Reproducible Data Analysis Workflow Modules Wheat CAP 2018

Jean-Luc Jannink

USDA-ARS

June 17, 2018

Document explanation

There are two kinds of narrative text in this document. On the one hand, I try to explain how to document an Rmarkdown script On the other hand, there is documentation of the actual analyses that the script is doing, exemplifying useful documentation…

Learning objectives

1. An idea:

i. A reproducible analysis workflow includes a predictable input file and data structure, and outputs that are described, interpreted, and put in the context of the overall project in English.

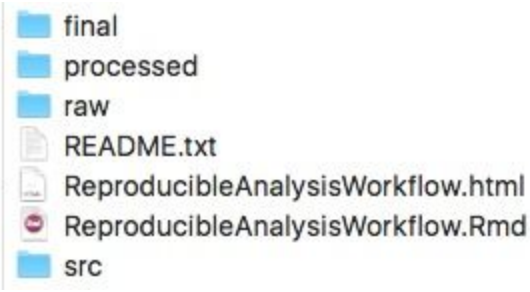
ii. The audience of this workflow is akin to someone who might be reviewing a manuscript derived from the work. The **most important audience is yourself**, six months later, or your close collaborators (e.g., lab group) who may need to carry on the work after you move on.

2. A tool:

i. R markdown language. It allows you to mingle script with formatted text, with script outputs. Note that Python and c++ scripts can be incorporated into R markdown.

ii. Jupyter notebook. I am not proficient, so will not describe, but if you are interested in this area, I encourage you to look into it.

iii. It can help to find what you need to have a standard directory file structure.



Standard script directory organization

For future reference

3. I think R markdown is very good for documenting **scripts**, but less so **programs**. If code is linear, single-purpose, and fairly simple, I call it a script. If code has loops, is potentially multi-purpose, and defines functions, I call it a program.

4. If you write a series of functions that you will use repeatedly, it's probably worth making a package out of them. That is not trivial, but it's less difficult than it sounds. You do not have to submit your package to **CRAN**, but can just use it internally. The documentation of functions that goes along with making a package is very helpful over time.

5. If you write a program that you imagine will develop over time, learn version control, here or here. Note that a public repository like github can be quite useful for making your data available once you publish your research.
6. Here are some useful articles:

i. Ten Simple Rules for Reproducible Computational Research

ii. Good enough practices in scientific computing

Single Marker QTL Mapping

This script executes a simple simulation of single marker QTL mapping. It starts from data downloaded from The Triticeae Toolbox of genotypes of a spring wheat association mapping panel. This data is loaded into a simulation environment of the BreedingSchemeLanguage. 50 QTL are simulated. Doubled haploid lines are derived from the SWAMP. Two DHs, polymorphic at a large QTL are crossed to form a RIL population that is advanced to the F5 stage. The RILs are genotyped and phenotyped and the -log_10(p-values) for markers and QTL are plotted. There is reference to making HIFs in the script, but that will be for another day.

Loading packages

If your script depends on external packages, load them at the beginning. This shows users early

on what the script dependencies are.

```
ip <- installed.packages()

l2e_installed <- "latex2exp" %in% rownames(ip)
if (!l2e_installed){
  stop("ERROR: you need to install the latex2exp package")
} else{
  library(latex2exp)
}

BSL_installed <- "BreedingSchemeLanguage" %in% rownames(ip)
if (!BSL_installed){
  stop("ERROR: you need to install the BreedingSchemeLanguage
package")
} else{
  library(BreedingSchemeLanguage)
}

tidyverse_installed <- "tidyverse" %in% rownames(ip)
if (!tidyverse_installed){
  stop("ERROR: you need to install the tidyverse package")
} else{
  library(tidyverse)
  library(readxl)
}
## ── Attaching packages
─────────────────────────────────────────────────────────────────────
tidyverse 1.2.1 ──
## ✔ ggplot2 2.2.1     ✔ purrr   0.2.5
## ✔ tibble  1.4.2     ✔ dplyr   0.7.5
## ✔ tidyr   0.8.1     ✔ stringr 1.3.1
## ✔ readr   1.1.1     ✔ forcats 0.3.0
## ── Conflicts
─────────────────────────────────────────────────────────────────────
─── tidyverse_conflicts() ──
## ✖ purrr::cross()  masks BreedingSchemeLanguage::cross()
## ✖ dplyr::filter() masks stats::filter()
## ✖ dplyr::lag()    masks stats::lag()
## ✖ dplyr::select() masks BreedingSchemeLanguage::select()
```

A first thing to notice from the output of loading tidyverse is the conflicts report. In particular, for example, two packages have been loaded that have a function called select. One is the BreedingSchemeLanguage package (BSL henceforth), and the other is the dplyr package. Since dplyr was loaded *after* BSL, if you use the function select, it will go to the dplyr version by default. But it's dangerous to rely on what order packages have been loaded to determine which select function you get. R syntax to prevent ambiguity is to write either dplyr::select or BreedingSchemeLanguage::select. Using that syntax will make your code more reproducible.

Document packages

Here, I am also saving the **versions** of the packages for future reference.

```
packages_used <- ip[c("BreedingSchemeLanguage", "latex2exp",
"tidyverse", "readxl"), c("Package", "Version", "Built")]
readme_file <- "README.txt"
write_lines(c("Single Marker QTL Mapping", date(), "The packages used
in this script are:"), readme_file)
write_tsv(as.tibble(packages_used), readme_file, append=T)
```

Go ahead and hyperlink the README file to the report. That makes is easy to find.

Set random seed

The BSL generates many random numbers (e.g., to simulate Mendelian random segregation). If you want the result of the analysis to come out the same each time (there are pros and cons) you need to set the random seed.

```
random_seed <- 45678
set.seed(random_seed)
write_lines(paste("The random seed is", random_seed), readme_file,
append=T)
```

Script parameters

If the behavior of your script depends on parameters that you set, initialize them early on.

```
init_pop_size <- 200
qtl_mapping_pop_size <- 150
num_qtl <- 50
hif_fam_size <- 60
```

Executable descriptions

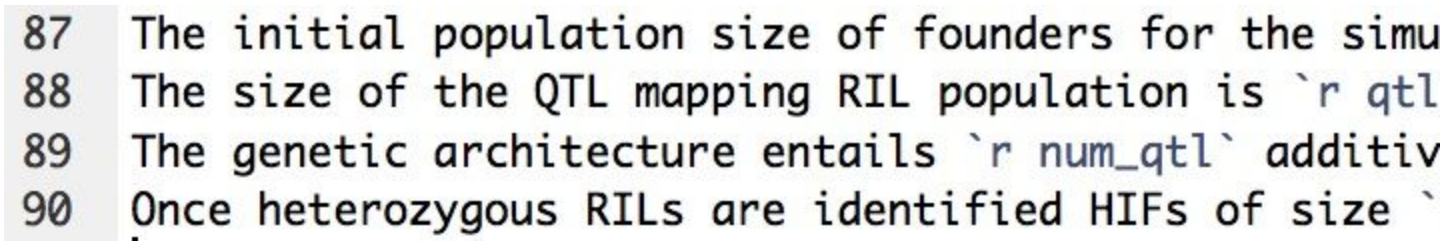
It is easy for text descriptions around your code to become out of sync with the code. Because the descriptions are ignored by the computer during execution, they may say one thing while the script says another. That becomes confusing later when you revisit the script. One way to avoid that is “executable descriptions”. For example, rather than setting parameters in a chunk of code, you can set them “inline” in R markdown:

The initial population size of founders for the simulation is 200.

The size of the QTL mapping RIL population is 150.

The genetic architecture entails 50 additive effect QTL.

Once heterozygous RILs are identified HIFs of size 60 will be made.



The R markdown for that looks like this

Parameters to README

This is starting to look **VERY** redundant, but it can't hurt to have the information in multiple places. Chunks of code like this do not need to be included in the report. To exclude them use the "include=FALSE" option in the chunk.

```
write_lines(c(
  paste("The initial population size of founders for the simulation
is", init_pop_size),
  paste("The size of the QTL mapping RIL population is",
qtl_mapping_pop_size),
  paste("The genetic architecture entails additive effect QTL
numbering", num_qtl),
  paste("With heterozygous RILs, HIFs will be made of size",
hif_fam_size)), readme_file, append=T)
```

Modify genotype output from T3

The genotype file is going to be input to the BreedingSchemeLanguage. BSL requires recombination map positions in cM. T3 gives an integer position because that's what TASSEL wants. T3 does that by multiplying the cM position by 1000. So divide by 1000 to get cM positions. Remove markers that don't have a map position (chrom == "UNK").

```
temp <- read_tsv("raw/genotype.hmp.txt")
temp <- temp[temp$chrom != "UNK",]
temp$pos <- temp$pos/1000 # Convert positions to cM
write_tsv(temp, path="processed/genoPos.hmp.txt")
```

BreedingSchemeLanguage Sim.

1. If a previous simulation was performed delete it. It's an R memory management issue.

2. Create the based population and then a set of doubled haploids.

i. Note that nQTL=50 is the default for defineSpecies. However, it's often best to be explicit in case defaults change, or just to jog your memory.

```
if (exists("simEnv")){
  rm(list=names(simEnv), envir=simEnv)
  rm(simEnv)
}
simEnv <-
BreedingSchemeLanguage::defineSpecies(importFounderHap="processed/geno
Pos.hmp.txt", saveDataFileName="processed/springWheatBSL",
nQTL=num_qtl)
BreedingSchemeLanguage::initializePopulation(nInd=init_pop_size) #
popID 0
BreedingSchemeLanguage::doubledHaploid(nProgeny=init_pop_size) # popID
1
```

Ensure QTL polymorphic

If a QTL was simulated at a locus with very low MAF, one of its alleles may have been lost in generating the DH population, in which case we would not want to construct the mapping population around it. The details here require knowing a substantial amount about the BSL.

```
sim <- BreedingSchemeLanguage::outputResults(summarize=F)[[1]]
qRank <- order(abs(sim$mapData$effects), decreasing=T)
which_ind_dh <- which(sim$genoRec$popID==1)
for (qtl in sim$mapData$effectivePos[qRank]){
  if (sd(sim$geno[which_ind_dh * 2, qtl]) > 0) break
}
```

Create RIL population

Figuring out which two parents to cross is opaque. After that, four generations of single seed descent are simulated in a straightforward way.

```
par1 <- which(sim$geno[which_ind_dh * 2, qtl] == -1)[1]
par2 <- which(sim$geno[which_ind_dh * 2, qtl] == 1)[1]
```

```
pedigree <- matrix(c(which_ind_dh[par1], which_ind_dh[par2], 1), 1, 3)
BreedingSchemeLanguage::cross(pedigree=pedigree) # popID 2 is F1
BreedingSchemeLanguage::selfFertilize(nProgeny=qtl_mapping_pop_size) #
popID 3 is F2
BreedingSchemeLanguage::selfFertilize(nProgeny=qtl_mapping_pop_size) #
popID 4 is F3
BreedingSchemeLanguage::selfFertilize(nProgeny=qtl_mapping_pop_size) #
popID 5 is F4
BreedingSchemeLanguage::selfFertilize(nProgeny=qtl_mapping_pop_size) #
popID 6 is F5
BreedingSchemeLanguage::genotype(popID=6)
BreedingSchemeLanguage::phenotype()
```

P-values of single marker tests

You have to know the BSL to extract genotypes. Then identify all polymorphic loci (including QTL) and run them through a simple linear model. Create a tibble that has the locus index and the p-value.

```
# Extract genotypes of phenotyped individuals
sim <- BreedingSchemeLanguage::outputResults(summarize=F)[[1]]
gid <- sim$phenoRec$phenoGID
pheno <- sim$phenoRec$pValue
simHaplo <- sim$geno[sort(c(gid*2 - 1, gid*2)),]
simGeno <- (diag(qtl_mapping_pop_size) %x% matrix(1, nrow=1, ncol=2))
%*% simHaplo / 2

# Find polymorphic markers
loc_poly <- which(apply(simGeno, 2, sd) > 0)

# Function returns the p-value for a simple single marker test
calc_loc_pval <- function(locus){
  fitMrk <- lm(pheno ~ simGeno[,locus])
  return(-log10(anova(fitMrk)["Pr(>F)"][1,]))
}
minus_log_pval <- sapply(loc_poly, calc_loc_pval)
names(minus_log_pval) <- loc_poly
qtl_pval_data <- tibble(lp=loc_poly, pv=minus_log_pval)
```

Make a plot

Often, you will see that the largest effect QTL do not generate the lowest p-values. Spend some time prettying up the plot and so save it to the final results folder. Also save the raw data that went into the plot: if this was an extensive analysis, but you just want to make cosmetic

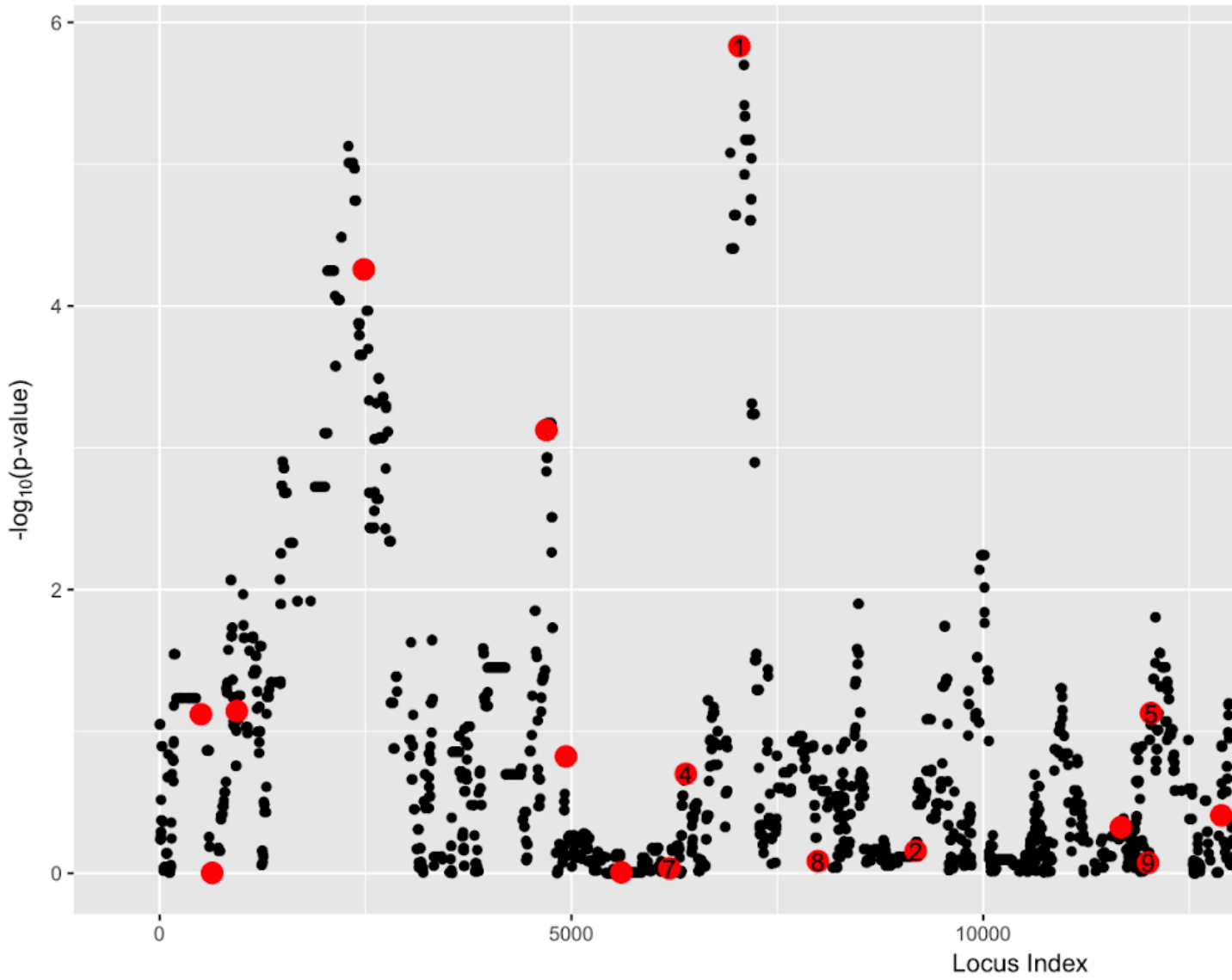
changes, you can go to that data directly.

```
qtl_this_pop <- sim$mapData$effectivePos[qRank]
qtl_this_pop <- qtl_this_pop[qtl_this_pop %in% loc_poly]
ordered_qtl <- tibble(lp=qtl_this_pop,
pv=minus_log_pval[as.character(qtl_this_pop)])

qtl_plot <- ggplot(qtl_pval_data) +
  geom_point(mapping=aes(lp, pv)) +
  geom_point(mapping=aes(lp, pv), data=ordered_qtl, size=4,
color="red") +
  geom_text(mapping=aes(lp, pv, label=1:9), data=ordered_qtl[1:9,]) +
  labs(x="Locus Index", y=TeX("-log_{10}(p-value)"))

# Save the final plot and associated information
saveRDS(qtl_plot, "final/plot_pvalues_markers_qtl.rds")
saveRDS(list(qtl_pval_data=qtl_pval_data, ordered_qtl=ordered_qtl),
"final/data_pvalues_markers_qtl.rds")

print(qtl_plot)
```


6
4
2
0
-log10(p-value)
0
5000
10000
Locus Index
1
2
4
5
7
8
9