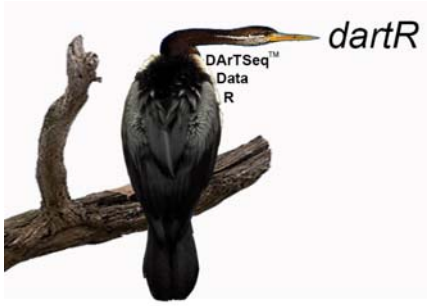
UNIVERSITY OF CANBERRA

SNP Analysis using dartR



For the Developer

Copies of these workshop notes are available from:

The Institute for Applied Ecology
University of Canberra ACT 2601
Australia

Email: georges@aerg.canberra.edu.au

Contents

Sample Code Proforma

```
#' <Short informative title>
#'
#' <Longer descriptive title>
#'
#' <Details>
#'
#' <Reference to literature>
#'
#' <parameters using @param, default in []>
#' @param x -- a genlight object containing the SNP genotypes
                                        [Required]
#' @param verbose -- verbosity: 0, silent or fatal errors; 1, begin
      and end; 2, progress log ; 3, progress and results summary;
      5, full report [default 2 or as specified using
      gl.set.verbosity]

#' @return <Description of returned object>
#' @export
#' @author <List authors names> (Post to
      \url{https://groups.google.com/d/forum/dartr})
#'
#' @import <Package>
#' @importFrom <Package function>

#' @examples
#' <Example of code that runs quickly using the testsets>

gl.<Function Name> <- function(x, verbose=NULL) {

# TRAP COMMAND, SET VERSION

  funname <- match.call()[[1]]
  build <- NULL # We will add this in later

# SET VERBOSITY

  if (is.null(verbose)){
    if(!is.null(x@other$verbose)){
      verbose <- x@other$verbose
    } else {
      verbose <- 2
    }
  }

  if (verbose < 0 | verbose > 5){
    cat(paste("  Warning: Parameter 'verbose' must be an integer
        between 0 [silent] and 5 [full report], set to 2\n"))
    verbose <- 2
  }

# FLAG SCRIPT START

  if (verbose >= 1){
    if(verbose==5){
      cat("Starting",funname,"[ Build =",build,"]\n")
    } else {
      cat("Starting",funname,"\n")
    }
  }

# STANDARD ERROR CHECKING

  if(class(x)!="genlight") {
    stop("Fatal Error: genlight object required!")
  }
```

```
  if (all(x@ploidy == 1)){
    stop("  Processing  Presence/Absence (SilicoDArT) data,
        heterozygosity can only be calculated for SNP data\n")
    data.type <- "SilicoDArT"
  } else if (all(x@ploidy == 2)){
    if (verbose >= 2){cat("  Processing a SNP dataset\n")}
    data.type <- "SNP"
  } else {
    stop("Fatal Error: Ploidy must be universally 1 (fragment P/A
        data) or 2 (SNP data)")
  }

# SCRIPT SPECIFIC ERROR CHECKING

  if (<Condition>) {
    cat("<Fatal Error and Stop; or Warning and rectify>\n")
  }

# DO THE JOB FOR POPULATIONS

  <Code to do the job, produce the object for output>

# ADD TO HISTORY
  nh <- length(x@other$history)
  x@other$history[[nh + 1]] <- match.call()

# FLAG SCRIPT END

  if (verbose > 0) {
    cat("Completed:",funname,"\n")
  }

  return(<Output Object>)
}
```

Programming Style

R is a programming language that allows a fair bit of flexibility, and so different programmers develop different approaches to coding. Two programmers writing code to address the same problem can generate quite different code, the code of one sometimes quite difficult to decipher by the other.

Notwithstanding the object orientation of R, loose constraint on the style of programming can be the enemy of collaborative effort. So the first principle in preparing code is:

Write your code for the future you

Indeed, write your code for the future programmer who might have to pick up from you.

In essence, this means writing for readability not efficiency, except where efficiency really matters. Embedded object chains might be clever compact coding, but at the expense of comprehensibility.

Comment, comment, comment. A person picking up your code, with programming experience, but perhaps little experience with R, should be able to comprehend your code. Takes time, but well worth the effort in the longer run.

Indent your code to clearly identify conditional blocks and repetition blocks. For example

```
if (all(x@ploidy==1)){
   cat("  Processing Presence/Absence (SilicoDArT) data. \n")
} else if (all(x@ploidy==2)){
   cat("  Processing a SNP dataset\n")
} else {
   stop("Fatal Error: Ploidy must be universally 1 or 2!")
}
```

Try to avoid R functions that act in specific ways depending on context, behaviour that requires a detailed understanding of R to interpret the actions. Use R base functions where possible, or well-established utility packages.

When using a function from another package, loaded as a dependency, be explicit. For example, when calling

```
fruits <- c("apples and oranges and pears and bananas")
str_split(fruits, " and ")
```

use

```
stringr:::str_split(fruits, " and ")
```

to improve comprehensibility.

Write efficient code

Notwithstanding the need to write comprehensible code, sometimes compromising on efficiency to do so, there is no need to make code unnecessarily or inadvertently inefficient.

Big inefficiencies can arise from the simplest mistakes. A common one is to compute and recompute a variable inside a loop when that variable or a component of it does not depend on the loop index.

```
for (i in 1:nLoc(x)) {
  n <- nInd(x)
  var[i] <- sum(as.matrix(x)[i,])/n
}
```

This code is very inefficient for a number of reasons. First, it calculates `n` over and over again for each iteration of the loop, when it need only be calculated once. Any calculations not involving the index `i` should be taken outside the loop.

Second, it calls upon a major deficiency in R, where object types are not declared. R must interrogate the object on each encounter to determine what it is and handle it appropriately. This interrogation of `x` by the `as.matrix` function, and the conversion of `x` to a matrix is repeated for each iteration of the loop, with a dramatic loss of performance.

Third, there is a calculation involving `n` which is not subject to index `i` and so does not need to be repeated for each iteration of the loop.

A much improved approach is

```
mat <- as.matrix(x)
n <- nInd(x)
loc <- nLoc(x)
for (i in 1:loc) {
  var[i] <- sum(mat[i,]
}
var <- var/n
```

A little more wasteful of memory, but a great improvement in computation time.

R affectionados will suggest alternatives that are even more efficient, and adegenet affectinados yet more efficient approaches drawing apon adegenet accessors.

Data Structure

dartR relies on the SNP data being stored in a compact form using a bit-level coding scheme. SNP data coded in this way are held in a genlight object that is defined in R package adegenet (Jombart, 2008; Jombart and Ahmed, 2011). Refer to the tutorial prepared by Jombart and Collinson (2015), called *Analysing genome-wide SNP data using adegenet 2.0.0*.

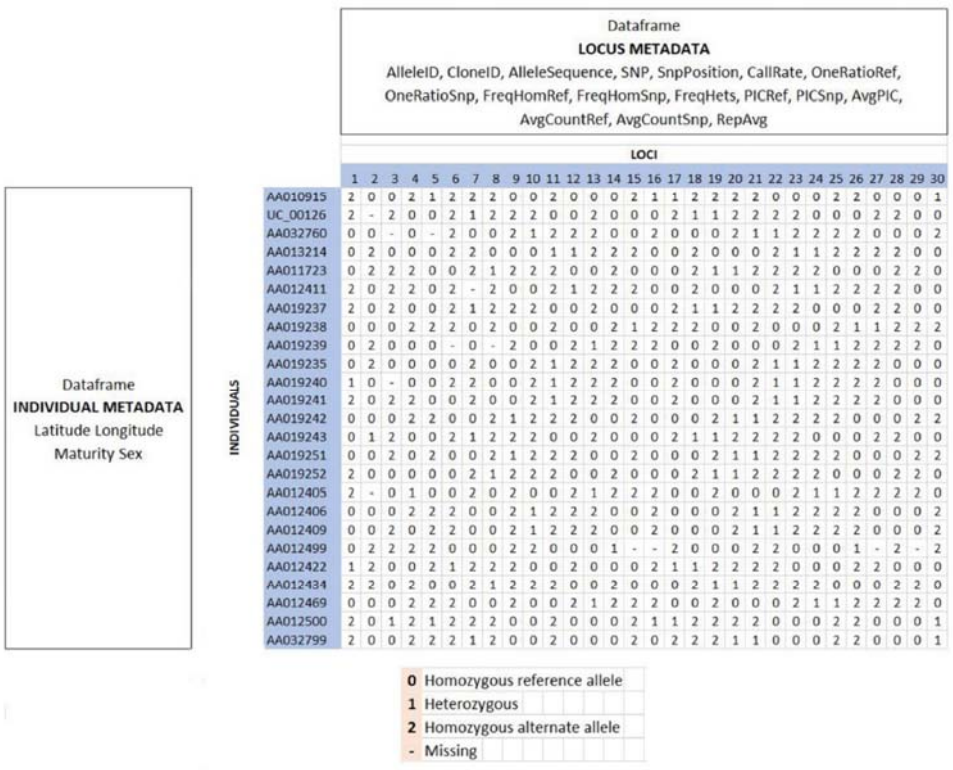
Note that for SNP data, the values are 0 for homozygous reference, 1 for heterozygous, 2 for homozygous alternate, NA for missing, with ploidy set universally to 2.

Note that for SilicoDArT data (sequence tag present/absent) that the values are set to 0 for absent, 1 for present, NA for missing, with ploidy set universally to 1.

The genelight objects used in dartR not only have the SNP/SilicoDArT data, but also allow for attachment of locus metadata to the loci, and attachment of individual metadata to the individuals/samples. This is represented diagrammatically below.

Locus metadata

The locus metadata included in the genlight object are those provided as part of your DArT PL report. These metadata are obtained from the DArT PL csv file when it is read in to the genlight object. The locus metadata are held in an R data.frame that is associated with the SNP data as part of the genlight object.



In addition, dartR calculates an estimate of read depth, at the time of reading the data in, and stores this in the locus metadata.

These metadata variables are held in the genlight object as part of a data.frame called loc.metrics, which can be accessed by your scripts as follows:

```
# Only first 10 entries shown
  gl@other$loc.metrics$RepAvg[1:10]
```

```
## [1] 1.000000 1.000000 1.000000 1.000000 0.989950 1.000000 0.993274
## [8] 1.000000 1.000000 0.980000
```

You can check the names of all available loc.metrics via:

```
names(gl@other$loc.metrics)
```

```
## [1] "AlleleID" "CloneID" "AlleleSequence" "SNP"
## [5] "SnpPosition" "CallRate" "OneRatioRef" "OneRatioSnp"
## [9] "FreqHomRef" "FreqHomSnp" "FreqHets" "PICRef"
## [13] "PICSnp" "AvgPIC" "AvgCountRef" "AvgCountSnp"
## [17] "RepAvg" "clone" "uid" "rdepth" "maf"
```

You can add new metrics by simple assignment.

```
  gl@other$loc.metrics$newvariable <- vector
```

Individual metadata

Individual (=specimen/sample) metadata are typically user specified. Individual metadata are held in a second dataframe associated with the SNP data in the genlight object. See the figure above.

Two special individual metrics are:

`id`	Unique identifier for the individual or specimen that links back to the physical sample
`pop`	A label for the biological population from which the individual was drawn

These special metrics can be accessed in your scripts using:

```
pop(gl)
popNames(gl)
indNames(gl)
```

You can check the names of all available `ind.metrics` via:

```
names(gl@other$ind.metrics)
```

```
[1] "id"       "pop"      "lat"      "lon"      "sex"      "maturity" "collector" "location"  "basin"    "drainage"
```

They can be accessed by your scripts in the following way:

```
# Only first 10 entries shown
  gl@other$ind.metrics$sex[1:10]
```

```
[1] Male    Male    Male    Male    Unknown Male    Female  Female  Male    Female
Levels: Female Male Unknown
```

You can add new metrics by simple assignment.

```
gl@other$ind.metrics$newvariable <- vector
```

Flags

A number of flags (scalar variables, type logical) are also stored in the genlight object, to indicate if the locus metadata are current, or if they require recalculation.

```
testset.gl@other$loc.metrics.flags
```

```
AvgPIC OneRatioRef OneRatioSnp PICRef PICSnp CallRate   maf FreqHets FreqHomRef FreqHomSnp
  TRUE         TRUE         TRUE   TRUE   TRUE     TRUE  TRUE     TRUE         TRUE         TRUE
monomorphs
     FALSE
```

Your scripts will need to attend to these flags. For example, if you remove monomorphic loci by direct coding, not uses `gl.filter.monomorphs`, then you will need to set the monomorphs flag to `TRUE`. In particular, if you delete individuals or populations from the data set, a number of the associated metrics will no longer be accurate. You will need to set the appropriate flags to `FALSE` in your script. The function `gl.recalc.metrics` may be useful.

History

The dartR genlight object also has its history recorded, and your scripts need to attend to maintaining this history should the genlight object be modified.

```
# ADD TO HISTORY
  nh <- length(x@other$history)
  x@other$history[[nh + 1]] <- match.call()
```

Traps for the unwary

The SNP or SilicoDArT data can be accessed by converting the genlight object to a matrix using as.matrix or by accessing the genlight object directly using adegenet accessors. Caution needs to be exercised to be sure that any manipulations to the genotypes are accompanied by companion manipulations to the metadata. Adegenet will not look after this for you in all instances.

Test datasets

Test datasets are included in the dartR package and you should use these for incorporating examples in your scripts. These datasets are small so that your examples will run quickly in accordance with the requirements of CRAN.

testset.gl SNP dataset

testset.gs Silico dataset

Further reading

Jombart T. and Caitlin Collins, C. (2015). Analysing genome-wide SNP data using adegenet 2.0.0. http://adegenet.r-forge.r-project.org/files/tutorial-genomics.pdf

Gruber, B., Unmack, P.J., Berry, O. and Georges, A. 2018. dartR: an R package to facilitate analysis of SNP data generated from reduced representation genome sequencing. Molecular Ecology Resources, 18:691–699

Ende