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/4.1/brachypoder/extdata/sim-example"
```

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

```
list.files(root)
#>  [1] "architecture.txt" "brachypode"       "demesizes.dat"    "gmon.out"
#>  [5] "individuals.dat"  "parameters.txt"   "paramlog.txt"     "patchsizes.dat"
#>  [9] "popsize.dat"      "time.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 save. See the repository of ‘brachypode‘ for details.

Each data file contains data on 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 assemble into a dataset 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     100
#>  2     377
#>  3     599
#>  4     558
#>  5     585
#>  6     499
#>  7     593
#>  8     479
#>  9     486
#> 10     526
#> # ... 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     100
#>  2   100     377
#>  3   200     599
#>  4   300     558
#>  5   400     585
#>  6   500     499
#>  7   600     593
#>  8   700     479
#>  9   800     486
#> 10   900     526
#> # ... 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         100           0           0           0           0           0
#>  2   100          14          52          28          62          36          43
#>  3   200          20          77          54          62          72          57
#>  4   300          21          83          37          69          63          51
#>  5   400          17          66          49          61          48          62
#>  6   500          15          64          37          68          38          55
#>  7   600          16          75          44          64          57          54
#>  8   700          16          57          35          53          43          46
#>  9   800          21          79          28          57          52          51
#> 10   900           8          58          39          62          59          56
#> # ... 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`, 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. For example,

```
read_data(root, c("time", "patchsizes"), ncols = c(-10, 1))
#> # A tibble: 1,010 x 2
#>     time patchsizes
#>    <dbl>      <dbl>
#>  1     0        100
#>  2     0          0
#>  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.

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: 54,871 x 6
#>     time  deme patch     x     y     z
#>    <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#>  1     0     0     0     0     0     0
#>  2     0     0     0     0     0     0
#>  3     0     0     0     0     0     0
#>  4     0     0     0     0     0     0
#>  5     0     0     0     0     0     0
#>  6     0     0     0     0     0     0
#>  7     0     0     0     0     0     0
#>  8     0     0     0     0     0     0
#>  9     0     0     0     0     0     0
#> 10     0     0     0     0     0     0
#> # ... with 54,861 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.

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] 100
#>
#> $pgood
#> [1] 5.0 0.8 0.6 0.5 0.3 0.1
#>
#> $maxgrowths
#> [1] 1 3
#>
#> $zopts
#> [1] 3 1
```