Introduction to `turner`

Gaston Sanchez
www.gastonsanchez.com

1 Introduction and Motivation

`turner` is an R package designed to provide a set of handy functions for manipulating vectors and lists of vectors. The main idea is to make it easier to **turn** vectors (and lists of vectors) into other data R structures.

1.1 Why `turner`?

The package `turner` was born out of necessity from my involvement with **Multiblock Methods** and other multivariate data analysis methods (eg PLS Path Modeling). Although `turner` is intended to be used as a lower-level package (i.e. for developing other packages), I hope that you may find it useful for your own computations.

1.2 A little bit of multiblocks

R is great for working with data in tabular format such as matrices and data frames. However, there's no data structure for representing the abstract concept of a *multiblock*. Basically, a multiblock could be seen as a matrix divided by blocks (or submatrices). This is a very informal and simplistic description, but it helps to understand the notion of a multiblock.

So how can we work with multiblocks in R? The trivial solution is to work with several matrices (one matrix per block). Another solution is to work with arrays. A third solution is to work with lists of matrices.

A different approach —the one I use— is to work parallelly with one matrix (or a data frame) and one list. In this case, all the blocks are in a single matrix (or data frame), while the list contains the information about the blocks. The main advantage of this approach is that you keep the data in one single object, while the relevant information about the structure of the blocks is kept in one list.

If we decide to work with the matrix-list duet, we need to be able to *extract* the information of the list, and **turn** it into indixed structures (or other objects) for manipulating the blocks in the data matrix. `turner` is my attempt to make it easier (at least for me) to perform such manipulations.

2 Indexification

To use `turner` (once you have installed it), simply load it with the function `library()`:

```
# load package turner
library(turner)
```

Data in Blocks

To see how we can apply `turner`, we need to consider some data under a multiblock perspective. First let's start by creating a data matrix with 10 observations and 9 variables.

```
# create a matrix
set.seed = 21
some_data = round(matrix(rnorm(90), 10, 9), 3)
rownames(some_data) = 1:10
colnames(some_data) = paste("X", 1:9, sep = "")

# take a peek
head(some_data, n = 5)

##       X1     X2     X3     X4     X5     X6     X7     X8     X9
## 1 -0.774  0.370  0.544  1.798 -0.347  1.046 -0.798 -0.627 -0.566
## 2  1.841 -0.929  2.208 -0.113  1.840  0.922  1.712  1.180  0.853
## 3 -1.660 -2.218  0.296 -0.839 -1.846 -0.335 -0.544  0.184  1.622
## 4 -0.828 -0.219  1.593  0.825 -0.951 -1.144  0.137 -0.330  0.423
## 5 -0.086  0.090 -0.098 -0.060 -1.043 -0.329 -0.651  1.139  0.841
```

Now, let's suppose that our data can be divided in 3 blocks. The first block is formed by variables `X1, X2, X3`. The second block is formed by variables `X4, X5`. And the third block is formed by variables `X6, X7, X8, X9`. All this information can be stored in a list:

```
# list of blocks
blocks = list(B1 = 1:3, B2 = 4:5, B3 = 6:9)
blocks

## $B1
## [1] 1 2 3
##
## $B2
## [1] 4 5
##
## $B3
## [1] 6 7 8 9
```

Indexed Structures

`turner` has been designed to work with lists (preferable of vectors) in order to turn them into *indexed structures*. Such structures are mostly vectors that map the position indices of the elements in the list.

`indexify()`

One common task is to **indexify** the list of blocks. The idea is to get a vector of indices representing the membership of the variables to their corresponding block. This is better understood with the following example:

```
# get indices of blocks
indices = indexify(blocks)
indices

## [1] 1 1 1 2 2 3 3 3 3
```

The *indexification* of `blocks` allows us to get an indexed vector `indices`. This vector contains as many elements as variables in `some_data`. Moreover, it tells us that: the first three elements belong to one block, the fourth and fifth elements belong to block 2, and the rest of the elements belong to block 3.

`list_to_dummy()`

Another interesting task is to produce a dummy matrix based on the blocks. This is done by using the funciton `list_to_dummy()`:

```
# get dummy matrix based on blocks
dummy = list_to_dummy(blocks)
dummy

##       [,1] [,2] [,3]
##  [1,]    1    0    0
##  [2,]    1    0    0
##  [3,]    1    0    0
##  [4,]    0    1    0
##  [5,]    0    1    0
##  [6,]    0    0    1
##  [7,]    0    0    1
##  [8,]    0    0    1
##  [9,]    0    0    1
```

As you can tell, `dummy` is matrix with as many rows as elements in blocks, and with as many columns as number of blocks. In turn, the columns of `dummy` are dummy indicators (hence the name).

`from_to()`

Sometimes, it is also useful to know the starting and ending positions of the blocks. This can be done by using the function `from_to()`.

```
# get starting and ending positions
start_end = from_to(blocks)
start_end

## $from
## B1 B2 B3
##  1  4  6
##
## $to
## B1 B2 B3
##  3  5  9


# vectors from and to
from = start_end$from
to = start_end$to
```

`from_to()` provides a list with two vectors: `$from` and `$to`. The first vector `$from` contains the indices of the starting positions; in turne the second vector `$to` contains the indices of the ending positions:
We can extract the first block in `some_data`:

```
# extract first block
some_data[, from[1]:to[1]]

##        X1     X2     X3
## 1  -0.774  0.370  0.544
## 2   1.841 -0.929  2.208
## 3  -1.660 -2.218  0.296
## 4  -0.828 -0.219  1.593
## 5  -0.086  0.090 -0.098
## 6   0.304 -0.220 -0.857
## 7  -0.443  0.250 -0.065
## 8   0.401  0.467  0.482
## 9   0.892  1.833 -0.428
## 10  1.115 -0.105 -1.684
```

Obviously we can argue that there is no need to use `from` and `to`. We can extract the first block by just typing:

```
# get first block
some_data[, blocks[[1]]]

##        X1     X2     X3
## 1  -0.774  0.370  0.544
## 2   1.841 -0.929  2.208
## 3  -1.660 -2.218  0.296
## 4  -0.828 -0.219  1.593
## 5  -0.086  0.090 -0.098
## 6   0.304 -0.220 -0.857
## 7  -0.443  0.250 -0.065
## 8   0.401  0.467  0.482
## 9   0.892  1.833 -0.428
## 10  1.115 -0.105 -1.684
```

Yes, we can use `blocks` to manipulate the data. But the advantage of `from_to()` comes when you work with string lists like the following one:

```
# string list
str_list = list(c("a", "b", "c"), c("d", "e"), c("f", "g", "h", "i"))
```

In this case you cannot extract the first block by simply typing:

```
# failed attempt
some_data[, str_list[[1]]]
```

You solve this problem by using `from_to()`:

```
# start-end position for 'str_list'
fromto_aux = from_to(str_list)
from1 = fromto_aux$from
to1 = fromto_aux$to

# successful attempt
some_data[, from1[1]:to1[1]]

##        X1     X2     X3
## 1  -0.774  0.370  0.544
## 2   1.841 -0.929  2.208
## 3  -1.660 -2.218  0.296
## 4  -0.828 -0.219  1.593
## 5  -0.086  0.090 -0.098
## 6   0.304 -0.220 -0.857
## 7  -0.443  0.250 -0.065
## 8   0.401  0.467  0.482
## 9   0.892  1.833 -0.428
## 10  1.115 -0.105 -1.684
```

2.1 Working with lists

Among other interesting features of `turner` is the set of functions for working with lists of vectors.

lengths()

The function `lengths()` allows us to get the length of each vector inside a list. This function has an argument `out` to specify whether the output is in vector format (default behavior) or in list format:

```
# say you have some list
some_list = list(1:3, 4:5, 6:9)

# length of each vector (vector output)
lengths(some_list, out = "vector")

## [1] 3 2 4


# length of each vector (list output)
lengths(some_list, out = "list")

## [[1]]
## [1] 3
##
## [[2]]
## [1] 2
##
## [[3]]
## [1] 4
```

It is important not to confuse `lengths()` with `length()`; the latter only gives you the number of elements in the list:

```
# compared to 'length()'
length(some_list)

## [1] 3
```

funlist()

Another interesting function in `turner` is `funlist()` which takes as arguments a list and a function. The purpose of `funlist()` is to apply the given function to the unlisted elements in the list:

```
# sum of all elements in 'some_list'
funlist(some_list, sum)

## [1] 45


# maixmum of all elements in 'some_list'
funlist(some_list, max)

## [1] 9


# product of all elements in 'some_list'
funlist(some_list, prod)

## [1] 362880


# mean value of all elements in 'some_list'
funlist(some_list, mean)

## [1] 5
```

listsize()

The function `listsize()` —as well as `sizelist()`— is another related function to work with lists. It allows us to get the total number of elements contained in a list:

```
# number of elements in 'some_list'
listsize(some_list)

## [1] 9
```

indexify()

Another handy function is `listify()` which creates a list from a vector of integers:

```
# vector of indices
number_elements = c(3, 1, 5)

# list of index vectors based on 'number_elements'
listify(number_elements)

## [[1]]
## [1] 1 1 1
##
## [[2]]
## [1] 2
##
## [[3]]
## [1] 3 3 3 3 3
```

2.2 Other Functions

The following table shows other functions available in `turner`:

Function	Description
`df_to_blocks()`	splits a data frame into blocks
`matrix_to_blocks()`	splits a matrix into blocks
`vector_to_dummy()`	creates a dummy matrix from the elements in a vector
`factor_to_dummy()`	creates a dummy matrix from the elements in a factor
`list_to_dummy()`	creates a dummy matrix from the elements in a list
`dummy_to_list()`	creates an indexed list from a dummy matrix
`list_to_matrix()`	creates a design-type matrix from the elements in a list