N E M O

release 2.4

version 2.4.0

User Manual

June 11, 2025

authors
Frédéric Guillaume
Jobran Chebib
Max Schmid
Champak Beeravolu

contributors
Jacques Rougemont (MPI version)
Samuel Neuenschwander
Alistair Blachford
Sam Yeaman
Kim Gilbert

availability
https://nemo2.sourceforge.io

Contents

Chapter 1

Introduction

1.1 Overview

Nemo is a forward-time, individual-based, genetically explicit, and stochastic simulation program designed to study the evolution of genetic diversity, life histories, and phenotypic traits in a flexible (meta-)population framework. Nemo implements different types of trait or genetic elements affecting individual fitness in various ways. The genetic elements of a simulation (genetic loci) can be linked together on a recombination map. The evolving traits provided are sex-specific dispersal rates, deleterious mutations, multivariate quantitative traits, Bateman-Dobzhansky-Muller incompatibility loci, and neutral markers (e.g., SNP, microsatellites). Nemo also allows for the simulation of the dynamics of an endosymbiotic parasite vertically transmitted and causing cytoplasmic incompatibility; *Wolbachia*. The number of populations, individuals per population or loci per trait in a simulation are only restricted by hardware capacities. Nemo is highly optimized to run in batch mode and a parallel computing version is part of the release, thus making it a very flexible and powerful simulation tool. Nemo's framework is coded in C++ and has been designed to be easily extended to include new evolving traits or population features.

Availability: Nemo comes free of charges and is distributed under the GNU General Public License (GPL3+). Binary files are provided for the Linux, MacOSX, and Windows platforms. Nemo is coded in C++ and runs on any platform supporting a console-like environment and allowing it to be compiled with standard C/C++ compilers (GNU gcc being the default).

Installing: Installing Nemo is straightforward, you just need to copy the binary file corresponding to your operating system from the hosting web site (`http://nemo2.sourceforge.net/`) and use it at once or, in the case your operating system is not supported, copy the source code, compile it and use the executable. See the documentation provided with the source package for instructions concerning the compiling process.

Using: The basic users' interface is a text file (a.k.a the 'init file') containing the input parameters and their argument(s) as whitespace-separated `key value` pairs. NEMO is then started from the console with that init file as an argument. Some run-time information (current running simulation, current generation/replicate, etc.) is written to the standard output (terminal window). NEMO also gives the possibility to save the simulation data to a variety of files in text or binary format, depending on the options chosen in input. The user may save the traits' complete genotypic information, the simulation's summary statistics, or the complete state of the population, periodically. See chapter 3 for input directions and chapter 4 for parameters description.

Extending: Nemo is designed as a flexible and extensible coding framework. It is aimed at facilitating the implementation of new components such as new evolving traits with their specific genetic architecture and new life cycle events, while taking advantage of the simulation management features offered by the framework (i.e. input/output management, interaction with existing components, etc.). The basic coding procedures are described on the coding documentation web site: `http://nemo2.sourceforge.net/start.html`.

Acknowledgments: The parallel computing version (Nemo_MPI) has been developed in collaboration with Dr. Jacques Rougemont at the Swiss Institute of Bioinformatics using the Message Passing Interface (MPI) standard (`http://www.mpi-forum.org`) allowing to run simulations on HPC cluster environments. That parallel version uses the Scalable Parallel Random Number Generators library (SPRNG; `http://sprng.cs.fsu.edu`) as a source of random numbers. The regular NEMO version implements a random number generator (i.e. the Mersene Twister) provided by the GNU Scientific Library (GSL; `http://www.gnu.org/software/gsl`) as well as several other mathematical routines defined in that library.

1.2 Using Nemo

Let's assume you have copied the executable file corresponding to your operating system on your disc and that you have launched a terminal window. The following guidelines will show you how to launch a simulation on your desktop computer on both *nix flavored operating systems and Windows. Guidelines to launch a parallel job on a computer grid or cluster environment are not provided here. These will vary according to the type of infrastructure you have access to.

1.2.1 Running Nemo from the command line

1.2.1.1 For Linux and Mac OS X users

On Mac OS X, the terminal application, called Terminal.app, is located in the `/Applications/Utilities` directory on your hard drive. Simply double click to start it. Then, assuming you have installed the executable file `nemo2.x.y` in a folder somewhere on your file system and that you set your working directory to that place (using the `cd` command). The following commands will allow you to run a simulation.

First, lets have a look at the content of the directory using the `ls` command:

```
> ls
nemo2.x.y*  Nemo2.ini
```

We have the executable file, `nemo2.x.y` and a configuration file, `Nemo2.ini`. Now, if we type the following command, Nemo will automatically search for the `Nemo2.ini` file in the local directory and try to initiate a simulation from it.

```
> ./nemo2.x.y
```

The './' characters in front of the executable filename simply means that the program file is to be searched in the local directory rather than in one of the directories specified by the PATH environment variable (type `echo $PATH` on the command line to see what it holds). You can copy Nemo in one of the directories on your PATH or add a directory containing Nemo to your PATH (search for tutorials on the Internet). The command `./nemo2.4.0` will run Nemo version 2.4.0 and produce the following output to your terminal window (or something approaching depending on the program's version):

```
> ./nemo2.4.0

  N E M O 2.4.0 [10 Nov 2023]

  Copyright (C) 2006-2023 The Authors
  This is free software; see the source for copying
  conditions. There is NO warranty; not even for
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
  https://nemo2.sourceforge.io; nemo-simul@googlegroups.com
  ------------------------------------------------
  reading parameters from "Nemo2.ini"
  setting random seed from input value: 486532

--- SIMULATION 1/8 ---- [ POLY_dcost01_ISM ]

  start: 10-11-2023 16:12:02
  mode: overwrite
  traits: delet, fdisp, mdisp, ntrl
  LCEs: breed_selection(1), save_stats(2), disperse_evoldisp(3), \
        aging(4), save_files(5), store(6), extinction(7),
  outputs: test/{*.log, delet/*.del, fstat/*.dat, fstat/*.freq, \
             data/*.txt, binary/*.bin}

  replicate 10/10 [16:12:21] 100/100

  end: 10-11-2023 16:12:21
--- done (CPU time: 00:00:18s)
setting random seed from input value: 486532

 --- SIMULATION 2/8 ---- [ MONO_dcost01_ISM ]

  start: 10-11-2023 16:12:21
  [...]
```

This terminal output shows the components of the simulation and the progress of the simulation with the replicate and generation counters. Nemo also prints the time when the current replicate started (in format hh:mm:ss), and the total elapsed computing time (`CPU time: hh:mm:ss`) once the simulation is done. The parameter file used in this example is the one present in the `example` directory of the distribution package.

1.2.2 Batch mode

Nemo accepts only one type of argument on the command line, the name(s) of the file(s) containing the parameters of the simulations (the *init* files). For instance, if three init files are passed to Nemo, the program will initiate three simulations from those files. Nemo may also run many simulations from a single init file if that file contains parameters with multiple values. Such parameters are then called *sequential* parameters (see section 3.5) and Nemo will build and run one simulation per parameter value (or combination of values if multiple sequential parameters are present). Of course, multiple init files each containing sequential parameters can be passed as arguments to Nemo as well. In all cases, Nemo builds a list of simulation parameters, one for each simulation and runs each simulation sequentially in the order provided on the command line. Simulations can thus be "chained", running one after the other.

Let's illustrate this by first running Nemo with more than one argument:

```
> ./nemo2.4.0 sim1.ini sim2.txt sim3
```

Here we have three init files called `sim1.ini`, `sim2.txt` and `sim3`, they are all text files, the extension does not matter. The filename parameter is set to sim1, sim2, and sim3 in each file, respectively. The command above will produce the following output:

```
  N E M O 2.4.0 [10 Nov 2023]
[...]
------------------------------------------------
reading parameters from "sim1.ini"
reading parameters from "sim2.txt"
reading parameters from "sim3"

--- SIMULATION 1/3 ---- [ sim1 ]

    [...]
    replicate 10/10 [10:04:54] 100/100

    end: 12-09-2018 10:04:54
--- done (CPU time: 00:01:26s)

--- SIMULATION 2/3 ---- [ sim2 ]

    [...]
    replicate 10/10 [10:06:36] 100/100

    end: 12-09-2018 10:06:36
--- done (CPU time: 00:01:26s)

--- SIMULATION 3/3 ---- [ sim3 ]

    [...]
    replicate 10/10 [10:08:13] 100/100
    end: 12-09-2018 10:08:1
--- done (CPU time: 00:01:26s)
```

Sequential parameters As an example of sequential parameters, let's assume that the first file "`sim1.ini`" has one parameter with several arguments:

```
patch_capacity 5 10 20
```

This will add two more simulations to the three previous ones:

```
> ./nemo2.4.0 sim1.ini sim2.txt sim3
[...]
reading parameters from "sim1.ini"
reading parameters from "sim2.txt"
reading parameters from "sim3"

--- SIMULATION 1/5 ---- [ sim1-1 ]

[...]

--- SIMULATION 2/5 ---- [ sim1-2 ]

[...]

--- SIMULATION 3/5 ---- [ sim1-3 ]

[...]

--- SIMULATION 4/5 ---- [ sim2 ]

[...]

--- SIMULATION 5/5 ---- [ sim3 ]

[...]
```

As you can see in this example, the simulation name of the first simulation (set as 'filename sim1' in `sim1.ini`) now has an extra number added to it to distinguish the three simulations triggered by the sequential parameter patch_capacity. The number is automatically generated but can be replaced with more informative chains of characters generated from the values assigned to the parameter. This is explained in section 3.5.

Chained simulations An example is provided when running the default simulation example Nemo2.ini followed by two simulations parameterized to use some of the output files as input simulation data to run from:

```
> nemo2.4.0 example/Nemo2.ini example/fstat_input_and_cross.ini \
example/binary_input_and_cross.ini
```

1.2.3 Parallel version (MPI)

Nemo also comes with a MPI layer fully integrated into the code. Nemo can thus be built to run on multiple CPUs using the message passing interface (MPI). See the INSTALL accompanying file for instructions. The MPI version distributes the replicates of a simulation among multiple CPUs (the workers) and a 'manager' CPU collects the data from each replicate, distribute remaining replicates to the workers and saves the concatenated result files at the end of the simulation (if results need be concatenated).

To properly run, an MPI job must be allocated $n\_replicates + 1$ CPUs, 1 for the manager, and *n_replicates*, or a divider of *n_replicates* for the workers. For instance to process 15 replicates, you can require 6 CPUs on your cluster, with each of the 5 workers CPU processing 3 replicates. The following `mpirun` and job scheduler commands sent to the cluster would correctly allocate the parallel job and execute it. The `mpirun` command would be:

```
mpirun -np 6 nemo2.3.53_mpi myinitfile.ini
```

That command, written in a Bash script '`sub_nemo_mpi.sh`', would then be sent to the cluster (e.g., a SLURM cluster, requesting a total walltime of one hour):

```
sbatch -n6 -c1 --time1:0:0 sub_nemo_mpi.sh
```

Chapter 2

Main Features

Nemo is built to simulate eco-evolutionary dynamics in large populations of a single species. To do so, Nemo evolves populations forward in time over successive iterations of a life cycle. Each iteration is either a generation in populations with non-overlapping generations or a year in age- or stage-structured populations. The life cycle itself is composed of a succession of events, whose nature and order of execution is freely chosen by the user. These events modify the state of the population and affect the individuals within it. The population dynamics can be a function of the genetic composition and distribution of trait values in the population and of the action of the life cycle events (LCEs). A variety of population models (Wright-Fisher and non-Wright-Fisher) can be implemented by assembling different life cycle events, as explained in section 2.2.

The fate of an individual during the execution of a life-cycle event may depend on the value of the traits determined by the genes it carries. Examples of genetic elements in Nemo are quantitative trait loci contributing to multiple phenotypes, neutral markers, or genes affecting fitness directly, for instance deleterious mutations or Dobzhansky-Muller incompatibility loci, and more (see chapter 6). In addition, genes can be placed on a genetic map, for instance parameterized with the recombination map of an existing species. Complex genotype-phenotype-fitness maps can thus be built when assembling different genetic elements on the genetic map. Alternatively, individuals may be devoid of any genetic information and Nemo can be used to simulate stochastic population dynamics.

The next sections present the main simulation components and the general philosophy and architecture of the program. We will explain how the simulation components are assembled to create a population model and simulate its dynamics. The user's interface of Nemo is a text file holding the parameters of the desired components of a simulation. Chapter 5 presents the parameters of the life cycle events (LCEs) and chapter 6, the parameters of the different trait types (*i.e.*, genetic elements). Components are optional and can be assembled at will.

2.1 A basic life cycle

Before we jump into the details of the different components of a life cycle, let's have a look at how those components are assembled to build a basic simulation. The simplest possible simulation would have just reproduction and generation replacement, or aging, thus simulating a population without selection or dispersal. We only present the life cycle here, without yet showing the other components.

A life cycle is built by giving a **rank** to each LCE that we decide to add to a simulation. The LCE's name and rank only need to be written to the input parameter file as shown here:

```
breed     1    # offspring are produced
aging     2    # adults are removed and offspring become adults
```

Here, adult individuals will mate according to the mating parameters of the breed LCE, which are used to specify the mating system (*e.g.*, random, monogamy, polygyny, etc.) After breed, the population will contain adults (the breeders) and their offspring. The number of offspring will depend on the value of the `mean_fecundity` parameter in breed, in a non-Wright-Fisher population (see section 2.2). Next, the aging LCE removes all adults and replaces them with the offspring, randomly chosen until the population is filled to its carrying capacity set with the `patch_capacity` parameter of the population component. This life cycle is then iterated as many times as the value passed to the `generations` parameter of the simulation component.

To this basic life cycle, dispersal and selection can be added. In Nemo, only offspring disperse between patches (synonym of deme or sub-population) and are under viability selection. Multi-stage/age populations are modeled with Nemo-age. Now, whether selection acts before or after the individuals disperse between patch will depend on the rank of the LCEs responsible for dispersal and selection. We can thus build two different life cycles where individuals are selected either in their natal patch, as in the next example:

```
breed               1  # offspring are produced
viability_selection 2  # offspring are selected (survival)
disperse            3  # offspring disperse among patches
aging               4  # generational substitution
```

or, offspring are selected in their patch of arrival after dispersal:

```
breed               1  # offspring are produced
disperse            2  # offspring disperse among patches
viability_selection 3  # offspring are selected (survival)
aging               4  # generational substitution
```

The timing of selection and dispersal will mater, for instance, when selection is acting on the match between a phenotypic value of the individuals with the value of the patch-specific phenotypic optimum. This is relevant when patches have different optimum values. Phenotypes are modeled with the quantitative trait and local phenotypic optima are set within the `selection` LCE (with the Gaussian and quadratic selection models). Note that populations without selection or without dispersal can also be modeled, when removing one or the other LCE from the life cycle.

These two examples represent typical life cycles of non-Wright-Fisher populations with demographic stochasticity because selection and dispersal are decoupled from reproduction and may result in patches with less or more individuals than specified by their carrying capacity. The `aging` LCE is then essential to regulate the population and avoid exponential growth. As explained in subsection 2.2.1, specific LCEs, and parameters, need to be used to model Wright-Fisher populations of constant size with dispersal and/or selection.

2.2 Population models

NEMO offers the possibility to model many different kinds of population, from Wright-Fisher populations of constant size with random mating and non-overlapping generations, to stage-structured populations with stochastic growth, overlapping generations and non-random mating, and anything in-between. Population models also differ in how individuals move among patches and on the timing of reproduction, dispersal, survival and density regulation. The population model simulated depends on how the different life-cycle events are chosen and assembled by the user. No pre-defined model is enforced for a simulation, although NEMO assumes non-Wright-Fisher populations by default. The different population models are described in this section.

2.2.1 Wright-Fisher populations

The simplest population model is one with constant population size where the number of offspring created corresponds exactly to the population size and where reproductive individuals die after reproduction, allowing for non-overlapping generations and removing demographic stochasticity. This basic model corresponds to the classic Wright-Fisher (WF) model in population genetics when reproduction is random. In that case, the population is called “panmictic”. WF populations can be sub-divided into patches connected by migration and be subject to the action of selection. A basic WF model only requires a switch to be turned on in the `breed` LCE (namely, the `mating_isWrightFisher` parameter). For WF populations with migration, dispersal must be handled specifically to ensure constant population sizes in all patches.

This is achieved when modelling *backward migration* with breed_disperse, where gametes (*e.g.*, pollen grains) rather than zygotes (*e.g.*, seeds) are moving between patches. Additionally, WF populations with selection can be modeled by randomly choosing progenitors proportionally to their relative fitness values within a patch. This is done by replacing the breed LCE by the breed_selection LCE in the life cycle. Finally, WF populations with migration and selection need to be handled differently since the fitness of the foreign parent(s) producing immigrant gametes do not have a local relative fitness value. The relative fitness in the residing patch of the parents can be used instead (see the fitness model options in the viability_selection LCE). In any case, the breed_selection_disperse LCE handles reproduction and selection with backward migration in populations of constant size. WF populations can thus never go extinct and are not a good option to model phenomena like evolutionary rescue.

Here is an example of a single WF population with selection:

```
## Population :
patch_number      1
patch_capacity    1e4   # 5000 males, 5000 females

## Life cycle :
breed_selection   1     # no need for ‘aging’ LCE here

## breed parameters :
mating_isWrightFisher   # necessary for Wright-Fisher population
mating_system     1     # random mating (males + females)

## selection parameters (not shown)
```

Here is an example of a WF population with dispersal:

```
## Population :
patch_number      5
patch_nbfem       2000  # females only
patch_nbmal       0

## Life cycle :
breed_disperse   1      # backward dispersal
aging            2      # aging is needed for adults substitution

## breed parameters :
mating_isWrightFisher   # necessary for Wright-Fisher population
mating_system     6     # random mating, hermaphrodites (females)
```

```
## dispersal parameters :
breed_disperse_rate         0.02
breed_disperse_model        3     # 1D stepping stone model
```

2.2.2 Non Wright-Fisher populations

In Nemo, populations are non-Wright-Fisher by default and thus subject to demographic stochasticity. In other words, patch sizes can fluctuate during a simulation depending on how individuals reproduce, disperse and survive. Population size, nevertheless, is controlled by the *carrying capacity* of the patches set within the population component. To prevent populations from growing exponentially, patches are regulated to their carrying capacities at the end of each iteration of the life cycle. This is handled by the aging LCE. The regulation of patches to their carrying capacity is a type of density regulation called *ceiling regulation*, because patches are capped to a fixed maximum value. On the other hand, nothing really prevents populations from having smaller population sizes than their carrying capacity and thus from going extinct. Therefore, all stochastic processes in a simulation will affect fluctuations in patch sizes. Demographic stochasticity is strongest when patch size is small (*e.g.,* < 100), mean fecundity is small (*e.g.,* < 3) and individuals have low fitness (survival) because of accumulation of deleterious mutations or mal-adaptation to their local trait optimum values. Here, fitness is *absolute* and directly represents offspring survival in the viability_selection LCE. This is in contrast to the relative fitness of a WF population. Note that relative fitness can also be modeled in a non-WF population within the viability_selection LCE when setting the fitness model.

Non-WF populations can implement more types of life cycles with both forward (disperse LCE) and backward dispersal (breed_disperse), selection on offspring survival (viability_selection) or parent's fecundity (breed_selection), and different types of population growth with breed_disperse. For instance, mixing forward and backward dispersal is possible and allows for simulations of tree populations with pollen flow (backward migration) and seed dispersal (forward migration), each with different dispersal rates and kernels.

2.2.3 Demographic scenarios

Independently of the type of population regulation, WF or Non-WF, Nemo also facilitates the simulation of complex demographic scenarios with population extinction and re-colonisation (with the extinction LCE), or population fission/fusion with the resize LCE. The patch size (or carrying capacity) can also change through time with temporal parameter values. Together with the variety of dispersal models, this allows for a wide variety of dynamics scenarios in which patches can be added or removed, or populations can grow at different rates after extinction or fission of an existing patch into patches connected by migration.

2.2.4 Space

In Nemo, patches do not have spatial coordinates or physical distances set between them. Instead, space is abstracted by the dispersal rates between the patches. In that respect, Nemo is very flexible as it allows the user to pass the full, or reduced dispersal matrices describing the probability of movement of either gametes or zygotes between any pair of patches, and for males and females separately. This way, specific models of dispersal with various kernels can be used to create the necessary dispersal matrices and pass them as parameter values to the program. In particular, this allows for the simulation of populations set on large grids modeled after real landscapes. Moreover, you can design reduced connectivity and rate matrices to reduce the amount of data passed to Nemo when patches are only sparsely connected. This saves on both execution time and disk space. In addition, Nemo can build those matrices for standard migration models as Wright's Island Model or the Stepping Stone model, in either one or two dimensions (*i.e,* on a lattice). The disperse LCE is responsible for setting and storing the dispersal matrices.

2.2.4.1 Continuous vs. discreet space

Because of the way how dispersal is set between patches, space in Nemo in inherently discreet. *Continuous space* can, however, be modeled as well when mixing backward and forward dispersal, for instance with breed_disperse and seed_disperse, respectively. The backward dispersal LCE can be used to define a "mating kernel", specifying the distance over which the mating partners of a focal individual are sampled within and outside its residing patch. Patches can then hold just a few, or just one reproductive individual while the offspring created are then dispersed with the forward dispersal LCE.

2.3 Phenotypes and genotypes

Nemo utilizes the term trait to encapsulate the concepts of genetic elements and their corresponding phenotype. In other words, traits are different kinds of **genotype-phenotype mappings**. Since traits are individual properties, they are transmitted by individuals. Obviously, genetic elements are transmitted vertically from parents to offspring through the processes of recombination, segregation and mutation. Non-genetic traits can be transmitted either vertically (*e.g.*, transmission of endosymbionts like *Wolbachia*, see section 6.7) or horizontally, as for instance when transmitting diseases (not implemented in Nemo yet).

The different mappings, or traits, are as follows:

- **ntrl** Neutral markers (from SNPs to microsatellites, **no phenotype**).
- **delet** Universally deleterious mutations (mutations affecting **fitness**).
- **quant** Quantitative trait loci (multiple correlated **phenotypes**).
- **dmi** Bateson-Dobzhansky-Muller incompatibility loci (mutations affecting **fitness**).
- **disp** Dispersal genes (male and female specific **dispersal**).
- **wolbachia** Endosymbiotic parasite causing cytoplasmic incompatibility with vertical transmission.

Three traits can be under direct selection for survival or reproductive output and linked to the viability_selection LCE: delet, quant, and dmi. Selection on the dispersal genes disp obviously also occurs although not directly since fitness is not a direct function of the genotypic values of the loci. Instead, it depends on indirect selection caused by kin selection or selection for inbreeding avoidance when deleterious mutations are added to the mix (e.g., see Guillaume and Perrin, 2006, 2009). Selection on the quantitative trait phenotypes is modeled via ad-hoc models of *quadratic* or *Gaussian* selection.

The loci of the first four traits (ntrl, delet, quant, and dmi) can be placed on the same genetic map, which describes how loci are grouped among chromosomes. Chromosome may vary in size and recombination rates. Recombination rates can also vary on a chromosome and the map position of each locus can be specified or randomly set. It is then possible to model existing genetic maps based on empirical estimates. We can then model the effect of linked selection on neutral variation, for instance.

Note that the phenotype of the quantitative trait can be multi-dimensional, with pleiotropic loci encoding multiple phenotypic traits under selection (Guillaume, 2011; Chebib and Guillaume, 2017). The other traits whose variation can also be under selection are classical population-genetics traits affecting fitness directly without an intermediate phenotype. The selection LCE is such that it can handle selection on multiple traits, each with a different genotype-fitness or genotype-phenotype-fitness mapping.

2.3.1 Genetic model

Unlike some other forward-time population genetics simulators, NEMO does not implement an infinite site model. Instead, NEMO implements di-allelic and *infinite allele* models on a fixed genomic representation. For di-allelic loci, it is similar to an infinite site model in which the positions and effects of the mutations have been pre-drawn at the start of the simulation. Indeed, the infinite site model implementation usually starts with a blank set of mutations in a given genomic region and slowly builds a register of segregating mutations. In contrast, Nemo allocates the memory space at startup for all simulated loci at which mutations will appear during the simulation. This has some computational advantages and disadvantages that we will not fully address here, if to say that it facilitates the simulation of large sets of polymorphic loci or of loci with a large number of alleles (*e.g.,* QTL). These are situations where existing infinite-site model implementations are comparatively less efficient (see Matthey-Doret, 2021).

This implementation allows for the modelling of loci with many alleles, such as QTL, where the precise base-pair location of a mutation matters less than its genotypic and phenotypic effect(s). Such large loci can, nevertheless, be placed on a genetic map together with other types of genetic elements, for instance neutral or deleterious sites with di-allelic loci. This way, genomic-like data consisting of SNP markers affected by selection on QTL can be simulated.

2.4 Simulation components

Life cycle events and traits are two types of simulation components. Two other main components are necessary to run a simulation, the simulation and population components. Each are described below. To those, we can add the components responsible for data handling, either as file handlers for data input and output, or as stats handlers for computing summary statistics on the population during a run. This last feature makes it easy to monitor the state of a simulation without having to post-process simulation output data (see chapter 8). The file handlers are mostly used to save the genetic and phenotypic information of a whole or sampled population to a file for post-processing. They can also be used to input genetic data to seed a simulation or for data analysis (using the stat handlers to compute summary statistics).

2.4.1 The Simulation

The `simulation` component manages the other components: the population, the LCEs, and the traits. The parameters of this component are used to set the simulation environment. They set the simulation output directory and base filename, the seed of the random number generator, the logging and verbosity of the program at runtime, and, more importantly, the number of replicates and generations. Several run modes are possible to help with testing and running simulations in different environments (see section 4.1 for details). At a deeper level, the `simulation` component manages all the other components, their initialization from the input parameters, and their execution within the replicate and generation loops. For instance, it initializes and builds the population itself, making it ready for the simulation. The `simulation` component also manages the record of all simulations to run from the specification of multiple parameter values in the input file (see section 3.5).

2.4.2 The Population

The `population` component is the backbone of the simulation. It contains the individuals, distributed among sub-populations, or patches (or demes). The individuals are distributed within patches in different containers according to their sex and age or stage. The population elements are thus set as a chain of containers: the population contains the patches, which contain the individuals, who contain the traits and their genes. The LCEs then change the state of those containers through time.

2.4.3 The Individual

An individual in Nemo is basically defined as a trait container. That means that the phenotypes of the individuals depend on which traits are modeled based on the parameters in the input file. By default, individuals don't carry any genetic information in absence of traits. The only pre-defined phenotypes are the individual's age and sex. Individuals also store information about their ancestry and demographics and have a unique ID and a pedigree class (informs if the two parents were a single individual, full-sib, half-sib, or unrelated individuals). Each individual also stores the number of offspring it had and the ID number of its mum, dad, and natal patch. These information tags are used to compute pedigree-based or age-/sex-specific statistics and are often saved to file by the different simulation components.

2.4.4 Statistics and outputs

Nemo provides several ways to record the population state during a simulation. Many summary statistics can be computed at different time points of a simulation. The statistics recorded depend on the simulation components used. Each simulation component can define its set of statistics that the user can choose from and monitor during a simulation. Here are examples of the summary statistics (see all in section 8.2):

- **Neutral trait**: Heterozygosities, F-stats (F_{ST} (G_{ST} and θ), F_{IS}, F_{IT}), allele numbers, number of fixed alleles per locus, coancestries, Nei's D genetic distance, etc.

- **Quantitative trait**: Mean trait values, additive genetic variance (V_{A}) and covariance, phenotypic variance (V_{P}), population genetic differentiation (Q_{ST}), eigenvalues and eigenvectors of the variance-covariance matrix (the **G**-matrix), etc.
- **Deleterious mutations stats**: mutation frequency, heterozygosity, homozygosity, genetic load, heterosis, number of lethal equivalents, viability by pedigree classes, etc.
- **Dispersal trait stats**: mean male and female dispersal rates
- **Population stats**: patch saturation, female and male number per patch, sex-ratio, mean fecundity, variance of reproductive output, count of migrants, effective extinction rate, etc.

The summary statistics are then written to a text file at the end of a simulation. This file is easily handled by classical statistical packages (such as R) for further analysis and graphical representation.

2.4.4.1 Output files

Beside the so-called “stats” files, each component of a simulation may define specific output files that can be written to disc at multiple time points of a simulation. For instance, the traits provide various ways of saving the population genotypes in text files (see the Traits chapter). These files can then be used in different analysis pipelines (e.g., the neutral trait can save SNP genotypes in PLINK, FSTAT, or GENEPOP file formats).

A specific **binary file format** will allow you the save the raw data of the whole population in (compressed) files at different time points during the simulation (see section 5.17). Binary files contain the whole population information, including all the traits and individuals data and the simulation parameters. Binary files can then be used by NEMO to load a saved population and run a new simulation from it (see subsection 4.2.2). This is useful as a backup strategy for long simulations and to store populations after a burn-in phase.

Chapter 3

The input parameters file

The configuration file (or "init" file) read by the program to set a simulation is a text file with one parameter per line in a key/value scheme where the key is the parameter name, and the value its argument value. Each line or string in a line that begins with a '#' character is treated as a comment and is ignored. Parameters are character strings (with no whitespace character within them) that may be followed by one or several argument values separated by at least one white space character (a space or tab). The only restriction is that a particular parameter must appear only once in the init file. The order of appearance of the parameters in the file does not matter.

3.1 Parameter types

Here is a list of the different types of argument a parameter can take:

- **boolean (bool)** : works on a presence (=`true`) / absence (=`false`) basis when no argument is passed. Also accepts `'1'` as `true` (or set) and `'0'` as `false` (or unset).
- **integer** : argument is a dot-less number value; a limit to the number of available values a parameter can take may be specified from case to case.
- **decimal** : argument may be a floating-point value. The following forms are equivalent: 0.0001, .0001 or 1e-4.
- **string** : argument is a character string that may not contain white-spaces.
- **matrix** : special argument that is enclosed by '{ }', inside these brackets, each row of the matrix is also enclosed by two brackets, see section 3.4 for details and examples.

3.2 Special characters

Here is a list of the reserved characters and their meaning during the process of reading and parsing the input parameters file.

- **comment** : `#` : any character that follows the comment character is removed until the end of the line is found. If a starting block comment string (`#/`) is found within a commented line, it is treated as such (see below).
- **block comment** : `#/.../#` : any line of text enclosed by those two-character strings is recursively removed from the init file. A block comment can also be specified on a single line.
- **line continuation** : `\` : the line that immediately follows that character is appended to the current line and the two lines are treated as one. This is particularly useful to split a sequence of argument values over several lines (see the matrix example below).
- **matrix** : `{{`*row1*`}{...}}` : any argument value starting and ending by two enclosing curly braces is considered as a matrix argument (see next section).
- **name expansion** : `%` : used in the character string of an argument to insert the value of another parameter when that parameter has multiple argument values (see sequential parameters in section 3.5).
- **external parameter file** : &: &*filename* : used to pass an argument value to a parameter when that argument value (e.g., a large matrix) is contained in a separate file. The character string *filename* contains the path to that separate file containing the argument value(s) (see section 3.6).
- **specifiers** : `@g` : this short character string is used to specify the generation at which a temporal argument value applies. For instance, "`@g100`" designates a temporal argument value that will be used at generation 100 (see section 3.7). Specifiers must be found within a block argument (see below).
- **block argument** : `(arg1, arg2, ...)` : argument values enclosed with two parentheses are treated in a special way. Parentheses are used when several arguments and their specifiers must be passed to a parameter without being interpreted as a sequence. Such a case appears when specifying temporal argument values (see section 3.7). Argument values are separated by commas within a block argument (e.g., `(@g0 0.02, @g5 0.5)`).

3.3 Macros

Arguments to a parameter in the init file may contain small expressions, called macros, that are interpreted by the program before assigning a value to a parameter. Those macros help the user to specify sequences of values that would be tedious to enter manually. They also allow the user to generate parameter values randomly when the program reads the init file. The basic syntax is copied from the R language to ease implementation. A macro resembles a function, with a name (e.g., `rep`) and a set of arguments within parenthesis (`macro(arg1, arg2=value)`). The basic macros are `rep()` and `seq()`, which both generate sequences of values either by repeating a pattern with `rep()` or generating a suite of numbers with `seq()`. The macros are listed below. They come with mandatory and optional arguments. The latter are shown with an assignment (`name=value`) and must be named when called (e.g., `mean=0`). The default value is shown in the preamble of the macro descriptions.

One typical optional argument to a macro is the character used to separate numbers in an array. Series of numbers are usually comma-separated, and the comma is the default separator specified with the `sep` argument (i.e., `sep=","`). The separator can be changed, for instance to generate white-space separated lists of numbers, which would then be interpreted as so-called *sequential* parameter values and initiate a sequence of simulations (see section 3.5).

Multiple macros can be used per parameter (e.g., `param rep() seq()`) or be nested, as for instance to concatenate and quote lists of numbers to pass as argument to another macro: `param rep(q(c(0.1234,seq(...))))`. See examples below.

3.3.1 rep(): repeat

`rep(x, n, each=1, sep=",") :` Repeats (copies and concatenates) the character string provided by x, n times, and joins each repetition with the character provided by `sep`. The string in x must be quoted if it contains more than one value with delimiter characters (‘`,`’ or space), or parentheses (‘`()`’ or ‘`{}`’). If argument *each* is present and different from one, each element of x is repeated *each* times each. Thus, when $each \neq 1$, then elements of string x are extracted assuming that x is a comma-separated list of elements.

```
examples
in:  what you would write in the init file
out: the result of the call to the macro, and how it replaces your
input string
```

```
in:  param_1 { rep("{0,1}{1,0}", 5, sep="") }
out: param_1 {{0,1}{1,0}{0,1}{1,0}{0,1}{1,0}{0,1}{1,0}{0,1}{1,0}}
```

```
in:  param_2 rep(1, 10, sep=" ")
out: param_2 1 1 1 1 1 1 1 1 1 1
```

```
in:  param_3 {{rep(100,10)}}
out: param_3 {{100,100,100,100,100,100,100,100,100,100}}
```

```
in:  param_4 {{rep("1,2,3", 1, each=2)}}
out: param_4 {{1,1,2,2,3,3}}
```

3.3.2 seq(): sequence

`seq(from, to, by, sep=",") :` Creates a sequence of numbers from $from$ to *to* (included if possible) by incrementing $from$ with value *by* until it reaches *to*. A value larger than *to* will never be returned. The *by* increment value must be $> 10^{-15}$. If $from >$ *to* then *by* must be a negative value, or positive otherwise.

```
in:  param_1 seq(0.01, 0.1, 0.01)
out: param_1 0.01,0.02,0.03,0.04,0.05,0.06,0.07,0.08,0.09,0.1
```

```
in: param_2 seq(0.01, 0.1, 0.02, sep=" ") seq(0.1, 0.5, 0.1, sep=" ")
out: param_2 0.01 0.03 0.05 0.07 0.09 0.1 0.2 0.3 0.4 0.5
```

```
in:  param_3 {{seq(0, -50, -10)}}
out: param_3 {{0,-10,-20,-30,-40,-50}}
```

3.3.3 c(): concatenate

`c(..., sep=",") :` This macro joins a lists of elements provided in input. Any number of comma-separated values can be passed as argument. Values need not be enclosed in quotes but `c()` also works with multiple quoted lists of elements and correctly joins them. The macro uses the character specified by argument `sep` as separator in the output string returned.

```
in:  param_1 c(1,2,3,4,5,sep="-") # silly example
out: param_1 1-2-3-4-5
```

```
in:  param_2 c(0,seq(10, 12, 0.5), sep=" ")
out: param_2 0 10 10.5 11 11.5 12
note: the character separator of seq() is here "," by default
note: the call to c(..., sep=" ") changes the separator to a space
```

```
in:  param_3 c("{{",q(seq(10, 12, 0.5)),"}}", sep="") #see q() below
out: param_3 {{0,10,10.5,11,11.5,12}}
equivalent expression: param_3 {{seq(10, 12, 0.5)}}
```

3.3.4 q(): quote

`q(..., sep=",")` : Use `q()` to quote the output of another macro with `""`. This is useful when the output of a macro is used as input to another macro, when macros are nested. See third example below. The separator char in the returned string is set with *sep*.

```
in:  param_1 q(seq(1, 5, 1), sep = " ")
out: param_1 "1 2 3 4 5"
note: q() has changed the separator from "," to a space
```

```
in:  param_2 q(c(0,seq(10,100,10),seq(200,1000,200)))
out: param_2 "0,10,20,30,40,50,60,70,80,90,100,200,400,600,800,1000"
```

```
in:  param_3 {rep( q(c("{",q(seq(1,5,1)),"}", sep="")), 4, sep="")}
out  param_3 {{1,2,3,4,5}{1,2,3,4,5}{1,2,3,4,5}{1,2,3,4,5}}
```

3.3.5 tempseq(): temporal sequence

`tempseq(at, seq)` : Creates a sequence of temporal argument values within parentheses following the syntax for temporal arguments shown in section 3.7. The first argument *at* to `tempseq()` is the list of generations at which the values given by its second argument *seq* must be assigned to the parameter. The two arguments *at* and *seq* must hold the same number of comma-separated values, passed as quoted strings.

```
in:  param_1 tempseq(at=q(c(0,seq(100,190,10))),
seq=q(seq(0, 0.1, 0.01)))
out: param_1 (@g0 0, @g100 0.01, @g110 0.02, @g120 0.03, @g130 0.04,
@g140 0.05, @g150 0.06, @g160 0.07, @g170 0.08, @g180 0.09, @g190 0.1)
```

3.3.6 matrix(): create matrix

`matrix(x, nrow, ncol)` **:** Generates a *nrow* × *ncol* matrix from a quoted list of numbers passed as first argument x. The number of elements in x must be equal to *nrow* × *ncol* as specified by the two other arguments *nrow* and *ncol*. The matrix is filled row-wise, meaning that the *ncol* first elements of x are copied to the first row of the matrix, and so on row by row until the matrix is filled. Further macros are provided to simplify the set up of diagonal matrices (see `diag()`) or symmetrical matrices (see `smatrix()`).

```
in:  param_1 matrix(q(rep("1,2,3",3)), 3, 3)
out: param_1 {{1,2,3}{1,2,3}{1,2,3}}
```

```
in:  param_2 matrix(q(rep(q(seq(1,10,1)),1,each=10)),10,10)
out: param_2 {{1,1,1,1,1,1,1,1,1,1}
    {2,2,2,2,2,2,2,2,2,2}
    {3,3,3,3,3,3,3,3,3,3}
    {4,4,4,4,4,4,4,4,4,4}
    {5,5,5,5,5,5,5,5,5,5}
    {6,6,6,6,6,6,6,6,6,6}
    {7,7,7,7,7,7,7,7,7,7}
    {8,8,8,8,8,8,8,8,8,8}
    {9,9,9,9,9,9,9,9,9,9}
    {10,10,10,10,10,10,10,10,10,10}}
```

3.3.7 diag(): create diagonal matrix

`diag(x,n)` **:** Creates a diagonal matrix with zeroes off the diagonal. The argument x is either a single number repeated n times, n being the size of the (square) matrix, or x is a quoted list of numbers, copied to the diagonal of the matrix, from which the number n is deduced.

```
in:  param_1 diag(1,3)
out: param_1 {{1,0,0}{0,1,0}{0,0,1}}
```

```
in:  param_2 diag(q(rep(1,3)))
out: param_2 {{1,0,0}{0,1,0}{0,0,1}}
```

```
in:  param_3 diag("1,5,12")
out: param_3 {{1,0,0}{0,5,0}{0,0,12}}
```

3.3.8 smatrix(): create symmetrical matrix

`smatrix(x, nrow, diag=0) :` Creates a symmetrical matrix by copying or repeating elements of x to the $nrow \times nrow$ matrix, where x is provided as a single number or a quoted list of numbers. The diagonal is set to zero by default unless the *diag* argument is provided. The *diag* argument is also either a list of *nrow* numbers or a single number. The upper-triangle is set row by row. The number of elements in x must be exactly $nrow * (nrow - 1)/2$ or 1.

```
in:  param_1 smatrix(0.05, 3, 0.25)
out: param_1 {{0.25,0.05,0.05}{0.05,0.25,0.05}{0.05,0.05,0.25}}
```

```
in:  param_2 smatrix(q(seq(1,45,1)), 10, diag=-10)
out: param_2 {{-10,1,2,3,4,5,6,7,8,9}
    {1,-10,10,11,12,13,14,15,16,17}
    {2,10,-10,18,19,20,21,22,23,24}
    {3,11,18,-10,25,26,27,28,29,30}
    {4,12,19,25,-10,31,32,33,34,35}
    {5,13,20,26,31,-10,36,37,38,39}
    {6,14,21,27,32,36,-10,40,41,42}
    {7,15,22,28,33,37,40,-10,43,44}
    {8,16,23,29,34,38,41,43,-10,45}
    {9,17,24,30,35,39,42,44,45,-10}}
```

3.3.9 random distributions: runif(), rnorm(), rlognorm(), rpois(), rbernoul(), rexp(), rgamma()

```
runif(n, min=0, max=1, sep=",")
rnorm(n, mean=0, sd=1, sep=",")
rpois(n, mean=1, sep=",")
rbernoul(n, p=0.5, sep=",")
rexp(n, mean=1, sep=",")
rgamma(n, a, b, sep=",")
rlognorm(n, mean, sd, sep=",")
```

These macros can be used to generate a list of random numbers passed to a parameter. The list is comma-separated by default, as set by the `sep=","` optional argument. The number of random deviates to generate is specified with first argument n.

Each function has a set of additional arguments depending on the distribution used to draw the deviates from. The Table 3.1 below summarizes the macros and their distribution.

Each macro has at least one mandatory argument n to specify the number of deviates returned. The arguments with an assignment `=` in the macro syntax summary shown above are optional, and must be named when called, as for instance in: `runif(10, min=10, max=100)` or `rnorm(10, mean=2.5)`. Arguments without assignment are mandatory.

3.4 Matrix parameters

A matrix argument may be passed to a parameter in the init file. This type of argument contains integer or floating-point values separated by commas and curled brackets. Here is an example:

```
patch_capacity {{20, 20, 5, 10, 5}}

dispersal_matrix { {0.2, 0.0, 0.0, 0.4, 0.4}
                   {0.4, 0.2, 0.0, 0.0, 0.4}
                   {0.4, 0.4, 0.2, 0.0, 0.0}
                   {0.0, 0.4, 0.4, 0.2, 0.0}
                   {0.0, 0.0, 0.4, 0.4, 0.2} } \ #<- \ is mandatory!
\
                 { {0.4, 0.0, 0.0, 0.3, 0.3}
                   {0.3, 0.4, 0.0, 0.0, 0.3}
                   {0.3, 0.3, 0.4, 0.0, 0.0}
                   {0.0, 0.3, 0.3, 0.4, 0.0}
                   {0.0, 0.0, 0.3, 0.3, 0.4} }
```

The matrix is enclosed by two external brackets '{ }' within which each row is specified by two internal enclosing brackets '{ }'. Inside a row, the column values are separated by commas ',' or semi-colons ';'. The rows can be separated by any kind of characters but a backslash '\'. A matrix argument can as well be used to pass only an array of values as in the first example above or a complete matrix.

Several matrices may be passed as arguments to a parameter. That parameter will then become a sequential parameter (see below). The different matrices must start on the same line to be sequential arguments. The line continuation character '\' is mandatory if one wants to split matrices over several lines (see example above). Note that the lines within a matrix do not count; the rows can be written over several lines without using the line continuation character .

Table 3.1: Description of the arguments of the macros generating random deviates and their random distributions.

Macro	Distribution	Arguments	Description
`runif`	Uniform	`min=0`, `max=1`	$x \in [min, max)$
`rnorm`	Normal	`mean=0`: mean μ, `sd=1`: standard deviation σ	$x \in (-\infty, \infty)$; $f(x) = \frac{1}{\sqrt{2\pi\sigma}} \exp[\frac{-(x-\mu)^2}{2\sigma^2}]$
`rpois`	Poisson	`mean=1`: mean λ	$x > 0$, integer; $$f(x) = \lambda^{(x)} \cdot \frac{e^{-\lambda}}{x}$$ mean = λ, variance = λ
`rbernoul`	Bernoulli	`p=0.5`	$x = 0$ or 1: $$f(x) = \begin{cases} 1 \text{ if } r < p \\ 0 \text{ otherwise} \end{cases},$$ with $r \in [0, 1)$; mean = p, variance = $p(1-p)$
`rexp`	Exponential	`mean=1`: rate λ	$x \in [0, \infty)$, $$f(x) = \lambda e^{-\lambda x}$$ mean = $1/\lambda$, variance = $1/\lambda^2$
`rgamma`	Gamma	`a`: shape > 0, `b`: rate > 0	$x \in [0, \infty)$, $$f(x) = \frac{1}{\Gamma(a)} \gamma(a, bx)$$ mean = a/b, mode = $\frac{a-1}{b}$ for $a > 1$, variance = a/b^2
`rlognorm`	Log-normal	`mean`: μ, `sd`: σ	$x \in [0, \infty)$, $$f(x) = 0.5 \left[1 + erf\left(\frac{\ln x - \mu}{\sigma\sqrt{2}}\right)\right]$$ mean = $\exp(\mu + \sigma^2/2)$, mode = $\exp(\mu - \sigma^2)$, variance = $(\exp(\sigma^2) - 1)\exp(2\mu + \sigma^2)$

3.5 Sequential parameters

A parameter with several argument values *on a single line* is called a "sequential parameter" in the sense that it will initiate a sequence of simulations. There will be as many simulations as the number of combinations of the sequential argument values present in the configuration file. Each simulation receives a different output filename that might be explicitly defined in the configuration file or automatically numbered. This section explains how to specify specific simulation output filenames based on the sequential parameter values. This mechanism also works for other parameters than `filename` that have a character string as argument (e.g., the output directory or input source file names).

Example of sequential parameters:

```
patch_number 10 50
patch_capacity 5 10
```

Choosing two values per parameter as above would initiate a series of 4 simulations, each run after the other and each receiving a different combination of patch number and patch size values. To distinguish between simulations and not overwrite the output files, Nemo will automatically append a simulation number to the filename provided in the init file. For instance, with the example above, if the filename in the init file is set to `filename mysim`, the simulation file names would become: `mysim-1`, `mysim-2`, `mysim-3`, and `mysim-4`.

Alternatively, instead of the simulation number, the values assigned to the sequential parameters in each simulation can be added to the filename by using an *expansion string* of the form `%'...'#`, as explained in the next sections.

3.5.1 Basic string expansion

If your configuration file comprises sequential parameters, you may add the special expansion character `%` followed by a number (e.g., `%1`) in the base filename argument string to build specific filenames for each simulation initiated by the sequential parameters (see description of the `filename` parameter in section 4.1). This expansion character (`%`) can also be used in any string argument of any simulation parameter throughout the init file and will be expanded in the exact same way as for the base filename. The number after the expansion character refers to a specific sequential parameter present in the init file, starting with 1 for the first. The sequential parameters are alphabetically sorted so that the number one is not the first in the file but the first in alphabetical order. You cannot refer to more sequential parameters than the number present in the init file, but if you use less or none at all, a number will be automatically added to the simulation filename as explained above (this does not apply to other parameters with string arguments). In this case, the simulation base filename will get an extra extension of the form `-#` at its very end, where `#` stands for the number of the simulation in the set of combinations of argument values.

For the previous example above, setting the base filename and other parameters' string argument this way:

```
filename %2pop_%1ind
source_pop %2pop/mysource_%1ind
```

will give the following simulation file names, one for each simulation:

```
10pop_5ind
50pop_5ind
10pop_10ind
50pop_10ind
```

and different source file names:

```
10pop/mysource_5ind
50pop/mysource_5ind
10pop/mysource_10ind
50pop/mysource_10ind
```

Here `%2` refers to patch_number and `%1` refers to patch_capacity, in alphabetical order.

3.5.2 Specifying the replacement string

The system presented above works fine when the sequential arguments are numbers (even floating-point numbers) that can easily fit into a filename string. However, when the sequential argument is a matrix, or is too long to fit in, we also want to have a way to format or specify the replacement string explicitly. This is done by adding a **format string** within the expansion string. That string helps to set the format of the argument value (number of digits to use) or provides an alternative set of replacement strings.

The **format string** is enclosed within two single quotes `' '` and placed between the expansion character and the parameter identifier: `%' '#`. The format string must contain at least one integer number: e.g. `%'4'#`. That number specifies the length of the replacement string to insert into the parameter's argument string. For instance,

`%'4'1` means that the numerical values of sequential parameter #1 will be written on 4 characters with *leading* zeros. A value of `10` for that parameter will be written as `0010` in the replacement string.

A dot '`.`' in front of the mandatory size specifier (e.g., `'.3'`) indicates that only the decimal part of the argument value must be used to create the expansion string, with *trailing* zeros (up to three digits in this example). Here, with `%'.3'1` the value `0.1` would be expanded as `100` in the replacement string.

When the parameter value used to build the replacement string is not easily formatable or is not a single number, then it becomes necessary to provide a set of replacement characters to substitute for the actual parameter value. This is done by adding the specific set of characters to use for the replacement string after the size specifier and enclosed with square brackets `[ ]`, for instance: `%'2[AaAbAcBaBbBc]'1`. In this example, it is supposed that parameter no.1 has 6 sequential values, and we want the replacement string to be only 2 characters long (`Aa`, `Ab`, etc.).

Here is a full example:

```
filename Num%'4'1_Disp%'.3'2_%'2[AaAbAcBaBbBc]'3
my_seq_param_1 1 10 1500
my_seq_param_2 0.001 0.01 0.1
my_seq_param_3 {{matrix no.1}} {{matrix no.2}} ... {{matrix no.6}}
```

These settings will give the following simulation filenames (54 total):

```
Num0001_Disp001_Aa
Num0001_Disp001_Ab
...
Num1500_Disp100_Bc
```

A last option is to replace the character string by a + to replace the argument value by its position value: `%'1[+]'3`. In the example above, the third sequential parameter has 6 argument values, therefore the + stands for the integer values 1 to 6 and the width specifier is 1 (no leading 0). The values 1-6 would be used as replacement string in the filename for that parameter:

```
filename Num%'4'1_Disp%'.3'2_mat%'1[+]'3
```

```
Num0001_Disp001_mat1
Num0001_Disp001_mat2
...
Num1500_Disp100_mat6
```

Other example:

```
random_seed  %'1[+]'1%'2[+]'1
replicates   1 1 1 1 1
```

This would create five different values of the random seed, one for each replicate:

```
random_seed  101
random_seed  202
random_seed  303
random_seed  404
random_seed  505
```

You can thus use the same parameter identifier multiple times in the same init file and in the same parameter argument string:

```
patch_number 10 50
patch_capacity 5 10

source_pop N%'2'1/%2patch/burnin-N%'2'1-P%2
```

which results in 4 simulations with the following arguments to the `source_pop`:

```
N05/10patch/burnin-N05-P10
N05/50patch/burnin-N05-P50
N10/10patch/burnin-N10-P10
N10/50patch/burnin-N10-P50
```

The number of simulations initiated by sequential parameters is equal to the product of the number of arguments of each sequential parameter. All the parameters value combinations are performed. There is currently no way to restrict the number of combinations.

3.6 External argument files

It is sometimes convenient to write large matrices, or large numbers of sequential parameter arguments in a separate text file and only specify the path to such file(s) in the init file. This is done by providing the path to the file with the '`&filename`' syntax:

- `filename` is a character string that contains the path to the external file, *relative to the directory from which* NEMO *is run.*
- More than one external file can be provided in argument to a parameter, in which case the parameter becomes a sequential parameter.
- The expansion character '`%`' can also be used in the `filename` character string.

Example:

```
param0 1 2 3
param1 &filename1.txt &filename2.txt &filename3.txt
param2 &path-%1/to/file%'1[ABC]'2.txt
```

Here, param1 and param2 have argument values stored in external files. The filename and the directory path to param2 depend on the argument value of param0 and param1 (i.e., `path-1/to/fileA.txt; path-2/to/fileB.txt;` etc.)

NOTE: the external file must be terminated by an empty line. Otherwise, it just needs to hold the argument(s) of a given parameter in exactly the same form as if it were written in the init file (i.e., without new lines between multiple arguments).

3.7 Temporal arguments

NEMO offers the possibility to change the value of a parameter during the course of a simulation and thus to modify the state of the population or of any particular component during a simulation. Temporal arguments are limited to the **non-trait components** for now. They are specified in the init file by using the temporal argument specifier "`@g#`" within the argument string, where the `#` stands for the generation at which the argument value has to be used. The state of the components that have temporal arguments is updated before the first event in the life cycle. Temporal argument string *must* always start with the initial argument value, specified as "`@g0`" and arguments are separated by commas:

```
param1 (@g0 value1, @g100 value2, @g10000 value3)
```

This example specifies three different parameter values that will be used throughout the simulation; '`value1`' is used at initialization of the simulation (and beginning of each replicate), '`value2`' and '`value3`' are used at generation 100 and 10 000, respectively. The component that declares and uses '`param1`' will update itself at the specified generations. Temporal parameters can thus be used to dynamically modify the state of the population through time to model population fragmentation or bottlenecks, for instance.

The following example shows how to progressively fragment a population while keeping its total size at 10 000 and number of migrants at 1.

```
patch_number (@g0 10, @g5000 15, @g10000 20)
patch_capacity (@g0 1000, @g5000 666, @g10000 500)
dispersal_rate (@g0 0.001, @g5000 0.0015, @g10000 0.002)
```

Important Note: Changing the number of patches during a simulation can lead to various problems at runtime as many features depend on it. For instance, the number of patch-specific stats cannot be updated (this would cause a lot of mess in the stat output files) and thus data will not be recorded for the added patches (they will be set to 0 or NaN otherwise). The size of the dispersal matrix also depends on the number of patches and cannot be automatically updated when specified in input. In that case, an error message is issued and the simulation is aborted. The best workaround is to set the number of patches constant from the start but set the initial carrying capacity of unwanted patches to 0 before adding them at a latter generation by increasing their carrying capacity, and for instance, updating the dispersal matrix at the same time.

3.8 Output files and naming conventions

As briefly explained in the previous section, the output files of a simulation have a common base name. That name is taken from the argument of the parameter filename (see section 4.1). If multiple simulations are specified with sequential parameters in the init file, the filename is modified as described in section 3.5 by substituting the expansion strings with their corresponding parameter value. The base filename then receives different suffixes and dot-extensions depending on the kind of files the simulation produces.

Counter extensions: A first kind of extension is the generation or replicate number, or both depending on the periodicity of the output. That extension start with an underscore "_" and is followed by a number "002". The number of digit depends on the maximum number of generations or replicates in the simulation. For instance, if a file is written every replicate and the simulation has 100 replicates, the counter will be made of three digits. The same is true for the generation counter. When both counters are added to the filename, the generation counter precedes the replicate counter and each start with an underscore like this:

```
mysim_1000_01
mysim_2000_01
...
mysim_5000_10
```

This way, the simulation can save each generation for each replicate in a different file. The behavior of the various output files (i.e., their periodicity) depends on the kind of data the simulation will generate, which depends on the user's defined parameters. Typically, trait genotype files are written per generation and per replicate, while binary output files are per replicate only.

Type extension: The second kind of extension string is the file type, or dot-extension (e.g. '.txt') and is a classical extension starting with a dot followed by a few characters added to the end of the file name. Nemo generates a few basic output files with different types. These are the:

".log": these files are automatically generated in every folder a simulation will create and contain all the input parameters of that simulation. One extra log-file is also created in the working directory but with a different base filename that can be specified by the "logfile" parameter (called "nemo.log" by default, see section 4.1) and that will store some runtime information about the simulations done. No replicate or generation counter is added to these files.

".txt": these files contain the statistics computed by a simulation and are created only when the simulation is asked to (see section 5.15). These files don't add any counter string to their filenames.

".bin": these files contain the complete set of individual data for each replicate of a simulation. Their filename thus contain the replicate counter appended after the base filename. See section 5.17 for more details about the binary output files and how they are handled.

".freq", ".quanti", ".delet", etc. : each component (especially traits) define their own output files and extensions, making it clearer what data is recorded in which file. See the next chapters for details.

Important Note: To make sure that the file manager of Nemo notifies the different simulation components at time of saving, you *must* include the `save_files` life cycle event (see section 5.16) in the life cycle, otherwise no files will be written during a simulation. See chapter 5 to understand how this is done. In absence of this life cycle event, only one type of file is automatically written during a simulation, this is the ".log" simulation file holding the simulation parameters with some info about the simulation run (value of the seed of the random generation, elapsed time and CPU time used).

Chapter 4

Simulation Components

This chapter presents the various simulation components and their parameters. It is through these parameters that you can select which components are part of a simulation or not. Two components are mandatory, the simulation and population components. Besides these two, it would make sense to select at least a basic sequence of life cycle events to run a basic simulation. Note that you can also use NEMO to simply load a previously saved population from a binary file (see the source_pop population parameter below) and compute statistics on it or extract genotypes and save them in a human-understandable format (usually text...).

Each component and its list of parameters are presented here. Some parameters are mandatory; they must be present in the init file in order to include a component to a simulation. Each component has at least one mandatory parameter. Optional parameters are marked as **(opt)** below and are used to add extra features needed to build a particular model. NEMO will not complain if a mandatory parameter is missing for a non-mandatory component (i.e., others than the simulation and population components) so you have to be careful while building the init file. The parameter type is given between two enclosing square-brackets '`[ ]`', see chapter 3 for details about the different types of parameters.

There are two main types of simulation components; the Traits (chapter 6) and the Life Cycle Events (chapter 5). The traits are carried by the individuals in the population while the LCEs act as modifiers of the population state, and hence act on the individuals state as well, as defined by their traits' state. The action of an LCE may depend on the values of the individual's traits or not. For instance, selection will remove individuals by checking the phenotype of their fitness trait against a fitness function, or aging will remove all adult individuals independently of their traits' value to make room for the new generation.

The simulation components can also declare different output files and statistics. The file extensions and stat outputs are indicated for each component. For a discussion and a complete list of output statistics, have a look at chapter 8.

4.1 Simulation

name: **simulation**
files: `.log`
stats: NA

replicates [integer]

Number of replicates to perform per simulation.

generations [integer]

Number of generations performed per replicate. One generation is one iteration of the life cycle.

filename [string]

This name will be used as the base filename of all output files of a simulation. The output file extensions are added to this base filename by the different simulation components that write data to files. If a file is written on a replicate-periodic basis, the replicate number will be added between the basename and the extension, so that the same file is not overwritten periodically. The same is true concerning generation-periodic files (see section 3.8).

The base name may include the special expansion character '%' used to build filenames when sequential parameters are present in the input parameter file. See the discussion on sequential parameters in section 3.5.

root_dir [string] (opt)

The path specified by this parameter will be used as the root directory path for all output files and directories declared by the simulation components. This path will thus be added in the front of any other paths defined subsequently (e.g., by param `stat_dir`).

run_mode [string] (opt)

This sets the simulation behavior, with the following options:

overwrite : previously saved files with the same base filename as the current one are overwritten. A warning is issued on the standard output (i.e. terminal window).

dryrun / dry : does not run the simulation but just sets the parameters and checks for the files and statistics. The output paths and log files are created.

create_init : similar to 'dryrun', but writes the parameters of each possible simulation in a separate init file in the working directory. This is handy when wishing to create many init files from a single one containing many sequential parameters.

skip : automatically skips simulations (do not run) whose base filename already exists on disk.

run : (default) the default running mode.

silent_run / silent: turns off all regular and warning messages, only the error messages are issued.

logfile [string] (opt)

This is the file in which the simulation logs are recorded. The simulation basename and each directory paths are recorded as well as the mean elapsed times for the simulation and the replicates and the dates of beginning and end of a simulation. By default, Nemo will save all this information in a file named "nemo.log" in its working directory.

random_seed [integer] (opt)

The seed of the random generator can be specified with this parameter. The upper value is system-dependent but should not be more than 4,294,967,295 on a Mac. By default, the random seed is set by the clock time of the computer (i.e. number of seconds since an arbitrary date in the past, usually around the 1970's).

postexec_script [string] (opt)

This parameter is used to specify the path to a shell script that will be executed once all the simulations have been processed. The script will be executed using a system call with the following command:

```
> sh my_script.sh
```

postexec_args [string] (opt) This parameter is used to add an argument to the above script when executing it. More than one argument can be passed to this parameter without causing Nemo to build multiple simulations out of them, as it would for the other parameters (see section 3.5). The expansion character '%' can be included in the arguments and will be expanded. Therefore, simulation specific arguments can by automatically built. That feature is mainly useful when scripts are executed replicate-wise. The script will be executed with the following command, with the argument added after the script name.

```
(in the init file:)
postexec_script my_script.sh
postexec_args arg1 arg2 arg3
```

```
(command executed:)
> sh my_script.sh arg1 arg2 arg3
```

postexec_replicate_wise [**bool**] **(opt)** If set (true), this parameter will cause the postexec script to be executed after every replicate of a simulation, instead after all simulations included in a single call to Nemo. Two more arguments are added to the command executed: the base filename of the simulation and the replicate number, before the user-specific arguments.

```
(in the init file:)
filename  mySim-10loci-10pop
postexec_script my_script.sh
postexec_args arg1 arg2 arg3
postexec_replicate_wise
```

```
(command executed after replicate 1:)
> sh my_script.sh mySim-10loci-10pop 1 arg1 arg2 arg3
```

4.2 Population

name: **population**
files: `.ped` (output of individual pedigree information)
stats: `pop, demography, migrants, kinship`, and more (see chapter 8)

patch_number [**integer**] **(opt)**

Number of patches in the population.

patch_capacity [**integer/matrix**] **(opt)**

Carrying capacity of each patch (K), this is the number of males and females. If given as a unique value, all the patches have the same size with equal numbers of males and females. May also be given as a matrix parameter containing the vector of the patches size. In that case, the length of the vector will give the number of patches in the population.

patch_nbfem/patch_nbmal [**integer/matrix**] **(opt)**

The number of males or females per patch can be given separately with these two optional parameters. Each or both of them can be a matrix parameter giving the sex-specific sizes of each patches. If one of the two sex-specific size parameters is missing, population initialization will abort.

Examples : The following setting will build a population of 5 patches of different sizes but with equal sex-ratio in each patch:

```
patch_capacity {{10, 4, 18, 20, 24}}
```

This parameter is sufficient to build a population as the size of the vector will tell the number of patches present. In this other example however, the number of patches must be given explicitly as no matrix arguments are present:

```
patch_number 5
patch_nbfem 8
patch_nbmal 4
```

This other example will also work fine:

```
patch_nbfem 5
patch_nbmal {{4, 4, 3, 3, 1}}
```

Note however that the following will issue a fatal error:

```
patch_capacity 10
patch_nbmal {{4, 4, 3, 3, 1}}
```

Indeed, `patch_capacity` has precedence over `patch_nbmal` and in that case, `patch_number` is missing. The correct form would be:

```
patch_nbfem {{6, 6, 7, 7, 9}}
patch_nbmal {{4, 4, 3, 3, 1}}
```

This also means that including both `patch_capacity` and the sex-specific size parameters will cause Nemo to ignore the later and use only the first one to build the population.

4.2.1 Saving the population pedigree

With the following parameters, a file can be periodically written to disc with demographic and pedigree information of all or a sample of the individuals present in the population. The information written for each individual is: ID (unique integer), dad (ID of the father), mum (ID of the mother), sex (0=male, 1=female), age (0=offspring, 2=adult), home (natal patch), patch (residence patch at time of sampling). The pedigree information can thus be gathered over two generations if the offspring and the parents are present at the time of sampling (depends on where the save_stats LCE is placed in the life cycle, see below). Note that a pedigree file containing only the first three columns can be used in input to create a population. See the cross life cycle event. This is different from using the set of parameters below in subsection 4.2.2.

pop_output [bool] (opt)

If present, will indicate that an output file must be written.

pop_output_dir [string] (opt)

If present, specifies where the output file should be written, relative to the root directory of the simulation.

pop_output_logtime [integer] (opt)

Indicates how often a pedigree file is written to disc. Must be specified if pop_output is present.

pop_output_sample_size [integer] (opt)

Indicates the maximum number of individuals of a given age class and sex in each patch of the population whose information will be recorded in the output file. For example, if this is 10 and offspring and adult males and females are present in each of 10 patches, then ($4 \times 10 \times 10 =$) 400 individuals will be sampled in total.

4.2.2 Loading a population from a file

This section describes the set of parameters needed to load a population from data saved on disc in a text or a binary file. The type of data that can be loaded depends on the file format. Binary data (non-text) is used to save the whole population data for all individuals and traits present in the simulation. The binary files are written by the store component (see 5.17) and are typically used as backup to then reload a whole population, for instance after a burn-in phase. Text files can also be written to disc with different kinds of individual or trait data depending on the simulation component. For instance, the ntrl and delet traits can save the genotypes of all individuals in text files (human-readable) and then use them as input to create a new population (see respective trait's description for details about those files).

When loading a population from a *binary* source file, you have to make sure that the parameters of the current (loading) simulation match with the trait and genetic parameters of the source simulation. Nemo will not be able to load the source population if the number of loci of the traits or the traits differ between the source and the current simulation. Otherwise, the population structure may change (i.e., the receiving and source population may have different numbers of patches and total individuals), and all other trait parameters can change (e.g., mutation and recombination rates). The life cycle events are not "storable" components and thus do not save data to the binary files, however, their parameters are saved in the binary file.

When loading a population from a *trait* source file in text mode (e.g., from an FSTAT file for the neutral markers), the individuals in the receiving population need only to carry the corresponding trait. Here too, the trait's number of loci must match. However, the genetic (recombination) maps may differ, simply because the loci positions are usually not stored in a trait file.

The loading process is described below under the source_preserve parameter description.

Filling the population: The population loaded is used to set the starting generation of a replicate. Each replicate may start from a different source replicate file, or from a single source file (see below). The default loading mode *randomly draws* individuals from that source population *without replacement* to fill the current population. The two populations may thus have different sizes. Unless the source population is loaded in `preserve` mode (see below), the structure of the source population is not preserved, all individuals in the different patches are pooled together.

Filling age class: The age class (offspring or adults, or both) used when loading a population depends on the individuals available in the source file and the age class required by the life cycle events of the current simulation. The class to load is determined by finding the first event in the current life cycle that requires a specific age class (see chapter 5 on life cycle events). NEMO then tries to load that class from the source file. Independently of the loading mode, if that required age class is not available in the source population, the alternate one is used instead (i.e. offspring for adults, and vice versa). A warning message is displayed if that case happens.

Using compressed binary files: Finally, when loading populations from binary files, NEMO will automatically check whether the binary fill is compressed. If so, NEMO will decompress it, read it, and recompress it. Files saved in an archive will however not be extracted. This feature is only possible if one of the two default compress formats is used (.gz or .bz2) and the corresponding programs are available on the system.

Parameters description:

source_pop [string] (opt)

The path to the simulation source file and the name of the file is given by this parameter. If the replicates of the current simulation use the same binary source file, the full name of the source file must be specified, i.e., including replicate (or `generation_replicate`) counters and extension strings (`.bin` or `.txt`, etc.) If the source population is to change during a simulation, the path given here must only contain the base filename of the source files, without any replicate/generation counter string and file extension. In that case, the replicate-specific file name will be built from the information provided by the parameters source_replicates, source_replicate_digits, source_start_at_replicate and source_file_type below. See the examples at the end of this section.

Only one file can be used for a given simulation or replicate. If multiple file names are provided, separated by space characters, then Nemo will run different simulations for each source population provided.

The path/filename of the source file may contain the expansion character and format strings (i.e., in the form of `%'pattern'#`, see chapter 3) to match the sequential parameter arguments values defined in the current configuration file.

source_file_type [string] (opt)

The argument here is the file extension string of the source file, including the dot (e.g. ".bin", ".txt", etc.). This determines how the source data is imported. The default is ".bin" for binary files saved with the store LCE.

source_preserve [bool] (opt)

With this parameter, the individuals from the source populations are sequentially imported into the receiving population. That means that the structure of the source population will be preserved if the source and the receiver have the same patch structure (same size and number). If the source population has less patches than the receiving population, then the receiving population will have empty patches. Similarly, if the source population does not contain enough individuals within patches to fill the receiving patches to their carrying capacity then the receiving population will not be full. No extra patches are added to match the number of patches in the source population. However, for now, all individuals from the source patches are imported into the receiving patches, irrespective of the carrying capacities of the receiving patches.

If this parameter is not present in the input file, then the individuals of the source population are gathered together in a single container from which they are randomly sampled without replacement until the receiving population is full or the source population is empty.

source_fill_age_class [adults, offspring] (opt)

This sets the age class to load from the source population. It overrides the rule described above using the required age class of the life cycle events of the current life cycle.

source_replicates [integer] (opt)

This parameter tells NEMO how many replicates of the source population have to be used during the simulation to load the population from. If the value given here matches the number of replicates of the current simulation (see replicates above), each replicate will use a different source file as a source population. In the case this value is smaller than the number of replicates of the current simulation, the source population will be changed periodically, every [replicates / source_replicates] replicates. The same source population may thus be used several times. The source filename is built from the value of the source_pop parameter to which the replicate counter and the file extension are added. Therefore, the source_pop parameter string value must not include these character strings. The replicate counter is built using the digit information given below by the source_replicate_digit parameter. By default, the counter starts at 1. This can be changed to start at a different replicate number with parameter source_start_at_replicate.

By default, if parameter source_replicates is not specified, NEMO will load the source population only once and use it for all replicates of the simulation. In that case, the replicate counter is not added to the filename specified with source_pop.

source_replicate_digit [integer] (opt)

This parameter is needed build the replicate counter of the binary source filename when the parameter source_replicates is specified. Its value must match the number of digits used in the replicate counter of the source filenames. For instance, it is 3 if one of the source filenames ends with, say, `'_032.bin'`.

source_start_at_replicate [integer] (opt)

The first replicate to load data from can be set using that parameter. The rules described above to set the replicate number applies but start at the value set here rather than 1.

source_parameter_override [string] (opt)

Use this parameter to specify the name of parameters that will be set from the parameter values (or component information) read in the source file. It thus works only with binary files. Multiple parameters can be specified if enclosed within brackets as (arg1, arg2, arg3, ...). This is useful, for instance, if values of a parameter or feature were set randomly in the source simulation and the same values need to be used in the new simulation. An example would be the fitness effects of deleterious mutations or the map positions of a set of loci when a random genetic map was used.

Example 1:

The first example shows how to load a population from a single binary source file in preserve mode:

```
replicates 10
source_pop binarydir/mysourcepop_001.bin
source_preserve
```

Here, the source population stored in the file named mysourcepop_001.bin is loaded once and used by each replicate of the simulation.

Example 2:

A different source population can be used for each replicate if we specify the following set of parameters (assuming the number of replicates match):

```
replicates 10
source_pop binarydir/mysourcepop
source_preserve
source_replicates 10
source_replicate_digit 3
```

In that second example, each replicate loads a population from a different source file which name is constructed according to the parameters specified above (esp. digit number for the replicate counter): mysourcepop_001.bin for replicate 1, mysourcepop_002.bin for replicate 2, etc.

Example 3:

If the simulation has more replicates than the number of source files, a single source can be re-used by multiple replicates. For instance, if we ran a burn-in simulation with 25 replicates and now want to run 100 replicates from those burn-ins, then the source population will need to be changed every four replicates in the new simulation. This is done automatically when the number of source replicates specified with source_replicates is smaller than the number of replicates in the current simulation. The parameters below illustrate this case values below. The source file names are then `binarydir/mysourcepop_01.bin` for replicates 1 to 4, `binarydir/mysourcepop_02.bin` for replicates 5 to 8, and so on until file `binarydir/mysourcepop_25.bin` for replicates 97 to 100.

```
replicates 100
source_pop binarydir/mysourcepop
source_preserve
source_replicates 25
source_replicate_digit 2
```

Example 4:

Finally, loading a population from a trait file is also possible. This can be done from a single or different files, depending on the type of data. The simulation parameters should match the data structure in the source file for optimality. The following example loads neutral markers data (e.g. from a field study) from a single FSTAT file (see section 6.2 for more details) and use it to compute the F-statistics available in Nemo:

```
replicates 1
generations 1

patch_number 5
patch_capacity 50

source_pop source/path/srce-fstat-file.txt
source_preserve
source_file_type .txt
source_fill_age_class adults

## LIFE CYCLE ##
save_stats 1
save_files 2

stat adlt.fstat adlt.fstWC adlt.weighted.fst
stat_log_time 1
stat_dir stat

## NEUTRAL MARKERS ##
ntrl_loci 20         #must match the number of loci in the file
ntrl_all 10          #same for the number of alleles
ntrl_mutation_rate 0 #useless here, but mandatory parameter
```

Chapter 5

Life Cycle Events

The life cycle events (hereafter LCE) are operators used to modify the state of the population and interact with the different components of a simulation. Each LCE is executed only once during the course of a generation, at the rank it has been assigned in the stack of LCEs that constitutes the life cycle. This rank is given by the user in the init file. The life cycle is thus an ordered list of LCEs selected by the user. Most LCEs act on a per generation basis. Some may however have a different periodicity set by the parameters they declare.

The ranks should start with value one for the first LCE and be incremented for each successive LCE. As the LCEs are placed in ascending order in the life cycle, their exact rank value does not matter so much as long as the order is conserved (i.e. the rank increment may be different from one). If two LCEs have same rank, one of these two is replaced by the other (usually following an alphabetical order). As each parameter may appear only once in the init file, each LCE must be given only one rank value. Giving several values to a LCE will make it a sequential parameter.

The way to build the life cycle in the init file is to write the LCEs names (given below) followed by their rank number. Here is an example (see chapter 7 for more details):

```
breed 1
save_stats 2
save_files 3
disperse 4
viability_selection 5
aging 6
```

This very simple life cycle starts with mating and breeding within the population that will generate a new offspring generation provided adults are present within patches. The statistics are then recorded and the simulation data is saved, at the right generation. Because the save_stats LCE is placed after breed, the data on both the offspring and adult individuals can be recorded. This wouldn't be the case if it was placed after aging where only the stats on the adults would be recorded, for instance. The disperse LCE then moves the offspring around according to the migration model chosen. The offspring then experience a round of viability selection within their patches where their survival probability is determined by the phenotypic value of the viability trait they carry. They are then moved to the adult age class, previously emptied of its previous occupants from the previous generation by the aging LCE. And the cycle starts again.

The Life Cycle Events described here are:

aging: increase the age of the individuals, perform patch regulation

breed: mate and breed, create new offspring generation

breed_wolbachia: breed and *Wolbachia* transmission/infection

breed_disperse: breed with backward migration (Wright-Fisher model)

breed_selection: breed with selection (faster)

breed_selection_disperse: all in one (Wright-Fisher with selection)

cross: perform a half-sib, full-sib mating design (NCI)

disperse: offspring dispersal

disperse_evoldisp: offspring dispersal with evolving dispersal rates

extinction: random patch extinction or harvesting

regulation: patch regulation (to carrying capacity)

resize: modify population size (patch number and/or size)

save_files: write output files to disk

save_stats: record statistics

selection: perform viability selection on the offspring generation

store: save simulation data to binary files

The LCEs often act as modifiers of the population state. Most of the time, this simply consists of changing the content of various individual containers either by moving individuals between them or by adding/removing individuals to/from them. Individual containers are ordered by age class and by sex and are aggregated within patches. The two main age classes are the adult and the offspring age classes. A particular LCE will in general be associated with one or more age class. This information is given below by the age flag values associated with each LCE (see Table 5.1). These age flags tell which individual container will actually contain individuals after having executed the corresponding LCE during the life cycle and which age class is needed by an LCE. This will help you design a proper life cycle.

Table 5.1: Modification of the population age state caused by the LCEs in the basic life cycle. (+) means that age class is added to the population by the LCE while (−) means the LCE will remove all individuals of that age from the population. (x) means the LCE will modify the state of that age class. *required* means that age class is the required age class for the LCE, and will be loaded first whenever that LCE begins the life cycle.

LCE	Offspring	Adults
aging	*move to adults*	−
breed	+	*required*
cross	+	*required*
disperse	x *(required)*	
extinction	x	x
regulation	x	x
resize	x	x
selection	x *(required)*	

5.1 Aging

LCE name: **aging**
age flags: removes the **offspring** flag
files: NA

aging moves *all* individuals from their age class to the next and performs patch regulation at the same time. For now, only two age classes are present, the offspring and the adults. Therefore, `aging` moves the offspring to the adults age class and all the adults are removed, they die. No other LCE removes the adults from the population. It is thus very important to add this LCE to the life cycle. For each patch, the offspring individuals are randomly chosen to fill the adult containers until the patch carrying capacity is reached. The process is performed for males and females separately (using the sex specific carrying capacities). This form of density-dependent regulation is called "ceiling" regulation. The patches will be at their carrying capacity only if enough offspring were present before aging. **Note:** Be careful about its position in the life cycle. If, for instance, placed before `disperse`, no offspring will be able to migrate in the population as they already aged.

The regulation LCE (section 5.14) also performs "ceiling" regulation but on each age class separately, thus keeping the offspring and adults in the population.

Parameters:

aging **[integer]**

The LCE name, which must be added to the ini file to use this LCE in the life cycle. The integer value is the rank in the life cycle.

5.2 Breeding

LCE name: **breed**
age flags: **adults** (required) and **offspring** (added)
files: NA
derived components: breed_wolbachia, breed_disperse, breed_selection, breed_selection_disperse

Performs mating and breeding of the new offspring generation following the mating system chosen. Adults are not removed here (see `aging` above). The number of offspring per female depends on the mean fecundity set by `mean_fecundity` below and may be a fixed number or a number drawn from different random distributions. The default distribution is Poisson. Wright-Fisher populations can also be modeled with the `mating_isWrightFisher` option, in which case exactly K offspring are produced in a patch by randomly drawing the parents of each offspring in the patch following the mating system chosen.

Parameters:

breed [integer]

The LCE name, which must be added to the ini file to use this LCE in the life cycle. The integer value is the rank in the life cycle.

mating_system [1 to 6]

Six mating systems are implemented in Nemo. The options are:

1 : promiscuity/random mating. One male is randomly chosen for each new offspring a female does.

2 : polygyny. One male only mates with all females in the patch. This can be changed by setting mating_proportion to a value < 1 in which case one male will monopolise a proportion equal to *mating_proportion* of the matings within a patch while the remaining matings are shared by all other males. The number of mating males may also be changed below with the mating_males parameter in which case the mating male for a given female is randomly chosen within the *mating_males* first males of a patch.

3 : monogamy. Each female mates with one male only and vice versa. If the number of males is less than that of females, some males will mate with more than one female. In the reverse case however, if there are more males than females, some males will not reproduce at all. A given proportion of random mating can be achieved by setting the mating_proportion parameter to a value < 1. Each female will then have on average a proportion of $1 - mating\_proportion$ of its offspring born from a random male in the population.

4 : selfing/hermaphrodite. Only females are used in that case. If mating_proportion $= 1$ all offspring are produced by self-fertilisation, otherwise, a proportion of $1 - mating\_proportion$ of the offspring are produced by randomly crossing two "females" together. Note that selfing is enforced in patches with a single reproducer even if mating_proportion $\neq 1$.

5 : cloning. Equivalent to selfing but without recombination. Individuals are produced by first copying the "mother's" genes and then computing mutations. The mating_proportion parameter is used in the same way as under selfing.

6 : random mating with selfing This corresponds to what is called the Wright-Fisher model where individuals may self with probability 1/N (N = patch size). The individuals are considered hermaphrodites here, that is *only the females are used*. It is then best to set the patch capacity specifically for the females with `patch_nbfem` and to zero for males (`patch_nbmal 0`). Also, a solution to limit selfing of hermaphrodite females is to use the selfing mating system (4) and set mating_proportion close or equal to zero.

mean_fecundity [integer] (opt)

Mean of the distribution used to set the females fecundity. It is used whatever the mating system selected but not when the population is a Wright-Fisher population (see above).

mating_proportion [decimal] (opt)

This parameter is used to set the proportion of random mating in the polygyny and monogamy mating systems, and the selfing rate for the selfing case. See the mating systems description above for more details. The actual proportion of random mating will be $1 - mating\_proportion$ on average. This can be used to set the degree of extra-pair mating when monogamy is modelled, for instance, or limit selfing when individuals are hermaphrodites.

mating_isWrightFisher [bool] (opt)

Option to set the population to breed as in a Wright-Fisher (WF) population. Breeding in WF populations produces exactly K individuals, if K is the carrying capacity (set with `patch_capacity`). Mating in a truly WF population is at random, among hermaphroditic individuals capable of self-fertilization (i.e., with effective selfing rate of $1/K$). This can be obtained by choosing `mating_system 6`. As the number of offspring produced does not depend on a female specific fecundity, the parameter `mean_fecundity` is not required for WF mating. Note that NEMO allows individuals in WF populations to also mate according to the other mating systems offered with `mating_system`. Demographic stochasticity is suppressed in WF breeding populations because exactly enough offspring are produced to fill the patches to their K. This also makes simulations faster than when the mean fecundity is set to > 2. Demographic stochasticity may still exist if `breed` is followed by `disperse` or `selection`. WF populations with dispersal are modeled with the `breed_disperse` LCE. WF populations with dispersal and selection are modeled with the `breed_selection_disperse` LCE.

fecundity_distribution [fixed, poisson, normal, lognormal] (opt)

The distribution used to set the females fecundity. Is Poisson by default. The "fixed" option sets the fecundity of each female equal to the mean (see `mean_fecundity` above).

fecundity_dist_stdev [decimal] (opt)

Standard deviation used in case the fecundity distribution is set to "normal" or "log-normal".

mating_males [integer] (opt)

This parameter sets the number of males that will be available for mating within each patch (*under polygyny only!*). The value given in argument should be equal to or smaller than the male's carrying capacity. Setting it to the carrying capacity is equivalent to setting the mating system to monogamy.

sex_ratio_mode [fixed, random] (opt)

By default, the sex of an offspring is randomly set (unless the individuals are considered hermaphrodites) and thus the offspring sex-ratio usually varies from one generation to the next. The "fixed" option sets the sex-ratio to exactly 1:1.

5.3 Breeding with Wolbachia

LCE name: **breed_wolbachia**
age flags: **adults** (required) and **offspring** (added)
files: NA
inherits from: breed

This is also a derivative of the first breeding LCE, it thus inherits the previous parameters and defines several parameters for the simulation of *Wolbachia* infections. See the Wolbachia trait for more details.

Parameters:

breed_wolbachia [integer]

The LCE name, which must be added to the ini file to use this LCE in the life cycle. The integer value is the rank in the life cycle.

wolbachia_fecundity_cost [decimal]

The fecundity of an infected female (as specified by parameter *mean_fecundity*) is reduced by an amount of $1 - s_f$, s_f being the cost to pay when infected by *Wolbachia*.

wolbachia_incompatibility_cost [decimal]

A zygote issued from a infected male gamete and an uninfected female gamete must pay the cost of cytoplasmic incompatibility caused by the parasite. This cost is the amount of reduction in the survival probability of the offspring.

wolbachia_inoculum_size [integer]

Wolbachia can be inoculated to a specified number of adults specified by this parameter. This number represents the number of females and the same number of males that will be inoculated in one deme of the population, randomly.

wolbachia_inoculum_time [integer]

Generation at which the population will be infected with *Wolbachia*.

5.4 Dispersal

LCE name: **disperse**
age flags: **offspring** (required)
files: NA
derived components: disperse_evoldisp, breed_disperse, breed_selection_disperse

The disperse LCE moves offspring among patches according to the migration scheme chosen. Dispersal rates are taken as forward migration rates, that is they represent the probability of an individual to move from patch i to patch j. These rates will be equivalent to immigration rates under the classical models of island model migration and stepping stone migration. Forward migration is equivalent to zygotic (diploid) migration, as opposed to backward migration modelled by the breed_disperse LCE as gametic (haploid) migration (*e.g.,* pollen dispersal).

There are three *mutually exclusive* ways of specifying the migration rates in Nemo: ***i)*** by specifying a (sex-specific) dispersal **rate** and migration **model** (e.g., Island Model, Stepping Stone model, etc.) as described in subsection 5.4.1, ***ii)*** by specifying the full **migration matrix**, which allows for more flexibility in the type of migration modelled (e.g., allowing for long-distance dispersal on a landscape), refer to subsection 5.4.2, ***iii)*** by specifying the **reduced migration matrices**, which holds the non-zero migration rates only, and allows the modelling of large landscapes with sparse dispersal matrices. This last option is an optimisation of the dispersal_matrix to model large grids with limited dispersal among patches, and brings a large speed-up compared to the previous implementations (see also subsection 5.4.2). All migration matrices are now reduced internally.

While dispersal rates are usually interpreted as probabilities of individuals moving between patches in the population, it may be sometimes preferable to set the exact number of dispersing individuals. This is allowed by choosing option dispersal_by_number. As explained in subsection 5.4.3, when this option is set, the exact number of individuals (males or females) moving between patches can be specified, and modified across generations with temporal arguments.

5.4.1 Pre-set dispersal models

A classic Island-model in 10 patches with $m = 0.02$:

```
patch_number      10
patch_capacity    100
dispersal_model   1
dispersal_rate    0.02
```

Parameters:

disperse [integer]

The LCE name, which must be added to the ini file to use this LCE in the life cycle. The integer value is the rank in the life cycle.

dispersal_rate [decimal] (opt)

This parameter sets the male and female dispersal rates in the model chosen. Nemo will build the dispersal matrices accordingly.

dispersal_model [1,2,3,4] (opt)

The dispersal models implemented so far are:

1 : Migrant-pool Island model. If the migration rate is m, the probability to disperse to any $n_p - 1$ non-natal patch is $\frac{m}{n_p-1}$ while the probability to stay at home is $1 - m$.

2 : Propagule-pool Island model. In that modified version of the Island Model, all offspring in a patch have a probability $m\varphi$ to move to the same (assigned) patch. With probability $\frac{m(1-\varphi)}{n_p-2}$, they will move to any patch but their home or propagule-assigned patches. With probability $1 - m$ they will stay home. The propagule patches are reassigned every generation.

3 : Stepping-Stone model. This is the one-dimension Stepping Stone model. By default, the patches are placed on a circle (ring population) and the dispersers can only move to one of the two adjacent patches. This model can be changed by using different border models (see below).

4 : Lattice model. Patches are placed on a squared grid (or lattice) and dispersers can move to at least four adjacent patches (set by the dispersal_lattice_range parameter below). This option must be followed by the dispersal_lattice_model and dispersal_lattice_range parameters. The shape of the lattice is given by dispersal_lattice_rows for the number of rows and dispersal_lattice_columns for the number of columns of the grid. If these two numbers are not specified, then the grid is assumed to be square, and the number of patches in the population must thus be a square number.

dispersal_propagule_prob [decimal] (opt)

Sets the probability that a disperser will move to the propagule-assigned patch in the dispersal model 2.

dispersal_lattice_range [1,2] (opt)

Sets the number of neighbouring patches used for dispersal in the lattice dispersal model. The dispersal probabilities to these adjacent cells are $m/4$ in the first case and $m/8$ in the second.

1 : 4 adjacent patches (up, down, left, and right)

2 : 8 adjacent patches (as 1 plus the diagonals)

dispersal_lattice_rows [integer] (opt)

dispersal_lattice_columns [integer] (opt)

The two parameters are used to specify the number of rows and columns of the grid on which the patches are placed. Care should be taken that the total number of patches specified with patch_number corresponds to the size of the grid.

dispersal_border_model [1,2,3] (opt)

In the stepping stone and lattice models (i.e. 1D and 2D lattices), three different ways of dealing with the world edges exist:

1 : Torus. This is the doughnut world, edges are connected together. It has thus no boundaries, eliminating any edge effects.

2 : Reflective boundaries. The borders of the lattice (1D or 2D) are reflective. Dispersers from the border cells cannot move beyond the border. Border cells have thus less cells connected to them and their dispersal probabilities to the adjacent cells are higher (e.g. m, $m/3$, or $m/5$ depending on the dimension and range of the lattice). No dispersers are lost outside the lattice.

3 : Absorbing boundaries. Dispersers from the border cells of the lattice are lost if they choose to move beyond the border. The dispersal probabilities of a border cell are not modified.

A 4×5 torus lattice model with $m = 0.02$:

```
patch_number               20
patch_capacity             100
dispersal_model            4
dispersal_rate             0.02
dispersal_range            1
dispersal_lattice_rows     4
dispersal_lattice_columns  5
dispersal_boder_model      1
```

The patches are arranged row-wise on the grid:

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20

5.4.2 Dispersal matrices

Instead of supplying a dispersal rate operating in a pre-defined model, we can specify the full dispersal matrix, omitting the previous parameters.

dispersal_matrix [matrix] (opt)

This parameter is used to specify the full dispersal matrix of the population. It must be patch_number x patch_number in dimensions. Each d_{ij} element of this matrix is the dispersal probability from row patch i to column patch j.

This parameter has precedence over the dispersal rate and model parameters shown above in subsection 5.4.1. The dispersal matrix can be replaced by the dispersal_reduced_matrix and dispersal_connectivity_matrix (see below) when the number of patches is too big and especially when the matrix contains a large number of zeros.

Note: setting the identity matrix as dispersal matrix

Any of the dispersal matrices can be set to the Identity matrix (1's on the diagonal) if the value 1 is passed as a matrix argument: `{{1}}`. This is handy when simulating a single dispersing sex, for instance females as here:

```
dispersal_matrix_mal {{1}}
dispersal_matrix_fem {{0.98, 0.01, 0.01} {...} {...}}
```

dispersal_reduced_matrix [matrix] (opt)

This matrix holds the non-zero dispersal rates from patch i (row-wise) to patch j (column-wise) where the identity of the connected patch j is provided by the dispersal_connectivity_matrix parameter (see below). Because not all patches may be similarly connected to other patches, the number of elements per row may vary. For each row (= focal patch), the number of elements must exactly be the same as in the dispersal_connectivity_matrix. The sum of each row must be one.

dispersal_connectivity_matrix [matrix] (opt)

This matrix specifies to which patch each focal patch (row-wise) is connected through migration. The number of elements per row can vary among rows but must be exactly the same as in the dispersal_reduced_matrix. It is advised to sort the connected patches in descending order of the migration probability.

Example:
Imagine you have a single dispersal matrix for a circular Stepping-Stone model with 5 patches:

```
dispersal_matrix {{0.9, 0.05, 0, 0, 0.05}
                  {0.05, 0.9, 0.05, 0, 0}
                  {0, 0.05, 0.9, 0.05, 0}
                  {0, 0, 0.05, 0.9, 0.05}
                  {0.05, 0, 0, 0.05, 0.9}}
```

This matrix can be reduced to the following connectivity and rate matrices:

```
dispersal_connectivity_matrix {{1, 2, 5}
                               {2, 1, 3}
                               {3, 2, 4}
                               {4, 3, 5}
                               {5, 1, 4}}

dispersal_reduced_matrix {{0.9, 0.05, 0.05}
                          {0.9, 0.05, 0.05}
                          {0.9, 0.05, 0.05}
                          {0.9, 0.05, 0.05}
                          {0.9, 0.05, 0.05}}
```

5.4.3 Dispersal by number

The *number* of individuals between patches can be specified instead of the *rate* or probability of an individuals to move from a patch to another. A single option changes the default rate interpretation: dispersal_by_number.

dispersal_by_number [bool] (opt)

When present in the init file, this option changes the interpretation of the dispersal parameters, including rates and matrices. When set, positive integers can be passed to disperal_rate, dispersal_matrix, dispersal_reduced_matrix, and all the corresponding sex-specific parameters.

5.4.4 Sex-specific dispersal

The dispersal rates can be sex-specific in each case presented in the previous sections. Two parameters are then needed, one for the males and one for the females, obtained by adding _mal and _fem to the general parameter.

dispersal_rate_fem / _mal [decimal] (opt)

The two parameters replace parameter dispersal_rate for the case of different males and females dispersal capabilities.

dispersal_matrix_fem / _mal [matrix] (opt)

Replace dispersal_matrix with sex-specific dispersal matrices.

dispersal_reduced_matrix_fem / _mal [matrix] (opt)

Similarly, the reduced dispersal matrix can be sex-specific.

dispersal_connectivity_matrix_fem / _mal [matrix] (opt)

Ditto for the dispersal connectivity matrix. The two ma

Note: At least one of the optional dispersal rate/matrix parameters above must be present in order to correctly set the disperse LCE.

5.5 Seed dispersal

LCE name: **seed_disperse**
age flags: **offspring** (required)
files: NA

This LCE is an alias for the disperse LCE, as just described above. It is used when two types of dispersal events are part of the life cycle, as, for instance, when pollen dispersal (i.e. backward gametic migration) is modelled using the breed_disperse LCE. The seed_disperse LCE is thus adequate to model zygotic, forward migration.

All parameters are identical to the disperse LCE, to the exception that the 'dispersal' prefix must be replaced with 'seed_disp' (e.g. 'dispersal_rate' becomes 'seed_disp_rate').

Parameters:

seed_disperse [integer]

The LCE name, which must be added to the ini file to use this LCE in the life cycle. The integer value is the rank in the life cycle.

5.6 Evolving Dispersal

LCE name: **disperse_evoldisp**
age flags: **offspring** (required)
files: NA
inherits from: disperse

The disperse_evoldisp LCE uses the genetically-encoded dispersal rates of individuals to move them between patches following the dispersal model set with `disperse_model` parameter of the disperse LCE. The movement of (offspring) individuals thus depends on the phenotypic value of their dispersal trait. The simulation will not run if the dispersal trait is not added and properly set.

Since disperse_evoldisp is a specialization of the disperse LCE, it inherits its parameters, although the dispersal rate parameters have no meaning here. The other parameters, however, do apply. In addition, disperse_evoldisp defines a few more parameters used by the evolving dispersal models.

Parameters:

disperse_evoldisp [**integer**]

The LCE name, which must be added to the ini file to use this LCE in the life cycle. The integer value is the rank in the life cycle.

dispersal_cost [**decimal**]

This is the probability that a dispersing offspring dies during dispersal. The female and male costs are identical.

dispersal_cost_fem /_mal [**decimal**] **(opt)**

These two parameters set the dispersal costs affecting male or female dispersers separately. They will be overridden if the previous parameter is also present and they must be set together to set this LCE correctly.

dispersal_fixed_trait [**female, male**] **(opt)**

One of the sex-specific dispersal gene can be turned off with this parameter. The individuals of the selected sex will then migrate following the dispersal rate given below.

dispersal_fixed_rate [**decimal**] **(opt)**

This is the dispersal rate of the non-evolving sex.

5.7 Selection

LCE name: **viability_selection**
age flags: **offspring** (required)
files: `".fit"`
derived components: breed_selection, breed_selection_disperse

Viability selection selectively removes individuals from a patch based on their survival probability given by their fitness trait. Currently, the fitness determining traits are delet (deleterious mutations), quant (quantitative traits), and dmi (Dobzhansky-Muller incompatibility loci), although any other trait may be used as long as the trait's phenotype is compatible with the fitness models implemented. Fitness can be either *absolute* (i.e., directly set from the individual's phenotype) or *relative* to the mean fitness value of the patch or of the whole population. For now, it only acts on the offspring age-class but can be placed anywhere in the life cycle where offspring individuals can be found. Future releases will extend this behaviour to selection on other age classes. This LCE also declares a set of fitness statistics that can be recorded during the simulation (see section 8.2). The parameters described here are the same as those used with the breed_selection and breed_selection_disperse composite LCEs (which inherit those parameters).

Selection can also act on multiple traits simultaneously. That is, the fitness (survival) of an individual is given by the multiplication of the fitness values provided by each trait under selection. See section 5.7.1 below.

This LCE also declares an output files which can be called to save individual fitness values to a file, for each individual in each patch of the whole population, see parameters selection_output below.

Parameters:

viability_selection [integer]

The LCE name, which must be added to the ini file to use this LCE in the life cycle. The integer value is the rank in the life cycle.

selection_trait [string]

The argument to this parameter must be the name of the trait under selection. Only one trait can be specified (would become a sequential parameter otherwise). The traits' name are found in the next section. Currently, the delet, quant, and dmi traits are the only traits under viability selection (i.e., their trait value is used to set the individual fitness).

selection_model [fix, direct, gaussian, quadratic] (opt)

The selection models are:

fix : The fitness of the individual is set according to its pedigree and the number of lethal equivalents. The model used here is the following: $W_F = W_0 * e^{-F\lambda}$ where W_F is the fitness of an individual with pedigree inbreeding coefficient F, W_0 is the base fitness of the population (set below), and λ is the number of lethal equivalents present in the population.

direct (default) : The fitness of the individual is directly given by the phenotype of the trait, as for the deleterious mutations trait. This is the default model. There are no additional parameters for this model.

Gaussian : Stabilising selection on a set of quantitative traits. The fitness of an individual with phenotypic values $\mathbf{z}$ is:

$$W(\mathbf{z}) = exp\left[-\frac{1}{2}(\mathbf{z}-\boldsymbol{\theta})^T\boldsymbol{\omega}^{-1}(\mathbf{z}-\boldsymbol{\theta})\right],$$

where $\boldsymbol{\theta}$ is a vector of local optimal trait values, and $\boldsymbol{\omega}$ is the symmetrical positive semi-definite variance-covariance matrix of selection describing the individual fitness surface. The variance terms ω_{ii}^2 describing stabilizing selection on each trait i are on the diagonal of $\boldsymbol{\omega}$. They should not be confounded with V_{S}, the strength of selection acting on the *genotypes*, with $V_{\mathrm{S},i} = \omega_{ii}^2 + V_{\mathrm{E,i}}$. Covariances ω_{ij}^2 between traits i and j are off the diagonal and describe the strength of correlated selection, i.e., selection for specific trait correlations. Patch specific values of the local optima are specified with parameter selection_local_optima. The selection matrix is specified with selection_matrix, or a combination of selection_variance and selection_correlation. The two last selection parameters can also be patch specific (see below).

quadratic : A quadratic model of stabilizing selection on a ***single*** quantitative trait. Individual fitness is given as:

$$W(z_{i,k}) = 1 - \frac{(z_{i,k} - \theta_k)^2}{\omega_k^2},$$

where $z_{i,k}$ is the phenotypic value of individual i in patch k, θ_k is the phenotypic optimum in patch k and ω_k is the inverse of the strength of selection on the trait in patch k. Parameter selection_local_optima specifies the values for the θ_k's, and parameter selection_variance the values for ω_k^2.

selection_fitness_model [default: **absolute**; other options see below] (opt)

This sets how the fitness of the individual is interpreted. By default, the fitness of the trait is taken as **absolute**; it does not depend on the fitness of the other individuals in the population. Alternatively, the fitness of an individual (or its survival probability) can be interpreted relative to the fitness of other individuals in its patch when local or in the whole metapopulation when global.

relative_local or relative_global The fitness value of each individual is divided by the patch mean fitness (for relative_local) or by the metapopulation mean fitness (for relative_global) before testing for survival. This has the disadvantage of allowing for fitness values > 1 for the individuals with fitness larger than the mean fitness value. In effect, this means that all individuals with fitness > 1 are treated equally here because fitness is translated into an individual survival probability (would be different if applied to fecundity).

relative_max_local or relative_max_global The fitness value of each individual is divided by the *maximum* fitness value of the focal patch (for relative_max_local) or the whole metapopulation (for relative_max_global) before testing for its survival. Fitness is thus bounded upward to 1, and no individual can have relative fitness > 1.

selection_output [bool] (opt)

If present, will write the individual fitness values to a text file with name equal to the simulation name and ending with the `.fit` extension. The **raw** (absolute) fitness of an individual, its age, and a boolean indicating its migrant status are saved on one line.

selection_output_logtime [integer or matrix] (opt)

The frequency, in number of generations, at which the fitness values should be saved to file. Multiple values can be specified using a matrix array to indicate at which specific generations the data should be saved. That parameter is mandatory for the output.

selection_output_dir [string] (opt)

The directory, relative to the root simulation directory (root_dir parameter), where the files will be saved. The directory is created if not present.

5.7.1 Multi-trait selection

Selection can be applied to more than one trait. The fitness value of an individual is then given by the **product** of the fitness values provided by each trait.

The traits under selection must be passed to the selection_trait parameter enclosed within parentheses and coma-separated (i.e., (trait1, trait2)), and likewise for the selection models associated with each trait, in the same order (i.e., (model_trait1, model_trait2)). For example, to specify a model with selection on the delet and quant traits, the following set of parameters would be necessary:

```
selection_trait (delet, quant)
selection_model (direct, gaussian)
#parameters specific to the Gaussian selection model:
selection_trait_dimension 1
selection_variance 4
selection_local_optima {{5}}
```

5.7.2 Fixed selection model parameters

selection_base_fitness [decimal] (opt)

Base fitness of the population (W_0).

selection_lethal_equivalents [decimal] (opt)

Number of lethal equivalents present in the population (λ).

selection_pedigree_F [matrix] (opt)

The values of F for each of the 5 pedigree classes present in Nemo. Must be an array of size 5. The 5 classes are: outbred between patches (might experience heterosis), outbred within patches, half-sib, full-sib, and selfed individuals.

5.7.3 Gaussian and quadratic model parameters

selection_matrix [matrix] (opt)

This is the selection matrix $\boldsymbol{\omega}$ describing the strength of stabilizing selection on a set of quantitative traits within a patch. The $\boldsymbol{\omega}$ matrix is a square, symmetrical, positive semi-definite covariance matrix. The diagonal elements set the strength of selection on each trait (selection variance), while the off-diagonal elements set the strength of correlated selection on pairs of traits (selection covariance). These values will be applied to all patches equally as only one selection matrix can be specified per simulation. Patch specific variance and covariance values for each trait under selection can be specified with the next two parameters.

selection_variance [decimal/matrix] (opt)

This sets the variance or diagonal elements of the selection matrix $\boldsymbol{\omega}$. A single value will be interpreted as an identical selection parameter for all traits in all patches. A matrix argument can also be passed to change the selection variance among demes and traits. This matrix has at most as many rows as the number of patches in the population and as many columns as the number of traits modeled. When a smaller number of rows than the number of patches are provided, the values will be recycled to fill the patch-specific selection matrices. Similarly for the trait values, although here only a single identical value for all traits or all trait values are accepted.

selection_correlation [decimal/matrix] (opt)

This specifies the correlated effect of selection on the different traits. This is NOT the same value as you would use in the selection matrix (i.e. covariances). A matrix argument can also be provided to set the patch and trait specific values, with as many columns as the number of trait pairs, or just one value if the correlation is meant to be identical for all trait pairs.

selection_trait_dimension [integer] (opt)

Sets how many dimensions or quantitative traits are under selection within the quantitative trait. The number of dimension must be at most equal to the number of traits set with `quanti_traits`. It can be smaller if selection is not acting on all of them. This can be the case when modeling neutral traits or when some traits of the quantitative traits are used for other purposes. One example is when modeling plastic phenotype with evolving plasticity and some of the traits code for the slope of the norm-or-reaction(s) (see the Phenotype Expression LCE)

selection_local_optima [matrix] (opt)

A single array of local phenotypic optima for each quantitative trait, or a matrix with at most as many rows as the number of patches to set the patch-specific optimum values for each trait. The spatially-explicit matrix is dealt with in the same way as for the selection variances and correlations.

selection_rate_environmental_change [decimal/matrix] (opt)

A [patches x traits] matrix with per-patch, per-trait rates of change of the phenotypic optimum value of each trait in each patch (patches are row-wise and traits column-wise). A single decimal number will be interpreted as the rate of change of the optimum values in all patches and for all traits. If the matrix contains less values (less rows or columns) than the number of patches or traits, the specified values will be repeated to fill the whole [patches x traits] matrix (i.e., values are recycled).

The rates provided are decimal values and represent the values added to the local optimum value each generation (each iteration of the life cycle). For instance, a rate of 0.1 will change the local phenotypic optimum by 0.1 units per generation (e.g., $3 \rightarrow 3.1 \rightarrow 3.2 \rightarrow 3.3 \rightarrow 3.4$, etc.) A negative value will decrease the optimum trait value. The rate is thus independent of the amount of genetic variation in a population. This can be changed by using the set of parameters below.

selection_std_rate_environmental_change [decimal/matrix] (opt)

Same as above to the difference that the rates are interpreted as unit of phenotypic standard-deviation. The exact rate of change of the local phenotypic optima will thus be set depending on the amount of phenotypic variation in the population. To set the actual rates, the two next parameters are necessary to measure the phenotypic standard-deviation.

selection_std_rate_set_at_generation [integer] (opt)

This is the generation at which the phenotypic standard-deviation must be measured to set the relative rate of change of the phenotypic local optima. It is generation 1 by default.

selection_std_rate_reference_patch [integer] (opt)

The phenotypic standard-deviation of the traits under shifting environmental conditions can either be the average over all patches or set from a single reference patch. This parameter is used to specify that reference patch. Otherwise, if that parameter is not present in the init file, the average of the patch-specific phenotypic standard-deviations will be used.

5.8 Extinction and Harvesting

LCE name: **extinction**
age flags: **unchanged**
files: NA

This LCE is used to either cause the random extinction of patches in the population following the extinction rate or reduce their size by a given amount or proportion (i.e. harvesting). If a patch goes extinct, it is completely emptied of all the individuals present. This LCE only acts on the content of the patches, it never modifies their capacities (see resize for that). An extinction threshold can also be set as a percentage of the patch capacity and is used to control for patch extinction. The extinction rate is used as the probability of an event to occur, for each patch, be it total extinction or harvesting. The rate, harvesting size and harvesting proportion parameters can be set differently for each patch when using a matrix argument. They will affect all age classes equally unless the harvesting size is drawn from a random distribution. The sex of the individuals that are removed is set randomly.

Parameters:

extinction [**integer**]

The LCE name, which must be added to the ini file to use this LCE in the life cycle. The integer value is the rank in the life cycle.

extinction_rate [**decimal/matrix**] **(opt)**

Probability, per generation, that a patch undergoes extinction or harvesting. Defaults to 1. The default behavior (if none other parameters are given) is to completely empty the patch of all its individuals when an extinction event occurs.

extinction_size [**decimal/matrix**] **(opt)**

The number of individuals to be removed from a patch when the event occurs. Alternatively, the mean of the distribution of harvesting sizes (see bellow).

extinction_proportion [**decimal/matrix**] **(opt)**

The proportion of individuals to be removed from the patches in case of harvesting. The size parameter has precedence over this one.

extinction_threshold [decimal] (opt)

The threshold is set as the minimum **density** of individuals relative to the patch carrying capacity that must be present in the patch to consider it as non-extinct, including **all** individuals in the patch (offspring and adults). If the patch density is below that threshold, the patch is emptied.

extinction_size_distribution [uniform, poisson, normal, exponential, lognormal] (opt)

The distribution used to randomly draw the harvesting size of a patch. The mean of the distribution is taken from the extinction_size parameter. In case of the normal and lognormal distributions, the standard deviation of the distribution must be specified with the parameter below. The harvesting size is drawn from the distribution for each age class separately (i.e. offspring and adults).

extinction_size_dist_stdev [decimal] (opt)

The standard deviation of the normal and lognormal random distributions for harvesting sizes.

5.9 Trait initialization

Patch-specific trait or allelic values cannot be specified with the trait parameters. Instead, we need to use an LCE to perform this task. Such LCEs are implemented for the quant, dmi, and ntrl traits.

5.9.1 Initialization of trait quant

LCE name: **quanti_init**
age flags: **unchanged**
files: NA

There are two possibilities to initiate the quantitative trait, one by specifying the mean trait value in each patch, and the other by specifying the mean allele frequencies per locus. The allele frequency initialization is performed **for bi-allelic loci only**.

Parameters:

quanti_init [integer]

The LCE name, which must be added to the ini file to use this LCE in the life cycle. The integer value is the rank in the life cycle.

quanti_init_trait_values [matrix] (opt)

The matrix must hold patch-specific trait values in each row. If the number of rows is lower than the number of patches, values will be recycled. The number of values per row must be either equal to the number of traits modeled or one. If only one initial trait value is specified per row (patch), that same value will be used for all traits.

quanti_init_freq [matrix] (opt)

Similarly, the matrix must hold the patch-specific allele frequencies row-wise, and locus-specific frequencies column-wise. The frequency of the first allele only needs to be specified. As said above, the initializer assumes there are only two alleles per locus (see quanti trait parameters quanti_allele_model and quanti_allele_value). The same remarks hold concerning value recycling.

Important note: the "first" allele refers to the trait-increasing allele ($+a$) in the `diallelic` model, while it refers to the *ancestral* allele in the `diallelic _assymmetrical` model. The ancestral allele is usually the allele with value 0, or the smallest absolute value.

Example 1: Initialization by trait value
In this example, the initial trait values of the three traits correspond to the values of the trait phenotypic optima in each of 5 patches. The phenotypic optima for each trait and patch are set with the `selection_local_optima` parameter of the selection LCE.

```
patch_number         5
patch_capacity       1000

quanti_traits        3
quanti_loci          100
quanti_allele_model  continuous
[...]

# life cycle:
quanti_init          1
breed                2
viability_selection  3

quanti_init_trait_values  {{0,2,5}{1,2,4}{2,2,3}{3,2,2}{4,2,1}}

selection_trait      quant
selection_model      gaussian
selection_local_optima {{0,2,5}{1,2,4}{2,2,3}{3,2,2}{4,2,1}}
```

Example 2: Initialization by allele frequencies
This time, we model di-allelic loci and set the initial allele frequencies of the derived alleles to a low frequency of 5% in all 5 patches equally. Derived allele values are set randomly from an exponential distribution.

```
patch_number         5
patch_capacity       1000

quanti_traits        1
quanti_loci          100
quanti_allele_model  diallelic_asymmetrical
quanti_allele_value  {{rep(0,100)}       # ancestral values = 0
                      {rexp(100, 0.02)}} # random, positive values
[...]

# life cycle:
quanti_init          1
breed                2
viability_selection  3

quanti_init_freq     {{0.95}}  # freq. of derived alleles = 0.05
                               # same in all patches (recycled)
selection_trait      quant
selection_model      gaussian
selection_local_optima {{0}}   # recycled across patches
```

5.9.2 Initialization of trait ntrl

LCE name: **ntrl_init**
age flags: **unchanged**
files: NA

This LCE can be used to set initial allele frequencies in each patch differentially. **It assumes loci carry only two alleles (e.g., loci are SNPs).**

Parameters:

ntrl_init [**integer**]

The LCE name, which must be added to the ini file to use this LCE in the life cycle. The integer value is the rank in the life cycle.

ntrl_init_patch_freq [matrix] (opt)

The matrix must hold the patch-specific allele frequencies row-wise, and locus-specific frequencies column-wise. The frequency of the first allele only needs to be specified since loci are bi-allelic. If only one initial frequency value is specified per row (per patch), that same value will be used for all loci.

5.9.3 Initialization of trait dmi

LCE name: **dmi_init**
age flags: **unchanged**
files: NA

This life cycle event is used to set the frequencies of the mutant alleles at first generation. It allows setting the frequencies in a patch-wise manner. The frequencies at first generations will match those specified here on average because they are used as probabilities to sample mutations within a deme. In absence of an initializer, all individuals are monomorphic for the wild-type allele at all loci.

Parameters:

dmi_init [integer]

The LCE name, which must be added to the ini file to use this LCE in the life cycle. The integer value is the rank in the life cycle.

dmi_init_freq [matrix]

A matrix with one row per patch and one column per locus specifying the initial allele frequency at each locus in each patch. Both the number of rows and the number of columns can be smaller than the actual number of patches and loci, respectively. If so, the pattern present in the matrix will be repeated over all patches/loci.

Examples with 6 demes and 8 loci:

```
#to set all loci in all demes to allele 1
dmi_init_freq {{1}}

#to set the allele frequency to 0.25 in every second deme
dmi_init_freq {{0.25} {0}}

#to set loci 1,2,5,6 to allele 1 in demes 1,3,5,
#and to allele 0 at the other loci in the other demes
dmi_inti_freq {{1,1,0,0} {0,0,1,1}}
```

```
#same as above but with explicit repetition
#of the pattern of frequencies over loci
dmi_inti_freq {{1,1,0,0,1,1,0,0} {0,0,1,1,0,0,1,1}}
```

dmi_init_patch [matrix] (opt)

This optional parameter allows to restrict the settings given above to a specified set of demes. This is usefull to set allele frequencies is some demes only. Will have an effect on gene dynamics under stepping-stone/lattice dispersal only.

Example with 6 demes and 8 loci:

- to set one patch with all loci to allele 1:

```
dmi_init_freq {{1}}
dmi_init_patch {{6}} # this is patch no. 6
```

- to set three first patches to allele 1 at all loci:

```
dmi_init_freq {{1}}
dmi_init_patch {{1,2,3}}
#etc.
```

Note that this would be equivalent to setting the frequencies in each deme explicitly. This option is a shortcut when the number of demes is large, e.g., this would be equivalent to the two examples above:

```
dmi_init_freq {{0}{0}{0}{0}{0}{1}} #set in patch 6 only
dmi_init_freq {{1}{1}{1}{0}{0}{0}} #set in patch 1, 2, and 3
```

5.10 Trait modifiers

Trait modifiers are LCEs which specializes in changing a trait's genetic composition or phenotypic value within the population during the simulation. It can be specialized on a specific trait or affect any trait.

5.10.1 Modifier of trait **quant** – Set V_e dynamically

LCE name: **quanti_modifier**
trait: **quant**
age flags: **unchanged**
files: NA

The quanti_modifier LCE can be used to set the environmental variance of the quantitative traits during a simulation and change the phenotypic value of the individuals accordingly. The phenotypes will be recalculated form the genotypic values and the new value of V_e within each patch. The age class targeted can be specified explicitly.

This LCE uses the heritability (h^2) parameters of the quant trait to determine V_e as a function of the value of the additive genetic variance V_a (for narrow-sense h^2) or the genotypic variance V_g (for broad-sense H^2). The target heritability is set with parameter `quanti_heritability`. See subsection 6.3.8 on heritability to see how this is done.

Parameters:

quanti_modifier [integer]

The LCE name, which must be added to the ini file to use this LCE in the life cycle. The integer value is the rank in the life cycle.

quanti_modifier_at_generation [integer/array]

This sets the occurrence(s) when V_e must be calculated during a simulation to reach a target h^2 value set with parameter `quanti_heritability`. The argument can hold multiple occurrences in an array to specify at which generations V_e must be calculated.

quanti_modifier_at_age [adults | offspring | all]

This specifies which individuals are used to calculate V_a and set V_e accordingly. It can be any of the two age classes or both of them.

quanti_modifier_is_permanent [bool] (opt)

When set, this parameter changes the occurrence number provided with `quanti_modifier_at_generation` into a frequency at which the modifier is called during the simulation, in generation time (= one iteration of the life cycle). When set to 1, V_e is re-calculated every generation to maintain h^2 constant.

5.11 Phenotype expression

LCE name: **phenotype_expression**
age flags: **unchanged**
files: NA

This LCE implements a basic model of **phenotypic plasticity** based on the quantitative trait. The model uses a linear reaction norm to determine the phenotype of an individual i in an environment e_j in patch j based on three parameters, $g_{0,i}$, the genotypic value of the individual, g_1 the slope of the norm-of-reaction (n-o-r), and g_2, the environmental value in a reference environment. The phenotypic value z_i of individual i is:

$$z_i = g_{0,i} + g_1(e_j - g_2).$$

The `phenotype_expression` LCE can handle more than one plastic phenotype, depending on how many traits are modeled with the `quant` trait. In the basic model with constant plasticity (constant g_1 values), the intercept values g_0 of the n-o-r are read from the traits within the quantitative trait linked to each plastic phenotype. The g_0 values are then the genotypic values of the quantitative traits. The environmental cue and characteristics of the n-o-r can be set explicitly for each plastic phenotype.

The norms of reaction of the different traits can also evolve. This is achieved when linking the values of g_1 of each phenotype to the genotypic value of one of the traits of the quantitative trait. The slope of the n-o-r can then evolve as a quantitative trait.

Additionally, the g_2 values of each n-o-r can also evolve if also linked to a genotype of the quantitative trait. Therefore, the quantitative trait should have at least as many different genotypic values (*i.e.*, different traits set with `quanti_traits`) as the number of evolving features of the reaction norms set in this LCE. For instance, if only a single trait with non-evolving plasticity is modeled, then the LCE needs to be linked to only one trait (or dimension) of the quantitative trait. If, however, two evolving n-o-r with fixed reference environments are modeled, then the quantitative trait should have a least four different traits modeled, each providing the genotypic values for g_0 and g_1 of each n-o-r.

The environmental values e_j, also called the environmental cues, can be set specifically for each n-o-r, and can vary among patches in the population and across generations (with temporal arguments). Since selection on quantitative traits eventually affects individual survival via the match of the phenotype with a local phenotypic optimum value (see the Gaussian model of the selection LCE), the z values, once computed by this LCE, are stored at the same position of quantitative trait as the g_0 value. This way, it is possible for the selection LCE to later retrieve the relevant phenotypic values to compute the individual fitnesses. It is here recommended that the trait indexes of the quantitative trait coding for the different g_0 values start from 1 and are contiguous (e.g., {{1,2,3}}, see parameter `pheno_plastic_g0_quanti_trait`).

Note that the values of the environmental cue e_j need not be the same as the local phenotypic values set with `selection_local_optima` parameter. This way, a disconnect between the environmental cue (*e.g.*, photoperiod) and the environmental value (*e.g.*, temperature) can be modeled. A per-generation rate of change can also be specified to update the e-values (see `pheno_plastic_e_rate_change`) or they can be linked to the values of the phenotypic optima to match them (see `pheno_plastic_e_is_selection_optimum_value`). Finally, the cue can be unreliable depending on the e-value noise parameter `pheno_plastic_e_noise`.

Lastly, plastic expression in by default set once during the lifetime of an individual, at the offspring stage. This option corresponds to *developmental* plasticity. Since the position of this LCE can vary in the life cycle, the environmental value influencing the phenotype's expression can be the individual's natal patch or its patch of arrival after dispersal depending on the rank of the `disperse` LCE, for instance. Environmental expression of the traits can also be *labile* and re-set at each stage of the individuals (or each year). The parameter controlling this behaviour is `pheno_plastic_is_labile`.

Parameters:

pheno_plastic_e_value [**matrix**] *mandatory*

The matrix should hold the per-patch (row) $\times$ per-phenotype (column) values of the environmental cues e_j. If less values are provided, the existing values are recycled across patches and phenotypes. The number of phenotypes is deduced from the number of values passed to `pheno_plastic_g0_quanti_trait`.

pheno_plastic_g0_quanti_trait [**array**] *mandatory*

The integer values passed to this parameter correspond to the index of the trait modeled by the `quant` trait, set with parameter `quanti_trait` (see section 6.3). The corresponding traits then code for the intercept of the n-o-r. The number of plastic phenotypes depend on the number of indices provided here. Each will correspond to the g_0 value of a different plastic phenotype. The same dimensionality applies to the other parameters, where applicable.

Important note: since selection (see section 5.7) uses the n first dimensions of the quantitative traits, it is important that the trait indices of the quantitative trait coding for the different g_0 values start from 1 and are contiguous (e.g., {{1,2,3}}). The value of n is set with parameter `selection_trait_dimension` of the Gaussian selection model. It is equally important that Gaussian selection is not acting on traits used to code for g_1 or g_2 (see below). The indices for those parameters of the n-o-r should be set carefully in relation to the traits under Gaussian selection.

pheno_plastic_e_noise [matrix] (opt)

This sets the noise, or unreliability, of the environmental cue(s), per patch (row) and phenotype (column).

The noise parameter value corresponds to the variance of the normal distribution utilized to draw the value of the environmental cue perceived by an individual. In other terms, the individual phenotype becomes:

$$z_i = g_{0,i} + g_1((e_j + \epsilon_{i,j}) - g_2),$$

with $\epsilon_{i,j} \sim \mathcal{N}(0, \sigma^2_{e,j})$, and $\sigma^2_{e,j}$ is the value of the noise parameter in patch j. Therefore, when $\sigma^2_{e,j} \neq 0$, each individual perceives a slightly different cue. The values can be patch-specific.

pheno_plastic_e_rate_change [matrix] (opt)

The rate of change of the environmental cue(s), which can be different for each patch (row-wise) and each plastic phenotype (column-wise). The rate values correspond to the constant per-generation (life-cycle iteration) increment/decrement of the e_j's.

pheno_plastic_e_is_selection_optimum_value [bool] (opt)

When set (or equal to 1), then the e_j values are copied, each generation, from the corresponding θ_j values of the trait phenotypic optimum set with `selection_local_optima` in the viability_selection LCE. To work properly, the two sets should have the same dimensionality (affect the same number of traits in quanti).

pheno_plastic_g1_value [array] (opt)

This sets the fixed per-phenotype slope values of the n-o-r's modeled. The number of values *must* be equal to the number of plastic phenotypes set with `pheno_plastic_g0_quanti_trait`. This parameter must be set if the n-o-r slopes do not evolve.

pheno_plastic_g1_quanti_trait [array] (opt)

The slope values of the plastic phenotypes, or their plasticity can also evolve. This parameter sets the link to the corresponding traits within the quantitative trait coding for the g_1 values of each n-o-r. The number of indices provided here must match the number of plastic phenotypes (see above). The values must also not exceed the number of traits modeled within the quantitative trait and set with `quanti_traits`.

Important: The indices provided here must be different from those provided for the g_0 values. They also should not be used to set individual fitness in the viability_selection LCE.

pheno_plastic_g2_value [array] (opt)

By default, the g_2 values are zero for each plastic phenotype. This can be changed when using this parameter. The number of values provided must correspond to the number of plastic phenotypes.

pheno_plastic_g2_quanti_trait [array] (opt)

When set, the g_2 values of the n-o-r then evolve as well and are linked to a set of trait indices within the quantitative trait that are not used to code for g_0 or g_1.

pheno_plastic_is_labile [bool] (opt)

When set (or equal to 1), the phenotypes are re-set at each iteration of the life cycle in all age classes of the population. Note that the environmental variance used within the quantitative trait will not affect the trait values at other stages than the offspring stage since the environmental deviations added to the genotypes are not recorded. The phenotype values in the adults are re-set from their trait genotypic values, used as base values for the evolving parameters of the n-o-r.

5.12 Resize Population

LCE name: **resize**
age flags: **unchanged**
files: NA

The resize LCE modifies the state of the meta-population during a simulation but with more control than by using temporal arguments within the population parameters. In particular, it allows the user to merge or split existing patches without losing individuals. It also allows for the addition of empty patches (as when using temporal parameters).

Parameters:

resize [integer]

The LCE name, which must be added to the ini file to use this LCE in the life cycle. The integer value is the rank in the life cycle.

resize_at_generation [integer/matrix]

This is the generation at which the population will be modified. Mandatory.

This parameter also accepts a matrix argument with all the generation numbers specified on a row. Temporal arguments at the other resize parameters then allows the modification of the population state at different points during a simulation (see examples below).

resize_patch_number [integer] (opt)

Specifies the new number of patches in the population.

resize_patch_capacity [integer] (opt)

Specifies the new patch carrying capacity, also accepts a matrix argument as the population parameter (see 4.2).

resize_female_capacity [integer] (opt)

Changes the patch carrying capacity for the females only (similar to pop_nbfem).

resize_male_capacity [integer] (opt)

Changes the patch carrying capacity for the males only (similar to pop_nbmal).

resize_age_class [offspring, adults, all] (opt)

Sets the age class of the individuals to use when filling up new or empty patches. If no individuals of the required age class are present in the population, the LCE does not modify the population. It defaults to 'all'.

resize_do_flush [bool] (opt)

This parameter tells what to do with supernumerary individuals that are produced when patches are removed from the population. It also conditions the way patches are filled.

When set (present), any supernumerary individuals will be flushed (removed) and patches may subsequently be filled using individuals created *de novo*; i.e. they are similar to first generation individuals and have no parents.

When not set (absent), supernumerary individuals are backed up and may then be used to fill the remaining patches. This option is necessary when simulating patch fusion (e.g. bring the individuals from two patches into one) or fission (e.g. create two patches from one).

resize_do_fill [bool] (opt)

If set, the patches will be filled after the patch number and/or the patch carrying capacities have been modified. The individuals used to fill the patches are either backed-up individuals (i.e. do_flush is not set) or first-generation individuals (i.e. do_flush is set, see comment above). Patches will be filled sequentially (starting from the first) until they reach their carrying capacity. If do_flush is not set and the backed-up individuals are not in sufficient number, the filling procedure will stop before all patches are filled (which will happen if the total population size is increased).

If not set, new patches will be empty and undersaturated patches will remain as such and be filled by breeding and immigration in subsequent generations.

resize_do_regulate [bool] (opt)

If set, the patches will be regulated to their carrying capacities. This will affect the offspring and adults similarly. The patch sizes will be at most equal to their carrying capacities. Regulation is random.

If not set, patches may still have individuals above carrying capacity after modifying the population. Note that if do_flush is not set but do_fill is set, patches are automatically regulated to be able to fill the empty/undersaturated patches with any supernumerary individuals available in the population.

resize_keep_patch [matrix] (opt)

This array parameter (1D matrix) specifies which patches must be kept when resizing a population. Its length will set the number of patches in the population after resizing. The patches are numbered from 1 to patch_number and they are ordered as specified by the patch_capacity parameter. The order of IDs specified here is kept; patches may thus be reordered with this option as shown is the next example:

```
patch_capacity {{5, 10, 5, 10, 100}}
resize_at_generation 1000
resize_keep_patch {{1, 5, 4, 2, 3}}.
```

Note that this reordering will not have any consequence on the evolution of the population unless the migration scheme is different from the island model.

Examples: Here is an example of the **fusion of two patches** into one:

```
patch_number 2
patch_capacity 100
resize 1 #rank in the life cycle
resize_at_generation 1000
resize_patch_number 1
resize_patch_capacity 200
resize_do_fill
```

Using the population parameters only would not lead to the fusion of the two patches, as shown in this next example. Instead, one patch (the first one) will be destroyed along with the individuals it contains while the carrying capacity of the remaining patch is increased to 200.

```
patch_number (@g0 2, @g1000 1)
patch_capacity (@g0 100, @g1000 200)
```

Temporal argument values can be used to model more **complex demographic scenario** as in this next example:

```
patch_number 6
patch_capacity 200
resize_at_generation {{100, 1000, 2000, 3000}}
resize_patch_number (@g0 1, @g1000 2, @g2000 4, @g3000 6)
resize_patch_capacity (@g0 200, @g1000 100, @g2000 150, @g3000 200)
resize_do_fill  (@g0 1, @g2000 0)
```

Here, the population starts with 6 patches of size 200. A massive extinction occurs at generation 100 reducing the population to one patch of size 200. The population then starts growing again from generation 1000 to 3000 with the fission of its unique patch into two smaller ones first (i.e. do_fill is true). Two empty patches are added at generations 2000 and 3000 (do_fill = 0) while the patch capacity increases from 100 to 200 over 2000 generations bringing the population to its original state.

Note that the temporal specifiers all start with 0 (as expected by default) which sets the argument values for the first time resize will run, that is at generation 100 in the above example. The next temporal values must be set at times corresponding to those within the resize_at_generation array argument. That parameter can also be a temporal argument, however the array form is preferred for its compactness. The following example illustrate this point; both statements are equivalent:

```
resize_at_generation {{100, 1000, 2000, 3000}}
resize_at_generation (@g0 100, @g1000 1000, @g2000 2000, @g3000 3000).
```

5.13 Crossing Designs and Pedigree cross

LCE name: **cross**
age flags: **adults** (required); **offspring** (add)
files: NA

The cross LCE lets you perform a North Carolina I crossing design (or half-sib, full-sib design) of the population at a given time point during a simulation. The LCE creates sire x dam x offspring offspring in each patch of the population. It is thus advised not to set the numbers of sires or dams higher than the number of males or females present in the patches. The crossing process will replace any offspring previously present in the patches (a warning is issued). Sires and dams are randomly selected with or without replacement within each patch, depending on the value of the cross_with_replacement parameter.

Crossing from a pedigree file: Alternatively, the cross LCE lets you create a population from a pedigree file (with parameter cross_with_pedigree_file), using the individuals present in the population at the time of processing as the base population for the creation of the individuals along the pedigree. This crossing is performed using individuals in the first patch only, if more than one patch have been simulated. Once the individuals along the pedigree have been produced, they are placed in the *female offspring* container of the first patch of the population after having removed all previous individuals in the whole population. This process thus destroys the population and only keeps the individuals of the pedigree (this can be change if needed, let us know). A pedigree population can also be generated without a base population if the first patch is empty. In that case, individuals at the root of the pedigree will be created “de novo”, as if they were first-generation individuals with randomly set allele frequencies.

Parameters:

cross [integer]

The LCE name, which must be added to the ini file to use this LCE in the life cycle. The integer value is the rank in the life cycle.

cross_at_generation [integer]

Generation at which crossing is performed.

cross_num_sire [integer] (opt)

Number of sampled males per patch. Each male will be mated with num_dam females as many times as num_offspring. Not used for pedigree-based crossing.

cross_num_dam [integer] (opt)

Number of sampled females per sire. Each female produces num_offspring with one given male. Not used for pedigree-based crossing.

cross_num_offspring [integer] (opt)

Number of offspring produced per dam. Not used for pedigree-based crossing.

cross_do_within_pop [bool] **(opt)**

If set (the default), dams and sires will be sampled within populations.

cross_do_among_pop [bool] **(opt)**

If set, the crossings will be performed by sampling a sire and a dam from two different populations. Sampling proceeds by first randomly selecting *num_sire* males within each patch and randomly assigning *num_dam* females to each sire taken from patches different from the sire's one. This ensures that the sire and the dam of each cross are from different patches.

Both within and among patch crosses can be performed if both options are set.

cross_with_replacement [bool] **(opt)**

If set to 1 (true), this option allows to sample individuals with replacement, that is, to sample several times the same individual when selecting dams or sires for the crossings. If not present (or set to 0), the sampling is done without replacement, which is the default.

cross_with_pedigree_file [string] **(opt)**

If present, crossing along a pedigree will be performed instead of the half-sib cross. This parameter specifies the input file containing the pedigree information for all individuals to be created.

Structure of the pedigree file The pedigree file must contain a white-space-separated table with three columns: ID (an integer uniquely identifying each individual), father (the ID of the father), and mother (the ID of the mother). The information for each individual is on a separate row. Only integer numbers are allowed in the pedigree file, or "NA", "na", "NAN", "NaN", or "nan" when the ID of a parent is missing. Duplicated IDs are not allowed. Don't use letters in IDs and don't use CSV files!

5.14 Population Regulation

LCE name: **regulation**
age flags: **adults** (required)
files: NA

Population regulation is used to remove all individuals in excess of the (sex-specific) carrying capacity of each patch. This very simple density-dependent mode of regulation is called "ceiling" regulation. Regulation is performed on each age class present in the population, that is on the offspring and adult individuals. The supernumerary individuals that are removed are chosen at random. It is not necessary to place regulation after aging in the stack of life-cycle events as the aging LCE also performs "ceiling" regulation. After regulation, the patches will be at their carrying capacity only if there was enough individuals present prior to regulation.

Parameters:

regulation [integer]

The LCE name, which must be added to the ini file to use this LCE in the life cycle. The integer value is the rank in the life cycle.

5.15 Saving Statistics

LCE name: **save_stats**
age flags: **unchanged**
files: `".txt", "_bygen.txt"`

This LCE is used to tell the stat-services of the simulation to record the summary statistics specified with the stat parameters (see below). The statistics recorded depend on the age state of the population. The position of this LCE in the life cycle is thus important. Putting it after breeding will allow you to record stats on both offspring and adults while putting it after aging will allow you to record the stats on the adults only. The recorded stats are dumped to a text file at the end of each replicates and at the end of a simulation for the averaged stats, but only if the save_files LCE is present in the life cycle. See chapter 8 for a description of the different output files declared by this LCE. Note that no results will be saved if none of save_stats or save_files are present in the life cycle.

Parameters:

save_stats [integer]

The LCE name, which must be added to the ini file to use this LCE in the life cycle. The integer value is the rank in the life cycle.

stat [string]

The string passed to this parameter must contain the stat options defined by the various simulation components. A list of these options is given in chapter 8.

Note: This is the only non-sequential parameter, the list of arguments is considered as one complete character string.

stat_log_time [integer]

This is the generation recording time of the summary statistics defined by the previous parameter.

stat_dir [string] (opt)

This optional parameter is used to specify a path to a directory where to save the stat files. It shall not end by a slash character ('`/`').

stat_output_compact [bool] (opt)

Changes the format of the output stat files by suppressing the pretty printing of each column with lots of space between them. Instead, each value is separated by a single space character. The value-separator can be changed to a comma with the next option below. Use this to save space on disk.

stat_output_CSV [bool] (opt)

Changes the column separator from a white space ' ' to a comma ','. Implies compact output format.

stat_output_width [integer] (opt)

Sets the column with in the output stat files. Is 12 characters by default.

stat_output_precision [integer] (opt)

Sets the decimal precision in the output stat files. Is 6 by default.

stat_output_no_means [bool] (opt)

Suppresses the writing of the output file containing the stat means, ending with '_bygen.txt'.

Output stats: `alive.rpl`

This stat appears in the `"_bygen.txt"` files only and is the number of alive replicateat each generation recorded. This is an automatic statistic, no additional token is needed to the stat parameter.

5.16 Saving Files

LCE name: **save_files**
age flags: **unchanged**
files: varies

This LCE tells the program when during the life cycle the simulation data must be saved on disk by the different simulation components. This excludes binary data that is saved by the store LCE (see below). The save_files LCE is mandatory if you want to have any output data saved by your simulation. Each simulation component (trait or LCE) may define different output files to save specific information (e.g. specific stats or genotypes/phenotypes of a specific trait, etc.). The program file manager is notified by save_files that is must initiate the file handlers' output process at the point it has been inserted in the life cycle. The type/composition of the data that is saved will thus depend on the rank of this LCE in the life cycle because the age composition and the state of the population is changed by other LCEs. It is not possible, for now, to use save_files more than once in the life cycle. This prevents, for instance, saving some data before and after a specific LCE (e.g. sequence data before and after disperse). This will probably change in future releases.

Some simulation components automatically upload their different file handlers to the file manager. For instance, the save_stat LCE defines two types of automatic output files, one ending with the `".txt"` and the other with the `"_bygen.txt"` extensions (see above and chapter 8) to save the statistics recorded during the simulation. Other component let the user chose what and when data must be saved on disk (see the trait components for e.g.).

Parameters:

save_files [integer]

The LCE name, which must be added to the ini file to use this LCE in the life cycle. The integer value is the rank in the life cycle.

files_sample_size [integer] opt

Number of per-patch, per-sex and per-stage individuals randomly sampled from each patch for file output. This mostly concerns trait output files when saving individual genotypes. A value of 100 would save the genotypes of 400 individuals per patch in a sexual population (males + females) with offspring and adults present (4×100).

files_sample_age [integer] (opt)

An age flag set to specify which age class must be sampled. Set it to 1 for offspring only, 4 for adults only, and to 7 to sample both age classes.

5.17 Storing Data in Binary Files

LCE name: **store**
age flags: **unchanged**
files: `".bin" (".tar", ".bz2")`

This LCE provides a way to save all the traits and individual's data into a binary file. That file can then be used to initiate a new simulation using the `source_pop` option in the population parameters. Binary files contain all the genetic and individual data plus the whole set of parameters that allowed to generate these data. Only one generation of one replicate can be saved in one binary file (new since v2.3.51). Several generations of the same replicate can be saved in separate files (with parameter `store_recursive`). By default, binary files are compressed after being written to disc (with `bzip2` by default) and put in a "tar" archive. This behaviour can be changed with the 'archive' and 'compress' parameters below.

Parameters:

store [integer]

The LCE name, which must be added to the ini file to use this LCE in the life cycle. The integer value is the rank in the life cycle.

store_dir [string] (opt)

Used to specify the directory where to save the binary files.

store_generation [integer]

The generation to save in the binary files. The last generation will always be saved whatever the value given here.

store_recursive [bool] (opt)

This option will tell the program to use the `store_generation` value as a generation logging time. Each generation will be saved in a separate file.

store_noarchive [bool] (opt)

This option suppresses the archiving of the binary files.

store_nocompress [bool] (opt)

This option will suppress the compression of the binary files.

store_compress_cmde [string] (opt)

The program used to compress the binary files is by default `bzip2`. You can change this default behavior by specifying a alternative program (or path to that program) to use here.

store_compress_extension [string] (opt)

The alternative used with the previous parameter will probability use a different file extension than `".bz2"`. Use this parameter to specify that alternative extension.

store_archive_cmde [string] (opt)

Similarly to the compression process, an alternative archiver program can be specified here to avoid the use of tar.

store_archive_extension [string] (opt)

The file extension used by the alternative archive program can be specified here.

5.18 Composite LCE

Composite life cycle events are LCEs that inherit the properties (parameters) of other LCEs (the base LCEs) and extend, or sometimes, redefine their functionalities. For instance, `breed_selection` inherits the parameters of the `breed` and `viability_selection` LCEs and performs both breeding and viability selection in one but doesn't add any new parameter. Other composite LCEs may add new parameters (see below). Also, composite LCEs sometimes rename the parameters of the base LCE. For instance, `breed_disperse` renames the dispersal parameters as `breed_disperse_...` instead of `dispersal_...`. This allows a parameter file to have both the `disperse` and `breed_disperse` parameters without conflating them. The composite LCEs are:

```
breed_selection
breed_disperse
breed_selection_disperse
```

The goal of merging `breed` and `viability_selection` into `breed_selection` is to avoid fully creating new individuals that do not survive. Instead, `breed_selection` first checks for offspring survival based on the inheritance (with mutation) of the traits under selection only, before computing inheritance and mutation on the traits not under selection if the offspring survives. This way, some runtime is spared by not having to compute inheritance of the neutral trait in non-surviving offspring, for instance. The `breed_selection` LCE also handles Wright-Fisher populations with selection, where parents are selected proportionally to their fitness and only K offspring are produced in a patch with capacity K. This is only possible if `breed_selection` and `mating_isWrightFisher` are both set in the init file. Note that the two cases differ in that selection acts on offspring viability in non W-F populations, while it acts on adult fitness (e.g., fecundity) in W-F populations. Their population dynamics also differ since W-F populations are of constant size, while non W-F populations can vary in size because of selection.

The goal of the two other composite LCEs, `breed_disperse` and `breed_selection_disperse` is to simulate populations of constant sizes, also proposing a gain in runtime by generating the exact number of offspring required to fill the patches to their capacity. It is thus similar to the case of W-F populations although setting the `mating_isWrightFisher` parameter is not required (and discouraged). In order to maintain population size constant with migration, the dispersal rates are interpreted as rates of *immigration* and not *emigration* as with the `disperse` LCE. We then call migration *backward migration* in that case, in contrast to the *forward* migration implemented with `disperse`. Backward migration can be compared to gametic migration (e.g., when gametes are released in the environment, as with pollen), while zygotes disperse with forward migration (e.g., larvae, seeds or fledged offspring). Therefore, `breed_disperse` offers the possibility to model a Wright-Fisher population with migration.

The `breed_selection_disperse` LCE also offers to model a Wright-Fisher population with migration but with selection on top. The difference with Wright-Fisher population with selection as with `breed_selection` and `mating_isWrightFisher` is that parents are not selected proportionally to their fitness. The cause is that it is not entirely clear how to set relative fitness at the scale of an entire metapopulation where individuals can be adapted to local conditions that vary across patches. Therefore, instead of selecting on the parental population, `breed_selection_disperse` performs selection on the offspring when they are created, implementing viability selection. The algorithm is such that offspring are created within a patch with backward migration until K_i surviving offspring are reached in patch i. If offspring survival is on average low (e.g., large migration with strong divergent selection between patches), then the simulation will slow down as NEMO tries to fill the patches to their full capacity. A workaround is proposed in section 5.21 below.

5.19 Breed with selection

LCE name: **breed_selection**
age flags: **adults** (required) and **offspring** (added)
files: NA
inherits from: breed, selection

This composite LCE performs breeding and viability selection on the offspring generation in one step. It inherits the parameters from the `breed` and the `viability_selection` LCE's parameters as described before. No additional parameters are required. The following features differ from the base LCE's:

- Fitness is always absolute.
- The realised fecundity of a female or male is set accordingly to the survival of their offspring (allowing the correct computation of the values of the `heterosis`, `load`, and females/males realised fecundities and fecundity variances).

- This LCE may be faster than having `breed` followed by `viability_selection` in the life cycle when more than one trait are simulated, because mutation and recombination are performed on the selected trait before checking for survival. Therefore, mutation and recombination of the traits not under selection are performed on the surviving offspring only.

Parameters:

breed_selection [integer]

The LCE name, which must be added to the ini file to use this LCE in the life cycle. The integer value is the rank in the life cycle.

breed_selection_fecundity_fitness [bool]

If this parameter is set (present in the init file), the selection mode is changed from acting on offspring survival to act on the number of offspring produced by each female. In other words, with this mode, it is the fitness of the female that matters rather than that of the offspring. The mean value of the fecundity distribution is multiplied by each female's fitness when drawing its number of offspring produced. This works best when the mean fecundity is large because only integer numbers of offspring can be produced, which is problematic when the mean of the Poisson distribution is too low (e.g. a fitness of 0.25 and a mean fecundity of < 4 will cause many more females to have no offspring than if the mean fecundity is 10). By having a too low mean fecundity, one looses precision in the selective process, and selection will be stronger.

5.20 Breed-disperse (gametic migration)

LCE name: **breed_disperse**
age flags: **adults** (required) and **offspring** (added)
files: NA
inherits from: breed, disperse

Important: the dispersal parameters that are inherited from the disperse LCE must be pre-pended with breed_disperse instead of dispersal as in the original LCE. For instance, dispersal_rate becomes breed_disperse_rate, dispersal_matrix becomes breed_disperse_matrix, etc.

This LCE performs breeding and dispersal in a single step. It inherits the parameters of the breed and disperse LCEs. For an offspring, each parent is randomly taken from the local patch with probability $1-m$ or from a different patch with probability m, where m is the dispersal rate. The dispersal rates are thus taken as *backward migration* or *immigration* rates in opposition to the *forward* emigration rates of the disperse LCE. This corresponds to the classical Wright-Fisher model if the mating system is hermaphroditism (mating_system 6). By default, exactly K offspring are produced per patch, if K is the patch capacity, unless the patch is extinct and the parameter breed_disperse_colonizers is specified, which limits the number of individuals grown locally from immigrant gametes. The number of offspring produced locally can also be density-dependent and set following different growth models using parameters breed_disperse_growth_model and breed_disperse_growth_rate.

The following features differ from the breed and disperse base LCE's:

- **backward migration**, the columns of the dispersal matrix must sum to 1 instead of the rows, because NEMO reads the immigration rates column-wise (element d_{ij} is the probability to get a migrant gamete *from* deme i *into* deme j, i being the row number and j the column number).
- There can be **no demographic stochasticity** (demes always at carrying capacity) if the growth model is set to 1 (instant growth, default value), and breed_disperse_colonizers is unset.
- Deme **extinctions** may cause the **program to hang indefinitely** if immigration into an extinct deme is impossible (e.g., because of source patch extinction or zero immigration set in the dispersal matrix).
- An extinct deme will be **instantly recolonised** (in a single generation) unless the number of immigrants is capped with breed_disperse_colonizers or a growth model is specified.
- Two dispersal matrices can be used for hermaphrodites to **model pollen migration** (i.e., fecundation of local ovules with immigrant pollen, without ovule migration), see breed_disperse_dispersing_sex.
- Mating systems 2 (polygyny) and 3 (monogamy) can not be used here.
- This LCE can be used to mimic the Wright-Fisher model when the mating system is set to 6 (random mating with selfing rate $= \frac{1}{N}$).
- This LCE is much faster than having breed followed by disperse in life cycle because exactly N offspring are produced and not $\frac{N}{2}\bar{f}$, $\bar{f}$ being the females mean fecundity. Usually, $\bar{f}$ should be greater than 2 to avoid too much demographic stochasticity, especially with small patch sizes.

Parameters:

breed_disperse [integer]

The LCE name, which must be added to the ini file to use this LCE in the life cycle. The integer value is the rank in the life cycle.

breed_disperse_colonizers [integer] (opt)

This parameter is used to restrict or set the number of individuals that will re-colonise an empty patch to a different value than the carrying capacity of that patch. That number is sex-specific, the actual number of colonisers will be twice the value for dioecious individuals (biparental reproduction).

breed_disperse_dispersing_sex [“female”, “male”] (opt)

Specifies the sex of the dispersing gamete, used when only females (monoecious individuals) are present in demes as for hermaphroditic or self-fertilising mating systems (models 6 and 4, respectively). Should be set to male to model pollen dispersal (i.e. male gamete dispersal) to indicate which dispersal matrix must be used to select the right “father” (which, in this case, is another female hermaphrodite individual, possibly in another patch). If hermaphrodites are sessile individuals (plants) and the ovules do not disperse, then the breed_disperse_matrix_fem must be set to the identity matrix (complete philopatry).

breed_disperse_growth_model [1-7] (opt)

1 – **instant growth**: patches are filled to their carrying capacity within one generation. This is the default model.

2 – **logistic growth**: the number of offspring produced in patch i is given by the classical logistic growth model with $N_J = N_B + rN_B * ((K_i - N_B)/K_i)$, with r the growth rate given by breed_disperse_growth_rate, N_B the number of breeding individuals, and N_J the numbers of juveniles produced, in patch i.

3 – **logistic stochastic**: the number of offspring is drawn from a Poisson distribution with mean set by the logistic model as above.

4 – **logistic conditional**: if the number of breeding adults is below K/2, use model 6, else use model 2.

5 – **logistic conditional stochastic**: if the number of breeding adults is below K/2, use model 7, else use model 3.

6 – **fixed fecundity**: the number of offspring produced in patch i is $N_{t+1} = N_t * \bar{f}$, $\bar{f}$ the mean fecundity set by mean_fecundity.

7 – **stochastic fecundity**: as in 6 but with the total number of offspring drawn from a Poisson distribution of mean equal to N_{t+1}.

8 – **exponential growth**: the number of offspring is calculated as

$$N_t = N_0 * exp[r * t],$$

with N_0 the number of breeders at time 0 when the growth function is first called, t the number of generations since time 0 and r the growth rate.

Importantly, *time 0* refers to the generation when the function is first called (can be different from generation 0 if growth model is changed).

breed_disperse_growth_rate [decimal] (opt)

The patch growth rate used in the logistic and exponential growth models.

5.21 Breed with selection and backward migration

LCE name: **breed_selection_disperse**
age flags: **adults** (required) and **offspring** (added)
files: NA
inherits from: breed_disperse, selection

This LCE aggregates the features of both previous composite LCEs. However, to perform selection and backward migration with populations of constant sizes, there must be some adjustments in the way selection is performed in the case where the mean fitness is too low to allow the patches to be filled with surviving offspring. The basic idea is therefore to define a minimum fitness threshold for the individuals. If the mean fitness of the adult (breeders) generation is below that threshold before mating, the offspring fitness is rescaled so that the mean patch fitness matches that threshold. In other words, the threshold is the minimum survival probability that the offspring in a patch can reach. The fitness scaling factor is $\frac{\text{fitness threshold}}{\text{mean fitness}}$. As soon as the mean patch fitness is above that threshold, the scaling factor is reset to 1. This trick helps boost the simulations when the starting conditions for the traits under selection are very far from their optimum, for instance. The threshold is set to 1% by default.

Parameters:

breed_selection_disperse [integer]

The LCE name, which must be added to the ini file to use this LCE in the life cycle. The integer value is the rank in the life cycle.

breed_selection_disperse_fitness_threshold [decimal] (opt)

The minimum fitness value used to rescale the individuals fitness when the mean patch fitness is too low to allow for the patch to be filled (see above). It is 0.01 by default (1% surviving probability).

Important: since breed_selection_disperse inherits parameter definitions from breed_disperse, the dispersal parameters must also use the breed_disperse prefix instead of dispersal, see section 5.20 above.

Chapter 6

Traits

The traits described here are:

- ntrl (neutral markers, including microsatellites, SNPs, etc.)
- quant (quantitative traits)
- delet (deleterious mutations)
- dmi (Dobzhansky-Muller Incompatibility loci)
- fdisp/mdisp (sex-specific dispersal)
- wolb (*Wolbachia* endosymbiotic parasites)

Each trait has an identifying name or type and may define different output files and stat options. For a complete description of the stat options, have a look at chapter 8.

6.1 The Genetic map

The four sequence-based traits (`ntrl`, `quant`, `delet`, and `dmi`) share a common genetic map on which the loci of the different traits are placed. The genetic map in Nemo is a recombination map where the locus positions are specified in centi Morgan (cM), in opposition to the base-pair unit (bp) of physical maps. The genetic map may be composed of more than one chromosome, each with a different number of loci. The recombination distances between loci can be specified explicitly or set randomly. This way, for instance, neutral markers (SNPs) can be located more or less closely to loci under selection. This is done thanks to a set of parameters that are common to all traits traits and which are described in this section.

The naming convention for the genetic map parameters is: *prefix*_parameter_name, where '*prefix*' stands for 'ntrl', 'quanti', 'dmi', or 'delet'.

The unit of the map is the centi-Morgan [cM] by default but can be changed if needed with parameter *prefix*_genetic_map_resolution. A distance of one centi-Morgan between two loci is interpreted as a 1% chance of a recombination event per individual per generation between the two loci.

The map parameters are optional by default and unlinked maps for each traits will be built if no parameters are specified in input (that is, all loci are unlinked). There are four types of maps:

- **fixed maps** (*prefix*_genetic_map), which specify the exact map position of each locus on each chromosome,
- **random maps** (*prefix*_random_genetic_map), which randomly set map positions according to the map length of each chromosome,
- **fixed maps with equally-spaced loci** (*prefix*_recombination_rate), which set locus positions according to recombination rates that can be specific to each chromosome and trait, and
- **unlinked maps** (by default, or if *prefix*_recombination_rate $= 0.5$), which correspond to completely unlinked loci.

The map resolution, that is, the minimum distance at which crossing-over will be placed, depends on the minimum resolution specified by the map parameters of the different traits and can be explicitly set by *prefix*_genetic_map_resolution.

The only limitation is that the number of chromosomes can not differ among traits (i.e. chromosomes without loci for a given trait are not accepted).

***prefix*_genetic_map [matrix] (opt)**

This corresponds to a fixed map and is used to specify the map position of each locus of a trait. The matrix argument provides the locus positions using one row per chromosome (in [cM] by default). The number of chromosomes is then deduced from the number of rows. Similarly, the number of loci per chromosome is deduced from the number of positions specified on each row of the matrix. This parameter thus allows you to set a variable number of loci per chromosome and to pass their positions on each chromosome with a variable matrix, that is, a matrix with a different number of elements on each row.

***prefix*_random_genetic_map [array] (opt)**

Loci position can be set randomly on the map. Here, the array holds the map size of each chromosome (in [cM] by default). The number of chromosomes is deduced from the length of the array and the loci positions are drawn randomly from a uniform distribution on the range [0, map_size). By chance, two loci may land on the same map position. The number of loci per chromosome is either equal among chromosomes and set by dividing the number of loci of the trait by the number of chromosomes or set by the parameter *prefix*_chromosome_num_locus below.

The random positions are saved in the *.log* output file of the simulation (and in the binary file as well).

Important: the minimum length of a chromosome is 1 cM with this method, because the default scale is set to 1 cM. However, to match with the other parameters of the genetic map, the positions of the loci can be set at any other scale. For instance, if you set one chromosome length to 1cM but want to set many loci on it, you have to use *prefix*_genetic_map_resolution to avoid having all locus positions set to 0 (all loci completely linked):

```
# genetic map for neutral markers:
ntrl_loci                      3000
ntrl_recombination_rate        {{1e-5, 1e-5, 1e-5}}  # r
# genetic map for a quantitative trait (QTL):
quanti_loci                    30
quanti_random_genetic_map      {{1, 1, 1}}  # 1 cM long, r = 1e-2
quanti_genetic_map_resolution 0.001   # to match r of the ntrl trait
```

If _genetic_map_resolution 0.001, the map positions on a 1cM chromosome will be drawn between [0, 1000), or more generally: [0, map_size/map_resolution).

***prefix*_recombination_rate [decimal / array] (opt)**

This parameter is used to build a map with equidistant loci, with one recombination rate per chromosome. A recombination rate of 0.01 corresponds to a map distance of 1 cM. Therefore, if smaller recombination rates are specified, the map resolution will be reset accordingly. The number of chromosomes is deduced from the number of recombination rates in the array and the number of loci per chromosome is either equal among chromosomes and set by dividing the number of loci of the trait by the number of chromosomes or set by the parameter *prefix*_chromosome_num_locus below.

If a single value is given, without using a matrix argument, a single chromosome is constructed, if consistent with the genetic maps of other traits included in the same simulation. Traits may not differ in the number of chromosomes in their genetic map.

If a single value is given and is not passed as an array and that value is 0.5 (e.g. with ntrl_recombination_rate 0.5), the loci are considered unlinked and recombination is handled independently of the genetic map. Therefore, if two traits have a recombination rate of 0.5, their loci will be considered as unlinked, altogether. This would however not happen if an array argument is passed (e.g. with ntrl_recombination_rate {{0.5}} and delet_recombination_rate {{0.5}}), in which case the loci of the traits will have same map positions, although they are unlinked to the next loci.

***prefix*_chromosome_num_locus [array] (opt)**

The number of loci per chromosome can be varied using this option, giving the number of loci per chromosome in an array. The sum of the array must then be equal to the total number of loci of the trait. The array must have as many elements as the number of chromosomes specified by one of the map options *prefix*_random_genetic_map or *prefix*_recombination_rate.

***prefix*_genetic_map_resolution [decimal] (opt)**

The map resolution is, by default, the centimorgan (cM). The map positions specified by *prefix*_genetic_map or *prefix*_random_genetic_map thus refer to that scale. For example, two loci with positions 34 and 35 have a recombination rate of 0.01. The scale can be changed here by specifying the corresponding *reduction* of scale. Thus, *prefix*_genetic_map_resolution must be smaller than 1, and, for instance, a value of 0.1 means the resolution is changed to the mili-Morgan (0.1×1cM). Our two loci above with positions 34 and 35 would now have a recombination rate of 0.1% instead of 1%. The interpretation of the distances between loci thus depends on this scale. The map resolution applies to all chromosomes and all traits equally. If a trait changes the map resolution, all trait's maps are rescaled to the smallest scale.

6.2 Neutral markers

name: **ntrl**
files: `".txt"`, `".dat"` (input/output)
phenotype: none

Neutral markers are genetic markers such as microsatellites or SNPs, which are not affected by selection. The markers implemented here are all diploid, nuclear markers. Two models of mutation are implemented, the SSM (Single Step Mutation) and the KAM (K-Allele Model) models (see below for details). The probability of crossing-over occurrences between two adjacent loci can be set with the parameters of the genetic map. The number of alleles, and the allelic mutation rate are constant across loci. New populations can be initiated by assigning random allelic values within the range `[1, ntrl_all]` to each locus thus assuring a very large initial variance, or by assigning the same value to all loci. Other initialisation options are given by the source_pop option above (see population parameters 4.2), which allows you to load a population's genotypes from an FSTAT input file (see below for a description of that file format), or with the ntrl_init LCE to specify patch-specific initial allele frequencies for di-allelic loci (see section 5.9.2).

ntrl_loci [**integer**]

Number of (diploid) neutral locus per individual.

ntrl_all [**2 to 256**]

Number of alleles per neutral locus (same number for each locus).

ntrl_mutation_rate [**decimal**]

Mutation rate of the neutral alleles, identical across loci. The mutation model is specified with the next parameter.

ntrl_mutation_model [**0,1,2**]

Available mutation models are:

0 : no mutations

1 : SSM (Single Step Mutation)

2 : KAM (K-Allele Model)

The no-mutation model (`0`) is simply a void model used for the case of a null mutation rate. The SSM model (`1`) changes the existing allele number (k) to the $k+1$ or $k-1$ value randomly. The boundaries are reflexives, the allelic value can not exceed the `ntrl_all` value or be less than 1. The KAM model (`2`) modifies the existing allele by assigning it a new random value within the `[1, ntrl_all]` range.

ntrl_init_model [0,1] (opt)

This option sets the way neutral loci are initialised in the population at the start of each replicate. There are two modes, `0` (no variance) and `1` (maximum variance). Mode `1` is the default mode. With mode `0`, all alleles as set to the same initial allele value (i.e., allele 1) such that the whole population is monomorphic. Mode `1` randomly sets each allele within the range `[1, ntrl_all]`. This option doesn't allow the initial values to differ between patches. Patch-specific initial allele values can be set with the `ntrl_init` LCE (see section 5.9.2) but only for di-allelic loci.

ntrl_recombination_rate, ntrl_genetic_map, ntrl_random_genetic_map (opt)

Recombination is handled by the genetic map. All genetic map parameters apply. See section 6.1.

ntrl_save_genotype [string] (opt)

If this parameter is present, the population genotypes will be saved in a text file with the `".txt"` extension or `".dat"` extension if option FSTAT is selected. Multiple file formats are proposed, depending on the argument passed to this parameter (capital or non-capital letters are accepted):

- TAB (tab) (default)

  The allelic values are saved on one line per individual and two columns per locus. Extra data is saved for each individual: age (0 = offspring, 2 = adult), sex (0 = male, 1 = female), home (natal patch), ped (see p66), isMigrant (number of immigrant parents), father, mother, ID; father and mother contain unique ID's of the parents of the individual with identifier ID, for pedigree reconstruction.

- SNP_TAB (snp_tab)

  The genotypes of di-allelic loci are saved in the classical SNP format 0, 1, or 2 for each locus, where 0 stands for homozygous '00' genotype (wild or ancestral type), 1 is for any of the two heterozygous genotypes '10' or '01', and 2 stands for the alternative (mutant) genotype '11'. This way, the phasing information is lost in the heterozygotes but this format corresponds to the format used in many bioinformatics tools. The same tabular file format is used as for the "TAB" option above.

- FSTAT (fstat)

  The file format is (almost) the same as that used by the FSTAT program (Goudet 1995). It adds information about each individual (age, sex, pedigree, and natal patch) to the genotype data that is not part of the original FSTAT file format. This extra information is used to correctly assign individuals to sex and age containers when sourcing a population from an input FSTAT file. An example of an output file is given below.

- GENEPOP (genepop)

  Same as for the FSTAT option, but saves the data in GENEPOP format (Reymond & Rousset 1995). The same individual data is added to the genotype data as for FSTAT files.

- PLINK (plink)

  It is possible to save di-allelic loci information to a PLINK1.9 `".ped"` file. The accompanying `".map"` is also generated (see https://www.cog-genomics.org/plink/1.9 for details). The .ped file stores pedigree information in the first four columns, the sex and phenotype value in the next two columns (phenotype is set to `-9` as missing). The allele values are then recorded in the next 2×`ntrl_loci` columns, with letter `A` for allele 1 and `G` for allele 2. The .map files records the chromosome ID, locus ID, map position (cM), and base-pair ID. These values depend on the parameters passed to the genetic map. Locus and base-pair ID are identical. If loci are unlinked (the genetic map is not used), then a map with a single chromosome is created, with 50 Morgans between each locus.

ntrl_save_freq [locus, allele] (opt)

The default locus option saves the per-locus variance components of the Weir & Cockerham (1984) F-statistic estimators (siga, sigb, sigw), along with their F_{ST} and F_{IS} values, the whole population (het) and patch-specific heterozygosity (het.p$_i$), the identity of the major allele (maj.al), its overall frequency (pbar.maj.al), and its patch-specific frequencies (freq.maj.p$_i$).

The allele option saves similar data for all *extant* alleles at each locus. The file contains the overall allele frequency (pbar) and heterozygosity (het), the Weir & Cockerham (1984) variance components and F-statistic estimators (siga, sigb, sigw, Fst, Fis), and the patch-specific heterozygosities (het.p$_i$) and frequencies (freq.p$_i$).

The file extension is `".freq"`. Each line contains the information for one locus or one allele at a time, including the locus and allele identifiers.
NOTE: if the population contains both adult and offspring individuals at the time of writing the file, only the offspring are used to calculate the statistics.

ntrl_save_fsti [bool] (opt)

This tells Nemo to save the within patch F_{ST} values per-locus using the Weir & Hill (2002) estimates (see note below). Each line of the output text file contains the values of a specific locus and each column is for a different patch. The first line takes the column labels. The file extension is `".fsti"`.

ntrl_output_dir [string] (opt) This parameter specifies a specific path used to save the genotype and 'fsti' output files. Should not end with a slash ('/').

ntrl_output_logtime [integer] (opt)

This is the frequency at which output files are produced during a simulation, set in generation time. Alternatively, the generations at which the files should be saved can be provided as multiple values in an array (e.g. `{{1,2000,10000}}` write output file at generation 1, 2000, and 10,000).

Note about reading an FSTAT file: as discussed in section 4.2.2, it is possible to load a population from genetic data saved in an FSTAT file. That file can use the original or the *extended* file format as described here. The original file format (Goudet 1995) does not include the `age`, `sex`, `ped`, and `origin` “loci”. Here is an example of the *extended* FSTAT file format :

```
5 9 20 2
loc1
loc2
loc3
loc4
loc5
age
sex
ped
origin
1 1414 1019 2002 0820 0307 0 1 1 1
1 0814 0219 2002 2020 0307 0 1 1 1
1 0808 0217 1902 0820 0907 2 1 2 1
1 0820 0209 1902 0805 0918 2 0 0 4
[...]
4 0307 1308 0220 0401 0115 0 1 1 4
4 0905 1213 0302 0312 0506 2 1 2 2
[...]
5 2017 1010 2013 1812 1505 0 0 1 5
5 2017 1008 2013 1811 1505 2 1 2 3
```

The first line contains the population number (5 pops here), the number of locus (5+4), which corresponds to the number of columns saved (minus the first one), the maximum number of alleles per locus (20) and the number of digits used to write each allele value.

The five next lines are the locus names plus the “locus names” for the four last values; the age, sex, pedigree class and population of origin of each individual. This extra information is not processed by the FSTAT program and should thus be removed to be used with that program. It is however extremely useful when using this file format to load a new population from a saved simulation file. The individuals information will thus be used to assign the individuals to their respective sex and age classes.

The following lines contain the individual's info, one individual per line. The first number is the population number in which the individual finds itself at the time of the recording. The 5 next numbers/columns are the genotype values of each of the 5 loci. As, in this example, we are using two digits per allele, the first two digits of a locus genotype number are the first allelic value (e.g. allele #14 for the first allele of the first locus of the first individual) while the two next digits are the second allelic value as individuals are diploids here (e.g. allele #14 for the second allele of the first locus of the first individual). Each line ends with four numbers. The first is the age class (0 = offspring, 2 = adult), the second is the sex tag (1 = female, 0 = male), the third is the individual's pedigree class, that is the pedigree relationship of its parents (0 = parents from different demes, 1 = parents from same deme but unrelated, 2 = parents are half-sib, 3 = parents are full-sib, and 4 = selfed mating), and the last one is the identifier of the population where that individual was born.

This file format is close to the `FSTAT` input file format (Goudet, 1995) (see Jérôme Goudet's software `http://www2.unil.ch/popgen/softwares/fstat.htm`) with the addition of the four last columns of the individual data. The `HIERFSTAT` **R** package (see: `http://www2.unil.ch/popgen/softwares/hierfstat.html`), by the same author, provides **R** routines (called `read.fstat.data`) to extract data from an `FSTAT` file within the **R** software (`http://www.r-project.org`).

Note about the statistics: Nemo lets the user choose between various estimates of gene diversity and genetic differentiation both within and between populations. The classical F-statistics are available by using the `'fstat'` stat option (see section 8.2 for more details). This option will give the estimates of heterozygosities (H_{O}, H_{S} and H_{T}) and of F-statistics (F_{IS}, F_{ST} and F_{IT}) using the weighting method of Nei and Chesser (1983) for unbiased estimates when population sizes vary.

Another set of F-statistics is given by the `'weighted.fst'` stat options that use the Weir and Hill (2002) unbiased estimates of within and between populations F_{ST}'s for varying sample sizes. These stat options may be used to output the whole population matrix of pairwise F_{ST} values (within and between populations). The mean total population weighted F_{ST} is also given (and may be different from the previous estimate using Nei and Chesser (1983)). That last value will be similar to Weir and Cockerham (1984) estimate when sample sizes are equal. Note that since version 2.0.8, the Weir and Cockerham (1984) F_{ST} estimate (θ) is also available (stat option: `'fstWC'`.

Finally, the within (θ) and between (α) population coancestry coefficients can also be directly computed using the `'coa'` stat options. These stats are sometimes referred as "kinship" or "allele sharing" coefficients. They use the explicit pairwise comparisons of individual sequences to compute the mean population θ's and between populations α's. This method will give exactly the same estimates of the within and between demes F_{ST} values using the Weir and Hill (2002) estimates but is more demanding of computer time. On the other hand, coancestries are given for smaller groups of individuals such as within and between sex or within pedigree classes (e.g. full-sib or half-sib coancestries, etc.). The F_{ST} estimates can be computed from the coancestries as follows:

$$F_{\mathrm{ST}} = \tfrac{\theta-\alpha}{1-\alpha},\ F_{\mathrm{ST}ii} = \tfrac{\theta_{ii}-\alpha}{1-\alpha},\ F_{\mathrm{ST}ij} = \tfrac{\theta_{ij}-\alpha}{1-\alpha},$$

with i and j are population indices with $i \neq j$. These estimates will be equivalents to the Weir and Hill (2002) estimates.

6.3 Quantitative traits

name: **quant**
files: `".quanti"` (output only)
phenotype: continuous value on $\mathbb{R}$.

Quantitative traits are traits affected by many loci and with a continuous distribution of phenotypic values in a population. Quantitative traits are also called *complex* or *polygenic* traits. A classic example is body size, or any phenotype that varies continuously among individuals in a population. The quantitative trait implementation in NEMO is based on two different allelic models: the continuum-of-allele (or *continuous*) and di-allelic models. The continuum-of-allele model consists in loci bearing mutations drawn from a Normal distribution with a potentially infinite number of alleles per locus. As such, it is also called the *infinite-allele* model. By mutation, the effect of the alleles at a continuous locus can grow or shrink during evolution, and are expected to be more or less normally distributed. The di-allelic model allows for only two allelic values per locus, usually pre-determined. Mutations then swap alleles between these two values. These models are classically used in quantitative genetics theory, hence their interest here. Quantitative loci can also affect multiple phenotypes and thus be **pleotropic**, with varying degree of pleiotropy among loci. **Dominance** and **epistatic** effects can be added to the **additive** effects at the loci. The dominance effects can be different for each trait affected by pleiotropic loci. The epistasis model follows the multi-linear model (Hansen and Wagner, 2001; Carter et al., 2005, see subsection 6.3.5) and is currently limited to non-pleiotropic loci and pairwise locus interactions (no higher order epistasis). However, both epistasis and dominance are allowed to affect the genotypic values of the traits.

Trait value: The phenotypic value P of an individual is:

$$P = G + E,$$

with G the genotypic value and E the environmental deviation. In all cases, G is given by the addition of all allelic values at the loci coding for the trait within one individual: $G = \sum_j^L (a_{j,x} + a_{j,y})$ (for L diploid loci, $a_{j,x}$ additive effect inherited from the mother, $a_{j,y}$ additive effect inherited from the father at locus j). With **dominance**, the genotypic value depends on the dominance deviation at each locus, modeled as a deviation from the additive value of the heterozygote genotype: $G = \sum_j^L (a_{j,x} + a_{j,y} + d_{j,xy})$. The dominance deviation $d_{j,xy}$ is computed following two equivalent models (see subsection 6.3.4). The homozygote values are as in the additive case. The **epistatic** model uses these genotypic values to compute the epistatic deviation using a linear scaling (see subsection 6.3.5). Finally, the environmental deviation E is a random effect drawn from a normal distribution with mean 0 and variance σ_e^2 or V_E. Environmental effects are optional. A value for V_E different from zero will set the **heritability** of the trait different from 1. Several options exist to set the narrow or broad sense heritability constant during a simulation, also when loci are dominant (see Heritability section below).

Multiple traits: When multiple traits are modeled, the loci can be **pleiotropic**, meaning that they can affect multiple traits. The pleiotropic models mostly applies to *continuous* loci, although di-allelic loci can affect up to two phenotypic traits. Loci need not be all pleiotropic, as in the case of the *variable pleiotropy* model, and can also only affect a single of the many phenotypic traits modeled. There is no limit to the number of traits modeled.

Correlated traits: In the continuum-of-allele model with pleiotropy, mutations at pleiotropic loci are drawn from a multivariate Normal distribution, which can be used to set the correlation between mutational effects. In this way, the evolution of correlated traits and genetic constraints on adaptation can be modeled, together with the multivariate Gaussian selection model in the selection LCE. Moreover, the pleiotropic degree of each quantitative locus can be set via a pleiotropy matrix, mapping loci to traits. If loci vary in their pleiotropic degree, then the mutation matrix of each locus also varies and must be set explicitly. Locus-specific mutational variance and co-variance can be set for all pleiotropic loci. This allows for great flexibility in the genetic architecture of quantitative traits. Note that di-allelic loci can also have correlated mutations when affecting two traits.

Statistics: The statistics implemented for the `quant` trait return the phenotypic mean value and variance (V_p) within populations, the additive (V_a) and genotypic (V_g) genetic variance within populations, the among populations genetic variance

(V_b), the Q_{ST} index of trait differentiation among populations ($Q_{\mathrm{ST}} = \frac{V_b}{V_b+2V_a}$), and the traits' genetic correlation, along with the eigenvalues and eigenvectors of the **G**-matrix within demes, when two or more traits are modeled.

6.3.1 Traits, loci and alleles

Parameters:

quanti_traits [integer]

The number of phenotypic traits to model. The number of traits is not limited. If two or more traits are modeled, the loci can be pleiotropic depending on the option set with `quanti_pleiotropy`. The number of traits set here will also determine the dimensionality of selection set in the `viability_selection` LCE (section 5.7).

quanti_loci [integer]

Total number of loci that determine the trait(s). Loci are diploid. That number may be divided by the number of traits in the "no-pleiotropy" case to set an equal number of loci per trait unless `quanti_loci_per_trait` is specified. Otherwise, the traits are affected by the same total number of loci under "full-pleiotropy" or by a number of loci that depends on the precise gene-trait mapping set with `quanti_pleio_matrix` for the "variable pleiotropy" case (see section Pleiotropy). In any case, this parameter must specify the total number of quantitative loci in the genome, counting each pleiotropic and non-pleiotropic locus affecting the traits (a pleiotropic locus counts as one locus irrespective of its pleiotropic degree).

quanti_loci_per_trait [integer / matrix] (opt)

An array specifying the number of loci per trait when the total number of loci is not equally distributed among traits. This is only used when loci are non pleiotropic but multiple traits are modelled such that different sets of loci can affect different traits. The sum of loci specified here must match the total number of loci specified with `quanti_loci`.

quanti_mutation_rate [decimal]

The mutation rate, identical for all loci. The mutation effect(s) depends on the allelic model, set above with `quanti_allele_model`.

quanti_allele_model [diallelic, diallelic_asymmetrical, <u>continuous</u>, continuous_in_place] (opt)

There are two main models of mutational effects at quantitative loci: `diallelic` if mutations can only take two discrete values (e.g., $\pm a$), or `continuous` if mutations are drawn from a Normal distribution, with specific variance (and correlation for pleiotropic loci). These models allow setting different distributions of allelic effects at quantitative loci with potentially different evolutionary dynamics.

The default option "`continuous`" corresponds to the classic continuum-of-allele model (Kimura, 1965), where loci have a distribution of allele values on the trait(s), and mutations are added to the existing allele values, allowing for a build-up of allelic effects over time (which may cause some loci to have larger genotypic effects than others). The alternative option "`continuous_in_place`" also draws the mutation from a normal distribution but replaces the existing allele at a locus with the mutation value instead of adding to it.

The "`diallelic`" option corresponds to the classic di-allelic model where locus i carries two alleles with effects $\pm a_i$ on the trait. Mutations then swap the sign of the allele at the locus, back-and-forth. The alternative model "`diallelic_asymmetrical`" allows loci to hold allelic values that are not symmetrically set around zero but where one of the alleles has value 0. This way, the allele with value 0 represents the "ancestral" state and an alternative value different from 0 can be specified for the "derived" allele. For instance values can be $\{0, 0.02\}$ or $\{0, 356\}$, arbitrarily (here with different values depending on the phenotypic scale of the trait modeled). The allele *identities* are $\{0, 1\}$ when referring to the allelic state instead of their values, for instance when saving the genotypes to a text file. Allele '0' is the trait-increasing allele ($+a$) in the `diallelic` model or the ancestral allele in the `diallelic_asymmetrical` model. Allele '1' is the alternative/derived allele.

6.3.2 Pleiotropy

This section describes the parameters used to set the gene-to-trait connectivity matrix (the genotype-phenotype map), and the way the quantitative loci influence the traits via mutations.

quanti_pleiotropy [n(o) || f(ull) || v(ar)(iable)] (opt)

The pleiotropic model is set to "no" by default, meaning loci affect only one trait at a time. There are three mutually-exclusive possibilities:

n [no]

By default, the loci affect a single trait at a time. The number of loci per trait is deduced from the total number of loci and number of traits, or is set explicitly with `quanti_loci_per_trait`. The mutation variance can be set for each locus separately. Note that loci affecting different traits cannot be mixed on the genetic map. The loci affecting each trait are contiguous on the genetic map. This can be avoided by using the variable pleiotropy model with a pleiotropy matrix setting the trait affected by each locus on the genetic map.

f [full]

Full pleiotropy. All loci affect all traits. A single mutation matrix can be used for all loci.

v [var || variable]

Variable pleiotropy. Each locus can affect a different set of traits and have a different pleiotropic degree and mutational characteristics. The connectivity matrix, or pleiotropy matrix, informing which trait is affected by each locus must be set (see `quanti_pleio_matrix` below). The mutation matrix of each locus can be set explicitly according to its pleiotropic degree (see section Mutations).

quanti_pleio_matrix [matrix] (opt)

The matrix passed must have as many rows as the total number of loci in the model (set with `quanti_loci`), and as many columns as the number of traits modeled (set with `quanti_traits`). The values must be either 0 or 1 to indicate, row-wise, which trait is affected (1) or not affected (0) by a locus. The pleiotropic degree of a locus is then obtained by summing the elements on its row of the pleiotropy matrix. Universal pleiotropy should not be modeled via the pleiotropy matrix. It is not necessary in that case (model option full of parameter `quanti_pleiotropy`), nor is it in absence of pleiotropy (model option no of parameter `quanti_pleiotropy`). However, in that latter case, the pleiotropy matrix can be used to mix non-pleiotropic loci affecting different traits on the genetic map. Otherwise, non-pleiotropic loci are grouped by trait on the genetic map (see comments for `quanti_pleiotropy` models).

6.3.3 Additive effects

By default, quantitative loci are additive; The genotypic value of a trait is the sum of the alleles present at the loci affecting it. The allelic values are not explicitly set in the case of the *continuum-of-allele model* (*continuous* loci), where they depend on both the initial trait values and the mutation effect size parameters. Initial trait values are set with `quanti_init_value` as discussed in subsection 6.3.7. In the case of di-allelic loci, the allele values are set explicitly with parameter `quanti_allele_value` as described here. The initialization of the di-allelic loci is also discussed in subsection 6.3.7.

quanti_allele_value [decimal/matrix] (opt)

This parameters holds the additive effect of the alleles at di-allelic loci. If a single value is given, that value is used for all loci (with a warning). A matrix can be used to pass locus-specific allele values with one or two rows. For the `diallelic` model, an array holding the $+a$ values should be passed. The allele values are then $\pm$ the given value at each locus. For the `diallelic_asymmetrical` model, an array holding the "derived" allele values should be passed, assuming that the ancestral allele value is 0. A matrix with two rows can also be passed to specify the two allelic values at each locus when the ancestral value should depart from 0 (in that case, allele '0' is the allele with smallest value). In all cases, if the number of elements in the array (or rows) does not match the number of loci, the allelic values provided will be recycled (copied) across blocs of loci.

6.3.4 Dominance

Dominance at the quantitative loci can be added to additive effects, for any type of allele model. With the following parameters, you can set dominance differently at each locus and for different traits, either randomly or specifically during initialization. The dominance effects at pleiotropic loci can be trait specific.

The dominance model implemented computes the genotypic value at a locus (g_i) as a function of the two additive effects $\{a_1, a_2\}$ (maternally and paternally inherited) and the dominance coefficient k_i at the locus:

$$g_i = a_1 + a_2 + k_i|(a_2 - a_1)|, \text{ with } a_1 < a_2.$$

The dominance of a locus depends on its k value as follows:

$$\begin{aligned} k < -1 &: \text{underdominance} \\ k = -1 &: \text{the smaller allele } (a_1) \text{ is dominant} \\ k = 0 &: \text{purely additive} \\ k = 1 &: \text{the bigger allele } (a_2) \text{ is dominant} \\ k > 1 &: \text{overdominance} \end{aligned}$$

The k values are set explicitly with the following parameter:

quanti_dominance_effects [decimal/matrix] (opt)

The parameter accepts either a single k value then copied to all loci and traits, an array of maximum L values of k (L = number of quantitative loci) then copied to all traits, or a *trait* $\times$ *locus* matrix of L values of k or less. In all cases, less than L values will be recycled across loci.

With this setting, trait-specific dominance effects at pleiotropic loci can be specified. This let us model traits that differ in their amount of dominance variance. The same applies to non-pleiotropic loci affecting different traits. Here are a couple examples:

Example 1: non-pleiotropic loci affecting multiple traits with different number of loci per trait. We model 4 traits, affected by different set and number of QTL. It is not necessary (and not possible) to provide a matrix of k values with 4 rows and different number of elements in each row. **For the case of non-pleiotropic loci, it is sufficient to provide a single array of L values**.

```
quanti_traits             4
quanti_loci               60
quanti_loci_per_trait     {{15,5,30,10}}
quanti_pleiotropy         no
quanti_dominance_effects {{ rnorm(10,0.05,0.1), rnorm(5,-0.1,0.05),
                            rnorm(30, 0, 0.05), rep(0,10) }}
[...]
```

We here set the dominance effects randomly, with different parameter values for each trait (arbitrarily here). Note the use of the macros to generate those random values and the fact that the last 10 loci have no dominance. The actual values will be saved in the '.log' file of the simulation (see section 3.8). All values could have been generated with a single call to `rnorm(60,0,0.1)` or been constant when provided as a single value (*e.g.*, `quanti_dominance_effects 0.2`).

Example 2: variable pleiotropy at loci affecting different sets of traits. The whole matrix of dominance effects should be provided, although missing values (either for traits or loci) will be copied from the provided values. Therefore, k values for traits not affected by a given locus will be set in the matrix even if not used to compute the traits' genotype. In such cases, setting $k = 0$ is best although it doesn't matter. The difficulty when setting the dominance effect matrix is to read the pleiotropy matrix upside down. Indeed, the first is a *trait* $\times$ *locus* matrix but the latter is a *locus* $\times$ *trait* matrix.

```
quanti_traits             3
quanti_loci               4
quanti_pleiotropy         var
# pleiotropic matrix is [locus x trait]
quanti_pleio_matrix       {{1,0,1}  #loc 1, PD=2
                           {1,1,1}  #loc 2, PD=3
                           {0,0,1}  #loc 3, PD=1
                           {1,1,0}} #loc 4, PD=2
```

```
# dominance matrix is [trait x locus]
# locus 1 is dominant on trait 1 only
# locus 2 receives 3 random k values
# locus 3 is additive (k=0)
# locus 4 is under-dominant
quanti_dominance_effects {{1,rnorm(1,0,0.1),0,-1.2} # trait 1
                          {0,rnorm(1,0,0.1),0,-1.2} # trait 2
                          {0,rnorm(1,0,0.1),0,0} }  # trait 3
```

Example 3: full pleiotropy. This case is simplest since all loci affect all traits. The matrix of k values is then easier to interpret and can be specified as an easily generated *trait* $\times$ *locus* matrix (here set between -1 and 1 using `runif()`).

```
quanti_traits            5
quanti_loci              1000
quanti_pleiotropy        full
quanti_dominance_effects matrix(q(runif(5000,-1,1)),ncol=1000,nrow=5)
```

6.3.5 Epistasis

Epistatic effects can be added to the additive and dominance effects. Nemo implements a model of pairwise epistatic interactions between quantitative loci. The epistatic model is based on the *multilinear* model of Hansen and Wagner (2001) where the genotype of an individual is given by the following sums over all loci:

$$G = \sum_i g_i + \sum_i \sum_{j>i} \epsilon_{ij} g_i g_j,$$

with additive genotypic value at locus i : $g_i = a_{i,x} + a_{i,y}$ and epistatic effect between locus i and j : ϵ_{ij}. Note that since dominance can also be modeled at the quantitative loci, dominance effects will be included in the genotypic values g_i. Dominance can thus also affect epistatic interactions in this implementation. Epistasis is currently not implemented for pleiotropic loci.

Epistatic effects can be specified with the following parameter.

quanti_epistatic_coefs [array] (opt)

The epistatic coefficients (the ϵ_{ij}'s) must be specified explicitly with an array of decimal numbers. The array is expected to hold $(L(L-1)/2)$ ϵ_{ij} values for the L quantitative loci set with `quanti_loci`. The array corresponds to the lower-triangular matrix of pairwise locus interactions (which is symmetrical by definition). Therefore, the $(L-1)$ first elements of the array are the $\epsilon_{1,j>1}$

coefficients for all pairs containing the first locus. The $(L-2)$ next elements are the coefficients involving the second locus minus the one already specified with the first locus, and so on for each locus barring the already set pairwise coefficients. For large L, it is easier to store the array of coefficients in a separate file or to randomly generate them with a macro, much like the dominance effects (see subsection 6.3.4 above).

6.3.6 Mutations

The mutation process depends on the allelic and pleiotropy models. The simplest case is when loci are **di-allelic** where only the mutation rate needs to be set. Mutations then proceed by swapping the identity of the affected allele between the two allelic states at the affected locus (e.g., $+a$ becomes $-a$ and vice-versa). Note that di-allelic loci can also be pleiotropic, affecting a max of two phenotypic traits. In that case, the correlation between mutational effects at a locus can be set with `quanti_mutation_correlation` (see 6.3.2 below). The correlation is interpreted as the probability of received twice the same allele for each trait (or same allele type, swapping between allele 0 (derived) and 1 (ancestral)).

In the continuum-of-allele model, mutations at a locus are drawn from a Normal distribution. The distribution will be univariate Normal if the affected locus is non-pleiotropic or multi-variate Normal if the locus is pleiotropic. In all cases, the variance of the distribution must be set, while the mean is always zero. The variance-covariance matrix (the **M**-matrix) can be fully specified for the pleiotropic loci, thus allowing for correlated mutational effects on the phenotypic traits affected.

The parameters `quanti_mutation_variance` and `quanti_mutation_correlation` are meant to be shortcuts to set mutational matrices with equal variance and covariance among all traits affected. If used with the variable pleiotropy model, the **M**-matrix at each locus and its dimensionality will be set according to the pleiotropic degree of the loci, automatically. To vary the variance and covariance values of the **M**-matrix at each locus, the parameter `quanti_mutation_matrix` must be used. Also, the **M**-matrix must be used to vary the per-trait mutation variance (diagonal elements of **M**) or the between traits covariance (off-diagonal elements of **M**).

Note that in no case will the mutations affect the set dominance or epistatic effects. Those are constant throughout a simulation.

quanti_mutation_rate [decimal] *mandatory*

See description in subsection 6.3.1.

quanti_mutation_variance [matrix or decimal] (opt)

This is the variance of the Normal distribution of the mutational effects (the mutation effect size) in the *continuous* allele model. If a single value is passed, then the same variance is used for all loci (and traits). To set locus-specific variances, an array must be passed with a set of values (recycled across loci). Those values will be equal across traits for a given locus. To set trait-specific mutational variances at pleiotropic loci, the full mutation matrix must be specified with `quanti_mutation_matrix` (see below).

For non-pleiotropic loci affecting a set of different traits, a multi-row matrix can be passed where each row sets the variance of the mutations at the loci affecting the different traits, each corresponding to a different row. A row with a single value will set the same mutational variance at all loci affecting a specific trait, while a row with multiple values will set different mutational variance at the loci affecting that trait (see Example 2 below).

Example 1: Pleiotropic loci with locus-specific variance. In this example, we use the macro `runif()` to randomly draw the 10 variances from a uniform distribution between 0.01 and 0.1:

```
quanti_traits             5
quanti_loci               10
quanti_allele_model       continuous
quanti_pleiotropy         full
quanti_mutation_variance  {{runif(10,min=0.01,max=0.1)}}
```

Here, each locus receives a mutation matrix of the form:

$$\mathbf{M} = \begin{bmatrix} var_1 & 0 & 0 & 0 & 0 \\ 0 & var_2 & 0 & 0 & 0 \\ 0 & 0 & var_3 & 0 & 0 \\ 0 & 0 & 0 & var_4 & 0 \\ 0 & 0 & 0 & 0 & var_5 \end{bmatrix}.$$

Example 2: Non-pleiotropic loci with trait-specific variance. Each set of 2 loci affecting each trait will have the same mutational variance because the matrix has five rows and a single column:

```
quanti_traits             5
quanti_loci               10    # 2 loci/trait
quanti_allele_model       continuous
quanti_pleiotropy         no
quanti_mutation_variance  {{0.02}{0.03}{0.1}{0.2}{0.3}}
```

Example 3: Non-pleiotropic loci with locus- and trait-specific variance:

```
quanti_traits            5
quanti_loci              10    # 2 loci/trait
quanti_allele_model      continuous
quanti_pleiotropy        no
quanti_mutation_variance {{0.02,0.03}{0.1,0.2}{0.3,0.2}{0.05,0.06}\
                          {0.08,0.1}}
# or with a macro:
quanti_mutation_variance matrix(q(rnorm(10, mean=0.05, sd=0.01)),\
                                nrow=5, ncol=2)
```

Note that we can also use a single array with all variances:

```
quanti_traits            5
quanti_loci              10    # 2 loci/trait
quanti_allele_model      continuous
quanti_pleiotropy        no
quanti_mutation_variance {{0.02,0.03,0.1,0.2,0.3,0.2,0.05,0.06,\
                           0.08,0.1}}
```

quanti_mutation_correlation [double] (opt)

The correlation of the effects of pleiotropic mutations, when two or more traits are modeled. It applies to both the di-allelic (two traits only) and the continuous allele models. For the di-allelic case, the correlation is interpreted as the probability of having the same allele by mutation. For the continuous model, the correlation is transformed into a covariance (using the value of `quanti_mutation_variance`) to build the mutation matrix.

If an array is passed, the values will be interpreted as locus specific trait correlations affecting the locus specific mutation matrices. The resulting matrix will have all covariance set according to the locus specific correlation set here and the mutation variance set with `quanti_mutation_variance`, which can also be locus specific.

quanti_mutation_matrix [matrix] (opt)

The parameter is used to specify the variance-covariance matrix of the multivariate Normal distribution used to draw the mutation effects in the continuous allelic model, also called the **M**-matrix. It can be used to set different mutational variance and covariance for the different traits and the different loci.

For the *full* pleiotropy model, the matrix is a square symmetrical and semi-definite positive matrix with trait mutational variance on the diagonal and the mutational covariance off the diagonal. A single **M**-matrix can be passed for this model, in which case it applies to all loci equally. The locus-specific variance and covariance are set with the combination of parameters quanti_mutation_variance and quanti_mutation_correlation as explained above.

For the *variable* pleiotropy model, the matrix has rows of variable length each containing the **M**-matrix at each locus. The locus specific **M**-matrices (the rows) are thus written as arrays. The length of a row then depends on the pleiotropic degree n of the locus for which it is set and on whether covariance values are set or not. The array first holds the n trait-specific mutational variances corresponding to the diagonal elements of the locus specific **M**-matrix. The variances are optionally followed by the $[(n-1)*n]/2$ covariance values among the n traits affected by the focal locus. These covariance values correspond to the upper triangle of the locus specific **M**-matrix provided row-wise (e.g., $cov_{1,2}, cov_{1,3}, cov_{2,3}$ for $n = 3$). A single value can also be passed, in which case the **M**-matrix will be a diagonal matrix with identical variance along its diagonal and covariance values set to zero. If only n values are passed, then the **M**-matrix will also be a diagonal matrix, this time with n different variances along its diagonal and covariance values set to zero.

For the *no* pleiotropy model, only the variance matters, set with quanti_mutation_variance.

Example 1: Full pleiotropy:

```
quanti_traits          4
quanti_loci            4
quanti_pleiotropy      full
quanti_allele_model    continuous
quanti_mutation_matrix {{0.1, 0.05, 0.05, 0.05}
                        {0.05, 0.1, 0.05, 0.05}
                        {0.05, 0.05, 0.1, 0.05}
                        {0.05, 0.05, 0.05, 0.1}}
```

In that simple case, it would be easier to use a macro:

```
quanti_mutation_matrix smatrix(0.05, 4, diag=0.1)
```

Example 2: Variable pleiotropy: locus-specific **M**-matrices are specified in *row* format: $\{var_1, var_2, var_3, cov_{1,2}, cov_{1,3}, cov_{2,3}\}$ (for a locus with PD=3). Note that covariance terms are optional.

```
quanti_traits         4
quanti_loci           4
```

```
quanti_pleiotropy    variable
quanti_allele_model  continuous
quanti_pleio_matrix  {{1,0,1,1}  #loc 1, PD=3
                      {1,1,1,0}  #loc 2, PD=3
                      {0,0,0,1}  #loc 3, PD=1
                      {1,0,0,1}} #loc 4, PD=2
quanti_mutation_matrix {{rep(0.1,3), rep(0.025,3)} #same var + cov
                        {0.05,0.1,0.3,0.032,0.055,0.078} #var + cov
                        {0.1}  # var = 0.1, no cov (PD=1)
                        {0.1}} # var = 0.1, cov = 0 (PD=2)
```

The resulting M-matrices from the above initialization code are:

$$\mathbf{M}_{loc1}\begin{bmatrix}0.1 & 0.025 & 0.025\\ 0.025 & 0.1 & 0.025\\ 0.025 & 0.025 & 0.1\end{bmatrix}; \quad \mathbf{M}_{loc2}\begin{bmatrix}0.05 & 0.032 & 0.055\\ 0.032 & 0.1 & 0.078\\ 0.055 & 0.078 & 0.3\end{bmatrix};$$

$$\mathbf{M}_{loc3}\begin{bmatrix}0.1\end{bmatrix}; \quad \mathbf{M}_{loc4}\begin{bmatrix}0.1 & 0.0\\ 0.0 & 0.1\end{bmatrix}$$

Example 3: Constant variance and covariance but variable pleiotropy:

```
quanti_traits        4
quanti_loci          4
quanti_pleiotropy    variable
quanti_allele_model  continuous
quanti_pleio_matrix  {{1,0,1,1}  #loc 1, PD=3
                      {1,1,1,0}  #loc 2, PD=3
                      {0,0,0,1}  #loc 3, PD=1
                      {1,0,0,1}} #loc 4, PD=2
quanti_mutation_variance    0.1
quanti_mutation_correlation 0.5
```

which would result in all loci having matrices of the form:

$$\mathbf{M}_{PD=3}\begin{bmatrix}0.1 & 0.05 & 0.05\\ 0.05 & 0.1 & 0.05\\ 0.05 & 0.05 & 0.1\end{bmatrix}, \text{ or} \quad \mathbf{M}_{PD=2}\begin{bmatrix}0.1 & 0.05\\ 0.05 & 0.1\end{bmatrix};$$

6.3.7 Initial trait values

The initial trait values can be set either to a unique value or to a defined average with some variation around that average. The different initialization modes are controlled by `quanti_init_model`. By default, the allelic values at *continuous* loci are first initialized to 0 unless a specific non-zero value is set with `quanti_init_value`. The initial variance of the traits can also be set to a specific value with `quanti_init_variance`. For *diallelic* loci, the default initialization consists in assigning the allele values corresponding to allele '0' (ancestral) at each locus. The convention for di-allelic loci is that the first allele (allele '0') is the trait-increasing value ($+a$) in the `diallelic` allele model or the ancestral value (0) in the `diallelic_asymmetrical` allele model. For example in that last case, if the allele values at a locus is $\{0.1\}$, then $a_0 = 0,\ a_1 = 0.1$.

The initialization process described here applies equally to all patches in the population. Patch specific trait or allele frequency initialization is provided by the quanti_init LCE.

We will separate the description of the process for the two allele types, continuous and di-allelic since the interpretation of the parameter values differs slightly.

6.3.7.1 Initialization of continuous-allele loci

quanti_init_model [0,1,2] (opt)

For the continuum-of-allele model, the default initialization model **0** results in a monomorphic population where all individuals have the same initial trait values as specified by the `quanti_init_value` parameter (or 0 if no value was specified). If set to **1**, a single random mutational effect is added to each locus, on top of the initial value. The mutated allele is chosen randomly between the two homologous copies at each locus. The mutation parameters of the traits apply here. Model **2** sets the initial mean and variance of the trait in the population by adding a random deviate to the initial allelic values at each locus drawn from a normal distribution with zero mean and variance equal to `quanti_init_variance`/(2L), with L = num. of quantitative loci.

quanti_init_value [matrix] (opt)

The initial genotypic value of the trait(s) can be set here for *continuous* loci. It is 0 by default (all loci receive {0, 0} as allele values). Trait-specific initial values can be passed as an array of values, with one *genotypic value* per trait (i.e., sum of additive effects at underlying loci). The value that each locus eventually receives is determined by dividing that initial genotypic value by two times the number of loci affecting a trait. Thus, all loci receive two identical alleles. The initial population will then be monomorphic for those trait values, unless the `quanti_init_model` is 1 or 2 (see below). As an example, if one sets `quanti_init_value 10` with `quanti_loci 20`, each locus will receive allele values `{0.25, 0.25}` in the whole population ($0.25 = 10/(2 * 20)$).

quanti_init_variance [decimal/array] (opt)

An initial trait variance can be set with this parameter. It can hold a single value applied to all traits or an array of trait-specific values. This parameter is necessary for the initialization model 2 described below for continuous loci.

6.3.7.2 Initialization of di-allelic loci

In the case of di-allelic loci, it is important to keep in mind that allele '0' (the first allele) refers to the trait-increasing value ($+a$) in the `diallelic` model and to the *ancestral* allele value (usually = 0) in the `diallelic_asymmetrical` model.

quanti_init_model [0,1,2] (opt)

For di-allelic loci, model **0** sets the two alleles at each locus to the first allele '0' (see above). The population will be monomorphic for that allele. Model **1** randomly assigns each allele at each locus with probability $P(+a) = 0.5$, maximizing initial trait variation, and Hardy-Weinberg genotype proportions. Model **2** ("polarized loci") initializes di-allelic loci as homozygote with opposite allelic values at alternative loci (e.g., all positive ($+a$) at one locus and all negative ($-a$) at the next). Models **0** and **2** generate monomorphic populations with not genetic variance at the trait(s) in the first generation. Note that allelic values are specified with `quanti_allele_value`.

quanti_init_value, quanti_init_variance These two parameters are not used for di-allelic loci. An error will be issued if they are set in the init file.

Initial trait average and variance: For continuous loci, the average trait value matches its initial value (or 0 if none is given in input). Individual trait values will differ by some amount when initial variation is added by models **1** and **2**. For di-allelic loci with option diallelic, the average and individual trait values will be $2\sum_L a_+$ in model **0**, zero in model **2**, and will vary around zero in model **1**. In the diallelic_asymmetrical case, the initial trait values will be zero in model **0**, will match the sum of average allelic values plus some variance around that average with model **1**. In model **2**, the mean trait value will be some value corresponding to the sum of the derived alleles across half of the loci if the ancestral value is set equal to 0. The initial allele frequencies can be changed with the quanti_init LCE, patch-wise.

Caveat: The initial trait distribution will have an important impact on the survival of the initial generation when selection is acting on the phenotype, for two main reasons. First, you should make sure that the initial trait mean in a patch is close enough to its phenotypic optimum as set with `selection_local_optima` (assuming the selection model is `gaussian` or `quadratic`, see section 5.7). Patch extinction will occur if selection is strong and the mean trait value is far from its optimum. This is particularly relevant for di-allelic loci where the mean trait value is zero in most cases and not straightforward to set to a different value unless the `quanti_init` LCE is used to adjust allele frequencies to match an expected average trait value. Second, you can expect a stark reduction in survival when large initial variation is modeled (*e.g.,* with initial models **1**). This is because extreme phenotypic values will be created when individuals start mixing their genomes. This may depress the initial mean absolute fitness in the population(s) and may cause population extinctions when patches are small or when the mean fecundity is small (in non-Wright-Fisher populations). Thus, it is better to double check what the mean fitness might be in a population at the first generation before running a simulation. To do so, the average fitness can be estimated from the formula of the selection model used in the `viability_selection` or related LCEs.

6.3.8 Heritability

The environmental variance of a quantitative trait needs to be set to be able to model traits with *heritability* different from 1. The narrow-sense heritability of a trait within a population is

$$h^2 = \frac{V_a}{V_a + V_i + V_e},$$

with V_a: additive genetic variance, V_i: interaction genetic variance ($V_i \neq 0$ only if dominance or epistasis are modeled), and V_e is the environmental variance. The sum $V_a + V_i = V_g$ gives the genotypic variance V_g of a population. The *broad* sense heritability is given by

$$H^2 = \frac{V_g}{V_g + V_e}.$$

The environmental variance V_e is set with the following parameter:

quanti_environmental_variance [double, array] (opt)

This sets the variance of the environmental deviation of the trait's phenotype, V_e. It is zero by default (no environmental variance). If set to a positive value, a random Gaussian deviate with mean zero is added to the genotypic value of the trait(s) when computing the phenotype(s) of each individual. A single value provided as argument will be reused for all traits. Trait-specific V_e's can be provided with an array, as in the following example:

```
quanti_traits                 3
quanti_environmental_variance {{0.2, 10, 650}}
```

Note that V_e should scale to V_a to set heritability $h^2 \in [0, 1]$ (see above). Thus, in the example above we expect the three traits to have very different phenotypic scales. The phenotypic scale mostly depends on the mutational variance (set with `quanti_mutation_variance` or `quanti_allele_value` for di-allelic loci), the number of loci and the mutation rate. In other words, it scale with the mutational variance of the trait:

$$V_m = 2L\mu\alpha^2,$$

with L, the number of loci (`quanti_loci`), μ, the allelic mutation rate (`quanti_mutation_rate`) and α^2, the mutation variance (`quanti_mutation_variance`).

When V_e is set here, it will remain constant during a simulation unless it is changed with temporal arguments. However, the parameter will be ignored when V_e is (re)set by LCE `quanti_modifier` to reach a specific h^2 value as explained in the next section.

6.3.8.1 Modelling constant heritability

Simulations with a fixed value of heritability can be modeled if V_e is set according to V_g during a simulation. To set V_e dynamically, you will need to add `quanti_modifier` in your life cycle. This LCE uses the heritability parameters presented here.

Setting V_e dynamically in a simulation can be tricky because V_g changes over time with changes in allele frequencies until it might reach an equilibrium in stable populations. It is easy without dominance or epistasis because, at linkage equilibrium, V_a can be computed directly as the within-population variance of the additive genotypic values. With dominance or epistasis, however, V_a will depart from the purely additive-effects variance because dominance and epistatic effects also contribute to V_a, besides their contribution to the non-additive variance V_i. Computing V_a with dominance or epistasis during a simulation is tricky in non-equilibrium population and with non-random mating. The state-of-the-art method is regression-based, a very inconvenient approach to implement in a simulation. Therefore, two approaches are proposed here. The first is to use a non-regression method based on theoretical expectations for a Wright-Fisher population. The second is to use the broad-sense heritability H^2 instead of the narrow-sense h^2 and thus directly compute V_g from the genotypic values of the individuals in a population during the simulation. This last option is the easiest and can give appropriate results if V_i is small compared to V_a.

The theoretical expectation used to estimate V_a with dominance is provided by the following sum across all L loci:

$$V_{a_{tot}} = \sum_{j}^{L} 2p_j q_j (a_j - (p_j - q_j)d_j)^2,$$

where a_j is the mid-homozygote value and d_j is the heterozygote value at locus j (Falconer and Mackay, 1996). This method works reasonably well in many cases but it should be tested for every specific case by comparing the "pure" $V_{a_{add}}$ (only accounting for additive effects) and the "true" $V_a = V_{a_{tot}}$ (accounting for additive and dominance effects) during a test simulation. The first measure $V_{a_{add}}$ is provided by the stat options of the trait (see section 8.2, Table 8.3). The second, $V_{a_{tot}}$ can be estimated with a regression or animal model based on the genotype values saved in the output file of the `quanti` trait obtained with the `quanti_output` parameter. An animal model could easily be implemented since the individual pedigree (over one generation) is also saved in the genotype files (a more extensive pedigree can be obtained with options `pop_output` in section 4.2). The difference $V_{a_{tot}} - V_g$ would give V_i, the dominance variance. Alternatively, the above theoretical sum could be calculated from the allele frequency file accessible for di-allelic loci (see option `quanti_freq_output`).

quanti_heritability [double] (opt)

A fixed heritability value can be set. This value will be used to then set the environmental variance of the trait(s) during a simulation as a function of V_a and V_i. This is done with the `quanti_modifier` LCE. As above, narrow sense heritability is $h^2 = \frac{V_a}{V_a+V_i+V_e}$. Therefore, when trait genotypes are affected by dominance, both V_a and V_i must be determined in order to set V_e and reach the targeted h^2. Note that broad sense heritability can be targeted instead of narrow-sense h^2 with parameter `quanti_heritability_isBroad` below. More details on how V_a and V_i are computed have been provided above.

quanti_heritability_isBroad [double] (opt)

When present in the ini file, this parameter changes the target from narrow-sense heritability h^2 to the broad-sense heritability H^2. Using broad-sense heritability is easier and faster when additive alleles have dominance effects.

6.3.9 Genetic map parameters

quanti_recombination_rate [double / matrix] (opt)

quanti_genetic_map [matrix] (opt)

quanti_random_genetic_map [matrix] (opt)

The recombination map parameters are managed by the genetic map. See section 6.1 for the details. When '`quanti_recombination_rate 0.5`' is set, all loci are independent from each other (freely recombining) and the genetic map is bypassed, even if other traits use the genetic map in the simulation.

If none of the genetic map parameters are provided, then loci are considered independent with recombination rate equal to 0.5.

6.3.10 Output

quanti_output [bool] or [genotype, snp_id, snp_genotype, plink] (opt)

If present without option, the phenotypes of the whole population are saved in a text file, with one individual per row, but without per locus allelic information. The genotypic trait values (G) are added if the environmental variance is not null and the additive genotypic values (A) are added if dominance is modelled.

The data saved is:

```
pop P1 G1 A1 age sex home ped isMigrant father mother ID
```

with pop the patch identifier, Pi the phenotypic, Gi the genotypic values of each trait (Gi is added only if environmental variance is not null), Ai the additive genotype (only when dominance is modelled), age (0 = offspring, 2 = adults), sex (0 = male, 1 = female), home the natal patch, ped the pedigree class (see p66), isMigrant the number of parents of the individual that are immigrants, father, mother, and ID are unique identifiers assigned to individuals that can be used to reconstruct pedigrees.

If one of the *genotype* options shown above in the synopsis is specified, then the allelic values will be printed in the file as well. Therefore, the file will contain an additional set of columns, with two columns per locus, times the number of traits modeled. The columns are added between the pop and P1 columns, with headers ti.lj.x for the mother-inherited allele at locus j affecting trait i and ti.lj.y for the father-inherited allele. The way genotypic or allelic values are reported depends on the option, as explained below.

genotype With this option, the allelic values at each locus and each individuals are reported similarly for both *continuous* and *diallelic* loci.

snp_id The allele ID at di-allelic loci is reported as '0' for the reference allele and '1' for the derived allele. The table below shows which allele value corresponds to each SNP-ID in the `diallelic` model with allelic values $\{+a, -a\}$ and the `diallelic_asymmetrical` model with allelic values $\{a, b\}$.

model	reference	derived
snp_id encoding	0	1
`diallelic`	$+a$	$-a$
`diallelic_asymmetrical` ($a = 0$)	0	b
`diallelic_asymmetrical` ($a \neq 0$)	$min(a, b)$	$max(a, b)$

snp_genotype Here only the genotype of diallelic loci are saved, encoded as 0, 1, or 2 with value 0 = homozygote for the "reference" allele (00), 1 = heterozygote (01 or 10), and 2 = homozygote for the alternative allele (11). In this encoding, the gametic phase is lost and only half the genotype columns are printed, one for each locus/trait.

plink The genotypes of diallelic loci are save in a PLINK1.9 ".ped" file, with corresponding ".fam" and ".map" files. See https://www.cog-genomics.org/plink/1.9 for details. The `.ped` file stores pedigree information in the first four columns, the sex (1 = male, 2 = female) and phenotypic value of the first trait in the next two columns. The whole set of trait values is saved in the `.fam` file, when more than one trait is modelled. The allele values are recorded in the next 2×`quanti_loci` columns of the `.ped` file, with letter `A` for allele 0 (reference) and `G` for allele 1 (derived, see above). The `.map` files records the chromosome ID, locus ID, map position (cM), and base-pair ID. These values depend on the parameters passed to the genetic map. Locus and base-pair ID are identical. If loci are unlinked (the genetic map is not used), then a map with a single chromosome is created, with 50 Morgans between each locus.

quanti_logtime [integer or matrix]

The value is frequency at which phenotypes should be saved, if a single number as in this example:

```
quanti_output  genotypes
quanti_logtime 1000
quanti_dir     QTL
```

or the specific generations at which the files should be saved if provided as multiple values in an array:

```
quanti_output  genotypes
quanti_logtime {{1,1000,50000}}
quanti_dir     QTL
```

quanti_dir [string]

The file directory (relative to the `root_dir` directory).

quanti_freq_output [bool] (opt)

A specific file writer to record, at multiple generations, the per-locus allele frequency at di-allelic loci (SNPs). The allele frequency at each locus for each trait in each patch is saved per line with a column for each generation recorded.

The recording frequency is set with the parameter below. The allele considered is the first provided in input (i.e., the positive allele or first value when two values are specified). A single file is saved for each replicate, with '`.qfreq`' extension.

quanti_freq_logtime [integer] (opt)

The recording frequency of the allele frequency file in generation time. Note that it only accepts a single value since it creates a single output file with columns corresponding to the generations recorded. An array of generation times would only save one file with the last value of the array utilized as generation frequency.

Example of an output *.qfreq* file:

Selecting the frequency output file and setting the frequency of the records:

```
generations            1000
[...]
quanti_traits          1
quanti_loci            10
quanti_allele_model    diallelic
quanti_allele_value    {{0.5,0.5,0.3,0.3,0.1,0.1,0.1,0.1,0.05,0.05}}
quanti_freq_output
quanti_freq_logtime    100
```

This results in the following structure in the output file:

```
pop  trait locus  allele g100    g200   ... g1000
1    1     1      0.5    0.4438  0.2582 ... 0.6693
1    1     2      0.5    0.5164  0.6716 ... 0.9995
1    1     3      0.3    0.4685  0.367  ... 0.0001
1    1     4      0.3    0.576   0.6717 ... 0.9899
1    1     5      0.1    0.4535  0.4902 ... 0.5691
1    1     6      0.1    0.4264  0.3665 ... 0.1459
1    1     7      0.1    0.5056  0.5719 ... 0.7686
1    1     8      0.1    0.5834  0.5568 ... 0.8549
1    1     9      0.05   0.4695  0.5664 ... 0.3376
1    1     10     0.05   0.5683  0.6164 ... 0.4841
...
```

quanti_ohta_output [bool] (opt)

If present in the init file, the Ohta stats will be computed at the generation(s) specified with the logtime parameter. The file contains, for each pair of loci, the r^2 within each patch, the D_{ST}, D_{IS}, D'_{ST}, and D'_{IS}. Fixed loci are skipped. The pairs are ordered numerically, starting from locus 1 until the last pair.

The file name extension is "*.qohta*".

quanti_ohta_logtime [integer/array] (opt)

The number correspond to the time interval at which the Ohta and LD stats are computed and written to the file. Specific times (generations) can be specified with an array containing the generation(s) at which the stats are computed and saved.

Example of an output *.qohta* file:

The simulation has two patches and one quantitative trait with 100 loci.

```
loc1 loc2 Dst      Dis      Dstp     Disp     r_1      r_2
1    2    0.15005  0.00292  0.00827  0.15003  0.01078  0.02772
1    3    0.19924  0.00394  0.01203  0.17083  0.06200  0.01649
1    4    0.21168  0.01482  0.04841  0.19653  0.27553  0.04879
1    5    0.25783  0.00129  0.04147  0.21209  0.02822  0.00809
[...]
98   100  0.06337  0.00035  0.00125  0.05827  0.00894  6.5219e-09
99   100  0.15080  0.00191  0.00419  0.15094  7.0833e-05  0.03069
```

6.4 Deleterious mutations

name: **delet**
files: `".del"` (input/output)
phenotype: a real value in $[0, 1]$, interpreted as the fitness value of the individual

Deleterious mutations are mutations that reduce the fitness of their carrier. This translates into a lower survival probability of the offspring bearing more mutations when applying viability selection on them (see section 5.7). Deleterious mutations are coded by bi-allelic loci, with value of 0 for the wild-type, healthy form, and 1 for the deleterious form. The strength of the deleterious effect of each mutation (i.e. strength of selection) and its dominance coefficient can be set using two different models: constant over loci, or following a given distribution over loci. The selection and dominance coefficients are set for a given locus and apply to all individuals within the species. The total fitness of an individual depends on the way the mutations interact and two fitness models are available; a multiplicative fitness model (independent action of the different mutations, the default) and an additive fitness model (non-independence among loci).

delet_loci [integer]

Number of deleterious loci per individual. The initial mutation frequency can be set below. By default, the initial genotype is all wild-type.

delet_mutation_rate [decimal]

Deleterious mutation rate (allelic mutation rate), from the wild-type to the deleterious form only. There is no reverse mutation rate for now.

delet_backmutation_rate [decimal] (opt)

Rate of backward mutations. Is 0 by defaults (and thus not processed). A backward, or reverse mutation resets an allele to the wild type. The rate of backward mutation must be smaller than the deleterious mutation rate (often 2-3 order smaller).

delet_mutation_model [1,2] (opt)

There are two different models of mutation.

1 (default) : the location of each new mutation is randomly drawn irrespective of the presence of a mutation at that location.

2 : the location of a new mutation is redrawn each time it appears at a homozygous deleterious locus.

delet_recombination_rate, delet_genetic_map, delet_random_genetic_map

Recombination is handled by the genetic map. All genetic map parameters apply. See section 6.1.

delet_init_freq [decimal] (opt)

Initial allele frequency of the deleterious allele. If the parameter is absent, the initial number of mutations of each individual is null. The initial mutations are randomly placed (number = initial frequency times the number of locus).

delet_effects [matrix] (opt)

An optional parameter to pass the fitness effects of the mutations explicitly. The matrix must be 2×delet_loci with the heterozygote fitness effects hs on the *first* row and the fitness effects s of the mutations (homozygote effects) on the *second* row.

If this parameter is set, the parameters below, necessary to randomly set the fitness effects from a distribution, are ignored.

This parameter is useful to pass the fitness effects directly, for instance from a previous simulation or from a binary file if used together with parameter source_parameter_override (see subsection 4.2.2).

delet_effects_distribution [constant, exponential, gamma, lognormal] (opt)

The mutational effects can either be a constant value across all loci (default option) or follow a distribution as set by this parameter. Possible distributions of effects are the exponential, gamma, and log-normal distributions. The mean effect size and the shape of the distribution are set by the parameter below. The dominance coefficient also follows a distribution and is scaled to the mutational effects using the following relationship: $h_i = exp(-ks_i)/2$, where k is a scaling factor chosen so that the average dominance coefficient of all mutants is equal to $\bar{h}$, i.e. $k = -log(2\bar{h})/\bar{s}$, and $\bar{s}$ is the mean effect size.

constant (default): all loci have same selection and dominance coefficients. This is the default, if not specified.

exponential: mutational effects follow a reverse exponential distribution. The mean of the distribution is taken from parameter delet_effects_mean.

gamma: the gamma distribution takes two extra parameters beside the mean effect. The first is the shape (delet_effects_dist_param1) and the second is the scale (delet_effects_dist_param2) of the distribution. Only the shape is mandatory. The scale can be deduced from the mean and shape parameter values ($mean = scale * shape$).

lognormal: the log-normal distribution is another leptokurtic distribution with two mandatory extra parameters, μ and σ, the mean and standard deviation of the mutational effect's logarithm. These two parameters are specified by delet_effects_dist_param1 and delet_effects_dist_param2, respectively. Note that the distribution is truncated to the right, no value greater than 1 is allowed.

Note: when the fitness effects are set from a distribution as here, the values drawn at random are saved in the .log file of the simulation, in the output directory of the simulation. The parameter used to save those values (and the heterozygote values) is the delet_effects param. This is useful when re-using the same population or configuration. Also see comment above about how to reload mutational effects stored in a binary file.

delet_effects_mean [decimal]

Mean fitness effect of the deleterious mutations, or mean selection coefficient s of the mutations. Is used to parameterize the effect sizes distribution.

delet_effects_dist_param1 [decimal] (opt)

Extra parameter used for the description of the distribution of mutational effects. This is the shape of the gamma distribution or the logarithmic mean effect in case of the log-normal distribution.

delet_effects_dist_param2 [decimal] (opt)

Second extra parameter used for the description of the distribution of mutational effects. This is the scale of the gamma distribution or the logarithmic standard-deviation in case of the log-normal distribution.

delet_dominance_mean [decimal]

Dominance coefficient, alternatively the mean of the distribution of dominance coefficients of the deleterious mutations.

delet_fitness_model [1,2] (opt)

Sets the fitness model used to compute the individual viability from the deleterious genome (the trait phenotype). The default is the multiplicative model (model 1):

1 : Multiplicative model. The individual fitness (or viability) is computed as the product of the fitness at each locus: $W = (1 - s)^{n_1} \times (1 - hs)^{n_2}$ where n_1 is the number of homozygote deleterious loci and n_2, the number of heterozygote loci, s is the selection coefficient and h the dominance coefficient (when identical for all loci).

2 : Additive model. Here, mutations act non-independently on fitness, this may be viewed as an epistatic model. The individual fitness is: $W = 1 - n_1 s - n_2 hs$. Symbols have same meaning as above. W is truncated at 0.

delet_fitness_scaling_factor [integer] (opt)

This parameter's value is used as a scaling factor for the individual's phenotype, i.e. its viability is multiplied by this value.

delet_save_genotype [bool] (opt)

Parameter used to save the population genotypes in a text file with the `".del"` extension. The first line holds the column labels. Each line starts with the population identifier followed by one column per locus plus the age, sex, pedigree class, and patch of origin of each individual. The allelic values are 0 for the wild type allele and 1 for the deleterious allele. For the cases where mutational effects are continuously distributed, the second row holds the selection coefficient (homozygous effect) of each locus, and the third one holds the heterozygous effects of each locus.

delet_genot_dir [string] (opt) This parameter specifies a specific path used to save the genotype output files. Should not end with a slash ('/').

delet_genot_logtime [integer] (opt)

This is the generation periodicity of the genotype files or the generations at which the files should be saved if provided as multiple values in an array. If the number is greater than the total number of generations, no data will be saved.

6.5 Dobzhansky-Muller Incompatibility loci

name: **dmi**
files: ".dmi"
phenotype: a real value in $[0, 1]$, interpreted as the fitness value of the individual

The DMI trait codes for so-called (Bateson-)Dobzhansky-Muller Incompatibilities that occur between pairs of loci when both loci are heterozygotes for diploids or for "heterozygous" pairs for haploids. In the latter case, loci in repulsion usually decrease fitness (i.e. aB or Ab have lower fitness than AB or ab). The trait is di-allelic, with allele 0 representing the wild-type (A, B, C, ...) and 1 the 'mutant' (a, b, c, ...). Many pairs of DMI loci can be modelled and located on a genetic map. Incompatible loci are contiguous on the genetic map, but can be set on different chromosomes.

The fitness effects of each incompatible pair must be set explicitly using a matrix argument (see dmi_genot_table below). The fitness effects are either 0 or negative, although they can be positive as well. Fitness effects inform about how fitness is affected relative to the most fit genotype with fitness 1. We usually assume that double homozygotes have fitness 1, and thus receive fitness effect $s_{AABB} = s_{aabb} = 0$. Other genotypes then usually receive negative s values.

The fitness model used is multiplicative across pairs of incompatible loci:

$$W = \prod w(pair_i),$$

where $w(pair_i) = 1 + s_{i,j}$ is the fitness value of genotype j at the locus pair i (remember that $s_{i,j} < 0$ for incompatible genotypes).

A specific initializer has also been added to set patch-specific initial frequencies (see dmi_init).

dmi_loci [integer]

Number of incompatible loci. The number of incompatible pairs is deduced from the number of loci by dividing it by two. The trait is diploid by default (see below). Incompatibilities come by pair, and pairs of locus are contiguous on the chromosome(s). Recombination is set with the genetic map parameters (see below).

Loci are di-allelic, they carry two alleles (0 and 1, or A/a and B/b for each pair). The fitness effect of each allele, and each allele combination is set with parameter `dmi_genot_table` below.

dmi_is_haploid [bool] (opt)

Can be used to change the ploidy of the trait from diploid to haploid. It is set to 'false' by default. The trait will be haploid if this parameter is add, or set to true 1 ('true').

dmi_mutation_rate [decimal]

Per-allele mutation rate. Mutations are both-way ($0 \rightarrow 1$ & $1 \rightarrow 0$).

dmi_recombination_rate, dmi_genetic_map, dmi_random_genetic_map (opt)

Recombination is handled by the genetic map. All genetic map parameters apply. See section 6.1.

dmi_genot_table [matrix]

This table sets the fitness of the genotypes at pairs of incompatible loci. The values provided in the table are the $s_{i,j}$ coefficients in the above formula and are thus relative to the wild type fitness (= 1). A positive $s_{i,j}$ value would cause positive selection on genotype j, while negative $s_{i,j}$ cause negative selection on that genotype.

The structure of the genotype table is: one row per incompatible pair and one column per genotype. Haploid or diploid genotype tables can be provided. If a single row (an array) is provided, the same fitness values are recycled across all pairs of loci.

In the haploid case, the fitness of all 4 genotypes must be given. There are two incompatible genotypes per pair, aB and Ab. The fitness associated with each genotype is written in the following order (for one pair):

{{AB, aB, Ab, ab}}

For the diploids, a minimum of 9 genotypic values must be given. With 9 genotypic values, we do not distinguish between single-locus heterozygotes (i.e. $Aa == aA$ and $Bb == bB$). The incompatible pair is the middle one, AaBb (element number five in the genotype array). The table below shows the ordering of the genotypes in the array:

	BB	*Bb*	*bb*
AA	1	2	3
Aa	4	**5**	6
aa	7	8	9

The fitness of each genotype is provided in the order shown in the table, but on one line/row:

{{AABB, AABb, AAbb, AaBB, **AaBb**, Aabb, aaBB, aaBb, aabb}}.

The values should be given relative to maximum fitness (= 1). Wild-type genotypes should thus get value 0 and incompatible genotypes should get negative values.

For diploid loci, you can also provide the full genotype set, with 16 values. This time, the heterozygotes may have different fitness values depending on the origin of the recessive allele:

	BB	*Bb*	*bB*	*bb*
AA	1	*2*	*3*	4
Aa	*5*	**6**	**7**	*8*
aA	*9*	**10**	**11**	*12*
aa	13	*14*	*15*	16

The genotypes' fitness are provided in a single array as before following the order in the table above. There are now four double-heterozygote genotypes (**bold** faces in the table) and eight single-heterozygotes (*slanted*). The first allele of each genotype comes from the father and the second from the mother.

Example for 5 pairs of incompatible loci where the four incompatible double-heterozygote genotypes 'AaBb' cause a 50% fitness reduction per pair. The minimum individual fitness then becomes $0.5^5 = 0.03125$.

```
dmi_loci               10     #5 pairs
dmi_mutation_rate      1e-5
dmi_recombination_rate 0.5    #unlinked loci
dmi_genot_table        {{0,0,0,0,-0.5,0,0,0,0}} #same for all pairs
```

Of course, the genotype table can be more complex than that, with some loci showing partial incompatibility for single heterozygotes (second pair below), or deleterious effect of the mutant co-dominant allele (third pair), or even recessive deleterious effects in addition to incompatibility:

```
dmi_genot_table  {{0,0,0,0,-0.5,0,0,0,0}
                  {0,0,0,0,-0.5,-0.02,0,-0.02,0}
                  {0,-0.03,-0.06,-0.03,-0.06,-0.09,-0.06,-0.09,-0.12}
                  {0,-0.001,-0.05,-0.001,-0.5,-0.05,-0.05,-0.05,-0.1}
                  {etc.}
                 }
```

dmi_save_genotype [bool] (opt)

Used to tell Nemo to write the genotypes to file. It writes each locus' genotype on two digits, with '0' for wild type allele and '1' as the mutant. Column "W" holds the multilocus fitness value of the individuals.

dmi_logtime [integer] (opt)

Tells every what generation the genotypes should be saved to a text file.

dmi_output_dir [string] (opt)

Tells where (relative to root_dir) to save the genotype files.

STATS

adlt./off.dmi Records the average frequency of allele '1' (the mutant) and the average frequency of incompatibility over all loci/pairs. The output (in the stat files) include patch-specific averages and overall means for both quantities (adlt./off.dmi.freq, adlt./off.dmi.p#)

The incompatible genotype is AaBb in the diploid case (or 01 01 as in the output genotype file) and Ab or aB in the haploid case. The frequency of these genotypes is recorded in the output stat file as adlt./off.dmi.icmp for the overall average, or adlt./off.icmp.p# for the per-deme frequencies.

6.6 Dispersal genes

name: **fdisp, mdisp**
files: NA
phenotype: a real value in $[0, 1]$

If the following parameters are added to the init file, two quantitative traits will be added to the individuals. One codes for the female dispersal rate and is expressed in females. The second codes for the male dispersal rate and is expressed in males only. Both traits are continuous quantitative traits coded by a single diploid locus whose allele values are real numbers ranging from 0 to 1. The two loci are co-inherited. The dispersal probability of an individual (i.e. the trait's phenotype) is the mean of the two allele values at the corresponding locus.

disp_mutation_rate [decimal]

Mutation rate of the dispersal alleles. This is the probability to change the allele value by an amount drawn from an inverse-exponential distribution with the mean set below.

disp_mutation_mean [integer]

This parameter is the mean of the exponential distribution used to draw the mutation step added to the genotype value.

disp_init_rate_fem [decimal] (opt)

Initial genotype (both alleles) of the female dispersal locus.

disp_init_rate_mal [decimal] (opt)

Initial genotype (both alleles) of the male dispersal locus.

disp_init_rate [decimal] (opt)

Initial genotype of both the male and female dispersal locus.

6.7 Wolbachia

name: **wolb**
files: NA
phenotype: a boolean representing the infection status of the individual

The Wolbachia trait is used to simulate the dynamics of an endosymbiotic parasite causing cytoplasmic incompatibility. Its transmission is vertical, through females only and is not perfect, the zygote may loose its parasite ("mutation" process represented by the transmission rate presented below). Zygotes issued from the mating between an infected male and an uninfected female must pay the cost of incompatibility that decreased their chance of survival at birth by a given amount (parameter incompatibility cost of the `breed_wolbachia` LCE). Being infected by *Wolbachia* also induces a cost that translates into a reduced fecundity of the infected females (parameter fecundity cost of the `breed_wolbachia` LCE). See the `breed_wolbachia` LCE for details on the breeding and infection parameters.

wolbachia_transmission_rate [decimal]

This is the rate of transmission of *Wolbachia* from a mother to its offspring. If different from one, the parasite may be lost during gamete formation.

Chapter 7

Examples

7.1 Life cycles

7.1.1 A basic life cycle

To start with, lets exemplify what a basic life cycle looks like:

```
breed 1
disperse 2
aging 3
```

It starts with the reproduction of the population (`breed`), thus adding offspring individuals to it. Then the offspring migrate within the population (`disperse`) before getting older and replacing the previous adult generation that will die because of `aging` (non overlapping generations). The new adult generation is also regulated to not exceed the patches carrying capacities.

Writing this life cycle in a different order would produce exactly the same result, given the sequence of LCEs is conserved (see the following examples). The only change is the population state at the beginning and the end of the cycle.

```
aging 1                   disperse 1
breed 2                   aging 2
disperse 3                breed 3
```

Writing the life cycle as above does not ensure that these LCEs will all be loaded into the life cycle as some of them define additional mandatory parameters that must be present in the init file as well. The `breed` and `disperse` LCEs define such mandatory parameters. The following example will allow to completely build the life cycle.

```
breed 1
disperse 2
aging 3

mating_system 3        # monogamy
mean_fecundity 3
mating_proportion 0.8  # 20% of extra-pair matings

dispersal_model 2   # Island Model with propagule pool migration
dispersal_propagule_prob 0.3 # 30% of propagule dispersers
dispersal_rate 0.125
```

7.1.2 Adding outputs

The previous basic life cycle misses two important features. It does not record statistics and does either not write any output files. To do so, you have to add the following LCEs, `save_stats` and `save_files`.

```
breed 1
save_stats 2
save_files 3
disperse 4
aging 5
```

This way, both the adults and offspring statistics are computed and the various files declared by the simulation components are saved to disc. Which age classes are present in the population at the time of statistics recording and file writing will determine the content of output files (especially the stats output files), the ranks of these LCEs are thus important in that perspective. A third output LCE could have been added here, it is the `store` LCE. Its rank in the life cycle will also determine the age-class content of the binary files.

7.2 Traits

To add a trait to a simulation, it is sufficient to add the mandatory parameters of that trait to the init file. Here is an example with three of the traits currently implemented in Nemo.

```
## NEUTRAL MARKERS ##
ntrl_loci 20
ntrl_all 20
ntrl_mutation_rate 0.0001
ntrl_mutation_model 1    # SSM model

## GENETIC LOAD ##
delet_loci 1000
delet_mutation_rate 0.0001
delet_effects_mean 0.05
delet_dominance_mean 0.36
delet_fitness_model 1    # multiplicative model

## DISPERSAL GENES ##
disp_mutation_rate 0.001
disp_mutation_mean 0.2
```

Each individuals in the simulation will thus carry four sets of genes. One coding for neutral markers with 20 loci, one with 1000 loci carrying deleterious mutations and two coding for female and male dispersal. The genotypes can be saved in binary files using the `store` LCE or by adding the trait-specific output parameters and the `save_files` LCE somewhere in the life cycle.

7.3 A complete example

The next example shows a complete init files with all the mandatory parameters and all the trait output parameters.

```
## SIMULATION ##
filename example
logfile logfile.log
root_dir test
random_seed 988889
run_mode overwrite

replicates 10
generations 1000

## POPULATION ##
patch_number 50
```

```
patch_capacity 20

## LIFE CYCLE ##
breed_selection 1
save_stats 2
save_files 3
disperse_evoldisp 4
aging 5
store 6
extinction 7

# breed and selection parameters #
selection_trait delet
selection_model direct
mating_system 3        #monogamy
mean_fecundity 15      #high enough to resist inbreeding depression
mating_proportion 0.8  #20% of extra-pair mating

# extinction parameter #
extinction_rate 0.05

# disperse parameters #
dispersal_model 2
dispersal_propagule_prob 0.3
dispersal_rate 0.125

# save_stats parameters #
stat off.fstat off.delet viability disp demography extrate
stat_log_time 10
stat_dir stat

# store parameters #
store_dir binary
store_generation 1000
store_noarchive

## NEUTRAL MARKERS ##
ntrl_loci 20
ntrl_all 256
ntrl_mutation_rate 0.0001
ntrl_mutation_model 1
# ouput #
ntrl_save_genotype
```

```
ntrl_output_dir ntrl
ntrl_output_logtime 1000

## GENETIC LOAD ##
delet_loci 100
delet_init_freq 0
delet_mutation_rate 0.0001
delet_effects_distribution exponential
delet_effects_mean 0.05
delet_dominance_mean 0.36
delet_fitness_model 1
# ouput #
delet_save_genotype
delet_genot_dir delet
delet_genot_logtime 1000

## DISPERSAL GENES ##
disp_mutation_rate 0.001
disp_mutation_mean 0.2
dispersal_cost 0.2
```

This example will produce the following files (with # representing the replicate number from 01 to 10).

```
logfile.log
test/example.log
test/stat/example_bygen.txt
test/stat/example.txt
test/ntrl/example_#.dat
test/delet/example_#.del
test/binary/example_#.bin.bz2
```

More elaborate examples can be found in the `example/` folder of the installation package.

Chapter 8

Output Statistics

The summary statistics computed during the course of a simulation depends on the options given to the stat parameter of the save_stats LCE (see section 5.15). The options available are declared by the various simulation components, the traits and the life cycle events. The complete list of these options are given below for each component.

A typical stat option string as found in the init file builds like this:

```
stat fstat off.delet viability disp demography
```

which will result in the computation of the F-statistics for the offspring and adults, the statistics for deleterious mutations on the offspring age class, the mean viabilities, the mean dispersal rates and additional statistics describing the population state. All these options are described below in section 8.2. Note that if one of the component stat option is present in the stat parameter argument but the component itself is missing, this will end the initialisation process of the simulation and abort the program. An example is given here, assuming the dispersal trait is missing but the "disp" stat option is given:

```
***ERROR*** the string "disp" is not a valid stat option
***ERROR*** could not run the sim !
```

8.1 Stat Output Files

The save_stats LCE declares two output files, the ".txt" and "_bygen.txt" files. The first filetype contains the stat records of each recorded generation (set with the stat_log_time parameter) for each replicate. By default, the first and last generations are automatically recorded. This file may be huge depending on the number of stats you are monitoring! It adds two columns, the `replicate` and the `generation` columns, containing the replicate number and the generation number, respectively. The `"_bygen.txt"` file only contains the `generation` column as each line contains the stats averages taken over all replicates. One extra stat is added (alive.repl); it counts the number of extant replicates at each generation.

The replicate stats are dumped to the `".txt"` file at the end of each replicate, whereas the stat average values are saved to the `"_bygen.txt"` file at the end of a simulation.

8.2 Stat Options

The following tables present the different summary statistics of the simulation components that can be monitored during a simulation run.

Output names beginning with `off` are computed on the offspring age class while those starting with `adlt` are computed on the adults. When a stat is described as being the mean of a particular value, this stat is the average of the patch means of the value.

Some stat options may take a prefix tag specifying on which age class they are computed. The naming convention is as follows. A stat argument specified as `[adlt./off.] name` has three possible forms, `adlt.name`, `off.name`, or `name`, meaning the statistics can be restricted to one of the two age classes or computed for both. Alternatively, a stat option described as `adlt./off.name` has only two forms, `adlt.name`, or `off.name`. Likewise, a stat option without any age-class prefix does not accept any such option and likely apply to all age classes, unless specified otherwise.

Table comment:
Stat option: the argument of the stat parameter in the input file.
Output name: the name of the stats as written in the output files.

8.3 Population

Table 8.1: Population stat options

Stat option	Output name	Description
`off.demography`	`off.nbr`	total number of offspring in the metapopulation
	`off.nbfem`	mean number of female offspring per extant patch
	`off.nbmal`	mean number of male offspring per extant patch
	`off.density`	average offspring density
	`off.dvar`	variance of the offspring density of extant patches
`adlt.demography`	`adlt.nbr`	total number of adults in the metapopulation
	`adlt.nbfem`	mean number of females per extant patch
	`adlt.nbmal`	mean number of males per extant patch
	`adlt.density`	average adult density
	`adlt.dvar`	variance of the adult density of extant patches
`demography`		the above demographic stats for offspring and adults
`extrate`	`extrate`	proportion of extinct patches in the population
`fecundity`	`adlt.femfec`	mean assigned females fecundity
	`adlt.femrealfec`	mean effective females fecundity, discounting offspring that do not survive, different from the previous one only when viability selection occurs with breeding
	`adlt.femvarfec`	mean variance in effective fecundity of females
	`adlt.malrealfec`	mean effective males fecundity
	`adlt.malvarfec`	mean variance in effective fecundity of males
`kinship`	`off.fsib`	mean proportion of full-sib
	`off.phsib`	mean proportion of paternal half-sib
	`off.mhsib`	mean proportion of maternal half-sib

Table 8.1 continued on next page

Stat option	Output name	Description
	`off.nsib`	mean proportion of non-sib
	`off.self`	mean proportion of selfed offspring
`pedigree`	`ped.outb`	mean proportion of offspring born from an outbred mating between (unrelated) parents born in different patches
	`ped.outw`	mean proportion of offspring born from an outbred mating between parents born in the same patch but unrelated (both parents' parents are different)
	`ped.hsib`	mean proportion of offspring born from parents with at least one identical parent (half-sib parents)
	`ped.fsib`	mean proportion of offspring born from an inbred mating between full-sib (brother-sister) individuals
	`ped.self`	mean proportion of offspring born from the mating of selfed parents
`migrants`	`emigrants`	mean number of emigrants per patch
	`immigrants`	mean number of immigrants per patch
	`residents`	mean number of residents per patch
	`immigrate`	effective immigration rate computed as $(\frac{immigrants}{immigrants+residents})$
	`colonisers`	mean number of immigrants per extinct patch
	`colonrate`	effective colonisation rate of extinct patches
`migrants.patch`	`emigr.p`i	number of emigrants from patch i
	`resid.p`i	number of residents in patch i
	`imrate.p`i	effective immigration rate into patch i computed as $(\frac{immigrants}{immigrants+residents})$
	`colo.p`i	number of colonizers of patch i; is -1 if patch wasn't extinct. A value of 0 means the patch was extinct but not recolonized.
`pop`		same as "demography", off/adlt.sexratio, and "extrate" together
`pop.patch`	`off./adlt.fem.p`i	number of females in patch i

Table 8.1 continued on next page

Stat option	Output name	Description
	`off./adlt.mal.p`*i*	number of males in patch *i*
	`age.patch`*i*	time since last extinction of patch *i*
	`patch.avrg.age`	mean time (generation) since last extinction of a patch
	`extrate`	proportion of extinct patches in the population
`off/adlt.fem.patch`	`off./adlt.fem.p`*i*	number of females in patch *i*
`off/adlt.mal.patch`	`off./adlt.mal.p`*i*	number of males in patch *i*
`adlt.sexratio`	`adlt.sexratio`	see above
`off.sexratio`	`off.sexratio`	offspring sex ratio

Table 8.1 population stat options

8.4 Neutral markers

Table 8.2: Neutral markers stat options.

Stat option	Output name	Description
Note: More details about the stats are given in section 6.2.		
`[adlt./off.] coa`	*age*`.theta`	mean within deme coancestry
	age`.alpha`	mean between demes coancestry
`adlt.coa.persex`	`adlt.thetaFF`	mean within deme, within females coancestry
	`adlt.thetaMM`	mean within deme, within males coancestry
	`adlt.thetaFM`	mean within deme, between sexes coancestry
`adlt./off. coa.within`	`adlt./off.theta`	as above
`adlt./off. coa.between`	`adlt./off.alpha`	as above
`[adlt./off.] coa.matrix`	*age*`.theta`	as above
	age`.alpha`	as above
	age`.coa`*i*`.`*i*	deme specific mean coancestry within deme *i*, for all demes.
	age`.coa`*i*`.`*j*	deme specific mean coancestry between demes *i* and *j*, for all pairwise comparisons.

`Stat option`	`Output name`	Description
`[adlt./off.] coa.matrix.within`	*age*`.theta`	as above
	age`.coa`*i*`.`*i*	deme specific mean coancestry within deme i, for all demes.
`sibcoa`	`prop.fsib`	mean proportion of full-sib
	`prop.phsib`	mean proportion of paternal half-sib
	`prop.mhsib`	mean proportion of maternal half-sib
	`prop.nsib`	mean proportion of non-sib
	`coa.fsib`	mean coancestry within full-sib
	`coa.phsib`	mean coancestry within paternal half-sib
	`coa.mhsib`	mean coancestry within maternal half-sibs
	`coa.nsib`	mean coancestry within non-sib
`[adlt./off.] ntrl.freq`	*age*`.ntrl.l`*i*`.a`*j*	frequency of allele j at locus i in the whole population
	age`.ntrl.l`*i*`.Het`	mean heterozygosity of locus i in each patch
`[adlt./off.] fstat`	*age*`.allnb`	mean number of alleles per locus in the whole population
	age`.allnbp`	mean number of alleles per locus within demes
	age`.fixloc`	mean number of fixed loci in the whole population
	age`.fixlocp`	mean within demes number of fixed loci
	age`.ho`	observed heterozygosity
	age`.hsnei`	expected demic heterozygosity (Nei & Chesser 1983)
	age`.htnei`	expected total heterozygosity
	age`.fis`	F_{IS} (Nei & Chesser 1983)
	age`.fst`	F_{ST} (G_{ST}; Nei & Chesser 1983)
	age`.fit`	F_{IT} (Nei & Chesser 1983)
`[adlt./off.] weighted.fst`	*age*`.fst.WH`	the Weir&Hill (2002) F_{ST} estimate
`[adlt./off.] weighted.fst.matrix`	*age*`.fst.WH`	the Weir&Hill (2002) F_{ST} estimate
	age`.fst`*i*`.`*i*	deme specific F_{ST} within deme i, for all demes.
	age`.fst`*i*`.`*j*	deme specific F_{ST} between demes i and j, for all pairwise comparisons.

Stat option	Output name	Description
`[adlt./off.] weighted.fst.within`	*age*`.fst.WH`	the Weir&Hill (2002) F_{ST} estimate
	age`.fst`*i*`.`*i*	deme specific F_{ST} within deme i, for all demes.
`[adlt./off.]fstWC`	*age*`.fis.WC`	the Weir&Cockerham (1984) F_{IS} estimate (f)
	age`.fst.WC`	the Weir&Cockerham (1984) F_{ST} estimate (θ)
	age`.fit.WC`	the Weir&Cockerham (1984) F_{IT} estimate (F)
`[adlt./off.] mean.NeiDistance`	*age*`.D`	mean b/n demes Nei's genetic distance (D).
`[adlt./off.] NeiDistance`	*age*`.D`*i*`.`*j*	pairwise Nei's genetic distance b/n demes i and j, for all pairs.
`[adlt./off.] Dxy`	*age*`.Dxy`	average pairwise sequence divergence between all pairs of patches
`[adlt./off.] Dxy.patch`	*age*`.Dxy.p`*i*`p`*j*	average pairwise sequence divergence between patch i and patch j

8.5 Quantitative traits

Table 8.3: Quantitative traits stat options

Stat option	Output name	Description
`[adlt./off.] quanti`	*age*`.q`*i*	mean phenotypic value of the trait in the whole population (equal to the average breeding value in case no environmental variance is set)
	age`.q`*i*`.Va`	average of the within patch additive genetic variance (Va) of the trait
	age`.q`*i*`.Vb`	among patch genetic variance (Vb) of the trait (variance of the patch means)
	age`.q`*i*`.Vp`	average of the within patch phenotypic variance (Vp) (present only if the environmental variance is different from zero)

Stat option	Output name	Description
	age.q*i*.Qst	index of population genetic differenciation for the quantitative trait, calculated from Va and Vb as $Q_{\mathrm{ST}} = \frac{V_b}{V_b+2V_a}$
	age.q*ij*.cov	average genetic covariance within patch between trait i and trait j, present only if more than 2 traits are modelled
[adlt./off.] quanti.eigen	*age*.q.eval*i*	eigenvalues of the D-matrix, the covariance matrix of population means
	age.q.evect*ij*	loadings of the i-th eigenvector of the D-matrix
[adlt./off.] quanti.eigenvalues	*age*.q.eval*i*	eigenvalues of the D-matrix, the covariance matrix of population means
[adlt./off.] quanti.eigenvect1	*age*.q.evect1*i*	loadings of the first eigenvector of the D-matrix
[adlt./off.] quanti.mean.patch	*age*.q*i*.p*j*	mean phenotypic value of trait i in patch j
[adlt./off.] quanti.var.patch	*age*.Va.q*i*.p*j*	additive genetic variance of trait i in patch j
	age.Vp.q*i*.p*j*	phenotypic variance of trait i in patch j (only if the environmental variance is not zero)
[adlt./off.] quanti.covar.patch	*age*.cov.q*ij*.p*k*	genetic covariance between trait i and j in patch k
[adlt./off.] quanti.eigen.patch	*age*.qeval*i*.p*j*	eigenvalues of the G-matrix in patch j (genetic covariance matrix)
	age.qevect*ij*.p*k*	loadings of trait j on eigenvector i of the G-matrix in patch k
[adlt./off.] quanti.eigenvalues.patch	*age*.qeval*i*.p*j*	eigenvalues of the G-matrix patch j
[adlt./off.] quanti.eigenvect1.patch	*age*.qevect1*i*.p*j*	loadings of trait i on the first eigenvector of the G-matrix in patch j
[adlt./off.] quanti.skew.patch	*age*.Sk.q*i*.p*j*	skew of the phenotypic distribution of trait i in patch j
[adlt./off.] quanti.patch		adds the stats from quanti.mean.patch, quanti.var.patch, quanti.covar.patch, and quanti.eigen.patch

8.6 Deleterious mutations

Table 8.4: Deleterious mutations stat options

`Stat option`	`Output name`	Description
`[adlt./off.] delet`	*age*`.delfreq`	mean deleterious mutation frequency
	age`.delhmz`	mean deleterious mutation homozygosity
	age`.delhtz`	mean deleterious mutation heterozygosity
	age`.delfix`	mean number of fixed mutation in the whole population
	age`.delfixp`	mean demic number of fixed mutation
	age`.delsegr`	mean number of segregating mutation in the whole population
	age`.delsegrp`	mean demic number of segregating mutation
	age`.delfst`	Fst of the deleterious mutations
	age`.lethequ`	mean number of lethal equivalents
	age`.heterosis`	heterosis computed as: $H = 1 - \frac{b_g}{b_p}$ b_g : the effective fecundity of within deme matings (mating partners are from the same patch) b_p : the effective fecundity of between deme matings (mating partners are from different patches)
	age`.load`	mean demic mutational load computed as: $L = 1 - \frac{\bar{W}}{W_{max}}$ where W_{max} is the maximum number of surviving offspring produced by a female in a patch
		Note: `heterosis` and `load` are computed from the female fecundities which are updated according to the offspring survival in the `breed_selection` LCE only, and are thus null when viability selection is performed differently. In that case, they can be inferred from the fitness stats.

Stat option	Output name	Description
`[adlt./off.] viability`	*age*.viab	mean patch viability (= mean trait value)
	age.viab.outb	mean viability of outbred individuals between demes
	age.viab.outw	mean viability of outbred individuals within demes
	age.viab.hsibs	mean viability of inbred individuals between half-sib parents
	age.viab.fsibs	mean viability of inbred individuals between full-sib parents
	age.viab.self	mean viability of inbred individuals descended from selfed parent
	age.prop.outb	proportion of between demes outcrosses
	age.prop.outw	proportion of within demes outcrosses
	age.prop.hsibs	proportion of within demes half-sib matings
	age.prop.fsibs	proportion of within demes full-sib matings
	age.prop.self	proportion of within demes selfed matings
`meanviab`	off.viab	see above
	adlt.viab	same for adults
`survival`		now part of the selection LCE's stats.

8.7 Dobzhansky-Muller Incompatibilities (DMI)

Table 8.5: DMI stat options

Stat option	Output name	Description
`[adlt./off.] dmi`	*age*. dmi.freq	overall average frequency of mutant alleles, across loci
	age. dmi.p*i*	patch-specific frequency of mutant alleles, across loci
	age. dmi.icmp	overall average frequency of the incompatible genotype(s) (AaBb for diploids, Ab and aB for haploids)

Stat option	Output name	Description
	age. `dmi.icmp.p`*i*	patch-specific frequency of the incompatible genotype(s), across loci

8.8 Selection

Table 8.6: Selection stat options

Stat option	Output name	Description
`fitness`	*age.* `fitness.mean`	mean of the within patch **offspring** fitness *before* viability selection, i.e., including all offspring
	`fitness.outb`	fitness of b/n demes outbred offspring
	`fitness.outw`	fitness of w/n demes outbred offspring
	`fitness.hsib`	fitness of half-sib crosses
	`fitness.fsib`	fitness of full-sib crosses
	`fitness.self`	fitness of selfed crosses
`fitness.prop`	`prop.outb`	proportion of b/n demes outbred **offspring**
	`prop.outw`	proportion of w/n demes outbreds
	`prop.hsib`	proportion of half-sib crossings
	`prop.fsib`	proportion of full-sib crossings
	`prop.self`	proportion of selfed progeny
`survival`	`survival.outb`	mean proportion of surviving **offspring** *after* viability selection, for each pedigree class
	`survival.outw`	
	`survival.hsib`	
	`survival.fsib`	
	`survival.self`	
`[off./adlt.] fitness.patch`	*age.* `W.avg.p`*i*	mean offspring/adult fitness of patch i
`[off./adlt.] fitness.var.patch`	*age.* `W.var.p`*i*	mean offspring/adult variance in fitness of patch i

8.9 Dispersal

Table 8.7: Dispersal stat options

`Stat option`	`Output name`	Description
`[adlt./off.]disp`	*`age`*`.disp`	mean dispersal rate
	`age``.fdisp`	mean female dispersal rate
	`age``.mdisp`	mean male dispersal rate
`[adlt./off.]` `[fem./mal.]` `disp.patch`	*`age`*`.`*`sex`*`.disp.p`i	age and sex specific mean dispersal rates per patch

8.10 Wolbachia

Table 8.8: Wolbachia stat options

`Stat option`	`Output name`	Description
`wolbachia`	`off.fwoinf`	mean infection frequency of offspring females
	`off.mwoinf`	mean infection frequency of offspring males
	`off.incmating`	mean number of incompatible matings
	`adlt.fwoinf`	mean demic infection in extant demes for adult females
	`adlt.mwoinf`	mean infection frequency of adults males in the whole population
	`wolb.infvar`	inter-demic variance in adult female infection
	`wolb.extrate`	proportion of demes having lost infection in adult females
`wolbachia` `_perpatch`	`off.p`i`fwoinf`	mean infection frequency of offspring females in patch i
	`off.p`i`mwoinf`	mean infection frequency of offspring males in patch i

Bibliography

Carter, A. J. R., Hermisson, J., and Hansen, T. F. (2005). The role of epistatic gene interactions in the response to selection and the evolution of evolvability. *Theor. Popul. Biol.*, 68(3):179–196.

Chebib, J. and Guillaume, F. (2017). What affects the predictability of evolutionary constraints using a G-matrix? The relative effects of modular pleiotropy and mutational correlation. *Evolution*, 71(10):2298–2312.

Falconer, D. S. and Mackay, T. F. (1996). *Introduction to Quantitative Genetics*. Longman, London, fourth edition.

Goudet, J. (1995). Fstat (version 1.2): A computer program to calculate f- statistics. *J. Hered.*, 86:485–486.

Guillaume, F. (2011). Migration-induced phenotypic divergence: the migration-selection balance of correlated traits. *Evolution*, 65(6):1723–1738.

Guillaume, F. and Perrin, N. (2006). Joint evolution of dispersal and inbreeding load. *Genetics*, 173(1):497–509.

Guillaume, F. and Perrin, N. (2009). Inbreeding load, bet hedging, and the evolution of sex-biased dispersal. *Am. Nat.*, 173(4):536–541.

Hansen, T. F. and Wagner, G. P. (2001). Modeling genetic architecture: A multilinear theory of gene interaction. *Theor. Popul. Biol.*, 59(1):61–86.

Kimura, M. (1965). Attainment of quasi linkage equilibrium when gene frequencies are changing by natural selection. *Genetics*, 52(5):875–890.

Matthey-Doret, R. (2021). SimBit: A high performance, flexible and easy-to-use population genetic simulator. *Molecular Ecology Resources*, 21(5):1745–1754. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/1755-0998.13372.

Nei, M. and Chesser, R. K. (1983). Estimation of fixation indices and gene diversity. *Ann. Hum. Genet.*, 47:253–259.

Weir, B. S. and Cockerham, C. C. (1984). Estimating F-statistics for the analysis of population structure. *Evolution*, 38:1358–1370.

Weir, B. S. and Hill, W. G. (2002). Estimating F-statistics. *Annu. Rev. Genet.*, 36:721?750.