R programming

José R. Valverde
jrvalverde@cnb.csic.es
CNB/CSIC

Pangenome analysis

(An agony in 8 fits)

2024

*Ignotum per ignotius**

**The meaning of this is Unknown* PRINCIPIA DISCORDIA
or
HOW I FOUND THE GODDESS & WHAT I DID TO HER
WHEN I FOUND HER
Malaclypse the younger

The challenge

- We have 4000+ genomes with various characteristics
- We want to know if there is a difference in recombination between groups for each of these.

Preparation

- We align the genomes using a suitable tool (Mauve, Cactus, Gubbins, Mugsy, Parsnp, Sibelia-Z, Snippy, ...)
- We build a phylogenetic tree (Fasttree, Raxml, IQ-Tree, ...)
- We analyze recombination events (ClonalFrame, Gubbins, ...)

Data

- To start with, we have
  - an annotated reference genome ('reference.gff3')
  - a metadata file to classify the genomes according to properties ('metadata.tsv')
- Phylogeny calculation gave us a tree ('labelled_final.tree')
- The recombination predictions ('recombination.predictions.gff')

Install RCandy

- Leave this running:

```
install.packages(
    c("ape", "dplyr", "graphics", "grDevices",
      "magrittr", "phytools","shape", "stats",
      "stringr", "tibble", "tidyr", "utils",
      "viridis", "knitr", "rmarkdown",
      "markdown", "devtools", "abind",
      "abind", "e1701"),
    dependencies=TRUE
)
install_github("ChrispinChaguza/RCandy")
```

Analysis tools

- We will need a tool to analyze the data:
  - www.phandango.net
    - Drag the data over the page (use metadata.**csv**)
    - That's cool! but is it signficant?
  - https://github.com/ChrispinChaguza/RCandy
    - To check in R we need to interpret our data

Fit the 1st: **constants aren't**

- You will often feel tempted to think of some values as constants.
- **Constants aren't!**
- You should always try to use meaningful names instead.
  - Practice will tell you.

Input data

- We will first assign the input data files to variables so we can refer to them by name:

```
tree.file <- "labelled_final.tree"
gubbins.gff <- "recombination.predictions.gff"
ref.genome.gff <- "reference.gff3"
metadata.file <- "metadata.tsv"

taxa.groups <- c("prop.1", "prop.2", "prop.3",
                 "prop.4", "prop.5", "prop.6")
```

Graphics!

- Now we will try to visualize our data
- Let us start having a look at the tree and groups

```
RCandyVis(
    tree.file.name=tree.file,
    midpoint.root=TRUE,
    ladderize.tree.right=TRUE,
    taxon.metadata.file=metadata.file,
    taxon.metadata.columns=taxa.groups)
```

Fit the 2nd: **Save, save, save...**

- You should be collecting all your commands in a script as you try them
- But when tasks are heavy, it is savvy to save as well the data you are generating.

More graphics

- Let us add the reference genome and recombination events predicted

```
# this wills save successive plots to new pages
pdf("plots.pdf",
    paper="a4", width=8, height=11)
# this will only save the last plot drawn
png("last.RCandy.plot", width=1024, height=1024)

RCandyVis(tree.file.name=tree.file,
        midpoint.root=TRUE,
        ladderize.tree.right=TRUE,
        taxon.metadata.file=metadata.file,
        taxon.metadata.columns=taxa.groups,
        gubbins.gff.file=gubbins.gff,
        ref.genome.name=ref.genome.gff)

dev.off() # this closes only the last open device (PNG file)
```

Saving plots

- You can also tell RCandy to save the plot to a specific file instead of sending it to the screen (and/or any other open devices)
- This will create the named PDF file, but will send **nothing** to any other devices (screen, png, pdf, whatever...)

```
RCandyVis(tree.file.name=tree.file,
        midpoint.root=TRUE,
        ladderize.tree.right=TRUE,
        taxon.metadata.file=metadata.file,
        taxon.metadata.columns=taxa.groups,
        gubbins.gff.file=gubbins.gff,
        ref.genome.name=ref.genome.gff,
        save.to.this.file="recombination_plot.pdf")
```

Fit the 3rd: **comments are not only comments**

- As we have done it, the last graphics went to files, not to the screen
  - We could have open/closed an additional device (dev.X11(), dev.cairo()...)
- You can comment out code lines to test your code, and uncomment them for production

Beautify it

- RCandyVis can take a large number of parameters to allow you customize what is shown and how.
- Let us try to improve visualization of recombination events using transparency
  - The trick here is to use RGBα instead of RGB color specifications.
  - R allows you to use any (it tells by the length of the hexadecimal color string)

```
RCandyVis(tree.file.name=tree.file,
        midpoint.root=TRUE,
        ladderize.tree.right=TRUE,
        taxon.metadata.file=metadata.file,
        taxon.metadata.columns=taxa.groups,
        gubbins.gff.file=gubbins.gff,
        ref.genome.name=ref.genome.gff,
        rec.heatmap.color=c("#FF000022", "#0000FF66"))
```

Visualize

- Now you can open the PDF file (at last!)
  - But we went blind.
- Comment out the 'pdf.dev' and the last 'dev.off' lines and run your script again
  - You should see now what it does
- When satisfied, uncomment these lines to get the output actually saved.
- Drawing takes long, add an "if" test to only draw when needed/desired (e.g. if (draw) { ... })

Fit the 4th: **Having utility functions is nice**

- You should try to create functions for tasks that you will use often
  - In your scripts
  - And not just in this script
- You should also try to create functions for conceptual tasks
  - even if you will only use them once
  - to keep your code conceptually clean
  - e.g. "visualize()", "analyze()", "analyze.group()"...
- You can put utility functions in a separate file and load them with "`source()`" at the start of your scripts or when you need them.

Assignment

- You can check if a file exists with function "`file.exists()`"
- You can copy files with "`file.copy()`"
  - Get help on both functions
- Create a `make.backup()` function that gets a filename and
  - creates a numbered backup file name
  - checks if new, numbered backup file exists
  - if it does, increments the number and checks again
  - if it does not exist, makes a copy of the file with the new backup name

Try before continuing

A possible solution

```
make.backup <- function(file) {
    MAXBACKUP <- 1000
    i = 0
    while (i < MAXBACKUP) {
        # since we only add a comma and a number,
        # any path will also be preserved
        bck <- paste(file, i, sep=',')
        #cat("checking if", bck, "exists\n")
        if (! file.exists(bck) ) break;
        #cat(bck, "exists, trying a higher number\n"
        i <- i + 1
    }
    # what will happen if there are already MAXBACKUP files?
    file.copy(file, bck)
}
```

Did we say save?

- Let's get on with the analysis
- To apply statistics we need data

```
tree <- read.tree.file(tree.file)
meta.data <- load.taxon.metadata(metadata.file)
ref.genome.GFF <- load.genome.GFF(ref.genome.gff)
rec.data <- load.gubbins.GFF(gubbins.gff,
                recom.input.type = "Gubbins")
rec.freq <- count.rec.events.per.base(gubbins.gff,
                recom.input.type="Gubbins")
rec.genome <- count.rec.events.per.genome(gubbins.gff,
                recom.input.type="Gubbins",
                taxon.names=tree$tip.label)
```

Fit the 5th: **don't be afraid of ^C**

- At some point, calculations will become onerous (may take hours).
  - You can press ^C to stop the calculation.
  - You will be returned to R.
  - Yet another reason to save everything:
    - Wait for data to be processed, save it and next time load the processed data instead of repeating the processing.
- We already saved this data for you in directory "R":
  - "R/R.rec.freq.Rds"
  - "R/R.rec.genome.Rds"

Assignment

- Modify your script so that
  - it checks if there is already a file with processed data saved
  - if there is, then loads the saved data
  - if there is not, then it processes the original file and saves the processed data as an RDS file.
- Don't forget to add explanatory comments

Try before continuing

```
tree <- read.tree.file(tree.file)
meta.data <- load.taxon.metadata(metadata.file)
ref.genome.GFF <- load.genome.GFF(ref.genome.gff)
rec.data <- load.gubbins.GFF(gubbins.gff,
                recom.input.type = "Gubbins")
# check if there is a saved R dataset to avoid recomputation
if (file.exists("Rdata/R.rec.freq.Rds" )) {
    rec.freq <- readRDS("Rdata/R.rec.freq.Rds")
} else {
    # if it doesn't, read and analyze the data, and save it for later
    rec.freq <- count.rec.events.per.base(gubbins.gff,
                recom.input.type="Gubbins")
    saveRDS(rec.freq, file="Rdata/R.rec.freq.Rds")
}

if (file.exists("Rdata/R.rec.genome.Rds")) {
    rec.genome <- readRDS("Rdata/R.rec.genome.Rds")
} else {
    rec.genome <- count.rec.events.per.genome(gubbins.gff,
                recom.input.type="Gubbins",
                taxon.names=tree$tip.label)
    saveRDS(rec.genome, file="Rdata/R.rec.genome.Rds")
}
```

Prepare the data sets

```
genome.recombinations <- data.frame(rec=rec.genome)
colnames(genome.recombinations) <- c('genome',
    'rec.Freq')
head(genome.recombinations)
```

...

We now face a decision: we have several groups that we may wish to analyze.

We can prepare all the groups first, or we can do them in a loop.

In the last case, we may do it from scratch or we can try to do one group manually first, then wrap it all inside a function, and then wrap it all inside a loop.

Here, again, using names instead of magic constants will help us expedite development.

Fit the 6th: **Aim for generality**

- There are many ways to skin a cat (or so the saying goes).
- Try to think ahead and, when you foresee repeating a task, look for a more general approach
  - The approach that will make it easier to generalize.
  - And that is easier to understand

Take a decision

- Selecting one group:

```
in.group.1 <- meta.data$group.1 == 'Y'
```

- Doing it more general:
  - head(meta.data)
  - Notice that groups are in columns 4:9, labeled prop.#

```
propnumber <- 3 + 1
in.group <- meta.data[, propnumber] == 'Y'

propname <- paste('prop', 1, sep='.')
in.group <- meta.data[ , propname] == 'Y'
```

Making a decision

- The first approach is immediate, but less general
- Using an index would allow us later to run a for loop and compute the column number.
  - for (i in 1:6) propnumber <- 2 + i
- Using the name would allow us to compute the name independently of in which column it is
  - for (i in 1:6) propnumber <- paste(prop, i, sep='.')
- But for now we don't care much (yet)

more decisions...

- Next we need to consider how are we going to use the data: we want to have, for a given group
  - genome names
  - frequencies
  - whether each genome belongs or not in the group

Prepare the data (2)

```
# from here we will be repeating for each group
propname <- paste('prop', 1, '.')
in.group <- meta.data[ , propname] == "Y"

# get the names of genomes in the group, whose value is "Y"
group.names <- meta.data$ID[in.group]

# get which genome recombinations belong to the group chosen
genome.rec.in.group <- genome.recombinations$genome %in%
                       names.in.group

# extract the names used by Gubbins for genomes in the group chosen
names.in.group <-
as.character(genome.recombinations$genome[genome.rec.in.group])

# extract the frequenciesv as well
freqs.in.group <-
genome.recombinations$rec.Freq[genome.rec.in.group]
```

Prepare the data (3)

```
# get data of genomes not in group
names.not.in.group <- as.character(
        genome.recombinations$genome[ ! genome.rec.in.group])
freqs.not.in.group <-
        genome.recombinations$rec.Freq[ ! genome.rec.in.group]

# join name, frequencies and their pertenence to the group
freq.data <- data.frame(
        genome=c(names.in.group, names.not.in.group),
        freqs=c(freqs.in.group, freqs.not.in.group),
        in.group=c(rep('Y', length(freqs.in.group)),
             rep('N', length(freqs.not.in.group ))
            )
        )
```

Statistics...

- Think in advance: we'll analyze several groups, it'd be nice to know which is which
- We'll get the summary data and conduct parametricity tests for
  - normality
  - homocedasticity
- And we'll draw some plots

Get basic statistics

```
summary(freq.data$freqs ~ freq.res$in.group)
shapiro.test(freqs.in.group)
shapiro.test(freqs.not.in.group)
bartlett.test(freq.data$freqs ~ freq.res$in.group)

# plot the freqs as reference and this data plot
hist(freqs)
Boxplot(reqs)
par(mfrow=c(2,1))
hist(freqs.in.group)
hist(freqs.not.in.group)
par(mfrow=c(1,2))
Boxplot(freqs.in.group)
Boxplot(freqs.not.in.group)
par(mfrow=c(1,1))
```

decisions, decisions, decisions...

- Or not?
- We should make informed decisions, but we are using a computer...
- We could just compute everything and decide later...
- But **we will have to decide anyway**

Fit the 7th: KISS (Keep It Simple, Stupid!)

- Ockham's razor (14th C): "entities should not be multiplied beyond necessity"
- If you flood users (even if it is just yourself) with lots of data, it is very easy to be misled into believing or misunderstanding it.
  - If you do things too complex they become more difficult to understand and fix.
  - Try to make everything self-explanatory

doing statistics

```
# Out of lazyness, we'll compute both,
# parametric and non-parametric tests.
# This way, if t-test fails we already have
# Wilconxon's/Mann-Whitney's U and we do not
# need to repeat the calculation.

t.test(freq.data$freqs ~ freq.data$in.group,
    alternative='two.sided', conf.level=.95,
    var.equal=FALSE)

wilcox.test(freq.data$freqs ~ freq.data$in.group,
    alternative='two.sided')$p.value
```

Simplify and explain

```
cat("\n\nSUMMARY\n")
cat('---------------------------------------------\n')
cat('\np < 0.05 => significant difference\n\n')
cat("    Welch T-test (2 sided) =",
    t.test(freq.data$freqs ~ freq.data$in.group,
    alternative='two.sided', conf.level=.95,
    var.equal=FALSE)$p.value,
    '\n')
cat("    Mann-Whitney U (2 sided) =",
    wilcox.test(freq.data$freqs ~ freq.data$in.group,
    alternative='two.sided')$p.value,
    '\n')
```

Fit the 8th: Beware the *boojum*

- "There's only one life-form as intelligent as me within thirty parsecs of here and that's me." (Marvin, the paranoid android. HHGTG. *D. Adams*)
- "Anything that can go wrong will go wrong" (Murphy's law)
- "ὅτι ἃ μὴ οἶδα οὐδὲ οἴομαι εἰδέναι" ("What I do not know, I do not think I know, either", Socrates, in Plato's Apology)

We all make mistakes

- If any calculation is important, we need to make sure it goes well.
- But sooner or later it will fail.
- That is why we have the "tryCatch" function.
  - help(tryCatch)
  - demo(error.catching)

tryCatch

- What it means is simple: we pass tryCatch some text to interpret as an R command.
  - tryCatch will **try** to run the command
  - If all goes well, the command will have been run
  - If anything goes wrong, tryCatch will **catch** the problem and let us know indicate what to do:
    - if the command produced a warning, tryCatch will execute the **warning** argument and return its result
    - if the command produced and error, tryCatch will execute the **error** argument and return its result
  - In any case, at the end it will execute the **finally** argument,

An example

```
log(2.7182818284)  # OK
log(-1)# Warning: not a number
log("one") # Error: your script dies here!

tryCatch(log(2.7182818284))
tryCatch(log(-1))
tryCatch(log("one"))
```

- The last three lines will catch any problem and the script will not die. But we will not know either.

Somewhat better

- warning and error take as argument a function. This function will be called with the exception that arose.
  - We may chose to ignore the exception:

```
w <- function(excep) print("There was a warning")
e <- function(excep) print("There was an error")

# now we will know that there was an
# exceptional situation, but not which
# exception it was (only it was a warning or
# and error
tryCatch(log(-1), warning=w, error=e)
tryCatch(log("one"), warning=w, error=e)
```

Best if we know

```
w <- function(excep)
    cat(paste("NOTE!", excep), '\n')
e <- function(excep)
    cat(paste("IMPORTANT", excep, '\n'))

# this time, we will know that there was
# and exceptional situation, if it was a
# warning or an error, and what kind of
# warning or error it was.
tryCatch(log(-1), warning=w, error=e)
tryCatch(log("one"), warning=w, error=e)
```

A reminder about statements

- tryCatch can execute a single command.
- But remember: we can group many commands with { and } and then they will be considered as one.
- Thus, we can use

```
tryCatch(
    {
        log(1)
        log(-1)
        log('one')
    }, warning=w, error=e)
```

Assignment

- Define two new functions, one to handle warnings and one to handle errors
- Encapsulate the statistics code within brackets
- Call the statistics code with tryCatch.

Try before continuing

A possible solution

```
tryCatch( {
    cat("    Welch T-test (2 sided) =",
        t.test(freq.data$freqs ~ freq.data$in.group,
        alternative='two.sided', conf.level=.95,
        var.equal=FALSE)$p.value,
        '\n')
    cat("    Mann-Whitney U (2 sided) =",
        wilcox.test(freq.data$freqs ~ freq.data$in.group,
        alternative='two.sided')$p.value,
        '\n')
    },
    warning=w,
    error=e)
```

Saving output

- As we start producing results, we'll reach a point where one screen is not enough.
- We can save all output to a file.
- **sink(file=**xxx**)** will send all screen output to a file
- **sink()** will stop saving output to a file
- insert "**sink('analysis.log', split=TRUE)**" at the beginning of your script to see output on the screen <u>and</u> save it to a file named 'analysis.log'.
- add "**sink()**" at the end of your script to stop saving:

```
...    ...    ...
dev.off()    # this one closes the PDF device
cat("\nOutput saved to 'analysis.log' and 'plots.pdf'\n\n")
sink()
```

The journey starts here

- Now, we can enclose everything from the comment "***`# from here we will be repeating for each group`***" until here in { } and write a for loop to go over all the properties, one by one.
- You may want to insert a command to indicate which property you are analyzing (e.g. "**`cat('Processing, propname,'\n')`**" after computing "propname".
- You may want to add more statistics in the tryCatch, e.g. unilateral t-test and Wilcoxon tests.

The only limit is your imagination

- You may also want to enclose the for loop in { } and write another loop over it, so you can chose to analyze not just the frequencies (freq.data$freqs), but also normalized and log-transformed frequencies.
- And on and on...

And that is not the end yet

- We have analyzed the frequencies
- Many papers report other data
  - saved in a "**gubbins.per_branch_statistics.csv**" file.
- You may read this CSV file and go over each of the columns in it, repeating the work we did for frequencies.
- And on and on...

Refactoring

- Is how we call the process of taking a program or script and rearranging it
  - hopefully for the best
- Look at your script and see if there are chunks of code that can be improved or made as functions
- Can variables use better names?
- Can you add more comments?
- And on and on...

The moral

- Just get started and take it step by step, you will eventually get there.

- Just get started and take it step by step, you will eventually get there.

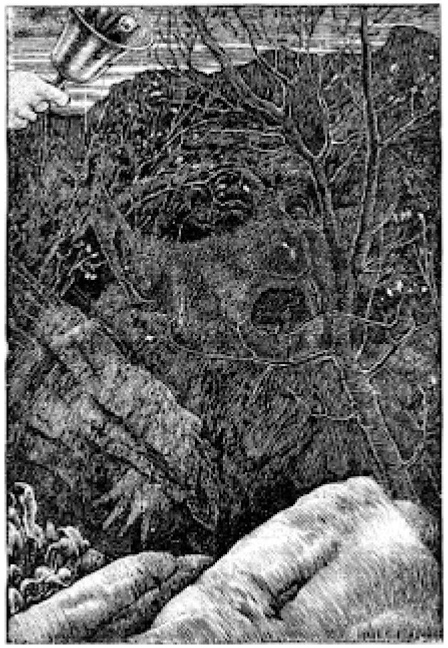
- Just get started and take it step by step, you will eventually get there.

I have said it thrice:

What I tell you three times is true.

(the Bellman, in "The Hunting of the Snark", *Lewis Carroll*).

Questions?



For the Snark *was* a Boojum, you see.
Lewis Carroll
The Hunting of the Snark