The Rknots package

Federico Comoglio[1] and Maurizio Rinaldi[2]

[1] D-BSSE, ETH Zurich, Basel, Switzerland
[2] DiSCAFF, University of Piemonte Orientale, Novara, Italy

federico.comoglio@bsse.ethz.ch

February 25, 2012

Abstract

From a topological point of view, polymers can be modeled as open polygonal paths that upon closure generate topological objects called knots. Multi component knots are known as links. The *Rknots* package contains functions for the topological analysis of knots and links with a particular focus on biological polymers like proteins.
This vignette explains the use of the package and is divided in three main parts. The first one deals with structure import, the second is focused on the methods that have been implemented and the third one is a case study that illustrates how to use the general functions presented in the previous two sections for the analysis of proteins. For a more formal exposition of the methods, especially of the HOMFLY polynomial computation, please refer to [1]. Should you have any question or suggestion, feel free to email us.

Contents

1 File import and input structure **1**
1.1 Datasets . . . 2
1.2 Example knots and links . . . 4
1.3 Create objects of class *Knot* . . . 4

2 Structure reduction and invariant computation **7**
2.1 Structure reduction algorithms . . . 7
2.2 Computation of the invariants of knots and links . . . 11
2.3 *Rknots* and protein knot analysis, a dedicated pipeline . . . 13

3 Session Info **19**

1 File import and input structure

The *Rknots* package deals with knots and links and therefore it expects the coordinates of N points in the three-dimensional space together with a set of integer separators as an input. It is worth to give a clear description of these two attributes. N points in 3D can be naturally represented by a $N \times 3$ matrix, where each row is a point and the columns are the x, y, z

coordinates of that point. If the topological structure is a knot, then it will be entirely defined by its points. However, if we are dealing with links, a collection of knots where each single entity defines a component of the link, the coordinates matrix will not be sufficient. A vector S of component separators (also ends) is then required to describe the boundaries between components. Notice that there is an arbitrary and potentially dangerous choice on the separators due to the duality points-edges. We defined each separator as the index of the edge that if not removed would connect a component with the following and that can thus be regarded as a 'phantom' edge. For example, the Hopf link represented in figure 1 is defined by an 8×3 matrix and separator $S = 4$. The 'phantom' edge is indicated by the gray dashed line.

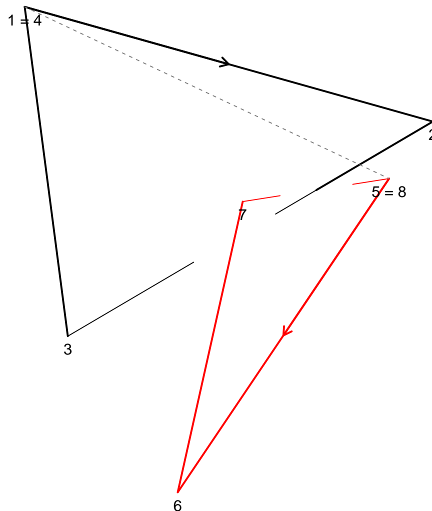


Figure 1: Polygonal Hopf link diagram. The component separator is $S = 4$.

1.1 Datasets

Rknots comes along with two datasets containing 250 knots having less than 11 crossings and enumerated according to Rolfsen [2] and 130 links up to 4 components. The 3D coordinates of knots are available as full representation or as minimal stickies representation and are stored in the dataset `Rolfsen.table`. The 3D coordinates of links and their separators are stored in the dataset `link.table`. Let's have a look at both datasets, that can be accomplished with the following commands:

```
> library("Rknots")
> data(Rolfsen.table)
> str( head(Rolfsen.table, 5) )
```

```
List of 5
 $ 3.1: num [1:48, 1:3] -5 -4.88 -4.39 -3.58 -2.54 ...
```

```
 $ 4.1: num [1:70, 1:3] -6.43 -6.26 -5.89 -5.32 -4.56 ...
 $ 5.1: num [1:70, 1:3] -5.58 -5.85 -5.75 -5.32 -4.66 ...
 $ 5.2: num [1:82, 1:3] -1.191 -0.262 0.803 1.922 3.027 ...
 $ 6.1: num [1:97, 1:3] -6.9 -6.46 -5.84 -5.05 -4.13 ...
```

Notice that each knot is an element of a list, and that a given knot can be accessed also by using its name.

```
> head( names(Rolfsen.table) )
```

```
[1] "3.1" "4.1" "5.1" "5.2" "6.1" "6.2"
```

For example, one can access the coordinates of the trefoil knot (knot 3_1) using `Rolfsen.table['3.1']`.

```
> str( Rolfsen.table['3.1'] )
```

```
List of 1
 $ 3.1: num [1:48, 1:3] -5 -4.88 -4.39 -3.58 -2.54 ...
```

The minimal stickies coordinates of knots are stored in the `Rolfsen.table.ms` list, which has the same names of the full representation.

```
> str( head(Rolfsen.table.ms, 5) )
```

```
List of 5
 $ 3.1: num [1:7, 1:3] 0.301 -0.976 0.976 -0.3 -1.276 ...
 $ 4.1: num [1:8, 1:3] -2.927 2.239 0.704 -0.907 -1.082 ...
 $ 5.1: num [1:9, 1:3] -0.0962 -0.7682 0.7677 0.0969 -1.4284 ...
 $ 5.2: num [1:9, 1:3] 1.4615 -1.4618 1.2112 0.0436 -0.9697 ...
 $ 6.1: num [1:9, 1:3] 0.8499 0.8001 -0.0797 -1.5796 2.6384 ...
```

```
> head( names(Rolfsen.table.ms) )
```

```
[1] "3.1" "4.1" "5.1" "5.2" "6.1" "6.2"
```

Graphically, the difference between the full representation of a knot and the minimal stickies representation is illustrated in figure 2 for the trefoil knot.
Links are stored in the `link.table` list. Each element of the list contains two slots, respectively the coordinates and separators of the link.

```
> data(link.table)
> str( head(link.table, 5) )
```

```
List of 5
 $ 2.2.1:List of 2
  ..$ points3D: num [1:74, 1:3] -6.3 -5.77 -5.13 -4.4 -3.59 ...
  ..$ ends    : num 37
 $ 4.2.1:List of 2
  ..$ points3D: num [1:83, 1:3] -0.603 -1.426 -2.171 -2.802 -3.292 ...
  ..$ ends    : num 40
 $ 5.2.1:List of 2
  ..$ points3D: num [1:79, 1:3] 0.333 -0.598 -1.518 -2.4 -3.213 ...
  ..$ ends    : num 39
```

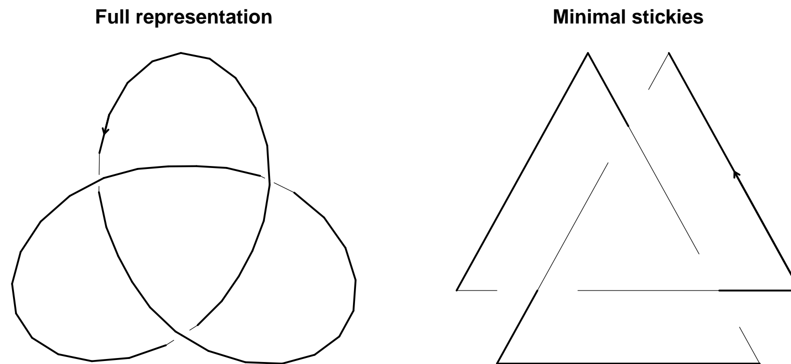


Figure 2: Full and minimal stickies representation of the trefoil knot.

```
 $ 6.2.1:List of 2
  ..$ points3D: num [1:102, 1:3] 2.131 1.328 0.447 -0.459 -1.338 ...
  ..$ ends    : num 50
 $ 6.2.2:List of 2
  ..$ points3D: num [1:98, 1:3] -6.57 -6.2 -5.67 -4.98 -4.14 ...
  ..$ ends    : num 48
```

```
> head( names(link.table) )
```

```
[1] "2.2.1" "4.2.1" "5.2.1" "6.2.1" "6.2.2" "6.2.3"
```

1.2 Example knots and links

For the first two sections of this vignette, we will deal with an example knot and link sampled randomly from the above described datasets. To pick a structure it is sufficient to type

```
> knot <- makeExampleKnot(k = TRUE) #for a knot
> link <- makeExampleKnot(k = FALSE) #for a link
```

This will turn out to be very convenient in the forthcoming sections, where we can use these structures to illustrate the core algorithms for structure reduction and polynomial invariants computation.

1.3 Create objects of class *Knot*

The main class in *Rknots* is an S4 class called *Knot*. The class has two slots:

1. `points3D`: an N x 3 matrix containing the x, y, z coordinates of points of a polygonal knot or link. Each row contains the 3D coordinates of a single point of the structure. This slot is of type `matrix`.

2. `ends`: a vector of integers containing the separators of the link components. This slot is by default set to `numeric(0)` for knots and is of type `numeric`.

A new object of class *Knot* can be created either using the generic constructor `new` or by means of the class constructor `newKnot`. The following example shows how to setup an object in both ways

```
> knot.cls <- new('Knot', points3D = knot)
> link.cls <- new('Knot', points3D = link$points3D, ends = link$ends)

> ( knot.cls <- newKnot(points3D = knot) )

An object of class 'Knot'
Slot points3D:  114 x 3 matrix
              x         y         z
 [1,] -3.621864 -7.859621 -2.811422
 [2,] -4.646111 -7.423467 -2.425915
 [3,] -5.546167 -6.849921 -1.935343
 [4,] -6.327957 -6.169424 -1.388736
 [5,] -7.004683 -5.402181 -0.820578
 [6,] -7.585773 -4.559776 -0.254575
 [7,] -8.073381 -3.646790  0.290954
 [8,] -8.457810 -2.662413  0.797152
 [9,] -8.714276 -1.603682  1.237616
[10,] -8.801561 -0.474100  1.570860
       ........  ........  ........
Slot ends:

> ( link.cls <- newKnot(points3D = link$points3D, ends = link$ends) )

An object of class 'Knot'
Slot points3D:  127 x 3 matrix
              x         y        z
 [1,] -9.162255 -1.509655 1.829496
 [2,] -9.245968 -2.659378 1.805300
 [3,] -9.113029 -3.803538 1.856751
 [4,] -8.744250 -4.892910 1.956594
 [5,] -8.136572 -5.871593 2.055047
 [6,] -7.310295 -6.682295 2.092592
 [7,] -6.311310 -7.268003 2.009992
 [8,] -5.216097 -7.575875 1.761636
 [9,] -4.130063 -7.568088 1.329447
[10,] -3.171682 -7.239913 0.731781
       ........  ........  ........
Slot ends:  49 100
```

If wished, the method `print` (e.g. `print(link.cls)`) allows to visualize the full output. To access the two class slots one can make use of the accessors `getCoordinates` and `getEnds` or the `[` operator. The use of `@` is discouraged.

```
> head( getCoordinates(knot.cls), 5 )

             x         y         z
[1,] -3.621864 -7.859621 -2.811422
```

```
[2,] -4.646111 -7.423467 -2.425915
[3,] -5.546167 -6.849921 -1.935343
[4,] -6.327957 -6.169424 -1.388736
[5,] -7.004683 -5.402181 -0.820578
```

```
> getEnds(knot.cls)
```

```
numeric(0)
```

```
> head( link.cls['points3D'], 5)
```

```
             x         y        z
[1,] -9.162255 -1.509655 1.829496
[2,] -9.245968 -2.659378 1.805300
[3,] -9.113029 -3.803538 1.856751
[4,] -8.744250 -4.892910 1.956594
[5,] -8.136572 -5.871593 2.055047
```

```
> link.cls['ends']
```

```
[1]  49 100
```

Accordingly, the content of the slots can be modified using the setters `setCoordinates`, `setEnds` or via `[<-`. For example, let's modify the object `knot.cls` by replacing its coordinates with some randomly generated ones

```
> knot.bu <- knot.cls #save the original
> new.coordinates <- matrix( runif(60), ncol = 3 )
> setCoordinates(knot.cls) <- new.coordinates
> knot.cls
```

```
An object of class 'Knot'
Slot points3D:  20 x 3 matrix
           [,1]      [,2]       [,3]
 [1,] 0.4089769 0.6405068 0.41372433
 [2,] 0.8830174 0.9942698 0.36884545
 [3,] 0.9404673 0.6557058 0.15244475
 [4,] 0.0455565 0.7085305 0.13880606
 [5,] 0.5281055 0.5440660 0.23303410
 [6,] 0.8924190 0.5941420 0.46596245
 [7,] 0.5514350 0.2891597 0.26597264
 [8,] 0.4566147 0.1471136 0.85782772
 [9,] 0.9568333 0.9630242 0.04583117
[10,] 0.4533342 0.9022990 0.44220007
       ........  ........  ........
Slot ends:
```

```
> knot.cls <- knot.bu #back to the original
```

Let's now try to modify the separators of the object `link.cls`

```
> link.bu <- link.cls #save the original
> setEnds(link.cls) <- c(10, 50, 90)
> getEnds(link.cls)
```

```
[1] 10 50 90

> link.cls <- link.bu
```

Notice that it is not required to replace n separators with n new separators. Certainly this operation changes the link we are dealing with, but it is very useful when local operations on the link are performed, giving rise to a different link (for example, by merging together two components).

Remarks *Rknots* has been developed to give users the most general purpose framework possible. Theoretically, there is no a priori limitation on the structure to be loaded, neither in terms of points nor link components. Practically, this depends on the downstream analysis to be performed. The coordinates of a structure to be loaded can be read in R with commands like `read.table`, `read.delim`, `read.csv`, etc. See the relevant help pages for details.

2 Structure reduction and invariant computation

Usually, the endpoint of the topological analysis of a knotted structure is the computation of an invariant, generally a polynomial and its computational cost is generally very high. Thus, a priori crucial step is to reduce the structure to a simpler form by reducing the number of point subject to the constraint of retaining all the topological information of the original structure. This is accomplished by applying structure reduction algorithms of knots and links [3, 4], described in the next section.

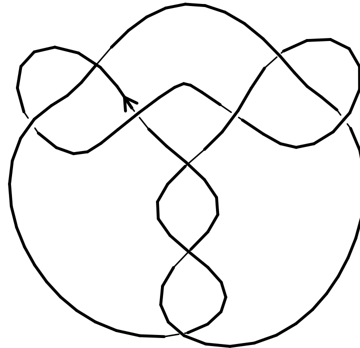
2.1 Structure reduction algorithms

Two structure reduction algorithms have been implemented in *Rknots*: the Alexander-Briggs (AB) algorithm [4] based on the elementary deformation and the Minimal Structure Reduction (MSR) algorithm [1] based on the generalized Reidemeister moves. The former is very efficient whereas the latter, by working on the knot projection, contains intrinsically more information at the price of being slower.
Let's first examine how to apply these algorithms outside the context of the *Knot* class. To reduce the knot we created, we can type

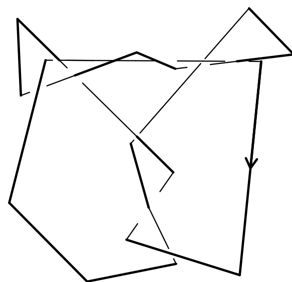
```
> knot.AB <- AlexanderBriggs(points3D = knot, ends = c())
> str(knot.AB)

List of 2
 $ points3D: num [1:17, 1:3] -3.62 -8.07 -6 6.32 5.09 ...
 $ ends    : num(0)

> knot.msr <- msr(points3D = knot, ends = c())
> str(knot.msr)

List of 3
 $ points3D: num [1:20, 1:3] -3.622 -6.782 2.587 7.076 0.714 ...
 $ ends    : NULL
 $ M       : num [1:19, 1:19] 0 0 0 0 0 0 0 0 0 -1 ...
```

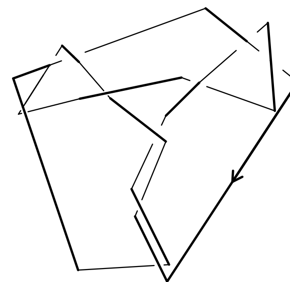
`Alexander-Briggs` returns a list with the reduced structure, `msr` additionally returns the intersection matrix M, which contains the position and sign of the crossings in the structure. The original structure and the two reduced ones can be visualized by plotting a knot diagram with `plotDiagram`.

```
> plotDiagram(knot, ends = c(), lwd = 2, main = 'Original Structure')
```




```
> par( mfrow=c(1,2) )
> plotDiagram(knot.AB$points3D, knot.AB$ends, lwd = 2, main = 'Reduced with Alexander-Briggs')
> plotDiagram(knot.msr$points3D, knot.msr$ends, lwd = 2, main = 'Reduced with MSR')
```

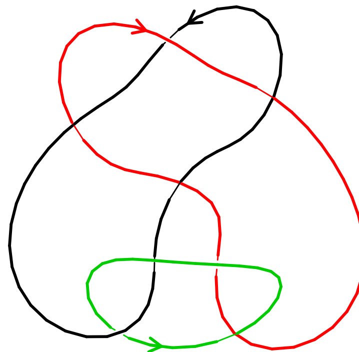




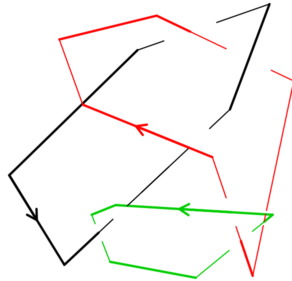
The same applies to the reduction of the link we created before, in its naive application just requires to include the link separators by using

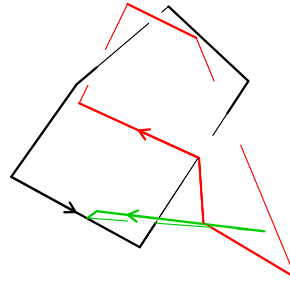
```
> link.AB <- AlexanderBriggs(points3D = link$points3D, ends = link$ends)
> str(link.AB)

List of 2
 $ points3D: num [1:19, 1:3] -9.16 -6.31 2.25 4.3 -2.53 ...
 $ ends    : num [1:2] 6 13

> link.msr <- msr(points3D = link$points3D, ends = link$ends)
> str(link.msr)

List of 3
 $ points3D: num [1:17, 1:3] -9.162 -1.974 4.102 -0.367 -5.506 ...
 $ ends    : num [1:2] 6 13
 $ M       : num [1:16, 1:16] 0 0 0 0 0 0 0 0 0 0 ...

> plotDiagram(link$points3D, link$ends, lwd = 2, main = 'Original Structure')
```

Original Structure



```
> par( mfrow=c(1,2) )
> plotDiagram(link.AB$points3D, link.AB$ends, lwd = 2, main = 'Reduced with Alexander-Briggs')
> plotDiagram(link.msr$points3D, link.msr$ends, lwd = 2, main = 'Reduced with MSR')
```



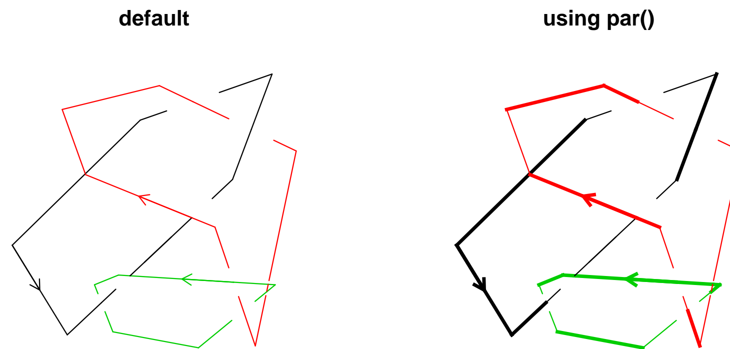


Notice that `msr` can be used for achieving a partial reduction by controlling the number of iterations `n` (default to 100).
An object of class *Knot* can be reduced by means of the `reduceStructure` method, that takes a *Knot* object and the algorithm to be applied as an input and returns an object of class *Knot* containing the reduced structure:

```
> knot.cls.AB <- reduceStructure(knot.cls, algorithm = 'AB' )
> knot.cls.MSR <- reduceStructure(knot.cls, algorithm = 'MSR' )
> link.cls.AB <- reduceStructure(link.cls, algorithm = 'AB' )
> link.cls.MSR <- reduceStructure(link.cls, algorithm = 'MSR' )
> link.cls.AB

An object of class 'Knot'
Slot points3D:  19 x 3 matrix
              x         y         z
 [1,] -9.162255 -1.509655  1.829496
 [2,] -6.311310 -7.268003  2.009992
 [3,]  2.249036  2.716286 -0.849297
 [4,]  4.296789  9.491975  0.390813
 [5,] -2.529795  6.531452 -0.926613
 [6,] -9.162255 -1.509655  1.829496
 [7,] -5.362280  3.025527 -1.999231
 [8,] -6.574879  7.215227 -1.337988
 [9,] -1.540334  8.744882  1.913969
[10,]  5.546724  4.548880 -2.562918
       ........  ........  ........
Slot ends:  6 13
```

The knot diagram can be drawn simply by using the method `plot` when the input is an *Knot* object.

```
> par( mfrow = c(1, 2) )
> plot(link.cls.AB, main = 'default') #default
> plot(link.cls.AB, lend = 2, lwd = 3, main = 'using par()') #thicker overcrossings.
> #see par() for additional options
```



2.2 Computation of the invariants of knots and links

Rknots can be used to compute the following invariants, listed according to the structure given as input.

- knots: Alexander, Jones and HOMFLY polynomials
- links: Jones and HOMFLY polynomials, the multivariable Alexander polynomial and the linking number

In contrast to the previous section, we will only examine how to compute polynomial invariants with an object of class *Knot*. We are currently working on the description on how the invariants are internally computed using low-level functions. Feel free to send us an email for a preliminary version.
Having an object of class *Knot* the desired invariant can be computed with the function `computeInvariant` that internally discriminates between knots and links and returns the appropriate polynomial as follows:

```
> ( delta.k <- computeInvariant( knot.cls.AB, invariant = 'Alexander') )

[1] "19 - 11*t1 - 11/t1 + 2/t1^2 + 2*t1^2"

> jones.k <- computeInvariant( knot.cls.AB, invariant = 'Jones', skein.sign = -1)
> homfly.k <- computeInvariant( knot.cls.AB, invariant = 'HOMFLY', skein.sign = -1)
```

and analogously for the previously created link

```
> ( delta.l <- computeInvariant( link.cls.AB, invariant = 'Alexander') )

[1] "t2*t3*(-t1 - t2 + t1*t2 - t1*t2*t3 + t1*t3*t2^2 + t2*t3*t1^2)"

> jones.l <- computeInvariant( link.cls.AB, invariant = 'Jones', skein.sign = -1)
> homfly.l <- computeInvariant( link.cls.AB, invariant = 'HOMFLY', skein.sign = -1)
```

The Jones and the HOMFLY polynomial of our sample knot and link are quite long and we printed the Alexander polynomial for sake of illustration. However, *Rknots* contains utilities to convert, when possible, a polynomial into another one. For example, we can convert the HOMFLY polynomial to the Jones polynomial by using

```
> converted <-HOMFLY2Jones( homfly.k )
> identical( converted, jones.k)
```

```
[1] TRUE
```

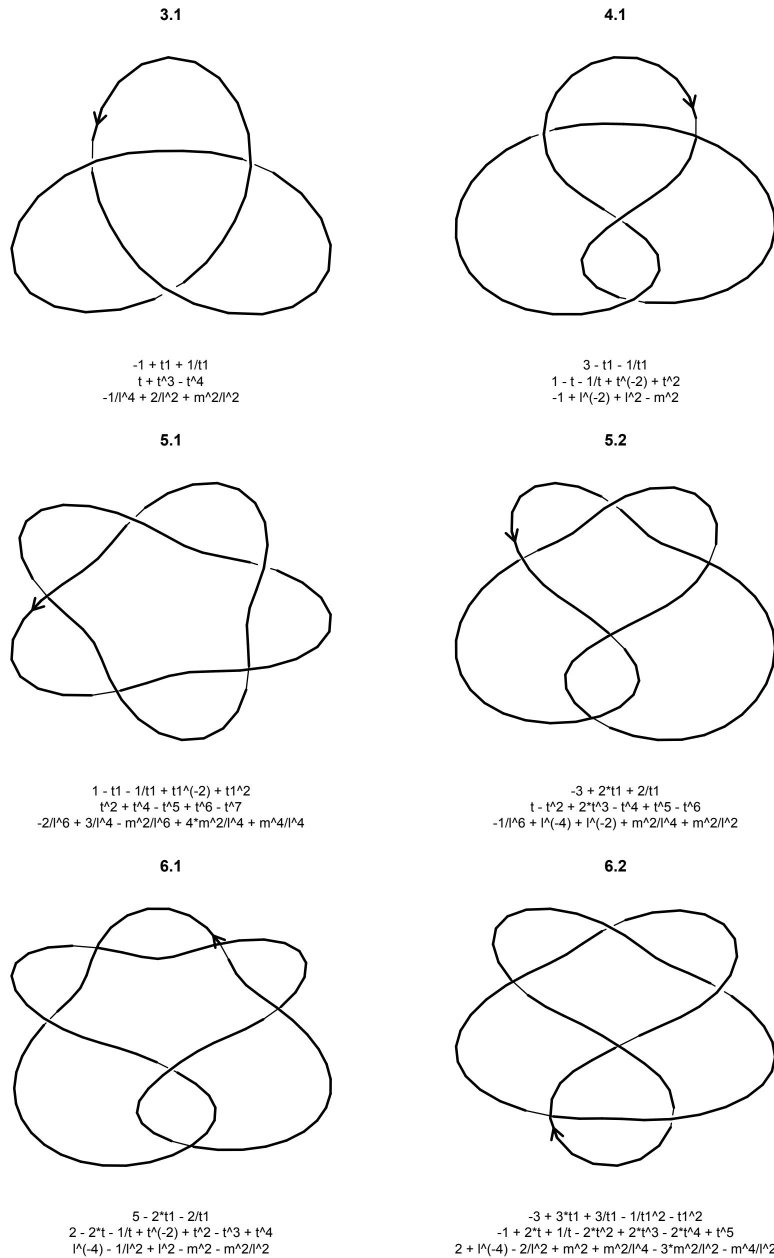
For some applications, the linking number of a link is desired. The function `linkingNumber` in *Rknots* computes the linking number of a polygonal link simply by

```
> ( computeInvariant( link.cls.AB, invariant = 'LK') )
```

```
[1] -3
```

Before moving to the last session, let's make use of what we have described so far by reproducing a modern version of the very beginning of the original Rolfsen knot table.

```
> data(Rolfsen.table)
> text <- names(Rolfsen.table)[1 : 6]
> par(mfrow = c(3,2))
> for(i in 1 : 6) {
+   k <- Rolfsen.table[[i]]
+   k <- newKnot(k)
+   plot(k, lwd = 2, main = text[i],
+        sub = paste( computeInvariant(k, 'Alexander'),
+                     computeInvariant(k, 'Jones'),
+                     computeInvariant(k, 'HOMFLY'), sep = '\n'))
+ }
```



2.3 *Rknots* and protein knot analysis, a dedicated pipeline

Rknots provides an optimized pipeline for detecting and characterizing knots in proteins and more generally in biopolymers. Two example `.pdb` files are part of the package data and will be partially used in the following case study.

1. the Rds3p protein (PDB identifier `2K0A`), a member of the U2 snRNP essential for pre-mRNA splicing, that has a left-handed trefoil knotted structure [5]

2. The first chain (A) of the E. Coli alkaline phosphatase (D153G mutant), as an example of protein presenting a structural gap, potential source of false positives.

A protein can be loaded from the file system or fetched from the Protein Data Bank (citare PDB) using the `loadProtein` function. This function returns a list of matrices, where each element contains the 3D coordinates of a given protein chain. By default, `loadProtein` performs gap finding for each chain backbone, with a parameter `cutoff` that allow a custom definition of a gap. `cutoff` represents the maximum allowed euclidean distance between two consecutive alpha-Carbon residues. If a distance is greater than `cutoff`, the chain is split in the corresponding position. The resulting subchains inherit the label from the chain that have been split, with an additional consecutive number for each of the subchain (e.g. if chain A is split into three subchains, they will be labeled A_1, A_2 and A_3).

```
> #from the file system
> protein <- loadProtein(system.file("extdata/2k0a.pdb", package="Rknots"))
```

```
[1] "PDB has multiple END/ENDMDL records"
[1] "multi=FALSE: taking first record only"
  HEADER    METAL BINDING PROTEIN                         31-JAN-08   2K0A
Summary of the distance vector
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  3.744   3.789   3.799   3.800   3.810   3.856
No gap found
```

```
> protein<- loadProtein('2K0A') #from the PDB
```

```
  Note: Accessing online PDB file
[1] "PDB has multiple END/ENDMDL records"
[1] "multi=FALSE: taking first record only"
  HEADER    METAL BINDING PROTEIN                         31-JAN-08   2K0A
Summary of the distance vector
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  3.744   3.789   3.799   3.800   3.810   3.856
No gap found
```

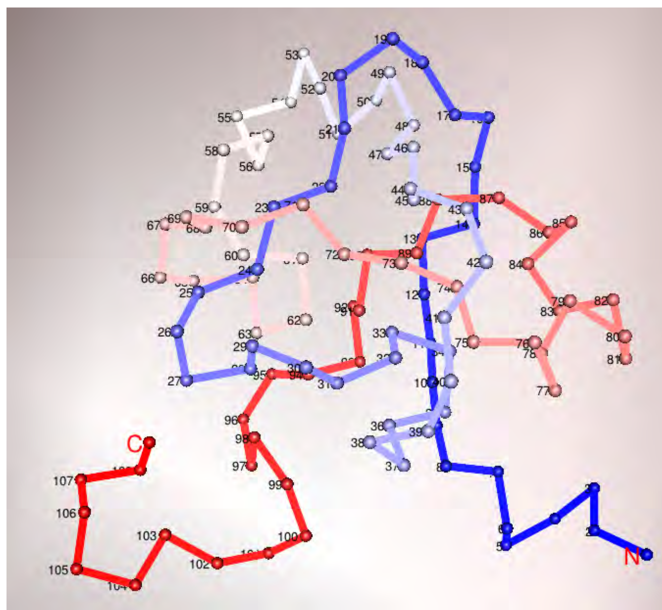
```
> str(protein)
```

```
List of 1
 $ A: num [1:109, 1:3] -9.22 -9.93 -8.36 -11.83 -14.64 ...
```

`loadProtein` can be called with any additional parameter of the `read.pdb` function of *bio3d*. See the relevant manual for details.
At this point, we may wish to visualize the 3D structure of the imported protein and the corresponding backbone model. For this purpose, we can use the function `plot3D` and exploit the superb graphics of *rgl* [6]. Particularly, we can supply parameters for the functions `lines3d` and `spheres3d`. Briefly, the code below will produce the desired plot. A snapshot is shown below. First, we prepare a color palette for each residue that will be passed as a list to the low-level function `plotKnot3D`. Its wrapper for an object of class *Knot* is simply `plot3D` (notice the capital D) and requires the same parameters as `plotKnot3D`.

```
> ramp <- colorRamp(c('blue', 'white', 'red'))
> pal <- rgb( ramp(seq(0, 1, length = 109)), max = 255)
> plotKnot3D(protein$A, colors = list( pal ),
+          lwd = 8, radius = .4, showNC = TRUE, text = TRUE)
```

The variable `showNC` labels the N-terminus and the C-terminus of the protein, whereas `text` add to each point the corresponding residue number.



Although possible due to a fully customizable representation, to prepare high-quality figures for publications we recommend to export the structure using `write.pdb` and load the result into dedicated software for protein structure visualization.

To find knots in proteins, a single subunit has to be supplied and coerced to a *Knot* class object. The available chains are simply the names of the list returned by `loadProtein`

```
> names(protein)
```

```
[1] "A"
```

and the coercion can be simply achieved through:

```
> protein <- newKnot(protein$A)
> protein
```

```
An object of class 'Knot'
Slot points3D:  109 x 3 matrix
             x       y      z
 [1,]   -9.225 -24.265 -3.881
 [2,]   -9.927 -20.804 -5.217
 [3,]   -8.363 -19.745 -8.522
 [4,]  -11.829 -18.718 -9.673
 [5,]  -14.645 -16.690 -8.082
 [6,]  -12.639 -15.926 -4.953
 [7,]  -10.105 -13.650 -6.562
 [8,]  -10.944 -10.485 -4.591
 [9,]   -9.794  -8.744 -7.783
[10,]   -6.918  -7.069 -5.969
```

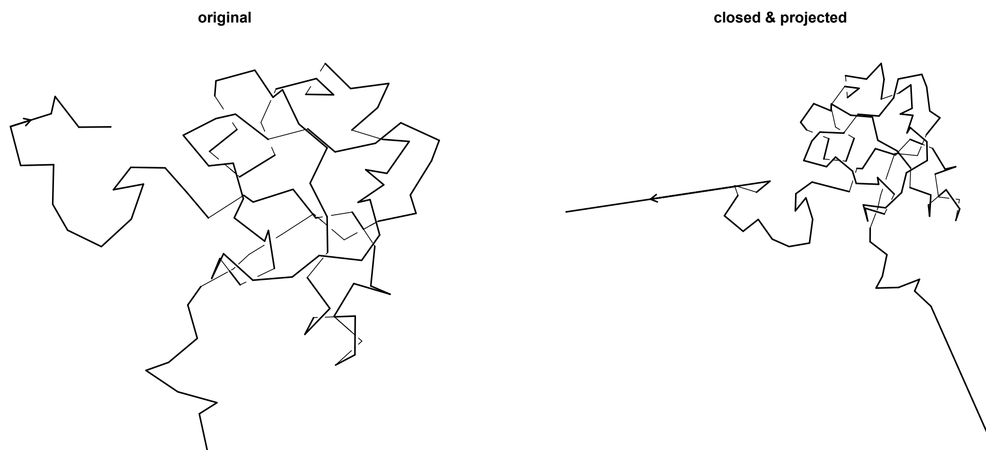
```
        ........  ........  ........
Slot ends:
```

After that, the structure has to be closed with the function `closeAndProject`, which also applies a Principal Component Analysis on the closed structure mostly to lead to optimized graphical representations and to minimize the possibility of numerical problems during the computation of the HOMFLY polynomial.

```
> protein.cp <- closeAndProject( protein, w = 2 )
```

where `w` controls the extension of the endpoints (we are using the default value in this example and generally, any value greater than 1 will do). Let's plot the two protein diagram for comparisons

```
> par(mfrow = c(1,2))
> plot(protein, main = 'original', lwd = 2)
> plot(protein.cp, main = 'closed & projected', lwd = 2)
```



Now, we can make use of what we already know for computing the polynomial invariants of this protein:

```
> ( delta.p <- computeInvariant( protein.cp, invariant = 'Alexander') )

[1] "-1 + t1 + 1/t1"

> ( jones.p <- computeInvariant( protein.cp, invariant = 'Jones', skein.sign = -1) )

[1] "1/t - 1/t^4 + t^(-3)"

> ( homfly.p <- computeInvariant( protein.cp, invariant = 'HOMFLY', skein.sign = -1) )

[1] "2*l^2 + l^2*m^2 - l^4"
```

As expected, the Alexander polynomial only tells us that we are dealing with a trefoil knot, but we can make use of the HOMFLY polynomial to establish the knot chirality. If you need support to determine the knot type, the function `getKnotType` does the characterization for you. The polynomial as returned by `computeInvariant`, the polynomial type and an optional additional parameter controlling the information to be returned are required as an input. For example we can characterize the knot type of the current protein with

```
> getKnotType(polynomial = delta.p, invariant = 'Alexander')
```

```
[1] "Left-handed Trefoil knot (3_1*)" "Right-handed Trefoil knot (3_1)"
```

```
> getKnotType(polynomial = homfly.p, invariant = 'HOMFLY')
```

```
[1] "Left-handed Trefoil knot (3_1*)"
```

```
> getKnotType(polynomial = homfly.p, invariant = 'HOMFLY', full.output = TRUE)
```

```
                        Knot.type                              HOMFLY
"Left-handed Trefoil knot (3_1*)"            "2*l^2 + l^2*m^2 - l^4"
                            Jones                           Alexander
            "1/t - 1/t^4 + t^(-3)"                 "-1 + t1 + 1/t1"
                  link.Knot.Atlas
     "http://katlas.org/wiki/3_1"
```

It turns out that the Rds3p protein has a left-handed knot that we can compare with the mirror image of the polynomial of the right-handed trefoil knot in the Rolfsen knot table by means of another *Rknots* utility:

```
> trefoil <- Rolfsen.table[[1]]
> trefoil <- newKnot(trefoil)
> ( homfly.tr <- computeInvariant(trefoil, 'HOMFLY') )
```

```
[1] "-1/l^4 + 2/l^2 + m^2/l^2"
```

```
> ( homfly.tl <- HOMFLY2mirror(homfly.tr) )
```

```
[1] "2*l^2 + l^2*m^2 - l^4"
```

```
> identical( homfly.p, homfly.tl )
```

```
[1] TRUE
```

Finally, notice that if a protein has more than one chain, to iterate over all the possible chains we can make use of `lapply`, as follows. First let's define two very simple functions. The first one can be used for processing a single chain and its very simple code summarizes what we have done so far, for example with the Rds3p protein above. The second function simply returns the length of a given chain.

```
> processChain <- function(protein, i) {
+         chain <- newKnot(protein[[i]])
+         chain <- closeAndProject( chain )
+         return( computeInvariant(chain, 'HOMFLY') )
+ }
> lengthChain <- function(protein, i) return( nrow(protein[[i]]))
```

Then, we will fetch from the PDB a protein having 2 chains (and we will perform gap finding) and we will compute for example the HOMFLY polynomial of each of them. Finally, we will merge the resulting list in a dataframe to obtain a handy report.

```
> protein <- loadProtein('1AJC', join.gaps = FALSE, cutoff = 7)

  Note: Accessing online PDB file
  HEADER    NON SPECIFIC MONO-ESTERASE              18-JUL-95   1AJC
Loading chains:
  #aminoacids
A         446
B         449
Summary of the distance vector
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  3.756   3.793   3.805   3.822   3.817  11.040
Chain split at position
 404
Summary of the distance vector
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  3.741   3.793   3.805   3.806   3.817   3.864
No gap found

> str(protein)

List of 3
 $ A1: num [1:404, 1:3] 66.9 68.4 67.3 67.4 63.9 ...
 $ A2: num [1:42, 1:3] 67.8 67.3 66.5 69.4 70.4 ...
 $ B : num [1:449, 1:3] 73.6 74.6 71.9 71.3 73.8 ...

> chains <- names(protein)
> polynomials <- lapply( 1: length(chains) ,
+                 function(i) {
+                         ifelse(lengthChain(protein, i) > 6, processChain(protein, i), 1) } )
> cbind(chains, polynomials)

     chains polynomials
[1,] "A1"   "-1/l^4 + 2/l^2 + m^2/l^2"
[2,] "A2"   "1"
[3,] "B"    "1"
```

As we can see, the first chain of this protein has been split and resulted in a two unknotted subchains. The second chain instead bears a right-handed trefoil knot. As a final remark, notice that if we would have ignored the gap finding, we would have not found a knot in the first chain.

```
> protein <- loadProtein('1AJC', join.gaps = TRUE)

  Note: Accessing online PDB file
  HEADER    NON SPECIFIC MONO-ESTERASE              18-JUL-95   1AJC
Loading chains:
  #aminoacids
A         446
B         449
```

```
> str(protein)

List of 2
 $ A: num [1:446, 1:3] 66.9 68.4 67.3 67.4 63.9 ...
 $ B: num [1:449, 1:3] 73.6 74.6 71.9 71.3 73.8 ...

> chains <- names(protein)
> polynomials <- lapply( 1: length(chains) ,
+                 function(i) {
+                         ifelse(lengthChain(protein, i) > 6, processChain(protein, i), 1) } )
> cbind(chains, polynomials)

     chains polynomials
[1,] "A"    "1"
[2,] "B"    "1"
```

3 Session Info

```
> sessionInfo()

R version 2.14.1 (2011-12-22)
Platform: x86_64-apple-darwin9.8.0/x86_64 (64-bit)

locale:
[1] en_US/en_US/en_US/C/en_US/en_US

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

other attached packages:
[1] bio3d_1.1-3    Rknots_1.2.1   rSymPy_0.2-1.1 rJython_0.0-2  rjson_0.2.6
[6] rJava_0.9-3    rgl_0.92.798

loaded via a namespace (and not attached):
[1] tools_2.14.1
```

References

[1] Comoglio F., Rinaldi M., A Topological Framework for the Computation of the HOMFLY Polynomial and Its Application to Proteins. *PLoS ONE*, 6(4), e18693

[2] The Rolfsen Knot Table on Knot Atlas, `http://www.math.toronto.edu/~drorbn/KAtlas/Knots`

[3] Reidemeister K. (1926), Abh Math Sem Univ Hamburg 5: 24-32.

[4] Alexander J.W. and Briggs G.B. (1926) On types of knotted curves. *Ann of Math*, 28, 562-586.

[5] van Roon A.M., Loening N.M., Obayashi E., Yang J.C., Newman A.J., Hernandez H., Nagai K. and Neuhaus D., (2008) Solution structure of the U2 snRNP protein Rds3p reveals a knotted zinc-finger motif, *Proc Natl Acad Sci USA*, 105, 9621-9626.

[6] Adler D., Murdoch D., rgl: 3D visualization device system (OpenGL). R package version 0.92.798