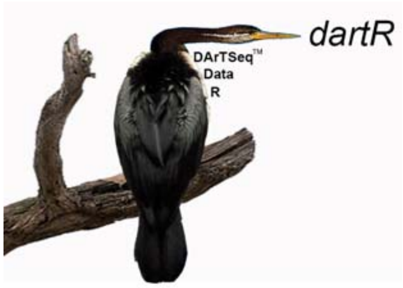
Wildlife Genomics@UC
wildlifegenomics.biomatix.org

UNIVERSITY OF CANBERRA

SNP Analysis using dartR



TechNote: For the Developer

The Institute for Applied Ecology
University of Canberra ACT 2601
Australia

Email: georges@aerg.canberra.edu.au

dartR is a collaboration between the University of Canberra, CSIRO and Diversity Arrays Technology, and is supported with funding from the ACT Priority Investment Program, CSIRO and the University of Canberra.

Contents

Purpose of this Document

We are hoping that dartR will develop through the efforts of many people who contribute scripts based on those developed in their own research. Indeed, this is how dartR was initiated, when it became clear that the scripts we were developing in our own research were of value to many around us. Rather than give advice and develop case by case solutions, we started to develop generic solutions. The package dartR was not intended to replace solutions already in existence as stand-alone packages available through CRAN or elsewhere, but rather to grease the wheels of the process between generating the data through the company DArT and analysing the data with specialized packages. This greasing the wheels included data manipulation, further filtering, and exploratory analysis. dartR was born.

There is a basic ethos to the package. While a working knowledge of RStudio is expected, users of dartR need only a rudimentary understanding of R programming. So dartR scripts have a particular structure that we hope any developers will be willing to follow. This document describes that structure. A common look and feel is delivered.

There are three ways in which a developer can contribute to dartR. Perhaps the easiest is to contribute your script that takes a genlight object and delivers an outcome. We can then provide the wrapping. You do not need to read further.

A second way is to fork the dartR package from github, make additions and improvements, then submit a pull request. We will then examine your contribution, perhaps modify it to comply with the structure outlined below, retain you as the author but add a custodian, and add it in to the dartR development pipeline.

A third way is for those wishing to make a substantial and ongoing contribution to dartR. This option is described in detail under the section Contributing your Work.

Function Structure

In a nutshell

A dartR function has a well specified structure, which includes

1. Introductory code based on roxygen2;
2. Examples again coded in roxygen2;
3. The initial function definition;
4. Setting the verbosity level with the help of gl.set.verbosity();
5. Validating the data passed to the function with the help of gl.check.datatype();
6. Checking that the required dependencies are in place;
7. Applying some function-specific checks on the values passed to the function;

8. Announcing the start of the function for the user with the help of utils.flag.start();
9. DO THE JOB
10. Delivering the graphical outputs
11. Saving the graphical outputs as gglplot and other outputs to the session temporary directory;
12. Adding to the history of the genlight object, for later recall;
13. Announcing the closure of the function for the user;
14. Returning any parameters
15. Closing the function

A proforma to guide you in the preparation of functions for dartR is attached as an appendix. The code we use to construct functions follows a standard format.

First there is introductory material to be interpreted by the R system to associate a name, description, authors and references to the function. This section also defines parameters to be called by the function, identifies functions from other packages to be used, other packages to be imported as dependencies, and a sample script that can be used to test the functions operation. The introductory material typically ends with an export statement to signal that the function is to be added to the NAMESPACE and so be available to users.

Let's look at this in more detail.

Provide some initial Roxygen2 documentation

dartR uses the R package *roxygen2* to document all the functions. Roxygen2 uses the first lines of the function (those lines that start with `#'`) to generate the R documentation files of the functions (.Rd files), the NAMESPACE file and the Collate field in DESCRIPTION file.

Here is an example

```
#' @name gl.report.callrate
#'
#' @title Report summary of Call Rate for loci or individuals
#'
#' @description
#' SNP datasets generated by DArT have missing values primarily
#' arising from failure to call a SNP because of a mutation at one
#' or both of the the restriction enzyme recognition sites. This
#' function reports the number of missing values for each of several
#' quantiles.
#'
#' @param x Name of the genlight object containing the SNP or
#' presence/absence(SilicoDArT) data [required].
#' @param method Specify the type of report by locus (method='loc')
#' or individual(method='ind') [default method='loc'].
#' @param plot_theme Theme for the plot. See Details for options
#'[default theme_dartR()].
#' @param plot_colors List of two color names for the borders and
#'fill of the plots [default two_colors].
```

```
#' @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]
#'
#' @details
#' The function \code{\link{gl.filter.callrate}} will filter out the
#' loci with call rates below a specified threshold.
#'
#' Tag Presence/Absence datasets (SilicoDArT) have missing values
#' where it is not possible to determine reliably if the sequence
#' tag can be called at a particular locus.
#'
#' Quantiles are partitions of a finite set of values into q subsets
#' of (nearly) equal sizes. In this function q = 20. Quantiles are
#' useful measures because they are less susceptible to long-tailed
#' distributions and outliers.
#'
#'\strong{ Function's output }
#'
#'  The minimum, maximum, mean and a tabulation of call rate
quantiles against thresholds rate are provided. Output also includes
a boxplot and a histogram to guide in the selection of a threshold
for filtering on callrate.
#'
#'  Plots and table are saved to the temporal directory (tempdir)
#'and can be accessed with the function
#'\code{\link{gl.print.reports}} and listed with the function
#'\code{\link{gl.list.reports}}. Note that they can be accessed only
#'in the current R session because tempdir is cleared each time that
#'the R session is closed.
#'
#'  Examples of other themes that can be used can be consulted in:
#'\itemize{
#'\item \url{https://ggplot2.tidyverse.org/reference/ggtheme.html}
#' and
#'\item
#'\url{https://yutannihilation.github.io/allYourFigureAreBelongToUs/
#'ggthemes/}
#'  }
#'
#'@return Returns unaltered genlight object
#'
#'@author Author: John Smith Custodian: James Wyneck -- Post to
#'\url{https://groups.google.com/d/forum/dartr}
#'
#'@examples
#' # SNP data
#'   gl.report.callrate(testset.gl)
#'   gl.report.callrate(testset.gl,method="ind")
#' # Tag P/A data
#'   gl.report.callrate(testset.gs)
#'   gl.report.callrate(testset.gs,method="ind")
#'
#'@references 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.
#'
#'@seealso \code{\link{gl.filter.callrate}}
#'
```

```
#'@family filters and filter reports
#'
#'@importFrom dplyr join
#'@import patchwork
#'
#'@export
#'
```

Note that some specific characters will interfere when roxygen creates the above files. Refer to the roxygen documentation for bells and whistles -- **https://cran.r-project.org/web/packages/roxygen2/vignettes/roxygen2.html**

Define the function

The next step in creating your dartR function is to define the function.

```
gl.filter.callrate <- function(x,
                               method = "loc",
                               threshold = 0.95,
                               mono.rm = FALSE,
                               recalc = FALSE,
                               recursive = FALSE,
                               plot = TRUE,
                               bins = 25,
                               verbose = NULL) {
```

Set the verbosity level

Package dartR allows the user to specify verbosity in two ways. They can specify it at the time of calling a function using the verbose = value. This parameter takes on one of six values:

0 – silent or fatal errors. Good for batch processing.
1 – notifies begin and end, and fatal errors.
2 – notifies warnings and progress log, in addition to above.
3 – reports progress and results summary, in addition to above.
4 – to be implemented.
5 – displays a full report.

Alternatively the verbosity level can be set as an environment variable using `verbosity <- gl.set.verbosity(value=n)`. This script will check if the user has specified the verbosity as a parameter in the call, or if not, pull it from the environment. If it cannot find it in the environment, verbosity is set to 2.

```
# SET VERBOSITY
  verbose <- gl.check.verbosity(verbose)
```

Apply checks of parameters

You now need to apply quality control checks on the user supplied parameters.

A utility script does some checks for you:

```
# GENERAL ERROR CHECKING
     datatype <- utils.check.datatype(x, accept=c(”SNP”,
       ”SilicoDArT”, verbose=verbose)
```

This function checks that the appropriate object has been supplied, that is, one of class

- genlight
- dist
- fd
- matrix

as specified in the accept= parameter. If it is a genlight object, the function ascertains whether it includes SNP data or SilicoDArT data. If none of the above, it returns the class of the object (strictly `class(x)[1]`).

The function also checks the data for loci or individuals scored as all NA, and notifies the user.

Next you should apply checks as to whether required functions are installed.

```
 # FUNCTION SPECIFIC ERROR CHECKING
pkg <- "HardyWeinberg"
if (!(requireNamespace(pkg, quietly = TRUE))) {
   stop(error("Package",pkg,"  needed  for  this  function  to  work.
       Please install it."))
}
```

You will also need to apply quality control checks on the user supplied parameters that are specific to the particular function.

```
# FUNCTION SPECIFIC ERROR CHECKING
  # Check that call rate is up to date and recalculate if necessary
    if (!x@other$loc.metrics.flags$callrate) {
      x <- utils.recalc.callrate(x, verbose = verbose)
    }
```

Flag the script start

To allow the user to follow the progress of their scripts, each function should notify when it has started, and when it has finished, for verbose = 1 or greater.

```
# FLAG SCRIPT START
   funname <- match.call()[[1]]
   utils.flag.start(func=funname,v=verbose)
```

Do the job

```
# DO THE JOB
```

Finally, we get to the point where we can write the code to meet the purpose of the function. Leave that to you.

Displaying output

We display graphical output using ggplot compatible plots, which scale nicely depending on the RStudio configuration particular users have in place. To do this we use packages ggplot and patchwork (see **https://patchwork.data-imaginist.com**) For example

```
# DISPLAY GRAPHICS
 # Boxplot
 p1 <- ggplot(ind.call.rate, aes(y=ind.call.rate)) +
     geom_boxplot(color=plot_colours[1], fill=plot_colours[2]) +
     coord_flip() +
     plot_theme + xlim(range=c(-1, 1)) +
     ylim(0,1) + ylab(" ") +
     theme(axis.text.y=element_blank(),
     axis.ticks.y=element_blank()) +
     ggtitle(title1)
 # Histogram
 p2 <- ggplot(ind.call.rate, aes(x=ind.call.rate)) +
     geom_histogram(bins=50,color=plot_colours[1],
     fill=plot_colours[2]) +
     coord_cartesian(xlim=c(0,1)) +
     xlab("Call rate") + ylab("Count") +
     plot_theme
 # Combine using package patchwork
 p3 <- p1/p2 + plot_layout(heights=c(1, 4))
 print(p3)
```

Note that dartR plots use a pre-established plot theme (`theme_dartR`), colours (`two_colors` and `three_colors`) and colour palettes (`discrete_palette`, `diverging_palette`, `convergent_palette` and `viridis_palette`), which are defined in the `zzz.r` file. To make these colours and theme available when testing your function you should source the `zzz.r` file with

```
source(paste0(getwd(),"/R/","zzz.r"))
```

Finish up

To finish up, there are a few housekeeping matters. The first is to save any plots and dataframes to the temporary directory `tempdir(),` so that they are available to the more sophisticated user. They may for instance wish to access the ggplot to modify it for publication.

```
# SAVE INTERMEDIATES TO TEMPDIR
  if(save2tmp){
# Create temp file names
    temp_plot <- tempfile(pattern
              =paste0("dartR_plot",paste0(names(match.call()),"_",a
              s.character(match.call()),collapse = "_"),"_"))
   temp_table <- tempfile(pattern =
              paste0("dartR_table",paste0(names(match.call()),"_",a
              s.character(match.call()),collapse = "_"),"_"))

# Save to tempdir
    saveRDS(<plot>, file = temp_plot)
    saveRDS(<table>, file = temp_table)
  }
```

If your function modifies a genlight object and returns the new genlight object, then you need to add your function call to the genlight object history:

```
# ADD TO HISTORY
if (class(x)[1] == "genlight") {
        nh <- length(x@other$history)
        x@other$history[[nh + 1]] <- c(match.call())
    }
```

This will enable the user to track the history of the genlight object with `gl.print.history()`, optionally replay it with `gl.play.history()`, or apply the history to another genlight object via assignment.

Print the end of the function, to match the message on function start:

```
# FLAG SCRIPT END
  if (verbose >= 1) {
    cat(report("\n\nCompleted:", funname, "\n"))
  }
```

Return what you need to:

```
# RETURN
  return(x)
```

or

```
  invisible(x)
```

depending on whether you want to script to print the returned object if the function is called as a statement rather than an assignment.

and close the function:

```
}
```

Colour code your messages to the user

dartR functions use the R package `crayon` to differentiate the types of messages within the functions:

- For fatal errors use `error()` which will print the message in red.
- For warning messages use `warn()` which will print the message in yellow.
- For reporting messages use `report()` which will print the message in green.
- For important messages use `important()` which will print the message in blue.
- For other messages as code use `code()` which will print the message in cyan.

The above colours should be used in function messages using `cat()` and `stop()`, for example:

```
cat(report("Completed:", funname, "\n"))
stop(error("Data must include Trimmed Sequences\n"))
```

Note that these colours for messages are defined in the `zzz.r` file. To make these colours available when testing your function you should source the zzz.r file with

```
source(paste0(getwd(),"/R/","zzz.r"))
```

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

Each line should be wrapped, normally at 80 characters. If you are using RStudio, there is a setting to help you do this. Go to `Tools | Global Options | Code | Display`, and select the option `Show Margin`, and set `margin column` to 80.

Try to avoid R functions that act in specific ways depending on context or 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 needs only be calculated once. Any calculations not involving the index i should be taken outside the loop.

Second, it calls in 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 aficionados might suggest alternatives that are even more efficient, and adegenet aficionados, yet more efficient approaches drawing upon adegenet accessors.

Follow best practice in Coding Style

We have talked about comprehensibility of code, through adequate commenting, but there is also the matter of style. We encourage you to follow the style guide of Hadley Wickham as outlined in Edition 1 of his book Advanced R. The relevant section can be found here -- http://adv-r.had.co.nz/Style.html.

Etiquette

When working in a team of developers, it is important to follow unwritten etiquette. In particular, do not stomp in someone else's garden. Do not modify their scripts without discussing with them the problem that needs to be solved, and what you propose to do to resolve it. When contributing to a script in which another team member is contributing, try to maintain their formatting style where possible. But basically, the message is communication, communication, communication.

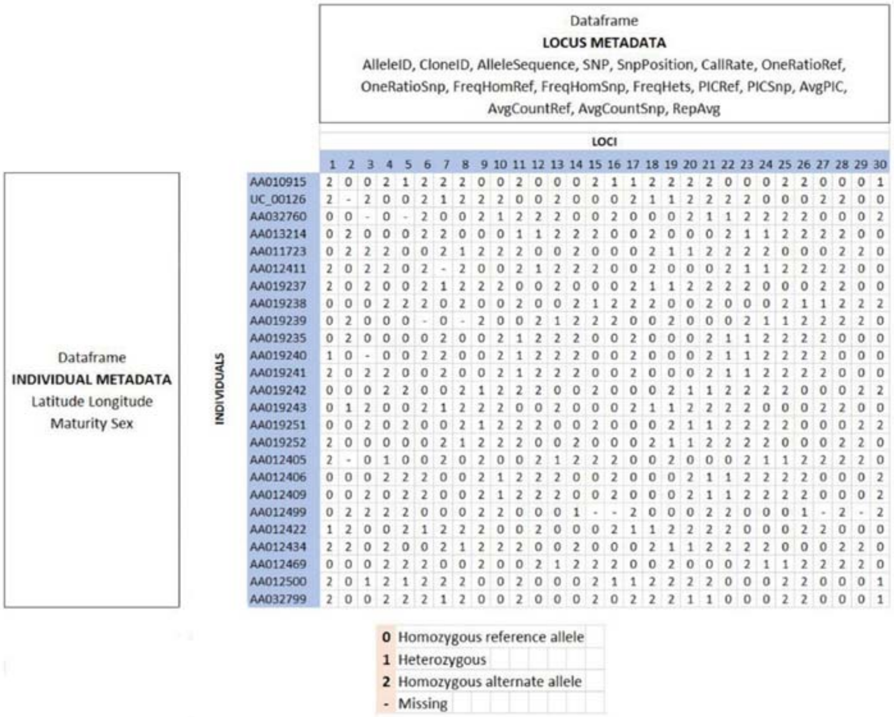
Data Structure

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. **This differs from the coding used by DArT.**

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

The genlight 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 in Fig. 1.


Dataframe
LOCUS METADATA
AlleleID, CloneID, AlleleSequence, SNP, SnpPosition, CallRate, OneRatioRef, OneRatioSnp, FreqHomRef, FreqHomSnp, FreqHets, PICRef, PICSnp, AvgPIC, AvgCountRef, AvgCountSnp, RepAvg

Dataframe
INDIVIDUAL METADATA
Latitude Longitude
Maturity Sex

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
AA010915	2	0	0	2	1	2	2	2	0	0	2	0	0	0	2	1	1	2	2	2	2	0	0	0	2	2	0	0	0	1
UC_00126	2	-	2	0	0	2	1	2	2	2	0	0	2	0	0	0	2	1	1	2	2	2	2	0	0	0	2	2	0	0
AA032760	0	0	-	0	-	2	0	0	2	1	2	2	2	0	0	2	0	0	0	2	1	1	2	2	2	2	0	0	0	2
AA013214	0	2	0	0	0	2	2	0	0	0	1	1	2	2	2	0	0	2	0	0	0	2	1	1	2	2	2	2	0	0
AA011723	0	2	2	2	0	0	2	1	2	2	2	0	0	2	0	0	0	2	1	1	2	2	2	2	0	0	0	2	2	0
AA012411	2	0	2	2	0	2	-	2	0	0	2	1	2	2	2	0	0	2	0	0	0	2	1	1	2	2	2	2	0	0
AA019237	2	0	2	0	0	2	1	2	2	2	0	0	2	0	0	0	2	1	1	2	2	2	2	0	0	0	2	2	0	0
AA019238	0	0	0	2	2	2	0	2	0	0	2	0	0	2	1	2	2	2	0	0	2	0	0	0	2	1	1	2	2	2
AA019239	0	2	0	0	0	-	0	-	2	0	0	2	1	2	2	2	0	0	2	0	0	0	2	1	1	2	2	2	2	0
AA019235	0	2	0	0	0	0	2	0	0	2	1	2	2	2	0	0	2	0	0	0	2	1	1	2	2	2	2	0	0	0
AA019240	1	0	-	0	0	2	2	0	0	2	1	2	2	2	0	0	2	0	0	0	2	1	1	2	2	2	2	0	0	0
AA019241	2	0	2	2	0	0	2	0	0	2	1	2	2	2	0	0	2	0	0	0	2	1	1	2	2	2	2	0	0	0
AA019242	0	0	0	2	2	0	0	2	1	2	2	2	0	0	2	0	0	0	2	1	1	2	2	2	2	0	0	0	2	2
AA019243	0	1	2	0	0	2	1	2	2	2	0	0	2	0	0	0	2	1	1	2	2	2	2	0	0	0	2	2	0	0
AA019251	0	0	2	0	2	0	0	2	1	2	2	2	0	0	2	0	0	0	2	1	1	2	2	2	2	0	0	0	2	2
AA019252	2	0	0	0	0	0	2	1	2	2	2	0	0	2	0	0	0	2	1	1	2	2	2	2	0	0	0	2	2	0
AA012405	2	-	0	1	0	0	2	0	2	0	0	2	1	2	2	2	0	0	2	0	0	0	2	1	1	2	2	2	2	0
AA012406	0	0	0	2	2	2	0	0	2	1	2	2	2	0	0	2	0	0	0	2	1	1	2	2	2	2	0	0	0	2
AA012409	0	0	2	0	2	2	0	0	2	1	2	2	2	0	0	2	0	0	0	2	1	1	2	2	2	2	0	0	0	2
AA012499	0	2	2	2	2	0	0	0	2	2	0	0	0	1	-	-	2	0	0	0	2	2	0	0	0	1	-	2	-	2
AA012422	1	2	0	0	2	1	2	2	2	0	0	2	0	0	0	2	1	1	2	2	2	2	0	0	0	2	2	0	0	0
AA012434	2	2	0	2	0	0	2	1	2	2	2	0	0	2	0	0	0	2	1	1	2	2	2	2	0	0	0	2	2	0
AA012469	0	0	0	2	2	2	0	0	2	0	0	2	1	2	2	2	0	0	2	0	0	0	2	1	1	2	2	2	2	0
AA012500	2	0	1	2	1	2	2	2	0	0	2	0	0	0	2	1	1	2	2	2	2	0	0	0	2	2	0	0	0	1
AA032799	2	0	0	2	2	2	1	2	0	0	2	0	0	0	2	0	2	2	2	1	1	0	0	0	2	2	0	0	0	1

0	Homozygous reference allele
1	Heterozygous
2	Homozygous alternate allele
-	Missing

Figure 1. Diagram of the structure of a genlight object in dartR. Note that it is a superset of the genlight object defined by adegenet in that it has additional data as flags. This means that all of the adegenet functions can be used on a dartR genlight object, but that you may need to run `gl.compliance.check()` if the genlight object accessed by your scripts is not generated within dartR.

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 into 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 Fig. 1).

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)
table(pop(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 using `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 but only when the genlight object has been modified.

```
# ADD TO HISTORY
if (class(x)[1] == "genlight") {
        nh <- length(x@other$history)
        x@other$history[[nh + 1]] <- c(match.call())
}
```

Traps for the unwary

The SNP or SilicoDArT data can be accessed by converting the genlight object to a matrix using the function `as.matrix()` or by accessing the genlight object directly using adegenet accessors. Caution needs to be exercised to be sure that any manipulation to the genotypes is accompanied by companion manipulation 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
```

Contributing your work

Workflow

dartR's code is hosted on GitHub and so anyone can contribute code to be incorporated into the package via well-established protocols. Individual developers willing to contribute to dartR need to understand how these protocols work in terms of a workflow and individual preferences for a pipeline to implement that workflow. Fig. 2 diagram outlines the workflow.

Figure 2 shows how you as a developer can interact with the workflow of the core dartR team. The end point of this workflow is a version of dartR distributed via the CRAN repository (orange box). Once the core dartR team has a revised version that has been adequately tested (`master`) they submit this to CRAN for their thorough checking and release. Hence the `master` branch of the dartR GitHub repository and the CRAN version of dartR remain identical. As a developer, you do not need to and should not access the `master` branch directly.

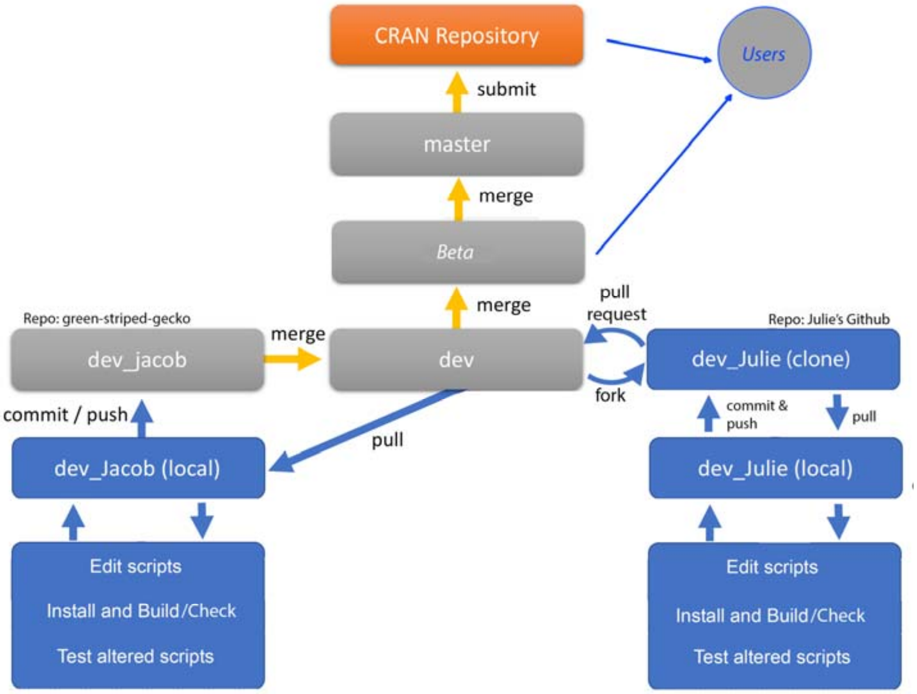


Figure 2. A workflow for the dartR development team using GitHub for version control. Grey boxes are on remote (=origin); blue boxes are local to the developer. Orange arrows show actions to be taken by the core dartR team; blue arrows show actions to be taken by individual developers. The flowchart on the left is for a regular contributor (in this case, Jacob); that on the right is for an occasional contributor (in this case, Julie). It is prudent to always follow the direction of the arrows, and not take actions that will flow in reverse directions.

Workflow for the regular contributor

The dev branch is a work in progress, incorporating all the latest changes from individual developers. It is available for downloading by end users who want the latest features at the risk of them not being fully tested. In other words, the dev branch acts as a beta version of dartR.

Individual developers who make regular contributions will request a personal branch of dev upon which to work, innovate, debug, etc. For example, developer Jacob will request and maintain a personal branch of dev called dev_jacob on the dartR remote GitHub site (green-striped-gecko). It is from this personal branch that the core dartR team will periodically evaluate and incorporate (merge) the changes made by Jacob into the dev branch available to all users and ultimately pass those changes up the workflow to the CRAN version.

To work on their branch, Jacob needs to create a local copy of their branch. They work locally on dev_jacob, and periodically commit and push their changes up to the remote version of dev_jacob.

At some point, the core dartR team will merge the remote versions of dev_jacob into dev. At this point, because both the modifications of Jacob are now incorporated into dev, both the local and remote versions of dev_jacob will become stale. The branch dev_jacob will not have the modifications made by other developers or indeed modifications made by the core dartR team.

Clearly there is the potential for the local and remote branches established by individual developers to diverge from the work that has been contributed by others. To avoid working on stale branches, regular contributors should follow the procedure below:

1. Pull the latest working version of dartR from the remote origin/dev to your local branch (*e.g*. dev_jacob).
2. Resolve any conflicts (hopefully few if you do all of this regularly).
3. Add new scripts or alter existing scripts, do a local build, and test the scripts function appropriately without error.
4. Commit any changes you have made to scripts, and include any files created by the local build.
5. Push your local branch (*e.g.* dev_jacob)to your remote branch (*e.g.* origin/dev_jacob)where it is then available to the Core dartR Team to evaluate your changes committed in 1 above, and ultimately merge with dev.

The rule is to follow the flow shown in Figure 2, and never take action that is in the reverse direction. Conflicts arising in step 4 should be minimal if you routinely follow this workflow. Ideally, resolving conflicts should be the task of the core development team.

Workflow for the occasional contributor

The occasional contributor should simply fork /dev from the dartR repository to create a clone on their personal github site. They can then pull a copy locally, work on it, and push it back to the cloned copy. Once they have finished the development (including a build and check), they need to submit a pull request to bring their work to the attention of the development team. The development team will examine the contribution for conflicts with the work of others, check the scripts for consistency, and then incorporate the new or revised scripts into /dev on the dartR repository.

The process can then begin again. It is best to avoid development in the period between when the pull request is submitted and when you fork a new copy of the /dev in order to undertake further work.

Quality Control

There are three levels of quality control over coding, needed to meet the stringent conditions placed on packages by CRAN.

The first layer of quality control occurs when you build your revised version of the package locally. Any errors at this point need to be addressed.

Then, when you push your local branch (e.g. `dev_jacob`) to your remote branch (e.g. `origin/dev_jacob`), a number of checks will be made to ensure your new scripts are consistent with the CRAN environment. This takes some time, and you will be sent an email outlining how to access the error log through Gighub. It will be challenging at first, but you will need to resolve any errors that are detected and then push your local copy to remote again.

A final check is made when the Core dartR team merge your remote copy of the package into `/dev`. They will notify you of any difficulties.

Pipelines

The above workflow shows what needs to be done and in what order to contribute to dartR via the core development team. It does not explain how to do it because each developer will have their own tools for interacting with Git and GitHub.

RStudio allows you to pull, commit and push between your local and remote branch (e.g. `dev_julie`), which provides the means of contributing new scripts, modifying existing scripts and making them available to the core dartR team for integration with `dev` and ultimately the *`Beta`* version then `master` and CRAN.

Unfortunately, the RStudio plug-in for Git does not allow you to pull from remote dev, the latest working copy, to your branch. You will need to find a way of doing this. A useful tool for executing the dartR workflow for Windows users is TortoiseGit available from **https://tortoisegit.org/download/**. This allows you to assess where your branch is in relation to the dev branch, and to pull from the `dev` branch to your local `dev_jacob` or `dev_julie` branches. TortoiseGit is fully functioned for all other tasks you might contemplate.

Alternatively, you can use the command line, for example:

```
git pull origin dev
```

will pull a copy of dev to the local branch `dev_jacob` (with the appropriate configuration in place). You can then push it up to remote branch `dev_jacob` to complete the synchronization.

Further reading

Jombart, T. (2008). adegenet: a R package for the multivariate analysis of genetic markers. Bioinformatics, 24(11), 1403-1405.

Jombart, T., & Ahmed, I. (2011). adegenet 1.3-1: new tools for the analysis of genome-wide SNP data. Bioinformatics, 27(21), 3070-3071.

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.

Wickham, H. Advanced R (1st Edition). Chapman and Hall. Style Guide -- http://adv-r.had.co.nz/Style.html.

Ende

Appendix: Sample function proforma

```
#'@name gl.<function name> (it may not contain ‘!’ ‘|’ nor ‘@’)
#'
#'@title <short informative title> (should not end in a full stop)
#'
#'@description <short description> (one paragraph, a few lines only)
#'
#'@param <name> <description> [required or default <value>]
#'@param <name> <description> [required or default <value>]
#'@param <name> <description> [required or default <value>]
#'
#'@details <detailed description of the function>
#'
#'@return <description of returned object>
#'
#'@author <author name(s)> (Post to
           \url{https://groups.google.com/d/forum/dartr})
#'
#'@examples <executable R code showing how to use the function>
#'
#'@references <reference to literature>
#'
#'@seealso \code{\link{<name of related function(s)>}}
#'
#'@family <name of the functions group>
#'
#'@importFrom <package> <function> (functions from other packages)
#'
#'@export (adds the function to the NAMESPACE file)
#'

gl.<function name> <- function(x,
                              <parameter> = < default value >,
                              plot_theme = theme_dartR(), (if plots)
                              plot_colours = two_colors, (if plots)
                              verbose = NULL) {


  # SET VERBOSITY
  verbose <- gl.check.verbosity(verbose)

  # FLAG SCRIPT START
  funname <- match.call()[[1]]
  utils.flag.start(func=funname,build="Jody",v=verbose)

  # CHECK DATATYPE
  datatype <- utils.check.datatype(x, verbose=verbose)

  # FUNCTION SPECIFIC ERROR CHECKING
  # check if packages are installed
  pkg <- "HardyWeinberg"
  if (!(requireNamespace(pkg, quietly = TRUE))) {
    stop(error("Package",pkg," needed for this function to work.
                              Please install it."))
  }

# FUNCTION SPECIFIC ERROR CHECKING

    if ( <Condition> ) {
        stop(error(" <message> \n")
    }

# DO THE JOB

    (if to be used in the function)
    # recalculate/modify @others slot (ind.metrics, loc.metrics)

    < main body of the function >

# DISPLAY OUTPUTS

(if plots and reports are to be displayed)
    # displayed plots and reports
    print( <plot> )
    print( <report> )

# SAVE INTERMEDIATES TO TEMPDIR

    (if plots and reports are to be saved to tempdir)
    # creating temp file names
      temp_plot <- tempfile(pattern = paste0("dartR_plot", paste0(
                names( match.call() ), "_" , as.character(
                match.call() ), collapse = "_"),"_"))
      temp_table <- tempfile(pattern = paste0( "dartR_table",paste0(
                names(match.call()), "_", as.character( match.call()
                ), collapse = "_"), "_"))

    # saving to tempdir
      saveRDS(p3, file = temp_plot)
      if(verbose>=2){
        cat(report(" Saving the plot in ggplot format to the
                tempfile as",temp_plot,"using saveRDS\n"))
       }
      saveRDS(df, file = temp_table)
      if(verbose>=2){
        cat(report(" Saving the outlier loci to the tempfile
                as",temp_table,"using saveRDS\n"))
       }
       if(verbose>=2){
         cat(report(" NOTE: Retrieve output files from tempdir using
                gl.list.reports() and gl.print.reports()\n"))
       }

# ADD TO HISTORY

    if (class(x)[1] == "genlight") {
        nh <- length(x@other$history)
        x@other$history[[nh + 1]] <- c(match.call())
    }

# FLAG SCRIPT END

  if (verbose >= 1) {
    cat(report("Completed:", funname, "\n"))
  }

# RETURN

  return(<object to return>) OR invisible(<object to return>)

}
```