```
 ____  _           ____  _ _
/ ___|(_)_ __ ___ | __ )(_) |_
\___ \| | '_ ` _ \|  _ \| | __|
 ___) | | | | | | | |_) | | |_
|____/|_|_| |_| |_|____/|_|\__| version 4.13.17
```

User Manual

Author
Remi Matthey-Doret

Last revised September 5, 2020

1 Table of Contents

2 Correspondence

I would be happy to hear about your questions, bug reports or requests for new features. If you receive an error message starting by ”Internal error”, then please send me an email with the error message, SimBit’s version, and the input data. Please, when applicable, always make sure to provide a reproducible example of your problem (input data, SimBit version) and report the entire error message.

Remi Matthey-Doret
remi.b.md@gmail.com
matthey@zoology.ubc.ca
Beaty Biodiversity Research Center, room 205
Dept. of Zoology, University of British Columbia 6270
University Blvd, Vancouver, BC, V6T 1Z4 Phone: +1 (604) 369-5929

3 How to cite

The software is not published yet, so just cite the github page https://github.com/RemiMattheyDoret/SimBit.

4 A little a priori information

4.1 SimBit in a few words

SimBit is a flexible and fast forward in time simulation platform for finite site mutation models with arbitrary genetic architecture, selection scenario (including local selection, epistasis and all possible types of dominance), and demography. SimBit can simulate several species with their ecological interactions too. SimBit has been created with two main ideas in mind: Having a simple user interface with very good error report and to be extremely fast for a wide variety of scenarios. One way SimBit achieve such high performance is by allowing a diversity of internal representations of the genetics of individuals.

SimBit comes with a simple R wrapper that you will likely want to use to create input files to SimBit. The R wrapper is explained in the last section of this manual.

4.2 Contributers

Thank you to Michael C. Whitlock for his feedback on SimBit’s user interface and for proofreading this manual. Thank you also to the main beta tester, Pirmin Nietlisbach. Pirmin has also provided much advice on how to improve the manual.

4.3 How to obtain SimBit

SimBit is available on GitHub at https://github.com/RemiMattheyDoret/SimBit. If you are experiencing trouble downloading from GitHub, you might

want to have a look at the following link https://stackoverflow.com/questions/6466945/fastest-way-to-download-a-github-project.

4.4 How to compile SimBit

Using the command line (terminal), cd to the downloaded directory and do make. For example:

```
cd SimBit

make
```

This is it! The Makefile is ridiculously simple. You should now have an executable called SimBit in /bin. By default, the Makefile is using the standard C++ of 2014 (c++14; or any more recent standards) but SimBit can also compile on C++11 if you do:

```
make c++11
```

SimBit also uses a few boost files that you might need to download if you have not yet. You can download them from https://www.boost.org/users/ download/. If you want to be able to call SimBit without having to specify the whole path, just do it manually. For example, in MacOS, you could do:

```
touch ~/bash profile

open ~/bash profile
```

and, then write:

```
export PATH=”/PathToSimBit/SimBit/bin/:$PATH”
```

by changing `/PathToSimBit/SimBit/bin/` with the correct path on your machine.

4.5 How to read this manual

The manual is meant to be read from start to finish. Just take two hours to read it and unleash SimBit's full potential!

The two sections "technical options" and "performance options" can initially be skipped close to the end of the manual. You will probably want to read the section "R wrapper" at the very end though. The subsections of "Species, Generation and Habitat-specific options" can also be skipped (it will made clear as you arrive to it).

4.6 How to give arguments to SimBit and Presentation of options in the manual

SimBit works through the command line (in the terminal). To use SimBit, just call the executable and follow it with options starting with the prefix `--` (double dash). Each option is followed by an entry. For example:

```
--nbGenerations 1000
```

indicates that you want a simulation lasting for 1000 generations. SimBit will list all available options if you just call the executable without giving it any arguments.

```
./SimBit --option1 arguments for option 1 -- option2
arguments for option 2
```

For some options, there exists a short and a long name (or even several long names). For example to set the carrying capacity per patch the short name is `--N` and the long name is `--PatchCapacity`. If you do not remember the name of an option (for example, if you are not sure if it is `--PatchCapacity`, `--patchCapacity` or `--CarryingCapacity`, you can just type whatever comes to your mind and SimBit will look for option names that look alike (which has the lowest Levenshtein distance) and will suggest you another name! You do not need to bother about the ordering of the options. If an option is missing SimBit will use the default when a default is available. If no default is available for a missing option, if an option is present more than once, if two entries do not coincide, or if an entry is nonsense, then SimBit will throw an error message. It is also possible to put all arguments (same format) in a file and specify the file and the line of the file that contains the argument. If the argument file is called `ArgFile.txt` and the arguments of interest are found in the 12th line (the first line is numbered 1, not 0), then one can do

```
./SimBit file ArgFile.txt 12
```

instead of `file`, one can equivalently just write `F`, `f` or `FILE`. If you want SimBit to read all the lines in the file then write all (or just a) instead of the line number. For example:

```
./SimBit file ArgFile.txt a
```

The advantage of this technique is that SimBit will then ignore anything that follows the # sign on a given line allowing the user to leave comment in the input file. See the section "Launching basic simulations" for examples.

4.7 Genetic architecture - The basic types of locus

The representation of the genetic architecture is a key factor affecting flexibility and performance. Indeed, different types of simulation require different representations of the genetic architecture in order to maximize the performance. SimBit offers 5 types of locus that are referred to as T1, T2, T3, T4 and T5, which will be defined below. Loci of different types are integrated on the same recombination map (see the options `--L` aka `--Loci` and `--r` aka `--recombinationRate` below). T1 and T5 loci are meant to perform the same types of simulations but have different performance. T1 are meant to be used when there is high per locus genetic diversity while T5 are meant to be used when there is moderate to low genetic diversity per locus. It is hard to provide a good threshold value as it will depend from other elements in your simulations (the selection scenario, the recombination rate, etc.) and might even depend upon the processor you are using.

4.7.1 T1 loci

T1 loci track binary variables (e.g. mutated vs wildtype). SimBit has in memory for each haplotype an array of bits of the length of the number of T1 loci simulated. The n^{th} bit indicates whether the n^{t}h T1 locus of this haplotype is mutated or not. T1 loci have high performance for simulations with very high per locus genetic diversity.

4.7.2 T2 loci

T2 loci are meant to represent aggregate blocks of loci and counts the number of mutations happening in this block. This type should be used only when 1) the genetic diversity per T2 locus is very high, 2) when performance is a major concern, 3) you are satisfied with the limited selection scenario it can model and 4) a simple count of the number of mutations happening per T2 locus for each haplotype is a sufficient output for your needs.

4.7.3 T3 loci

T3 loci are quantitative trait loci (QTL) and code for a *n*-dimensional phenotype. The user can set the phenotypic effect of each T3 locus on each of the *n* axes of the phenotype and this can also be set to be dependent on the environment in order to simulate a plastic response. A user can also add random developmental noise (drawn from a gaussian distribution) in the production of a phenotype in order to reduce heritability. For T3 loci, the user can define a fitness landscape and an individual's fitness is given by its phenotype.

In the current version, a T3 locus is coded as a single byte. It can therefore only take $2^8 = 256$ different values (from -128 to 127). This is therefore only an approximation of a truly perfectly quantitative locus.

4.7.4 T4 loci

For T4 loci, SimBit computes the coalescent tree of the population over time and add the mutations onto the tree when the user asks for output. T4 loci are extremely fast when the recombination rate is low. T4 loci are inspired from Kelleher et al. (2018; already implemented in SLiM; Haller et al., 2018). T4 loci are necessarily neutral.

The advantage of T4 loci over T1 or T5 loci comes from the computational time. Use T4 loci if 1) you want many neutral loci (or the order of, say, 10^5 at least), 2) recombination rate is relatively low (of the order of, say, 10^{-7}, on average between any two locus). T4 loci are extremely fast when dealing with recombination rate of the order of 10^{-9} and lower but will be much slower than T1 and T5 loci for cases of high recombination and cases with few loci. Note that variance in recombination rate throughout the genome typically helps making T4 faster. General advice: don't assume that one is faster but try it out for your parameters.

T4 loci can also be used to "paint haplotypes" at a given generation and study the spread of these haplotypes (or segments of those haplotypes rather) through time and space. Finally, SimBit can also make a burn in until all T4 loci have coalesced (described in section "Initial population").

4.7.5 T5 loci

T5 loci are very similar to T1 loci. Two simulations with the same random seed differing only by the fact that one uses T1 loci and the other uses T5 loci will produce the same output. The big difference is how SimBit tracks their values. For each haplotype, SimBit has an array with the position of each T5 locus that is mutated. T5 loci tend to perform better than T1 loci for moderate to low genetic diversity.

Behind the scene, SimBit will track separately T5 loci that are under selection (which it calls T5sel) and T5 loci that are neutral (which it calls T5ntrl) for improved performance. SimBit can also compress T5 loci (T5ntrl and/or T5sel) information in memory. Behind the scenes again, the compressed T5 loci are actually called T6 (T6sel and T6ntrl for selected and neutral loci, respectively) but I don't think you really needed to know that! Compression reduces the RAM usage by (up to) a factor of 2. It can also increase or reduce CPU time depending on the simulation scenario. By default, SimBit makes this compression (on the neutral T5 loci only) only when it is certain it will improve performance (which is when the number of T5 neutral loci is between 10 and 2^{16}). For advanced users, it is also possible to ask SimBit to invert the meaning of some loci depending on their frequencies. For example, if the locus 23 is fixed or quasi-fixed, then SimBit can invert the meaning of having the number 23 in its haplotype description. As a result, a haplotype would track this 23rd locus only if they carry the non-mutated allele. These advanced performance tweaks are explained in the section "performance options".

4.8 unif and A

In order to indicate input data in a convenient way SimBit uses a number of different Modes of input. Each specific option has its list of Modes but two Modes that appear over and over again are `unif` and `A`. `A` stands for "All entries" saying that you want to input as many values as needed. `unif` stands for "uniform" saying that you want all elements to be set to the same value. As an example, to set four patches

```
--PN 4
```

or

```
--PatchNumber 4
```

to a carrying capacity of 1000, then you could do

```
--N unif 1000
```

or equivalently

```
--N A 1000 1000 1000 1000
```

Note that SimBit also understand the scientific notation with the a`e`^{b} standing for a×10^{b}. For examples, `1e-4` is `0.0001` and `5.27e3` is `5270`. As such, the above entry could also be written

```
--N unif 1e3
```

4.9 seq, seqInt, rep and fromToBy

There are special keywords `seq`, `seqInt`, `rep` and `fromToBy`. When SimBit reads the input, it first split the input by the option names (which start with a double dash such as `--T1_fit`), and it then directly evaluates the keywords `seq`, `seqInt`, and `rep`.

The keywords `seq` and `seqInt` are analogous to the function "seq" in R. They both expects three values: the "from" value, the "to" value and the "by" value. `seqInt` is to be used for integer values while `seq` is for float values. For example the input `seqInt 5 17 2` can be read as "from 5 to 17 by 2" and is equivalent to `5 7 9 11 13 15 17`.

The keyword `rep` is analogous to the function "rep" in R. `rep` expects two values the "whatToRepeat" value, the "howManyTimes" value. For example the input `rep 4 5` is equivalent to `4 4 4 4 4`. It is also possible to feed a vector as the first argument. While the `seq` keyword expect numbers only, the `rep` keyword expects only the second argument to be an integer. The first argument can be any string.

The keywords can mixed at will. For example `3 4 rep hello 3 0 seq 1 2 0.3` is equivalent to `3 4 hello hello hello 0 1.0 1.3 1.6 1.9 3`. In older versions of SimBit a keyword `R` existed and was equivalent to the current `rep`. `R` is now deprecated.

The keyword `fromToBy` (or `fromtoby or FromToBy`) can only be used for output related option (see section “Outputs”). `fromToBy` works exactly like `seqInt` excepts that it accepts the keyword `end` to indicate the last generation of the simulation. For example:

```
--nbGenerations 60000 --fitnessStats_file fitFile fromToBy
0 end 100
```

asks for the output file showing the fitness summary statistics, the file name will be "fitFile" and the outputs will be printed every hundred generations from generation 0 and up to the last generation of the simulation (generation 60000).

4.10 Listing all options

If you run SimBit without giving it any arguments, then it will list all the available options.

4.11 SimBit Version

Every time you run the executable, SimBit prints its version (bottom right of the logo) in standard output.

5 Species, Generation and Habitat-specific options

Most options are species-specific, generation-specific and/or habitat-specific. SimBit uses the markers `@S` for species, `@G` for generation and `@H` for habitat ("at" symbol followed by either S, G or H). As such to refer to the, say, 120th generation one would write `@G120`. If you want to simulate a single species, a single type of habitat (no environmental heterogeneity) and no change over time, then you do not have to bother but these `@S`, `@G`, `@H`.

All these species-, habitat-, and generation-specific markers must come in order. For example,

```
--N @G0 50 @G130 1000
```

is correct but

```
--N @G130 1000 @G0 50
```

leads to an error message. Similarly, habitats that are named by indices must come in order. For example,

```
--T1_fit @H0 unif 1 1 0.99 @H1 unif 1 1 0.9
```

is correct but

```
--T1_fit @H1 unif 1 1 0.9 @H0 unif 1 1 0.99
```

leads to an error message. Finally, species are named but you have to follow the ordering you used when naming these species. For example

```
--species Quercus Fagus --N @SQuercus A 1000 @SFagus A
60000
```

is correct but

```
--species Quercus Fagus --N @SFagus A 60000 @SQuercus A
1000
```

leads to an error message. Note that with the species markers, you can also use species index instead of species name. For example, you can do

```
--species Quercus Fagus --N @S0 A 1000 @S1 A 60000
```

In general, it is quite intuitive what options are species-, generation-, and habitat-specific. All options regarding the fitness are both species-specific and habitat-specific. All options regarding the number of patches, the carrying capacity and the migration rates are both species-specific and generation-specific. All options regarding the genetic architecture, mating systems and fecundity are species-specific only. Note that while some options are meant to indicate species interaction (`--eco` aka `--speciesEcologicalRelationships`), it does not mean that the option is species-specific, in the sense that it does not take any `@S` marker.

Here is the entire list of options that are species-specific and generation-specific (but not habitat-specific)

```
--N (aka --patchCapacity)
--H (aka --Habitats)
--m (aka --DispMat)
```

Here is the entire list of options that are species-specific and habitat-specific (but not generation-specific)

```
--T1_epistasis (aka --T1_EpistaticFitnessEffects)
--T1_fit (aka --T1_FitnessEffects)
--T2_fit (aka --T2_FitnessEffects)
--T3_pheno (aka --T3_PhenotypicEffects)
--T3_fit (aka --T3_FitnessLandscape)
--T3_DN (aka --T3_DevelopmentalNoise)
```

Here is the entire list of options that are species-specific (but not habitat-specific or generation-specific)

```
--nbSubGens (aka --nbSubGenerations)
--T5_approximationForNtrl (aka --T56_approximationForNtrl)
--T5_fit (aka --T5_FitnessEffects)
--T5_compressData (aka --T5_compress)
--L (aka --Loci)
--fec (aka --fecundityForFitnessOfOne)
--DispWeightByFitness
--gameteDispersal
--InitialpatchSize
--cloningRate
--selfingRate
--matingSystem
--additiveEffectAmongLoci
--selectionOn
--T1_mu (aka --T1_MutationRate)
--T2_mu (aka --T2_MutationRate)
--T4_mu (aka --T4_MutationRate)
--T4_maxAverageNbNodesPerHaplotype
--T5_mu (aka --T5_MutationRate)
--T5_toggleMutsEveryNGeneration
--T5_freqThreshold (aka --
T5_frequencyThresholdForFlippingMeaning)
--T3_mu (aka --T3_MutationRate)
--r (aka --RecombinationRate)
--recRateOnMismatch
--FitnessMapInfo
--indTypes (aka --individualTypes)
--resetGenetics
--indIni (aka --individualInitialization)
--T1_ini (aka --T1_Initial_AlleleFreqs)
--T5_ini (aka --T5_Initial_AlleleFreqs)
--popGrowthModel
--stochasticGrowth
--swapInLifeCycle
--readPopFromBinary
--geneticSampling_withWalker
--individualSampling_withWalker
```

Here is the entire list of options that are generation-specific (but not species-specific or habitat-specific)

```
--PN (aka --PatchNumber)
```

Note that the number of patches is the same for all species so that we can identify them as being in the same patch. Simulating a species absent from a given patch is achieved by setting is carrying capacity for this patch at 0.

Finally, here is the entire list of options that are neither species-specific, generation-specific or habitat-specific

```
--seed (aka --random_seed)
--printProgress
--nbGens (aka --nbGenerations)
--startAtGeneration
--S (aka --species)
--LogfileType
--sequencingErrorRate
--GP (aka --GeneralPath)
--T1_vcf_file (aka --T1_VCF_file)
--T1_LargeOutput_file
--T1_AlleleFreq_file
--Log (aka --Logfile and --Logfile_file)
--T1_MeanLD_file
--T1_LongestRun_file
--T1_HybridIndex_file
--T1_ExpectiMinRec_file
--T2_LargeOutput_file
--SaveBinary_file
--T3_LargeOutput_file
--T3_MeanVar_file
--fitness_file
--fitnessSubsetLoci_file
--fitnessStats_file
--T1_FST_file
--T1_FST_info
--extraGeneticInfo_file
--patchSize_file
--extinction_file
--genealogy_file
--coalesce (aka --shouldGenealogyBeCoalesced)
--T4_LargeOutput_file
--T4_vcf_file (aka --T4_VCF_file)
--T4_SFS_file
--T1_SFS_file
--T4_printTree
--T4_coalescenceFst_file
--T5_vcf_file (aka --T5_VCF_file)
--T5_SFS_file
--T5_AlleleFreq_file
--T5_LargeOutput_file
--outputSFSbinSize
--eco (aka --speciesEcologicalRelationships)
--Overwrite
--DryRun
--centralT1LocusForExtraGeneticInfo
--killOnDemand
```

If you feel the need to read more on that matter, please read the following three subsections. Otherwise, you should probably skip these sections and go straight to section "Launching basic simulations".

5.1 Species-specific options

Most options are species-specific. All species must share the same geographic location. Therefore, the number of patches is not specific per species. However, the carrying capacity can vary among species. If you want a species to be absent from a patch, just set its carrying capacity to zero. All options regarding the genetic architecture, selection scenarios and demography are species-specific.

You need to name your different species with the option

```
--species name1 name2 name3 …
```

Species

This option is aka `--S`. Each species needs a unique name. By default, SimBit assumes a single species (called “sp”). The only species name that is not accepted is “seed” for reasons explained in the “Outputs” section.

5.2 Generation-specific options

The following two examples are equivalent

```
--nbGenerations 2000 --PatchNumber 1
```

```
--nbGenerations 2000 --PatchNumber @G0 1
```

and the two following examples are also equivalent.

```
--nbGenerations 2000 --PatchNumber @G0 1 @G1000 5 @G1200 1
```

```
--nbGenerations 2000 --PatchNumber 1 @G1000 5 @G1200 1
```

Consider the following example

```
--nbGenerations 2000 --PatchNumber @G0 1 @G1000 3 @G1200 1
--N @G0 unif 100 @G500 unif 1500
```

This command would indicate that temporal changes will occur at generations 0, 500, 1000 and 1200. From generation 0 to 500, there is 1 patch of 100 individual. From generation 500 to 1000, there is still one patch but with 1500 individuals. From generation 1000 to 1200 there are 3 patches of 1500 individuals each and from generation 1200 to generation 2000, there is one patch of 1500 individuals.

The following two examples are also equivalent

```
--nbGenerations 2000 --PatchNumber @G0 1 @G1000 3 --N @G0 A
10 @G500 A 100 @G1000 A 100 100 100
```

```
--nbGenerations 2000 --PatchNumber @G0 1 @G1000 3 --N @G0 A
10 @G500 unif 100
```

It simulates one patch up to generation 1000 and three patch then. The patch sizes are 10 up to generation 500, then it is 100 up to end of the simulation. I think our intuition want us to rewrite `@G1000 A 100 100 100,` but it is in fact not required as `@G500 unif 100` means to apply `unif 100` any time after generation 500.

Many options are both species-specific and habitat-specific or species-specific and generation-specific. For such cases, always indicate the species first, and then the habitat or generation. For example,

```
--species wolf rabbit --PatchNumber 1 --N @Swolf @G0 unif
100 @G500 unif 200 @Srabbit unif 2e3
```

specifies two species that live over a single patch. The carrying capacity of wolf is 100 for the first 500 generations and then it increases to 200. The carrying capacity for rabbits is 2000 all the way through the simulation.

5.3 Habitat-specific options

All options that relate to the selection scenario (such as `--T1_fit` for example) and phenotypes (`--T3_pheno`) are habitat-specific. A habitat has to be understood in its ecological definition. A habitat could also have been called an environment. Several patches may belong to the same habitat and the habitat a given patch belong to can change over time. Patches are associated to habitat and habitat associated to specific selection scenario allowing local and temporal variation in selection. Let me explain how.

To associate patches to habitat, use the following option

```
--Habitats mode int
```

Habitats

This option is a generation-specific and is also written as `--H`. One can therefore change the association between patch and habitat over time and therefore to change selection pressure over both space and time. Two modes are available, `A` and `unif`. When using mode `A`, the number of entries must be of the same length as the number of patches. Consider for example

```
--nbGens 1000 --PatchNumber @G0 1 @G500 4 --Habitats @G0
unif 0 @G500 A 0 0 1 0
```

This indicates that from generation 0 to 500 there is only one patch which belongs to habitat 0. From generation 500 to 1000, there are 3 patches, the first two patches as well as the last one belong to habitat 0 and the third patch belong to habitat 1. Note that habitat 0 must always exist and it is impossible to specify an habitat index without specifying the previous one. For example it is impossible to specify habitat 5 without specifying habitats 0, 1, 2, 3 and 4. If the option `--Habitats` is absent, SimBit assumes that all patches belong to habitat 0.

This system of associating each patch to a habitat in a time-specific manner is a very convenient solution to indicate variation in selection pressures through time and space. For all options concerning selection, one must indicate for which patch a given selection scenario applies using the `@Hx` notation, where x is the habitat in question. More information about fitness related options in the section “Selection”.

6 Life cycle

Here is the ordering of events at each generation. Many of the details of the following can only be understood when reading below the specific of the option that affect each step.

For each generation

- Print generation counter

- Update parameters and populations for each species (only when using temporally variable parameters).

- For each species

- For each sub-generation (see --nbSubGeneration)
    - Compute fitness of each individual in the population (see section Selection and Phenotypes)
    - Compute backward migration matrix (see section Demography and Species Interactions)
    - for each patch
        - for each offspring to produce
            - sample which patch the first parent come from
            - sample the parent from this patch (based on finesses if selection is on fecundity)
            - Depending on the details of whether dispersal happen at the gametic or zygotic phase, whehter we have males and females or hermaphrodites, whether we authorize cloning and/or selfing, repeat the previous two points for the other parent.
            - Simulate recombination and segregation and save this info into the T4 coalecent tree (only when using T4 loci).
            - Simulation mutations (and directly modify fitness values when using the assumption of multiplicative fitness)
            - If selection is on viability, test if offspring produced is fit enough and if not, then go back to sample the patch the first parent comes from.
            - Save new offspring into genelogy (see `--genealogy_file`)
    - Check if species got extinct
    - Eventually simplify coalescent tree if T4 loci are used
    - Eventually toggle meaning of T5 loci (see --`T5_toggleMutsEveryNGeneration`)

    - if we reach last generation, free memory from parent population (that can help reduce RAM usage at the end of the simulation as user often ask to compute outputs at the last generation)

    - Rescale T2 loci if they reached 256 (only when using T2 loci).

- For each species

    - Redefine individual types (see `--redefIndTypes`)

    - Reset population genetics following instructions to option `--resetGenetics`

    - Kill specific pre-determined individuals (see `--killIndividuals`).

    - Run the inserted code (see codeToInsert.cpp for advanced users)

    - write outputs if needed

    - print population to binary file if needed

    - test if simulation should get killed before the last generation and kill if needed (see `--killOnDemand`)

- offsprings become parents

- Increment generation counter

7 Launching basic simulations

SimBit requires a quantity of basic information to make a simulation. This information is the number of patches (`--PN` aka `--PatchNumber`), the number of individuals per patch (`--N` aka `--PatchCapacity`), the number of loci, their types and physical ordering on the chromosome (`--L` aka `--Loci`), the number of generations (`--nbGens` aka `--nbGenerations`), the mutation rate for the type of locus indicated to option `--L` (aka `--Loci`). Mutation rates for each locus of each type `--T1_mu` (aka `--T1_MutationRate`), `--T2_mu` (aka `--T2_MutationRate`), `--T3_mu` (aka `--T3_MutationRate`), `--T4_mu` (aka `--T4_MutationRate`), `--T5_mu` aka `--T5_MutationRate),` and the dispersal rate `--m` (aka `--DispMat`; only required if there is more than one patch). Finally, the recombination rate is set with `--r` (aka `--RecombinationRate`). Note, by the way, that I call each different panmictic patch in a structured population, a patch and not a subpopulation or a deme.

For a start we will consider the following example:

```
SimBit --nbGeneration 5000 --PatchNumber 4 --N A 100 100
100 100 --m Island 0.01 --L T1 3 --T1_mu A 0.00001 1e-8 1e-
8 --r rate A 1e-6 1e-6
```

As explained already in section “How to give arguments to SimBit”, if you prefer to write your command on a file than directly in the terminal, you can do

```
<In example1.txt>

    --nbGeneration 5000 --PatchNumber 4 --N A 100 100 100
    100 --m Island 0.01 --L T1 3 --T1_mu A 0.00001 1e-8 1e-
    8 --r rate A 1e-6 1e-6

<In terminal>

    SimBit file example1.txt 1
```

The number `1` at the end indicates that SimBit must read the first line from the file “example1.txt”. This allows a user to gather a number of different commands in the same file. As explained above, one could also split that command over several lines (which has the added advantage that you can comment in the file) and then read the command with

```
<In example2.txt>

    # Set the number of generations
    --nbGenerations 5000      # 5000 generations

    # Set the number of patches
    --PatchNumber 4          # four patches

    # Set the carrying capacity.
    --N A 100 100 100 100    # N=100 per patch

    # Set the migration scenario
    --m Island 0.01          # island model

    # Set the genome map
    --L T1 3                 # Three T1 loci

    # Set the mutation rate
    --T1_mu A 0.00001 1e-8 1e-8

    # Set the recombination rate
    --r rate A 1e-6 1e-6



<In terminal>

    SimBit file example2.txt all
```

This simulation should run in a few seconds. Some elements of the above command may make little sense to you so far, so let's go through it. The command asks for a simulation lasting 5000 generations with 4 patches of 100 individuals each. Migration follows a classical island model in which the probability of migrating is 0.01. Each haplotype is made of 3 loci of type T1. The mutation rate per locus is 0.00001 (10^{-5}), 10^{-8} and 10^{-8}, respectively. The recombination rate between any adjacent locus and the next is 10^{-6}.

The first option is quite straightforward, the argument of `--nbGeneration` is a single integer number.

```
--nbGenerations int
```

Number of generations

The option is aka `--nbGens`

The second option here indicates number of patches

```
--PatchNumber int
```

Number of patches

The option generation-specific and is aka `--PN`. It is a generation-specific option.

The third indicates the carrying capacity for each patch

```
--N mode int(s)
```

Carrying capacity

This option is species- and generation-specific aka `--PatchCapacity`. The two modes are `A` and `unif`. Above, I set all four patches at the same carrying capacity

```
--N A 100 100 100 100
```

Instead, I could have used `unif`

```
--N unif 100
```

or I could have used rep

```
--N A rep 100 4
```

The following option sets the migration scenario

```
--m mode value
```

Dispersal

This option is species- and generation-specific and is aka `--DispMat`. The modes are `A`, `LSS`, `OnePatch`, `Island` (or `island`), `LinearNormal` and `2DSS`. `A` is the most flexible mode of entry and expect the user to input the entire matrix of dispersal probabilities. Note that the probabilities inputted are forward migration rates. Backward migration rates are being computed by SimBit based on various options presented (e.g is migration probabilities weighted by patch mean fitness) More information about these different modes in table 1. By default `--m` is set to `OnePatch`. More information about demography and how to interpret migration rates in section Demography and Species Ecology.

Table 1: Input format for migration scenario

Mode	Meaning	Example
`A`	Input a PatchNumber x PatchNumber square matrix. With three patches, the fourth element is the element of the second row, first column of the matrix and hence indicate the probability of migrating from the second patch to the first.	`--m A 0.9 0.1 0 0 0.1 0.8 0.1 0 0 0.1 0.8 0.1 0 0 0.1 0.9` It simulates a 4 patches stepping stone model
`LSS`	It stands for 1D Linear Stepping Stones. Here the first element is the number of probabilities to expect next. Then is a vector of probabilities and finally is which element of this vector (zero based counting) corresponds to the probability of not migrating. The resulting dispersal matrix is corrected at the edge (reflective boundary effect). The input format is LSS NbProbabilities <probabilities> center	`--m LSS 4 0.05 0.05 0.75 0.15 2` indicates that the probability of not migrating is 0.75, the probability of migrating one patch on the left is 0.05, the probability of migrating two patches on the left is 0.05, the probability of migrating one patch on the right is 0.15
`OnePatch`	Outside of `A` and `isolate`, `OnePatch` is the only possible entry when there is only a single patch. It is the default.	`--PN 1 --m OnePatch`
`Island`	This creates a classical island model. One value is expected which is the probability of migrating.	`island 0.01` It creates an island model where the probability of migrating from any patch to any other patch is 0.01
`LinearNormal`	This creates a 1D dispersal kernel that is approximated by a normal (Gaussian) function. The two entries expected are the standard deviation of this Gaussian distribution (in number of patch) and the number of standard deviations above and below which the probability of migrating will be approximated to zero.	`--m LinearNormal 2 4` It indicates a 1D gaussian distribution kernel with 2 patches of standard deviation and that after 4 standard deviation, the probability of migrating will be considered sufficiently low to be approximated to 0.
`2DSS`	This creates a rectangular world with a 2 dimensional Stepping Stone migration model. It expects three entries; the number of rows, the number of columns and the probability to migrate to any of the four adjacent patches (no dispersal in the diagonal).	`--m 2DSS 10 50 0.05` It creates a world of 10 rows by 50 columns (for a total of 500 patches) and set a migration rate of 0.05 between any patch and its direct neighbour.
`isolate`	This creates a world where all patches are perfectly isolated from each other.	`--m isolate`

It is often pleasant to graph input parameters and esp. to graph the migration scenario just to make sure that what is being simulated matches what we want to simulate. For this, please use `--LogfileType 2` (see subsection Logfile in section Output) and consider having a look at the resulting log file and at the example in the graphParameters folder.

The next option sets the number of loci of each type as well as their ordering on the chromosomes

```
--L LocusType nbLoci LocusType nbLoci LocusType NbLoci …
```

Loci

This option is species-specific and is aka `--Loci`. For example, if you want 23 T3 loci, followed by 1000 T1 loci followed by 12 T3 loci, you could input

```
--L T3 23 T1 1e3 T3 12
```

Note that instead of `T3 12`, you could have `T3 6 T3 4 T3 2`, and it would be equivalent. Lower case T is also accepted (e.g. `t1`). You can even ignore the Ts (e.g. `--L 3 23 1 1e3 3 12`) but it not very human readable. The recombination rate between any of these loci, including, the placing of loci on independent chromosomes is indicated via the option `--r` (aka `--RecombinationRate`) presented below. Because there are 23 + 1e3 + 12 = 1045 loci, the option `--r` will expect 1044 entries. You will likely only need one type of loci and your input might simply look like

```
--L T5 1e4
```

This input asks for 10,000 T5 loci.

The next option sets the mutation rate on T1 loci

```
--T1_mu mode float(s)
```

Mutation rate on T1 loci

This option is species-specific and is aka `--T1_MutationRate`. The two modes are `A` and `unif`.

The options --T2_mu, --T3_mu, --T4_mu and --T5_mu are not present in the above code, but I will mention it here for their similarity with --T1_mu.

```
--T2_mu mode float(s)
```

Mutation rate on T2 loci

```
--T3_mu mode float(s)
```

Mutation rate on T3 loci

```
--T4_mu mode float(s)
```

Mutation rate on T4 loci

```
--T5_mu mode float(s)
```

Mutation rate on T5 loci

All are species-specific and all can also be named `--Tx_MutationRate` (e.g. `--T5_MutationRate`).

By default, a mutation toggle the allelic value from *wild type* to *mutated* and vice-versa. However, this can be changed with the options

```
--T1_mutDirection bool
```

T1 Mutation Directionality

```
--T4_mutDirection bool
```

T4 Mutation Directionality

```
--T5_mutDirection bool
```

T5 Mutation Directionality

Note that mutations do not "stack" as in SLiM and mutations do not disappear once fixed. Note also that backward mutations can happen before a mutation has reached fixation. The last option in the above example sets the recombination rate between any two loci (whatever their type)

```
--r unit mode float(s)
```

Recombination

This option is species-specific and is also written as `--RecombinationRate`. There are 3 possible units; `rate`, `cM` and `M`. `rate` means that values represent rate of recombination. `cM` indicates that values represent centiMorgans. `M` indicates that values represent Morgans (1 Morgan = 100 centiMorgans). Of course, there is not much difference between `M` and `rate` if distances are not too high (say below 0.1).

Again there are two modes; `A` and `unif`. With `A`, the number of entries should equal the total number of loci minus 1. As an example

```
--r rate unif 1e-7
```

It sets the recombination rate between any two adjacent loci to 0.0000001. To indicate perfectly independent loci (a chromosome break), just give it a rate of `0.5`. If you are using `cM` or `M`, you can just say `-1,` and it will be understood as a chromosome break.

Consider the following complex example

```
--L T5 100 T3 3 T5 100 --r cM A rep 1e-5 99 -1 0.1 0.1 -1
rep 1e-5 1e-5
```

It creates three independent chromosomes. Two of them are (almost) 0.01 cM in length and contained 100 T5 loci each (each T5 locus is at a distance of 0.00001 cM from the previous locus on this chromosome) and one chromosome of 0.2 cM with three T3 loci (each T3 locus is at a distance 0.1 from the previous locus on this chromosome). Note that it would not be quantitatively different to change the order of chromosomes as in the following example.

```
--L T5 100 T5 100 T3 3 --r cM A rep 1e-5 99 -1 rep 1e-5 1e-
5 -1 0.1 0.1
```

8 Selection and phenotype

8.1 General concepts

The selection scenario can be set independently for each type of locus with options `--T1_fit`, `T1_epistasis`, `--T2_fit`, `--T3_fit` and `--T5_fit` (T4 loci are always neutral, there is therefore no `--T4_fit`). Fitness effects among loci types of different types are always multiplicative.

For some types of loci (see below), SimBit can make use of an assumption about the selection scenario that can provide substantial improvement in run time. I call this assumption the "multiplicative fitness" assumption (abbreviated "multfit")". The multiplicative fitness assumption assumes that fitness effects are multiplicative among loci too and that the fitnesses of the three possible genotypes are 1, 1 - s and $(1 - s)^2$. With this assumption, dominance coefficients are very close to 0.5 (additivity) especially for small selection coefficients. For examples, if the double mutant homozygote fitness is 1 - t = 1 - 0.001, then h ≈ 0.5001. If 1 - t = 1−0.1, then h ≈ 0.51. When taking advantage of the assumption of multiplicative fitness, SimBit partitions a haplotype into blocks and computes the fitness value for each block. If, during reproduction, no recombination events happen within a given block, then SimBit will not need to recompute the fitness for this specific block as the fitness of the block can simply be multiplied over by the fitness of the same block on the other haplotype. This technique yields substantial performance improvement in terms of CPU time (especially when recombination rate within blocks is relatively low). SimBit does a decent job at choosing the size of blocks, but a user can have complete control over the block sizes with the option `--FitnessMapInfo` (see section "Performance options"). Unless the exact dominance relationship is of central importance, it is generally recommended to make use of this assumption (especially when recombination rate is low and when there are a fair amount of loci).

All selection scenarios described below (including epistasis) are habitat-specific, hence allowing any kind of spatial and temporal variation in selection pressures (as a reminder, the matching between patches and habitats with option `--H` (aka `--Habitats`) is generation-specific). An example of spatial and temporal variation of selection scenario is provided at the end of the "Selection and phenotype" section. By default, selection happens on fertility, but it can also be simulated on viability or on both fertility and viability.

```
--selectionOn info
```

Selection on fertility and/or viability

This option is species-specific. The “info” is either `fertility`, `viability` or `both`. Selection on fertility is faster than selection on viability. It is the default and recommended mode. As an example,

```
--selectionOn viability
```

require selection to be on viability and not on fertility.

8.2 T1

On T1 loci, a user can either set the fitness values of each of the three genotypes or take advantage of the multiplicative fitness assumption and only provide a single fitness value per locus.

```
--T1_fit mode float
```

Selection on T1 loci

This option is species- and habitat- specific and is aka --T1_FitnessEffects. The possible modes are `A`, `domA`, `cstH`, `unif`, `multfitA` (aka `MultiplicityA`), `multfitUnif` (aka `MultiplicityUnif`), `multfitGamma` (aka `MultiplicityGamma`). Table 2 summarizes the different mode of entries. All modes starting by `multfit` (or "`multiplicity`" for alternative names) means that you are willing to assume "multiplicative fitness" that is genotype 00 (unmutated homozygote) has a fitness of 1, genotype 01 (heterozygote) has a fitness 'x' given in input and genotype 11 (mutated homozygote) has a fitness x^2. Note that in the current version, it is impossible to make the `multfit` assumption for a given habitat but not for another one. It will raise an error message. There is no good reason for this limitation. If you need to get rid of it, you can ask me for help.

Table 2: Input format for selection on T1 loci

Mode	Meaning	Example
A	Indicates the fitness of all three genotypes at all loci.	--L T1 2 --T1_fit A 1 1 1 1 0.98 0.9
cstH	first indicate a dominance coefficient "h", then indicate the keyword "hetero" or "homo". Then indicate for each locus the fitness of either the heterozygote or the double mutant homozygote fitness (depending on the keyword). The fitness of the other genotype will be computed from h. The fitness of the double wild type is always 1.0.	--L T1 3 –T1_fit 0.5 hetero 0.95 0.95 0.9
domA	First indicate either the keyword cst (stands for “constant”) or fun followed by a float number indicating the average dominance coefficient H. if cst is used, then all loci have the same dominant coefficient H. If fun is used, then the dominance coefficient at all loci is given by $h_i = e^{-k s_i}/2$, where k = -log(2H) / S, where S is the average selection coefficient.	--L t1 3 domA fun 0.2 0.95 0.98 0.9
unif	Indicates the fitness components for all three genotype and assume that all loci are under the same selection scenario.	--L t1 –T1_fit 1e5 unif 1 0.98 0.6
multfitA	Indicates the fitness components of the genotype 01 at each locus separately and makes the assumption of multiplicative fitness.	--L T1 3 –T1_fit MultiplicityA 0.9 1.0 0.9
multfitUnif	Indicates the fitness components of the genotype 01 for all loci makes the assumption of multiplicative fitness.	--L T1 1e6 –T1_fit multfitUnif 0.99
multfitGamma	Just like multfitA or multfitUnif except that the fitness values are 1 minus the selection coefficient which are drawn from a gamma distribution with parameters alpha and beta given by the user.	--L T1 1e6 –T1_fit multfitGamma 0.111 2.22

For epistatic interactions use

```
--T1_epistasis loci <lociSet> fit <fitnesses> loci
<lociSet> <fitnesses> …
```

Epistasis

This option is species- and habitat- specific and is aka `--T1_EpistaticFitnessEffects`. Note that `--T1_epistasis` is independent of `--T1_fit`. As such a locus could be under both non-epistatic selection and epistatic selection. The effects would be multiplicative. It is up to the user to decide whether (s)he want such thing or not. If this is the case SimBit will throw a warning message though just to make sure you know what you are doing. Note also that one locus can belong to more than one set of loci for which an epistatic interaction is defined. Note also, that epistasis is only available on T1 loci and not on T5 loci (mainly for performance reasons).

The user can specify any number of set of loci that are in epistatic interactions and each set can contain any number of loci. If you specify n loci after keyword loci, then SimBit will expect 3^n fitness values after the keyword fit. For a three-locus interaction, SimBit expects $3^3 = 27$ fitness values. The first fitness value entered is the fitness for when the individual is homozygous wild type for the three loci. As an example

```
--T1_epistasis loci 0 5 fit 1 0.9 0.8 0.9 0.9 0.9 0.8 0.9 1
loci 2 3 fit 1 0.9 0.8 0.8 0.9 1 1 0.9 0.8
```

In this example, the loci 0 and 5 have an additive by additive epistatic interaction while the loci 2 and 3 have an additive by dominance epistatic interaction.

8.3 T2

```
--T2_fit mode float(s)
```

Selection on T2 loci

This option is species- and habitat- specific and is aka `--T2_FitnessEffects`. Here there are only three modes: `A`, `unif` and `gamma`. In all cases, it assumes “multfit” of dominance effects (one may argue that the modes might better be renamed `multfitA`, `multfitUnif` and `mulfitGamma`).

8.4 T3

For T3 loci, one must indicate how the genotypes match to phenotypes with

```
--T3_fit @S0 int @H0 mode float @H1 mode float ... @S1
```

T3 phenotype

This option is species- and habitat- specific and is also known as `--T3_PhenotypicEffects`. The input format is a bit unusual here as this option expects an integer value and then habitat-specific arguments each with a mode float numbers. This is the reason I included the `@S` and `@H` in the presentation of the option. The integer value is the number of dimensions of the phenotypic space. There are two modes, `A` and `unif`, the expected number of entries is the number of dimensions of the phenotype and all loci will have the same impact on the phenotype. For Mode `A`, the expected number of entries is the number of dimensions times the number of T3 loci. For example

```
--Loci T3 2 --T3_pheno 3 A 0 0 0.5 1.1 0.1 0.2
```

indicates a case where there are 2 T3 loci, the first locus affects only the last dimension of the phenotypic space and the second locus affects all dimensions. If the first locus has value -5 and the second locus has value 10, then the contribution to the phenotype for this specific haplotype (which will be added to the contribution of the other haplotype) is 11, 1, -0.5. The phenotypic value along the ith, Z_i is therefore

$$Z_i = \sum_{j}^{L} loc_{i,j}\left(A_{1,j} + A_{2,j}\right),$$

where L is the number of loci, $loc_{j,I}$ is the effect of the jth locus on the ith phenotypic axis and $A_{1,j}$ and $A_{2,j}$ are the allelic values at the jth locus of the alleles on the first and second haplotype, respectively. Because `--T3_pheno` is a habitat-specific variable, one can model phenotypic plasticity (but not the evolution of the reaction norm in the current). Here is a simple example involving a plastic response

```
--Loci T3 2 --T3_pheno 1 @H0 A -0.5 -0.5 @H1 A 0.5 0.5
```

To match the phenotype to a fitness (fitness landscape), use the following option

```
--T3_fit @S0 selectionMode @H0 entryMode mean
gradient/omega @H1 ... @S1
```

Selection on T3 phenotypes

This option is species- and habitat- specific and is also known as `--T3_FitnessLandscape`. Again, the input is a little unusual, hence the inclusion of `@S` and `@H` in the presentation of the option. The “selectionMode” can be `simple` or `gauss`. If “selectionMode” is `simple` then the fitness component of a given phenotypic axis is calculated as a linear regression with 'mean' and 'gradient' (or slope) given after the EntryMode. Let *D* be the number of dimensions, z_i be the phenotypic value along the ith axis, $z_{opt,i}$ be the optimal phenotype and g_i be the gradient (or slope) along this same axis, then the fitness is defined by

$$W = \prod_{i}^{D} 1 - g_i|z_i - z_{opt,i}|,$$

where |*x*| means absolute value of *x*. If for any dimension *i* the quantity $1 - g_i|z_i - z_{opt,i}|$ is zero or negative, then of course, the fitness is set to 0. If “SelectionMode” is `gauss`, then it expects and optimum (mean; $z_{opt,i}$) and selection strength omega. Fitness is then given as

$$W = \prod_{i}^{D} exp\left[-\frac{\left(z_i - z_{opt,i}\right)^2}{omega}\right]$$

The two possible “entryMode” are `A` and `unif`. For `unif`, two values are expected (`unif mean gradient` or `unif mean omega`). For `A`, 2D values are expected (e.g. `A mean`$_1$ `gradient`$_1$ `mean`$_2$ `gradient`$_2$ or `A mean`$_1$ `omega`$_1$ `mean`$_2$ `omega`$_2$).

8.5 T4

T4 loci are necessarily neutral.

8.6 T5

```
--T5_fit mode float(s)
```

Selection on T5 loci

This option is species- and habitat- specific and is aka --T5_FitnessEffects. The modes for selection on T5 loci are very similar to those on T1 loci. The possible modes are `A`, `domA`, `cstH`, `unif`, `multfitA` (aka `MultiplicityA`), `multfitUnif` (aka `MultiplicityUnif`), `multfitGamma` (aka `MultiplicityGamma`). The only difference with T1 loci is that if you do not want to take advantage of the multfit assumption, then you can specify only two fitness effects per locus, the fitness effect of the heterozygote individual and of the double mutant homozygote (in this order). The fitness of the homozygote wild type is always assumed to be 1.0 for T5 loci.

Just for practice, here is an example of a somewhat complex simulation

```
--PN @G0 1 @G5e3 10
--H @G0 unif 0 @G5e3 A rep 0 8 1 1
--m @G0 OnePatch @G5e3 island 0.01
--N @G0 A 500 @G5e3 unif 50
--L T5 200 T1 10 T5 200
--T1_fit @H0 multfitUnif 1 @H1 multfitUnif 0.98
--T5_fit unif 0.99 0
--T5_mu unif 1e-6
--T1_mu unif 1e-6
--r cM unif 1e-6
--nbGens 15e3
```

Here, we ask for one patch for the first 5,000 generations and 10 patches in an island model afterward up until the end of the simulation at generation 15,000. The total carrying capacity remains at 500 during the entire simulation, as the carrying capacity of the single patch of the first 5,000 generations are set at 500 and the carrying capacity of each of the 10 patches from generation 5,000 are set at 50. The genetic map is made of 10 T1 loci surrounded on each side by 200 T5 loci. Mutation rate is 10^{-6} over all T1 and T5 loci and the recombination rate between adjacent loci is uniform at 10^{-6}. Throughout the entire simulation, T5 loci are always lethal in the double homozygote state and have a selection coefficient of 0.01 in the heterozygote state (and 1 in the double homozygote non-mutated state). The selection on T1 loci vary with the habitat though. In habitat 0, there is no selection at all on T1 loci. In habitat 1, all T1 loci are under selection with fitnesses 1, 0.98 and 0.98^2 at all three genotypes. During the first 5,000 generations, the single patch is in habitat 0 (no selection on T1 loci). For the remaining 10,000 generations, the first 8 patches are in habitat 0 (no selection on T1 loci) and the remaining 2 patches are in habitat 1 (selection on T1 loci). The simulation runs in 7 seconds on my machine.

9 Demography and species ecology

Here I do not mean to talk much about the basic options `--N` and `--m` (see section "Launching basic simulations" for their uses). Instead, I want to talk about how change in patch sizes is modelled and how fecundity and selection, species interaction, and migration affect patch sizes. Note that I talk about carrying capacity to refer to the absolute maximal number of individuals in a patch and I talk about patch size to refer to the number of individuals in a patch (which can differ from the carrying capacity on user's demand).

SimBit assumes non-overlapping generations (although different species can have different generation times) and assumes discrete patches (although patches can be made arbitrarily small, essentially mimicking continuous space). Outside of these two assumptions, SimBit can simulate very diverse types of demographies. SimBit can simulate any number of patches with any migration matrix (see option `--m` in section "Launching basic simulations"), carrying capacity (see option `--N` in section

"Launching basic simulations"), variation of the patch size from the carrying capacity based on realized fecundity with exponential or logistic growth model (the growth model can be set for each patch independently; see more on that below). Each patch can be initialized at the desired size, and all of the above parameters can vary over time.

Dispersal can happen at the gametic or at the zygotic phase and may be a function of the patch mean fitness (hard vs soft selection). This is modified with the following option

Hard or soft selection

```
--DispWeightByFitness bool
```

This option is species-specific. If the migration rate is weighted by fitness, it would simulate a case of hard selection, otherwise it would be a case of soft selection. Note that when the fecundity differs from -1 (see below), then dispersal is necessarily weighted by fitness. Two entries are possible `f` (or 0 or `false, FALSE` or `False`) and `t` (or 1 or `true, TRUE` or `True`). `false` is default and means soft selection and `true` means hard selection. Choosing `false` (the default) might make the simulation a little bit faster especially for simulations with lots of patches.

Gametic or Zygotic dispersal

```
--gameteDispersal bool
```

This option is species-specific. If the option is set to `true`, then gametes disperse and the offspring will leave wherever the gametes meet. If set to `false`, then the offspring migrate. Concretely, if gamete disperse, the two parents can be from different patch. If offspring disperse, then the two parents must be from the same patch. Because offspring dispersal is computationally slightly faster (although often negligibly so) and is a standard in simulations, the default is `true`.

Fecundity

```
--fec float
```

This option is species-specific and is aka `--fecundityForFitnessOfOne`. By default, this number is set to -1. In such case, the patch size is always at carrying capacity (it is like infinite fecundity). You could set the fecundity to an arbitrarily large value to obtain the same effect as -1 but that would (slightly) slow down the simulation. The fecundity is understood as a per individual measure. Take note that when using males and females, only the fecundity of females will affect the number of offspring produced (as long as there is at least one male in the patch). If we assume all fitnesses are at 1, then if we have hermaphrodites a fecundity of at least 1 will be necessary to replenish the entire patch of individuals (there can be stochastic variation that will cause on average a decrease in the patch size; see `--stochasticGrowth`. If you use a males-and-females mating system with a sex ratio of say, 0.75 (3 male for 1 female; see `--matingSystem`), then you will need a fecundity of at least 4 to have subsistence.

SimBit can simulate realistic changes in population in response to patch mean fitnesses. Let's denote at time *t* the expected number of offspring of a species *s* produced in patch p as $\overline{P_{t,s,p}}$. Let's also denote the patch growth rate $r_{t,s,p} = f \sum w_i$ as the product of *f*, the theoretical maximum fecundity of an individual having a fitness of 1.0 (set by the user), and $\sum w_i$, the sum of finesses in this patch. If the user allows the patch size to vary from the carrying capacity of this species and that at time *t*, in patch *p*, for species *s*, the carrying capacity is set to $K_{t,s,p}$ then the expected number of offspring produced is $\overline{P_{t,s,p}} = rN_{t,s,p}$ for the exponential model and $\overline{P_{t,s,p}} = N_{t,s,p} + rN_{t,s,p}\left(1 - \frac{N_{t,s,p}}{K_{t,s,p}}\right)$ for the logistic model, where $N_{t,s,p}$ is the size of the patch *p* of species *s* at time *t*. The actual number of offspring produced, $P_{t,s,p}$ can then either be set deterministically $\left(P_{t,s,p} = \overline{P_{t,s,p}}\right)$ or stochastically $\left(P_{t,s,p} = Poisson(\overline{P_{t,s,p}})\right)$. With more than one patch, these offspring produced are then spread out through migration. With a single patch (or in absence of immigration and emigration for the patch p), $N_{t+1,s,p}$ is simply set to $P_{t,s,p}$.

Into the above framework, we can add the fact that different species can affect each other through their ecological relationships. This can be achieved through a "competition matrix" that implements a Lotka-Volterra model of competition and/or through an "interaction matrix" that implements a consumer-resource model (or predator-prey model) with a linear rate of resource consumption (introduction to these models in Otto & Day, 2007; discrete-time example of a predator-prey model in Çelik & Duman, 2009). Let $\alpha_{i,s}$ be an element of the "competition matrix" describing the competitive effect of species i on focal species *s*. The expected number of offspring produced is then given by $\overline{P_{t,s,p}} = N_{t,s,p} + rN_{t,s,p}\left(1 - \frac{\sum_i \alpha_{i,s} N_{t,i,p}}{K_{t,s,p}}\right)$. Note that competitive effects can only be set on species and on patches having logistic growth. Let $\beta_{i,s}$ be an element of the "interaction matrix" describing the effect of species i on species s. The interaction effect is added to the expected number of offspring produced $\overline{P'_{t,s,p}} = \overline{P_{t,s,p}} + \sum_i \beta_{i,s}$. In this last equation, I assumed that all effects $\beta_{i,s}$ are independent of the patch sizes of both the causal and recipient species but in practice a user can specify for each $\beta_{i,s}$ whether the effect should be multiplied by the causal species patch size ($N_{t,i,p}$), by the recipient species patch size ($N_{t,s,p}$) or by both. SimBit enforces that all the diagonal values $\alpha_{s,s} = 1.0$ and that all the diagonal values $\beta_{s,s} = 0.0$ by enforcing the user to input the keyword `self` (see below).

The growth model can be independently for each patch set with the option

```
--popGrowthModel mode values
```

Growth model

This option is species-specific. The two possible modes are `unif` and `A`. The value(s) accepted are either the keyword "logistic" (aka -2), the keyword "exponential" (aka -1) or any positive integer value. A positive integer value means that you want a logistic growth but you want to set the carrying capacity for this growth calculation to some other value than the upper limit set by option `--N`. This is a neat way to allow more realistic demographics such as overshooting of the carrying capacity for example. Note that the patch size can never be greater than what is set with option `--N`. By default the growth rate is logistic. This growth rate only matters if the fecundity (set in option `--fec`) differs from -1.

```
--stochasticGrowth bool
```

Stochastic growth

If set to `true` (or other equivalent such as `1`, `t`, `T`, ...), then the number of individuals in the next patch is drawn from a Poisson distribution as explained above. Be aware that in absence of stochasticity, systematic rounding of the expected number of offspring can lead to a systematic bias in growth rate, particularly important when patch are of small sizes. By default, it is set to `false`.

In above equations, I specified that the realised fecundity is a function of fitness. It is however sometimes handy to have this realized fecundity independent of fitness (for example to ensure that simulations with different selection scenarios have similar growth rates). For this, use the following option.

```
--fecundityDependentOfFitness bool
```

By default, this option is set to `true.`

```
--eco interaction <matrix> competition <matrix>
```

Species ecology

This option is aka `--speciesEcologicalRelationships`. The terms `interaction` and `competition` are keywords that precede the interaction matrix and the competition matrix, respectively. The ordering of the elements of the interaction and competition matrices makes sense by considering the ordering of the species as entered in option `--species` (aka `--S`). For the competition matrix SimBit expects S^2 $\alpha_{i,j}$, where S is the number of species. For the effect of a species on itself (that is all the diagonal $\alpha_{i,j}$), SimBit expects the keyword `self`. Each element of the interaction matrix is made of two entries; The first entry is the "type" (either `A`, `B`, `C`, `D` or `0`) and the second entry is the $\beta_{i,j}$ value. Just like for the competition matrix, for the effect of a species on itself, SimBit expects the keyword `self` (no letter, no $\beta_{i,j}$, just write `self`). The "type" indicates whether the interaction effect must be multiplied by the causal species patch size $N_{t,i,p}$ (type `B`), by the recipient species patch size ($N_{t,s,p}$) (type `C`), by both (type `D`) or by none (type `A`). The type `0` (the number zero, not the letter o) means no interaction (which is the default). If you want to use the default matrix, just enter `default` as matrix description. The default input is therefore `interaction default competition default`, which would lead to simulate different species that are completely independent.

When the number of patches increase or the patches increase in size, when you use a fecundity of -1, SimBit need to ensure that patch is at carrying capacity. SimBit will therefore simply copy individuals over from the patch that has the highest forward migration rate (and is not empty) toward the considered patch. This is somewhat unrealistic. When demographic details is of concern and when changing number of patches and patch size, it is recommended to use.

There is also an option that simplify the simulation of bottlenecks or of various partial extinction events by just killing a pre-defined number of individuals in the population. Of course, this option only work if the patch size can vary from the patch carrying capacity (that is if the fecundity is different from -1).

Killing individuals

```
--killIndividuals kill <howManyToKillInfo> patch
<patch_index> at <generations> kill <howManyToKillInfo> …
```

The option is a list of killing information, each list starting with the keyword `kill`. The `howManyToKillInfo` is either a number indicated the number of individuals to kill or it is they keyword `allbut` followed by a number, specifying the number of individuals not to kill. It is following by the keyword `patch` and the patch index where those individuals should be killed. And then followed by the keyword at and a list of generations at which the killing must happen. For example, you might want to reduce the population of patch index 6 (zero based counting as usual) to 10 individuals every 100 generations until generation 500 where you want to kill all individuals in the patch. In parallel, we also want to kill 5 individuals from patch index 0 every odd generation from generation 125 to 195. For this, you can do

```
--killIndividuals kill allbut 10 patch 6 at 100 200 300 400
kill allbut 0 patch 6 at 500 kill 5 patch 0 seqInt 125 195 2
```

Of course, if there are fewer individuals in the patch than individuals that must be left in the patch, then SimBit just does not kill anyone. Simillarly, if there are fewer (or an equal number of) individuals than individuals to kill, then SimBit will just kill everyone and stop there.

9.1 Calculation of the backward migration matrix

We first compute the number of offsprings produced in each patch given their mean fitness (if fecundity is affected by fitness; see `--fecundityDependentOfFitness`), the species ecologies, the fecundity and the population growth model. Details calculations are explained above. Then, these expected number of individuals are distributed among the patches following the forward migration dispersal scenario indicated at option `—m` and whether dispersal should be weighted by by patch mean fitness (see `--DispWeightByFitness`). From these, we can compute the backward migration matrix.

When the fecundity differs from -1, when the dispersal is weighted by mean fitness or when there are interacting species which fecundity differs from -1, then the backward migration matrix need to be computed at every generation. Otherwise, the backward migration matrix is only computed at every temporal change.

10 Mating system

Cloning

```
--cloningRate float
```

This is a species-specific option. It sets the proportion of reproduction happening through cloning. Note while cloning, mutations are still happening so that offspring might not be exactly a clone of its parent.

Selfing

```
--selfingRate float
```

It sets the proportion of reproduction happening through selfing. When using a cloning rate different from zero, the selfing rate is the selfing rate for offspring that are not produced via cloning. A selfing rate of -1.0 (default value) means that selfing rate occurs just like it does in a Wright-Fisher population, that is at frequency 1/N.

Note that cloningRate has precedence over selfingRate. This means that if you use both options, then the selfing rate will be conditional on no cloning happened. For example, if you set the cloningRate to 1, then no selfing (or any other sort of sexual reproduction) will ever occur. If you set the cloningRate to 0.5 and the selfing rate to 0.1, then 50% of the offspring will be made through cloning, 5% through selfing (because 10% of 50% is 5%) and the other 45% through normal reproduction.

Mating System

```
--matingSystem system (sexRatio)
```

There are two possible mating systems; `h` (or `H`) for hermaphrodites, and `fm` (or `mf`, `FM`, `MF`) for males and females. Hermaphrodite is the default setting. If you specify males and females, you will then need to specify the sex ratio (the proportion of males) in the population. For example

```
--matingSystem fm 0.5
```

sets all species to a males and females mating system with an even sex ratio.

It is also possible to vary generation time between species. This is achieved with

Sub-generations

```
--nbSubGenerations int
```

This is a species-specific option and is aka `--nbSubGens`. A "SubGeneration" is a generation within a generation. This allows you to simulate a species that has, say 4 generations every time other species have one generation. The number of "SubGenerations" per generation must be an integer. For example

```
--nbSubGenerations @S0 1 @S1 2
```

would cause species 1 to have two generations for every generation of species 0. It would make little sense to input something like

```
--nbSubGenerations @S0 3 @S1 6
```

or

```
--nbSubGenerations 3
```

Instead, just triple the number of generations to option `--nbGenerations`. By default, of course, the number of sub-generations per generation is set to 1.

11 Defining individual types

As a user, you can define what I call “individual types”. An individual type is an abstract individual for which you have fully specified its genome. You can then use those individual types to initialize the population (with option `--individualInitialization`; see section "Initial Population" below) or to reset the genetics of the population during run time (with option `--resetGenetics`; see section "Reset genetics" below).

```
--individualTypes ind <individualName> haplo0 <haplotype
description> haplo1 <haplotype description> ind
<individualName> ...
```

Individual types

This option is species-specific and is aka `--indTypes`. Each individual description starts with the keyword `ind`. It is then followed the name of this individual type by the keyword `haplo0` with the haplotype description and the keyword `haplo1` and its haplotype description. It is also possible to do `ind bothHaplo <haplotype description>` if both haplotypes are to be identical (perfect homozygosity). As an example, the haplotype description is `T1 0 0 0 0 1` indicates that the first 4 loci of T1 loci carry the wildtype allele while the last locus carries the mutated allele. Let’s say we want to define an individual type named “wildtype” and another named “mutant” that are fixed for the 0 and the 1 allele, respectively. We would do

```
--L T1 50 --individualTypes ind wildType bothHaplo T1 rep
0 50 ind mutant bothHaplo T1 rep 1 50
```

Note that you cannot do `bothHaplo T1 unif 1` but need to use `rep` instead if you do not want to write the number 1 50 times. You can also use the keyword `empty` to specify that the haplotype must be empty from any mutation. For example,

```
--L T1 50 --individualTypes ind wildType empty ind mutant
bothHaplo T1 rep 1 50
```

Haplotype description must be made for each type of locus that you asked for with option `--L` (`--Loci`). For example, you asked for the types T1, T3 and T5 with, then you can simply do `—T1 <T1 loci description> T3 <T3 loci description>`. Note that the ordering does not matter (`T5 <T5 loci description> T1 <T1 loci description> T3 <T3 loci description>` is also valid). Table 3 explains how to provide a description for each type of locus.

Table 3: Input format for individual types

Locus type	Entry	Example
T1	Indicate the value of each locus	`T1 0 1 0`
T2	Indicate the number of mutation in each T2 block	`T2 0 0 12 2`
T3	Indicate the value of each QTL	`T3 0 0 5`
T4	Indicate the T4ID, that is the positions of the haplotype in the coalescent tree. This is unlikely to ever be of use and might be considered for advanced users only.	`T4 130`
T5	Indicate the position of each mutation	`T5 0 23`

Individual types can then be redefined during run time based on real haplotypes that have evolved. This allows to sample haplotypes at predetermined generations and re-insert them later in the population. One among many uses for this is to copy some individual from one patch and insert them into another patch. In other words, it allows to force migrations of an exact number of individuals. Another use for it, would be to compare survival probabilities between a population that have evolved longer than another that would have been “frozen” in the states of individual types. To redefine individual types, use

Redefine individual types at run time

```
--redefIndTypes redef <name> <haploInfo> at <generation>
basedOn <patch> <ind> <haploInfo> redef <name> …
```

What comes before the keyword basedOn concern what haplotype of what individual type must be redefined and at what generation. What comes after basedOn concerns which currently existing individuals / haplotypes in the population the redefined haplotype(s) will copied from.

More precisely, name is the name an already defined individual types. haploInfo is either haplo0, haplo1 or bothHaplo. generation is the generation at which redefinition of haplotype type must be made.

Concerning the current haplotype(s) to copy, patch and ind are the patch index and individual index, respectively that will be copied to redefine the individual type. The second haploInfo is the information about which haplotypes to copy. Again, it can be haplo0, haplo1 or bothHaplo. If the second haploInfo is bothHaplo, then the first one must necessarily also be bothHaplo because it would makes no sense to copy two haplotype into a single (unless SimBit had a way to recombine haplotypes while redefining them but it does not for the moment). However, if the first haploInfo is bothHaplo and the second one is either haplo0 or haplo1, then the individual type will be redefined as perfectly homozygous.

Let's imagine for example that we want to "freeze" two individuals (that we will call Alice and Bob) at generation 20 and reinsert five of each of these two "frozen" individuals into the population at generation 100. For simplicity, I will assume a single patch.

```
--indTypes Alice bothHaplo empty Bob bothHaplo empty --
redefIndTypes redef Alice bothHaplo at 20 basedOn 0 0
bothHaplo redef Bob bothHaplo at 20 basedOn 0 1 bothHaplo
—resetGenetics eventB 100 0 Alice 5 Bob 5
```

Consider also the following example. We evolve two isolated small patches of 10 individuals for 100,000 generations to create pure lines. We then want to pick one haplotype from each pure line to create an F1 individual and compute its fitness. Different solution exist to do that but I will here show an example where we just reduce the whole population to a single F1 individual in a single patch. For a change, I will here wrote a complete input to SimBit.

```
SimBit --nbGenerations 1e5 --fitness_file test 1e5 --PN
@G0 2 @G1e5 1 --N @G0 unif 10 @G1e5 unif 1 --m isolate --L
T1 1e5 --T1_mu unif 1e-5 --T1_fit multfitUnif 0.9999 --
indTypes ind F1 bothHaplo empty --redefIndTypes redef F1
haplo0 at 99999 basedOn 0 0 haplo0 redef F1 haplo1 at
99999 basedOn 1 0 haplo0 --resetGenetics eventB 1e5 0 F1 1
```

On my machine, the simulation runs in 1.4 seconds.

A few extra notes; an individual types can be redefined an indefinite number of times. An individual type does not need to be first defined as “empty” in order to be redefined (even though my two examples only redefined individual types that are first created as empty). Redefining individual types happened before resetting the genetics (`--resetGenetics`) allowing to copy individuals in one generation and insert them in the same generation.

12 Initial Population

By default, the patch size is initiated at carrying capacity (set by —N; aka `--PatchCapacity`). To set the initial patch size for each patch, use the following option

```
--InitialpatchSize mode ints
```

Initial patch size

The two possible modes are `A` and `unif`. Of course, no initial patch size can be larger than the initial carrying capacity at generation 0. By default, all T2 loci are set to carrying 0 mutations, and all T1 and T5 loci are set to 0.

One can use individual types (defined with option `--individualTypes`; see section "Defining individual types" above) to initialize a population with the following option

```
--individualInitialization patch0 <indTypeName>
<nbIndividuals> <indTypeName> <nbIndividuals> .. patch1
<indTypeName> <nbIndividuals> ...
```

Initialization with individual types

This option is species-specific and is aka `--indIni`. Here, use the keywords `patch0`, `patch1`, ... `patchx` to describe how to initialize each patch. In each patch name, an individual type and the number of individuals of this type to put in this patch. SimBit will ensure that the number of individuals that you put in a patch is equal to the patch size at the beginning of the simulation. (Reminder: if the initial patch size is not specified, then it is set to the carrying capacity).

Alternatively, one can use the following option to initialize T1 loci and T2 loci independently but, while they can be useful, these options are less flexible that using individual types. By default, all T2 loci are set to carrying 0 mutations and all T1and T5 loci are set to 0. It is possible however to indicate the per patch allele frequency with `--T1_ini` (aka `--T1_Initial_AlleleFreqs`) and `--T5_ini` (aka `--T5_Initial_AlleleFreqs`)

```
--T1_ini mode (value)
```

T1 initialization

```
--T5_ini mode (value)
```

T5 initialization

These are an option-specific options. The modes are `AllOnes`, `AllZeros`, `A` and `Shift`. The parenthesis around "value" above indicates that the input of values depend on the mode. See table 4 for more information. For example,

```
--L T1 2 --PN 3 --T1_ini A 0 0.2 0.5 0.2 1 0.2
```

It will set the allele frequency at the zeroth locus to 0, 0.5 and 1 in the three different patches and the frequency of the locus index one will be set to 0.2 in all patches. For more flexibility in initialization, please use the option `--indIni` (`--individualInitialization`) instead.

Table 4: Input format for --T1_ini and --T5_ini

Mode	Meaning	Example
`AllZeros`	Set all loci to 0.	`AllZeros`
`AllOnes`	Set all loci to 1 (this option is not available for T5 as it is a bad idea for performance reasons).	`AllOnes`
`A`	Specify the per patch and per locus allele frequency. The initialization will be done in a pseudo random manner in order to keep LD low	`A 0 0 0.2 0.2 0 0`

`Shift`	Specify a given patch before which allele frequencies (at all loci) are set to 0 and after which allele frequencies (at all loci) are set to 1	`Shift 12`

Another option is to directly import a population saved in a binary file from a previous simulation. For this use

```
--ReadPopFromBinary file
```

Import population

In order to read a binary file correctly, one must know what it is reading. For this reason, it is essential to specify correctly the number of type of loci, number of patches and number of individuals per patch with the usual `--PatchNumber`, `--N` and `--L` options. Note that in the current version, T4 loci cannot be dumped into a binary file and therefore can be read from it either.

SimBit can also perform a burn in until all T4 loci have coalesced with the following option

```
--burnIn int
```

Burn in until T4 loci coalesce

This option is aka --burnInUntilT4Coal. The integer value that follows specifies how often should SimBit check whether all T4 loci have coalesced. It is typically a value of the order of the population size although it might be strategic to use a bigger number when recombination rate is relatively important. Note that the burn in happens using the parameters at generation 0. As such, at the end of the burnIn, at generation 0, the patch sizes might differ from what is specified with option `--InitialpatchSize` may not

13 Reset genetics

It is often useful to make arbitrary modification to the genetics of a population during runtime. It is common for example to introduce mutations at specific times in order to track its evolution. One might also want to introduce individuals into the populations that come from a fictional, non-simulated infinite population. Such features can be done with the following option.

Reset genetics

```
--resetGenetics <eventType> <eventDescription> <eventType>
<eventDescription>
```

This option is species-specific. There are two types of events (“eventType”) called `eventA` and `eventB`. `eventA` refers to the input of specific mutations. `eventB` refers to the input of individual types (see section “Individual types”).

For `eventA`, the “eventDescription” is structured as

```
eventA generation  TraitType  (typeOfMutationsIfNeeded)
<lociListInformation>  <haplotypesInformation>
<patchAndIndividualInformation>.
```

Directly after the keyword `eventA` comes the generation and then the trait type to be affected in this event (T1, T2, T3 or T5; T4 not accepted). If the trait type is T2 or T3, then SimBit will just reset the designated loci/individuals/patch to zero (set the number of mutation in the T2 locus to zero). If the trait type is T1 or T5, then an extra specification (“typeOfMutations”) must be given to describe the type of mutations. There are three possible types of mutations `setTo0`, `setTo1` and `toggle`.

It is followed by loci list information (“lociListInformation”). The user can either say `allLoci` and all loci will be affected or list the loci with the keyword `lociList`. For example to affect loci indices 0, 4 and 7, indicate `lociList 0 4 7`.

Afterward comes the haplotypes information. It must start with the keyword `haplo` and be followed by either `0`, `1` or `both` to indicate on which set of chromosomes (individuals are diploid as a reminder) the mutation will happen (e.g. `haplo both`)

Finally comes the patch and individual information. It must be formatted as `patch 0 <individuals information> patch 4 <individuals information>`. To affect all individuals within a patch, use the keyword `allInds` and to affect only specific individuals within a patch use the keyword `indsList` (e.g. `indsList 1 2 3 4 5`). For example to affect all individuals of patch 0 and only individuals 10, 20 and 25 of patch 3, you would write `patch 0 allInds patch 3 10 20 25`.

Let's do a full example. Let's assume we would like to simulation a selective sweep. We want only one mutation on the 11th T1 locus (index 10 by zero based counting) of the first (index 0) individual in the third (index 2) patch to happen at generation 100. We could do

```
--resetGenetics eventA 500 T1 setTo1 lociList 11
haplo both patch 2 indsList 0
```

Note that the genetic reset will happen at the end of the specified generation but just before writing the outputs for this generation. So, in the above example, the offspring of the generation 500 (that is the parents of the generation 501) are going to be reciprocally fixed at loci 10, 15 and 20. If the fecundity (`--fec`) is not set to -1, it is possible that the patch size differs from the carrying capacity. If an individual does not exist, SimBit can of course not mutate an inexistent individual. If you want to simulate a single mutation, it is therefore more strategic to use individuals with a low index.

For `eventB`, the input is formatted as

```
eventB generation patch_index <indTypeName> <nbInds>
<indTypeName> <nbInds>…
```

If the patch size differs from the carrying capacity, SimBit will start by adding the individual types in the patch until it reaches carrying capacity, then only will SimBit start to replace currently existing individuals with the individual types. For example, imagine you want to model genetic rescue of a small population. For this, imagine you want to model the immigration of individuals coming from two populations of infinite sizes into our patch index 2. Let's say we want to introduce 5 `migrantTypeOne` and 5 `migrantTypeTwo` at generations 10, 20 and 30. You define individual types `migrantTypeOne` and `migrantTypeTwo` with option `--indTypes` and then you introduce them with

```
--resetGenetics
    eventB 10 2 migrantTypeOne 5 migrantTypeTwo 5
    eventB 20 2 migrantTypeOne 5 migrantTypeTwo 5
    eventB 30 2 migrantTypeOne 5 migrantTypeTwo 5
```

Note that the option `--killIndividuals` allows for arbitrary killing of individuals in the population at pre-determined times. The option could have been presented under this section but was already presented at the end of the section Demography and Species Ecology.

14 Output

```
--GeneralPath path
```

General path

This option is aka `--GP`. indicates the path where all the output will be printed. Other paths relative to outputs are all relative to the "GeneralPath". SimBit does not add a terminal "/" to the path. So, if your path does not end with a "/", then the characters after the last "/" are taken as a prefix of all output files. There is no default for this option, so you have to input at least an empty string if you wish any outputs, otherwise an error will be thrown.

In all of below outputs you have to input a filename that comes after the "GeneralPath". There are two important keywords; `nfn` and `NFN`, which stand for "No File Name". If you indicate `nfn` (or `NFN`) as filename for a specific output, then the file will have a specific name (but only a specific extension and generation-specific information or other specific information). This is handy because it allows the user to give a standard name for files directly in the "GeneralPath". For example

```
--GeneralPath /path/to/directory/firstSimulation --
T1_vcf_file nfn 200 300 400 500 --T1_AlleleFreq_file
nfn 500
```

will create the same output as

```
--GeneralPath /path/to/directory/ --T1_vcf_file
firstSimulation 200 300 400 500 --T1_AlleleFreq_file
firstSimulation 500
```

There are 3 general classes of outputs; the logfile, a binary file of the population, and various user-friendly (tab separated values and VCF) outputs. Each input requires first a file name and then, if applicable, timing of when the output must be produced. Please don’t use spaces in the names of files or directory as this might eventually be misinterpreted. Outputs that are species-specific are automatically produced for each species, and the file name is then preceded by the species name. For all the outputs provided, you can ask for a version of these outputs containing sequencing errors.

```
--sequencingErrorRate rate
```

Sequencing errors

Using this option with a rate different from 0.0 will cause SimBit to produce all outputs as asked (original data without sequencing error) plus an extra set of outputs with simulated sequencing errors. The files with sequencing error will have the string "_sequencingError" added to the file name just before the extension.

14.1 Logfile

The 'logfile' can be used to 1) remember what input has been given to SimBit and 2) to make sure SimBit interpreted the input as we wished although this second usage might not be for beginners.

```
--Logfile filename
```

Logfile

SimBit will automatically add the extension '.log' to your filename. By default, the filename is 'logfile.log', that is the default entry is

```
--Logfile  logfile
```

It is possible to specify the type of logfile we want

```
--LogfileType int
```

Logfile type

Three possible entries are possible; 0,1 and 2. The default value is 0. See table 5 for more information.

Table 5: Input format for logfile type

Entry	Meaning
0	No Logfile is being printed
1	Logfile contains only the arguments that SimBit has received through the command line
2	Logfile contains the arguments that SimBit has received through the command line and all the parameters that have been set for the simulation. Logfile might be very big! All the parameters are written in a code that can simply be copy-pasted into R. This is handy for graphing the parameters and for making sure you got what you want. The directory "GraphParameter" contains an example on how to graph the dispersal matrix and the patch capacity over time.

For all file outputs for which generations at which we need the outputs must be indicated, one can either write out every generation or use the keyword `fromtoby` (or `fromToBy` or `FromToBy`) to indicate a sequence. For examples

```
--nbGens 100 --T1_vcf_file filename 10 20 30 40 50 60
70 80 90 100
```

is equivalent to

```
--nbGens 100 --T1\_vcf\_file filename seqInt 10 100 10
```

and is also equivalent to

```
--nbGens 100 --T1_vcf_file filename fromtoby 10 100 10
```

and to

```
--nbGens 100 --T1\_vcf\_file filename fromtoby 10 end
10
```

`fromtoby` is therefore very similar to `seqInt`. The two differences are 1) `fromtoby` only works for outputs (options ending in `_file`) and 2) `fromtoby` can accept the keyword `end` to specify the last generation of the simulation. In all cases, the first value is the start of the sequence, (from), the second value is the end of the sequence (to) and the third value is the increment (by). One can also mix up these methods. For example

```
--T1_vcf_file filename 1 10 20 30 40 50 60 70 80 90
100 107 200 300 400 500 550 600 650 700 750 800 850
900 950 1000
```

can be rewritten

```
--T1_vcf_file filename 1 fromtoby 10 100 10 107
fromtoby 200 500 100 fromtoby 550 1000 50
```

14.2 Export population to a binary file

It may be of interest to export the population in a binary file. The main reason why you would want to do that would be to reuse the saved population as starting population for another simulation (with the option `--readPopFromBinary`). It can also be useful to recalculate statistics about this population later via SimBit by simulating 0 generation or (for advanced users) by directly treating the data yourself from a format that takes very little storage. To export the population in a binary file use the following option

```
--SaveBinary_file file generations
```

Binary file

such as for example

```
--S HomoSapiens PanTroglodytes --SavePopBinary MyBin
50 100 500
```

save the files "HomoSapiens_MyBin_G50" and "PanTroglodytes_MyBin_G50", "HomoSapiens_MyBin_G100" and "PanTroglodytes_MyBin_G100", "HomoSapiens_MyBin_G500" and "PanTroglodytes_MyBin_G500" at the generations 50, 100 and 500, respectively. At each generation it will also save a binary file with the seed information. Its name is just like the above except species name is replaced by "seed". This also mean by the way that the "seed" cannot be used as a species name (SimBit will send an error message if you try).

14.3 User friendly outputs

Outputs that are specific to certain type of loci have it clearly indicated in their name (e.g. `--T1_FST_file` or `--T3_MeanVar_file`). Just like before the output start with the name of the file (or the keywords `nfn` or `NFN`), followed by the generations at which the output is requested. Some options can also take a `subset` keyword that will allow the user to indicate the set of loci over which the output should be computed. For example:

```
--T1_FST_file evenLoci 50 100 subset 0 2 4 6 8 10 12
14
```

It will output data at generations 50 and 100 for loci 0 2 4 6 8 10 12 and 14. For these outputs that can take a `subset` keyword, you can give the option several times to ask for different subset. For example

```
--T1_FST_file evenLoci 50 100 subset 0 2 4 6 8 10 12
14 --T1_FST_file oddLoci 50 100 subset 1 3 5 7 9 11 13
15
```

When using the same option several times, it is the user's responsibility to give different names to these files otherwise the data will be confounded in the same file in ways that can be confusing. The option `--T1_fitness_file` does not accept the `subset` keyword because a more advanced subsetting solution exists already via the option `fitnessSubsetLoci_file`.

14.3.1 Outputs for T1 loci

```
--T1_LargeOutput_file file generations
```

Outputs: T1 Large

This outputs the complete genotype of every individual in a TSV (Tab separated Values). This can be a very large file (hence the silly name). The extension ".T1LO" is added to the filename. For example,

```
--T1_LargeOutput_file mySimulation 100 200 300 400 500
```

will print the large outputs for generations 100, 200, 300, 400 and 500.

```
--T1_vcf_file file generations
```

Outputs: T1 VCF

This outputs a VCF (Variant Call Format) file. This can be very handy for usage with the command line vcftools. With the help of the software PGDspider, one can reach almost any commonly used file format from this VCF file. Loci that are fixed for the mutatnt allele are also included. The extension .T1vcf is added to the filename. This option accepts the `subset` keyword.

```
--T1_AlleleFreq_file file generations
```

Outputs: T1 Allele freqs

This outputs the allele frequency at each locus for each patch. The extension .AlleleFreq is added to the filename. This option accepts the `subset` keyword.

```
--T1_FST_file file generations
```

Outputs: F_{ST}

This outputs F_{ST} measures (Weir and Cockerham, as well as Nei estimates for both averaging over all loci and as the ratio of the averages of numerator and denominator over all loci). This output file has not really been tested yet. Please consider it with precaution. The extension .T1FST is added to the filename. More info can be given via `--T1_FST_info`. This option accepts the `subset` keyword.

```
--T1_FST_info nbPatchesToConsider
```

Outputs: F_{ST} info

Gives info about what patch comparisons must be performed for F_{ST} calculations. It is easier to explain with examples. If you input `--T1_FST_info allInteractions 2`, then SimBit will output all pairwise F_{ST} measures. If you input `--T1_FST_info allInteractions 3`, then SimBit will output F_{ST} measures for all possible triplets of patches.

```
--T1_MeanLD_file filename generations
```

Outputs: T1 Mean Linkage Disequilibrium

It outputs the within and among chromosomes average linkage disequilibrium per patch. The extension .MeanLD is added to the filename. This option accepts the `subset` keyword.

```
--T1_LongestRun_file filename generations
```

Outputs: T1 Longest Run

It outputs the longest run (or longest consecutive series) of 0 or longest run of 1 for each haplotype (of each individual in each patch). The extension .LR is added to the filename. This option accepts the `subset` keyword.

```
--T1_HybridIndex_file filename generations
```

Outputs: T1 Hybrid Index

It outputs the hybrid index of each individual. Here, I call the hybrid index of an individual, the fraction of T1 loci of this individual that carry the "1" allele. This is a helpful statistic when used alongside `--T1_ini`. It allows the user to specify specific patches where all individuals are fixed for the "0" allele and other patches fix for the "1" allele and see how they interbreed through time. With selection against heterozygotes, you can have some barrier to gene flow. This option accepts the `subset` keyword.

Outputs: T1 Average minimal number recombination

```
--T1_ExpectiMinRec_file filename generations
```

This outputs the average (averaged among all haplotypes within each patch) number of times a run of zero is stopped by a run of 1 or vice versa. For example the haplotype '1111011111100000000' has 3 such events. The extension .EMR is added to the filename. This option accepts the `subset` keyword.

```
--T1_ExpectiMinRec_file filename generations
```

14.3.2 Outputs for T2 loci

```
--T2_LargeOutput_file filename generations
```

Outputs: T2 Large

This outputs all genotypes of all individuals. This can be a very large file. The extension .T2LO is added to the filename.

14.3.3 Outputs for T3 loci

```
--T3_LargeOutput_file filename generations
```

Outputs: T3 Large

This outputs all genotypes of all individuals (but not the phenotypes). This can be a very large file. The extension .T3LO is added to the filename.

```
--T3_MeanVar_file filename generations
```

Outputs: T3 MeanVar

This outputs the mean and variance in phenotype (along each dimension of the phenotypic space) per patch.

14.3.4 Outputs for T4 loci

For T4 loci, there are the following two types of output. They work exactly like the T1 outputs with the same name except that they can't take the `subset` keyword.

```
--T4_LargeOutput_file filename generations
```

Outputs: T4 Large

```
--T4_vcf_file filename generations
```

Outputs: T4 VCF

The option `--T4_SNPfreq_file` output the frequency of every SNP (and loci that are fixed for the mutant allele) in the population, appending all generations together in one file.

```
--T4_vcf_file filename generations
```

The option `--T4_printTree` is currently unavailable. It meant to output the entire Ancestral Recombination Coalescence Tree for the T4 loci. Note that the tree is being outputted each time the current states are being computed, which is each time you asked for some T4 outputs and each time the average number of nodes per haplotype overpass the limit set by the option `--T4_maxAverageNbNodesPerHaplotype`. Interpreting several trees might be tricky. Hence, if you are not asking for any specific T4 output before the last generation, you might want to set `--`

T4_maxAverageNbNodesPerHaplotype to a very large number (like --T4_maxAverageNbNodesPerHaplotype 1e9 for example) to make sure the current states will never be computed before the very end of the simulation. That might affect performance though if the recombination rate is relatively large.

```
--T4_printTree filename
```

Outputs: T4 Tree

For every output that involve placing mutations on the coalescent tree, SimBit can perform the placement of mutations a number of times and create one output file everytime. For this use the following option

```
--T4_nbMutationPlacingsPerOutput int
```

Placing mutations on the tree

By default, this option is set to 1. For the special case, where this option is set to 1, SimBit can ensures that mutations placed at an early generation will be remembered for outputs at later generations in order to have congruency for mutation placing at different generations. For this use the following option

```
--T4_respectPreviousMutationsOutput bool
```

Congruency when placing mutations at different generations

By default, this option is set to false when --T4_nbMutationPlacingsPerOutput is set to greater than 1 or when you use painted haploytpes outputs. It is by default set to false otherwise. Note that in the current version, SimBit cannot “respect previous mutations” when you use haplotype painting.

A user might also want to make sense of the coalescent tree without placing mutations on it. SimBit allows painting haplotypes at a given generations and then, at a later generation, observe (for a desired subset of haplotypes in a subset of patches) how different segments have segregated throughout the population. SimBit can either output the “color” (by color, I refer to the ID of an ancestral haplotype at the generation the haplotypes were painted) for every segments in the population with the option --T4_paintedHaplo_file or a summary of the diversity of segments throughout the sample with option --T4_paintedHaploSegmentDiversity_file.

Painted haplotypes

```
--T4_paintedHaplo_file generations <generationInfo> patch
ints nbHaplos ints generations <generationInfo> patch ints
nbHaplos ints …
```

Summary of painted haplotypes

```
--T4_paintedHaploSegmentDiversity_file generations
<generationInfo> patch ints nbHaplos ints generations
<generationInfo> patch ints nbHaplos ints …
```

Both options follow the same input semantic. It starts with the keyword "`generations`" and is followed by the information about at what generation the haplotypes need to be painted and at what generation at they observed. The generation information is formatted as `paintedGeneration-observerGeneration.` For example, `generations 20-100` asks for haplotypes to be painted at generation 20 and observed at generation 100. It is followed by the keyword `patch` and the indices of the patches being sampled and finally by the keyword `nbHaplos` and the number of haplotypes being sampled in each patch listed above. Of course, the number of integer after `nbHaplos` must match the number of integers after `patch`. Then, we can ask for other painting events and other generations at which we make observations by adding a new `generations … patch … nbHaplos …` line.

Note that if you perform a burn in until all T4 loci coalesce (see option –burnIn), it is possible to specify that you want to paint the generation at which the burnIn started. For that you can use the keyword "anc", such as for example `generation anc-100`.

As an example,

```
--T4_paintedHaploSegmentDiversity_file generations 100-120
patch 0 1 2 nbHaplos 25 25 25 generations 100-130 patch 0 1
2 nbHaplos 25 25 25 generations 110-120 patch 1 nbHaplos
100
```

Asks for painting haplotypes at generations 100 and observing them at generations 120 and 130 in patches 0, 1 and 3 and sampling 25 haplotypes each time. It also paints haplotypes at generation 110 and observe them at generation 120 in patch 1 only sampling 100 haplotypes.

Note that when you let the patch size differ from the carrying capacity (option --fec), you might not know whether you have enough haplotypes in the patch considered to sample when the time comes. SimBit will just sample as many as possible and this will be clearly explicit in the output file.

By default, SimBit throws an error at initialization of the simulation if, for a given patch, you ask to sample more haplotypes than twice the carrying capacity of that patch. However, sometimes it is pleasant to just ignore the commands that are non-sense (because it may make the building of input commands easier). In such case, you might want to use the following option

Painted haplotypes: ignore some some inputs that makes no sense instead of throwing an error

```
--T4_paintedHaplo_ignorePatchSizeSecurityChecks bool
```

14.3.5 Outputs for T5 loci

```
--T5_vcf_file filename generations
```

This outputs a VCF file for each patch (appending the patch number in the name of the file). Loci that are fixed for the mutant allele are also included.

```
--T5_AlleleFreq_file file generations
```

This outputs allele frequencies at each locus for each patch

```
--T1_LargeOutput_file file generations
```

This outputs the state of each locus in each haplotype. Watch out, the file can be huge!

14.3.6 Other Outputs

```
--Tx_vcf_file filename generations
```

Just like the option `--T4_SNPfreq_file`, `--Tx_SNPfreq_file` output the frequency of every SNP (and loci that are fixed for the mutant allele) in the population but not only for T4 loci but for all types of loci confounded (except T3 loci where the concept does not apply). For T2 locus, it outputs the frequency of alleles that differ from 0. The locus position indicated are the locus index irrespective of the type of locus. Note that this option is also affected by options `--T4_nbMutationPlacingsPerOutput` and `--T4_respectPreviousMutationsOutput` even if it is not listed in category T4.

```
--fitness_file filename generations
```

Outputs: Fitnesses

it outputs the fitness of every individual in the population. The extension .fit is added to the file name.

```
--fitnessSubsetLoci_file filename generations @S0 LociSet
T1 ints T2 ints T3 ints T1epistasis ints LociSet ints ...
@S1 LociSet T2 ints ...
```

Outputs: Fitnesses subset of loci

This option is very similar to `--fitness_file` except that it allows outputting fitness for specific subset of the genome. The option is hence species-specific and allows definition of an infinite number of subset of the genome (called `LociSet`) that a user may want. The argument comes like other output arguments with the filename followed by the time at which output must be produced. What follows the time indication is a little bit unusual. First, you have the species-specific markers. Then, you can create an indefinite number of sets of loci from which fitness will be computed. Each set starts with the keyword `LociSet`. After this keyword, you specify what types of loci you are willing to consider and their associated indices. The four possible types are T1 T2 T3 and T1epistasis. You can specify several types per set if you want. For example

```
--fitnessSubsetLoci_file myFile LociSet T1epistasis 0 3 4 T1 0 5 10 T3 0 1 2 3
LociSet T1epistasis 0 1 2 3 4
```

will lead SimBit to consider two sets of loci. One for which it will compute normal selection on the T1 0 5 and 10 as well as epistatic selection on T1 loci 0 3 and 4 and selection on T3 loci 0, 1, 2 and 3. The second set only computes the epistatic selection on loci 0, 1, 2, 3 and 4. Note that for both `LociSet`, the 4 components of fitness (T1Fitness, T2Fitness, T3Fitness and T1epistasisFitness) are printed. Hence, it would serves no purpose to add a third `LociSet` (`LociSet 1 0 5 10`) as this information is already contained in the first `LociSet`. The example lack any species-specific marker and therefore assumes either a single species or that the same `LociSet` are required for all species.

```
--fitnessStats_file filename generations
```

Outputs: Fitness stats

This outputs the fitness mean and variance per patch. The extension .fitStats is added to the filename.

```
--patchSize_file filename generations
```

Outputs: Patch sizes

This outputs the number of individuals in each patch. The extension .patchSize is added to the filename.

```
--exctinction_file filename generations
```

Outputs: Extinction

This outputs the extinction time (if it applies) for every species.

```
--genealogy_file filename generations
```

Outputs: Genealogy

It outputs the entire genealogy between the two time points indicated (expects only two time points). Because, this represents a lot of data, SimBit does not keep the entire genealogy in the RAM but prints it out in a temporary file that it later merges together into a single file. Because a lot of files are being printed during the simulation, we recommend that you indicate a directory to SimBit, Something like

```
--genealogy_file  familyTree  1000 5000
```

At the end of the simulation, there is a single file left. Each generation gives a line that looks like "G_1005 P0I0_P0I34_P3I102 P0I1_P0I121_P0I97 etc...". "G_1005" indicates the generation (generation 1005), and is followed by tokens with three values separated by "_" such as "P0I0_P0I34_P3I102". The first one is ID for the offspring and the last two are IDs for the two parents. "P0I0_P0I34_P3I102" means that the individual index "0" of patch index "0" is the descendent of a parent from patch index "0" individual index "34" and from a parent from patch index "3" (a migrant) individual index "102". I would not quite call it a user-friendly output, but this format can actually become quite handy especially that it allows matching individuals as identified here with their other attributes (such as their fitness) as given by other output files.

The option `--coalesce` allows SimBit to directly compute the coalescent tree from the genealogy files. In other words, it removes all individuals that did not leave offspring at the last generation sampled for the genealogy.

Outputs: Genealogy

```
--coalesce int
```

It takes a single value which is either 0 (which means don't coalesce) or a positive number. If a positive number is used, then SimBit will remove from the genealogy all ancestors that did not leave any offspring at the last generation sampled. Note that when cloning / selfing rates are high, coalescence happens fast but in presence of sexual reproduction, very few ancestors leave absolutely nothing in the current generation and the option becomes almost pointless.

The positive value chosen matters only for performance reasons. A value of 250 for example, means that SimBit will look back at previous generations (to remove all ancestors that did not leave any offspring) every 250 generations. Because looking back at ancestors is a little bit slow (because the information is kept on the hard drive, not on the RAM), SimBit will run faster if you input a large number. However, keeping a very large genealogy on the hard drive may become problematic and may saturate the hard drive. As such, it may be of interest to remove all ancestors regularly enough so as to free up the storage. A priori, we would suggest that hard drive storage is rarely a limitation, and we would invite our user to use a large enough number. Without having done much testing, I would a priori recommend using a value of about 4N (where N is the total population size) generations.

15 Technical options

Random seed

```
--random_seed int
```

This is aka `--seed`. It specifies the random seed for the simulation. The seed will be printed on the logfile if this info has been demanded. Knowing the random seed allows one to replicate the same simulation exactly which is often very handy. Instead of specifying an integer, you can also use the keyword `binfile` or `f` and give the path to a binary file containing the seed. Such binary files can be produced from other simulations (see section "Outputs"). By default, the random seed is set to the computer current time.

A user can terminate a simulation upon some condition depending on the stat of the population simulated.

```
--killOnDemand <functionToCall> <args>
```

Kill on demand

There is for the moment, only one function available, it is called `isT1LocusFixedAfterGeneration`. The arguments expected are the T1 locus to consider and the generation after which to call the function. The function kills the simulation if the chosen T1 locus is found fixed any time after the generation indicated. Please feel free to let me know if you need another killOnDemand function. An advanced user can also try to modify the function “void KillOnDemand::readUserInput(InputReader& input)” in "KillOnDemande.cpp" and add a new function with the desired name in this same file).

It is sometimes helpful to start a simulation at a generation other than 0. This is typically useful when restarting a simulation (from a binary file) who crashed because of overpassing the wall time limit on a cluster). In such case, you can use `--startAtGeneration`

```
--startAtGeneration int
```

Start at generation

By default, `--startAtGeneration` is set to 0.

```
--Overwrite int
```

Overwrite

Entry values are explained in the table 6. By default Overwrite is set to 2.

Table 6: Input format for --Overwrite

Entry	Meaning
0	Do not overwrite
1	Overwrite even if the logfile already exists but not if the last output files exist
2	Overwrite in any case

Dry run

```
--DryRun int
```

To ask for a DryRun indicate `--DryRun 1`. In such case, SimBit will do a normal initialization of the simulation but will abort just before simulating the first generation.

Print progress

```
--printProgress bool
```

If `true`, SimBit prints on standard output the progress of the simulation (prints the generation, more generations are printed at the beginning of the simulation than later on). By default, it is set to `true`.

16 Performance options

Performance options do not change what is being simulated but only how it is simulated. To the exception of `--swapInLifeCycle`, `--geneticSampling_withWalker`, and `--individualSampling_withWalker`, all options will produce exactly the same outputs given the same random seed.

Fitness map info

```
--FitnessMapInfo mode float
```

This option is species-specific. It is probably the most important performance option. Tweaking this option can eventually make your simulation faster if you use the assumption of "multfit". There are two modes `prob` and `descr`. The default entry cannot be copied but is something along the lines of `prob 0.008`. SimBit sums up the recombination rate between loci until it reaches the probability indicated before creating a new fitness map block. With `descr`, you can specify exactly which block each locus belongs to. It works as `descr <nbLociFirstBlock> <nbLociSecondBlock> <nbLociThirdBlock> ...` The total number of loci must be the total number of loci all types summed. For example,

```
--L T1 7000 --r cM A rep 0 999 -1 rep 0 4999 -1 rep 0 999 --FitnessMapInfo
descr 1000 5000 1000
```

In this example, we are asking for three chromosomes of 1000, 5000 and 1000 loci, respectively. I used `--FitnessMapInfo descr` in order to specify that fitness map block boundaries match with the chromosomal boundaries.

T4 loci are not directly simulated. Instead a coalescent tree is being computed and mutations are added on the coalescent tree. In presence of recombination, every locus can potentially have its own evolutionary history. Instead of representing every lineage, SimBit records an ancestral recombination graph inspired by Keheller et al. (2018). At any time, a given haplotype can be described by a number of nodes in the ARG. As the average number of nodes per haplotypes increase, the simulations get slower. It is in general not too much of an issue, as drift is often sufficient to avoid this average number of nodes per haplotypes reaching a high value. That being said, it can become an issue. For this, SimBit can redefine the ancestral nodes and clear the tree on demand when the average number of nodes per haplotype is too high. The threshold for such action to be taken is given through the option.

```
--T4_maxAverageNbNodesPerHaplotype mode float
```

T4 loci performance tweak

This option species-specific. By default this option is set to 100.

As a reminder, for T5 loci, each haplotype tracks the indices of the loci that have been mutated. When T5 locus, say, index 23 reaches fixation, then every haplotype tracks this locus 23, which is not optimal. Instead SimBit can flip the meaning of having 23. If SimBit flips the meaning for index 23, it means that every haplotype that carries the index 23 are not mutated at this locus while those who do not carry index 23 are mutated. SimBit can do this flipping not only for fixed loci but for any desired frequency greater than 0.5. To set this frequency above which meaning of haplotypes are flipped use the following option.

```
--T5_freqThreshold float
```

T5 loci threshold for flipping meaning

This option species-specific. By default, this frequency is 1.0. To set how often SimBit will check for the loci that have reached a high frequency, use

```
--T5_toggleMutsEveryNGeneration int
```

T5 loci rate meaning flip

This option is species-specific. A value of -1 means "never try to flip meaning". By default, SimBit never tries to flip meaning.

```
--T5_compress bool bool
```

Compress T5

This option is species-specific and is aka `--T5_compressData`. The first bool tells whether the T5ntrl loci must be compressed, and the second bool tells whether the T5sel loci must be compressed. By default, T5sel loci are never compressed, and T5ntrl loci are compressed if the number of T5ntrl loci is lower than 2^{16} - 1 = 65535.

Avoid copying last reproduction of haplotype

```
--swapInLifeCycle bool
```

This option is species-specific. SimBit can either copy each parental haplotype to create each offspring haplotypes. But if a given parental haplotype creates its last offspring haplotype without recombination, then it would be faster to just copy a pointer to this haplotype. The down side of this is that by copying pointers, we reduce memory contiguity and, also, we have to figure out when is the last reproduction event of each haplotype. By default, `--swapInLifeCycle` is set to true only if the total number of loci is greater than 100 and if the total recombination map is shorter than 10cM.

In some simulations, sampling individuals for reproduction can be time consuming. When there is no selection, then sampling is relatively fast as it only requires getting a uniformly distributed random integer value (which is in fact harder that it may sound if you want to have truly unbiased uniformly distributed values). When there is selection, however, the probability of sampling an individual depends upon its fitness. Under such circumstance, SimBit produces a random float number uniformly distributed between 0 and the sum of fitnesses in the patch. Then, do a binary search on an array containing the cumulative sum of fitnesses of the individuals in the patch. An alternative method is the alias method described by Walker (1974). I also implemented the alias method but as with some quick benchmarks, it appears to perform a little worse than the other, by default, SimBit does not use the alias method. If you want to use the alias method, please set the following option to `true`.

Alias method for sampling individuals

```
--individualSampling_withWalker bool
```

A very similar sampling scheme exists for sampling position of mutations and recombination breakpoints. To use the alias method for these sampling, please set the following option to `true`.

Alias method for mutation and recombination positions

```
--geneticSampling_withWalker bool
```

17 R wrapper

The R wrapper is very simple and basic, yet very helpful. First, install and load the package with

```
install.packages(“devtools”)

require(devtools)

install_github("RemiMattheyDoret/SimBitWrapper")

require(SimBitWrapper)
```

The core of this package is a class of object called Input. This class of object eases the creation of input commands to SimBit. On top of that, the package contains a few functions that ease the creation of a grid of parameters, the extraction of parameter values for a given simulation input and also eases the gathering of output data to analyze the outputs. The following two sections explain the Input class and these extra functions, respectively.

17.1 Input

Input is a R6 class of object that help create the input command to a SimBit simulation. When you want to build a simulation input, start by initializing an Input object. You do that with

```
input = Input$new()
```

If you are not used to R6 class and if you are not used to OOP (Object-Oriented Programming), the semantic may appear a little unusual, but don’t worry, it’s easy! `new` is a method of the class `Input` and it is meant to initialize a new object of the class `Input` that, in the above example, we called `input`.

Then you can set each option separately with the `set` method. The `set` method expects a first argument (called `optionName`) that indicates the name of the SimBit option to set (without the starting "`--`"). The second argument is the entry for this option. Here is an example, for a simple simulation

```
input$set("PN", "10")
input$set("m", "island 0.01")
input$set("N", "unif 100")
input$set("L", "T5 1e3")
input$set("T5_fit", "multfitUnif 0.99")
input$set("T5_mu", "unif 1e-7")
input$set("r", "cM unif 1e-8")
input$set("nbGens", "1000")
```

The set method can take some of the formatting away from you as well. You can actually input several arguments for the entry. They will be taken by an ellipsis and will be pasted and collapsed together (adding a space in between; `paste(entry, collapse=" "))` to make a string out of them. Similarly, if you input something else than a string (a vector, a matrix, etc…), then the set method will also paste and collapse the argument into a string. For example, the input

```
input$set("PN", 3)
input$set("N", "unif 100")
```

is equivalent to

```
input$set("PN", 3)
input$set("N", "unif", "100")
```

and equivalent to

```
input$set("PN", 3)
input$set("N", "unif", 100)
```

The following outputs are also equivalent

```
input$set("PN", 3)
input$set("N", "A 100 100 100")
```

```
input$set("PN", 3)
input$set("N", "A", rep(100,3)")
```

If, for example, you want an exponential distribution of mutation rates with an average rate of 1e-7 and bounded to 0.01, you could do

```
mutRates = rexp(1e3, 1e-7)
mutRates[mutRates>0.01] = 0.01
input$set("T5_mu", "A", mutRates)
```

Be aware of how R `paste` function works. For example, when applied on a matrix, the `paste` function parse by columns and not by rows (while SimBit expect matrices to come by rows). For example,

```
migMatrix = matrix(c(0.9,0.1,0,1),byrow=TRUE, ncol=2)
migMatrix
     [,1] [,2]
[1,]  0.9  0.1
[2,]  0.0  1.0

paste(migMatrix, collapse=” “)
"0.9 0 0.1 1"
```

So, do not forget to take the transpose of your argument.

```
input$set("m", "A", t(migMatrix))
```

Because, paste when applied to data.frames give results that are not very helpful, the set method also automatically converts data.frame into matrices before pasting and collapsing.

Behind the scene, the Input object keeps every entry as an element of a vector. You can then print them all together on a single line in a file or on several lines.

```
input$print("/path/where/save/command.txt")
```

By default, the file is erased and the input is printed one option per line. However, you can use the optional arguments `append` and `oneLine` to change that. By default, `append=FALSE` and `oneLine=FALSE.` If you want to combine several input on the same files, you will likely want to do

```
input$print("/path/where/save/command.txt",
append=TRUE, oneLine=TRUE)
```

You will likely not need it but there is also a `write` method that allows you to write any arbitrary text into the input. For example,

```
input$write("--PN 5")
```

is equivalent to

```
input$set("PN", 5)
```

I sometimes use `write` to write comments

```
input$write("# This is my comment")
```

If you write comments, do not forget that you will need to `print` your input on several lines (`oneLine=FALSE`) as comments are accepted only if the whole file is read by SimBit at once. By default, `write` prints in a new line but if you want it to be on the line with the previous entry you set, then you can set the optional argument `appendToPrevious` to TRUE.

I personally like to start every input file with the simulation ID and the creation date of the input file. I use something like

```
input$write(paste0(
    "# Simulation ID = ", ID,
    " – input file created on '", date(), "'"
))
```

I would rarely run a simulation directly from R personally as I run my simulations from a cluster, but if you want to, you can use the method `run` in which you specify the executable. For example,

```
input$run("/path/to/SimBit")

     ____  _           ____  _ _
    / ___|(_)_ __ ___ | __ )(_) |_
    \___ \| | '_ ` _ \|  _ \| | __|
     ___) | | | | | | | |_) | | |_
    |____/|_|_| |_| |_|____/|_|\__| version 4.9.21

        Time for initialization: 0 seconds (0.006293 seconds)

        Generation: 1000 / 1000

        Time for the simulation: 2 seconds (1.932 seconds)

           --> Simulation is over. Good Job! <--
```

If you do not specify an executable as argument, the Input class will try to run "SimBit" without specifying a path which will work if you placed the executable in your PATH.

17.2 ParameterGrid

SimBitWrapper also contains a three simple functions called `fullFactorial` , `getDataFromParameterGrid` , and `gatherData` that ease the creation of a grid of parameters, the extraction of data from this grid and the gather of data for analyzing results, respectively.

`fullFactorial` is a function that expects an undefinite number of lists. Each list must contain one or more named elements. `fullFactorial` expands in a full factorial manner elements of lists that are found in different lists but not those found in the same list. Hence, elements within the same list must contain the same number of elements (or a number that is a multiplicative of each other). It is much easier to explain with an example

```
PGrid = fullFactorial(
    list(nbLoci = c(8, 16, 32, 64, 128)),
    list(
         selectionStrength = c(0, 0.001, 0.01),
         dominanceCoefficient = c(1, 0.5, 0.2)
    ),
    list(
         patchNumber = 1,
         patchCapacity = 1000,
         nbGenerations = 1e5,
         mutationRate = 1e-7
    ),
    list(replicate = 1:10)
)
```

It creates a data.frame of 150 rows because there are five values in the first list, 3 values in the second list and 10 values in the last list (5×3×10=150). Because `selectionStrength` and `dominanceCoefficient` are within the same list, the value s=0 will always be associated to h=1, the value s=0.001 always associated to h=0.5 and the value s=0.01 always associated to h=0.2. The third list contains variables that are 1 in length so they don't add any rows to the grid. The last list contains 10 elements and is used here to consider several replicate simulations of the same set of parameters. It is often recommended to indicated a unique ID at each row. Something like

```
PGrid$ID = paste0("simulationSet_A.", 1:nrow(PGrid))
```

And directly set the name of the files where outputs will be printed

```
PGrid$outputFile =
paste0("/Users/Remi/simulationSet_A/", PGrid$ID)
```

The function `GetParameterGridData`, now. This function takes two arguments; a data.frame (a parameter grid) and a row index and it creates a new variable for each column. The variable has the name of the column and is set to the value that the column takes at the specific row. For example,

```
GetParameterGridData(PGrid,  12)
```

creates the variables `nbLoci, selectionStrength, dominanceCoefficient, patchNumber, patchCapacity, nbGenerations`, and `mutationRate` to the corresponding values at row 12.

The third function is called `getData`. This function takes three arguments; a data.frame (a parameter grid), the name of a column in the parameter grid (called `columnNameOutputFile`), with the name of the outputted file (extension is optional), and a function `fun` that explains how to extract data from a file. By default the column name is set to `outputFile` and by default, `fun` is a function that look for a file that start with the file indicate in the column pointed by `columnNameOutputFile` and end with any extension. It ensures there is only one such file and ensure the file only contains one line (excluding the header). It then simply add the column of this file to the parameterGrid.

Let's consider a complete example of code that will test how different migration rates and number of patches in an island model affect F_{ST}. The first step is to create a grid of parameters (a "data.frame"), where each row contains information for a single simulation. We will use a full factorial design with three distinct migration rates and four distinct number of patches. We will run 20 replicates for each of these 3×4=12 combinations resulting in a grid of parameters of 240 rows.

```
########################
## Load SimBitWrapper ##
########################

require(“SimBitWrapper”)

###############################
## Create grid of parameters ##
###############################

### Initialize parameter grid
parameterGrid = fullFactorial(
    list(PatchNumber = c(2,4,8,16)),
    list(migrationRate = c(1e-4, 1e-3, 1e-2)),
    list(N = 1e5),
    list(nbLoci = 1e6),
    list(nbGenerations = 5e4),
    list(recRate = 1e-4),
    list(mu = 1e-4),
    list(replicate = 1:20)
)

### Create a column containing names of output files
parameterGrid$outputFile = paste0(
     “Users/Remi/MySims/output_”,
     1:nrow(parameterGrid$outputFile)
)
```

The second step is to loop through the rows of the parameter grid in order to run the simulations (or to create the input file to run them later on). For this, we use the function “getDataFromParameterGrid”, which, for each column of the grid of parameters, sets a variable with name equal to the column name and value equal to the row of the parameter grid given in input.

```
##############################################
## Create input commands and run simulation ##
##############################################

For (row in 1:nrow(parameterGrid))
{
   getDataFromParameterGrid(parameterGrid, row)

   ### Initialize the input
   input = Input$new()

   ### Set the values
   input$set(“PN”, PatchNumber)
   input$set(“m”,“island”, migrationRate)
   input$set(“N”, “unif”, N)
   input$set(“nbGenerations”, nbGenerations)
   input$set(“L”, “T1”, nbLoci)
   input$set(“T1_mu”, “unif”, mu)
   input$set(“r”, “rate”, “unif”, recRate)
   input$set(“T1_FST_file”, outputFile)

   ### Run the simulation
   input$run()
}
```

Please see manual for more information about what executable is used by the ‘run’ method. Note also that this example directly run the simulation. It is, however, often more practical to print the input command into a file (with “input$print(filename)”) and run the simulations directly from the shell at a later time. Finally, the last step is to gather the outputs and graph the results. In order to gather the outputs, we use the function “gatherData”. This function uses a number of optional parameters (see manual) but default parameters work fine for our simple example.

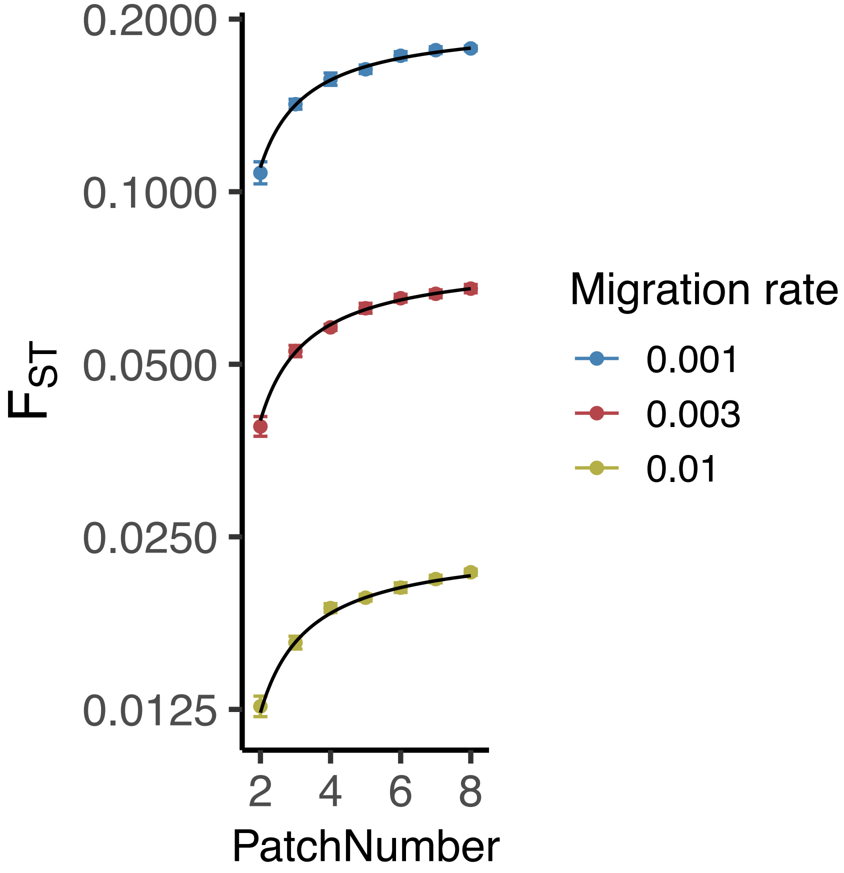
```
##############################
## Gather and graph outputs ##
##############################

### Gather simulation outputs
data = gatherData(parameterGrid)

### Graph
require(“ggplot2”)
ggplot(data, aes(y=(HT-HS)/HT, x=PatchNumber,
color=migrationRate)) + geom_point()
```

In this simple example, the entire study (defining the parameters, creating the inputs, running the simulations, gathering and graphing the results) takes 19 lines of code (19 expressions; including loading packages, excluding the curly braces! It can actually be reduced to 6 expressions. Here it is (without the comments) for fun

```
parameterGrid = SimBitWrapper::fullFactorial(
    list(PatchNumber = c(2,4,8,16)),
    list(migrationRate = c(1e-4, 1e-3, 1e-2)),
    list(N = 1e5),
    list(nbLoci = 1e6),
    list(nbGenerations = 5e4),
    list(recRate = 1e-4),
    list(mu = 1e-4),
    list(replicate = 1:20)
)

ParameterGrid$outputFile = paste0(
     “/Users/Remi/MySims/output_”,
     1:nrow(parameterGrid$outputFile)
)

for(row in 1:nrow(parameterGrid)){

    SimBitWrapper::getDataFromParameterGrid(parameterGrid, row)

    SimBitWrapper::Input$new()$set(“PN”,
    PatchNumber)$set(“m”,“island”, migrationRate)$set(“N”,
    “unif”, N)$set(“nbGenerations”, nbGenerations)$set(“L”,
    “T1”, nbLoci)$set(“T1_mu”, “unif”, mu)$set(“r”, “rate”,
    “unif”, recRate)$set(“T1_FST_file”, outputFile)$run()
}

ggplot2::ggplot(SimBitWrapper::gatherData(parameterGrid),
aes(y=(HT-HS)/HT, x=PatchNumber, color=migrationRate)) +
geom_point()
```

Below is the resulting figure (with a few extra ggplot2 commands to tweak the appearance). The black lines represent analytical expectations from Charlesworth (1998). The F_{ST} values computed with Weir and Cockerham (1984) estimator and therefore correspond to the column “FST_WeirCockerham_ratioOfAverages” in the output files. Note that the y-axis is on a logarithmic scale. Error bars represent 95%

confidence interval. Most error bars are too small to be distinguished from the dot representing the mean F_{ST}.


F_{ST}
PatchNumber
Migration rate