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

1 Introduction

The `ExplicitGenomeSpeciation` program is an simulation of a speciation event with explicit genetics and genotype-phenotype map (see the main page for details). This vignette introduces `egssimtools`, an R package that comes with the simulation program, containing a series of tools to read and analyze the simulation data within R. Here we will show how to use it with a few use cases. We assume that the program has been run and that simulation data have already been saved. Throughout the vignette we will use example simulation data from the `data` folder.

The functions in `egssimtools` provide an interface between the data saved by the simulation, which consist in binary files (see details on the main page), and the R environment. Specifically, these functions try as much as possible to produce data frames allowing to process, plot and analyze the many types of data that can be retrieved from the simulations in multiple ways, using the `tidyverse` workflow. As such, the functions make heavy use of the `tidyverse` packages and their outputs are tailored to eing use in `tidyverse` pipelines, especially plotting with `ggplot2`. We recommend the user to be familiar with the `tidyverse` and some of its extensions, such as `patchwork`, which we will use throughout this vignette. We refer the reader to the `ggplot2` documentation to customize the plots as needed, as this is out of the scope of this vignette.

Because of the diversity of the simulation data, and the large number of ways they can be viewed, this package avoids providing ready-made functions to plot specific results directly from the simulation folders. Instead, we provide functions such that pretty much any plot can be produced in a few chunks of code only, with a common flow. We will go through examples here and explain the usage of the functions as we go.

In a first part we will cover several use-cases with short snippets of code as mentioned above, without going too much into the details of how the functions work. In a second part, we dig deeper into the working of the functions.

2 Installation

As this package comes as part of the `ExplicitGenomeSpeciation` repository, it cannot be installed from GitHub using `devtools::install_github`. Instead, you can install it by running `devtools::install()` from within the `egssimtools` folder, or by opening the project `egssimtools.Rproj` in RStudio and clicking on "Install and Restart", in the "Build" menu.

3 Use cases

Here we show how to use the package through a series of examples. First, we load the packages we will need:

```
library(egssimtools)
library(tidyverse)
library(patchwork)
```

Next, we set the paths to the our simulation data. We have a few example simulations located in the `data` folder. Each simulation is folder named `example_`, followed by a number, and contains several binary files. We first get a vector of paths to the simulation folders using `fetch_dirs`:

```
roots <- fetch_dirs("../data", pattern = "example", level = 1)
roots
#> [1] "../data/example_1" "../data/example_2" "../data/example_3"
# we are within the "vignettes" folder, hence the ".."
```

And we will use `root` to refer to the first simulation in the following examples,

```
root <- roots[1]
```

The `fetch_dirs` function recursively searches for subdirectories within a folder, based on a `pattern` to match and a recursion `level`. It is a very versatile function that can be recycled for use in many other contexts, but comes in very handy here, when the folder structure of the data is very nested, for example.

3.1 Plot speciation metrics through time

The following code chunks read and plot ecological divergence, reproductive and spatial isolation through time for a given simulation. First, we read the data in:

```
data <- read_sim(root, c("EI", "RI", "SI"))
data
#> # A tibble: 20 x 4
#>     time    EI      RI       SI
#>    <dbl> <dbl>   <dbl>    <dbl>
#>  1     0 0.662 -0.0892 -0.00245
#>  2   100 0.909  0.151   0.901
#>  3   200 0.932  0.226   0.897
#>  4   300 0.943  0.196   0.918
#>  5   400 0.938  0.190   0.898
#>  6   500 0.942  0.192   0.901
#>  7   600 0.946  0.0847  0.902
#>  8   700 0.939  0.147   0.895
#>  9   800 0.942  0.222   0.900
#> 10   900 0.936  0.195   0.903
#> 11  1000 0.945  0.200   0.898
#> 12  1100 0.943  0.208   0.904
#> 13  1200 0.943  0.191   0.903
#> 14  1300 0.947  0.116   0.894
#> 15  1400 0.949  0.0775  0.907
#> 16  1500 0.946  0.0571  0.893
#> 17  1600 0.949  0.0994  0.904
#> 18  1700 0.956  0.0870  0.898
#> 19  1800 0.955  0.115   0.895
#> 20  1900 0.955  0.0116  0.901
```

Here, `read_sim` reads `EI.dat`, `RI.dat` and `SI.dat` data files from the simulation and assembles them into a data frame. This function also reads `time.dat` by default.

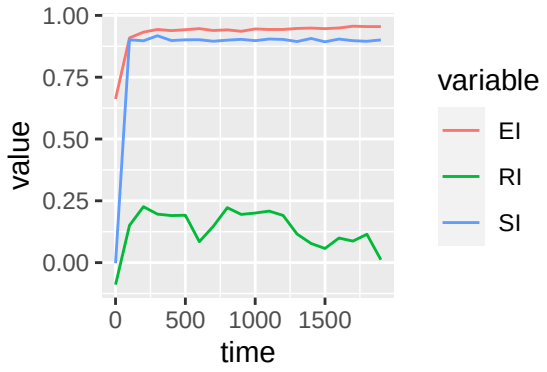
Next, we pivot the data frame to the long format (as opposed to the wide format in the tidyverse nomenclature) so the three variables are gathered into a single column (this will make plotting easier):

```
data <- pivot_data(data, c("EI", "RI", "SI"))
data
#> # A tibble: 60 x 3
#>     time variable    value
#>    <dbl> <chr>       <dbl>
```

```
#>  1     0 EI         0.662
#>  2     0 RI        -0.0892
#>  3     0 SI        -0.00245
#>  4   100 EI         0.909
#>  5   100 RI         0.151
#>  6   100 SI         0.901
#>  7   200 EI         0.932
#>  8   200 RI         0.226
#>  9   200 SI         0.897
#> 10   300 EI         0.943
#> # ... with 50 more rows
```

We can now plot lines through time for each variable using `ggplot2`:

```
ggplot(data, aes(x = time, y = value, color = variable)) +
  geom_line()
```



3.2 Plot trait distributions through time

Here we want to plot "bin2d" plots, which are heatmap-density plots, of individual trait values through time. We first read the data:

```
data <- read_pop(root, "individual_trait", by = 3)
data
#> # A tibble: 33,953 x 4
#>     time individual_trait1 individual_trait2 individual_trait3
#>    <dbl>             <dbl>             <dbl>             <dbl>
#>  1     0             -1.20            0.0673            0.958
#>  2     0             -0.885            0.138           -0.0359
#>  3     0             -0.493           -0.0451            0.585
#>  4     0             -1.02             0.128             0.898
#>  5     0             -1.11            -0.237             0.430
#>  6     0             -1.00            -0.131            -0.0151
#>  7     0             -1.30            -0.415             0.255
#>  8     0             -0.803            0.190             0.698
#>  9     0             -0.743           -0.0497           -0.0427
#> 10     0             -0.596            0.384             0.822
#> # ... with 33,943 more rows
```

Here, `read_pop` is the equivalent of `read_sim` for individual-wise data, where the resulting data frame has one row per individual per time point. This format is ensured by specifying `by = 3`, meaning that the content

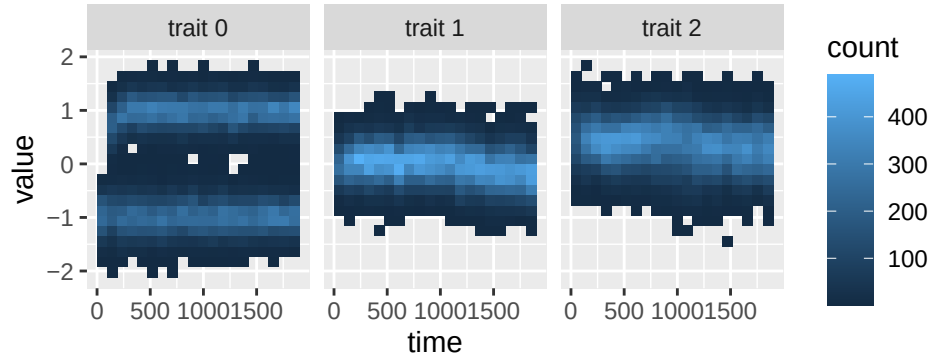
of the `individual_trait.dat` file must be splitted into three columns, one for each trait.

Next, we pivot the data to the long format to be able to plot all three traits in one go.

```
cols <- paste0("individual_trait", 1:3)
newnames <- paste0("trait ", 0:2) # to match the C++ numbering of traits
data <- pivot_data(data, cols, newnames = newnames)
data <- data %>% rename(trait = "variable")
data
#> # A tibble: 101,859 x 3
#>     time trait     value
#>    <dbl> <fct>     <dbl>
#>  1     0 trait 0 -1.20
#>  2     0 trait 1  0.0673
#>  3     0 trait 2  0.958
#>  4     0 trait 0 -0.885
#>  5     0 trait 1  0.138
#>  6     0 trait 2 -0.0359
#>  7     0 trait 0 -0.493
#>  8     0 trait 1 -0.0451
#>  9     0 trait 2  0.585
#> 10     0 trait 0 -1.02
#> # ... with 101,849 more rows
```

Here, we gathered the three trait-columns into one, and used the `newnames` argument of the `pivot_data` function to replace the levels `individual_trait1`, `individual_trait2` and `individual_trait3` by `0`, `1` and `2`, within this column. We also renamed this column `trait`, which will make our plot more intuitive. We can now plot the distribution of trait values through time, facetted by trait:

```
ggplot(data, aes(x = time, y = value)) +
  geom_bin2d(bins = 20) +
  facet_grid(. ~ trait)
```



3.3 Compare genome-wide Fst across simulations

Here, we want to plot the genome-wide Fst for each trait and each simulation. Because we want to read data from multiple simulations, we use `collect_data` instead of `read_*`:

```
variables <- c("time", "Fst")
data <- collect_data(
  roots, variables, by = c(1, 3), check_extant = FALSE, level = 0
)
```

```
#> Reading data...
data
#> # A tibble: 60 x 5
#>    sim    time    Fst1     Fst2     Fst3
#>    <chr> <dbl>   <dbl>    <dbl>    <dbl>
#>  1 1         0 0.00158 0.000659 0.000451
#>  2 1       100 0.0371  0.0167   0.0165
#>  3 1       200 0.0454  0.0176   0.0228
#>  4 1       300 0.0509  0.0232   0.0247
#>  5 1       400 0.0517  0.0257   0.0254
#>  6 1       500 0.0568  0.0280   0.0304
#>  7 1       600 0.0608  0.0268   0.0328
#>  8 1       700 0.0617  0.0351   0.0338
#>  9 1       800 0.0646  0.0411   0.0379
#> 10 1       900 0.0646  0.0472   0.0440
#> # ... with 50 more rows
```

The `collect_data` function assembles data from multiple simulations into a single data frame. It internally calls `fetch_dirs`, hence the `level` argument, and so it can go find simulation folders in a nested folder structure provided a certain `pattern` in the naming.

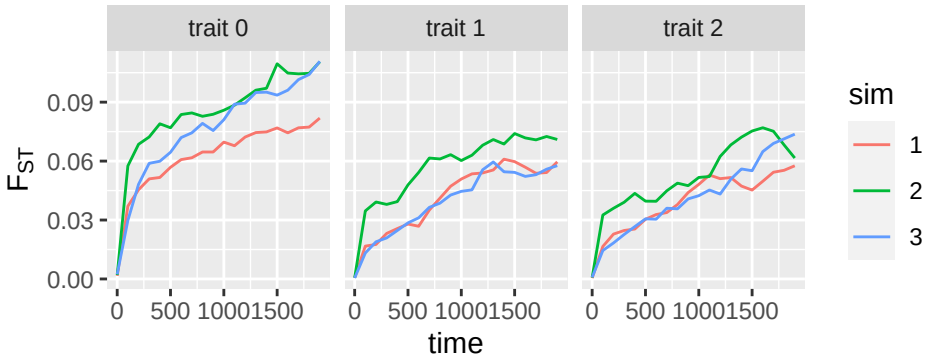
Here, we explicitly provided `time` as a variable to read, in contrast to our calls to `read_sim` and `read_pop` in the previous sections. This is because these previous functions are wrappers around a more general function called `read_data`, which does not read in `time.dat` by default. The `collect_data` function internally calls `read_data` and is the equivalent of this function for multiple simulations. It does not have a simplified equivalent for specific types of data, unlike `read_data`.

This also means that the `by` argument must be provided for each variable separately. Here, the `time.dat` file contains a single column while `Fst.dat` must be splitted into three columns, one for each trait.

The `check_extant` option can be used to find out which simulations did succesfully complete and retain only those. It specifically looks for simulations that went extinct before the end, or that did not run at all.

Again, we pivot the data by gathering the `Fst` columns into one, and we plot the data by facetting by trait:

```
data <- pivot_data(data, paste0("Fst", 1:3), newnames = newnames)
data <- data %>% rename(trait = "variable")
data
#> # A tibble: 180 x 4
#>    sim    time trait      value
#>    <chr> <dbl> <fct>      <dbl>
#>  1 1         0 trait 0 0.00158
#>  2 1         0 trait 1 0.000659
#>  3 1         0 trait 2 0.000451
#>  4 1       100 trait 0 0.0371
#>  5 1       100 trait 1 0.0167
#>  6 1       100 trait 2 0.0165
#>  7 1       200 trait 0 0.0454
#>  8 1       200 trait 1 0.0176
#>  9 1       200 trait 2 0.0228
#> 10 1       300 trait 0 0.0509
#> # ... with 170 more rows
ggplot(data, aes(x = time, y = value, color = sim)) +
  geom_line() +
  facet_grid(. ~ trait) +
  ylab(parse(text = "F[ST]"))
```



3.3.1 Compare genome scans across simulations

Here we want to plot the genome-wide scan of Fst values at the end of the simulation, for each simulation. Again we use `collect_data` to combine data from multiple simulations:

```
data <- collect_data(
  roots, c("time", "genome_Fst"), dupl = c(300, 1), check_extant = FALSE,
  level = 0, architecture = TRUE
)
#> Reading data...
data
#> # A tibble: 18,000 x 10
#>    sim    time genome_Fst locus location trait  effect dominance chromosome
#>    <chr> <dbl>      <dbl> <int>    <dbl> <fct>   <dbl>     <dbl>      <int>
#>  1 1         0  0.0000528     1  0.00309 0     -0.0793   0.0245           1
#>  2 1         0  0.000304      2  0.00339 1      0.103    0.0223           1
#>  3 1         0  0             3  0.00404 0      0.0940   0.0240           1
#>  4 1         0  0.00129       4  0.00622 0     -0.0632   0.0168           1
#>  5 1         0  0             5  0.00758 1     -0.0933   0.143            1
#>  6 1         0  0             6  0.00867 0      0.0703   0.00695          1
#>  7 1         0  0.0000659     7  0.00968 1      0.0778   0.0379           1
#>  8 1         0  0.00161       8  0.0121  0     -0.107    0.0900           1
#>  9 1         0  0.00188       9  0.0159  0     -0.0734   0.105            1
#> 10 1         0  0.000534     10  0.0233  2     -0.0219   0.0370           1
#> # ... with 17,990 more rows, and 1 more variable: degree <dbl>
```

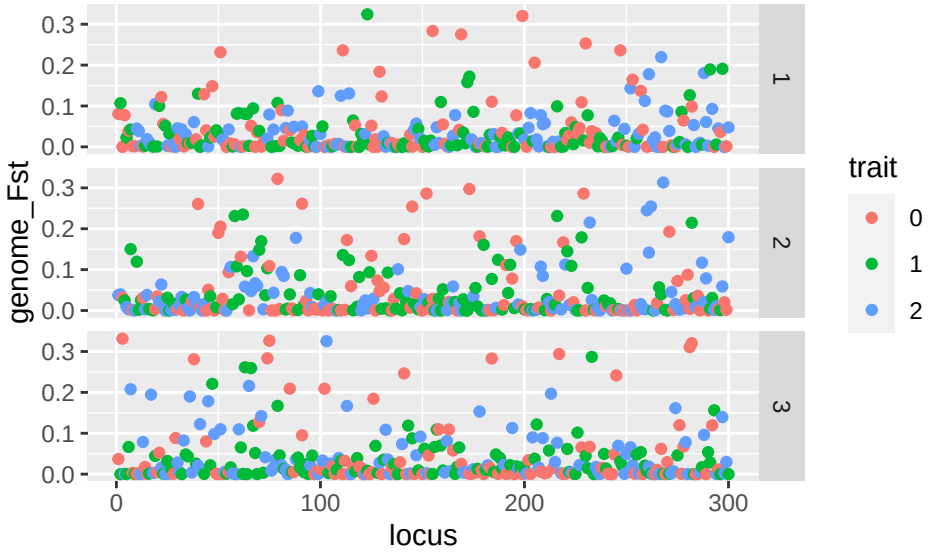
Here, however, `genome_Fst.dat` contains Fst values for each locus at each time step, and there are 300 loci in the simulations. Because we want to keep the locus as the unit of observation here, we cannot split `genome_Fst` to bring it down to the same size as `time`. Instead, we have to duplicate the `time` column as many times as there are loci. We do this with the `dupl` arguments, which takes how many times each variable should be duplicated.

We also introduce here the `architecture` argument, which is relevant only when locus-specific data are being read. If TRUE, the function will read the genetic architecture file accompanying each simulation and append it to the data, thus providing more information about the loci, such as their position, encoded trait or effect sizes.

Note that beause `genome_Fst` represents a single column, the data is already in the long format so we do not need to use `pivot_data` here.

We now make sure to only keep the last time point for each simulation, and we plot the genome scans:

```
data <- data %>% filter(time == last(time))
ggplot(data, aes(x = locus, y = genome_Fst, color = trait)) +
  geom_point() +
  facet_grid(sim ~ .)
```



3.3.2 Compare Fst through time across traits and simulations

Here we want to produce line plots showing the trajectories through time of all the loci, facetted by trait, in multiple simulations. We could choose to show traits and simulations as two facetting dimensions in `facet_grid`, but for the sake of the example we will show each simulation in a separate plot and assemble them using `patchwork`.

As always, we start with reading the data:

```
data <- collect_data(
  roots, c("time", "genome_Fst"), dupl = c(300, 1), check_extant = FALSE,
  level = 0, architecture = TRUE
)
#> Reading data...
data <- data %>% mutate(trait = str_replace(trait, "^", "trait "))
data
#> # A tibble: 18,000 x 10
#>    sim    time genome_Fst locus location trait  effect dominance chromosome
#>    <chr> <dbl>      <dbl> <int>    <dbl> <chr>   <dbl>     <dbl>      <int>
#>  1 1         0  0.0000528     1  0.00309 trai~ -0.0793   0.0245           1
#>  2 1         0  0.000304      2  0.00339 trai~  0.103    0.0223           1
#>  3 1         0  0             3  0.00404 trai~  0.0940   0.0240           1
#>  4 1         0  0.00129       4  0.00622 trai~ -0.0632   0.0168           1
#>  5 1         0  0             5  0.00758 trai~ -0.0933   0.143            1
#>  6 1         0  0             6  0.00867 trai~  0.0703   0.00695          1
#>  7 1         0  0.0000659     7  0.00968 trai~  0.0778   0.0379           1
#>  8 1         0  0.00161       8  0.0121  trai~ -0.107    0.0900           1
#>  9 1         0  0.00188       9  0.0159  trai~ -0.0734   0.105            1
#> 10 1         0  0.000534     10  0.0233  trai~ -0.0219   0.0370           1
#> # ... with 17,990 more rows, and 1 more variable: degree <dbl>
```

In order to make one plot per simulation, we will first prepare a function to make such a plot, which we will then repeatedly use for each simulation using the group-nest-map worflow of the tidyverse. The plotting function should take only one argument, which should be a data frame representing the subset of our data corresponding to a single simulation. If we pretend that `data` is such as subset, then the function

```
plot_this <- function(data) {

  ggplot(data, aes(x = time, y = genome_Fst, alpha = factor(locus))) +
    geom_line() +
    guides(alpha = FALSE) +
    facet_grid(trait ~ .)

}
```

will plot Fst trajectories of all loci, facetted by trait, for that simulation.

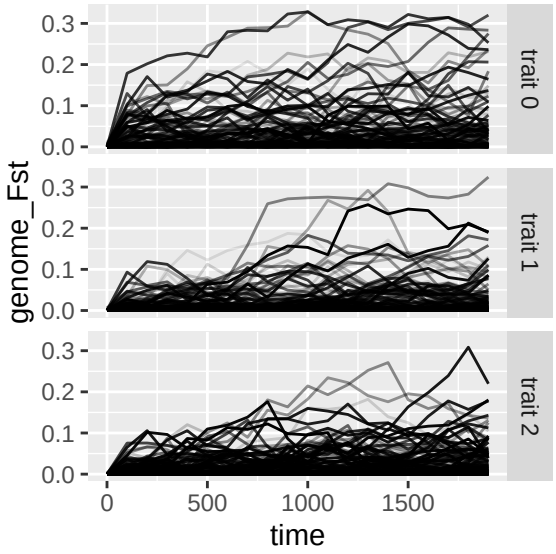
To apply the function to all simulations, we proceed as follows:

```
data <- data %>%
  group_by(sim) %>%
  nest() %>%
  mutate(fig = map(data, plot_this))
data
#> # A tibble: 3 x 3
#> # Groups:   sim [3]
#>   sim   data                fig
#>   <chr> <list>              <list>
#> 1 1     <tibble [6,000 x 9]> <gg>
#> 2 2     <tibble [6,000 x 9]> <gg>
#> 3 3     <tibble [6,000 x 9]> <gg>
```

This is a typical tidyverse workflow. Here, we first group our dataset by simulation and nest ("compress") it, such that each row corresponds to a simulation and the data for each simulation has been "compressed" into a list-column, called `data`. The data has not disappeared, it is just contained into the different elements of the list column `data`.

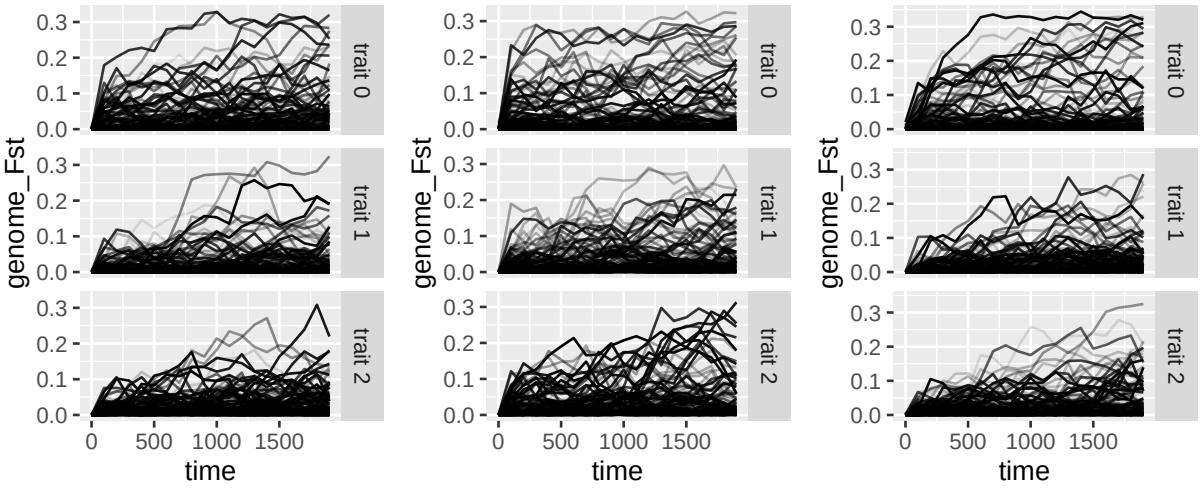
So, each element of the `data` column is a subset of the dataset that corresponds to a single simulation, which is what we want to apply our `plot_this` function to. This is done by using `mutate` to create a new list-column called `fig`, which will contain multiple `ggplot` objects. This new list-column is constructed by calling `map` from the `purrr` package, which will take care of applying `plot_this` to each element of the list-column `data`. We can check individual plots, for example, using:

```
data$fig[[1]]
```



We can assemble the multiple plots of the `fig` column with `patchwork`,

```
wrap_plots(data$fig)
```



Alternatively, we could save the different plots as separate figures by doing:

```
data$figname <- sprintf("sim%s.png", 1:3)
save_this <- function(figname, fig) {
  ggsave(figname, fig, width = 4, height = 3, dpi = 300)
}
#data <- data %>% mutate(saved = walk2(figname, fig, save_this))
# uncomment to actually save the plots
```

where we first prepare a column of file names for each figure, and then apply a custom `save_this` function to each combination of figure (`fig`) and file name (`figname`) using `walk2` (similar to `map`).

3.4 Guided tour of `read_data`

Here we present the pivotal function of the package.

The simulation outputs data with many different units of observation: some data are recorded as one value every time point, such as `EI`, while others are recorded as one value per individual per time point, one one locus per time point. It is even possible to save the actual genome sequences of each individual, resulting in one value per locus per individual, per time point (but this takes a massive amount of space). The `egssimtools` package provides an interface between this messy collection of data formats and a consistent, standardized format in R following the tidyverse recommendations.

This interface is encapsulated within the `read_data` function, arguably the most important function of the package. `read_data` reads data from binary files saved by the simulation and combines them into tables with a specific unit of observation, such as *simulation-wise*, *individual-wise* or *locus-wise*. These different formats can be generated using the following functions, respectively:

- `read_sim`
- `read_pop`
- `read_genome`

which are wrappers around `read_data` (see the use cases in the previous section). But `read_data` is more flexible and it is possible to use it instead, for example:

```
data <- read_data(
  root,
  variables = c("time", "individual_trait", "individual_ecotype"),
  by = c(1, 3, 1),
  dupl = list("population_size", 1, 1),
  parnames = c("ecosel", "hsymmetry")
)
data
#> # A tibble: 33,953 x 7
#>     time individual_trai~ individual_trai~ individual_trai~ individual_ecot~
#>    <dbl>            <dbl>            <dbl>            <dbl>            <dbl>
#>  1     0            -1.20           0.0673           0.958                0
#>  2     0            -0.885           0.138          -0.0359               1
#>  3     0            -0.493          -0.0451           0.585                1
#>  4     0            -1.02            0.128            0.898                0
#>  5     0            -1.11           -0.237            0.430                0
#>  6     0            -1.00           -0.131           -0.0151               0
#>  7     0            -1.30           -0.415            0.255                0
#>  8     0            -0.803           0.190            0.698                1
#>  9     0            -0.743          -0.0497          -0.0427               1
#> 10     0            -0.596           0.384            0.822                1
#> # ... with 33,943 more rows, and 2 more variables: hsymmetry <chr>,
#> #   ecosel <chr>
```

The `by` and `dupl` arguments are telling the function how to assemble the data.

As we saw in the use cases, `by` specifies in how many columns each variable should be split. Here, only `individual_trait` has multiple values per individual, so in order to obtain an individual-wise table at the end, we need to split this variable into three columns, one for each trait.

The `dupl` arguments specifies how many times each variable should be duplicated to be in the right format. Here, our resulting table will be individual-wise, and will therefore contain multiple individuals for each time point. So, we need to duplicate the `time` column multiple times. How many times exactly? If there were 1,000 individuals per time point, we would need to provide `1000` in `dupl`, but in our model, the number of individuals per generation is variable. `dupl` takes this into account, and it is possible to provide it with

the name of a variable into which to look up for how many times to repeat each time point. Here, we want to repeat each time point by the number of individuals in that time point, which we can find in `population_size.dat`. If you use a string in `dupl`, make sure to provide it as a `list` and not a vector.

The `read_data` function can also read specified parameter values for a given simulation and attach them to the resulting data frame by internally calling `read_parameters`.

Note that the above example is equivalent to:

```
data <- read_pop(
  root,
  variables = c("individual_trait", "individual_ecotype"),
  by = c(3, 1),
  parnames = c("ecosel", "hsymmetry")
)
data
#> # A tibble: 33,953 x 7
#>     time individual_trai~ individual_trai~ individual_trai~ individual_ecot~
#>    <dbl>            <dbl>            <dbl>            <dbl>            <dbl>
#>  1     0            -1.20           0.0673           0.958                0
#>  2     0            -0.885           0.138          -0.0359               1
#>  3     0            -0.493          -0.0451           0.585                1
#>  4     0            -1.02            0.128            0.898                0
#>  5     0            -1.11           -0.237            0.430                0
#>  6     0            -1.00           -0.131           -0.0151               0
#>  7     0            -1.30           -0.415            0.255                0
#>  8     0            -0.803           0.190            0.698                1
#>  9     0            -0.743          -0.0497          -0.0427               1
#> 10     0            -0.596           0.384            0.822                1
#> # ... with 33,943 more rows, and 2 more variables: hsymmetry <chr>,
#> #   ecosel <chr>
```

which takes care of the `dupl` argument for us because it knows that the resulting table will be individual-wise.

Similarly, the `read_genome` function, which returns a locus-wise table, will take care of the `dupl` argument by repeating the `time` column as many times as there are loci. The number of loci is fixed throughout a simulation, but you may not have its value on the top of your head. `read_genome` therefore internally calls `guess_nloci` to figure out the number of loci in the simulation (the actual number, if known, can be passed).

Locus-wise reading will often involve attaching locus-wise information from the genetic architecture to the data, using the `architecture` argument in `read_genome`. The `read_genome` and `read_data` both internally call `read_arch_genome`, which will read genetic architecture parameters from a dedicated file and append it to the data.

The `collect_data` function is the equivalent of `read_data` for multiple simulations. It works pretty much in the same way. Similar to `read_data`, `collect_data` has simplified equivalents for simulation-wise, individual-wise and locus-wise data, respectively:

- `collect_sim`
- `collect_pop`
- `collect_genome`

Note, however, that `collect_genome` cannot (yet) guess the number of loci and requires you to enter it explicitly, assuming that all simulations have the same number of loci. So, to combine simulations with different numbers of loci, make a custom assemblage using individual `read_data` or `read_genome` statements.