Project Comments

Stefano Ghirlanda

February 21, 2017

I see that you have switched to using `data.table` and the `apply` function to aggregate data. That's great, but `data.table` has many more features that can simplify our work greatly. I'll explain below.

Using `data.table` can help us fix some issue with your current code:

1. We will not have always four files for each face. So we need an approach that does not hard-code the number of files for each face. This is very easy using `data.table`.
2. Using `data.table`, the calculations of `sd` and other quantities can be streamlined even more. `data.table` is also coded in C, very efficiently, and it is *very* fast.
3. Landmark files will normally not be in the current directory. We should have a variable such as `data.dir` and then use `paste0( data.dir, "/*.xlsx" )` to generate the string that is passed to `list.files`. (This point is not related to `data.table`.)

Below I will guide you to solve these problems. I am not going to write the code for you :) but I will tell you how to use `data.table`, and I will provide some examples and pseudo-code. The idea is that, after studying a bit what I write below, you should be able to write more concise and more generalizable code using `data.table`. You goal is to solve points 1–3 above. Please let me know if you have any questions!

(As an aside: you don't need to generate a separate results file for each face, but we can talk about this later.)

A note on separation of function

In your current code, you read and process one file at a time, but I think it is cleaner to split the code into two separate stages, data acquisition and data processing. Pseudo-code for data acquisition can be as follows:

```
landmarks <- NULL # data structure, initially empty
for( file in file.list ) {
    ## 1. read file
    ## 2. append file to landmarks
}
```

For step 1, just use `read_excel` as you are doing already. Say that the result is in variable `x`. For step 2, use the function `rbind` (bind by row) to glue together `x` and `landmarks`:

```r
## step 2. in the loop above:
landmarks <- rbind( landmarks, x )
```

The loop then produces a `landmarks` data structure that we can convert to `data.table` for further processing:

```r
landmarks <- data.table( landmarks )
```

At this point you may ask: but now how do I know which face a data point refers to? The answer is that the landmarks file have been designed with this kind of data processing in mind. Each line in a landmark file includes information about which point and which face it belongs to, and about who marked the point. Using this information, `data.table` can do quite a bit of magic for us.

Data table summary

Introduction

- A regular `data.frame` is naturally indexed as a 2D array: if `x` is a `data.frame`, then `x[i,j]` is the elemnt in row *i* and column *j*.
- In other words, the `data.frame` operator `[]` is used for indexing.
- The `data.table` operator `[]`, on the other hand, is redefined to also enable computing on the contents of the `data.table`.
- This operator can take 3 arguments, rather than 2, that are usually called (from first to last):
  - the `i` argument
  - the `j` argument
  - the `by` argument

We can explain these arguments with a few examples. In the following, you will see R code and, right below, the results it produces. Consider the following data assumed to be in a `data.frame` called `d`:

Person	Sex	Age
John	M	20
Jack	M	22
Ann	F	21
Sue	F	30

We first convert to `data.table`:

```r
library(data.table)
d <- data.table( data )
```

Use the first argument to index:

You can index using the `i` argument. For example, you can index by `Sex`:

```
d[ Sex=="F" ]
```

```
   Person Sex Age
1:    Ann   F  21
2:    Sue   F  30
```

Or you can index by age:

```
d[ Age>21 ]
```

```
   Person Sex Age
1:   Jack   M  22
2:    Sue   F  30
```

Note that the variables `Sex` and `Age` are automatically in scope (if `d` had been a simple `data.frame`, you would have had to write `d[ d$Sex=="F" ]` etc.).

Use the second argument to perform computations:

You can perform computations on the data using the `j` argument:

```
d[ Sex=="M", mean(Age) ]
```

```
[1] 21
```

And you can even perform multiple computations, using the syntax `.()` to build lists of results:

```
d[ Sex=="M", .( mean(Age), sd(Age) ) ]
```

```
   V1       V2
1: 21 1.414214
```

The result is a `data.table`, in which the results of your computation have been assigned names `V1` and `V2` (you can rename these, an example is below).

You can perform computation on the whole `data.table` by leaving the `i` argument empty. The number of females can be calculated like this:

```
d[, sum( Sex=="F" ) ]
```

```
[1] 2
```

Use the third argument to group data

What if we want to calculate the mean age separately by sex? We can do this in a single call using the `by` argument:

```
d[, mean(Age), by=Sex ]
```

```
   Sex   V1
1:   M 21.0
2:   F 25.5
```

The result is another `data.table` with `Sex` as one column and the result of the computation automatically named `V1`. You can give a better name as follows:

```
d2 <- d[, mean(Age), by=Sex ]
setnames( d2, "V1", "meanAge" )
d2
```

```
   Sex meanAge
1:   M    21.0
2:   F    25.5
```

You can also group by multiple variables. Suppose you have this other `data.frame`:

Person	Sex	Age	Weight
John	M	25	150
Jack	M	22	170
Ann	F	21	140
Sue	F	25	145
Al	M	22	180
Lucy	F	21	160

You can calculate mean weight by age and sex as follows:

```
d[, mean(Weight), by=.(Age,Sex) ]
```

```
   Age Sex  V1
1:  25   M 150
2:  22   M 175
3:  21   F 150
4:  25   F 145
```

Finally, note that the `by` argument can itself contain computations. Suppose we want to split age in “old” and “young” using a cutoff of 23. We can do it simply like this:

```r
d[ , mean(Weight), by=Age<23 ]
```

```
     Age    V1
1: FALSE 147.5
2:  TRUE 162.5
```

Note that the grouping variable `Age` retains its name, but its value is the result of the computation in `by` rather then original age value.