Before we begin

On a desktop

1. Login with your McGill credentials.
2. Open RStudio
3. Install the packages and download the data.

On your laptop

Install the packages and download the data.

Packages and data

`packagesData.R` available at `goo.gl/FJtdXu`.

HGSS Workshop : Analysis and Visualization of Large Genomics Data in R

Jean Monlong

Human Genetics department

March 28, 2017

Today's topic

- Reading large genomics data.
- Analyzing large genomics data.
- Visualizing large genomics data.

Let's get started !

1. Open R/Rstudio or whatever you use.
2. Prepare a folder for the workshop and set it as working directory.

Disclaimer

- Some things might be technical. Follow what you can.
- Feel free to interrupt or suggest other ways.
- No need to type/do everything.
- You can follow the live script.

Today's packages

Installation

- Using `install.packages` for CRAN packages.
- Using `biocLite` for Bioconductor packages.

Run this

```
install.packages(c("data.table","dplyr","ggplot2","parallel",
  "BatchJobs", "parallel", "magrittr"))
source("http://bioconductor.org/biocLite.R")
biocLite(c("GenomicRanges","Rsamtools", "VariantAnnotation",
  "rtracklayer", "Gviz"))
```

Shared folder

Use the script in the shared Google Drive folder: `goo.gl/FJtdXu`.

Today's data

Get the data

```
download.file("ftp://ftp.sanger.ac.uk/pub/gencode/Gencode_human/release_19/
    gencode.v19.annotation.gtf.gz", "gencode.gtf.gz")

download.file("ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/release/20130502/ALL
    .chr22.phase3_shapeit2_mvncall_integrated_v5a.20130502.genotypes.vcf.gz"
    , "tgp22.vcf.gz")
download.file("ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/release/20130502/ALL
    .chr22.phase3_shapeit2_mvncall_integrated_v5a.20130502.genotypes.vcf.gz.
    tbi", "tgp22.vcf.gz.tbi")
```

Gencode file

- Human gene reference annotation: genes, exons, transcipts, ...
- More than 2 million lines.
- See GTF format.

1000 Genomes Project

- Variants: SNPs, indels, structural variants.
- Chr 22, ~1 million variants and genotypes across 2500 samples.
- See VCF format

Reading large genomics data

data.table package and fread

fread function

- + Very fast.
- + Usually no need for additional parameters.
- - Has its specific format (*data.table*)...
- + ... which can be converted into *data.frame* .
- + Very fast.

Example

```
library(data.table)
myDT = fread("myFile.tsv")
myDF = as.data.frame(myDT)

myDT = fread("gunzip -c myFile.tsv.gz")
```

Exercise

1. Read Gencode file (`gencode.gtf.gz`) with `read.table`.
2. Same with `fread`.
3. Have a look at the data.
4. List the different chromosomes.
5. How many different gene types ?
6. How many different genes ?

※ How many different genes per gene type ?

Chunk-by-chunk approach

When you can analyze the data in slices.

+ Only a slice of the file in memory.
- A bit painful/ugly.

```
con = file(file.name)
while(length((chunk.df = read.table(con,nrows=1000)))>0){
... Instructions
}
close(con)
```

Bioconductor packages

- GFF format with `import` (*rtracklayer* package).
- VCF format with `readVcf` (*VariantAnnotation* package).

+ Parse the format.
- Sometimes parse too much $\rightarrow$ complicated object.
+ Read indexed files.

Exercise

1. Read `gencode.gtf.gz` in another object using `import`.
2. Have a look at the data.

Using file indexing

Why indexing ?

To **quickly import a slice** of a file. In genomics, one region.

Indexing workflow

1. Order file by position (chr + start).
2. Compress with bgzip.
3. Index.

How ?

With command lines or with R functions.

Indexing files with R

data.table package is faster to order large files than conventional R.

```
library(data.table)
dt = fread("file.bed")
setkey(dt, chr, start)
write.table(dt, file="file-ordered.bed")

library(Rsamtools)
bgzip("file-ordered.bed")
indexTabix("file-ordered.bed.bgz")
```

Using indexed files

```
reg = GRanges(...)

library(VariantAnnotation)
vcf <- readVcf("variants.vcf.gz", "hg19", reg)

library(rtracklayer)
gtf = import(TabixFile("annotation.gtf.bgz"), which=reg)
```

Exercise

1. Read variants in VCF between coordinates 30 Mb and 31 Mb.
2. Order, write, compress and index the gencode file.

⋇ Tile the Mb in 10 bins. In each bin count the number of variants in the VCF.

Analyzing large genomics data

Avoid loops, use sapply/lapply instead

+ Avoid manual init/update of objects.
+ No temporary object polluting the environment.
+ More optimized.
+ Easy to parallelize.
- More painful to debug.
- An error and everything must be computed again.

```
perm.l = lapply(1:1000, function(ii){
  data.frame(perm=ii, est=..SOMETHING..)
})

perm.df = do.call(rbind, perm.l)
## or
perm.df = as.data.frame(rbindlist(perm.l))
```

Stupid exercise

Sample 10 elements of the gencode annotation and count the number of exons, 100 times.

Parallel processing

Easiest solution with *parallel* package

- Using `mclapply` instead of `lapply`.
- `mc.cores=` the number of processors to use.

Example

```
perm.l = mclapply(1:1000, function(ii){
  data.frame(perm=ii, est=..SOMETHING..)
}, mc.cores=4)
```

Exercise

Parallelize the previous exercise.

Using computing clusters directly with *BatchJobs*

What you need

- A function that can independently the code
  - Load packages.
  - Load necessary data.
  - Run the code.
- A parameter list (used to define the jobs).
- Some global objects (same for all jobs). Optional.
- Your favorite cluster configured (or your computer).

More info

Checkout PopSV documentation on BatchJobs. To use Guillimin, Abacus, Briaree or Mammouth ask me for the configuration files.

BatchJobs example

```
library(BatchJobs)

reg = makeRegistry("perm")

jobFun <- function(ii, necessaryData){
    library(...)
    ... Instructions using 'ii' and 'necessaryData'
    data.frame(perm=ii, ...)
}

batchMap(reg, jobFun, 1:10, more.args=list(necessaryData=myData))

submitJobs(reg))

showStatus(reg)

perm.l = reduceResultsList(reg)
```

data.frames

- Mix between *matrix* and *list*
- Array form.
- Columns can have different data types.

matrix

```
      samp1 samp2 samp3
gene1  -1.3  -1.8  -4.1
gene2  -1.5  -1.2   4.9
```

data.frame

```
 gene sample expression
gene1  samp1       -1.3
gene2  samp1       -1.5
gene1  samp2       -1.8
gene2  samp2       -1.2
gene1  samp3       -4.1
gene2  samp3        4.9
```

Pros/Cons

+ Dense representation of large data.
- Accepts only one data type.
- manual combination with other information often required.

+ Flexible.
+ Accepts several data types.
+ Can represent all the data needed for an analysis.
- Takes more space/memory due to repetitions.

dplyr package

"A Grammar of Data Manipulation"

dplyr provides functions which can be combined for data manipulation.

mutate	add a new column using others.
filter	filter rows (similar as `subset` function).
select	select specific columns only.
arrange	order rows using specific columns.
group_by	groups rows according to specific columns.
summarize	summarizes each group of rows.
do	applies a function to a group of rows.

+ Works with pipes.
+ Fast.
- Has its own format *tbl_df*...
+ ... which is almost the same as *data.frame* .

Pipes are cool !

- Pipe functions instead of embedding them.
- More readable.
- Easier to combine several functions.
- Avoid temporary objects.
- Pipe argument %>%.

Example

```
head(sort(round(sqrt(myVec))))
myVec %>% sqrt %>% round %>% sort %>% head

head(sort(round(sqrt(myVec),digits=3),decreasing=TRUE),10)
myVec %>% sqrt %>% round(digits=3) %>%
                         sort(decreasing=TRUE) %>% head(10)
```

Grouping rows

Operation by block

- Using `group_by()` function.
- Further operations are applied separately per group of rows.

Example

```
myDF %>% group_by(colA) %>% summarize(colB.mean=mean(colB))

myDF %>% group_by(colA, colB) %>% summarize(nbAB=n())

myDF %>% group_by(colAB) %>% summarize(nbAB=n()) %>%
                 ungroup %>% arrange(desc(nbAB)) %>% head
```

Tips

- `n()` gives the number of rows in the group.
- `ungroup` removes groups.
- `desc()` means descending order (in `arrange()`).

Exercise

1. For each gene, compute the number of exon.
2. Print the top 10 genes with the most exons.
3. For each gene type, compute the average number of exon per gene.
4. For each gene type, compute the average gene size.
5. For each chromosome, compute the number of genes for each gene type.
6. In protein-coding genes, compute the median size of the first exon, second exon, etc (i.e. per `exon_number`).

GenomicRanges

Introduction

Represents genomic intervals. All annotation can be represented through *GenomicRanges* objects.

Which function fit your exact need ?

overlapsAny	Test overlaps of one *GRanges* into second *GRanges* .
countOverlaps	For each region in one *GRanges* , count how many overlaps from another.
findOverlaps	Finds overlaps between two *GRanges* objects.
distanceToNearest	Computes the distance from each regions in a *GRanges* object to the nearest in another *GRanges* object.
subsetByOverlaps	Keep the regions from one *GRanges* that overlaps another.

Overlaps between two *GRanges* sets

findOverlaps function

- Two *GRanges* objects as input.
- Extra parameters available for specific overlaps.
- Returns the index of regions in object 1 and 2 that overlap.
- `queryHits` and `subjectHits` functions to retrieves those index.

Example

```
> ol12 = findOverlaps(gr1, gr2)
> ol12
Hits of length 3
queryLength: 6
subjectLength: 4
  queryHits subjectHits
   <integer>   <integer>
 1         1           1
 2         2           2
 3         5           3
> queryHits(ol12)
[1] 1 2 5
```

Better one big overlap than many small ones

Exercise

1. For each gene type, how many genes have at least one variant in their body ?

※ For each gene type, what is the average number of variant per gene ?

※ For each gene type, what is the average allele frequency of the variant ?

Hints

`lapply`, `tapply`, *dplyr*.

Other tricks

- Create a *GRanges* from a *data.frame* .
- Change chromosome names ('chr' or no 'chr', that is the question).

```r
library(GenomicRanges)

gr = makeGRangesFromDataFrame(df, keep.extra.columns = TRUE)

seqlevels(genc) = gsub("chr","",seqlevels(genc))
## or
seqlevels(genc) = paste0("chr",seqlevels(genc))
```

GenomicRanges + dplyr

tapply

1. The values to use (vector).
2. How to group these values (vector).
3. The function to run (input is one vector).

```
tapply(...subjectHits(ol)..., ...queryHits(ol)..., function(x)...)

ol %>% as.data.frame %>% mutate(...queryHits..., ...subjectHits...)
   %>% group_by(...) %>% summarize(...)
```

`tapply` works but *dplyr* is faster and more flexible.

Visualizing large genomics data

Gviz for multi-track graphs

```
library(Gviz)

gatrack = GenomeAxisTrack()
snp.t = DataTrack(snp.gr, data="AF", type='h', name="SNP freq")
plotTracks(list(gatrack, snp.t))
```

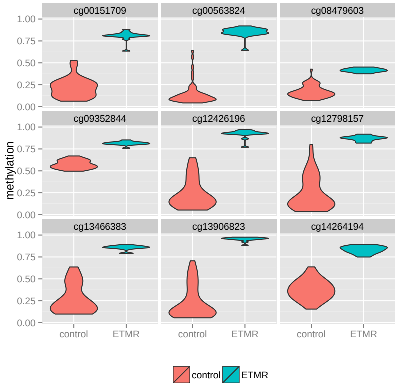
More info

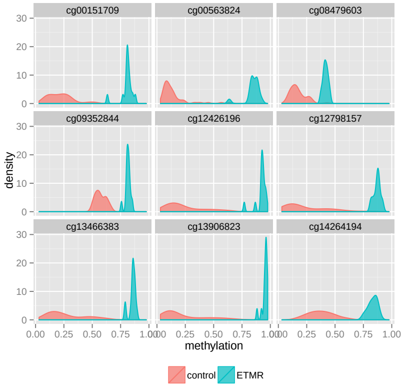
See slides/code/links from MonBUG Gviz demo.

ggplot2 package

Introduction

A package to construct pretty and/or complex graphs. Many aspects of the graph are arranged automatically but everything can be specified. Easy layers addition.





ggplot2

Input : *data.frame*

- Each row represents one *"observation"*.
- Columns represent the different information about the *"observations"*.

Concept

- Start with a `ggplot(...)` part and the input *data.frame* .
- `aes(...)` defines how to use the input *data.frame* columns.
- Add layers : `geom_*(...)`, `scale_*(...)`, ...

Example

```
library(ggplot2)
ggplot(myDf, aes(x=colA, y=colB, colour=colC, linetype=colD))
   + geom_point() + geom_line() + scale_y_log10()
```

Useful online resources

- http://docs.ggplot2.org/current/
- http://www.cookbook-r.com/Graphs/

Exercise

1. Show the distribution of the gene size (histogram), colored by gene type.
2. Show for each chromosome the number of genes, colored by gene type.

※ Show the median size of the first exon, second exon, etc, of protein coding genes.

※ For each gene type, what is the average allele frequency of the variant ? Maybe a boxplot ?

Final recommendations

- Use **names** that makes sense (to you and future you).
- **Nothing in the console**, everything in an organized script.
- The script should be **sequential and commented** when complex.
- Save the graphs **in the code**, not manually through RStudio.
- **Split** long scripts and **save** temporary files.
- **Overwriting** is fine if in the same paragraph.
- Use **functions/pipes** to avoid environment/code pollution.
- Use **R Markdown** to produce a readable report while keeping the code.