egssimtools: an R package for the analysis of a model of speciation

This package comes with the ExplicitGenomeSpeciation program, an individual-based simulation of a speciation event with explicitly modelled diploid genomes and a flexible genotype-phenotype map. This package includes tools to facilitate the extraction and processing of the data generated by the simulation model. Throughout we will use example simulations from the `data` folder.

Installation

To install the package, click on the RStudio project `egssimtools.Rproj` to open it in RStudio, then in the "Build" tab, click "Install and Restart". The package should be installed on your machine.

Let us now load the package.

```
library(egssimtools)
library(tidyverse)
library(cowplot)
```

Each simulation is stored in a simulation folder. The content of a simulation folder depends on the settings used when running the program (see detailed description). This package introduces functions to retrieve data, but also parameters and genetic architectures saved during the simulations.

Read data

Read a single simulation

The most important function is `read_data`. This function allows the user to read the data from one simulation. The user provides the names of the variables to be read, optional parameters to be read too, and the output is returned in a data frame, suitable for further analyses or merging with other data frames. We can for example read speciation metrics through time for one simulation, and add a specific set of parameters to them:

```
root <- "../data/example_1"
data <- read_data(
  root, variables = c("time", "EI", "RI", "SI"),
  parnames = c("ecosel", "hsymmetry")
)
head(data, 4)
#>   time        EI         RI          SI hsymmetry ecosel
#> 1    0 0.6487914 0.08065974 -0.03599767         0      1
#> 2  100 0.9079386 0.09343563  0.89491305         0      1
#> 3  200 0.9267081 0.12130542  0.91147903         0      1
#> 4  300 0.9241057 0.24163096  0.91386162         0      1
```

For these variables, there is one value per time point throughout the simulation, while the values of the two parameters are unique to the simulation. Parameter values are therefore duplicated as many times as there are rows in the resulting table.

Here, all variables have the same length (one value per time point). Simulation output sometimes has different dimensions (see the main page for details). Genome-wide Fst, for example, is computed for each of the three traits at every time point, resulting in a linear, binary data file that is three times as long as `time.dat`. To append genome-wide Fst to time points, we need to split the `Fst.dat` file in three columns. We do this using the `by` argument:

```
data <- read_data(root, variables = c("time", "Fst"), by = c(1, 3))
head(data, 4)
#>   time        Fst1         Fst2         Fst3
#> 1    0 0.002065983 0.0006179214 0.0007931022
#> 2  100 0.039678401 0.0199642985 0.0236227320
#> 3  200 0.052379067 0.0250270132 0.0282848407
#> 4  300 0.059239381 0.0286670440 0.0244520078
```

The argument `by` specifies how many columns to split each data vector into. We can also use it, for example, to split locus-specific Fst (calculated on a per locus per time point basis) into as many columns as there are loci:

```
data <- read_data(root, variables = c("time", "genome_Fst"), by = c(1, 300))
head(data[, 1:6], 4)
#>   time  genome_Fst1  genome_Fst2  genome_Fst3 genome_Fst4 genome_Fst5
#> 1    0 1.225744e-07 0.0001254115 0.0000000000 0.001106494 0.001609676
#> 2  100 5.090849e-04 0.0002890381 0.0014468991 0.002417703 0.016465732
#> 3  200 3.375611e-03 0.0000000000 0.0002859601 0.000000000 0.021076349
#> 4  300 3.884728e-03 0.0060602235 0.0028030961 0.000000000 0.013249167
```

But we may also want to add to this dataset other locus-specific data such as allele frequencies or locus-specific additive variance. Then, keeping the variables as single columns would allow to easily identify multiple values measured at a specific locus at a specific time by finding its row. To bind data recorded on a per locus, per time point basis to time point data, for example, we need to duplicate the time column by the number of loci. We do this using the `dupl` argument:

```
data <- read_data(
  root, variables = c("time", "genome_Fst", "genome_freq", "genome_varA"),
  dupl = c(300, 1, 1, 1)
)
head(data, 4)
#>   time   genome_Fst genome_freq  genome_varA
#> 1    0 1.225744e-07 0.243540052 0.0015509339
#> 2    0 1.254115e-04 0.004521964 0.0000485710
#> 3    0 0.000000e+00 0.000000000 0.0000000000
#> 4    0 1.106494e-03 0.008397933 0.0002620648
```

The combination of `by` and `dupl` allows to perform all kinds of combinations of data computed at different frequencies and with different dimensions.

One important note is that `dupl` can also take a character string as one of its elements. In this case, the function is expecting this character string to be the name of a binary file to read, in which are stored the numbers of times that each element of the corresponding variable must be repeated. This is useful when different values must be duplicated different numbers of times. A typical example is the case of individual trait values through time, because the number of individual varies from one generation to the next. The syntax may look like this:

```
data <- read_data(
  root, variables = c("time", "individual_trait"), by = c(1, 3),
  dupl = list("population_size", 1)
)
```

```
head(data, 4)
#>   time individual_trait1 individual_trait2 individual_trait3
#> 1    0        -1.1038080         0.1643576        -0.2408598
#> 2    0        -0.2880653        -0.3503242         1.3731253
#> 3    0        -1.1876418         0.5833415         0.7015119
#> 4    0        -1.4026417        -0.1505369         0.2625690
```

Here, `individual_trait` is split into three columns (one per trait), then each element in the `time` column (i.e. each time point) is repeated as many times as there are individuals in this time point, which is read in `population_size.dat`. Note that `dupl` is now a list because we cannot combine a character string and an integer into the same vector.

Read parameters

Simulation parameters can be added to the resulting dataset by specifying the names of the parameters to read in `parnames`. Parameter values will be read from a parameter file, assuming it exists in the simulation folder. By default the function looks for a file called `paramlog.txt`, but this can be changed. Not all parameter values are numbers, and for this reason the parameters are read as factors. You can specify in `as_numeric` the names of parameters to convert into numeric. Some parameters are composite and contain multiple values, such as `nvertices` which contains the number of loci for each trait. Use `combine = FALSE` to split these parameters into their constituent values and give one column to each.

Note that it is possible to only read the parameters of a simulation by using the `read_parameters` function, which returns a list, e.g.:

```
read_parameters(root, c("ecosel", "hsymmetry"))
#> $hsymmetry
#> [1] "0"
#>
#> $ecosel
#> [1] "1"
```

Combining simulations

Use the `collect_sims` function to combine the data from multiple simulations. For example, we can combine speciation metrics across all simulations in our toy data folder:

```
data <- collect_sims(
  root = "../data",
  variables = c("time", "EI"),
  parnames = c("ecosel",  "hsymmetry"),
  id_column = "sim",
  level = 1,
  pattern = "example",
  verbose = FALSE)
head(data, 4)
#>   sim time        EI hsymmetry ecosel
#> 1   1    0 0.6487914         0      1
#> 2   1  100 0.9079386         0      1
#> 3   1  200 0.9267081         0      1
#> 4   1  300 0.9241057         0      1
```

This will run the `read_data` function on the simulation folders found in the `root` directory, and bind the resulting data frames together by rows. Arguments of `read_data` can therefore be used in `collect_sims`.

The argument `id_column` gives the name of the column that should identify the different simulations.

The argument `root` can be a vector of paths to each of the simulation folders, but can also (as in the example) be one (or multiple) folder(s) where to find the simulations. If folders where simulations are stored are provided, the function will recursively look into these folders for a `pattern` characterizing simulation folders. The level of recursion can be specified by `level`, where zero (the default) means that `root` is assumed to be a vector of simulation folders. Use, for example, `level = 1` to access simulations from one or several parent directories. The recursive search for simulation folders is done with the `fetch_dirs` function, which may be useful in a number of cases where folder search is needed.

The `collect_sims` function has a `check_extant` argument. If TRUE, the function will specifically look for non-missing, non-extinct, completed simulations. The two latter conditions rely on SLURM output log files being present in the folder, and so are applicable only for simulations run on the cluster. For simulations run locally, set this argument to FALSE.

Note that in many of the functions dealing with multiple simulations, there is a `verbose` argument specifying whether to display messages, and a `pb` argument to display progress bars.

Genetic architecture

The genetic architecture can be retrieved from an architecture text file. Use `read_architecture` to do so and store the genetic architecture into a list with various fields, like this:

```
arch <- read_architecture(root)
str(arch)
#> List of 6
#>  $ chromosomes: num [1:3] 0.333 0.667 1
#>  $ traits     : num [1:300] 2 2 2 1 2 2 2 2 1 1 ...
#>  $ locations  : num [1:300] 0.000513 0.002334 0.004182 0.00466 0.006378 ...
#>  $ effects    : num [1:300] 0.0814 -0.0737 0.2094 0.1177 0.136 ...
#>  $ dominances : num [1:300] 0.0464 0.106 0.0902 0.1246 0.1177 ...
#>  $ networks   :List of 3
#>   ..$ ecotrait:List of 3
#>   .. ..$ edges0 : num [1:100] 296 60 296 60 296 169 60 169 103 130 ...
#>   .. ..$ edges1 : num [1:100] 60 103 169 169 130 241 99 74 74 226 ...
#>   .. ..$ weights: num [1:100] 0.1657 0.1771 0.1196 -0.123 -0.0624 ...
#>   ..$ matepref:List of 3
#>   .. ..$ edges0 : num [1:100] 50 50 177 177 177 31 163 31 177 177 ...
#>   .. ..$ edges1 : num [1:100] 177 31 31 255 163 19 19 33 33 195 ...
#>   .. ..$ weights: num [1:100] -0.1024 0.0739 0.1093 -0.0264 -0.0278 ...
#>   ..$ neutral :List of 3
#>   .. ..$ edges0 : num [1:100] 218 196 218 218 218 218 218 128 218 128 ...
#>   .. ..$ edges1 : num [1:100] 196 263 263 288 128 276 5 5 1 1 ...
#>   .. ..$ weights: num [1:100] -0.0552 0.0992 -0.0436 0.0585 0.1773 ...
```

Note that by default this function looks for a file called "architecture.txt", but another name can be specified using the `filename` argument.

Alternatively, the function `read_genome_architecture` returns a locus-based data frame version of the architecture, therefore omitting elements specific to network edges, but with additional information such as chromosome location or degree.

```
loci <- read_genome_architecture(root)
head(loci, 4)
#>   locus    location trait     effect dominance chromosome degree
#> 1     1 0.000513158     2  0.0814406 0.0464368          1      1
```

```
#> 2     2 0.002334140     2 -0.0737139 0.1059970          1      3
#> 3     3 0.004181620     2  0.2093660 0.0901998          1      0
#> 4     4 0.004660240     1  0.1176940 0.1245750          1      0
```

Another one for networks specifically? I would have to look into network plotting functions to see what sort of stuff they can handle...

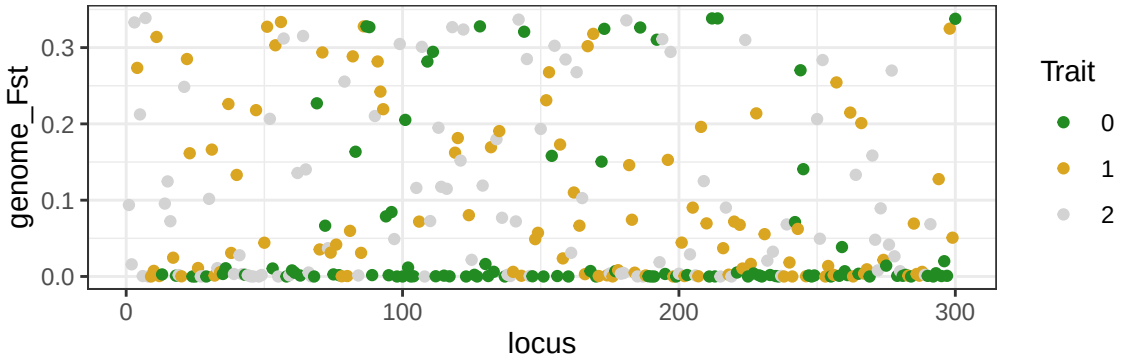
Plotting functions

Diagnostic plotting functions to eyeball single simulation results are provided.

Locus-wise variables

To visualize a genome scan of a given locus-specific variable at a specified time point (which defaults to the last time point if `t` is unspecified), for example, use:

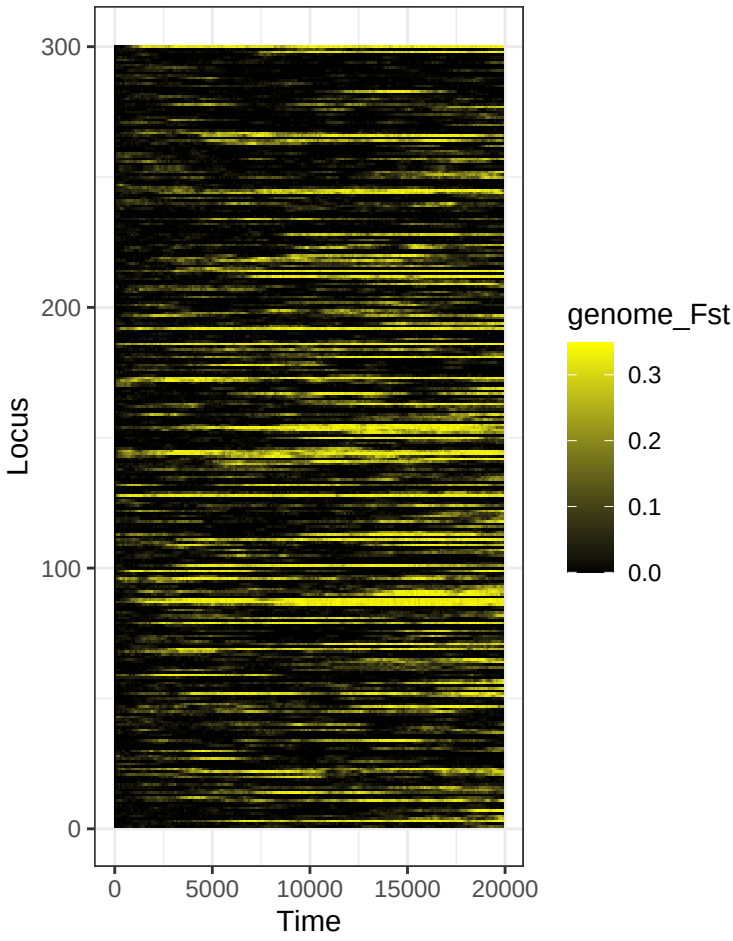
```
dplot_genome_scan(root, y = "genome_Fst")
```



The output of these quick-plotting functions are all `ggplot` objects, so they can be further customized. For more details on how to make and customize plots using `ggplot2`, please refer to relevant online documentation.

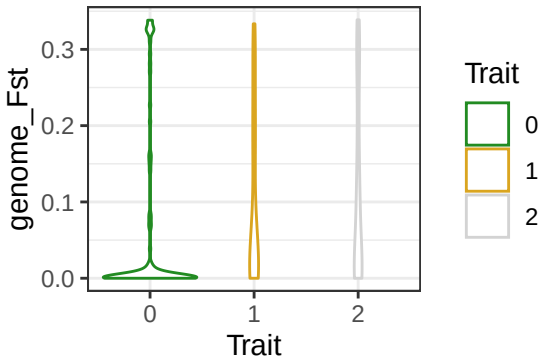
It is also possible to plot a genome scan through the entire duration of the simulation using a heatmap:

```
dplot_genome_heatmap(root, y = "genome_Fst")
```



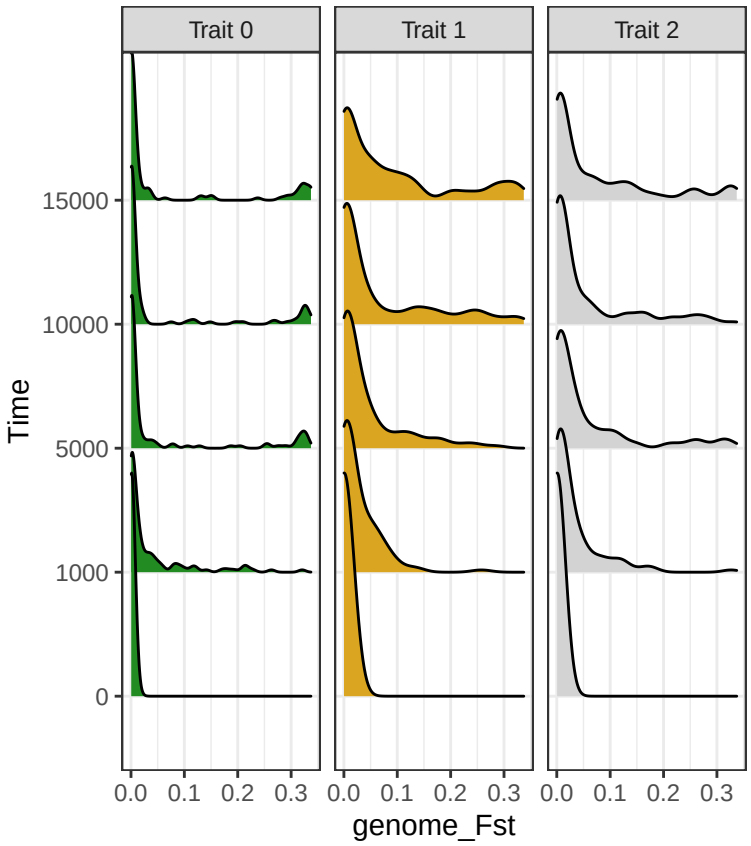
Yet another way to explore genome scans is to look at densities of a given variable across groups of loci, e.g., loci underlying different traits:

```
dplot_genome_violin(root, y = "genome_Fst", x = "trait")
```



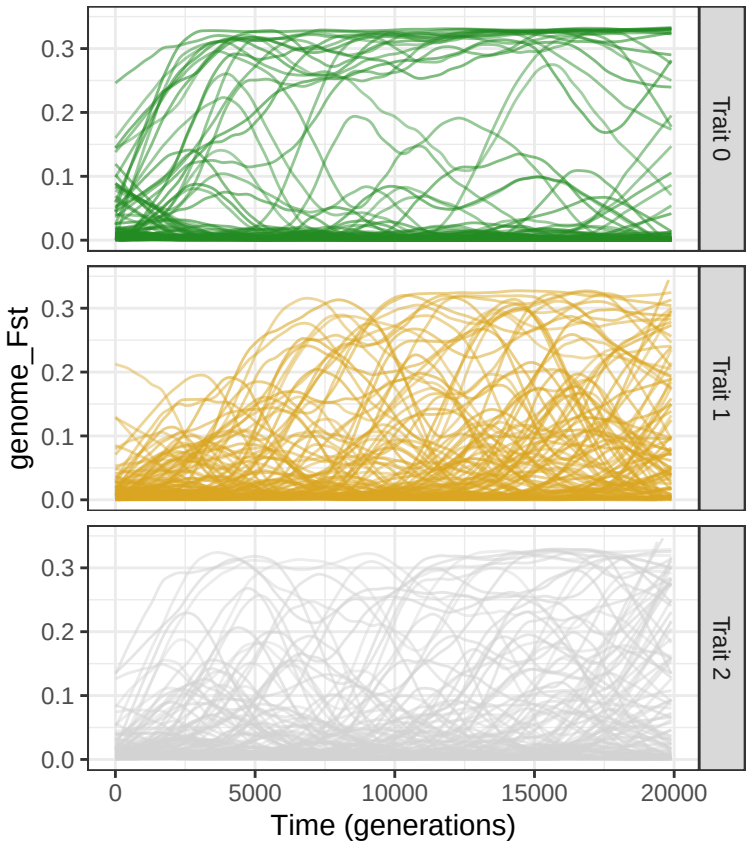
Densities across the genome can be viewed through time, using:

```
dplot_genome_ridges(root, y = "genome_Fst", times = c(0, 1000, 5000, 10000, 15000))
#> Picking joint bandwidth of 0.00673
#> Picking joint bandwidth of 0.0184
#> Picking joint bandwidth of 0.0155
```



The same goes for locus-wise lines through time:

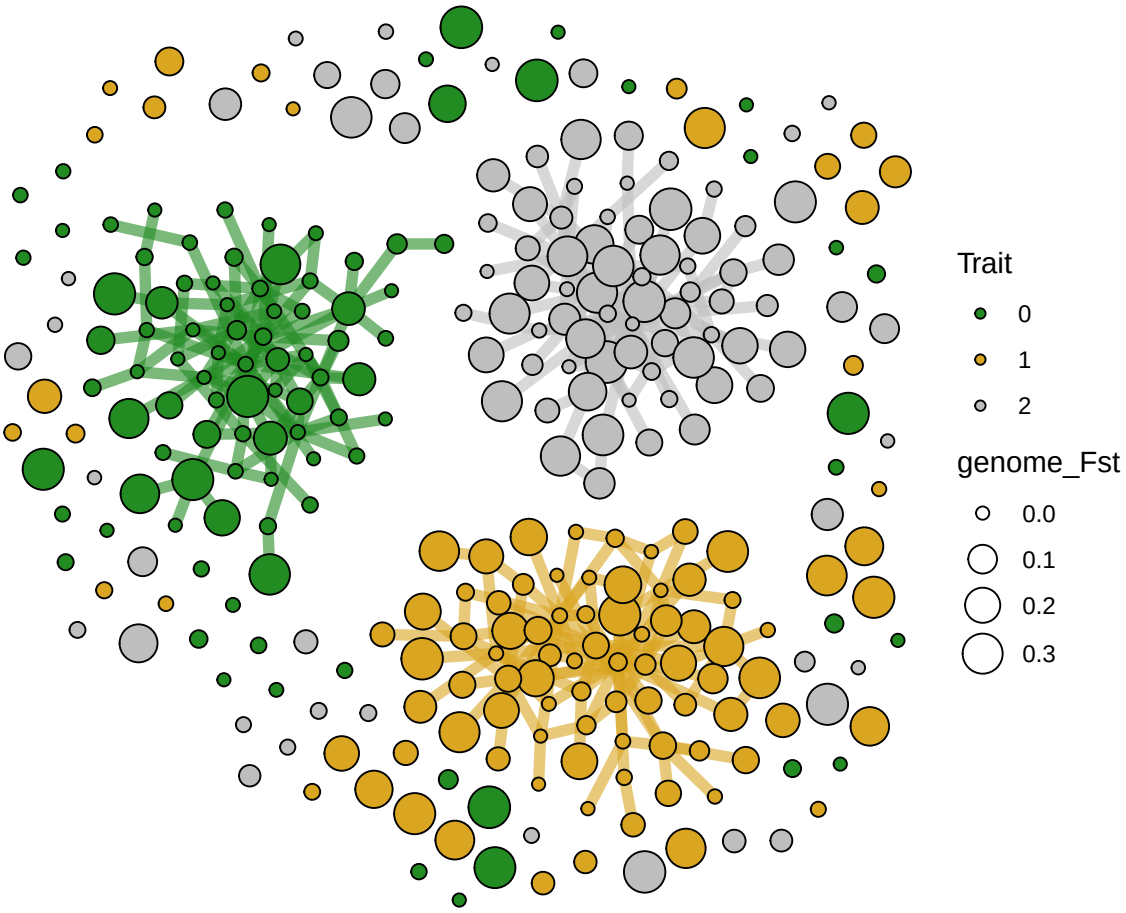
```
dplot_genome_lines(root, y = "genome_Fst")
#> Warning: Removed 399 row(s) containing missing values (geom_path).
```



which takes a `span` argument determining the span of the smoothing of the curves used.

It is also possible to map locus-wise variables onto the gene network itself:

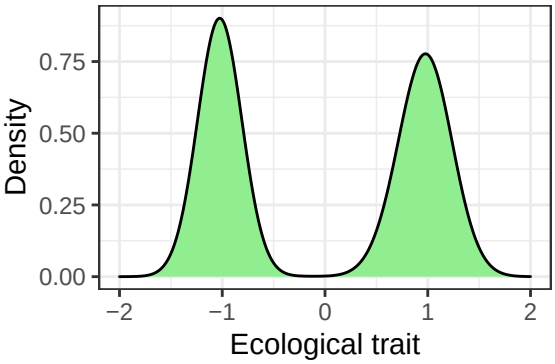
```
dplot_network(root, y = "genome_Fst")
#>
#> Attaching package: 'tidygraph'
#> The following object is masked from 'package:stats':
#>
#>     filter
#> Joining, by = c("locus", "location", "trait", "effect", "dominance", "chromosome", "degree")
```



Individual-wise variables

We can also plot individual attributes such as trait values across the population at a certain time to make sure that speciation has happened:

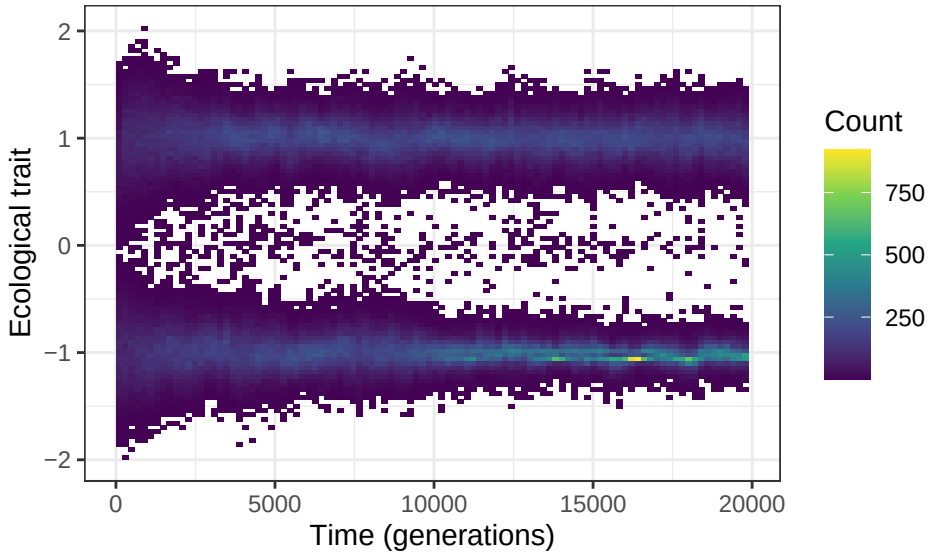
```
dplot_population_density(
  root, y = "individual_trait", by = 3, j = 1, fill = "lightgreen"
) +
  labs(x = "Ecological trait", y = "Density") +
  xlim(c(-2, 2))
```



Here, by setting `j = 1` we specify that we want to plot the density of the first trait. Because `individual_trait.dat` contains information for three traits for each individual, this long vector has to be

split into three columns in `read_data`, hence the `by = 3`. Again, a `t` argument can be specified, and if not the last time point is shown.

```
dplot_population_bin2d(
  root, y = "individual_trait", by = 3, j = 1, bins = 100
) +
  labs(x = "Time (generations)", y = "Ecological trait", fill = "Count") +
  scale_fill_continuous(type = "viridis")
```



Simulation-wise variables

It is possible, of course to plot simulation-wise observation through time or against each other. For example, we can plot the genome-wide Fst for all loci underlying the ecological trait through time (A), or we may want to show how reproductive isolation builds up with ecological divergence (B):

```
p1 <- dplot_simulation_line(root, y = "Fst", by = 3, j = 1) +
  labs(x = "Time (generations)", y = parse(text = "F[ST]"))
p2 <- dplot_simulation_line(root, y = "RI", x = "EI") +
  labs(x = "Ecological divergence", y = "Reproductive isolation")
plot_grid(p1, p2, labels = c("A", "B"))
```