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

1 Introduction

The `ExplicitGenomeSpeciation` program is a 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 take a tour of the main functions in the package, and in a second part we will cover several use-cases with short snippets of code showing how to read, process, and plot the data.

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, with increasing complexity. You can find all examples in the `scripts/examples.R` script. For more specific information about each function, please refer to the documentation. Pretty much all use cases go through the same repeated phases of (1) reading, (2) processing and (3) plotting the data.

We start by loading the packages we will need.

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

```
library(tidygraph)
library(ggraph)
```

Next, we set up the path to one of our simulations.

```
root <- "../data/example_1"
```

3.1 Simulation-wise data

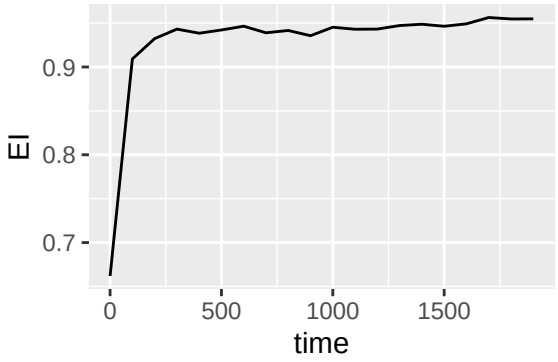
Data saved as a single value per time point, such as the degree of ecological divergence, are the easiest to read and plot. We can read these simulation-wise variables using `read_sim`:

```
data <- read_sim(root, "EI")
data
# # A tibble: 20 x 2
#     time    EI
#    <dbl> <dbl>
#  1     0 0.662
#  2   100 0.909
#  3   200 0.932
#  4   300 0.943
#  5   400 0.938
#  6   500 0.942
#  7   600 0.946
#  8   700 0.939
#  9   800 0.942
# 10   900 0.936
# 11  1000 0.945
# 12  1100 0.943
# 13  1200 0.943
# 14  1300 0.947
# 15  1400 0.949
# 16  1500 0.946
# 17  1600 0.949
# 18  1700 0.956
# 19  1800 0.955
# 20  1900 0.955
```

Here, the function `read_sim` reads the files `time.dat` (by default) and `EI.dat`, which have the same dimensions. Note that most reading functions read `time.dat` by default.

We can then use the regular `ggplot2` workflow to plot the data:

```
ggplot(data, aes(x = time, y = EI)) +
  geom_line()
```



3.2 Pivotting the data

It is possible to read multiple variables, for example:

```
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
```

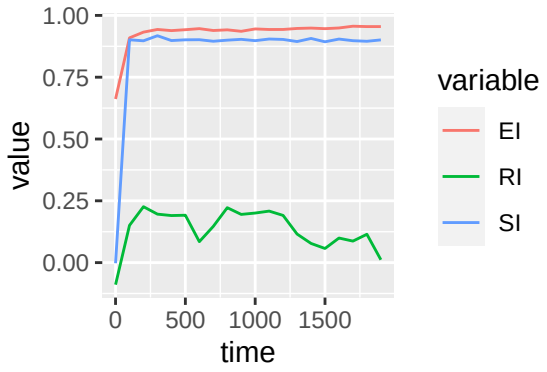
We could plot the variables indepdently from this data frame, but for practical purposes it is often handy to reshape such "wide" data frame into its "long" counterpart (according to the nomenclature of the tidyverse), where several columns are gathered in a single one. We use `pivot_data` (which internally calls `pivot_longer` from `tidyr`) for this:

```
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
```

where we specify that the `EI`, `RI` and `SI` columns must be gathered in a single one. We can then use this long data frame to plot the variables, for example, in different colors:

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



3.3 Splitting variables into several columns

Some files contain data that may have to be splitted into multiple columns in order to be arranged with other variables in a single data frame with a common unit of observation. For example, the file `Fst` contains genome-wide Fst values for each trait, therefore consisting of three values for each time point. To read this file into a simulation-wise data frame (and combine it with the `time` column), we must split it into three columns, one for each trait. The `by` argument of `read_sim` does exactly that:

```
data <- read_sim(root, "Fst", by = 3)
data
# # A tibble: 20 x 4
#     time    Fst1     Fst2     Fst3
#    <dbl>   <dbl>    <dbl>    <dbl>
#  1     0 0.00158 0.000659 0.000451
#  2   100 0.0371  0.0167   0.0165
#  3   200 0.0454  0.0176   0.0228
#  4   300 0.0509  0.0232   0.0247
#  5   400 0.0517  0.0257   0.0254
#  6   500 0.0568  0.0280   0.0304
#  7   600 0.0608  0.0268   0.0328
#  8   700 0.0617  0.0351   0.0338
#  9   800 0.0646  0.0411   0.0379
# 10   900 0.0646  0.0472   0.0440
# 11  1000 0.0697  0.0509   0.0481
```

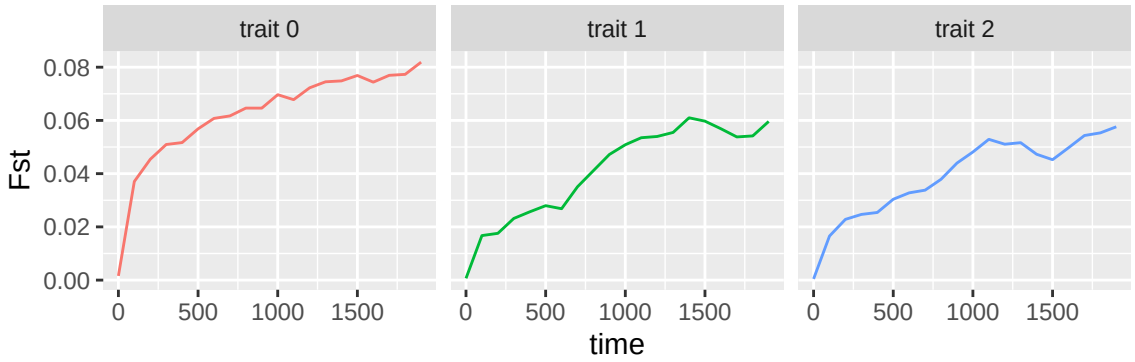
```
# 12  1100 0.0678  0.0535   0.0529
# 13  1200 0.0722  0.0540   0.0511
# 14  1300 0.0745  0.0555   0.0516
# 15  1400 0.0749  0.0610   0.0473
# 16  1500 0.0769  0.0597   0.0452
# 17  1600 0.0744  0.0569   0.0497
# 18  1700 0.0769  0.0538   0.0543
# 19  1800 0.0773  0.0542   0.0553
# 20  1900 0.0819  0.0596   0.0576
```

Again, we may want to pivot this table to the long format to facilitate plotting:

```
data <- pivot_data(data, paste0("Fst", 1:3), newnames = paste0("trait ", 0:2))
data <- data %>% rename(trait = "variable", Fst = "value")
data
# # A tibble: 60 x 3
#     time trait        Fst
#    <dbl> <fct>      <dbl>
#  1     0 trait 0 0.00158
#  2     0 trait 1 0.000659
#  3     0 trait 2 0.000451
#  4   100 trait 0 0.0371
#  5   100 trait 1 0.0167
#  6   100 trait 2 0.0165
#  7   200 trait 0 0.0454
#  8   200 trait 1 0.0176
#  9   200 trait 2 0.0228
# 10   300 trait 0 0.0509
# # ... with 50 more rows
```

Here, we used the `newnames` argument of `pivot_data` to replace the labels `Fst1`, `Fst2` and `Fst3` by the mentions `trait 0`, `trait 1` and `trait 2`, and we used the `rename` function from `dplyr` to rename the `variable` column to `trait`. This produces a nicer graph:

```
ggplot(data, aes(x = time, y = Fst, color = trait)) +
  geom_line() +
  facet_grid(. ~ trait) +
  theme(legend.position = "none")
```



3.4 Individual-wise data

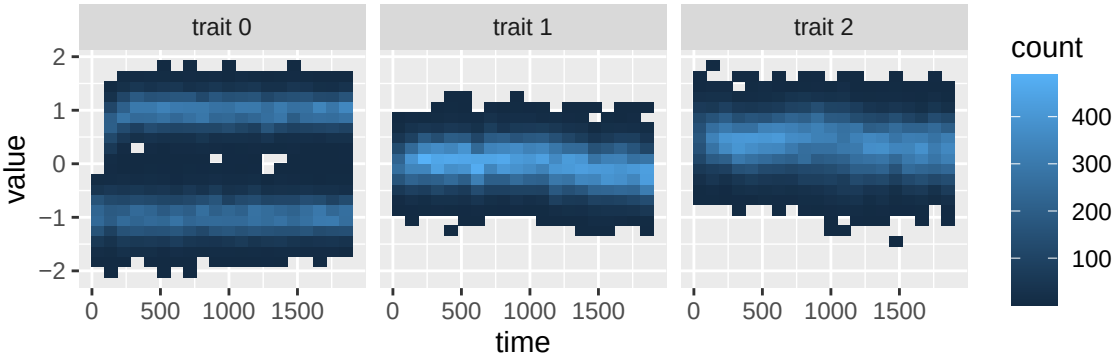
While variables such as `time`, `EI` or `Fst` are (or can easily be converted to) simulation-wise variables, some files contain data with other units of observation, such as the individual organism, thus consisting of one value per individual per time point. To read individual-wise data, use `read_pop`:

```
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, `individual_trait` consists of three trait values per individual and must be splitted into three columns to yield an individual-wise data frame.

We pivot the data again to allow plotting the three traits as different facets, except that now we show a density map of individual traits through time:

```
newnames <- paste0("trait ", 0:2)
data <- pivot_data(data, paste0("individual_trait", 1:3), newnames = newnames)
data <- data %>% rename(trait = "variable")
ggplot(data, aes(x = time, y = value)) +
  geom_bin2d(bins = 20) +
  facet_grid(. ~ trait)
```



3.5 Locus-wise data

Yet other variables are recorded for every locus at every time point. Use `read_genome` to read these data in a locus-wise data frame:

```
data <- read_genome(root, "genome_Fst")
data
# # A tibble: 6,000 x 3
#     time genome_Fst locus
#    <dbl>      <dbl> <int>
#  1     0  0.0000528     1
#  2     0  0.000304      2
#  3     0  0             3
#  4     0  0.00129       4
#  5     0  0             5
#  6     0  0             6
#  7     0  0.0000659     7
#  8     0  0.00161       8
#  9     0  0.00188       9
# 10     0  0.000534     10
# # ... with 5,990 more rows
```

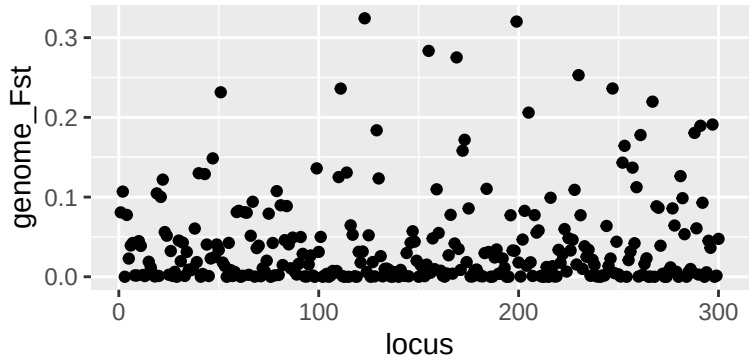
Note that a given set of data may be read in different formats. Here, `genome_Fst` is read on a locus-wise basis, but we could have read it on a simulation-wise basis, with 300 variables recorded at every time point:

```
read_sim(root, "genome_Fst", by = 300)
# # A tibble: 20 x 301
#     time genome_Fst1 genome_Fst2 genome_Fst3 genome_Fst4 genome_Fst5 genome_Fst6
#    <dbl>       <dbl>       <dbl>       <dbl>       <dbl>       <dbl>       <dbl>
#  1     0   0.0000528    0.000304   0           0.00129     0           0
#  2   100   0.0604       0.0348     0           0.0000345   0           0
#  3   200   0.0713       0.0240     0           0.00711     0.00340     0
#  4   300   0.105        0.0987     0.000275    0.0218      0.00165     0
#  5   400   0.134        0.146      0.00375     0.0107      0.000104    0.000266
#  6   500   0.185        0.123      0           0.0234      0.00789     0.000724
#  7   600   0.179        0.146      0           0.0128      0.00426     0.000275
#  8   700   0.208        0.159      0           0.0441      0.000273    0.000317
#  9   800   0.180        0.167      0           0.0343      0.0166      0.0000747
# 10   900   0.151        0.187      0.000268    0.0216      0.00756     0.00111
# 11  1000   0.189        0.184      0.000864    0.0585      0.0149      0.00389
# 12  1100   0.140        0.142      0.000293    0.0520      0.0197      0.00591
# 13  1200   0.103        0.130      0.000924    0.0219      0.0504      0.0125
# 14  1300   0.166        0.134      0           0.0658      0.0582      0.00485
# 15  1400   0.128        0.171      0.000275    0.0698      0.0655      0.0111
# 16  1500   0.107        0.129      0           0.0954      0.0681      0.0201
# 17  1600   0.0541       0.0819     0.000523    0.0898      0.0590      0.0352
# 18  1700   0.0577       0.125      0           0.0780      0.0403      0.0228
# 19  1800   0.0671       0.102      0.00154     0.0951      0.0317      0.0163
# 20  1900   0.0807       0.107      0.0000157   0.0776      0.0228      0.0393
# # ... with 294 more variables: genome_Fst7 <dbl>, genome_Fst8 <dbl>,
# #   genome_Fst9 <dbl>, genome_Fst10 <dbl>, genome_Fst11 <dbl>,
# #   genome_Fst12 <dbl>, genome_Fst13 <dbl>, genome_Fst14 <dbl>,
# #   genome_Fst15 <dbl>, genome_Fst16 <dbl>, genome_Fst17 <dbl>,
# #   genome_Fst18 <dbl>, genome_Fst19 <dbl>, genome_Fst20 <dbl>,
# #   genome_Fst21 <dbl>, genome_Fst22 <dbl>, genome_Fst23 <dbl>,
# #   genome_Fst24 <dbl>, genome_Fst25 <dbl>, genome_Fst26 <dbl>,
# #   genome_Fst27 <dbl>, genome_Fst28 <dbl>, genome_Fst29 <dbl>,
# #   genome_Fst30 <dbl>, genome_Fst31 <dbl>, genome_Fst32 <dbl>,
# #   genome_Fst33 <dbl>, genome_Fst34 <dbl>, genome_Fst35 <dbl>,
```

```
# #   genome_Fst36 <dbl>, genome_Fst37 <dbl>, genome_Fst38 <dbl>,
# #   genome_Fst39 <dbl>, genome_Fst40 <dbl>, genome_Fst41 <dbl>,
# #   genome_Fst42 <dbl>, genome_Fst43 <dbl>, genome_Fst44 <dbl>,
# #   genome_Fst45 <dbl>, genome_Fst46 <dbl>, genome_Fst47 <dbl>,
# #   genome_Fst48 <dbl>, genome_Fst49 <dbl>, genome_Fst50 <dbl>,
# #   genome_Fst51 <dbl>, genome_Fst52 <dbl>, genome_Fst53 <dbl>,
# #   genome_Fst54 <dbl>, genome_Fst55 <dbl>, genome_Fst56 <dbl>,
# #   genome_Fst57 <dbl>, genome_Fst58 <dbl>, genome_Fst59 <dbl>,
# #   genome_Fst60 <dbl>, genome_Fst61 <dbl>, genome_Fst62 <dbl>,
# #   genome_Fst63 <dbl>, genome_Fst64 <dbl>, genome_Fst65 <dbl>,
# #   genome_Fst66 <dbl>, genome_Fst67 <dbl>, genome_Fst68 <dbl>,
# #   genome_Fst69 <dbl>, genome_Fst70 <dbl>, genome_Fst71 <dbl>,
# #   genome_Fst72 <dbl>, genome_Fst73 <dbl>, genome_Fst74 <dbl>,
# #   genome_Fst75 <dbl>, genome_Fst76 <dbl>, genome_Fst77 <dbl>,
# #   genome_Fst78 <dbl>, genome_Fst79 <dbl>, genome_Fst80 <dbl>,
# #   genome_Fst81 <dbl>, genome_Fst82 <dbl>, genome_Fst83 <dbl>,
# #   genome_Fst84 <dbl>, genome_Fst85 <dbl>, genome_Fst86 <dbl>,
# #   genome_Fst87 <dbl>, genome_Fst88 <dbl>, genome_Fst89 <dbl>,
# #   genome_Fst90 <dbl>, genome_Fst91 <dbl>, genome_Fst92 <dbl>,
# #   genome_Fst93 <dbl>, genome_Fst94 <dbl>, genome_Fst95 <dbl>,
# #   genome_Fst96 <dbl>, genome_Fst97 <dbl>, genome_Fst98 <dbl>,
# #   genome_Fst99 <dbl>, genome_Fst100 <dbl>, genome_Fst101 <dbl>,
# #   genome_Fst102 <dbl>, genome_Fst103 <dbl>, genome_Fst104 <dbl>,
# #   genome_Fst105 <dbl>, genome_Fst106 <dbl>, ...
```

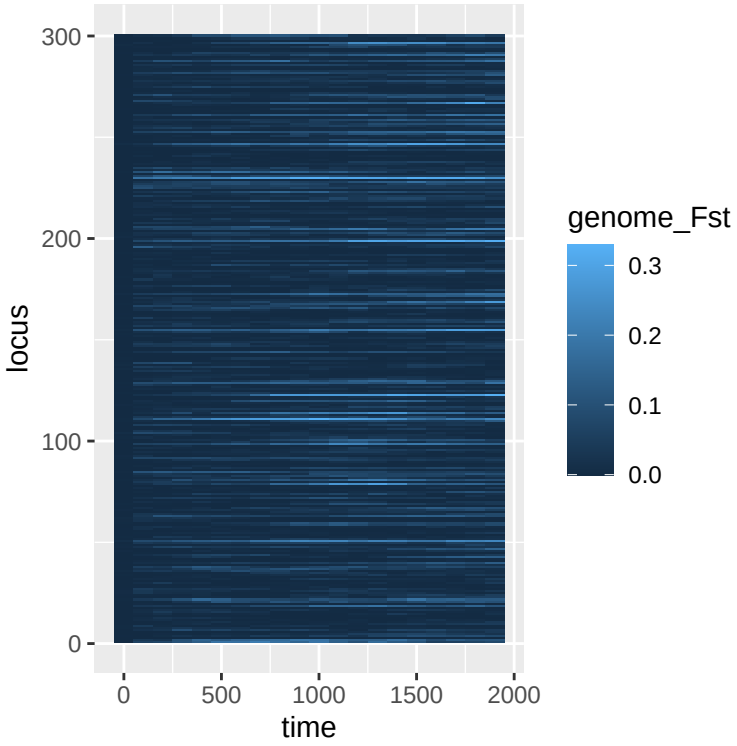
From our long, locus-wise data frame we can plot the Fst scan along the genome at the last generation with some help from `dplyr`'s `filter` function:

```
data <- data %>% filter(time == last(time))
ggplot(data, aes(x = locus, y = genome_Fst)) +
  geom_point()
```



But we could also plot Fst through time using a heatmap across the genome:

```
data <- read_genome(root, "genome_Fst")
ggplot(data, aes(x = time, y = locus, fill = genome_Fst)) +
  geom_tile()
```



3.6 Reading the genetic architecture

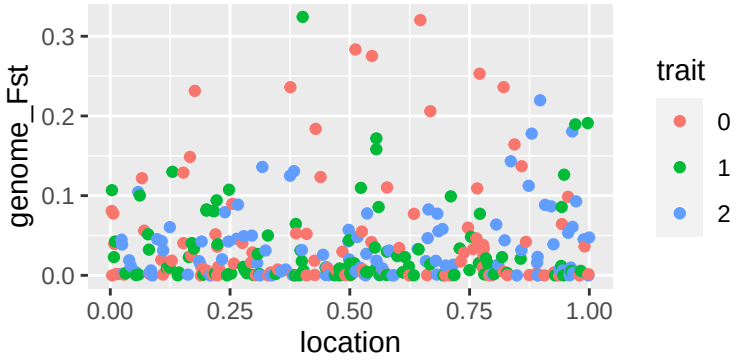
Locus-wise data are best analyzed in the light of the underlying genetic architecture of the loci, which can be read from the corresponding architecture file. Use the `architecture` argument of `read_genome` to append locus-wise architecture information to the data:

```
data <- read_genome(root, "genome_Fst", architecture = TRUE)
# Joining, by = c("time", "locus")
data
# # A tibble: 6,000 x 9
#     time genome_Fst locus location trait  effect dominance chromosome degree
#    <dbl>      <dbl> <int>    <dbl> <fct>   <dbl>     <dbl>      <int>  <dbl>
#  1     0  0.0000528     1  0.00309 0     -0.0793   0.0245           1     19
#  2     0  0.000304      2  0.00339 1      0.103    0.0223           1     14
#  3     0  0             3  0.00404 0      0.0940   0.0240           1     15
#  4     0  0.00129       4  0.00622 0     -0.0632   0.0168           1     12
#  5     0  0             5  0.00758 1     -0.0933   0.143            1     11
#  6     0  0             6  0.00867 0      0.0703   0.00695          1     13
#  7     0  0.0000659     7  0.00968 1      0.0778   0.0379           1      5
#  8     0  0.00161       8  0.0121  0     -0.107    0.0900           1     14
#  9     0  0.00188       9  0.0159  0     -0.0734   0.105            1     18
# 10     0  0.000534     10  0.0233  2     -0.0219   0.0370           1     16
# # ... with 5,990 more rows
```

Now that we have more information about each locus, we can refine our plot, for example by adding colors based on the encoded trait of each locus:

```
data <- data %>% filter(time == last(time))
ggplot(data, aes(x = location, y = genome_Fst, color = trait)) +
```

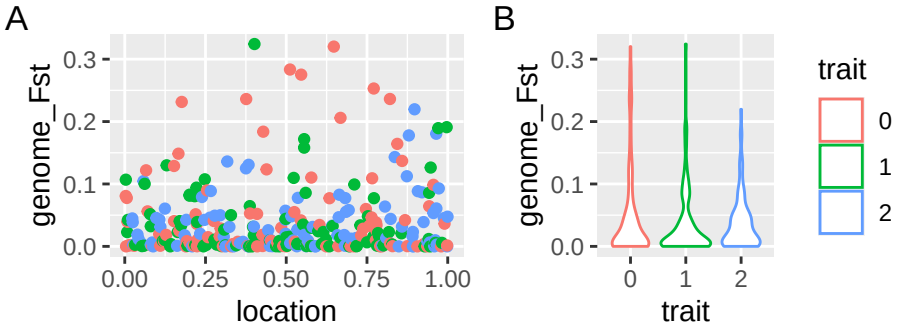
```
  geom_point()
```



3.7 Combining plots

It may be handy to pool different plots into the same figure (as in, not different facets from the same plot). We can use the `patchwork` package to do this (no new `egssimtools` function here):

```
data <- read_genome(root, "genome_Fst", architecture = TRUE)
# Joining, by = c("time", "locus")
data <- data %>% filter(time == last(time))
p1 <- ggplot(data, aes(x = location, y = genome_Fst, color = trait)) +
  geom_point() +
  theme(legend.position = "none")
p2 <- ggplot(data, aes(x = trait, y = genome_Fst, color = trait)) +
  geom_violin()

# From patchwork
wrap_plots(p1, p2, widths = c(2, 1)) + plot_annotation(tag_levels = "A")
```



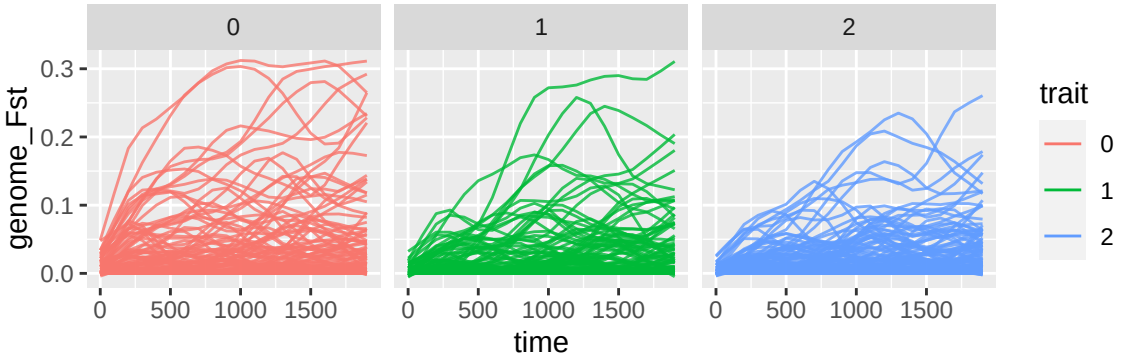
3.8 More complex plots

We can use the full breadths of tricks from `ggplot2` to make our plots look their best. For example, it is difficult to plot many lines on the same plot (without a lot of copy-and-paste), yet it is often needed in simulation studies. One way to go around that is to plot each line with a different transparency, using the `alpha` aesthetics. One can then constrain the possible `alpha` values to a very narrow range so differences in

transparency between the lines are not noticeable. Note that this is what the `gglineplot` function from the `ggsim` package does, but here we will stick to base `ggplot2`.

Let us apply this trick to plotting Fst through time on a per-locus basis. We also use the `smoothen_data` (from `egssimtools`) to smoothen our Fst curves prior to plotting.

```
data <- read_genome(root, "genome_Fst", architecture = TRUE)
# Joining, by = c("time", "locus")
data <- smoothen_data(data, "time", "genome_Fst", span = 0.3, line = "locus")
ggplot(data, aes(x = time, y = genome_Fst, alpha = factor(locus), color = trait)) +
  geom_line() +
  facet_grid(. ~ trait) +
  scale_alpha_manual(values = runif(length(unique(data$locus)), 0.79, 0.81)) +
  guides(alpha = FALSE)
```



3.9 Edge-wise data

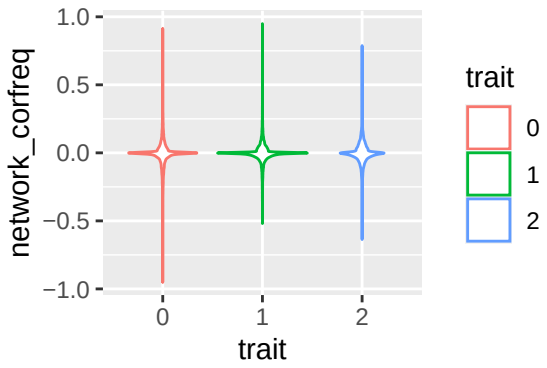
Our simulation implements a genotype-phenotype map involving a gene regulatory network, and some variables are specific to edges in the network, such as the correlation in allele frequencies between interacting loci. To read them into an edge-wise data frame, use `read_network`:

```
data <- read_network(root, "network_corfreq", architecture = TRUE)
# Joining, by = c("time", "edge")
data
# # A tibble: 60,000 x 7
#     time network_corfreq  edge  from    to    weight trait
#    <dbl>           <dbl> <int> <int> <int>     <dbl> <fct>
#  1     0          0.0528     1    62   247 -0.0722   0
#  2     0          0.0412     2    62   161 -0.00219  0
#  3     0         -0.0350     3   247   161  0.0449   0
#  4     0          0.0725     4   247   230  0.00321  0
#  5     0          0.0648     5   161   230  0.00181  0
#  6     0         -0.0229     6    62   230  0.000663 0
#  7     0         -0.0293     7    62   118  0.0644   0
#  8     0         -0.0296     8   230   118 -0.00207  0
#  9     0         -0.0348     9   247   118  0.0124   0
# 10     0         -0.0445    10   161   118  0.0118   0
# # ... with 59,990 more rows
```

Here, the `architecture` argument tells the function to read edge-wise parameters from the genetic architecture, such as the indices of the partner genes or the interaction weights, and append them to the data.

From this dataset, we can for example plot distributions of edge-wise variables across traits,

```
ggplot(data, aes(x = trait, y = network_corfreq, color = trait)) +
  geom_violin()
```



3.10 Reading data: what happens backstage?

So far we have used the functions `read_sim`, `read_pop`, `read_genome` and `read_network` to read the data. These functions are all wrappers around one same function, `read_data`. The difference between them is in the preformatting of the arguments to be passed to `read_data` in order to produce a simulation-wise, individual-wise, locus-wise or edge-wise data frame, respectively. But what happens within `read_data`?

`read_data` is a flexible function that can be used to read data into a variety of formats. It can be used instead of its simplified, preformatted counterparts, but this will typically result in longer snippets of code and should be reserved to cases when the desired output cannot be achieved using the simplified versions. For example, the following code:

```
read_data(
  root,
  c("time", "genome_Fst", "genome_Cst"),
  by = c(1, 1, 1),
  dupl = c(300, 1, 1)
)
# # A tibble: 6,000 x 3
#     time genome_Fst genome_Cst
#    <dbl>      <dbl>      <dbl>
#  1     0  0.0000528  0.0149
#  2     0  0.000304   0.000567
#  3     0  0          0.00277
#  4     0  0.00129    0.00562
#  5     0  0          0.00278
#  6     0  0          0.00503
#  7     0  0.0000659  0.00155
#  8     0  0.00161    0.00163
#  9     0  0.00188    0.0488
# 10     0  0.000534   0.0000189
# # ... with 5,990 more rows
```

is equivalent to

```
read_genome(root, c("genome_Fst", "genome_Cst"))
# # A tibble: 6,000 x 4
```

```
#     time genome_Fst genome_Cst locus
#    <dbl>      <dbl>      <dbl> <int>
#  1     0  0.0000528  0.0149        1
#  2     0  0.000304   0.000567      2
#  3     0  0          0.00277       3
#  4     0  0.00129    0.00562       4
#  5     0  0          0.00278       5
#  6     0  0          0.00503       6
#  7     0  0.0000659  0.00155       7
#  8     0  0.00161    0.00163       8
#  9     0  0.00188    0.0488        9
# 10     0  0.000534   0.0000189    10
# # ... with 5,990 more rows
```

In `read_data`, the file `time.dat` is not read in by default and must be explicitly provided. This means that the `by` argument needs to account for the presence of `time` in the `variables` argument, i.e., `by` must be provided for *all* the variables (unless all of them take a value of one, in this case leave unspecified). The `dupl` argument is being taken care of for you in the simplified versions of `read_data`, while it must be provided here. This argument determines how many times each variable should be duplicated (with copies stacked on top of each other). This matters when, even after splitting some variables into multiple columns, some are still much shorter than the rest. Here, for example, we need to duplicate the `time` column 300 times to get a viable data frame, because we have 300 loci recorded for each time point in `genome_Fst` and `genome_Cst`.

Sometimes, different values of a too short column may need to be duplicated different numbers of times. This is the case for `time` in individual-wise data, for example, because the number of individuals changes from one generation to the next, so each time point has to be duplicated by the number of individuals in that specific time point. For this reason, `dupl` accepts as some of its values the name of a variable in which to look up the number of times the corresponding column should be duplicated. For example,

```
read_data(
  root,
  c("time", "individual_trait", "individual_ecotype"),
  by = c(1, 3, 1),
  dupl = list("population_size", 1, 1)
)
# # A tibble: 33,953 x 5
#     time individual_trait1 individual_trait2 individual_trait3 individual_ecoty~
#    <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
```

looks up in `population_size.dat` the number of times each time point in the `time` column should be duplicated. Note the use of a `list` instead of a vector here, because "population_size" is not a number. Also note that here we splitted `individual_trait` into three columns but `individual_ecotype` remained a single column.

The above snippet is equivalent to:

```
read_pop(root, paste0("individual_", c("trait", "ecotype")), by = c(3, 1))
# # A tibble: 33,953 x 5
#     time individual_trait1 individual_trait2 individual_trait3 individual_ecoty~
#    <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
```

3.11 Plot a gene network

We previously saw how to read edge-wise data, but the data frame resulting from `read_network` is not enough in order to plot the actual gene network. Networks are graphs, and graphs are complicated plots to make because they do not fit into a "tidy" data frame representation, where one of the criteria for tidyness (sensu tidyverse) means that the data should have one unit of observation. This is because a graph typically has *two* equally valid units of observation: *nodes* and *edges*, both of which are needed to produce a plot. The packages `tidygraph` and `ggraph` solved this problem by introducing the `tbl_graph` object. This object contains a representation of *two* tibbles (data frames): one for nodes, the other one for edges. This sort-of-tidy object can readily be interpreted by graph-plotting functions from `ggraph`.

In `egssimtools`, the function `read_arch_network` reads the content of a genetic architecture file and returns a `tbl_graph` (unless its `as_list` argument is `TRUE`, which is the case e.g. when it is called from within `read_network`) containing locus-wise as well as edge-wise data.

So, to plot a gene regulatory network, we first read the genetic architecture as a `tbl_graph`:

```
arch <- read_arch_network(root)
arch
# # A tbl_graph: 300 nodes and 3000 edges
# #
# # A directed acyclic simple graph with 3 components
# #
# # Node Data: 300 x 7 (active)
#   locus location trait  effect dominance chromosome degree
#   <int>    <dbl> <fct>   <dbl>     <dbl>      <int>  <dbl>
# 1     1  0.00309 0     -0.0793   0.0245           1     19
# 2     2  0.00339 1      0.103    0.0223           1     14
# 3     3  0.00404 0      0.0940   0.0240           1     15
# 4     4  0.00622 0     -0.0632   0.0168           1     12
# 5     5  0.00758 1     -0.0933   0.143            1     11
# 6     6  0.00867 0      0.0703   0.00695          1     13
# # ... with 294 more rows
# #
# # Edge Data: 3,000 x 5
#    from    to   weight trait  edge
#   <int> <int>    <dbl> <fct> <int>
```

```
# 1    62   247 -0.0722  0         1
# 2    62   161 -0.00219 0         2
# 3   247   161  0.0449  0         3
# # ... with 2,997 more rows
```

This is already enough for plotting. But let us say that we want to map some locus-specific data onto the nodes of the network, for example, the locus-specific Fst at the final generation. To do that we must read these locus-specific data:

```
data_n <- read_genome(root, "genome_Fst", architecture = TRUE)
data_n <- data_n %>% filter(time == last(time))
```

and then attach them to the `tbl_graph`:

```
arch <- arch %>% activate(nodes) %>% right_join(data_n)
arch
# # A tbl_graph: 300 nodes and 3000 edges
# #
# # A directed acyclic simple graph with 3 components
# #
# # Node Data: 300 x 9 (active)
#   locus location trait  effect dominance chromosome degree  time genome_Fst
#   <int>    <dbl> <fct>   <dbl>     <dbl>      <int>  <dbl> <dbl>      <dbl>
# 1     1  0.00309 0     -0.0793   0.0245           1     19  1900  0.0807
# 2     2  0.00339 1      0.103    0.0223           1     14  1900  0.107
# 3     3  0.00404 0      0.0940   0.0240           1     15  1900  0.0000157
# 4     4  0.00622 0     -0.0632   0.0168           1     12  1900  0.0776
# 5     5  0.00758 1     -0.0933   0.143            1     11  1900  0.0228
# 6     6  0.00867 0      0.0703   0.00695          1     13  1900  0.0393
# # ... with 294 more rows
# #
# # Edge Data: 3,000 x 5
#    from    to   weight trait  edge
#   <int> <int>    <dbl> <fct> <int>
# 1    62   247 -0.0722  0         1
# 2    62   161 -0.00219 0         2
# 3   247   161  0.0449  0         3
# # ... with 2,997 more rows
```

where `right_join` attaches our `data_n` tibble to the "top-facing" tibble in the `arch` `tbl_graph`, which we set to `nodes` using the `activate` function from `tidygraph`. The alternative activation of `nodes` and `edges` allows `tbl_graph` objects to be treated as a single tibble (the activated one) by other functions, which makes their incorporation smooth inside tidyverse pipelines.

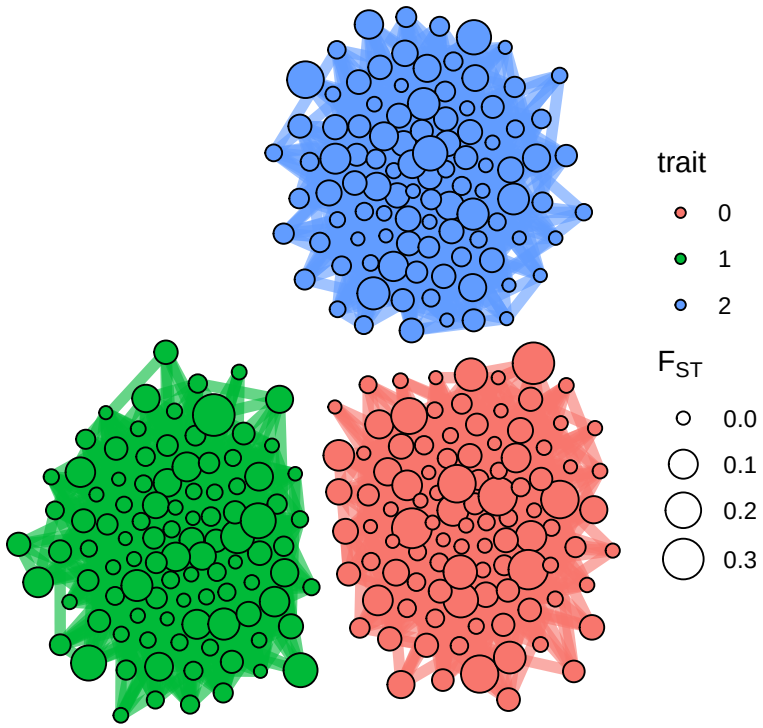
We could also map edge-specific variables onto the network, following the same logic:

```
data_e <- read_network(root, "network_corfreq", architecture = TRUE)
data_e <- data_e %>% filter(time == last(time))
arch <- arch %>% activate(edges) %>% right_join(data_e)
```

We can now plot the network using the `ggraph` package, which produces a `ggplot` object:

```
# This may take a while
ggraph(arch, layout = "graphopt", charge = 0.1, mass = 30, niter = 100000) +
  geom_edge_link(aes(color = trait), width = 2, alpha = 0.6) +
  geom_node_point(aes(fill = trait, size = genome_Fst), shape = 21) +
  scale_size_continuous(range = c(2, 7)) +
```

```
  scale_alpha(range = c(0.6, 1)) +
  labs(size = parse(text = "F[ST]"), fill = "trait") +
  theme_void() +
  guides(edge_color = FALSE)
```



Of course, one can play with the graphical parameters as needed. Visit the ggraph website for more details on how to use its functions!

3.12 Combining simulations

So far we have worked with single simulations. Some use cases may require to work with multiple simulations, however, such as looking at the effect of some parameter value. To combine data from multiple simulations, we use the `combine_data` function, which is essentially a wrapper around the `read_*` functions we previously used for single simulations:

```
# First we reset the working directory to where multiple simulations are
root <- "../data"

data <- combine_data(
  root, pattern = "example", level = 1, type = "genome",
  variables = "genome_Fst", architecture = TRUE,
  parnames = c("ecosel", "hsymmetry")
)
data
# # A tibble: 18,000 x 12
#    sim    time genome_Fst hsymmetry ecosel locus location trait  effect
#    <chr> <dbl>      <dbl> <chr>     <chr>  <int>    <dbl> <fct>   <dbl>
#  1 1         0  0.0000528 0         1          1  0.00309 0     -0.0793
#  2 1         0  0.000304  0         1          2  0.00339 1      0.103
```

```
#  3 1         0  0         0         1          3  0.00404 0      0.0940
#  4 1         0  0.00129   0         1          4  0.00622 0     -0.0632
#  5 1         0  0         0         1          5  0.00758 1     -0.0933
#  6 1         0  0         0         1          6  0.00867 0      0.0703
#  7 1         0  0.0000659 0         1          7  0.00968 1      0.0778
#  8 1         0  0.00161   0         1          8  0.0121  0     -0.107
#  9 1         0  0.00188   0         1          9  0.0159  0     -0.0734
# 10 1         0  0.000534  0         1         10  0.0233  2     -0.0219
# # ... with 17,990 more rows, and 3 more variables: dominance <dbl>,
# #   chromosome <int>, degree <dbl>
```

Here, we are combining the results of three simulations called `example_1`, `example_2` and `example_3` into one data frame. To do this, we have to give the function:

- a place to look for the simulations, here `root`
- the `level` of recursion at which the function has to search, here one means that the simulation folders are *directly* within `root`, two would mean than they are in one or multiple folders within `root`, and so on. A value of zero would mean that `root` is actually a vector of simulation folders (in our case it would be `c("../data/example_1", "../data/example_2", "../data/example3")`)
- a `pattern` to match to identify simulation folder names (if `level` is nonzero)

The function internally calls `fetch_dirs` to search for simulation folders, which comes in handy when the folder containing the simulation has some substructure (e.g. multiple batches of simulations stored in different subfolders).

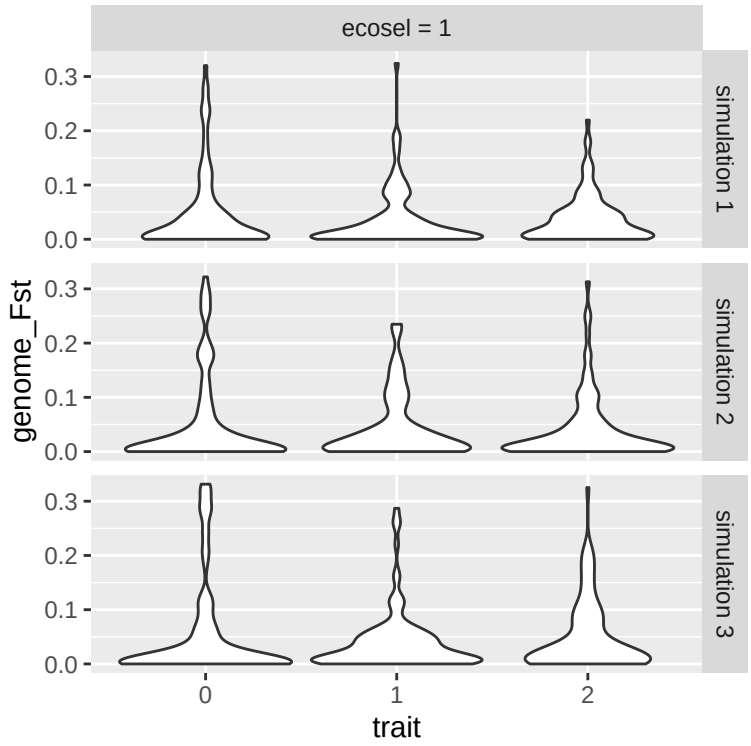
In addition, the `type` argument tells `combine_data` what version of `read_data` to use to read the multiple simulations. Here, we are internally calling `read_genome` because we are reading locus-wise Fst values from multiple simulations, and we attach to them their respective architecture data.

All other parameters passed to `combine_data` are passed to the `read_*` function you have chosen, so make sure those parameters are relevant by checking the documentation of the different functions. For example, it would not make sense to use `architecture = TRUE` if `type` is `pop`, because `read_pop` does not have an `architecture` argument.

Among the arguments passed to the `read_*` function are arguments regarding the reading of simulation parameters, which are read from a parameter file. Here, for example, we read the `ecosel` and the `hsymmetry` parameter values for each simulation. The function being called internally for that is `read_param`. All `read_*` functions can take arguments to pass on to `read_param`. We did not use them before because each simulation is generated from a single set of parameter values, but parameters start to matter when we combine data from multiple simulations together.

Once we have a data frame summing up multiple simulations, we can manipulate it and plot it just like the tibbles we get from single simulations:

```
data <- data %>% filter(time == last(time))
data <- data %>% mutate(sim = str_replace(sim, "^", "simulation "))
data <- data %>% mutate(ecosel = str_replace(ecosel, "^", "ecosel = "))
ggplot(data, aes(x = trait, y = genome_Fst)) +
  geom_violin() +
  facet_grid(sim ~ ecosel)
```



3.13 The split-apply-combine routine

We may want to produce a different plot for different subsets of a data frame, for example, for different simulations. This can be done using the split-apply-combine routine from the tidyverse. We first read the data from multiple simulations:

```
data <- combine_data(
  root, pattern = "example", level = 1, type = "genome",
  variables = "genome_Fst", architecture = TRUE
)
```

Then, we nest our tibble by simulation using some tidyverse functions:

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

This returns a nested tibble, where the data have been split between the different simulations and stored in a new list-column called `data`. The data has not disappeared, instead it is now located in a list of tibbles, with one tibble per simulation. Looping through this list-column will allow us to produce one plot per simulation. But before that, we must define a function to apply to each simulation, that will do the plotting. This

function must take a tibble as argument, because it will be called on each tibble within the `data` list-column. One example is:

```
plot_this <- function(data) {

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

}
```

This function will plot Fst values through time for each locus, facetted by trait. But in practice, any plotting routine that can work with a tibble containing data for one simulation will do.

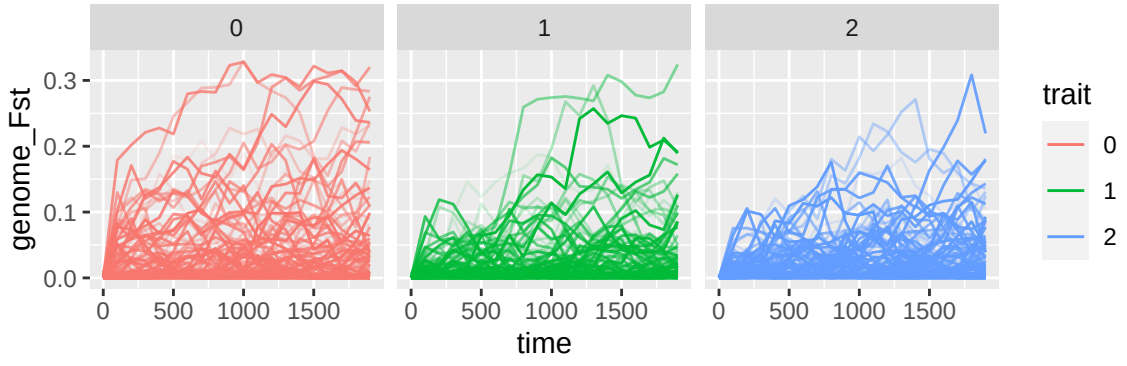
We now use the `map` function from the `purrrr` package to iterate through the `data` list-column within the `data` tibble, and apply our custom `plot_this` function to each of its tibbles:

```
data <- data %>% 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 has added a new column called `fig` to our tibble. `fig` is also a list-column, but instead of being a list of tibbles, it is a list of `ggplot` objects returned by `map` (`map` always returns a list), one for each simulation.

We check out the plots individually:

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



But we can also perform other operations, such as saving them separately in their own respective image files, using:

```
data <- data %>% mutate(figname = sprintf("sim%s.png", sim))
data
# # A tibble: 3 x 4
# # Groups:   sim [3]
```

```
#   sim   data                 fig    figname
#   <chr> <list>               <list> <chr>
# 1 1     <tibble [6,000 x 9]> <gg>   sim1.png
# 2 2     <tibble [6,000 x 9]> <gg>   sim2.png
# 3 3     <tibble [6,000 x 9]> <gg>   sim3.png
save_this <- function(x, y) ggsave(x, y, width = 4, height = 3, dpi = 300)

# This is commented to not save when rendering the vignette
# data %>% mutate(saved = walk2(figname, fig, save_this))
```

Here, we have created a new column called `figname`, which contains the names of the PNG files where each plot should be saved. We then used the `walk2` function from `purrr` to save each plot in `fig` to its respective file `figname`. Check out the `purrr` documentation for more details on how `map`, `walk` and their extensions work.

3.14 Looping through simulation folders

So far we have dealt with a situation where the data from multiple simulations fits into a single data frame. However, we may want to save the same kind of figure (e.g. a genome scan of Fst values through time) for thousands of replicate simulations, which may be too large to store in a single data frame using `combine_data`. For such cases, it may be better to loop through the different simulation folders, read the data for each of them using a `read_*` function, plot it and store the output plot in a list for later manipulation or save it directly from within the loop (using e.g. `ggsave`).

We would also run into this problem if we wanted to plot a gene network for multiple simulations, because the data for a single network does not fit within a standard tibble, and therefore it may be very clunky to try to combine the data from multiple networks into a single object.

We can get a vector of simulation folders using the `fetch_dirs` function:

```
roots <- fetch_dirs("../data", pattern = "example_", level = 1)
roots
# [1] "../data/example_1" "../data/example_2" "../data/example_3"
```

Note that the `find_extant` function can be used on this vector of simulation folder names to find out and retain the ones that did not go extinct or did not crash.

To loop through these folders and plot the data, we can e.g. refine our `plot_this` function so it operates at the level of the simulation folder, and not on a tibble. This means that `plot_this` would have to read the data too. For example, we could have:

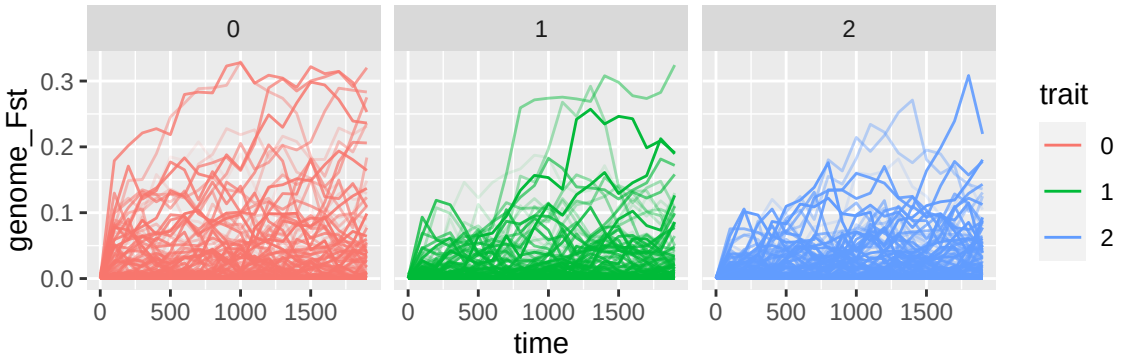
```
plot_this <- function(root) {

  # This is the only line added
  data <- read_genome(root, "genome_Fst", architecture = TRUE)

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

}
```

which we then would use within a loop through the simulation folders:

```
plots <- map(roots, plot_this)
```

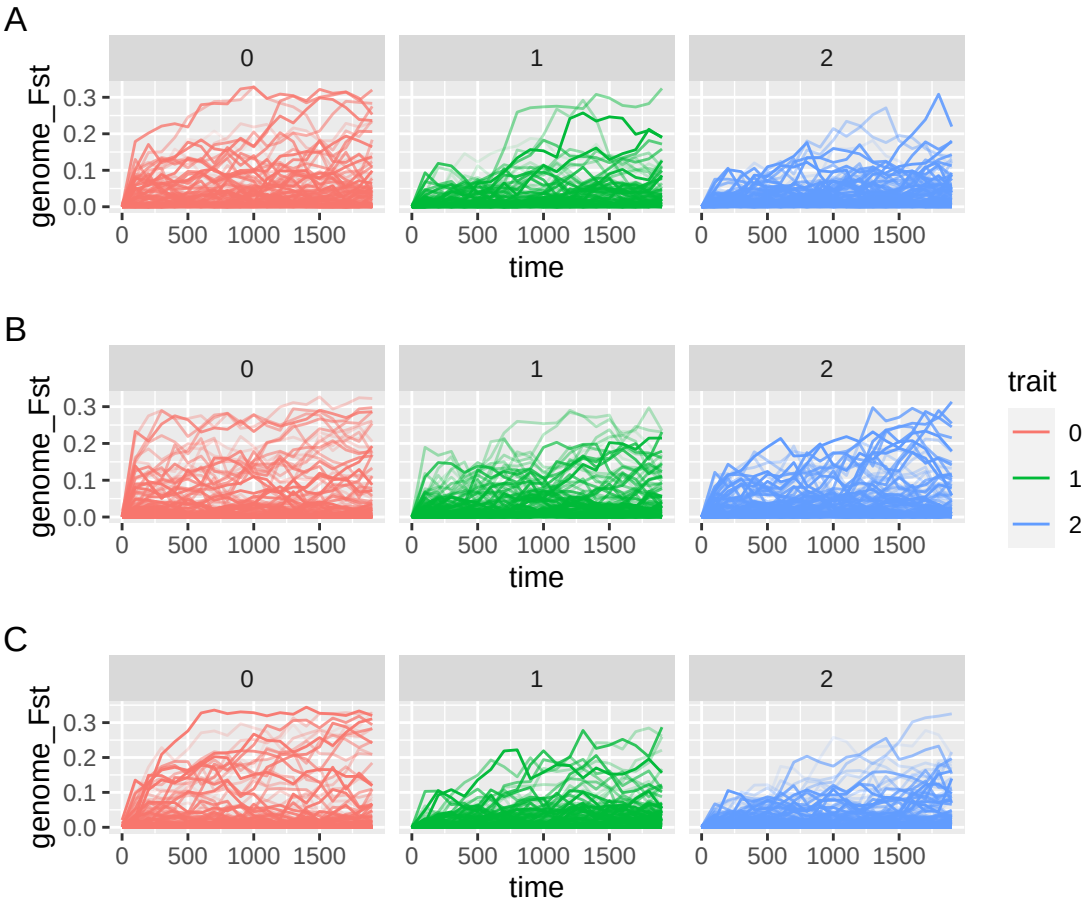
This has produced a list of `ggplot` objects:

```
plots[[1]]
```



which we can then manipulate as needed, for example, by combining them into a single figure using `patchwork`:

```
# We remove the legend of all plots but one
for (i in 2:length(plots)) {
  plots[[i]] <- plots[[i]] + theme(legend.position = "none")
}

# Then we assemble the plots with a common legend
wrap_plots(plots) +
  plot_layout(guides = 'collect', nrow = length(plots)) +
  plot_annotation(tag_levels = 'A')
```



4 To sum up

The complexity and diversity of the data saved by the `ExplicitGenomeSpeciation` simulation require an analysis toolkit that is both flexible and very standardized. The tools of the tidyverse allow for exactly that purpose with the R computing environment. As often in R, the same result can be obtained in multiple ways. For these reasons we strongly recommend its use when analyzing the data from our simulaitons, but of course other computing languages can be used (e.g. Python, MATLAB or C++), as long as they can read the binary files saved by `EGS`.