependent entries with a 'not needed' value. The values of the entries so marked may be determined by a combination of the imposed symmetry relations and the unmarked values. The unmarked entries constitute a `minsymvec` object (see above). These are the *real* degrees of freedom in the symmetrical knot. Only these unmarked values are modified by the optimization routines in `knotoptim()`

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'. 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.

NB: objects of class `minobj` have no *redundancy* in the sense that changing any entry of either `$node` or `$handle_A` results in modifying the corresponding inkscape diagram. However, doing this to a knot which has imposed symmetry conditions (ie a nontrivial `symobj`) may introduce asymmetry into the inkscape diagram. For example, one might take a `minobj` object whose knot diagram is left-right mirror symmetric (eg `k6_2`) and alter one of the handle positions. Then the resulting inkscape object will be asymmetric. There is an example using `k6_2` below.

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`. There is lots of redundancy in a `controlpoints` object because the inkscape nodes are symmetric nodes with diametrically opposed handles.

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

`as`,`symmetrize`

Examples

```
a <- as.minobj(k6_3)
plot(a$node,asp=1,pch=16)
segments(x0=a[[1]][,1],y0=a[[1]][,2],x1=a[[2]][,1],y1=a[[2]][,2],
main="handle direction follows the string path")
points(getstringpoints(a),type='l',col='gray',lwd=0.4)
```

Index