Title:

slimr: An R package for integrating data and tailor-made population genomic simulations over space and time

Running Title: slimr for population genomics simulation

Authors:

Russell Dinnage[1, 2]
Stephen D. Sarre[2]
Richard P. Duncan[2]
Christopher R. Dickman[3]
Scott V. Edwards[4]
Aaron C. Greenville[3]
Glenda M. Wardle[3]
Bernd Gruber[2]

Corresponding Email: rdinnage@fiu.edu

Author Affiliations

1. Institute of Environment, Department of Biological Sciences, Florida International University, Miami, FL, USA
2. Centre for Conservation Ecology and Genomics, Institute for Applied Ecology, University of Canberra, Canberra, ACT, Australia
3. Desert Ecology Research Group, School of Life and Environmental Sciences, University of Sydney, NSW 2006, Australia
4. Department of Organismic and Evolutionary Biology, Harvard University, Cambridge, MA 02138, USA

Abstract

Software for realistically simulating complex population genomic processes is revolutionizing our understanding of evolutionary processes, and providing novel opportunities for integrating empirical data with simulations. However, the integration between simulation software and software designed for working with empirical data is currently not well developed. Here we present slimr, an R package designed to create a seamless link between standalone software SLiM >3.0, one of the most powerful population genomic simulation frameworks, and the R development environment, with its powerful data manipulation and analysis tools. We show how slimr facilitates smooth integration between genetic data, ecological data and simulation in a single environment. The package enables pipelines that begin with data reading, cleaning, and manipulation, proceed to constructing empirically-based parameters and initial conditions for simulations, then to running numerical simulations, and finally to retrieving simulation results in a format suitable for comparisons with empirical data – aided by advanced analysis and visualization tools provided by R. We demonstrate the use of slimr with an example from our own work on the landscape population genomics of desert mammals, highlighting the advantage of having a single integrated tool for both data analysis and simulation. slimr makes the powerful simulation ability of SLiM directly accessible to R users, allowing integrated simulation projects that incorporate empirical data without the need to switch between software environments. This should provide more opportunities for evolutionary biologists and ecologists to use realistic simulations to better understand the interplay between ecological and evolutionary processes.

Keywords: population genomics; simulation; landscape genomics; evolution; ecology; evolutionary ecology; application; software

Introduction

Mathematical modelling and simulation are critical cornerstones of population genetic practice. At a fundamental level, empirical datasets demand analytical tool-kits that can accomodate their high complexity, and recent developments in sophisticated simulation software have the potential to provide mechanistic insight into increasingly complex evolutionary scenarios (Carvajal-Rodríguez, 2010; Haller & Messer, 2019; Hoban, 2014; Kelleher, Etheridge, & McVean, 2016; Messer, 2013; Strand, 2002; Yuan, Miller, Zhang, Herrington, & Wang, 2012). However, utilising flexible simulations requires exploration of large parameter space, which often generates large amounts of data that need sophisticated computational tools to unpack, interrogate and synthesize. Likewise, using simulations to model empirical data is an emerging field because it allows researchers to deal with complex situations where it is difficult to obtain a closed likelihood (Beaumont, Zhang, & Balding, 2002; Brehmer, Louppe, Pavez, & Cranmer, 2020; Cranmer, Brehmer, & Louppe, 2020; Marjoram, Molitor, Plagnol, & Tavare, 2003; Sisson, 2018; Torada et al., 2019; Wang et al., 2020). To facilitate more rapid and seamless interrogation and synthesis between empirical data and population genetics simulation, we present `slimr` (https://rdinnager.github.io/slimr/). `slimr` is an R package designed to link the very large and widely used ecosystem of analysis and visualization tools in the R statistical language to the SLiM scripting language (Haller & Messer, 2019), a popular, powerful and flexible population genetics simulation tool. The package creates a smooth fusion between the computational power and flexible model specification of SLiM with the advanced statistical analysis, visualisation, and metaprogramming tools of R.

Package Description

`slimr` is an R package that interfaces with SLiM >3.0 software for forward population genetics simulations (Haller & Messer, 2019 for full details on SLiM, as well as the website at

https://messerlab.org/slim/ ; see Messer, 2013). slimr has most recently been updated to work with SLiM version 4.0 and greater but should also be compatible with any version greater than 3.0 (previous versions may work but have not been tested)

`slimr` implements a Domain Specific Language (DSL) that mimics the syntax of SLiM, allowing you to write and run SLiM scripts and capture resulting simulation output, all within the R environment. Much of the syntax is identical to SLiM, but `slimr` offers additional R functions that allow users to manipulate SLiM scripts ("`slimr` verbs") by inserting them directly into any SLiM code block. This enables R users to create SLiM scripts that explore large numbers of different parameters and also automatically produce output from SLiM for powerful downstream analysis within R.

The features of `slimr` fall into three categories: 1) SLiM script integrated development, 2) data input/output, and 3) SLiM script metaprogramming. The first set of features is designed to make it easy to develop SLiM scripts in an R development environment such as RStudio, and mostly recapitulates features that SLiM users already have access to in the form of SLiMgui and QtSLiM (https://messerlab.org/slim/). The second and third features are implemented using what we call "`slimr` verbs", allowing SLiM and R features to be combined in advanced ways. The integration between R and SLiM provided by `slimr` compensates knowledgeable users of R for a lack of knowledge of SLiM, helping to lower the barrier to learning and using SLiM.

Each of the 3 categories has subcategories of features as follows:

1) Integrated Development
    a) Autocomplete and Documentation (within R) for SLiM code (Fig. 1)
    b) Code highlighting and pretty printing of SLiM code
    c) Run code in SLiM from R
2) Data Input/Output (Fig. 2)

a) Automatic output generation and extraction from SLiM to R (`r_output()`)

b) Insert arbitrary R objects into SLiM scripts through inlining (`r_inline()`)

3) Metaprogramming (Fig. 2)

a) Code templating for SLiM scripts (`r_template()`)

b) Flexible general metaprogramming tools (support for `rlang`'s `!!` and `!!!` forcing operators)

slimr Verbs

Note that `r_` prefixed functions are what we call "`slimr` verbs", used for data input and output, and for metaprogramming. `slimr` verbs are special functions that can be inserted directly into `slimr` coding blocks (e.g. code called within the slim_block() function). These verbs are pure R functions that modify how the SLiM script will be generated and run in SLiM. They are not passed directly to SLiM, but make it easy for R to interact with SLiM. In this way, `slimr` code appears to be a hybrid between SLiM and R code. `slimr` verbs allow all setup and logic required to use SLiM with R to occur inside the `slim_block()` coding blocks comprising the `slimr_script` object, thus requiring fewer arguments to be set in preparation for downstream analysis (for example `slim_run` does not require many complex arguments because most of what it needs to know is embedded in the `slimr_script` object). In our experience, this leads to a very smooth experience using `slimr` by reducing the frequency of switches between different mental modes. By convention, all `slimr` verbs have the prefix `r_`, to denote they are R functions that will be executed from within R, typically to insert something into the eventual SLiM script. They are meant to be used only inside `slim_block` calls, and will do nothing if called outside this context. All other `slimr` functions are prefixed with `slim_`, which generally means they are to be used on `slimr_script` objects (or to create them), and not inside a `slim_block` call. Examples of all `slimr` verbs can be seen in an example script in Figure 1.

In the next section we describe `slimr` features in greater detail, showing examples through screenshots and code snippets.

Integrated Development

`slimr` allows the you to write SLiM code from within an R integrated development environment (IDE). `slimr` is designed to work well with RStudio, but can be used in any R IDE. The syntax used to write SLiM code in slimr is very similar to the native SLiM syntax, with a few modifications to make it work with the R interpreter (Table 1). As an example, here is a minimal SLiM program, and its counterpart written in `slimr`.

SLiM code:

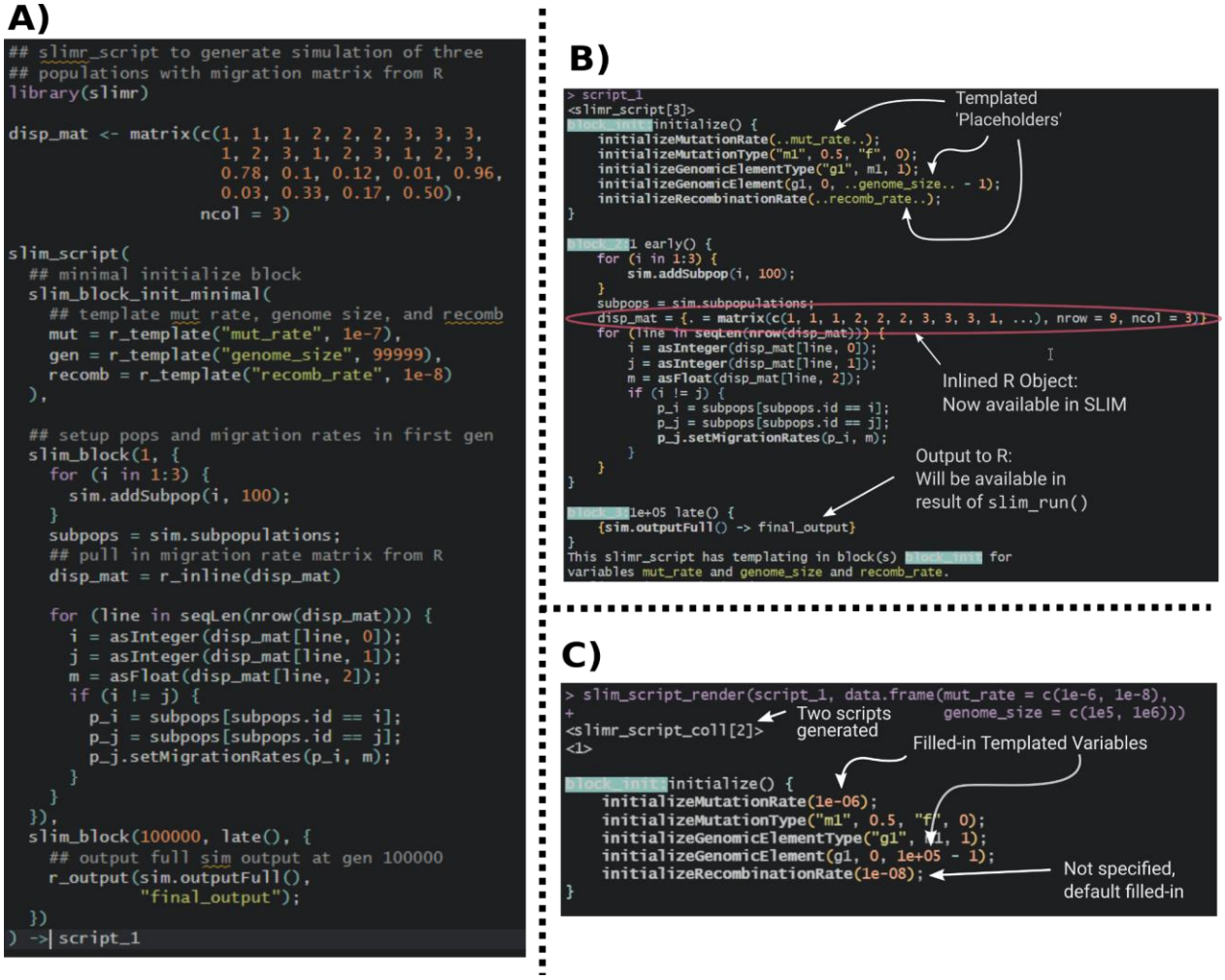
```
initialize()
{
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0);
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);
}
1
{
  sim.addSubpop("p1", 500);
}
10000
{
  sim.simulationFinished();
}
```

slimr code:

```
slim_script(
  slim_block(initialize(),
  {
     initializeMutationRate(1e-7);
     initializeMutationType("m1", 0.5, "f", 0.0);
     initializeGenomicElementType("g1", m1, 1.0);
     initializeGenomicElement(g1, 0, 99999);
```

```
      initializeRecombinationRate(1e-8);
   }),
   slim_block(1,
   {
      sim.addSubpop("p1", 500);
   }),
   slim_block(10000,
   {
      r_output_full();
      sim.simulationFinished();
   })
) -> script_1
```

The above code assigns the script to an R object `script_1`, which can then be further manipulated, printed prettily, and sent to SLiM to run. See Fig. 1 to see what the above script looks like in the RStudio IDE, and examples of things you can do with it. A script is specified using the `slim_script` function, within which you create `slimr` coding blocks, using the `slim_block` function. The user can create as many `slimr` code blocks as desired within a `slim_script`. We've added a `slimr` "verb" (`r_output_full`), which tells SLiM to output the full state of the simulation and return it to R during the execution of the block. We will discuss `slimr` verbs in more detail in the next section.



181

182 ***Figure 1.*** *Example of a single script using the main* `slimr` *verbs (*`r_template`*,* `r_output`*, and*

183 `r_inline`*). A) Code to specify the* `slimr_script`*. B) Pretty printing of the script, showing special*

184 `slimr` *syntax. C) Example of running* `slim_script_render` *on the* `slimr_script` *object,*

185 *demonstrating how placeholder variables specified in r_template are replaced with provided*

186 *values. All code from the above example can be accessed as a package vignette*

187 *(https://rdinnager.github.io/slimr/articles/simple_example_using_migration_and_fst.html).*

188 `slimr` makes it easy to write SLiM code in R after you learn a few differences between SLiM and

189 slimr. This means you can learn how to write complex SLiM simulations by reading SLiM

190 documentation and the examples found within it (https://messerlab.org/slim/). To make this

191 process easier for `slimr` users, the entire reference documentation for functions in SLiM and

Eidos scripting language (on which SLiM is based) is included in `slimr` (with the original author's permission). Hence, not only can you look up relevant SLiM functions in their R session (by typing ? followed the name of the function, e.g. `?slimr::addRecombinant`), but the R IDE can also perform autocompletion (Figure 2).

`slimr_script` objects (and `slimr_script_coll`, which is a list of multiple `slimr_script` objects, explained more later) can be run in SliM, and their results collected and returned, using the slim_run function.

Table 1. Demonstrating the main differences in syntax between SLiM and slimr. Most of these differences are required to make the code work with the R interpreter, such that the code can be written, executed and autocompleted in R without error.

SLiM Code	**slimr Equivalent**	**Notes**

`something = "hello world";`
`print(something);`
	`something <- "hello world"`
	`print(something)`
	-OR-
	`something = "hello world";`
	`print(something);`
		In SLiM, semicolons at the end of lines are mandatory, and assignment is always with `=`. In `slimr`, R-style assignment operator `<-` is allowed and semicolons are optional. Code will be appropriately converted to work in SLiM.

`return T;`	`return(T);`	In SLiM, `return` is a keyword, but in R, `return` is a function. Use `return()` in `slimr`, which will be automatically converted for SLiM. Note also that for `TRUE` and `FALSE`, SLiM uses `T` and `F`, so use `T` and `F` in `slimr` as well.
`cat(fixed ? "FIXED\n" else "LOST\n");`	`cat(fixed %?% "FIXED\n" %else% "LOST\n");`	SLiM can make use of Eidos trinary operator `?`, which is a compact form of an `if … else` statement, or a non-vectorized form of `ifelse()`. `slimr` supports this by using the

		operators `%?%` and `%else%`.

`if (fixed)`
`  cat("FIXED\n");`
`else`
`  cat("LOST\n");`
	`if (fixed)`
	`  cat("FIXED\n"); else`
	`  cat("LOST\n");`
		`if ... else` statements are formatted slightly differently between SLiM and R. In SLiM, the `else` statement must follow a newline after the final line of the `if` statement. In R, the `else` statement must be on the same line as the final line of the `if` statement. In `slimr` use the R form, it will be converted to work with SLiM.

`sim.addSubpop("p1", 500);`	`sim.addSubpop("p1", 500);`
	-OR-
	`sim$addSubpop("p1", 500);`
	-OR-
	`sim%.%Species$addSubpop("p1", 500);`
		The other varieties may look a little weird but they allow autocomplete to work in R. If you don't need autocomplete it works best to just write as you would in SLiM.

`[id] [t1 [: t2]] first() { ... }`	`slim_block([id,] [t1, [t2,]] first(), { ... })`	This is how an Eidos block is specified in SLiM and `slimr`. Things inside square brackets are optional. Instead of `first()`, any event or callback function can be used. `id` is a name for the code block, `t1` is the first time to run the block, `t2` is the last time. `...` is arbitrary Eidos or `slimr` code.
`1: late() { ... }`	`slim_block(1, .., late(), { ... })`	In SLiM, using `t: ` , means to run the code from time `t` until the end of the simulation. In `slimr`, this can be accomplished by using `` `..` `` for the end time.

`do`
`{...}`
`while (runif(1) < 0.8);`
	`proceed <- 0;`
	`while(proceed < 0.8) {`
	`  ...`
	`  proceed <- runif(1);`
	`}`
		R does not have a `do ... while` loop construction, it only has `while` loops. The main difference is that `do ... while` tests the condition at the end of the `{...}` code, whereas `while()` tests the condition before the `{...}` code. Similar functionality can be achieved with a slightly more verbose `while` loop in R, which will also work in SLiM.

Autocomplete

Autocomplete is supported for SLiM code, though with some unavoidable limitations. More specifically, autocomplete is limited to the suggestions of the names of functions and suggestions for what arguments are available for those functions. Autocomplete of regular eidos code is straightforward, because slimr simply maintains a function stub for eidos functions, which contains their arguments and help information for them. On the other hand functions that are methods within SLiM classes (the majority of functions since SLiM is an object-oriented language), create more of a complication. This is because the operator to access elements inside a object in SLiM is '.' (similar to Python), but in R '.' is not an operator, R assumes the '.' is part of the name of an object. This means if you type 'sim.addSubpop`, R will not recognize 'addSubpop' as a function to look up for autocompletion.

In `slimr`, there are two ways you can get around this limitation for accessing autocomplete for methods from within R. The simplest way that results in the most readable code is to use the `slim_load_globals()` function. When run this function will load object stubs for commonly used SLiM class instances into the R global environment. This includes the `` `sim` `` object, which is the main SLiMSim class instance used to track the simulation from within SLiM (or an instance of class Species in SLiM 4.0 or greater, where it represents the main species simulation in a single

species simulation). It can also load as many numbered global variables used in SLiM as desired, named as in SLiM like `p1, p2, … pn` for the first n Subpopulations, `g1, g2, … gn` for GenomicElementTypes, etc (see documentation of `slim_load_globals()` for details). Functions (or properties) can then be accessed within these instances using the standard R `` `$` `` operator for accessing elements of a list. For convenience, `slimr` converts any `` `$` `` in SLiM code into `` `.` ``, allowing you to leave their code as is after autocompletion has been used. An example of this technique is shown in Figure 2. The second method is less readable but avoids having to load otherwise unnecessary objects into your global environment. To use it, you type the object name followed by the R operator `%.%`, which is included in `slimr`, they then type the class name of the object (such as Genome, or SLiMSiM) and then using the '`$`' operator, methods and properties of that class can be accessed and autocompleted. An example would be `sim%.%Species$addSubpop()`. This is a little verbose so `slimr` also includes abbreviated versions of all SLiM classes. For Species the abbreviation is `Sp`, so you could type `sim%.%Sp$addSubpop()`. Just as for the other solution, `slimr` knows how to properly replace the above code with the correct SLiM code, which would be `sim.addSubpop()`. See Figure 2 for a screenshot of both the methods in action in the RStudio IDE.

Figure 2. *Screenshots of working with* `slimr` *autocomplete in RStudio. a) An example of autocomplete for SLiM code using the slim_load_globals() function which loads a set of objects that store what SLiM classes contain, in this case the sim object, which is a SLiM 4.0 Species class. This let's the use press tab to bring up the arguments of addSubpop() function, which is a method of Species. The lower panel shows that slimr automatically replaces the expression with the correct SLiM code. b) An example of using the alternative method for autocomplete by typing the name of an object (sim), followed by %.% and then the class name or an abbreviation of the*

class name (Sp), and then using '$' to access methods or properties of the class. Again, the lower panel shows how slimr correctly replaces the construction with valid SLiM code. By prefixing any of the methods or properties inside the slimr class objects with ?, you can also bring up the full documentation of the method or property from the SLiM manual.

Data Input/Output

The input/output and metaprogramming features of `slimr` are achieved using The following are the main `slimr` verbs supported:

`r_inline`

`r_inline` allows you to embed (or “inline”) an R object directly into a SLiM script so that it can be accessed from within a SLiM simulation. This is a powerful way to use within a simulation empirical data that has been generated, loaded, and / or cleaned in R. `r_inline` automatically detects the class of R object and attempts to coerce it into a format compatible with SLiM. Currently supported types are all atomic vectors, matrices, arrays, and Raster* objects from the raster package, which will allow you to insert maps for use in spatial simulations. Internally, `r_inline()` simply embeds a text version of the object into the script, so the type will be automatically determined by SLiM and its heuristics. We have found that this produces satisfactory results in general, though we plan to implement more control over this aspect of SLiM programming in future versions of `slimr`. An example of using `r_inline()` is shown in Figure 1.

`r_output`

`r_output` makes it simple to output data from the simulation by wrapping a SLiM expression. Where it is called in the `slimr_script,` it will produce SLiM code to take the output of the expression and send it to R. After calling `slim_run` on a script object, the output will be available as a data.frame within the returned object. Output can even be accessed live during the

simulation run via the use of callback functions (an example of how this is done can be found in the package vignetter called Custom_vis.Rmd (https://rdinnager.github.io/slimr/articles/Custom_vis.html). A do_every argument tells `r_output` not to output every time it is called, but rather only after every `do_every` generations. An example showing `r_output()` in action can be seen in Figure 1.

`slimr` includes several functions to create different commonly desired outputs and visualizations, which use the `r_output_` prefix (e.g. `r_output_nucleotides()`, which outputs DNA sequences data for nucleotide-based simulations). These outputs can be extracted from the `slimr_results` object created by `slim_run()` using the `slim_extract_` prefixed functions (e.g. `slim_extract_nucleotides()`, `slim_extract_genlight()`, etc.)

Metaprogramming

Metaprogramming is programming that generates or manipulates code itself. `slimr` has facilities for manipulating SLiM programming code and generating scripts. The main `slimr` verb for doing this is `r_template`. `slimr` also supports the metaprogramming operators for forcing (`!!`) and splicing (`!!!`), as used in the `{rlang}` R package. Here we briefly describe `r_template`, designed to help you easily generate many versions of a slimr_script with different parameters.

r_template

`r_template` allows you to insert “templated” variables into a `r_script;` the call to `r_template` will be replaced in the SLiM script with a placeholder `var_name` chosen by the user. This placeholder can be replaced with values of your choice by calling `slim_script_render(slm_scrpt, template = tmplt)`, and providing a template – a list or data.frame containing values with names matching `var_name`. This action can be performed on multiple `r_template` variables simultaneously, as well as producing multiple replicate scripts with different combinations of replacements. This feature can create a swathe of parameter

values to be run (automatically) in parallel to explore parameter space, conduct sensitivity analyses, or fit data to simulation output using methods requiring many simulation runs, such as Approximate Bayesian Computation (ABC). You can provide a default value for each templated variable, which will be used if you do not specify a replacement for that variable. An example of using `r_template()` can be seen in Figure 1.

These features together make `slimr` far more than a simple wrapper for SLiM – its goal is to enhance and complement SLiM by creating a hybrid, domain specific language for R. We plan to continue to increase integration of our package with SLiM, and to continuously update it as new SLiM versions are released in the future.

`slim_run`

Once a `slimr_script` or `slimr_script_coll` object has been created, with all SLiM simulation logic and `slimr` verbs for interacting with R, it can be sent to the SLiM software to be run using the `slim_run` function. To access this functionality, users must install SLiM on their computers, and link it to `slimr` (instructions for doing so are on the package website).

Calling `slim_run` will run the simulation. While the simulation is running, `slim_run` produces progress updates if requested, as well as any output generated by calls to `r_output` with custom callbacks. If called on a `slimr_script_coll` containing multiple `slimr_script` objects, each `slimr_script` object will be run, optionally in parallel, and the result returned in a list.

Once finished, `slim_run` will return a `slimr_results` object, which contains information about the simulation run, such as whether it succeeded or failed, any error messages produced, all output generated from r_output calls, and any file names where additional data from the run are stored. This can then be used for any downstream analysis you desire.

`slimr` and Open Science

It is increasingly being seen as vital for biologists to share code used to generate their results in the spirit of open science. A researcher may spend months perfecting a SLiM script that simulates a particular scenario of interest, but this scenario and those similar to it are likely of interest to other researchers as well. `slimr` allows the sharing of simulations in a very open and easy to use way, through the R software ecosystem. It provides tools that can allow researchers, with very little additional code, to make their simulations accept user-defined input, and output to common formats used by R users. Simulations can easily be wrapped into an R package, which can then be installed by any R user with a command. Because `slimr` provides general interfacing functionality from SLiM to R, it allows open development of simulations by developers with much less experience with SLiM coding, and requiring far less time.

Examples

Here we demonstrate the use of `slimr` on a short and simple example, and one more extensive example.

Simulating Nucleotide Evolution

The following script simulates a population of 100 individuals that randomly splits into two equally sized subpopulations with a probability split_prob in each generation, after which the subpopulations are reproductively isolated from each other. It simulates genomic evolution with an explicit nucleotide sequence evolution model (Jukes-Cantor model). By default SLiM only simulates and keeps track of 'mutations' in a more abstract sense (these could be thought of as generating new alleles at a gene, or SNPs, or however the researcher wants to interpret them). This example demonstrates the easiest way to get data from R into a slimr simulation, by using the forcing operator `!!`. The forcing operator forces R to replace the expression following it with

the value to which the expression evaluates, counter to R's default form of evaluation, which evaluates an expression the first time it is used in the Abstract Syntax Tree of the code (hence the term 'forcing'), where, in this case, it would be treated as SLiM code. In simple terms, the !! is used to flag that the expression following is not a SLiM expression, but a reference to a value that is generated by the corresponding R expression, often referring to an object in the your R environment. In the script below, we have highlighted where this is occurring in bold. The below script is also available as a vignette in the `slimr` package (which can be viewed here: https://rdinnager.github.io/slimr/articles/simple_nucleotide_example.html).

```
## set some parameters
seed <- 1205
split_prob <- 0.001
max_subpops <- 10

## specify simulation
split_isolate_sim <- slim_script(

  slim_block(initialize(), {

    setSeed(!!seed);

    ## tell SLiM to simulate nucleotides
    initializeSLiMOptions(nucleotideBased=T);
    initializeAncestralNucleotides(randomNucleotides(1000));
    initializeMutationTypeNuc("m1", 0.5, "f", 0.0);

    initializeGenomicElementType("g1", m1, 1.0, mmJukesCantor(1e-5));
    initializeGenomicElement(g1, 0, 1000 - 1);
    initializeRecombinationRate(1e-8);

  }),

  slim_block(1, {

    defineGlobal("curr_subpop", 1);
    sim.addSubpop(curr_subpop, 100)

```

```
  }),

  slim_block(1, 10000, late(), {

    if(rbinom(1, 1, !!split_prob) == 1) {
      ## split a subpop
      subpop_choose = sample(sim.subpopulations, 1)
      curr_subpop = curr_subpop + 1
      sim.addSubpopSplit(subpopID = curr_subpop,
                         size = 100,
                         sourceSubpop = subpop_choose)
      ## if too many subpops, remove one randomly
      if(size(sim.subpopulations) > !!max_subpops) {
        subpop_del = sample(sim.subpopulations, 1)
        subpop_del.setSubpopulationSize(0)
      }
    }

    ## output nucleotide data
    r_output_nucleotides(subpops = TRUE, do_every = 100)

  }),

  slim_block(10000, late(), {
    sim.simulationFinished()
  })

)

results <- slim_run(split_isolate_sim)
```

Next, we extract the data, and print it to see what he results object looks like. The we plot one of the sequence alignments as an image plot (Figure 3).

```
res_data <- slim_results_to_data(results)

res_data
## # A tibble: 100 x 6
##    type             expression               generation name  data
##    <chr>            <chr>                         <int> <chr> <list>
```

```
##  1 slim_nucleotides slimr_output_nucleotide~       100 seqs  <DNAStrnS>
##  2 slim_nucleotides slimr_output_nucleotide~       200 seqs  <DNAStrnS>
##  3 slim_nucleotides slimr_output_nucleotide~       300 seqs  <DNAStrnS>
##  4 slim_nucleotides slimr_output_nucleotide~       400 seqs  <DNAStrnS>
##  5 slim_nucleotides slimr_output_nucleotide~       500 seqs  <DNAStrnS>
##  6 slim_nucleotides slimr_output_nucleotide~       600 seqs  <DNAStrnS>
##  7 slim_nucleotides slimr_output_nucleotide~       700 seqs  <DNAStrnS>
##  8 slim_nucleotides slimr_output_nucleotide~       800 seqs  <DNAStrnS>
##  9 slim_nucleotides slimr_output_nucleotide~       900 seqs  <DNAStrnS>
## 10 slim_nucleotides slimr_output_nucleotide~      1000 seqs  <DNAStrnS>
## ... with 90 more rows

image(ape::as.DNAbin(res_data$data[[100]]))
```

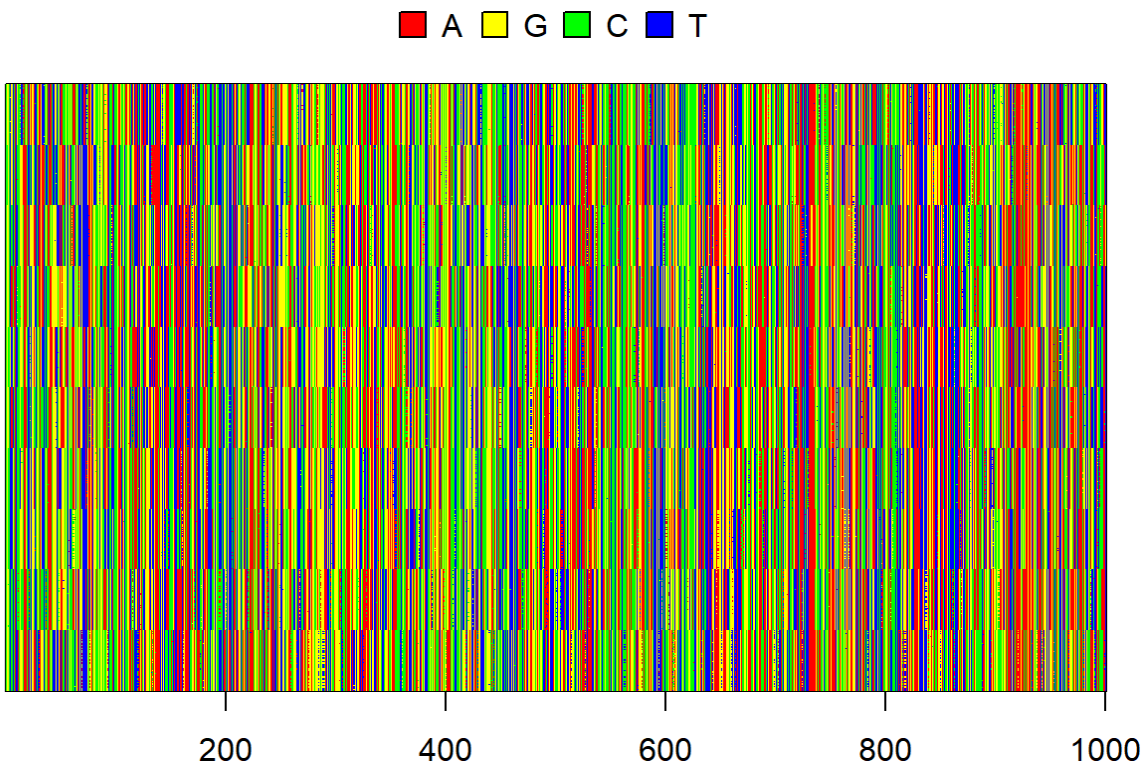


Figure 3. *Simulated sequences for each individual.* Subpopulation clustering is obvious as a pattern or strong horizontal bands in ten distinctive patterns.

And then we use some other R packages to quickly build a tree based on the simulated nucleotides, to see if it looks like what we would expect from a sequentially splitting population (Figure 4).

```
## convert to ape::DNAbin
al <- ape::as.DNAbin(res_data$data[[100]])
dists <- ape::dist.dna(al)
upgma_tree <- ape::as.phylo(hclust(dists, method = "average"))
pal <- paletteer::paletteer_d("RColorBrewer::Paired", 10)
plot(upgma_tree, show.tip.label = FALSE)
ape::tiplabels(pch = 19, col = pal[as.numeric(as.factor(res_data$subpops[[100]]))])
```

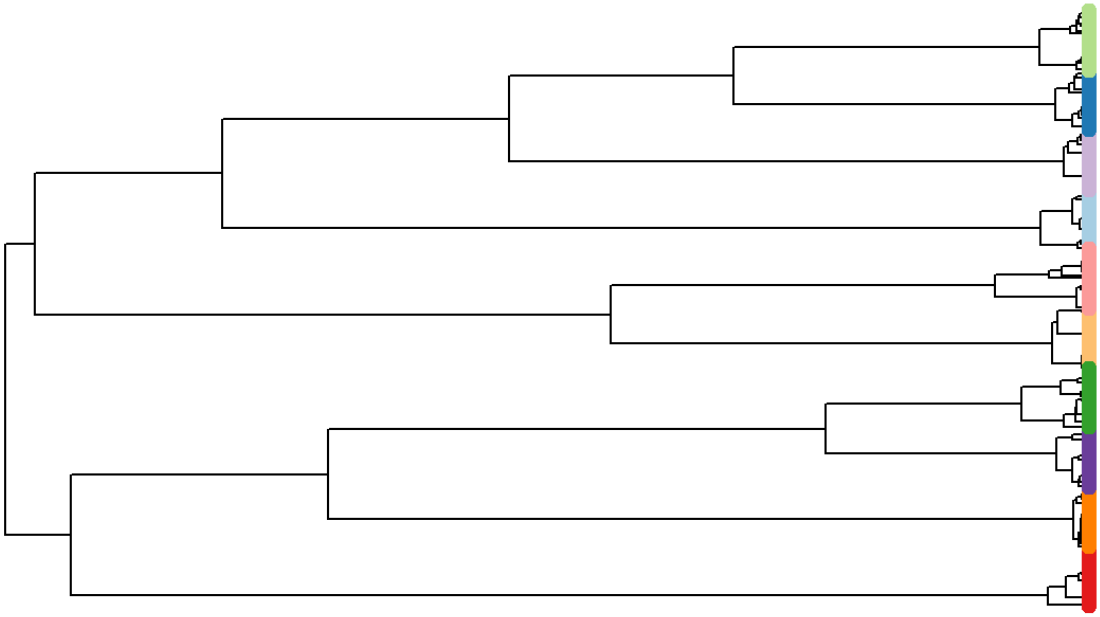


Figure 4. *UPGMA tree of simulated subpopulations, tip points coloured by subpopulation.*

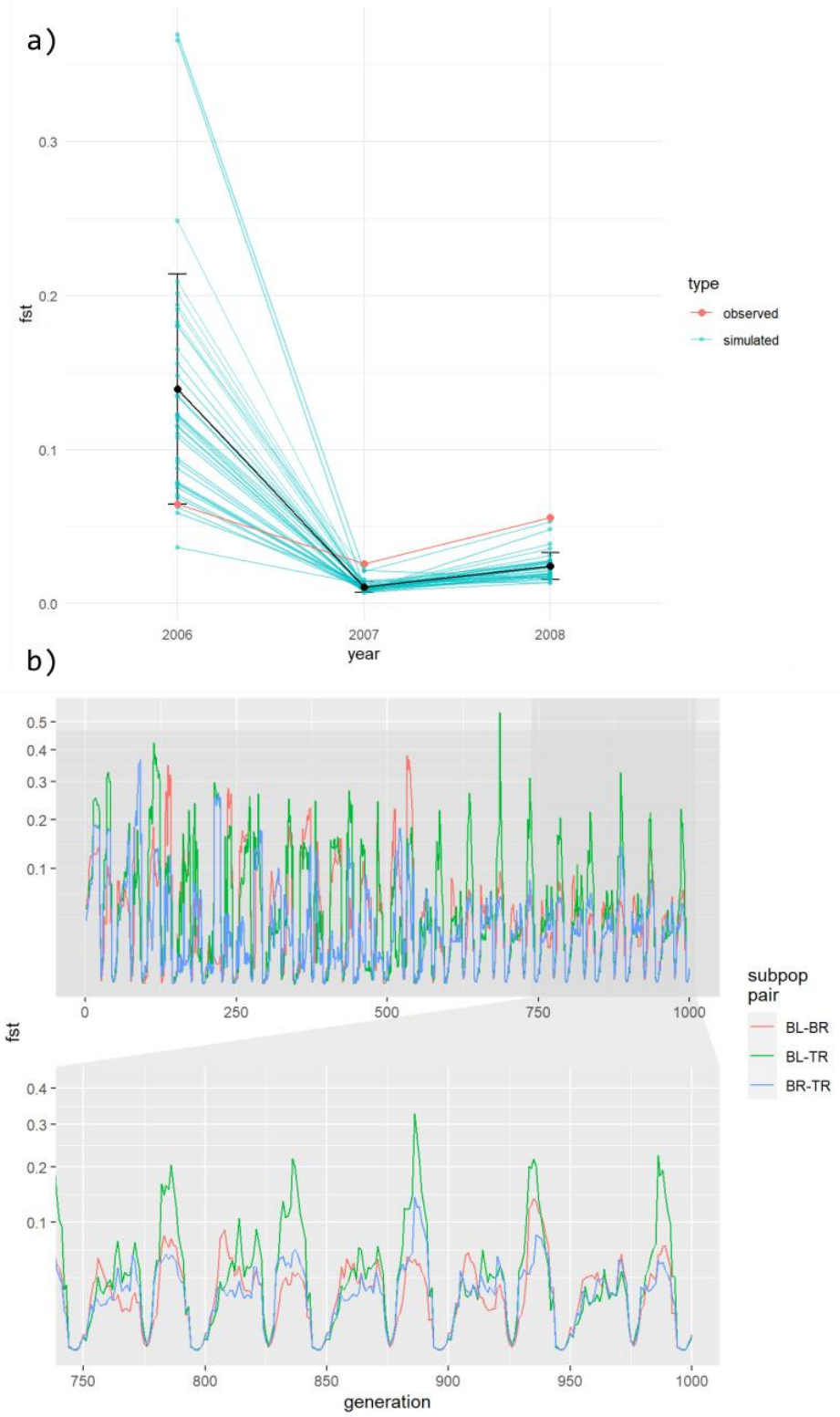
Scientific Hypothesis Exploration Example: Investigating population genomics of small mammals in a periodic environment

In this section we provide a brief description of a full example analysis using simulation l That is fully described in the accompanying Supplementary Material and a substantial R vignette available here: https://rdinnager.github.io/slimr/articles/Main_manuscript_example_v2.html.

The context for this example is a long-term ecological study in the Simpson Desert in central Australia. Several authors of this paper have studied the population dynamics of small mammals and reptiles in this desert for more than 30 years (C. Dickman, Wardle, Foulkes, & de Preu, 2014; Greenville, Dickman, & Wardle, 2017; Greenville, Wardle, Nguyen, & Dickman, 2016). Recently, we have begun sequencing tissue samples taken from animals captured during the past 15 years, and obtained single nucleotide polymorphism (SNP) data using DArT (Diversity Arrays Technology Pty Ltd) technology. Here, we use SNP data from 167 individuals of a common native rodent species, the sandy inland mouse *Pseudomys hermannsburgensis*, sampled at 7 sites over three years (2006-2008), and subsequently aggregated to 3 subpopulations for analysis. The three sample years span periods before and after a major rainfall event at the end of 2006; big rains occur infrequently in the study region (every 8-12 years) (Greenville, Wardle, & Dickman, 2012) but drive major population eruptions.

We used the SNP data to calculate pairwise F_{st} values among the three subpopulations in each year, revealing that pairwise F_{st} values dropped rapidly to nearly zero immediately after the rainfall event from a high recorded just prior to the event when the populations were more genetically differentiated. We interpreted this result to mean that the rainfall event, which caused the sandy inland mouse population to rapidly increase, also allowed animals to move out of spatially scattered refuge patches to which they had been confined during the preceding dry period (C. R. Dickman, Greenville, Tamayo, & Wardle, 2011). This movement allowed the subpopulations to mix, leading to a decrease in population genetic structure as measured by F_{st}.

In the example, we use simulations to evaluate our interpretation regarding the processes driving changes in F_{st} values. We found that our initial hypothesis, that the rainfall event led to the mixing of previously unconnected populations in refuge patches, provided a good match to the data when we simulated the population and genomic processes (Figure 5). However, we also identified several other processes that could generate similar outcomes, which raises the question as to what data or analyses would be required to distinguish among these competing processes.



471

472 ***Figure 5. A)*** *Mean F_{st} values from 36 replicate simulations simulated under our hypothesized*

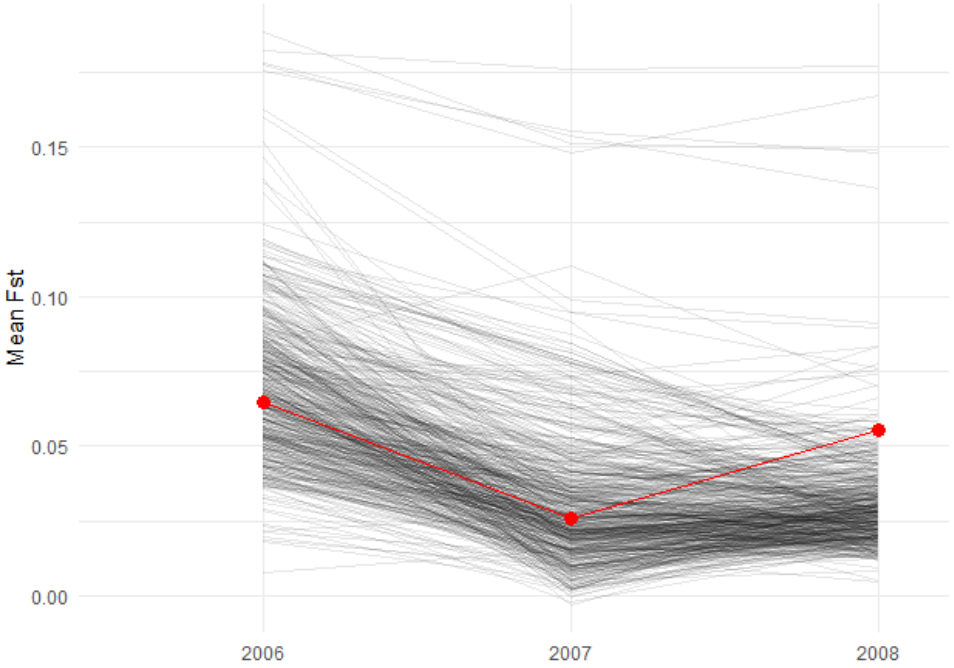
473 *mechanism to explain Fst fluctuations in small mammal population in the Simpson Desert, using*

hand chosen parameter values. Blue values and lines represent simulated values, red values and line represents the observed F_{st} values. Details of simulation including code is in the Supplementary material. ***B)*** *Same simulation run over many generations, showing the three subpopulation pairs separately. The subpop pair refers to pairwise combinations of three subpopulations named BL, BR, and TR (further explanation can be found in the vignette of this example in the Supporting Information, or as included in the slimr package)*

To formalize our ideas a little more we ran an Approximate Bayesian Computation (ABC) analysis to derive an approximate posterior distribution of model parameters that produced a good fit to our short F_{st} time series (see Supplementary Materials: ABC Analysis for code used). We were able to easily move from simulation exploration to a more formal fitting exercise because the simulation was already in R (thanks to `slimr`), and so only a small amount of code was required to convert the input and output of our simulation to the format required by the `easyABC` package, which we used for this analysis.

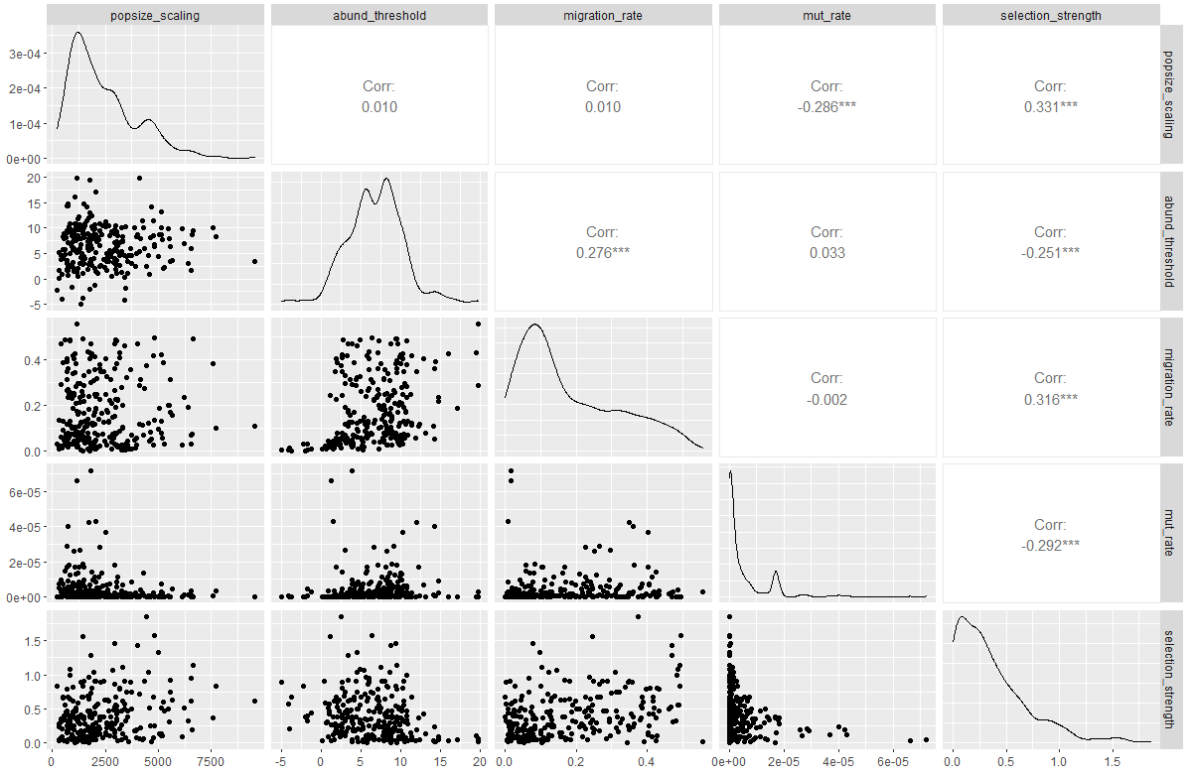
Simulations using parameter values drawn from the approximate posterior of our ABC analysis are shown in Figure 6a, and the pairwise joint posterior distribution of those parameter values are shown in Figure 6b. Overall the ABC analysis confirms that these three data points do not constrain much the parameter space of plausible models, though in some cases joint distribution appears to be somewhat constrained for certain combinations of parameters.

493 **a)**



494

495 **b)**



496

Figure 6. *a) Fst values calculated from simulations based on 500 parameter value sets drawn from the approximate posterior distribution of our simulation, based on an ABC analysis. Partially transparent black lines represent the simulations. Red points and line represent the observed Fst values from the study described in this section. b) Samples from the approximate posterior distribution for our simulation model fit with Approximate Bayesian Computation. Lower left panels show the 5 parameters of our simulation model estimated, plotted against each other in a pairwise fashion, giving an indication of their joint posterior distribution. Some parameters are highly correlated in the posterior. Upper right panels show the pearson correlation coefficient, and the panels in the diagonal show the marginal distribution of each parameter estimated using kernel density estimation on the samples.*

The marginal posterior distribution based on samples from the ABC analysis confirms that our data do not constrain individual parameters much, with a fairly wide distribution for most parameters providing a good fit to our data (Figure 6b, diagonal panels). The only exception was perhaps mutation rate, for which the lower values that we simulated tended to provide a better fit. The parameter of most interest to us was the abundance threshold (abund_threshold in Figure 6b), which specified the population size above which a subpopulation would 'turn on' migration, that is, start exporting individuals to the other subpopulations (in the real system this population size change is driven by rainfall). In this simulation an abundance threshold of zero or less would be migration always happening, and one of 20 or over would be migration almost never happening. Some simulations produced well fitting Fst values for nearly all relevant values of the abundance threshold, with some falloff at either end. However, when we start looking at combinations of multiple parameters we see that the value of the abundance threshold parameter does constrain what values of other parameters will make for a good fit to the data. For example, if the abundance threshold is low, and thus migration is always on, only simulations with very low migration rates and very low mutation rates can provide a good fit to the data (figure 6b, panels in rows 3 and 4 in column 2). All in all this suggests that there are two approaches to improving our

ability to distinguish how different processes lead to the patterns we see (besides just collecting more data): 1) try adding new summary statistics besides just pairwise Fst, which may capture some other aspect of the data, and 2) use some independent sources of data or information to estimate and constrain the parameter space of our simulations closer to that of the real system. In particular, approach 1 could be tested without having to collect more data by doing more simulations: we could simulate our model, then simulate data collection and calculate our new summary statistic on the simulated data. We can then see if we can recover the parameters of our simulation better than we could before incorporating our new statistic.

The results from these preliminary simulations will thus be invaluable in guiding which individuals and time periods we should focus our sequencing on, and what summary statistics to use, to maximize the chances of distinguishing among competing hypotheses that might explain the combined population and genetic patterns in the data. Ultimately, we aim to use this approach to understand how future climate change could alter the population and genetic structure of desert animals, highlighting the value of `slimr` in a scientific workflow.

Data Availability Statement

The data that support the findings of this study are openly available in figshare at http://doi.org/[to be determined], or are included within the `slimr` package which can be installed or downloaded at https://github.com/rdinnager/slimr

Acknowledgments

We thank Benjamin C. Haller and Phillip W. Messer for permission to reproduce the documentation and examples of SLiM in `slimr`, and for valuable feedback on the package (from B.C.H.). Thanks to Emily Stringer for providing additional test data to help develop the methods used in this manuscript.

Author Contributions

RD, BG, SS, and RPD developed the concept for the package. SE, CD, GW, and AG provided feedback on the package design. RD coded the package and wrote the manuscript draft. CD, GW, and AG contributed data for testing of the package, and BG helped test the package as a user. All authors contributed critically to manuscript drafts and gave final approval for publication.

References

Beaumont, M. A., Zhang, W., & Balding, D. J. (2002). Approximate Bayesian computation in population genetics. *Genetics*, *162*(4), 2025–2035.

Brehmer, J., Louppe, G., Pavez, J., & Cranmer, K. (2020). Mining gold from implicit models to improve likelihood-free inference. *Proceedings of the National Academy of Sciences of the United States of America*, *117*(10), 5242–5249.

Carvajal-Rodríguez, A. (2010). Simulation of genes and genomes forward in time. *Current Genomics*, *11*(1), 58–61.

Cranmer, K., Brehmer, J., & Louppe, G. (2020). The frontier of simulation-based inference. *Proceedings of the National Academy of Sciences of the United States of America*, *117*(48), 30055–30062.

Dickman, C., Wardle, G., Foulkes, J., & de Preu, N. (2014). Desert complex environments. *Biodiversity and Environmental Change: Monitoring, Challenges and Direction*, 379–438.

Dickman, C. R., Greenville, A. C., Tamayo, B., & Wardle, G. M. (2011). Spatial dynamics of small mammals in central Australian desert habitats: the role of drought refugia. *Journal of mammalogy*, *92*(6), 1193–1209.

Greenville, A. C., Dickman, C. R., & Wardle, G. M. (2017). 75 years of dryland science: Trends and gaps in arid ecology literature. *Plos One*, *12*(4), e0175014.

Greenville, A. C., Wardle, G. M., & Dickman, C. R. (2012). Extreme climatic events drive mammal

irruptions: regression analysis of 100-year trends in desert rainfall and temperature. *Ecology and Evolution*, *2*(11), 2645–2658.

Greenville, A. C., Wardle, G. M., Nguyen, V., & Dickman, C. R. (2016). Population dynamics of desert mammals: similarities and contrasts within a multispecies assemblage. *Ecosphere*, *7*(5), e01343.

Haller, B. C., & Messer, P. W. (2019). SLiM 3: Forward Genetic Simulations Beyond the Wright-Fisher Model. *Molecular Biology and Evolution*, *36*(3), 632–637.

Hoban, S. (2014). An overview of the utility of population simulation software in molecular ecology. *Molecular Ecology*, *23*(10), 2383–2401.

Kelleher, J., Etheridge, A. M., & McVean, G. (2016). Efficient coalescent simulation and genealogical analysis for large sample sizes. *PLoS Computational Biology*, *12*(5), e1004842.

Marjoram, P., Molitor, J., Plagnol, V., & Tavare, S. (2003). Markov chain Monte Carlo without likelihoods. *Proceedings of the National Academy of Sciences of the United States of America*, *100*(26), 15324–15328.

Messer, P. W. (2013). SLiM: simulating evolution with selection and linkage. *Genetics*, *194*(4), 1037–1039.

Sisson, S. A. (2018). *Handbook of approximate bayesian computation*. Boca Raton, Florida : CRC Press, [2019]: Chapman and Hall/CRC.

Strand, A. E. (2002). metasim 1.0: an individual-based environment for simulating population genetics of complex population dynamics. *Molecular ecology notes*, *2*(3), 373–376.

Torada, L., Lorenzon, L., Beddis, A., Isildak, U., Pattini, L., Mathieson, S., & Fumagalli, M. (2019). ImaGene: a convolutional neural network to quantify natural selection from genomic data. *BMC Bioinformatics*, *20*(Suppl 9), 337.

Wang, Z., Wang, J., Kourakos, M., Hoang, N., Lee, H. H., Mathieson, I., & Mathieson, S. (2020). Automatic inference of demographic parameters using generative adversarial networks. *BioRxiv*.

Yuan, X., Miller, D. J., Zhang, J., Herrington, D., & Wang, Y. (2012). An overview of population genetic data simulation. *Journal of Computational Biology*, *19*(1), 42–54.