Running SLiM from R

Nick O'Brien

01/05/2020

Introduction

Running SLiM through RStudio (or calling it through an R command line) is quite simple and very convenient: with this, we can contain the entire workflow of running the simulation and the analysis in a unified script. We can also use R to run SLiM in parallel: running an instance of SLiM on each of your processor's cores.

Most of the code here has been modified from or inspired by code from the SLiM-Extras repository, which can be found here: https://github.com/MesserLab/SLiM-Extras

In this example, I'll run through an example of a simple model based on the default SLiM configuration (the default script when you create a new model in SLiMGUI). We will add deleterious mutations and track the default neutral mutations as an additive effect on an arbitrary phenotype. We will adjust the seed of the simulation, the recombination and mutation rates, and population size as treatments and we will automate this process through SLiM's command line arguments.

The SLiM Script

This script uses a custom function (defineCfgParam) to set up some constants for our parameters if they are not given in the command line (this is akin to a default state which allows the model to run in SLiMGUI). These won't be used when running from the command line, but if you wanted to see the dynamics of your model in SLiMGUI before running the full-fat, multiple-run model, this is a good option.

```
// set up a simple neutral simulation
initialize() {

    defineCfgParam("seed", asInteger(runif(1, 1, (2^63 - 1))));
    defineCfgParam("N", 500);
    defineCfgParam("r", 1e-8);
    defineCfgParam("mu", 1e-7);

    setSeed(seed);
    initializeMutationRate(mu);

    // m1 mutation type: neutral, modified to give normally distributed phenotype effects
    initializeMutationType("m1", 0.5, "n", 0.0, 1.0);

    // m2 mutation type: deleterious
    initializeMutationType("m2", 0.5, "g", -0.03, 0.2);

    // g1 genomic element type: uses both m1 and m2
```

```
    initializeGenomicElementType("g1", c(m1, m2), c(1.0, 0.1));

    // uniform chromosome of length 100 kb with uniform recombination
    initializeGenomicElement(g1, 0, 99999);
    initializeRecombinationRate(r);
}

// Helper Function for setting up command line variables: only creates the variable
// if it has not already been specified by command line arguments

function (void) defineCfgParam (string$ name, lifs value) {
    if (!exists(name))
        defineConstant(name, value);
}


// create a population of N individuals
1 {
    sim.addSubpop("p1", N);
}

fitness(m1) {
    return 1.0;         // Override the fitness of m1 mutations so they are all neutral
}

1:10000 late() {

// Each generation at the late stage in the generation cycle - grab the
// additive effects of each individual and sum them to a total phenotype,
// then get population means and variance
    inds = p1.individuals;
    phenotypes = inds.sumOfMutationsOfType(m1);
    popmean = mean(phenotypes);
    popvar = var(phenotypes);

// Set up a line of output to eventually stick together into an output file

    line = paste(c(paste(c(sim.generation, ", ", N, ", ", popmean, ", ", popvar,
                        ", ", getSeed(), ", ", r, ", ", mu, ""))));
    file = paste(line, "");
// Write the line to an output csv, with each generation appending
// another line to that csv
    writeFile("slim_output.csv", file, append = T);

}
```

This script overrides the fitness effects of m1 mutations to 1.0 (neutral), and repurposes the original fitness effects as a phenotypic effect. By summing them, we get the total additive effects of m1 mutations. More information on this process is in workshop sheet 13 in the SLiM Online Workshop.

The output of this model is taken by grabbing the generation number, population size, mean phenotype, variance of the phenotype, seed, recombination rate, and mutation rate. This is put into a single line and pasted into a .csv file line by line. This file can be imported into R for analysis using read.csv() or read.table() However, the file gets very large very quickly - it may be worth only printing a line once every 10 or 100

generations rather than every generation. This can be done via:

```
1:10000 late() {
  if (sim.generation % 100 != 0)     // Run every 100 generations
    return;
  ...
  }
```

This if statement simply checks if the simulation generation is a multiple of 100, and only runs the rest of the event if it is.

Now that we've got the SLiM script, we'll set it up for running through R.

Setting up parameters for SLiM and running SLiM from R

To feed parameters to SLiM, we can define them as vectors in R and then include them as arguments for SLiM in the command line.

```
rsample <- sample(1:2147483647, 100) # Get 100 seeds from the range of a 32 bit signed int
mu <- c("1e-8", "1e-7", "1e-6") # Mutation rates
recom <- c("0.0", "1e-8", "1e-7") # Recombination rates
N <- c(100, 250, 500) # Population sizes
```

This stores rsample (our seeds) as a vector of 100 integers, and both mu and recom as vectors of 3 characters each. We do this because when using small numbers in the command line they can be rounded to 0 if the number is small enough. By ensuring it is only treated as a numeric when it's actually in the SLiM script, we avoid this problem.

We're using a 32 bit integer's range to grab seeds from because R only supports 32 bit signed integers. Using packages it may be possible to use 64 bit integers for a more "random" sample set, but I'm unsure of if these will be compatible with the rest of R's functions. Another solution is to coerce the sample to a character (using as.character()). SLiM will still interpret these as integers when they are fed to it via the command line (64 bit in this case, since SLiM supports 64bit signed integers), but R will not. Ultimately this won't matter, because seeds are usually treated as discrete rather than continuous variables anyway, and analysis will likely require that you treat them as factors/characters. For this example, we will keep using 32 bit integers to generate seeds for simplicity.

We'll be using three packages to run SLiM in parallel: foreach, doParallel, and future

```
# Parallelisation libraries

library(foreach)
library(doParallel)
library(future)
```

Now that we've loaded these packages, we use doParallel and future to set up a local cluster of cores to run our simulation on: here, we'll use all of our computer's cores -

```
cl <- makeCluster(future::availableCores())
registerDoParallel(cl)
```

With the cluster set up, we can use foreach to loop each parameter in parallel:

```
#Run SLiM through the WSL, defining parameters in command line

foreach(i=rsample) %:%
  foreach(n=N) %:%
  foreach(j=mu) %:%
  foreach(k=recom) %dopar% {
  # Use string manipulation functions to configure the command line args,
  # then run SLiM with system(),
  slim_out <- system(sprintf("wsl slim -d seed=%i -d N=%i -d mu=%s -d r=%s tutorialR.slim",
                             i, n, j, k), intern=T)
}

stopCluster(cl)
```

Each variable in the foreach loops come from the vectors we defined earlier.

%:% is the nesting operator, allowing for the nesting of multiple foreach loops. %dopar% is the parallel operator, telling R to use the configured cluster.

Within the loop we use system() to call commands from the command prompt, using sprintf to include all the variables we need from the loop: i, n, j, and k (seeds, population sizes, mutation rates, recombination rates). %i, and %s refer to the data types that the call is expecting - integers or strings, respectively. %f refers to floats (or doubles as referred to in R). This coincides with the vector types we set up earlier (rsample is an integer vector, mu is a character/string vector etc.).

The command itself is pretty simple - it runs SLiM, defining various script-mentioned constants with -d and loads a script (which needs to be in the working directory of your R environment). wsl refers to the Windows Subsystem for Linux, a Windows 10 feature that allows for a Linux interpreter to be run within Windows. This is pretty easy to install; here is an installation guide: https://www.windowscentral.com/install-windows-subsystem-linux-windows-10

An important (but inconvenient) step if you are running through WSL is to turn off real-time protection in Windows Security: when running certain programs in the WSL environment, Windows Security hogs a lot of CPU time, slowing your simulation down considerably. This is a known bug which has been reported to Microsoft multiple times but has yet to be fixed: hopefully it will be soon! In the mean time, remember to turn protection back on after your simulations are done!

More details of how to set up the WSL are at the end of this tutorial. If you're running SLiM from a physical Unix environment (i.e. an actual Linux installation), obviously you don't need to prefix your command with wsl.

When you run this loop, you'll have all of your cores running a separate instance of SLiM, which will speed up your run significantly. The script will spit out a .csv in your working directory, which you can then import into R:

```
# Load the output file
slimout_tutorialR <- read.csv("slim_output.csv")

# Give it a header

names(slimout_tutorialR) <- c("gen", "N", "mean", "var", "seed", "r", "mu")
```

Now that you have your data set in R, you are free to do any kind of analysis available in R on any of the output variables you have stored. Hopefully this guide gives you an idea of how to get started, and inspires you to iterate and improve on the ideas and code shown here. The next step is to apply this to a remote cluster, such as Awoonga or Tinaroo, for remote execution of SLiM.

Bonus: WSL Setup

After following the instructions in the link I gave earlier, you should end up with a Linux distro installed. You'll need to build SLiM under this new environment. To do this, you'll need some build tools, which are included in the build-essential package. To install this, run the following command in your new linux command prompt:

```
sudo apt-get install cmake gcc clang gdb build-essential
```

You'll end up downloading and installing all the tools necessary to build SLiM from this package.

Now, we need to build SLiM. To do this, download the source code from http://benhaller.com/slim/SLiM.zip Extract the zip to a new folder called SLiM. Navigate to that folder in the linux environment and build SLiM:

```
cd /mnt/c/SLiM
# here, we are going to C:/SLiM
cd .. #go up one level
mkdir build # make a new directory called build
cd build # go to build
cmake ../SLiM # run cmake, making the necessary files to build SLiM and
              # putting them in the build folder
make slim # build slim
```

With all that done, double check that both eidos and slim have built properly using:

```
slim -testEidos
slim -testSLiM
```

More information on building SLiM from source (although, not specifically for the WSL) can be found in the SLiM Manual, pp. 49-51.