brachypoder

This is the vignette of `brachypoder`, an R package developed to help read and assemble the data simulated by the `brachypode` program.

To install the package from GitHub, use:

```
devtools::install_github("rscherrer/brachypoder")
```

To load the package, use:

```
library(brachypoder)
```

Some example data are provided with the package. To extract where they are saved, run:

```
root <- system.file("extdata", "sim-example", package = "brachypoder")
root
#> [1] "/home/raphael/R/x86_64-pc-linux-gnu-library/4.2/brachypoder/extdata/sim-example"
```

If we have a look at what is inside this folder,

```
list.files(root)
#>  [1] "architecture.txt" "brachypode"       "gmon.out"         "individuals.dat"
#>  [5] "parameters.txt"   "paramlog.txt"     "patchsizes.dat"   "popsize.dat"
#>  [9] "time.dat"         "traitmeans.dat"
```

we see a few `.dat` files. These are the files that contain the data saved from one simulation. Different variables are saved in different files, e.g. `popsize.dat` contains the total population size at every time point saved. See the repository of `\textcolor{blue}{brachypode}` for details.

Each data file contains data with different resolutions. For example, `individuals.dat` contains five values per individual and covers all individuals of all the time points saved. The data files are saved in binary format, which is one-dimensional. This means that the arrays of saved values may have to be reshaped in order to be assembled into a data set that is workable within R. This is where the `read_data` function comes in.

The function `read_data` is the core function of the package. It takes the location of the simulation data as input, the variables to read, and rules that determine how each variable should be reshaped. For example,

```
read_data(root, "popsize")
#> # A tibble: 101 x 1
#>    popsize
#>      <dbl>
#>  1      10
#>  2     192
#>  3     235
#>  4     218
#>  5     235
#>  6     218
#>  7     225
#>  8     226
#>  9     219
#> 10     256
#> # ... with 91 more rows
```

reads the binary data in `popsize.dat` back into numbers and places them into a one-column tibble. If we now run:

```
read_data(root, c("time", "popsize"))
#> # A tibble: 101 x 2
#>     time popsize
#>    <dbl>   <dbl>
#>  1     0      10
#>  2   100     192
#>  3   200     235
#>  4   300     218
#>  5   400     235
#>  6   500     218
#>  7   600     225
#>  8   700     226
#>  9   800     219
#> 10   900     256
#> # ... with 91 more rows
```

we get multiple variables (`time.dat` and `popsize.dat`) read and assembled together. Now, these two variables both have one value saved every saved time point, and so have the same number of values. When the variables to read have different dimensions (e.g. `patchsizes.dat` contains one value per patch, per site, per time point), we use the `ncols` argument:

```
read_data(root, c("time", "patchsizes"), ncols = c(1, 10))
#> # A tibble: 101 x 11
#>     time patchsizes1 patchsizes2 patchsizes3 patchsizes4 patchsizes5 patchsizes6
#>    <dbl>       <dbl>       <dbl>       <dbl>       <dbl>       <dbl>       <dbl>
#>  1     0           0          10           0           0           0           0
#>  2   100          13          29          33          39          18          22
#>  3   200          20          66          24          33          26          28
#>  4   300           8          54          36          42          32          24
#>  5   400          18          51          29          41          35          33
#>  6   500          11          51          28          48          27          12
#>  7   600          15          64          24          32          23          36
#>  8   700          15          54          31          50          19          30
#>  9   800          11          37          22          44          35          36
#> 10   900           9          54          30          53          32          30
#> # ... with 91 more rows, and 4 more variables: patchsizes7 <dbl>,
#> #   patchsizes8 <dbl>, patchsizes9 <dbl>, patchsizes10 <dbl>
```

which splits the `patchsizes` data into 10 columns, and so its number of rows becomes equal to the number of time points in `time` (the 1 tells the function not to split that column), with which they can be attached. The columns in the output data are named after their respective data file or origin, with numbers appended for each column.

In some cases some columns may need to be duplicated instead of split into multiple columns. Then, we use negative numbers in `ncols`. For example,

```
read_data(root, c("time", "patchsizes"), ncols = c(-10, 1))
#> # A tibble: 1,010 x 2
#>     time patchsizes
#>    <dbl>      <dbl>
#>  1     0          0
#>  2     0         10
#>  3     0          0
#>  4     0          0
#>  5     0          0
#>  6     0          0
#>  7     0          0
#>  8     0          0
#>  9     0          0
#> 10     0          0
#> # ... with 1,000 more rows
```

reads the same data as the previous chunk, but in a "longer" format, where each number of individuals is taken as an observation and therefore as a row. And so here, `patchsizes` was kept in a single column but each value in `time` was duplicated as many times as there are patches and sites (so 10 times).

Some use-cases may be more intricate. For example, reading `individuals.dat` and assigning each individual its own time point requires to know how many individuals there are at each time point. For this we provide a wrapper around `read_data` that reads `individuals.dat`, `time.dat` and `popsize.dat` to perform this task:

```
read_individual_data(root)
#> # A tibble: 22,169 x 6
#>      time  deme patch     x     y     z
#>     <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#>  1     0     0     1     0     0     0
#>  2     0     0     1     0     0     0
#>  3     0     0     1     0     0     0
#>  4     0     0     1     0     0     0
#>  5     0     0     1     0     0     0
#>  6     0     0     1     0     0     0
#>  7     0     0     1     0     0     0
#>  8     0     0     1     0     0     0
#>  9     0     0     1     0     0     0
#> 10     0     0     1     0     0     0
#> # ... with 22,159 more rows
```

This data set contains all the relevant information for each individual saved in the simulation. Because this function has only one use-case (i.e. it is not designed to be flexible, unlike `read_data`) it gives the output tibble specific column names.

Same, we provide a function that directly reads and formats the data from the `traitmeans.dat` data set, which contains the mean trait value for each trait in each patch of each site, at each saved generation.

```
read_trait_mean_data(root)
#> # A tibble: 1,010 x 6
#>      time  deme patch     x     y     z
#>     <dbl> <int> <int> <dbl> <dbl> <dbl>
#>  1     0     1     0     0     0     0
#>  2     0     1     1     0     0     0
#>  3     0     2     0     0     0     0
#>  4     0     2     1     0     0     0
#>  5     0     3     0     0     0     0
#>  6     0     3     1     0     0     0
#>  7     0     4     0     0     0     0
#>  8     0     4     1     0     0     0
#>  9     0     5     0     0     0     0
#> 10     0     5     1     0     0     0
#> # ... with 1,000 more rows
```

And, similar to that, there is a wrapper function to directly read and format the data from the `patchsizes.dat` file:

```
read_patch_size_data(root)
#> # A tibble: 1,010 x 4
#>     time  deme patch     n
#>    <dbl> <int> <int> <int>
#>  1     0     1     0     0
#>  2     0     1     1    10
#>  3     0     2     0     0
#>  4     0     2     1     0
#>  5     0     3     0     0
#>  6     0     3     1     0
#>  7     0     4     0     0
#>  8     0     4     1     0
#>  9     0     5     0     0
#> 10     0     5     1     0
#> # ... with 1,000 more rows
```

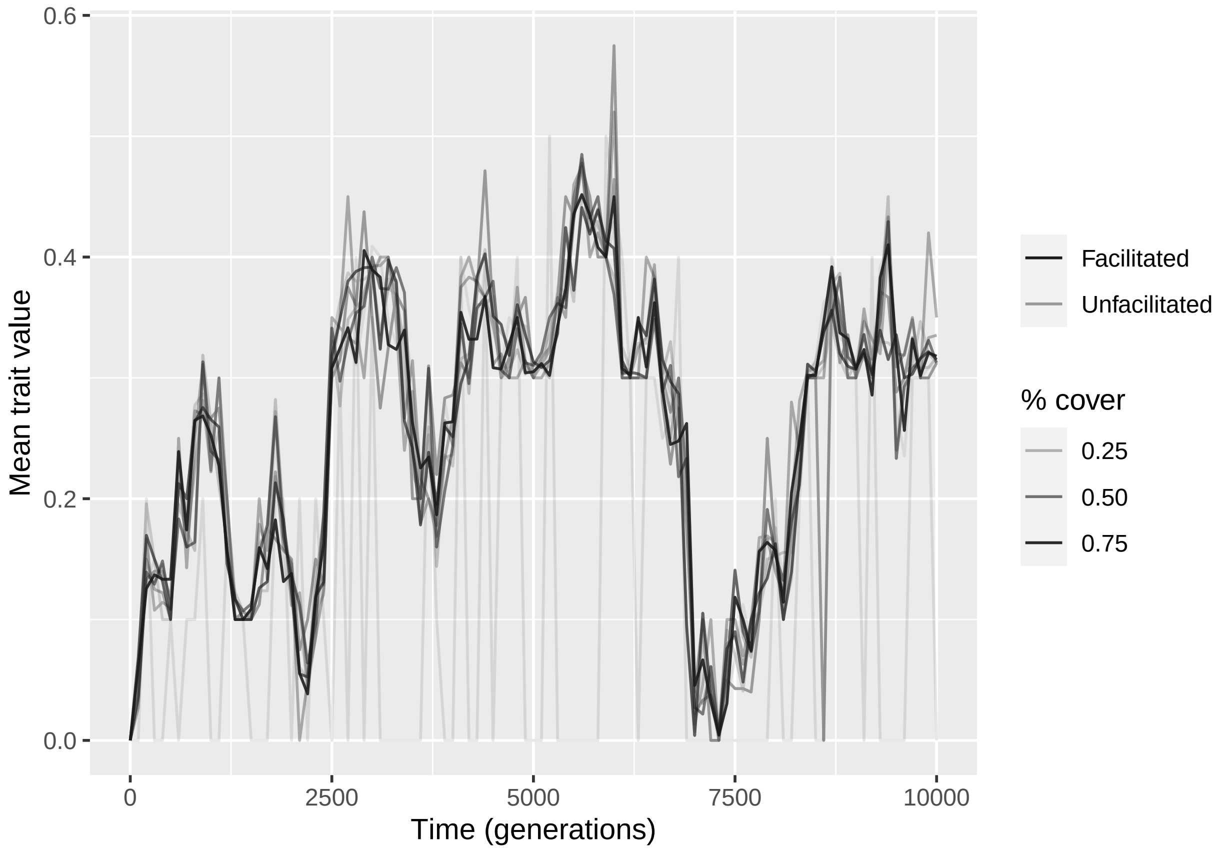
Besides the functions to read simulation data, one may want to read back the parameters that were used in a given simulation. To do this, use:

```
pars <- read_parameters(root)
pars[1:5] # just a few parameters
#> $type
#> [1] 1
#>
#> $popsize
#> [1] 10
#>
#> $pgood
#> [1] 5.0 0.8 0.6 0.5 0.3 0.1
#>
#> $maxgrowths
#> [1] 2 3
#>
#> $zwidths
#> [1] 1 2
```

Once the data are properly loaded and reshaped, they can be plotting using the usual tools available in R. Using `ggplot2`, for example:

```
library(tidyverse)
#> -- Attaching packages --------------------------------------- tidyverse 1.3.1 --
#> v ggplot2 3.3.6     v purrr   0.3.4
#> v tibble  3.1.7     v dplyr   1.0.9
#> v tidyr   1.2.0     v stringr 1.4.0
#> v readr   2.1.2     v forcats 0.5.1
#> -- Conflicts ------------------------------------------ tidyverse_conflicts() --
#> x dplyr::filter() masks stats::filter()
#> x dplyr::lag()    masks stats::lag()

# Read the proportion of good patches
pgood <- read_parameters(root)$pgood
pgood <- pgood[-1]
```

```
# Read trait means per patch per deme
data <- read_trait_mean_data(root)
data$patch <- as.logical(data$patch)

# Relabel
data <- data %>% mutate(patch_lab = if_else(patch, "Facilitated", "Unfacilitated"))

# Add information about the coverage in good patches
data <- data %>%
  group_by(deme) %>%
  nest() %>%
  ungroup() %>%
  mutate(pgood = pgood) %>%
  unnest(data) %>%
  mutate(patch_size = if_else(patch, pgood, 1 - pgood))

# Plot trait z through time
data %>%
  ggplot(aes(x = time, y = z, group = interaction(deme, patch), color = patch_lab, alpha = patch_size))
  geom_line() +
  xlab("Time (generations)") +
  ylab("Mean trait value") +
  labs(color = NULL, alpha = "% cover") +
  scale_color_manual(values = c("gray10", "gray60"))
```


Mean trait value
Time (generations)
Facilitated
Unfacilitated
% cover