Genomalicious tutorial 1: Basic ingredients

J.A. Thia

29 July 2019

Preamble

Every good recipe is built around simple ingredients. Likewise, bioinformatic pipelines, while complex, require some fundamental methods and data types that are the bare neccessities to any study implementing population genomic approaches.

In this tutorial, you will:

1. Become familiar with some basic features of *genomalicious*.
2. Develop familiarity with SNP data structures.
3. Learn to use *genomalicious* functions to import and manipulate SNP data structures.

Getting to know *genomalicious*

Let's start by loading the *genomalicious* library into your R session.

```
library(genomalicious)
```

Genomalicious contains a number of demonstrative toy datasets that you can experiment on. These need to be loaded in with the `data()` function and follow the naming convention, `genomalicious_[data name]`, where `[data name]` is a unique identifier.

For example, let's take a look at a toy dataset of allele frequencies:

```
data("genomalicious_Freqs")

# Take a look at the structure of this data, the dim() function reports the dimensons.
dim(genomalicious_Freqs)
```

```
## [1] 4 8
```

```
# Print the data to screen.
genomalicious_Freqs
```

```
##      Chrom_1_8 Chrom_2_16 Chrom_3_24 Chrom_4_32 Chrom_5_40 Chrom_6_48
## Pop1 0.5882639  0.6985712  0.9281353  0.7154336  0.2783644  0.5004546
## Pop2 0.5143548  0.6962864  0.9250919  0.3973361  0.3866906  0.4006825
## Pop3 0.7224595  0.6567121  0.8721565  0.2434911  0.6373447  0.9227758
## Pop4 0.6771910  0.5523064  0.9278697  0.4081590  0.6456204  0.8862767
##      Chrom_7_56 Chrom_8_64
## Pop1  0.4777536 0.06605165
## Pop2  0.3632988 0.06384908
## Pop3  0.2049917 0.19425487
## Pop4  0.2442950 0.15228191
```

You will notice that `genomalicious_Freqs` is a matrix with 4 rows and 8 columns. The rows contain the populations, whereas the columns contain the loci, and the cells contain allele frequencies. This data is in **wide format**, but we will dig into what this means a little later.

You can learn more about a dataset by using `?` and the data object name, e.g. `?genomalicious_Freqs`. We will look at other datasets in later tutorials, but you can peruse the various datasets by typing `genomalicious_` and hitting the TAB key to get a list of options.

Loading VCFs into R

Variant call files (VCFs) are one of the typical end products from read assembly/variant calling pipelines. These files contain information about genotypes attributed to each sample and associated mapping and genotype calling statistics.

Genomalicious offers a very simple way to import VCFs into R as **long format** data tables, whereby loci and and populations are both in rows. C.f. with the wide format data described above.

We will now import a demo VCF into R using the `vcf2DT()` function. First, we need to find where *genomalicious* is installed and make a path to the demo file.

```
# Create a link to raw external datasets in genomalicious
genomaliciousExtData <- paste0(find.package('genomalicious'), '/extdata')

# This command here shows you the VCF file that comes with genomalicious
list.files(genomaliciousExtData, pattern='_poolseq.vcf')
```

```
## [1] "genomalicious_poolseq.vcf"
```

```
# Use this to create a path to that file
vcfPath <- paste0(genomaliciousExtData, '/genomalicious_poolseq.vcf')

# The value of vcfPath with depend on your system
vcfPath
```

```
## [1] "C:/Users/Josh/Documents/R/win-library/3.4/genomalicious/extdata/genomalicious_poolseq.vcf"
```

You can naviagte yourself to the path stored in `vcfPath` and open the VCF using a text editor. However, we can simply read the lines of the file and print them to screen:

```
# Read in and print the first 10 lines of the demo VCF
head(readLines(vcfPath), 10)
```

```
##  [1] "##This is a toy dataset for the R package genomalicious - it emulates a VCF file"
##  [2] "##INFO=<ID=DP,Number=1,Type=Integer,Description='The total depth across samples'>"
##  [3] "##FORMAT=<ID=DP,Number=1,Type=Integer,Description='The total depth in a sample'>"
##  [4] "##FORMAT=<ID=RO,Number=1,Type=Integer,Description='The reference allele counts in a sample'>"
##  [5] "##FORMAT=<ID=AO,Number=1,Type=Integer,Description='The alternate allele counts in a sample'>"
##  [6] "#CHROM\tPOS\tID\tREF\tALT\tQUAL\tFILTER\tINFO\tFORMAT\tPop1_Rep1\tPop1_Rep2\tPop1_Rep3\tPop2_R
##  [7] "Chrom_1\t8\t.\tA\tT\t50\tPASS\tDP=854\tDP:RO:AO\t113:63:50\t48:32:16\t46:26:20\t119:63:56\t55:1
##  [8] "Chrom_2\t16\t.\tC\tG\t40\tPASS\tDP=785\tDP:RO:AO\t82:60:22\t47:38:09\t48:26:22\t85:61:24\t58:4
##  [9] "Chrom_3\t24\t.\tG\tA\t45\tPASS\tDP=864\tDP:RO:AO\t101:90:11\t52:50:02\t48:47:01\t85:75:10\t52:5
## [10] "Chrom_4\t32\t.\tT\tC\t43\tPASS\tDP=792\tDP:RO:AO\t101:73:28\t55:41:14\t55:37:18\t98:41:57\t39:
```

This demo VCF was constructed more simply for the purpose of this tutorial, but follows the same basic structure of that produced from variant calling software. You will see that the text is interspersed with `\t`: these indicate tabs that separate the various columns in the file.

A VCF has the following components:

1. A **comments section** marked with double hashtags, `##`. These typically contain details about the contents of the VCF or how the reads were called.
2. A **heading section** marked with a single hashtag, `#`. This is effectively the column names for the wide format SNP data.
3. All lines proceeding contain SNP data.

VCFs can be a bit tricky to interpret when you first see them. All columns **before** `FORMAT` contain information about the SNP (i.e. the chromosome, its position, the alleles). `FORMAT` **itself** contains a text string detailing the contents of the sample columns. Note, the contents of `FORMAT` are typically described in the comments section. We therefore know that `DP:RO:AO` represent the total read depth (`DP`), and the counts of the referece (`RO`) and alternate (`AO`) allele. Also note the difference in `DP` stored in the `INFO` column (read depth across all samples) and that in `FORMAT` (read depth within each sample).

All sample columns occur **after** `FORMAT` (e.g. `Pop1_Rep1`, `Pop1_Rep2`) and continue to the end of the line. Within each sample column, the values described in `FORMAT` are stored, with each value separated by a `:`. This demo VCF contains read data from a replicated pool-seq experiment. Individuals from four populations were pooled into a single library, and three replicate libraries were sequenced. The reads associated with each population replicate, for each SNP, are detailed in the i-by-j row-column coordinates.

For more in depth descriptions of VCFs, you should check out the resources provided by the developers of *samtools*, here.

Now that we understand the data we are working with, let's import it into R.

```
# Import VCF into R using the path name
poolSnps <- vcf2DT(vcfPath)

# First 8 rows of the imported SNP data
head(poolSnps, 8)
```

```
##      CHROM POS ID REF ALT QUAL FILTER   INFO    SAMPLE      LOCUS  DP RO
## 1: Chrom_1   8  .   A   T   50   PASS DP=854 Pop1_Rep1  Chrom_1_8 113 63
## 2: Chrom_2  16  .   C   G   40   PASS DP=785 Pop1_Rep1 Chrom_2_16  82 60
## 3: Chrom_3  24  .   G   A   45   PASS DP=864 Pop1_Rep1 Chrom_3_24 101 90
## 4: Chrom_4  32  .   T   C   43   PASS DP=792 Pop1_Rep1 Chrom_4_32 101 73
## 5: Chrom_5  40  .   A   C   30   PASS DP=825 Pop1_Rep1 Chrom_5_40 100 22
## 6: Chrom_6  48  .   T   G   37   PASS DP=854 Pop1_Rep1 Chrom_6_48 101 51
## 7: Chrom_7  56  .   C   G   38   PASS DP=820 Pop1_Rep1 Chrom_7_56 109 50
## 8: Chrom_8  64  .   T   A   52   PASS DP=838 Pop1_Rep1 Chrom_8_64 101  3
##    AO
## 1: 50
## 2: 22
## 3: 11
## 4: 28
## 5: 78
## 6: 50
## 7: 59
## 8: 98
```

```
# The imported data is stored as a data.table object
class(poolSnps)
```

```
## [1] "data.table" "data.frame"
```

As you can see, the function `vcf2DT()` converts the wide format VCF data into a long format data table. Instead of a single column for each sample (and loci in rows), all possible sample-by-locus combinations are stored in the rows of `poolSnps`, with a single column each for samples and loci. Using the `$` notation, you can access these columns as vecotrs.

```
# Column vector for sample and loci
head(poolSnps$SAMPLE)
```

```
## [1] "Pop1_Rep1" "Pop1_Rep1" "Pop1_Rep1" "Pop1_Rep1" "Pop1_Rep1" "Pop1_Rep1"
```

```
head(poolSnps$LOCUS)
```

```
## [1] "Chrom_1_8"  "Chrom_2_16" "Chrom_3_24" "Chrom_4_32" "Chrom_5_40"
## [6] "Chrom_6_48"
```

Typically in these tutorials, and throughout the documentation for *genomalicious*, I will use `$` to indicate nested vectors of R objects (e.g. columns, list items).

Data tables for storing SNP data

Genomalicious is largely built around `data.table` classed objects. Data tables feel and more-or-less function as per standard R `data.frame` objects, but they differ in two major ways. Firstly, they are much more efficient at storing large volumes of data. Secondly, they have some nifty features that facilitate easy data manipualtions. Though the purpose of this tutorial is not to provide a detailed demonstration of `data.table` object features (you can find that, here), let's just take a quick look at how you can harness the power of data tables:

```
# Subset rows by depth.
poolSnps[DP > 110,]
```

```
##      CHROM POS ID REF ALT QUAL FILTER   INFO    SAMPLE      LOCUS  DP RO
## 1: Chrom_1   8  .   A   T   50   PASS DP=854 Pop1_Rep1  Chrom_1_8 113 63
## 2: Chrom_1   8  .   A   T   50   PASS DP=854 Pop2_Rep1  Chrom_1_8 119 63
## 3: Chrom_6  48  .   T   G   37   PASS DP=854 Pop2_Rep1 Chrom_6_48 116 43
## 4: Chrom_1   8  .   A   T   50   PASS DP=854 Pop4_Rep1  Chrom_1_8 120 82
## 5: Chrom_5  40  .   A   C   30   PASS DP=825 Pop4_Rep1 Chrom_5_40 113 71
##    AO
## 1: 50
## 2: 56
## 3: 73
## 4: 38
## 5: 42
```

```
# You can apply functions to columns, for example,
# take the mean depth.
poolSnps[, mean(DP)]
```

```
## [1] 69.08333
```

```
# You can even specify subsets of the data that you want to
# apply the function to, for example, take the mean depth
# for each locus.
poolSnps[, mean(DP), by=LOCUS]
```

```
##         LOCUS       V1
## 1:  Chrom_1_8 71.16667
## 2: Chrom_2_16 65.41667
## 3: Chrom_3_24 72.00000
## 4: Chrom_4_32 66.00000
## 5: Chrom_5_40 68.75000
## 6: Chrom_6_48 71.16667
## 7: Chrom_7_56 68.33333
## 8: Chrom_8_64 69.83333
```

```
# You can also use this feature of column manipulation to
# apply a function to the data and create a new column using
# the ':=' notation. For example, add a new column
# containting the frequency of reads that contain the
# reference allele.
poolSnps[, R.FREQ:=RO/DP]
head(poolSnps, 4)
```

```
##      CHROM POS ID REF ALT QUAL FILTER   INFO    SAMPLE      LOCUS  DP RO
## 1: Chrom_1   8  .   A   T   50   PASS DP=854 Pop1_Rep1  Chrom_1_8 113 63
## 2: Chrom_2  16  .   C   G   40   PASS DP=785 Pop1_Rep1 Chrom_2_16  82 60
## 3: Chrom_3  24  .   G   A   45   PASS DP=864 Pop1_Rep1 Chrom_3_24 101 90
## 4: Chrom_4  32  .   T   C   43   PASS DP=792 Pop1_Rep1 Chrom_4_32 101 73
##    AO    R.FREQ
## 1: 50 0.5575221
## 2: 22 0.7317073
## 3: 11 0.8910891
## 4: 28 0.7227723
```

Many functions in *genomalicious* take long format `data.table` objects as their direct input. From personal experience, I find data tables from 100s of individuals, and 1,000s to 10,000s of SNP loci are very manageable, though they may take a moment to load into R (especially when importing from a VCF). However, I believe the ease, simplicity, and versatility of working with data tables makes them an excellent object class for reduced representation SNPs.

Data tables are also easily transformed into matrices and pure data frames:

```
# Converting to a matrix
matDat <- as.matrix(poolSnps)
class(matDat)
```

```
## [1] "matrix"
```

```
head(matDat, 4)
```

```
##      CHROM     POS  ID  REF ALT QUAL FILTER INFO     SAMPLE
## [1,] "Chrom_1" " 8" "." "A" "T" "50" "PASS" "DP=854" "Pop1_Rep1"
## [2,] "Chrom_2" "16" "." "C" "G" "40" "PASS" "DP=785" "Pop1_Rep1"
## [3,] "Chrom_3" "24" "." "G" "A" "45" "PASS" "DP=864" "Pop1_Rep1"
## [4,] "Chrom_4" "32" "." "T" "C" "43" "PASS" "DP=792" "Pop1_Rep1"
##      LOCUS        DP    RO    AO   R.FREQ
## [1,] "Chrom_1_8"  "113" " 63" "50" "0.55752212"
## [2,] "Chrom_2_16" " 82" " 60" "22" "0.73170732"
## [3,] "Chrom_3_24" "101" " 90" "11" "0.89108911"
## [4,] "Chrom_4_32" "101" " 73" "28" "0.72277228"
```

```
# Converting to a data frame
dfDat <- as.data.frame(poolSnps)
class(dfDat)
```

```
## [1] "data.frame"
```

```
head(dfDat, 4)
```

```
##     CHROM POS ID REF ALT QUAL FILTER   INFO    SAMPLE      LOCUS  DP RO AO
## 1 Chrom_1   8  .   A   T   50   PASS DP=854 Pop1_Rep1  Chrom_1_8 113 63 50
## 2 Chrom_2  16  .   C   G   40   PASS DP=785 Pop1_Rep1 Chrom_2_16  82 60 22
## 3 Chrom_3  24  .   G   A   45   PASS DP=864 Pop1_Rep1 Chrom_3_24 101 90 11
## 4 Chrom_4  32  .   T   C   43   PASS DP=792 Pop1_Rep1 Chrom_4_32 101 73 28
##      R.FREQ
## 1 0.5575221
## 2 0.7317073
## 3 0.8910891
## 4 0.7227723
```

Data structures: Long-to-wide format and genotype values

Though many functions in R expect data structured in long format, there are lots of others that require data to be wide format, especially many population genetics/genomics packages. Remember, in long form, loci-by-sample combinations are all in rows, whereas in wide format, samples are in rows and loci are in columns.

There are two functions in *genomalicious* for long-to-wide conversions: `DT2Mat_freqs()` for **allele frequencies**, and `DT2Mat_genos()` for **individual genotypes**. Both return an R `matrix` object.

Let's first try with the pooled SNP data we imported at the start of this lesson. Earlier, you made a column in `poolSnps`, `$R.FREQ`, that contained the frequnecy of reads per sample of the reference allele. We will treat column `$R.FREQ` as an estimate of the sample allele frequency.

`DT2Mat_freqs()` requires specification of the sample, locus, and allele frequency columns in the long format data table as the arguments, `popCol`, `locusCol`, and `freqCol`, respectively:

```r
# Convert long format data table of allele frequencies to a matrix
freqMat <- DT2Mat_freqs(poolSnps, popCol='SAMPLE', locusCol='LOCUS', freqCol='R.FREQ')
freqMat
```

```
##           Chrom_1_8 Chrom_2_16 Chrom_3_24 Chrom_4_32 Chrom_5_40 Chrom_6_48
## Pop1_Rep1 0.5575221  0.7317073  0.8910891  0.7227723  0.2200000  0.5049505
## Pop1_Rep2 0.6666667  0.8085106  0.9615385  0.7454545  0.3703704  0.5098039
## Pop1_Rep3 0.5652174  0.5416667  0.9791667  0.6727273  0.2558140  0.4800000
## Pop2_Rep1 0.5294118  0.7176471  0.8823529  0.4183673  0.4204545  0.3706897
## Pop2_Rep2 0.5090909  0.7068966  0.9615385  0.3846154  0.4375000  0.3650794
## Pop2_Rep3 0.5000000  0.6530612  0.9558824  0.3658537  0.2641509  0.5128205
## Pop3_Rep1 0.7674419  0.6769231  0.9313725  0.1666667  0.6739130  0.9176471
## Pop3_Rep2 0.7187500  0.7228916  0.8250000  0.3544304  0.6627907  0.9310345
## Pop3_Rep3 0.6739130  0.5517241  0.8611111  0.1875000  0.5588235  0.9393939
## Pop4_Rep1 0.6833333  0.5757576  0.9259259  0.4019608  0.6283186  0.9090909
## Pop4_Rep2 0.6666667  0.5142857  0.9583333  0.5000000  0.6290323  0.8620690
## Pop4_Rep3 0.6774194  0.5428571  0.9166667  0.3333333  0.7115385  0.8888889
##           Chrom_7_56 Chrom_8_64
## Pop1_Rep1  0.4587156 0.02970297
## Pop1_Rep2  0.4375000 0.08196721
## Pop1_Rep3  0.5476190 0.09090909
## Pop2_Rep1  0.2755102 0.06315789
## Pop2_Rep2  0.3770492 0.04838710
## Pop2_Rep3  0.4883721 0.08695652
## Pop3_Rep1  0.2048193 0.17500000
## Pop3_Rep2  0.2000000 0.21794872
## Pop3_Rep3  0.2142857 0.18918919
## Pop4_Rep1  0.2688172 0.20000000
## Pop4_Rep2  0.2166667 0.12698413
## Pop4_Rep3  0.2424242 0.09859155
```

```r
# Check the class
class(freqMat)
```

```
## [1] "matrix"
```

```r
# Sample names are stored in the rows of the matrix
rownames(freqMat)
```

```
##  [1] "Pop1_Rep1" "Pop1_Rep2" "Pop1_Rep3" "Pop2_Rep1" "Pop2_Rep2"
##  [6] "Pop2_Rep3" "Pop3_Rep1" "Pop3_Rep2" "Pop3_Rep3" "Pop4_Rep1"
## [11] "Pop4_Rep2" "Pop4_Rep3"
```

```r
# Loci names are stored in the columns of the matrix
colnames(freqMat)
```

```
## [1] "Chrom_1_8"  "Chrom_2_16" "Chrom_3_24" "Chrom_4_32" "Chrom_5_40"
## [6] "Chrom_6_48" "Chrom_7_56" "Chrom_8_64"
```

There is also a way to go in reverse — that is, convert the wide format matrix back into a long format data frame. This is done by specifying the argument `flip=TRUE` (default vlaue is set to `FALSE`). Doing so

requires the input to be a wide format matrix, and specifying `popCol`, `locusCol`, and `freqCol` is required to determine the names of these columns in the new output long format data table.

```
freqDT <- DT2Mat_freqs(freqMat, popCol='SAMPLE', locusCol='LOCUS', freqCol='R.FREQ', flip=TRUE)
head(freqDT, 8)
```

```
##       SAMPLE     LOCUS    R.FREQ
## 1: Pop1_Rep1 Chrom_1_8 0.5575221
## 2: Pop1_Rep2 Chrom_1_8 0.6666667
## 3: Pop1_Rep3 Chrom_1_8 0.5652174
## 4: Pop2_Rep1 Chrom_1_8 0.5294118
## 5: Pop2_Rep2 Chrom_1_8 0.5090909
## 6: Pop2_Rep3 Chrom_1_8 0.5000000
## 7: Pop3_Rep1 Chrom_1_8 0.7674419
## 8: Pop3_Rep2 Chrom_1_8 0.7187500
```

Up to this point, we have worked with pooled sequence data and allele frequencies. Let's now take a look at individually genotyped data. We will import a *genomalicious* dataset of four populations:

```
# Import the dataset
data(genomalicious_4pops)
indSnps <- genomalicious_4pops

# Have a look at its structure
head(indSnps)
```

```
##     POP   SAMPLE    CHROM      LOCUS  GT
## 1: Pop1 Ind1.115  Chrom_3  Chrom_3_4 0/0
## 2: Pop1 Ind1.115  Chrom_5  Chrom_5_1 0/0
## 3: Pop1 Ind1.115  Chrom_7  Chrom_7_5 0/1
## 4: Pop1 Ind1.115  Chrom_8  Chrom_8_4 0/0
## 5: Pop1 Ind1.115 Chrom_12 Chrom_12_5 0/1
## 6: Pop1 Ind1.115 Chrom_15 Chrom_15_3 0/1
```

```
# The number of unique samples, per population
indSnps[, length(unique(SAMPLE)), by=POP]
```

```
##     POP V1
## 1: Pop1 30
## 2: Pop2 30
## 3: Pop3 30
## 4: Pop4 30
```

```
# The number of unique loci
length(unique(indSnps$LOCUS))
```

```
## [1] 1205
```

You would have noticed a column in `indSnps` called `$GT`: this column contains information on each sample's genotype at a particular locus. '0' is the reference allele and '1' is the alternate allele; each allele is separated

by a '/'. This is a **separated** format of genotype values. However, another way of representing genotypes is via **counts** of one of the alleles, e.g. '0', '1', or '2'.

Typically when genotypes are represented as allele counts, we assume biallelic data. If a diploid individual possess two alleles per locus, then it is only possible to keep track of a maximum of two unique alleles. In reality, there are other ways to work around this, but many population genomic analyses are constrained to be on biallelic SNPs.

The function `genoscore_converter()` can be used to convert between the different ways of scoring genotype values. The input into this function is simply a vector of genotypes. If inputting the separated format, the vector must be a `character` class object. If inputting allele count format, the vector is ideally `integer` class object. The function bases its counts of the Alternate allele, hence a genotype of '0/0', '0/1', and '1/1', are equivalent to 0, 1, and 2 (respectively).

```
# Practise run with simple vectors
genoscore_converter(c('0/0', '0/1', '1/1'))
```

```
## [1] 0 1 2
```

```
genoscore_converter(c(0L, 1, 2))
```

```
## [1] "0/0" "0/1" "1/1"
```

```
# Now manipulate the data table of genotypes
indSnps[, GT:=genoscore_converter(GT)]

head(indSnps, 8)
```

```
##     POP   SAMPLE    CHROM      LOCUS GT
## 1: Pop1 Ind1.115  Chrom_3  Chrom_3_4  0
## 2: Pop1 Ind1.115  Chrom_5  Chrom_5_1  0
## 3: Pop1 Ind1.115  Chrom_7  Chrom_7_5  1
## 4: Pop1 Ind1.115  Chrom_8  Chrom_8_4  0
## 5: Pop1 Ind1.115 Chrom_12 Chrom_12_5  1
## 6: Pop1 Ind1.115 Chrom_15 Chrom_15_3  1
## 7: Pop1 Ind1.115 Chrom_16 Chrom_16_2  0
## 8: Pop1 Ind1.115 Chrom_17 Chrom_17_3  2
```

Finally, let's practise converting a long data table of genotypes into a genotype matrix, using `DT2Mat_genos()`. The arguments are the same as `DT2Mat_freqs()`.

```
genoMat.sep <- DT2Mat_genos(indSnps, sampCol='SAMPLE', locusCol='LOCUS', genoCol='GT')

genoMat.sep[1:8, 1:4]
```

```
##           Chrom_1001_4 Chrom_1003_2 Chrom_1004_1 Chrom_1006_3
## Ind1.1040            2            0            0            2
## Ind1.1116            2            2            0            2
## Ind1.115             2            2            0            2
## Ind1.1153            2            1            0            2
## Ind1.1161            2            0            0            2
## Ind1.1184            2            1            0            1
## Ind1.1240            2            0            0            2
## Ind1.1315            0            2            0            2
```

Postamble

This concludes the 'Basic Ingredients' tutorial. You should now be comfortable with the basic functionality of *genomalicious* for importing SNP data into R. You have also familiarised yourself with SNP data structures and learnt how *genomalicious* can be used to do some basic manipulations that are common in population genetic/genomic analyses.