Programming, just like other fields using engineering methods and thinking, is about making choices, and trading off between certain aspects. A simple example is the fairly well-known trade-off between memory use and speed: think *e.g.* of a hash map allowing for faster lookup at the cost of some more memory. Generally speaking, solutions are rarely limited to just one way, or just one approach. So if pays off to know your tools, and choose wisely among *all* available options. Having choices is having options, and those tend to have non-negative premiums to take advantage off. Locking yourself into one and just one paradigm can never be better.

In that spirit, I want to (eventually) show a few *simple* comparisons of code being done two distinct ways.

But then this week I found a much simpler and shorter example, and quickly converted its code. The code comes from the inaugural [datascience 1](http://www.thecrosstab.com/datascience/r-1/) lesson at the [Crosstab](http://www.thecrosstab.com/), a fabulous site by G. Elliot Morris (who may be the highest-energy undergrad I have come across lately) focusssed on political polling, forecasts, and election outcomes. Lesson 1 is a simple introduction, and averages some polls of the 2016 US Presidential Election.

**Complete Code using Approach "TV"**

Elliot does a fine job [walking the reader through his code](http://www.thecrosstab.com/datascience/r-1/) so I will be brief and simply quote it in one piece:

## Getting the polls

library(readr)

polls\_2016 <- read\_tsv(url("http://elections.huffingtonpost.com/pollster/api/v2/questions/16-US-Pres-GE%20TrumpvClinton/poll-responses-clean.tsv"))

## Wrangling the polls

library(dplyr)

polls\_2016 <- polls\_2016 %>%

filter(sample\_subpopulation %in% c("Adults","Likely Voters","Registered Voters"))

library(lubridate)

polls\_2016 <- polls\_2016 %>%

mutate(end\_date = ymd(end\_date))

polls\_2016 <- polls\_2016 %>%

right\_join(data.frame(end\_date = seq.Date(min(polls\_2016$end\_date),

max(polls\_2016$end\_date), by="days")))

## Average the polls

polls\_2016 <- polls\_2016 %>%

group\_by(end\_date) %>%

summarise(Clinton = mean(Clinton),

Trump = mean(Trump))

library(zoo)

rolling\_average <- polls\_2016 %>%

mutate(Clinton.Margin = Clinton-Trump,

Clinton.Avg = rollapply(Clinton.Margin,width=14,

FUN=function(x){mean(x, na.rm=TRUE)},

by=1, partial=TRUE, fill=NA, align="right"))

library(ggplot2)

ggplot(rolling\_average)+

geom\_line(aes(x=end\_date,y=Clinton.Avg),col="blue") +

geom\_point(aes(x=end\_date,y=Clinton.Margin))

It uses five packages to i) read some data off them interwebs, ii) then filters / subsets / modifies it leading to a [right (outer) join](https://en.wikipedia.org/wiki/Join_(SQL)#Right_outer_join) with itself before iv) averaging per-day polls first and then creates rolling averages over 14 days before v) plotting. Several standard *verbs* are used: filter(), mutate(), right\_join(), group\_by(), and summarise(). One non-verse function is rollapply() which comes from [zoo](https://cran.r-project.org/web/packages/zoo/index.html), a popular package for time-series data.

**Complete Code using Approach "DT"**

As I will show below, we can do the same with fewer packages as [data.table](https://github.com/Rdatatable/data.table/wiki) covers the reading, slicing/dicing and time conversion. We still need [zoo](https://cran.r-project.org/web/packages/zoo/index.html) for its rollapply() and of course the same plotting code:

## Getting the polls

library(data.table)

pollsDT <- fread("http://elections.huffingtonpost.com/pollster/api/v2/questions/16-US-Pres-GE%20TrumpvClinton/poll-responses-clean.tsv")

## Wrangling the polls

pollsDT <- pollsDT[sample\_subpopulation %in% c("Adults","Likely Voters","Registered Voters"), ]

pollsDT[, end\_date := as.IDate(end\_date)]

pollsDT <- pollsDT[ data.table(end\_date = seq(min(pollsDT[,end\_date]),

max(pollsDT[,end\_date]), by="days")), on="end\_date"]

## Average the polls

library(zoo)

pollsDT <- pollsDT[, .(Clinton=mean(Clinton), Trump=mean(Trump)), by=end\_date]

pollsDT[, Clinton.Margin := Clinton-Trump]

pollsDT[, Clinton.Avg := rollapply(Clinton.Margin, width=14,

FUN=function(x){mean(x, na.rm=TRUE)},

by=1, partial=TRUE, fill=NA, align="right")]

library(ggplot2)

ggplot(pollsDT) +

geom\_line(aes(x=end\_date,y=Clinton.Avg),col="blue") +

geom\_point(aes(x=end\_date,y=Clinton.Margin))

This uses several of the components of data.table which are often called [i, j, by=...]. Row are selected (i), columns are either modified (via := assignment) or summarised (via =), and grouping is undertaken by by=.... The outer join is done by having a data.table object indexed by another, and is pretty standard too. That allows us to do all transformations in three lines. We then create per-day average by grouping by day, compute the margin and construct its rolling average as before. The resulting chart is, unsurprisingly, the same.

**Benchmark Reading**

We can looking how the two approaches do on getting data read into our session. For simplicity, we will read a local file to keep the (fixed) download aspect out of it:

R> url <- "http://elections.huffingtonpost.com/pollster/api/v2/questions/16-US-Pres-GE%20TrumpvClinton/poll-responses-clean.tsv"

R> download.file(url, destfile=file, quiet=TRUE)

R> file <- "/tmp/poll-responses-clean.tsv"

R> res <- microbenchmark(tidy=suppressMessages(readr::read\_tsv(file)),

+ dt=data.table::fread(file, showProgress=FALSE))

R> res

Unit: milliseconds

expr min lq mean median uq max neval

tidy 6.67777 6.83458 7.13434 6.98484 7.25831 9.27452 100

dt 1.98890 2.04457 2.37916 2.08261 2.14040 28.86885 100

R>

That is a clear relative difference, though the absolute amount of time is not that relevant for such a small (demo) dataset.

**Benchmark Processing**

We can also look at the processing part:

R> rdin <- suppressMessages(readr::read\_tsv(file))

R> dtin <- data.table::fread(file, showProgress=FALSE)

R>

R> library(dplyr)

R> library(lubridate)

R> library(zoo)

R>

R> transformTV <- function(polls\_2016=rdin) {

+ polls\_2016 <- polls\_2016 %>%

+ filter(sample\_subpopulation %in% c("Adults","Likely Voters","Registered Voters"))

+ polls\_2016 <- polls\_2016 %>%

+ mutate(end\_date = ymd(end\_date))

+ polls\_2016 <- polls\_2016 %>%

+ right\_join(data.frame(end\_date = seq.Date(min(polls\_2016$end\_date),

+ max(polls\_2016$end\_date), by="days")))

+ polls\_2016 <- polls\_2016 %>%

+ group\_by(end\_date) %>%

+ summarise(Clinton = mean(Clinton),

+ Trump = mean(Trump))

+

+ rolling\_average <- polls\_2016 %>%

+ mutate(Clinton.Margin = Clinton-Trump,

+ Clinton.Avg = rollapply(Clinton.Margin,width=14,

+ FUN=function(x){mean(x, na.rm=TRUE)},

+ by=1, partial=TRUE, fill=NA, align="right"))

+ }

R>

R> transformDT <- function(dtin) {

+ pollsDT <- copy(dtin) ## extra work to protect from reference semantics for benchmark

+ pollsDT <- pollsDT[sample\_subpopulation %in% c("Adults","Likely Voters","Registered Voters"), ]

+ pollsDT[, end\_date := as.IDate(end\_date)]

+ pollsDT <- pollsDT[ data.table(end\_date = seq(min(pollsDT[,end\_date]),

+ max(pollsDT[,end\_date]), by="days")), on="end\_date"]

+ pollsDT <- pollsDT[, .(Clinton=mean(Clinton), Trump=mean(Trump)),

+ by=end\_date][, Clinton.Margin := Clinton-Trump]

+ pollsDT[, Clinton.Avg := rollapply(Clinton.Margin, width=14,

+ FUN=function(x){mean(x, na.rm=TRUE)},

+ by=1, partial=TRUE, fill=NA, align="right")]

+ }

R>

R> res <- microbenchmark(tidy=suppressMessages(transformTV(rdin)),

+ dt=transformDT(dtin))

R> res

Unit: milliseconds

expr min lq mean median uq max neval

tidy 12.54723 13.18643 15.29676 13.73418 14.71008 104.5754 100

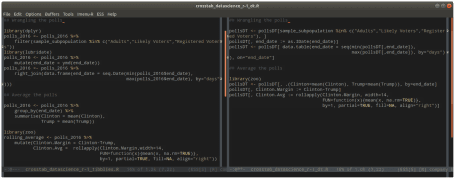
dt 7.66842 8.02404 8.60915 8.29984 8.72071 17.7818 100

R>

Not quite a factor of two on the small data set, but again a clear advantage. data.table has a reputation for doing really well for large datasets; here we see that it is also faster for small datasets.

**Side-by-side**

Stripping the reading, as well as the plotting both of which are about the same, we can compare the essential data operations.



**Summary**

We found a simple task solved using code and packages from an increasingly popular sub-culture within R, and contrasted it with a second approach. We find the second approach to i) have fewer dependencies, ii) less code, and iii) running faster.

Now, undoubtedly the former approach will have its staunch defenders (and that is all good and well, after all choice is good and even thirty years later some still debate vi versus emacs endlessly) but I thought it to be instructive to at least to be able to make an informed comparison.