N E M O – A G E

version 0.32

User Manual

March 10, 2023

authors
Frédéric Guillaume
Olivier Cotto
Max Schmid

contributors
Jacques Rougemont (MPI version)
see more in Nemo manual

availability
https://bitbucket.org/ecoevo/nemo-age-release

Contents

Chapter 1

Introduction

Nemo-age is a forward-time, individual-based, genetically and spatially explicit, stochastic simulation program designed to study the evolution of genetic diversity, life histories, and phenotypic traits in a flexible stage-structured population framework. Nemo-age implements a recombination map on which loci coding for different types of trait can be placed together. The evolving traits currently provided are universally deleterious mutations, quantitative traits, and neutral markers (e.g., SNP, microsatellites). The numbers of populations, individuals per population or loci per trait in a simulation are only limited by hardware capacities. Nemo-age 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-age's framework is coded in C++ and has been designed to be easily extended and include new evolving traits or population features.

1.1 Availability

Nemo-age comes free of charges and is distributed under the GNU General Public License (GPL3+). Binaries and source code are provided for the Linux, MacOSX, and Windows platforms. Nemo-age 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). The git public repository is available at: https://bitbucket.org/ecoevo/nemo-age-release.

1.2 Installing Nemo-age

Nemo-age is distributed as code source with compiling instructions and as executable binaries for the Max OS and Windows operating systems. Compiling and

installing instructions are provided in the INSTALL file present in the distribution archive. To build NEMO-AGE from source, the Gnu Scientific Library (GSL) is necessary. The GSL can be easily installed on most OSes and is available at http://www.gnu.org/software/gsl/. The parallel MPI version requires SPRNG: the Scalable Parallel Random Number Generators library version 5.0 or higher available at: http://sprng.cs.fsu.edu/. The Windows version is built and run in the CygWin environment (https://cygwin.com). This manual, as well as software packages containing compiled (binary) versions of NEMO-AGE are available at: https://bitbucket.org/ecoevo/nemo-age-release/downloads. The following packages, for version `x.y.z` are available:

- `NemoAge-x.y.z-MaxOSX.tgz` (for MaxOSX, see subsubsection 1.4.1.1)
- `NemoAge-x.y.z-Win64.tgz` (Windows/CygWin, see subsubsection 1.4.1.2)

Please refer to the `README` and `INSTALL` files of the distribution for detailed instructions about compilation and installation.

1.3 Extending Nemo-age

NEMO-AGE is designed as a flexible and extensible coding framework. Its API is written in C++. 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.

1.4 Using Nemo-age

NEMO-AGE is a command line tool without a graphic user interface. The basic user interface is a text file (a.k.a., the 'init file') containing the input parameters and their argument in a `key value` format. NEMO-AGE is then launched from the terminal with that init file as an argument. Some runtime information (current running simulation, current generation/replicate, etc.) is written to the standard output (terminal window). NEMO-AGE 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 2 for input specification and chapter 3 for parameters description.

1.4.1 Running Nemo-age from the command line

1.4.1.1 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, whatever your operating system is, we assume you have installed the executable file `nemoage` 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 (note: we will use the symbole `>` for the command prompt, i.e., the symbol starting an input line on your terminal):

```
> ls
nemoage Nemoage.ini
```

So, we have the executable file, `nemoage` and a configuration file, `Nemoage.ini`. Now, if we type the following command, NEMO-AGE will automatically search for the `Nemoage.ini` file in the local directory and try to initiate a simulation from it.

```
> ./nemoage
```

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 (access it with `echo $PATH` on the command line). This command will produce the following output to your terminal window (or something approaching depending on the program's version):

```
> ./nemoage

N E M O A G E 0.28 [13 Apr 2020]

Copyright (C) 2020
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://bitbucket.org/ecoevo/nemo-age-release
------------------------------------------------
reading parameters from "Nemoage.ini"
setting random seed from input value: 486532
```

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

start: 12-09-2018 16:29:56
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 [16:30:43] 100/100

end: 12-09-2018 16:30:48
--- done (CPU time: 00:00:51s)
setting random seed from input value: 486532

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

start: 12-09-2018 16:30:48
[...]
```

This terminal output shows the components of the simulation and the progress of the simulation with the replicate and generation counters. NEMO-AGE 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.

1.4.1.2 For Windows users

On Windows, NEMO-AGE can be built and ran using CygWin. CygWin is an emulation of a full Linux environment on Windows. It is the only option that we currently recommend to run simulations with the Windows operating system. And it works fine! You can find the instructions to compile NEMO-AGE using CygWin in the INSTALL file in the software package. You can also directly use the compiled version of NEMO-AGE provided in the `NemoAge-x.y.z-Win64.tgz` package, making sure you follow the instructions in the INSTALL file to install CygWin. You can install CygWin from `http://www.cygwin.com`.

Once you have installed CygWin on your machine, navigate to your home directory within the `C:/cygwin/home/` folder (adapt the path according to your installation). Copy the NEMO-AGE binary `nemoagex.y.z.exe` there, in your home directory (or

below it). You will find `nemoagex.y.z.exe` in the folder `NemoAge-x.y.z/Win/binaries` once you have expended the `NemoAge-x.y.z-Win64.tgz` package, downloaded from the repository. It is simpler to start by using your cygwin home directory because it is your default working directory when you start the cygwin console. Once you are all set, you can start using NEMO-AGE in cygwin following the same instructions as for any Linux/Unix environment (see above and the rest of this manual).

1.4.2 Batch mode

NEMO-AGE 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-AGE, the program will initiate three simulations from those files. NEMO-AGE may also run multiple simulations from a single init file if that file contains parameters with multiple values. Such parameters are then called *sequential* parameters (see section 2.5) and NEMO-AGE 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-AGE. In any case, NEMO-AGE 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-AGE with more than one argument:

```
> ./nemoage 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 string does not matter. Their filename parameter are set to sim1, sim2, and sim3 respectively. This command will produce the following output:

```
N E M O A G E 0.28 [13 Apr 2020]
[...]
-----------------------------------------------
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 the following parameter with several arguments:

```
patch_capacity 5 10 20
```

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

```
> ./nemoage 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 ]

[...]
```

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

```
> nemoage example/Nemoage.ini example/fstat_input_and_cross.ini \
example/binary_input_and_cross.ini
```

1.4.3 Parallel version (MPI)

NEMO-AGE also comes with an MPI layer fully integrated into the code. NEMO-AGE can thus be built to run on multiple CPUs using the message passing interface (MPI). See the INSTALL file in the distribution 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.

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 'worker' CPU processing 3 replicates.

1.5 What are the main features of Nemo-age?

NEMO-AGE is built to simulate eco-evolutionary dynamics in large populations of a single species. To do so, NEMO-AGE 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 by affecting the individuals within it. The population dynamics are then

function of the genetic composition and distribution of trait values in the population and of the action of the life cycle events. Examples of life cycle events are reproduction (with mutation and recombination), migration, survival, age and stage transition, population density regulation, among the most important ones.

The fate of an individual during the execution of a life-cycle event may depend on the value of the traits and the genes it carries. To this end, NEMO-AGE offers the possibility to simulate the dynamics of different genetic elements. Genes carried by an individual can be placed on a genetic map consisting of multiple chromosomes, or linkage groups, with variable recombination rates. The main genetic elements proposed are neutral loci, quantitative trait loci (with pleiotropy), and deleterious mutations. The selection acting on those genetic elements can be precisely parameterized. Not all genetic types (called traits) need to be modeled in a given simulation. The assemblage of the genetic traits and life cycle events depends entirely on the user's needs. NEMO-AGE also offers the possibility to input and output genetic and phenotypic data in various formats, depending on the traits.

In the following sections, we will present the main components and the general philosophy and architecture of the program. The simulation components are assembled into a simulation depending on the parameters given in input to the program. The main and only interface of NEMO-AGE is a text file holding the component parameters. Chapter 4 presents the parameters of the life cycle events (LCEs) and chapter 5, the parameters of the traits. All these components are optional and can be assembled at the user's wish. The parameters for the simulation and the population components are, however, mandatory. We will start with them.

1.6 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 of the simulation. Several run modes are possible to help with testing and running simulations in different environments (see section 3.1 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 2.5).

1.7 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.

NEMO-AGE provides a highly versatile spatial population model. The spatial model is defined by the dispersal model. It ranges from the classical island model of migration, with equal dispersal probabilities among them, to a spatially explicit population model with sex-specific and patch-specific dispersal rates, provided in input by the user (see disperse). Patches can be added, removed, fused or split during a simulation as well. This way, complex spatial scenarios can be modeled, like range shrinkage and expansions (e.g., Gilbert et al., 2007; Cotto et al., 2017), and more.

NEMO-AGE also provides a highly flexible population density regulation model. By default, the population size within each patch is dynamical. The populations may vary in size during a simulation as the result of the stochastic events of birth, dispersal, selection, death, etc. Therefore, *demographic stochasticity* is a built-in property of NEMO-AGE. The built-in demographic stochasticity of NEMO-AGE allows for the simulation of complex *eco-evolutionary dynamics* where the population size within each patch depends on both demographic (growth, competition, mortality, migration) and adaptive (survival) processes. The population dynamics set by extinction-recolonization and evolutionary processes in fluctuating environmental conditions can be studied in details (Cotto et al., 2017, 2020).

1.8 The Individual

An individual in NEMO-AGE is defined as a trait container. Which traits are added to an individual depends on the parameters present in the input file. By default, individuals do not carry any trait (and thus do not carry any genetic information). The only pre-defined individual characteristics are age and sex. The default model therefore simulates population dynamics only.

When traits are added (e.g., quantitative traits), individuals have a set of traits that can be neutral or under selection. The *fitness* of an individual depends on its trait(s) under selection. Individuals additionally store information about their ancestry (dad's and mum's IDs) and demographics (natal patch, fertility) 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). These information tags are used to compute

pedigree-based or age-/sex-specific statistics.

1.9 The Life Cycle Events

We give here the list of the major events during a life cycle and the corresponding LCEs. Because each LCE is an independent simulation component, its parameters need to be added to the init file to be operational during a simulation. The set of LCE of a simulation defines the life cycle followed by each individual and iterated over the number of years set by the user. There is no default life cycle, the set of LCEs and their ordering must be set explicitly by the user in the init file. Note that LCEs are also necessary to compute and save output statistics. The LCE's parameters are described in chapter 4.

1.9.1 Mating and reproduction

The following LCEs handle mating and reproduction, sometimes jointly with other events in one LCE:

`breed`, `breed_selection`, `breed_disperse`, `breed_selection_disperse`, `cloning`

The **mating systems** implemented are:

- **random mating** (promiscuity)
- **polygyny** (where number of mating males can vary)
- **monogamy** (couples potentially re-set each year)
- **selfing** (fusion of self-gametes, set as a rate)
- **cloning** (self-copy without meiosis, suppresses recombination, set as a rate)
- **hermaphroditism** (random mating in monoecious organisms, capable of selfing)

The outcome of reproduction is the addition of a certain number of offspring individuals to the patches depending on the average fecundity of the females. It does not depend on selection unless reproduction is coupled with selection as with `breed_selection` or `breed_selection_disperse`. The average fecundity is a mandatory input parameter used to set the mean of the Poisson distribution from which the actual number of offspring a female does is drawn. Other distributions can be used. The sex of the offspring is set randomly by default. Fixed fecundity and sex-ratio can also be modeled.

The `cloning` LCE is a separate reproduction event that can add clones (of adults) to any other stage within a patch, in addition to sexual reproduction if `breed` is also part of the life cycle. This is useful to model the vegetative growth of plants or parthenogenesis.

1.9.2 Dispersal

Forward (zygotic) and backward (gametic) migration can be modeled with the following dispersal LCEs:

`disperse` (forward), `breed_disperse` (backward), `breed_selection_disperse` (backward)

- with *forward* migration, the dispersal probabilities are interpreted as probabilities to move from a focal patch of origin i to a patch of arrival j. `disperse` acts on a set of stages and moves individuals among patches according to the rates specified in a dispersal matrix. It thus changes local population size.
- with *backward* migration, the interpretation is reversed: individual movement is from patch of origin j into focal patch of arrival i. Backward rates are used when population sizes are kept constant as in the Wright-Fisher model. They are used to determine the parents of each new offspring. The parents themselves do not move, only their gametes do. Therefore, backward migration is always used together with mating and reproduction. It is implemented in `breed_disperse`, and `breed_selection_disperse`.

Pre-defined dispersal models are available to all dispersal LCEs:

- Island Model with migrant- or propagule-pool migration
- Stepping Stone Model (nearest-neighbour migration on a string of patches)
- 2D lattice model on a grid with n rows and m columns, set as a torus or with reflective or absorbing borders, depending on user's choice.

Otherwise, any dispersal pattern can be specified with:

- the stages that disperse,
- sex-specific dispersal matrices,
- separate seed and pollen dispersal matrices (for monoecious organisms), see `seed_disperse` and `breed_disperse`,

– reduced dispersal and connectivity matrices to simplify large and sparse migration matrices in simulations of large geographical grids, e.g., with $10^3 - 10^5$ patches or more.

As a random demographic process, *forward* migration will contribute to demographic stochasticity at the meta-population level, while *backward* migration will not.

1.9.3 Selection and fitness

Natural selection is modeled with the following LCEs:

`viability_selection`, `breed_selection`, `breed_selection_disperse`

Individual fitness depends on trait values. Therefore, a selection LCE needs to be linked to at least one trait carried by the individuals. Selection can also act on multiple traits.

As in the present version, NEMO-AGE implements two types of genetically-determined traits that can be under selection: the `deleterious mutation` trait and the `quantitative` trait. The trait value determined by *deleterious mutations* is directly interpreted as the fitness value of the individual, pending different standardizations, and is a function of the distribution of fitness effect (DFE) chosen or specified by the user in input. The *quantitative* trait values result in a uni- or multi-variate phenotype whose fitness value depends on a (potentially multivariate) patch-specific "environmental" value called the *local optimum* of the traits. The selection model for quantitative traits can be either the *Gaussian* or *quadratic* fitness models, which define a fitness peak at the optimum trait values. A parametric model is also implemented to model fitness as the sole function of the inbreeding level of the individual.

Survival *versus* fecundity selection – The consequence of a lower fitness is a decrease in *survival probability* or in females *fecundity*, either because of an increase in the number of deleterious mutations carried or in the mismatch between the individual phenotype and the local optimum.

The `viability_selection` LCE, and the two composite LCEs, remove from a patch all individuals who failed at a viability test. The fitness of the individual is its survival probability and the test is performed by randomly drawing a number uniformly distributed between 0 and 1. The individual survives if its fitness is above that random deviate. Viability selection acts on offspring by default (stage 0) but can also be set to act on any other stage in a structured population. Moreover, the strength of selection (for quantitative traits) can be set differently at each stage under selection.

Selection can also act on *fecundity* instead of survival when the right option in `breed_selection` is used. In the `breed_selection` LCE, the fecundity of a female is weighted by her fitness value. The population size after reproduction thus depends

on the average fitness of females. Selection on males is done via selection on their daughters.

In the `breed_selection_disperse` LCE, with backward migration, patch sizes are constant, and exactly K surviving offspring are produced. Therefore, survival is checked before adding an offspring to a patch. It results that the lower the mean fitness of the new generation, the higher the number of offspring that need to be produced to fill a patch. Different parameters allow for the control of this process.

Fitness standardization – The fitness value of an individual may depend on the fitness values of other individuals in its patch, or in the whole population. If so, fitness is said *relative* because standardized either to the *mean* or to the *maximum* fitness value of the patch, calculated on the individuals under selection. Standardization to the mean or max of the whole population (metapopulation) is also possible. Otherwise, the raw fitness values are taken directly from the trait phenotypes, in which case fitness is *absolute*. The default fitness model in Nemo-age is the *absolute* fitness model. Relative fitness values should be handled with care because their effect on selection depends on the standardization. When relative to the mean, the mean patch fitness is one. Therefore, some individuals will have fitness values above one, which will not translate into higher survival than the average (with a maximum probability of one). Mean-relative fitness thus only makes sense in the case of selection acting on fecundity rather than survival. Max-relative fitness, on the other hand, can be used to model viability selection.

Hard *versus* soft selection – The demographic and genetic consequences of selection depend on the fitness standardization and regulation modes. Selection is *hard* if there is variation in mean fitness among patches. Hard selection results from patches with higher mean fitness generating more offspring and growing faster than patches with lower mean fitness. To obtain hard selection, fitness must be absolute and population regulation must keep variation in local population sizes. *Soft* selection can be obtained by setting a regulation mode (see below) that equalizes population size in all patches such that there is no variation in patch growth. Note that a switch between hard and soft selection can occur in Nemo-Age when the mean fitness or the mean fecundity is too low to allow production of enough offspring to fill the patches to their carrying capacities and fitness is absolute.

1.9.4 Density regulation

Patch density regulation is performed with the following LCEs:

`regulation`, `aging`, `breed_disperse`, `breed_selection_disperse`

Density regulation is necessary to avoid infinite growth of the population, or to

keep the patches and total population at a constant size. There are two types of regulation in NEMO-AGE. First, there is K-regulation, regulation to the carrying capacity of a patch, also called ceiling regulation. This is achieved by randomly removing individuals above the carrying capacity of a patch until it reaches K. Ceiling regulation is implemented in `regulation` and `aging` (but not in `aging_multi`). This type of regulation does not completely eliminate demographic stochasticity because it allows patches to be bellow their carrying capacity, depending on their mean effective fecundity. Another implementation of ceiling regulation is in so-called Wright-Fisher populations, as provided by `breed_disperse` and `breed_selection_disperse`. In Wright-Fisher populations, exactly K viable offspring are produced in each patch, thus avoiding to produce supernumerary individuals. The `regulation` LCE is thus not necessary in conjunction with `breed_disperse` and `breed_selection_disperse`. Wright-Fisher populations eliminate demographic stochasticity and require less computing time because they generate less individuals.

The second regulation mode is density-dependent regulation implemented in the `regulation` LCE. Density regulation reduces the number of individuals at a given stage/age within a patch according to the density of that patch (e.g., number of adults) and to a competition coefficient which translates density into reduced survival. Density dependence is often referred to as intra-specific competition for resources. The form of the relationship between density and competition is implemented in two models, the Beverton-Holt (Beverton and Holt, 1957) and Ricker (Ricker, 1954) functions (see details in `regulation`). The properties of these functions have been extensively developed in theoretical population ecology.

1.9.5 Age and stage transition

Transition of individuals between age classes and stages is provided by:

`aging` and `aging_multi`

Whether an individual stays or transitions within or between stages depends on a number of life history events (e.g. survival, growth, metamorphosis) that occur with some probabilities. These probabilities, together with the stage-specific fecundities define the transition matrix of the species and in turn the dynamics of the population.

The `aging_multi` LCE takes care of the process of within-stage survival and between-stage transitions. The transitions between stages depend on the age of the individuals in regard to the age definition of each stage. The age definition of a stage corresponds to the age from which an individual can reach a stage and the number of years it can stays (upon survival) within this stage (set with parameter `pop_age_structure`, see subsection 3.2.1). For instance, we can model the life cycle of bi-annual plants with three stages: offspring (seed), sub-adult (rosette), and adult (flowering plant). The age of transition between the stages is a function of the

biology of the species and can be 1 year for the seed-rosette transition and 2 years for the rosette-flowering plant transition. These "years" are counted as a number of iterations of the life cycle spent by each individual after birth (called a `generation` in NEMO). The two `aging` LCEs update the age of the individuals in the population each time they are executed, that is at each iteration. The `aging` LCEs are usually set at the end of the list of LCEs within a life cycle. See subsection 3.2.1 for more details about the stage survival and transition processes in stage-structured populations.

In simple two-stage populations with non-overlapping generations, transition between offspring individuals (non-reproducing) and adults (reproducing) followed by density regulation (ceiling regulation only) is provided by the `aging` LCE. This LCE removes all adults (mortality = 1) and replaces them with K randomly chosen offspring. The same result can be obtained with `aging_multi` followed by `regulation`.

No density regulation is performed with `aging_multi`. The `regulation` LCE is thus needed to implement density dependent regulation in stage-structured populations. The growth rate of a structured population can be calculated from the transition matrix (the matrix holding the vital rates and fecundity of the species) following matrix population model theory (Caswell, 2001). Adjusting the competition coefficient of the density regulation function in `regulation` defines implicitly a carrying capacity (see Examples).

1.10 Traits and their genetics

Traits implement the genetics part of a simulation. Traits are independent definitions of specific genotype-phenotype mappings. All traits share the same genetic map. NEMO-AGE proposes a set of traits with three different genotype-phenotype mappings:

- neutral markers (from SNPs to microsatellites) with no phenotype,
- universally deleterious mutations (di-allelelic) directly affecting fitness,
- pleiotropic quantitative trait loci (QTL) coding for multiple (correlated) quantitative traits.

The advantages of this approach are many. First, which trait is incorporated into a simulation depends on a user's choice. Second, each trait can have its own data structure to optimize execution of the simulations. Neutral markers are coded on bytes (1 locus = 1 byte), deleterious mutations are coded on bits (1 locus = 1 bit = 1/8 of a byte), and QTL are coded as double-precision numbers (1 locus = 4 bytes). The number of possible alleles at a neutral locus is thus 256, it is 2 for deleterious mutations, and virtually infinite for QTL (but can be limited to two).

Third, despite this difference in data structure, all genetic elements can be simulated simultaneously on the same genetic map. This feature (introduced in NEMO v2.3.0) allows for the stacking of mutations of any type on the genetic map.

The genetic map is a recombination map (i.e., not a physical map) in that it specifies the locus positions in units of recombination, the centimorgan [cM] or a fraction of it. The trait-specific locus positions can be set randomly on chromosomes or linkage groups, or precisely depending on known or unknown recombination rates. For instance, simulations with one QTL per linkage group surrounded by thousands of equidistant neutral SNPs can be easily implemented. The number of loci and the recombination rates between them will set the map length of each linkage group (or chromosome). Furthermore, with mutation stacking, a given map position can thus harbor multiple alleles of different types. Stacking of loci on the same map position means those loci will be completely physically linked, although in reality they might have different base location in the genome. The degree of stacking thus depends on the scale of the recombination map used to specify the positions of the loci. Stacking allows for the study of the effects of linked selection on patterns of neutral genetic diversity, to give just one example.

To understand the capabilities and computational performances of NEMO and NEMOAGE when simulating genetic data, a word about the implementation is necessary. NEMO uses memory-aligned, fixed-length arrays to store allelic values of each trait in each diploid individual. This differs from other forward-time simulators which store mutations in time-ordered arrays (or linked lists), stacking mutations as they appear (e.g., SLiM3). In NEMO, the length of a sequence array is set by the number of loci modeled, a parameter set at the start of a simulation by the user. The locus map positions must also be known at the start of the simulation. This approach has the advantage of providing fast access and fast copy of genetic data during computation, especially during recombination, because there is no need to sort mutations according to their map position, nor to look for mutations in a (time-ordered) list of mutations. The disadvantage, however, is that individuals will carry redundant information if they differ little in their genetic composition. Typically, fixed loci are not removed from the data arrays. Therefore, simulations of a very large number of elements ($> 10,000$) with very low mutation rates, and thus very low polymorphism, will have a large memory footprint and be slower than models recording only polymorphic sites in time-ordered arrays. On the other hand, for the same number of polymorphic sites (e.g., at high mutation rate), the fixed-length array structure of NEMO is several order faster than variable-length array structures (see e.g., Matthey-Doret, 2020).

1.11 Statistics and outputs

NEMO-AGE provides several ways of recording 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 7.2):

- **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.
- **Population stats**: patch density, 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. For this to work, the `save_stats` and `save_files` LCEs must be present in the life cycle.

1.12 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 4.10). Binary files contain the whole population information, including all

the trait (genetic) and individual data and the simulation parameters. Binary files can then be used by NEMO-AGE to load a saved population and run a new simulation from it (see subsection 3.2.3). This is useful as a backup strategy for long simulations and to store populations after a burn-in phase.

Chapter 2

The input parameter 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 to several argument values separated by at least one white space character (spaces or tabs). A particular parameter must appear only once in the init file, this is the only restriction for now. The order of appearance of the parameters in the file does not matter.

2.1 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 interpreted an integer value (non-decimal). Any decimal part of a number will be ignored.
- **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 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 2.4 for details and examples.

2.2 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-characters 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** : {{a_{11}, a_{12}, ..., a_{1n}}{*row 2*}...} : any argument value starting and ending by two enclosing curly braces is considered as a matrix argument (see next section). It includes values passed as **arrays** (single-row matrix).
- **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 2.5).
- **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 2.6).
- **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 2.7). 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 2.7). Argument values are separated by commas within a block argument (e.g., `(@g0 0.02, @g5 0.5)`).

2.3 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 macros detailed below.

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 2.5).

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.

2.3.1 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}}
```

2.3.2 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}}
```

2.3.3 c(): concatenate

`c(..., sep=",")` : This macro joins a lists of numbers 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)}}
```

2.3.4 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("{",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}}
```

2.3.5 tempseq(): temporal sequence

`tempseq(at, seq)` : Creates a sequence of temporal argument values within parentheses following the syntax for temporal arguments shown in section 2.7. 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 parameters. 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)
```

2.3.6 matrix(): create matrix

`matrix(x, nrow, ncol) :` Generates a $nrow \times ncol$ matrix from a quoted list of numbers passed as first argument x. The number of elements in x must be equal to $nrow \times ncol$ 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}}
```

2.3.7 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}}
```

2.3.8 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}}
```

2.3.9 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 2.1 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.

Table 2.1 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 = $(\exp(\sigma^2) - 1)\exp(2\mu + \sigma^2)$

2.4 Matrix arguments

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 specify several matrices on separate 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 .

2.5 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 generated. This section explains how to specify specific simulation output filenames

based on the sequential parameter values. This mechanism also works throughout the whole set of string parameter arguments (e.g. the output directory or input binary file arguments).

Basic filename/argument string expansion: If your configuration file comprises sequential parameters, you may add the special expansion character `%` followed by a number (`%1` for e.g.) 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 3.1). 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 use more expansion characters than the number of sequential parameters but if you use less or none at all, a number will be added to the simulation filename to prevent overwriting the same file(s) several times (does not apply to other string arguments). 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 sequence.

ex: if we have these two sequential parameters:

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

Setting the base filename this way:

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

will give the following basenames, one for each simulation:

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

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

If the `filename` parameter is specified without expansion character:

`filename mysim`

the simulation basenames will be:

`mysim-1`, `mysim-2`, `mysim-3`, and `mysim-4`

Advanced filename expansion: 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 for instance the sequential argument is a matrix, or is too long to fit in, we also want to have a way to get a specific filename that we can refer to more explicitly than by a number. This is done by adding a **format string** within the expansion string. That string helps setting the format of the argument value (number of digits to use) or provides an alternative set of argument value identifiers as a character string.

The **format string** is enclosed with two single quotes '`' '`' and is composed first of an optional dot '`.`' followed by a mandatory integer number, and finally followed by an optional character string enclosed with two square brackets '`[]`'. The optional dot and character strings are mutually exclusive. Here is an example of each possible option: '`'4'`', '`'.3'`', or '`'2[AaAbAcBaBbBc]'`'. The format string is placed in-between the expansion character and the sequential parameter number, like this: '`%'4'1`', '`%'.3'2`', or '`%'2[AaAbAcBaBbBc]'3`' supposing we have three sequential parameters in an input file.

The mandatory integer value of the format string is the width of the argument name string. For instance, '`%'4'1`' means that the values of sequential parameter no.1 will be written on 4 characters with leading zeros. A value of `10` for that parameter will thus be added to the filename string as `0010`. The dot preceding the width specifier simply indicates that only the decimal part of the argument value must be taken with trailing zeros. In the example above, a value `0.1` for sequential parameter no.2 will be added as `100` to the filename string. Finally, a set of character strings can be specified as in the last example above. These characters will be used sequentially as replacement values for the actual parameter values found in the input file. The width specifier tells how much characters must be read within the format string and added to the filename. For instance, for value no.4 of sequential parameter no.3 above, the string `Ba` will be added to the filename string.

A last option is to replace the character string by a + to replace the argument value by its position value: '`%'1[+]'3`'. As here the third sequential parameter is supposed to have 6 argument values, the + stands for the integer values 1 to 6 and the width specifier is 1 (no leading 0).

Here is the full example:

```
filename a%'4'1_b%'.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):

```
a0001_b001_Aa
a0001_b001_Ab
...
a1500_b100_Bc
```

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.

2.6 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, where *filename* is a character string that contains the path to the external file relative to the directory from which NEMO-AGE 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.

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 way as it would be written in the init file (i.e., without new lines between multiple arguments).

Example:

```
param_a 1 2 3
param_b &filename1.txt &filename2.txt &filename3.txt
param_c &path-%1/to/filename-%'1[abc]'2.txt
```

Here, param_b and param_c have argument values stored in external files. The filename and the directory path to param_c depend on the argument value of param_a and param_b (i.e., `path-1/to/filename-a.txt; path-1/to/filename-b.txt;` etc.)

2.7 Temporal arguments

NEMO-AGE 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.

2.8 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 3.1) in the init file and any expansion strings are substituted with their corresponding parameter value. Several extension strings are then added to that base name.

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 (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-AGE 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 3.1) 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 4.8). 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 4.10 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-AGE notifies the different simulation components at time of saving, you *must* include the `save_files` life cycle event (see section 4.9) in the life cycle, otherwise no file will be written during a simulation. See chapter 4 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 and some info about the simulation (value of the seed of the random generation, elapsed time and CPU time used).

Chapter 3

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. The life cycle and trait components are all optional. Note that you can also use NEMO-AGE 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 the stored genotypes and write them to different kinds of text files.

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)** and are used to add extra features needed to build a particular model. NEMO-AGE 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 2 for details about the different types of parameters.

There are two main types of simulation components; the Traits (chapter 5) and the Life Cycle Events (chapter 4). 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. 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 7.

3.1 Simulation

Parameters:

replicates [integer]

Number of replicates to perform per simulation.

generations [integer]

Number of generations performed per replicate.

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 2.8).

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 chapter 2.

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 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-AGE 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-AGE to build multiple simulations out of them, as it would for the other parameters (see section 2.5). 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-AGE. 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 21:)
> sh my_script.sh mySim-10loci-10pop 21 arg1 arg2 arg3
```

3.2 Population

Parameters:

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 argument is 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-AGE to ignore the later and use only the first one to build the population.

patch_init_stage_size [matrix] (opt)

Optional parameter to specify the number of individuals created at the first generation of a replicate to fill the patches. Each row is for a patch, and each element in a row is for a stage. There must be as many elements per row as the number of stages specified in `pop_age_structure`. The user can pass one array (1-row matrix) with the number of individuals in each stage, in which case the values will be used for all patches. The values are *not* sex-specific. If this parameter is not specified, the initial patch sizes are taken from the patch size parameters above, and applied to *all* stages (> 0) equally.

```
patch_number 10
patch_nbfem  5000  #carrying capacities used for density regulation
patch_nbmal  0
patch_init_stage_size {{0, 100}}  #will build each patch with 100
                                  #females and 100 males in stage 1
```

3.2.1 Multi-stage populations

patch_capacity, patch_nbfem, patch_nbmal [integer/matrix] (opt)

These parameters work similarly as in single-stage populations with the difference that input values correspond to the number of adult individuals in each stage *at initialization*. For example, if `patch_nbfem` is set to 100 and `pop_age_structure` is {{0,1,4,7}}, simulations start with 100 females in all stages but the first, i.e., 300 adult females in total.

However, the carrying capacities provided here are interpreted as the *total* carrying capacity of the patches by the `regulation` LCE, when counting individual in *all* stages. These values are thus treated as the maximum total number of individuals in a patch when the patches are regulated to their carrying capacities (see the `regulation` LCE).

To initialize the simulation with a different number of individuals per patch and age class/stage, use the parameter `patch_init_stage_size` above.

pop_age_structure [matrix] (opt)

A one-line (row) matrix. The number of columns sets the number of stages in the life-cycle. The matrix is a list of increasing integers **always starting from 0**. The first column must correspond to the offspring stage (offspring produced by sexual reproduction of adult stages) and **must** be set to 0. Numbers in columns i ($i \geq 1$) set the age (number of years or iterations of the life cycle) at which individuals in stage $i-1$ need to transit to stage i. The value in column i thus sets the maximum age an individual can reach in stage $i-1$. Therefore, the difference between integers in two consecutive stages (columns) gives the maximum time an individual can remain within a stage.

To change the interpretation of the stage-specific transition ages into *minimum* ages before which an individual *cannot* transition to the next stage, the parameter `pop_transition_by_age` is necessary (see description below). This can be useful when individuals must stay in a stage for some years before maturing into a reproductive stage, for instance.

The minimum and default matrix is `{{0,1}}` (offspring and adults), with a corresponding transition matrix (see `pop_transition_matrix` below) with non-overlapping generations and a mean fecundity of 2.7.

IMPORTANT: the first stage (offspring) is always referred to as **stage 0** throughout (also applies to other simulation components). All stages after the first (offspring) are considered adults when it comes to recording statistics, and may be pooled by some stat recorders.

Examples:

```
pop_age_structure {{0, 1, 2, 3}}
```

The population is composed of four stages: offspring (**stage 0**), stage 1, stage 2, and stage 3. An individual can stay only one year within each stage but the last, depending on the corresponding last entry of `pop_transition_matrix`. In this example, stages correspond to age-classes.

```
pop_age_structure {{0, 1, 4, 7}}
```

The population is also composed of four stages but individuals can survive and remain 3 years in stage 1 and 2. If an individual survives three years in one of these stages, i.e., if it reaches age 4 or 7 respectively, it must either transition to the next stage with probability given by `pop_transition_matrix` or die. All individuals must transition to stage 3 (the last stage) at age 7, where they may survive several years depending on the corresponding entry in `pop_transition_matrix`.

pop_transition_matrix [matrix] (opt)

A $s \times s$ square matrix whose size corresponds to the number of stages s (i.e, number of elements in `pop_age_structure`). The matrix reads as columns contributing to rows (see examples below). The matrix defines the species' stage specific fecundity (on first row), within-stage survival (along the diagonal) and between-stage transition (off the diagonal). The stage survival and transitions (based on `pop_transition_matrix` and `pop_age_structure`) are handled by the `aging_multi` LCE.

Structure and interpretation of the transition matrix: As an example, the transition matrix for a three-stage population in NEMO-AGE has the form (can be generalized to more stages):

$$\begin{pmatrix} s_{0,0} & f_1 & f_2 \\ t_{1,0} & s_{1,1} & 0 \\ 0 & t_{2,1} & s_{2,2} \end{pmatrix},$$

with f_j the mean fecundity of stage j, $s_{j,j}$ the *survival* probability within stage j and $t_{i,j}$ the *transition* probability from stage j to stage i, $(i = j + 1)$. In its current version, NEMO-AGE only models transitions to the next, upper stage and sexual reproduction must produce offspring, hence the zeros in the matrix. The total probability to survive in stage i is $s_{j,j} + t_{i,i+1}$, such that the

probability to die in this stage is $1-(s_{j,j}+t_{i+1,i})$. Note that this implies that $s_{j,j}+t_{i,i+1}$ *must* satisfy $0 \leq s_{j,j}+t_{i,i+1} < 1$.

If the probability to survive the last stage is set to zero, the maximum age given in `pop_age_structure` is the lifespan. Otherwise, individuals remain in this last stage with probability $s_{j,j}$ until they die, reaching an age that depends on that probability.

Important notice: For the other stages, the maximum age that can be reached in a given stage as set in `pop_age_structure` (see above) is a hard limit. When it is reached, individuals only have the option to transit with probability $t_{i,j}$, or die with probability $1-t_{i,j}$. However, when survival within that stage is $s_{j,j} \neq 0$, individuals may in theory survive past that age. To avoid the hard age limit very large age limits can be set within `pop_age_structure` to ensure that the maximum age is never reached. For instance, with $s_{1,1} = 0.9$, setting `pop_age_structure {{0, 1, 400}}` would give a probability to reach the age of 400 years in stage 1 close to 0 (i.e., $0.9^{400} \approx 0$). This way, the dynamics will match their deterministic expectations.

Except for the first element, the first row of the matrix gives the mean number of offspring (stage 0) expected from *sexual* reproduction by females from each stage. Sexual reproduction is handled by the `breed` LCE. Clonal reproduction is treated by another LCE (see below). The first element of the first row gives the contribution of the offspring stage to itself, thus corresponding to survival or dormancy, as for e.g., in a seedbank.

Default transition matrix: The default matrix is

$$\begin{pmatrix} 0 & 2.7 \\ 1 & 0 \end{pmatrix}$$

which corresponds to a population with non-overlapping generations and mean fecundity = 2.7, with an age structure of `pop_age_structure {{0,1}}`).

Examples:

```
pop_age_structure     {{0, 1, 2, 3}}
pop_transition_matrix {{0.0,  10,   20,  5}
                       {0.1,  0.0,  0.0, 0}
                       {0.0,  0.3,  0.0, 0}
                       {0.0,  0.0,  0.3, 0}}
```

The above example corresponds to an age-structured model. Offspring (**age 0**) do not reproduce. Individuals of age 1, 2, and 3 produce on average 10, 20, and 5 offspring, respectively. Offspring have a probability 0.1 to survive

to age 1. Individuals of age 1 and 2 have probability 0.3 to survive to age 2 and 3, respectively. Individuals of age 3 die after one cycle (one year).

```
pop_age_structure     {{0, 1, 4, 7}}
pop_transition_matrix {{0.0,  10,   20,  5}
                       {0.1,  0.1,  0.0, 0}
                       {0.0,  0.3,  0.1, 0}
                       {0.0,  0.0,  0.3, 0}}
```

Similar to the previous example except that this time individuals in stages 1 and 2 may remain in their stage depending on their age. If they have reached the age to transition to the next stage as specified in `pop_age_structure` (4 and 7 years, respectively), they either survive and go to the next stage or die. Otherwise, at each year before reaching the transition age, they may survive with probability $\sigma = (0.1 + 0.3)$ and transition to the next stage with probability $\gamma = 0.3/(0.1 + 0.3)$, survive (σ) and stay in the same stage with probability $1 - \gamma$, or die with probability $1 - \sigma = 1 - (0.1 + 0.3)$. Individuals die after 1 year when in stage 3. The maximum lifespan is 7 years.

```
pop_age_structure     {{0, 1, 4, 7}}
pop_transition_matrix {{0.0,  10,   20,  5}
                       {0.1,  0.1,  0.0, 0}
                       {0.0,  0.3,  0.1, 0}
                       {0.0,  0.0,  0.3, 0.8}}
```

Same as the second example above but individuals stay in stage 3 with probability 0.8 until they die. The maximum lifespan is above 7 years.

Matching simulations in Nemo-age with a matrix population model

We here emphasize that the parameter `pop_transition_matrix` differs from the matrix population model corresponding to the simulations. We further show how to build a matrix population model that corresponds to a simulation in NEMO-AGE.

A basic succession of LCEs would read like this in the input parameter file:

```
breed       1  # mating and reproduction
aging_multi 2  # stage transition
save_stats  3  # census
```

Consider the case where the input matrix in NEMO-AGE is of the form:

$$L_{nemo} = \begin{pmatrix} 0 & F_1 & F_2 \\ s_0 & 0 & 0 \\ 0 & s_1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 2 & 2 \\ 0.8 & 0 & 0 \\ 0 & 0.5 & 0 \end{pmatrix}$$

where F stands for fecundity and s stands for survival, and the indices correspond to the age-classes. The life cycle is as specified above (`breed` - `aging_multi` - `census`). Starting from a population with 10 individuals in each age class, one iteration of the life cycle in NEMO-AGE will give (numbers are expectations):

$$\begin{bmatrix} 10 \\ 10 \\ 10 \end{bmatrix}_t \xrightarrow{\text{breed}} \begin{bmatrix} 40 \\ 10 \\ 10 \end{bmatrix}_{t'} \xrightarrow{\text{aging\_multi}} \begin{bmatrix} 0 \\ 32 \\ 5 \end{bmatrix}_{t+1}.$$

At census time, the population thus has 32 individuals in age-class 1 and 5 in age-class 2. However, the expected individual numbers in a matrix population model using L_{nemo} as transition matrix are:

$$\begin{bmatrix} 0 & 2 & 2 \\ 0.8 & 0 & 0 \\ 0 & 0.5 & 0 \end{bmatrix} \times \begin{bmatrix} 10 \\ 10 \\ 10 \end{bmatrix}_t = \begin{bmatrix} 40 \\ 8 \\ 5 \end{bmatrix}_{t+1}$$

The difference is caused by a census time after `aging_multi` in NEMO-AGE. Offspring individuals never appear in that census because they transit to the first adult stage within each cycle (each year). A census after reproduction would show the new offspring (the original 10 are discarded when $L_{nemo}[1,1] = 0$) and the reproducing adults *before* transition (at step t' above). There isn't a way to match the theoretical census expected from the corresponding matrix population model unless the transition matrix is modified in NEMO-AGE.

The modified matrix population model that corresponds to NEMO-AGE's life cycle shown above is a 2x2 transition matrix aggregating reproduction and transition of the offspring in a single "stage":

$$L = \begin{pmatrix} F_1 s_0 & F_2 s_0 \\ s_1 & 0 \end{pmatrix} = \begin{pmatrix} 1.6 & 1.6 \\ 0.5 & 0 \end{pmatrix}$$

so that offspring created during LCE `breed` enter age 1 during the subsequent LCE `aging_multi` *within one execution of the life cycle*:

$$\begin{bmatrix} 1.6 & 1.6 \\ 0.5 & 0 \end{bmatrix} \times \begin{bmatrix} 10 \\ 10 \end{bmatrix}_t = \begin{bmatrix} 32 \\ 5 \end{bmatrix}_{t+1}.$$

Therefore, matrix L is the correct matrix for deterministic population model projections corresponding to L_{nemo} used for simulations.

On the other hand, to align census sizes in NEMO-AGE with their mathematical expectations based on L_{nemo} (i.e., $n_{t+1} = [40, 8, 5]$), the transition matrix in NEMO-AGE needs to be extended with a dummy stage

$$L'_{nemo} = \begin{pmatrix} 0 & 0 & 2 & 2 \\ 1 & 0 & 0 & 0 \\ 0 & 0.8 & 0 & 0 \\ 0 & 0 & 0.5 & 0 \end{pmatrix}$$

such that individual numbers after `breed` and `aging_multi` correspond to:

$$\begin{bmatrix} 0 \\ 10 \\ 10 \\ 10 \end{bmatrix}_{t} \xrightarrow{\text{breed}} \begin{bmatrix} 40 \\ 10 \\ 10 \\ 10 \end{bmatrix}_{t'} \xrightarrow{\text{aging\_multi}} \begin{bmatrix} 0 \\ 40 \\ 8 \\ 5 \end{bmatrix}_{t+1} .$$

pop_transition_by_age [bool] (opt)

This option forces the transitions to happen at the maximum age set for each stage in the `pop_age_structure` parameter, and *not before*. It is particularly useful for long-lived species (e.g., trees) when juveniles require decades before transiting to a reproductive stage. For an individual below the maximum age the total probability of surviving in stage j is unchanged and set to $s'_{j,j} = s_{j,j} + t_{j+1,j}$, with $t_{j+1,j}$ and $s_{j,j}$ as set in the transition matrix. For an individual that has reached the maximum age set for stage j in `pop_age_structure`, transition is computed normally, with transition probability $t_{j+1,j}$. If this individual does not transition to the next stage, it dies. The following examples illustrate the alternative behaviors:

```
pop_age_structure     {{0, 1, 4}}
pop_transition_matrix {{0.0,  0.0,  6.0}
                       {1.0,  0.9,  0.0}
                       {0.0,  0.08,  0.0}}
```

Consider an individual with age $x < 4$, by default, when `pop_transition_by_age` is not set, it would survive and stay in stage 1 with probability $s_{1,1} = 0.9$, survive and transition to stage 2 with probability $t_{2,1} = 0.08$ or die with probability 0.02. With `pop_transition_by_age`, it would survive and stay in age 1 with probability $s_{1,1} + t_{2,1} = 0.98$ until age 4 and then transit to stage 2 with probability $t_{j+1,j} = 0.08$ or die.

Compare with the transition matrix below that allows juveniles to take 50 years before reaching the reproductive stage. The probability that a juvenile

reaches age 50 is $0.98^{50} \approx 0.36$. Moreover, individuals can transition to the reproductive stage from age 1 which might not model accurately long lived species.

```
pop_age_structure       {{0, 1, 51}}
pop_transition_by_age
pop_transition_matrix   {{0.0,  0.0,  6.0 }
                         {1.0,  0.0,  0.0}
                         {0.0,  0.98,  0.9}}
```

3.2.2 Saving the population pedigree

With the following parameters, a file can be periodically written to disk 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 (number of cycles), stage, home (natal patch), patch (residence patch at time of sampling). The pedigree information can thus be gathered over multiple 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 3.2.3.

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.

3.2.3 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 4.10) 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 some of the parameters of the source simulation. NEMO-AGE will not be able to load the source population if the number of loci of the traits differ between the source and the current simulation, or if a trait is added/removed in the loading simulation relative to the information found in the source. 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 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 but it is a good idea to have a source population that is at least as big as the receiving one to completely fill the first generation. 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 stages (offspring or adults, or both) used when loading a population depends on the ones present in the source file and the ones required by

the life cycle events of the current simulation. Which stage to load is specified with `source_fill_stage` or determined by finding the first event in the current life cycle that requires a specific stage (generally offspring or adults, see chapter 4 on life cycle events). NEMO-AGE then tries to load the matching stages from the source file.

Using compressed binary files: Finally, when loading populations from binary files, NEMO-AGE will automatically check whether the binary fill is compressed. If so, NEMO-AGE will decompress the file, 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 and .bz2) and the corresponding programs are available on the system.

Parameters:

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 is 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. The path/filename of the source file may contain the expansion character and format strings (i.e., in the form of `%'pattern'#`, see chapter 2) 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_stage [integer] or [adults, offspring, all] (opt)

This sets the stages that must be loaded from the source population. The stages used to load from will be: only stage 0 with `offspring`, all stages but stage 0 with `adults`, and all stages with `all`. If this parameter is not used, NEMO-AGE will determine the minimum necessary stage to load by looking at the LCE age-class requirements (see LCE chapter below). Only the option `all` will allow to completely preserve the age structure of the source population (with `source_preserve`).

The integer passed is interpreted as a set of bit flags where each bit corresponds to a stage. If a bit is set for a stage, then that stage will be used as a load source for the current population. Therefore, stage 0 (offspring) has first (lowest) bit set to 1, which corresponds to integer value 1. By combining the bits, you can chose among the stages. For instance, choosing stages 3 & 4 would result in having bits 3 & 4 set (01100), which corresponds to integer value 12.

source_replicates [integer] (opt)

By specifying this parameter, you can tell NEMO-AGE how many replicates of the source population have to be used 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. If this value is smaller than the current number of replicates, then the source population will be changed every [replicates / source_replicates] replicates. The source filename is built using 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.

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.

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

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

Here, the same population from the file named mysourcepop_001.bin is loaded by each replicate of that simulation.

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.

If the simulation has more replicates to run than the number of source file replicates, a single source can be re-used by multiple replicates. For instance, if 25 replicates were performed for the source (burn-in) simulation while we need to

now run 100 replicates, the source population/file will be changed every four replicates. Thus, with the parameter values below, replicates 1 to 4 will use data from `mysourcepop_01.bin`, replicates 5 to 8 will use `mysourcepop_02.bin`, and so on until file `mysourcepop_25.bin` for replicates 97 to 100. For that case, the source-loading parameters would be:

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

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. Often, age or stage information is not available if the genotype file. It is thus important to specify in which age class the individuals must be loaded. The following example loads neutral markers data (e.g. from a field study) from a single FSTAT file (see section 5.2 for more details) and use it to compute the F-statistics available in NEMO-AGE:

```
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_stage 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 4

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 one iteration of the life cycle (one 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 1 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 by mistake two LCEs have same rank, only one is retained (set by 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 an LCE will makes it a sequential parameter (will initiate multiple *different* simulations).

The way to build the life cycle in the init file is to write the LCE name (given below) followed by the rank number. Here is an example (see **??** for more details):

```
breed 1
save_stats 2
save_files 3
disperse 4
viability_selection 5
aging_multi 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. 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 (stage > 0) would be recorded, for instance. The disperse LCE then moves the offspring around according to the migration model chosen. The individuals can then experience a round of viability selection within their patch 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_multi: stage transition and aging in multi-stage populations

aging: stage transition and patch regulation in 2-stage populations with non-overlapping generations

breed: mate and breed, create new offspring generation

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

breed_selection: a fast breed with selection on fecundity and survival

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

disperse: offspring dispersal

regulation: patch density regulation

save_files: write output files to disk

save_stats: record statistics

selection: viability selection

store: save simulation data to binary files

4.1 Stage transition with `aging_multi` and `aging`

`aging_multi`

Performs the transition of individuals between stages according to the transition probabilities given in `pop_transition_matrix` and the age structure provided in `pop_age_structure` (see subsection 3.2.1). It is this LCE that increments the age and stage of the individuals and decides whether an individual survives or dies according to the transition matrix and (st)age structure. Its position relative to the reproductive LCEs (`breed` and `cloning`) will be important to determine whether individuals reproduce at a given stage or not. In general, we advised to place '`aging_multi`' after '`breed`'. The `aging_multi` LCE does not handle regressive transitions from late stages to earlier stages.

Parameters:

aging_multi [integer]

LCE name. The value is the rank in the life cycle.

aging

This LCE applies only to 2-stage populations without overlapping generations. It is not linked to the stage structure parameters specified in the population component. The `aging` LCE moves *all* offspring individuals from stage 0 to the adult stage 1, removes all adults already present in stage 1, and performs patch regulation at the same time. In 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 4.7) also performs "ceiling" regulation but on each age class separately, thus keeping the offspring and adults in the population.

Parameters:

aging [integer]

LCE name. The value is the rank in the life cycle.

4.2 Mating and reproduction with `breed`

breed

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 or by `pop_transition_matrix`. A female's fecundity may be a fixed number or a number drawn from a random distribution. The default distribution is a Poisson.

Parameters:

breed [integer]

LCE name. The value is the rank in the life cycle.

mating_system [1 to 6]

Six mating systems are implemented in NEMO-AGE. The options are:

1 : promiscuity/random mating. One male is randomly chosen from all males in a patch for each new offspring a female does.

2 : polygyny. One male only mates with all females in the patch. This can be changed in two ways. First, by setting mating_proportion to a value < 1, one male will monopolise a proportion equal to *mating_proportion* of the matings within a patch while the remaining matings will be shared by all other males. Second, the number of mating males may be changed with the mating_males parameter, which sets how many males will share the reproductive output of the pach. Females will then randomly mate with 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 descended from a random male in the population.

4 : selfing/hermaphrodite. Only females are used in that case. If mating_proportion $= 1$, then all offspring are produced by self-fertilisation, otherwise, a proportion of $1 -$ *mating_proportion* of the offspring are produced by randomly crossing two "females" to mate together.

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/hermaphroditism. 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* (watch the patch size parameters!).

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.

mean_fecundity [integer]

Mean of the distribution used to set the females fecundity. It is used whatever the mating system selected.

fecundity_distribution [fixed, poisson, normal] (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".

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 another. The "fixed" option proposed here sets the sex-ratio to exactly 1:1.

4.3 Clonal reproduction

`cloning`

Cloning allows for the production of new individuals by cloning (e.g., vegetative growth). In the current version of NEMO-AGE only females (or hermaphrodite individuals if `mating_system` option is `6`) perform clonal reproduction. Cloning produces females in stage 1 by default (first *non-offspring* class). The individuals produced have exactly the same genotype as their mothers.

Parameters:

cloning [integer]

LCE name. The value is the rank in the life cycle.

cloning_rate [double]

Set the number of individuals produced by a female during clonal reproduction. Note that this parameter actually takes any value > 0.

cloning_add_to_stage [integer] (opt)

Optional parameter used to specify to which stage the clones are added. It must be between 0 (offspring) and (*number of stages* – 1) (where *number of stages* corresponds to the size of the `pop_age_structure` array). By default, cloning adds individuals to stage 1.

cloning_competition_coefficient [double] (opt)

Identical to the Beverton-Holt competition coefficient of the LCE `regulation`, but competition affects the survival of individuals produced by cloning. This parameter is optional with default value = 0 (no competition). Density is computed from individuals in all stages but stage 0.

4.4 Dispersal

`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 4.4.1, ***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 4.4.2, ***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 4.4.2). 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 4.4.3, 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.

By default, dispersal is performed on offspring individuals only (stage 0). The dispersing stage(s) can be changed with the parameter `dispersal_stage`, but dispersal rates and models are the same for all dispersing stages (for now).

Parameters:

disperse [integer]

LCE name. The value is the rank in the life cycle.

dispersal_stage [integer/matrix] (opt)

Optional parameter to change the stage of dispersing individuals, or perform dispersal at multiple stages during the life cycle. The argument is either a single value or an array with minimum one value and maximum the number of stages (specified with `pop_age_structure` parameter). Each value identifies a stage, starting from 0 up to [*number of stages* − 1]. The default stage is 0 (first stage).

4.4.1 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
```

dispersal_rate [decimal] (opt)

This parameter sets both the male and female dispersal rates (identical value for both). NEMO-AGE will build the dispersal matrices according to the dispersal model chosen below.

dispersal_rate_fem / _mal [decimal] (opt)

Replaces the previous parameter for the case of different male and female dispersal capabilities.

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

Parameter to set the dispersal model followed by dispersing individuals. It is omitted when the dispersal matrix (or reduced matrices) is provided instead of a dispersal rate. 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, each offspring in a patch has 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 rectangular 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 a 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

4.4.2 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 matrix parameter is used to specify the dispersal matrix of the model. It must be patch_number x patch_number in dimensions. Each d_{ij} element of this matrix is the dispersal probability from patch in row i to patch in column j. This parameter has precedence over the dispersal rate and model parameters. If too big, and especially when containing a large number of zeros, this matrix can be replaced by the `dispersal_reduced_matrix` and `dispersal_connectivity_matrix` below.

dispersal_matrix_fem / _mal [matrix] (opt)

The dispersal matrices are in fact sex-specific and this parameter can thus be used to specify sex-specific dispersal patterns. Same comment about parameter precedence as above.

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 by dispersal. 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.

It is **critical** that the rates in the `reduced_matrix` are provided in the exact same order as the patch ID in the `connectivity_matrix`.

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}}
```

4.4.3 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.

4.4.4 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.

4.4.5 Seed dispersal

seed_disperse

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 modeled 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_disperse' (e.g. 'dispersal_rate' becomes 'seed_disperse_rate') for *all* dispersal parameters.

Parameters:

seed_disperse [integer]

LCE name. The value is the rank in the life cycle.

4.5 Selection

viability_selection

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) and quant (quantitative traits), although any other

trait may be used as long as the trait's phenotype is compatible with the fitness models implemented. Selection can act on multiple traits simultaneously. That is, the fitness of an individual is given by the multiplication of the fitness values provided by each trait under selection. See section 4.5.1 below. 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.

This LCE also declares a set of fitness statistics that can be recorded during the simulation (see section 7.2), and individual fitness values can be saved to a file, for each individual in each patch of the whole population (see parameters `selection_output_` below).

The parameters described here are the same as those used with the breed_selection and breed_selection_disperse composite LCEs (which inherit those parameters).

Multi-stage selection: Selection can differ among stages, or act on a single stage only. By default, viability selection acts only on juveniles, in stage 0. The stages on which selection acts are set with parameter `selection_at_stage` below. More options to model stage-specific selection are provided by mixing viability selection with fecundity selection. For this, see `breed_selection` in subsection 4.11.1.

Parameters:

viability_selection [integer]

LCE name. The 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 traits are the only traits under viability selection (i.e., their trait value is used to set the individual fitness). Multi-trait selection can be initiated by providing trait names enclosed in parentheses, see subsection 4.5.1.

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.

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_at_stage [array] (opt)

Optional parameter to set the stage(s) at which viability selection acts. The default stage is the offspring stage 0. This parameter may change or add stages under selection. The argument must be an array of the index of each stage under selection (one-line matrix). The indices are within the range $[0, number\ of\ stages - 1]$. Multi-stage selection applies to all selection models. Note that in the case of the Gaussian model for quantitative traits, the stage-specific selection parameters must be passed with `selection_matrix`, as explained in subsection 4.5.4 below.

Example:

```
selection_at_stage {{0,1}}
```

Here, selection affects survival at stage 0 and 1 (note that transitions between two stages is conditional on survival through the first stage, which also depends on transition parameters set in `pop_transition_matrix`).

4.5.1 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 3   #quant trait can be multi-dimensional
```

```
selection_variance 4
selection_local_optima {{5}}
```

4.5.2 Fixed selection model parameters

We specify here the parameters necessary for the **fix** option to parameter `selection_model`, as described above. As a reminder, the fitness of an individual depends on its inbreeding coefficient F: $W_F = W_0 * e^{-F\lambda}$, with 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-AGE. 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.

4.5.3 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 modeled. The value must match the value of parameter `quanti_traits` (see section 5.3).

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/array] (opt)

A single decimal number interpreted as the rate of change of the optimum phenotypic values in all patches and for all traits, or an array of trait-specific rates of change of the phenotypic optima in all patches. The array may contain less values than the number of traits, in which cases the values are recycled among traits. The rates are here absolute rates. For instance, a rate of 0.1 will change the local phenotypic optima by 0.1 units per generation (e.g., $3 \rightarrow 3.1 \rightarrow 3.2 \rightarrow 3.3 \rightarrow 3.4$, etc.) This 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/array] (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.

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.

4.5.4 Multi-stage selection in the Gaussian model

selection_matrix [matrix]

The interpretation of the selection matrix for Gaussian and Quadratic selection models differs if `selection_at_stage` is set. In this case, each row corresponds to the selection matrix used for each stage under selection (given in `selection_at_stage`), specified in the the same order. If only one row is given, it is assumed that the selection matrix is identical for all stages under selection. Each column contains the strength of selection (i.e., the selection variance, or diagonal elements of the selection matrix) for each quantitative trait under selection (e.g., when simulating multiple quantitative traits). If the number of columns is inferior to the number of trait under selection (set with `selection_trait_dimension`), it is then assumed that the strength of selection on the remaining traits is the repetition of the previous values (values are recycled). It is possible to set the selection covariance (correlated selection) between traits by specifying the selection variance for each trait and adding all the covariances in additional columns. No correlational selection is assumed if the number of elements in the array is smaller or equal to the number of traits under selection (`selection_trait_dimension`). The stage-specific selection matrices are the same in every patch. Spatial variation in the strength of selection can only happen if a single stage is under selection, and is specified with parameter `selection_variance`.

Examples:

```
selection_trait_dimension 2
selection_at_stage {{1,2,3}}
selection_matrix {{v_{1,1}, v_{1,2}} {v_{2,1}, v_{2,2}, cov_{2,12}} {v_{3,1}, v_{3,2}}}
```

Stage-specific selection variance $v_{1,i}, v_{2,i}, v_{3,i}$ can be different for each trait i. Here, correlational selection on the two traits is only present in stage 2, because the covariance is specified. All unique pairwise covariances must be specified, starting from the first to the last trait, unless they are zero. Selection covariance is 0 by default.

To remove selection on a trait during a stage-specific selection episode, the selection variance for that trait must be set to a very large value corresponding

to very weak selection. This is because selection is always computed for each trait during each episode of selection. For instance, to have a different trait selected at each stage, the example above could become:

```
selection_trait_dimension 2
selection_at_stage {{1,2}}
selection_matrix {{5, 10000000} {10000000, 5}}
```

4.5.5 File output

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. Note that a temporal argument can change the logtime value during the simulation.

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 already present.

4.6 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.

4.6.1 Initialization of trait quant

quanti_init

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 di-allelic**

loci only. Parameter values set for the init model in trait `quant` will apply (see `quanti_init_model`).

Parameters:

quanti_init [integer]

LCE name. The 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.

4.6.2 Initialization of trait ntrl

`ntrl_init`

This LCE can be used to set initial allele frequencies in each patch differently. **It assumes loci carry only two alleles.**

Parameters:

ntrl_init [integer]

LCE name. The value is the rank in the life cycle.

ntrl_init_patch_freq [matrix] (opt)

This is the same as for quanti_init_freq above, although for the ntrl trait instead.

4.7 Population Regulation

regulation

LCE to control population size. For simulations running for more than a few generations, we strongly advise to add this LCE to the simulation to control exponentially growing populations. NEMO-AGE proposes two non mutually exclusive modes of regulation based on density-dependent competition and on ceiling regulation.

Density-dependent competition is based on Beverton-Holt or Ricker competition functions, set with `regulation_by_competition_model`. Both use parameter b defined as the competition coefficient, set with `regulation_by_competition`.

Ceiling regulation removes individuals in excess of the sex-specific carrying capacity of a patch. Supernumerary individuals are randomly drawn and moved to the garbage collector. This regulation mode is useful to limit the total number of individuals in a simulation and prevent bursts of population growth to overfill a computer's memory. Celing regulation is active if `regulation_carrying_capacity` is set.

Note that ceiling regulation is also performed in the `aging` LCE, the stage transition aging LCE specific to 2-stage populations with non-overlapping generations. `Regulation` is thus not needed when `aging` is present in the life cycle, but should be always used with `aging_multi` and stage-structured populations.

Parameters:

regulation [integer]

LCE name. The value is the rank in the life cycle.

regulation_by_competition [double/matrix] (opt)

Parameter used to specify the competition coefficient b of the Beverton-Holt or Ricker functions. Regulation by competition is performed only if this parameter is included in the init file, and correctly set.

Patch-specific competition coefficients can be passed as an array (single-row matrix) with one value per patch.

regulation_by_competition_model [`Beverton-Holt` or `Ricker`] (opt)

There are two models of density-dependent competition, either the `Ricker` or `Beverton-Holt` model. The probability of surviving competition $c[t]$ then varies with the number of individuals within a certain stage at time t, $n[t]$, and the strength of competition b (see Caswell, 2001, p. 506, section 16.1.1.).

`Beverton-Holt`: $c[t] = 1/(1 + b * n[t])$ (default)

non linear density-dep model, pop stabilizes as it approaches the carrying capacity

`Ricker`: $c[t] = exp(-b * n[t])$

exponential model with density-dep regulation, more complex dynamics like cycles and oscillations

regulation_by_competition_affected_age [integer/matrix] (opt)

Parameter used to define the stage(s) affected by density-dependent competition. The default value is 0 (only offspring are affected by competition). If a single stage is affected by competition, the parameter takes an integer argument corresponding to the index of the stage affected. Otherwise, the parameter takes a matrix argument containing the indices of the stages affected. The indices for the stages under competition have to be between 0 and number of columns - 1 in `pop_age_structure`.

regulation_by_competition_count_age_flag [integer: 1 - 4294967295] (opt)

The age flag (binary flag) used to set which stage is part of the count of individuals $n[t]$ exerting competition on the affected stage(s). By default, the value is 4294967294 (all stages except the first offspring stage, identified by the first digit).

Technical note: NEMO-AGE can handle a maximum of 32 stages, corresponding to the same number of binary digits in a 4-bytes integer. The most convenient way to define a flag is to first use hexadecimal digits, where each hexadecimal digit represents four binary digit: all stages corresponds to flag FFFF FFFF (= 4294967295), all stages but offspring (stage 0) corresponds to FFFF FFFE, all stages but stages 0 and 1 is FFFF FFFC etc. Hexadecimal digit then have to be converted to decimal numbers (it is convenient to use an online converter), which are used as arguments for this parameter.

Examples:

```
hexadecimal   decimal      binary
 FFFF FFFF = 4294967295 = (1111)x7 1111 (all stages)
 FFFF FFFE = 4294967294 = (1111)x7 1110 (all adults, no offspring)
 FFFF FFFC = 4294967292 = (1111)x7 1100 (all except stages 0 and 1)
 0000 0001 =          1 = (0000)x7 0001 (only stage 0, offspring)
 0000 0003 =          3 = (0000)x7 0011 (only stages 0 and 1)
 etc.
```

regulation_carrying_capacity [bool] (opt)

If set, `regulation` will randomly remove individuals from each stage within a patch until the patch carrying capacity is reached. The carrying capacity is set with parameters `patch_capacity` (for identical sex-specific carrying capacities) or `patch_nbfem` and `patch_nbmal` (for different sex-specific carrying capacities, see section 3.2). The relative proportions of each stage within a parch are kept after random removal (i.e., the proportions are used as probabilities for the multinomial random draw).

4.8 Recording Statistics

save_stats

This LCE is used to tell the stat-services of the simulation to record the summary statistics specified with the stat parameters of the different simulation components included in the current simulation. The statistics recorded are usually stage-specific, and often target either offspring (stage 0) or adults (all stages > 0). 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` or `aging_multi` may not allow you to record the stats on the offspring, depending on age composition of the population. The recorded stats are written to a text file at the end of each replicate and at the end of a simulation for the averaged stats (text file ending with "_bygen.txt"), but only if the save_files LCE is also present in the life cycle. See chapter 7 for a description of the different statistics available for each simulation component. 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]

LCE name. The 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 7.

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 replicate at each generation recorded. This is an automatic statistic, no additional token is needed to the stat parameter.

4.9 Saving Files

`save_files`

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 to disk during your simulation by NEMO-AGE. This concerns, in particular, the simulation stats recorded by the `save_stats` LCE. Each simulation component (trait or LCE) may define different output files to save specific information (e.g., specific stats or genotypes and phenotypes for a specific trait, etc.) The `save_files` LCE notifies the file manager to initiate the file output process at the point where it has been inserted in the life cycle. The individuals to which the file handlers will have access will thus depend on the position (rank) of the `save_files` LCE in the life cycle. It is not possible, for now, to use `save_files` more than once in the life cycle.

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 7) to save the statistics recorded during the simulation. Other components let the user chose what and when data must be saved on disk (see the trait components).

Parameters:

save_files [integer]

LCE name. The 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 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 with offspring and adults present. In stage-structured populations, with overlapping generations, adults are sampled across all non-offspring stages (i.e., stage > 0).

4.10 Store Data in Binary Files

`store`

This LCE provides a way to save all the trait and individual data to 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. 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.

output files: `".bin" (".tar", ".bz2")`

Parameters:

store [integer]

LCE name. The value is the rank in the life cycle.

store_generation [integer]

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

store_dir [string] (opt)

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

store_recursive [bool] (opt)

This option will tell the program to periodically save the data to a binary file, using the `store_generation` value as the periodicity in generation time. Each generation will be saved to 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.

4.11 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 go and allow for selection to act on the fecundity of the reproducing females instead of their offspring's survival. Composite LCEs often add new parameters (see below). Because the ini file cannot have more than one copy of a parameter, the composite LCE and its base LCEs cannot have different parameters values. To avoid that constraint, the composite LCE redefine the names of the base parameters inherited from their base class. This way, two LCEs can be used in a simulation,

the base one (e.g., `viability_selection`) and a composite LCE derived from the base LCE (e.g., `breed_selection`). The composite LCEs are:

```
breed_selection
breed_disperse
breed_selection_disperse
```

4.11.1 Breed with selection

`breed_selection`

This composite LCE performs **breeding** and **selection** in one step. It inherits the parameters from the `breed` and `selection` LCEs. **It re-names all `selection` parameters, substituting the `selection` prefix with `breed_selection`**. The following features differ from the base LCEs:

- This LCE can simulate **selection on fecundity AND survival**.
- If `breed_selection_fecundity_fitness` is selected (see below), selection occurs on females' fecundity *instead* of offspring (stage 0) survival. Fitness can be relative to other females in the same patch. **The parameters for selection on fecundity are always taken from those set for stage 0**.
- If `breed_selection_fecundity_fitness` is *not* selected, **selection acts on offspring survival or other stages**. Then, offspring fitness cannot be relative to the mean offspring survival. The fitness model must be absolute.
- Selection on females' fecundity still allows for the simulation of **populations with demographic stochasticity**. It is thus not equivalent to a Wright-Fisher model with constant population sizes (for this, see `breed_disperse`).
- **This LCE is faster** than having `breed` followed by `viability_selection` in the life cycle, because genetic inheritance is computed on the selected trait(s) before checking for survival. Inheritance and mutation of the traits not under selection are performed on survivors only.

Parameters:

breed_selection [integer]

LCE name. The value is the rank in the life cycle.

breed_selection_fecundity_fitness [bool] (opt)

If this parameter is added to the init file, then selection acts on the number of offspring produced per female *instead* of acting on offspring survival. The fitness of the female then determines the mean value of the fecundity distribution from which her number of offspring is drawn. This works best when the mean fecundity (set in `pop_transition_matrix`) is large, because only a round number of offspring can be produced. A too low mean fecundity can drive a female's reproductive output to zero if her fitness is also too low, making selection stronger when the mean patch fitness is low. Using relative fitness values can avoid this problem (set `breed_selection_fitness_model` to `relative_(max_)local`). Selection can still affect other stages if needed.

Examples: A simple three-stage population with selection on fecundity and pre-adult survival (stage 1).

```
## POPULATION ##
patch_number          1
patch_capacity        10000
pop_age_structure     {{0, 1, 2}}
pop_transition_matrix {{0, 0, 20}   #fecundity, under selection
                       {0.7, 0, 0}  #pre-adults under selection
                       {0, 0.9, 0}} #reproducing adults
## Life Cycle ##
breed_selection 1
aging_multi     2
regulation      3
save_stats      4
save_files      5

## BREED_SELECTION ##
mating_system                   1 #random mating, see the breed LCE
#to ensure that selection acts on fecundity, we need to add this:
breed_selection_fecundity_fitness
breed_selection_trait           quant
breed_selection_fitness_model   absolute
breed_selection_model           gaussian
breed_selection_trait_dimension 1
breed_selection_at_stage        {{0, 1}}
breed_selection_matrix          {{10}    #fecundity selection
                                 {50}}   #survival (weaker selection)
breed_selection_local_optima    {{2.4}}  #optimum trait 1
```

```
## REGULATION ##
regulation_carrying_capacity

## QUANTITATIVE TRAIT ## trait name is 'quant' for selection
quanti_traits            1
quanti_loci              50
quanti_mutation_rate     0.0001
quanti_allele_model      continuous
quanti_mutation_variance 0.1      #Vm = 2Luv = 2*50*1e-4*0.1 = 1e-3
quanti_init_value        {{2.4}}
quanti_init_model        1
```

Next example is for a case of two traits under selection, one is linked to fecundity selection and the other to offspring survival. Because selection parameters at stage 0 are taken for fecundity selection, we need to add a dummy stage if we want to perform viability selection on the second trait at stage 0 as well. The dummy stage is thus stage 0 "moved up" to stage 1 with transition probability = 1. To prevent selection on a trait during one episode of selection, we drastically decrease the selection strength on the non-targeted trait. We show here only the changes relative to the previous example:

```
pop_age_structure     {{0, 1, 2, 3}}  # we add one stage
pop_transition_matrix {{0, 0, 0, 20}  #fecundity, under selection
                       {1, 0, 0, 0 }  #dummy stage, under selection
                       {0, 0.7, 0, 0} #pre-adults, now no selection
                       {0, 0, 0.9, 0}} #reproducing adults
## Life Cycle stays as above

## BREED_SELECTION ##  add selection on another trait
breed_selection_trait_dimension  2
breed_selection_at_stage {{0, 1}} #now 1 is juveniles
breed_selection_matrix   {{10, 1000000}  #no selection on trait 2
                          {1000000, 20}} #no selection on trait 1
breed_selection_local_optima {{2.4, 10}}
breed_selection_fecundity_fitness

## QUANTITATIVE TRAIT ## add a trait
quanti_traits            2
quanti_loci              50      #50 per trait, same allele model
quanti_init_value        {{2.4, 10}} #start at optimum
```

Caveat: In the above example, the two selection events (fecundity and survival)

won't affect both the adult fecundity and the survival of the individuals born within a year because transition to the dummy stage happens only after the `aging_multi` LCE is called. There might thus be a timing issue depending on the life cycle if, for instance, the local trait optima change at every time step (year). This needs to be taken into consideration when setting the timing of the LCEs.

To avoid this timing problem, you can use `breed_selection` and `viability_selection` in the same simulation. In this case, the selection parameters remain set for two quantitative traits as the `quant` trait dimensionality hasn't changed. However, this time the selection on the second trait in the offspring is right after fecundity selection on the first trait in the adult population, within the same year. This way, we can model selection on seed production and seed survival, for instance. And both can be influenced by variation in local conditions within the same year.

```
## we reset the age-structure to three stages
pop_age_structure     {{0, 1, 2}}
pop_transition_matrix {{0, 0, 20}
                       {0.7, 0, 0}
                       {0, 0.9, 0}}
##  Life Cycle ##
breed_selection     1
viability_selection 2
aging_multi         3
regulation          4

## BREED_SELECTION ##  fecundity selection only
breed_selection_fecundity_fitness
breed_selection_trait            quant
breed_selection_fitness_model    relative_local
breed_selection_model            gaussian
breed_selection_trait_dimension  2
breed_selection_at_stage     {{0}}          #for fecundity selection
breed_selection_matrix       {{10, 100000}} #selection on trait 1
breed_selection_local_optima {{2.4, 10}}

## VIABILITY_SELECTION ## on individuals born this year
selection_trait           quant
selection_fitness_model   absolute
selection_model           gaussian
selection_trait_dimension 2
selection_at_stage        {{0}}             #newborns
selection_matrix          {{100000, 20}}    #selection on trait 2
selection_local_optima    {{2.4, 10}}
```

4.11.2 Wright-Fisher population with migration

`breed_disperse`

This LCE performs breeding and dispersal in a single step and implements backward migration (gametic migration) in populations of constant size. 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 two base LCE's:

- **backward migration**, the columns of the dispersal matrix must sum to 1 instead of the rows, because NEMO-AGE 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).
- This LCE can be used to simulate the **Wright-Fisher model** when the mating system is set to 6 (random mating with selfing rate = $\frac{1}{N}$) or 1 (random mating with two sexes).
- 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 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.

IMPORTANT: the dispersal parameters that are inherited from the disperse LCE must now 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.

Parameters:

breed_disperse [integer]

LCE name. The 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. Growth rates can be patch specific if passed as an array of values.

4.11.3 Wright-Fisher with selection and migration

breed_selection_disperse

This LCE performs reproduction, selection and migration in populations of constant sizes. Therefore, it aggregates the features of both previous composite LCEs, and inherits from breed_disperse and breed_selection. However, performing viability selection and backward migration in populations of constant size can be challenging if the mean fitness is too low to allow the patches to be filled with surviving offspring. The solution implemented uses a minimum fitness threshold for the individuals. If the mean patch fitness of the reproductive adults is below that threshold before mating, the offspring fitness is rescaled so that the mean patch fitness matches that threshold. In other word, the threshold is the minimum survival probability an offspring in a patch can reach and the 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 values, or the populations have accumulated a large genetic load.

Parameters:

breed_selection_disperse [integer]

LCE name. The 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 0.05 by default (5% surviving probability).

IMPORTANT: since `breed_selection_disperse` inherits parameter definitions from `breed_selection` and `breed_disperse`, the selection parameters must use the `breed_selection` prefix, and the dispersal parameters must use the `breed_disperse` prefix, see sections 4.11.1 and 4.11.2 above.

Chapter 5

Traits

The traits included in the current version of NEMO-AGE are:

- `ntrl` (neutral markers, including microsatellites, SNPs, etc.)
- `quant` (quantitative traits)
- `delet` (deleterious mutations)

Un-ported traits from NEMO are:

- `dmi` (Dobzhansky-Muller Incompatibility loci)
- `fdisp/mdisp` (sex-specific, evolving dispersal traits)
- `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 7.

5.1 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-AGE 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).

Parameters:

***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 line per chromosome (in [cM] by default). The number of chromosomes is then deduced from the number of lines.

Note: because matrices in input must carry the same number of elements per line, this parameter does not allow for different number of loci per chromosome. This is not true for the other types of map.

***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).

***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 ($0.1 \times$ 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. For instance, our two loci would receive positions 340 and 350 to keep a 1% recombination rate when the scale is changed by another trait to 0.1cM.

5.2 Neutral markers

name: **ntrl**
files: `".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 by 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 3.2) 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 allele frequencies for di-allelic loci (see section 4.6.2).

ntrl_loci [integer]

Number of (diploid) neutral markers per individual.

ntrl_all [1 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 `[0, ntrl_all[` range.

ntrl_init_model [0,1] (opt)

This option sets the way marker genes are initialised. The mode #0 means "no variance"; all alleles have same value (i.e. 0) at the start of a replicate. Mode #1 means "maximum variance"; the allele values are set randomly within the range `[1, ntrl_all]`. Mode #1 is the default mode. See section 4.6.2 for a different way of initialising allele frequencies within patches.

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 5.1.

ntrl_save_genotype [string] (opt)

If this parameter is present, the population genotypes will be saved in a text file with the `".dat"` extension. Three 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. This format is ideal for the R software and analysis with the HIERFSTAT R package by J. Goudet. Extra data is saved for each individual: stage, age (0 = offspring), sex (0 = male, 1 = female), home (natal patch), ped (pedigree class, see section 1.8), 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.

- FSTAT (fstat)

  The file format is (almost) the same as that used by the FSTAT program (Goudet 1995). It adds information about each individuals (stage, age, sex, pedigree, and natal patch) to the genotype data. An example of an output file is given below.

- GENEPOP (genepop)

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

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-AGE to save the within patch F_{ST} values per-locus using the Weir and 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 generation periodicity of the output files, or the generations at which the files should be saved if provided as multiple values in an array.

Note about reading an FSTAT file: as discussed in section 3.2.3, 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 `stage`, `age`, `sex`, `ped`, and `origin` "loci".

Here is an example of the *extended* FSTAT file format :

```
5 10 20 2
loc1
loc2
loc3
loc4
loc5
stage
age
sex
ped
origin
1 1414 1019 2002 0820 0307 0 0 1 1 1
1 0814 0219 2002 2020 0307 0 0 1 1 1
1 0808 0217 1902 0820 0907 1 1 1 2 1
1 0820 0209 1902 0805 0918 1 2 0 0 4
[...]
4 0307 1308 0220 0401 0115 0 0 1 1 4
4 0905 1213 0302 0312 0506 1 3 1 2 2
[...]
5 2017 1010 2013 1812 1505 0 0 0 1 5
5 2017 1008 2013 1811 1505 1 2 1 2 3
```

The first line contains the population number (5 pops here), the number of locus (5+5), 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 genotype.

The five next lines are the locus names plus the "locus names" for the five last values; the stage, 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 stage (depending on the age structure defined by the `population` component).

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 digit 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 five numbers. The first is the stage (0 = offspring), the second is the age (number of cycles survived), the third is the sex tag (1 = female, 0 = male), the fourth is the individual's pedigree class, based on 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 (natal patch).

This file format is close to the original `FSTAT` (Goudet, 1995) input file format (`http://www2.unil.ch/popgen/softwares/fstat.htm`) with the addition of the five last columns of the individual data. The `HIERFSTAT` **R** package (see: `http://www2.unil.ch/popgen/softwares/hierfstat.html`; Goudet, 2005), 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-AGE 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 7.2 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. The Weir and Cockerham (1984) F-statistics estimates are also available with the stat option `fstatWC`.

5.3 Quantitative traits

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

Quantitative traits are traits that show a continuous distribution of values, also sometimes called *metric* traits. A classic example is body weight, a trait that varies continuously both among and within individuals. The trait implementation models these aspects of trait variation by using a continuum-of-allele model of mutation where each mutational effects are drawn from a Normal distribution (see parameters below). In addition, a di-allelic model is also implemented where mutations can only take two values ($\pm a$). This model is provided for comparisons with classical quantitative genetics models.

The trait architecture is kept simple, with **additive** action of the loci (no dominance, no interactions). When multiple traits are modeled, the loci are completely **pleiotropic**, meaning that each locus has an effect on each trait and the mutation

effects, drawn from a multivariate Normal distribution, can be correlated. In this way, the evolution of correlated traits and genetic constraints on adaptation can be modeled. Environmental variance can also be modeled, as well as spatially and temporally varying selection pressures (see the selection LCE).

The statistics implemented return the additive genetic variation within populations (V_a), 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 or the **D**-matrix among demes, when two or more traits are modeled.

quanti_traits [integer]

The number of traits to model. The number of traits is not limited. If two or more traits are modelled, the mutational covariance can be set and the statistics returned include the genetic correlation of the traits, and the eigen decomposition of the genetic covariance matrix both within (**G**-matrix) and among (**D**-matrix) demes.

quanti_loci [integer]

Number of additive loci that determine the trait(s). Loci are diploid. The trait value is set by summing the allelic values at all loci. When two or more traits are modelled, they share the same loci and each locus has an effect on each trait (i.e., fully pleiotropic loci). The mutation effects on the traits can be more of less correlated depending on the mutational covariance (see below).

quanti_mutation_rate [double]

The mutation rate, identical for all loci. The mutation effect(s) depends on the allelic model, set below.

quanti_allele_model ["diallelic", "diallelic_HC", "continuous", "continuous_HC"] (opt)

Two ways to model the mutational effects: "diallelic" if mutations can only take ± a given value (or two different values, see below), or "continuous" if mutations are drawn from a Normal distribution, with variance (and correlation for the multiple traits) set below. The default model is "continuous".

The two Hous-of-Cards (HC) variants specify a different way of modelling mutations. In the non-HC models, a new mutation effect is added to the existing allelic value, whereas in the HC models, the new effect replaces the existing allele.

quanti_allele_value [double/matrix] (opt)

The effect size of the mutation(s) or allelic values at a loci in the di-allelic mutation model. If a single value is given, that value is used for all loci. A

matrix can be used to pass locus-specific values. If the matrix has a single row (an array), the mutational effects are ± the given values at each locus. Two different values per locus can be specified if two rows are provided instead of one. The number of columns of the matrix must match the number of loci.

quanti_mutation_variance [double] (opt)

The variance of the Normal distribution of the mutational effects (the mutation effect size) in the "continuous" mutation model. The same variance is used for all traits unless the full mutation covariance matrix is specified (see below).

quanti_mutation_correlation [double] (opt)

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

quanti_mutation_matrix [matrix] (opt)

The covariance matrix of the multivariate Normal distribution used to draw the mutation effects in the continuous allelic model. Can be used to set different mutational variances for the different traits. This must be a square symmetrical and semi-definite positive matrix (with trait mutational variance on the diagonal and the mutational covariance off the diagonal). This matrix is often referred to as the **M**-matrix.

quanti_recombination_rate [double / matrix] (opt)

The recombination parameters are now (v2.3) managed by the genetic map. See section 5.1 for the details.

quanti_init_value [matrix] (opt)

The initial genotypic value of the traits can be set here with an one-dimension array containing one value for each trait. The default trait value is 0 by default. This parameter is valid for the whole metapopulation. The allelic values at each locus are set by dividing the initial trait value by two times the number of loci affecting that trait. Thus, the loci all receive the same initial allelic value. The amount of initial allelic variation in the population can be set with the `quanti_init_model` parameter.

This parameter sets all individuals with the same initial trait values in every patch. The LCE quanti_init must be used to set patch-specific initial values for each trait.

quanti_init_model [0,1,4] (opt)

If the initialisation model is set to "0", the initial population will be monomorphic for the initial trait values specified with `quanti_init_value`. If the init model is set to "1", a random mutational effect is added to each locus, on top of the initial value. Model "1" is the default. Models "2" and "3" are still experimental and should not be used. Model "4" initializes di-allelic loci with opposite allelic values at alternative loci (positive at one locus and negative at the next), and does not add any mutation to the initial allele values.

quanti_environmental_variance [double] (opt)

Variance of the environmental deviation of the trait's phenotype. Is zero by default (no environmental variance). A random Gaussian value with mean zero is added to the genotypic value otherwise.

quanti_output [bool, "genotypes"] (opt)

If present, the phenotypes of the whole population are saved in a text file, with one individual per row. The genotypic trait values are added if the environmental variance is not null.

The data saved is:

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

with pop the patch identifier, Pi the phenotypic and Gi the genotypic values of each trait i (Gi is added only if environmental variance is not null), stage (0 = offspring), age (number of cycles survived), sex (0 = male, 1 = female), home the natal patch, ped the pedigree class (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), isMigrant the number of parents of the individual that are immigrants, and the father, mother, and own ID, which are unique identifiers assigned to individuals that can be used to reconstruct the population's pedigree.

If the option genotypes is passed, the allelic values are also saved and 2×(number of loci)×(number of traits) columns are added to the file between pop and Pi, with headers ti.lj.1 for the mother-inherited allele at locus j affecting trait i and ti.lj.2 for the father-inherited allele.

quanti_logtime [integer or matrix]

The frequency at which phenotypes should be saved, if a single number, or the specific generations at which the files should be saved if provided as multiple values in an array.

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. *(Originally contributed by Sam Yeaman)*

quanti_freq_logtime [integer] (opt)

The recording frequency in generation time of the allele frequencies.

Example of an output '.qfreq' 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
...
```

5.4 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 4.5). 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 5.1.

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 3.2.3).

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. Individual information (stage, age, sex, pedigree class, patch of origin and unique ID) is added in the last six columns of the file.

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.

Chapter 6

Examples

6.1 Demography in a stage-structured population

We present how to model the demography of the bark beetle *Scolytus ventralis* with NEMO-AGE. We model the demography of a single population for 1000 time units (generations). The population is initiated with 10 males and 10 females in each stage but stage 0. An estimate of the transition matrix for this species can be found in the Comadre database (Salguero-Gómez et al., 2016, the original publication for the matrix is Berryman (1973)). The population is structured in six developmental stages (eggs, larva I-II, larva III-IV, larva V-VI, pupa and brood adults, reproductive adults). Individuals can survive as reproductive adults. We assume that competition affects the survival of the first larval stage through a Beverton-Holt competition function. Census was performed immediately after breeding for correspondence with the transition matrix in the deterministic matrix population model (from Caswell, 2001, see also Cotto et al., 2020 for a more complete treatment of this example). Figure 6.1 illustrates the results of the individual-based simulations (for 10 replicates) with comparison with the deterministic expectation. We give here the full parameter specification, also available in the file `examples/Example2-demographyScolytus.ini` of NEMO-AGE distribution.

```
## NEMO CONFIG FILE ##
run_mode overwrite
random_seed 2
root_dir examples
filename bark_beetle

## SIMULATION ##
replicates 10
generations 1000
```

```
## POPULATION ##
patch_number 1
patch_nbfem  10
patch_nbmal  10

### STAGE STRUCTURE and TRANSITION MATRIX ###
pop_age_structure {{0,1,2,3,4,10}}
pop_leslie_matrix {{0, 0, 0, 0, 0, 124}
                   {0.83, 0, 0, 0, 0, 0}
                   {0, 0.92, 0, 0, 0, 0}
                   {0, 0, 0.93, 0, 0, 0}
                   {0, 0, 0, 0.26, 0, 0}
                   {0, 0, 0, 0, 0.43,0.53}}

## LIFE CYCLE EVENTS ##
breed       1
save_stats  2
save_files  3
regulation  4
aging_multi 5

## REGULATION ##
regulation_by_competition_model Beverton-Holt
regulation_by_competition 0.0005
regulation_by_competition_affected_age 1

## MATING SYSTEM ##
mating_system 1       #random mating

## STATS ##
stat demography       #only record patch sizes
stat_log_time 1
```

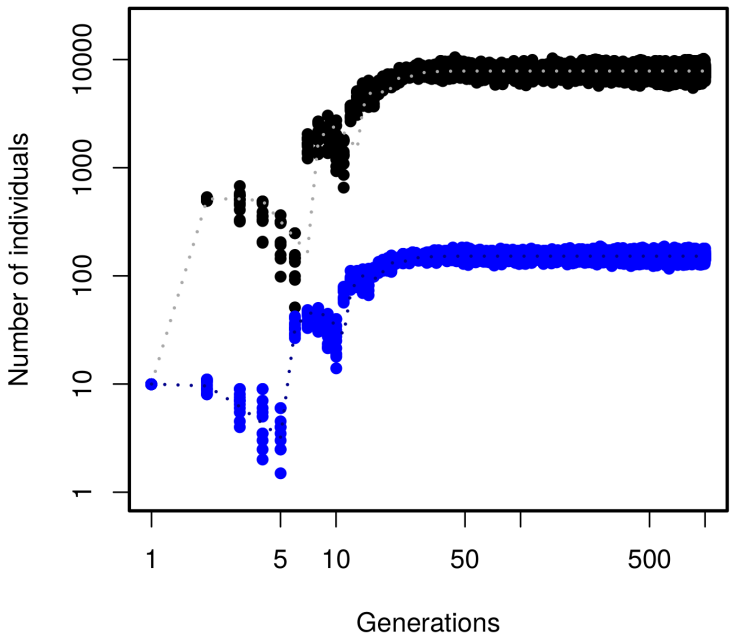

Figure 6.1 Population dynamics of a population of *Scolytus ventralis*. The dots corresponds to the results of the individual based simulations (10 replicates) and are compared to the deterministic expectation (dotted lines). Blue: reproductive females, Black: stage 1.

6.2 Evolution in a stage-structured population

We illustrate a simple situation where taking into account stochastic processes is crucial (Figure 6.2). We follow from example 1 above to investigate the effect of a change in the environment affecting survival at the larval stages. The example aims to illustrate the functionality of the program without sticking to a precise biological reality. For simplicity, however, we present the analysis as an investigation of a possible biological scenario. A sudden environmental change affecting survival can correspond to a chemical treatment to eliminate the insects from trees with some economical values. We assume that the treatment affects survival in the larval stages (stages 1 to 3) by shifting some physiological optimum genetically encoded. The population would survive the treatment only by evolutionary adaptation (i.e., evolutionary rescue). Results provided by NEMO-AGE simulations are presented in Figure 6.2 and compared to a deterministic expectation (recursions from Barfield et al., 2011). Prior to the results presented, we performed burn-in simulations for

20,000 generations to reach the variance expected at the mutation-selection equilibrium (0.1 here). NEMO-AGE allows to store the state of the population at given generations. We initialized the simulations with the population stored at the last generation of the burn-in period (5 replicates).

The first simulation thus runs the burn-in under a constant phenotypic trait optimum and saves the whole population after 20,000 years to a set of compressed binary files (one per replicate). Those files will then serve as source populations for the second simulation, which models the evolutionary rescue scenario and runs it for 10 replicates and 2000 years. Both simulations can be run from a single command in the terminal with both init files (`Example3-rescueScolytus-popini.ini`, `Example3-rescueScolytus.ini`) passed as arguments to `nemoage` in the right order (chained simulations):

```
\$ nemoage Example3-rescueScolytus-popini.ini Example3-rescueScolytus.ini
```

Burn-in parameters in the first init file:

```
## BURN-IN SIMULATION ## Example3-rescueScolytus-popini.ini
run_mode overwrite
random_seed 2
root_dir examples
filename bark_beetle_popini

## SIMULATION ##
replicates  5
generations 20000  # in years

## POPULATION ##
patch_number 1
patch_nbfem  10
patch_nbmal  10

### AGE STRUCTURE and LESLIE MATRIX ###
pop_age_structure {{0,1,2,3,4,5}}
pop_transition_matrix {{0, 0, 0, 0, 0, 124}
                       {0.83, 0, 0, 0, 0, 0}
                       {0, 0.92, 0, 0, 0, 0}
                       {0, 0, 0.93, 0, 0, 0}
                       {0, 0, 0, 0.26, 0, 0}
                       {0, 0, 0, 0, 0.43,0.53}}

## LIFE CYCLE EVENTS ##
```

```
quanti_init         0
breed               1
save_stats          2
save_files          3
regulation          4
viability_selection 5
aging_multi         6
store               7

## STORE ## saves the whole pop in a binary file at each replicate
store_generation 20000
store_dir popini       # saved below root_dir: examples/popini
store_noarchive        # do not collect the files in a tar archive

## REGULATION ##
regulation_by_competition_model Beverton-Holt
regulation_by_competition       0.0005   #competition coefficient
regulation_by_competition_affected_age 1 #only juveniles affected

## QUANTITATIVE TRAITS ##
quanti_traits 1                 # simulate a single quanti trait
quanti_loci 10                  # L
quanti_mutation_rate 0.0001     # u
quanti_mutation_variance 0.05   # a^2; expected Vm ~ 0.0001 (2Lua^2)
quanti_environmental_variance {{0.03}} # h2 ~ 0.5
quanti_init_trait_values {{1}}  # for the quanti_init LCE

## SELECTION ##
selection_trait quant           # selection on the quanti trait
selection_model gaussian        # specific to quanti trait
selection_trait_dimension 1     # one quanti trait under selection
selection_local_optima {{1}}    # the trait optimum value
selection_at_stage {{1,2,3}}    # multi-stage selection
selection_matrix {{5}{10}{25}}  # w^2, selection variances

## MATING SYSTEM ##
mating_system 1                 # random mating

## STATS ##
stat demography quanti          # pop and quanti stats
stat_log_time 1000
```

The parameters of the second simulation only differ for the life cycle and the shift

of the trait optimum value after 10 years. The following init file initializes each replicate from a source file of the burn-in simulation (individuals are copied from the source binary files). We run 10 replicates of the evolutionary rescue scenario from the 5 source replicates, changing the source population every other replicate. Parameters identical to the burn-in simulation are not repeated below. The full init file can be found in the NEMO-AGE distribution. The shift in the trait optimum (set with `selection_local_optima`) is specified by using temporal arguments to change the value from 1 at generation 0 (`@g0 {{1}}`) to 4 at generation 10 (`@g10 {{4}}`):

```
selection_local_optima (@g0 {{1}}, @g10 {{4}})
```

The full parameter file is:

```
## EVOLUTIONARY RESCUE SIMULATION ## Example3-rescueScolytus.ini
run_mode overwrite
random_seed 2
root_dir examples
filename bark_beetle_rescue

replicates  10       # we run 10 replicates
generations 2000     # 2000 years

## POPULATION ##
# skipped, same as in the burn-in init file

## SOURCE POPULATION FROM BURN-IN SIMULATION ##
source_pop examples/popini/bark_beetle_popini #check source directory
source_preserve 1              # no random draw with replacement
source_replicate_digit 1       # e.g. bark_beetle_popini_1_20000.bin
source_replicates 5            # source had only 5 replicates

### AGE STRUCTURE and LESLIE MATRIX ###
# skipped, same as in the burn-in init file

## LIFE CYCLE EVENTS ##
breed               1
save_stats          2
save_files          3
regulation          4
viability_selection 5
aging_multi         6
```

```
## REGULATION ##
# skipped, same as in the burn-in init file

##QUANTITATIVE TRAITS##
# skipped, same as in the burn-in init file

## SELECTION ## now we shift of the optimum at year 10
selection_trait quant
selection_model gaussian
selection_trait_dimension 1
selection_local_optima (@g0 {{1}}, @g10 {{4}}) # SHIFT OF OPTIMUM
selection_at_stage {{1,2,3}}
selection_matrix {{5}{10}{25}}

## MATING SYSTEM ##
# skipped, same as in the burn-in init file

## STATS ##
stat demography off.quanti adlt.quanti
stat_log_time 1               # record every year/generation
```

6.3 Eco-evolutionary dynamics of two alpine plants during climate change

We provide an example demonstrating how Nemo-age can be used to investigate the ecological and evolutionary responses of a species to a changing environment within a spatially explicit framework. This example emphasizes the main features of Nemo-age and shows how to combine species data with simulation components. The aim of the simulation is to model the changes over time of a species' range under two climate change scenarios. The full model, and result analyses can be found in Cotto et al. (2017). Along the way, we will further propose solutions to issues that can occur as a result of a lack of estimates for some parameters. The .ini and other parameter files are provided in the `examples` folder of the distribution.

Stage-specific demography

We here focus on the response of two alpine plants, *Campanulla pulla L.* and *Dianthus alpinus L.*, endemic to the Austrian Alps, to the projected changes of climatic conditions. For both species, we modeled a life cycle with four stages: seeds,

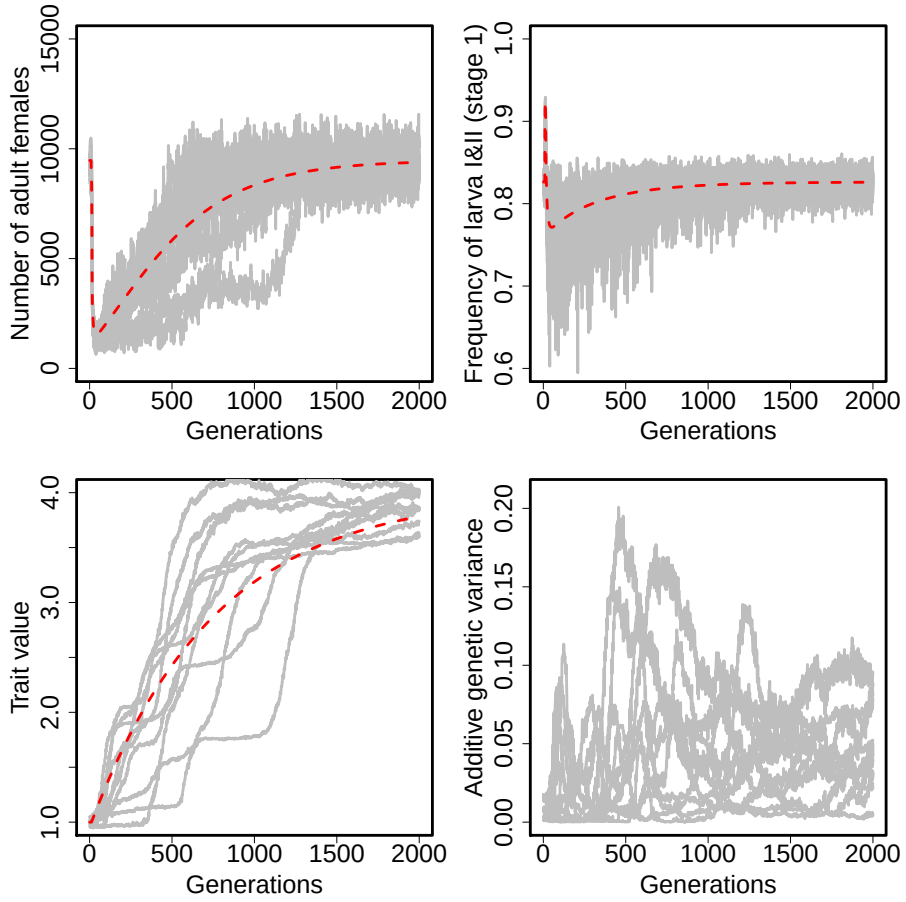

Figure 6.2 Demographic and evolutionary dynamics of the *Scolytus ventralis* population following a shift in the environment. Dots represent the results of the simulations (10 replicates) while the dashed line represents the deterministic expectation. The demographic model is the same as in Figure 6.1. Prior to generation 10, selection is stabilizing around an optimum phenotype that we set at 1 (arbitrary units). At generation 10, the optimum was shifted to 4. Strength of selection (ω^2): 5, 10 and 25 on stages 1 to 3 respectively. Genetic parameters: 10 loci, per-locus mutation rate = 0.0001, per-locus mutation variance = 0.05, heritability = 0.5.

seedlings, preadults and adults. Below, we provide the transition matrix parameter (M) where fecundities (i.e. the first row) correspond to the number of seeds from sexual reproduction.

$$M = \begin{pmatrix} s_0(1-g_r) & 0 & 0 & F_{ad}f_{fl} \\ g_r & 0 & 0 & 0 \\ 0 & s_{sdl} & 0 & 0 \\ 0 & 0 & s_{prea} & s_a \end{pmatrix}$$

The cloning rate cl_r is independent from the transition matrix in the parameter file. With cloning, adult individuals contribute asexually to the seedling stage (stage 1). The biological significance and values for parameters are given by

Parameter	Significance	*Campanula pulla*	*Dianthus alpinus*
s_0	survival in the seedbank	0	0.33
$F_{ad} \cdot f_{fl}$	seed yield * flowering rate	96	9.6
g_r	germination rate	0.165	0.165
cl_r	cloning rate	0.5	2.6
s_{sdl}	seedling survival rate	0.71	0.71
s_{prea}	pre-adult survival rate	0.71	0.71
s_a	adult survival rate	0.7	0.7

The parameter values are from Dullinger et al. (2012); Hülber et al. (2016); Cotto et al. (2017). *Dianthus* differs from *Campanula* essentially by producing less seeds from sexual reproduction, but those seeds can remain in a seedbank. *Campanula* produces large quantities of seeds that do not survive over winter.

The succession of life cycle events in the life cycle as specified in NEMO-AGE parameter file is:

```
quanti_init          0  #patch-specific init trait values
breed                1  #with random mating and hermaphroditism
disperse             2  #with seed dispersal, affects only stage 0
cloning              3  #adds individuals to stage 1 -- seedlings
regulation           4  #d.d. regulation with BH competition function
save_stats           5  #census time
viability_selection  6  #on stage 1 -- seedlings
aging_multi          7  #stage transition
save_files           8  #write summary stats to file
store                9  #save whole pop in a binary file
```

The population and reproduction parameters are:

```
## valid for all simulations with Campanula

### AGE STRUCTURE and TRANSITION MATRIX ###
pop_age_structure {{0,1,2,3}}
pop_transition_matrix {{0, 0, 0, 96}
                       {0.165, 0, 0, 0}
                       {0, 0.71, 0, 0}
                       {0, 0, 0.71, 0.7}}  #surv. adlt, sa = 0.7
## MATING SYSTEM ##
mating_system 6

## CLONING ##
cloning_rate 0.5
```

Population regulation

We implemented density regulation on seedlings that suffer competition from established adult plants within patches. We used the Beverton-Holt (BH) density-dependence model. We determined the competitive weight of individuals of both species, corresponding to the competition coefficient in the BH model, by simulating a single population and adjusting the competition coefficient to reach a difference in equilibrium population size between the two species proportional to the one expected from Dullinger et al. (2012) (see Figure 6.3).

```
## valid for all simulations with Campanula
## REGULATION & COMPETITION ##
regulation_by_competition              0.0014  #for Campanula
regulation_by_competition_affected_age 1       #seedlings
regulation_carrying_capacity                   #also ceiling reg.
```

Setting the competition parameters. To set the competitive weight of each individual (we did not distinguish between the different stages for this example), one first needs to define a target population size for a single cell (without dispersal, but with selection if necessary). The second step is to run preliminary simulations to explore numerically the equilibrium population size given a competitive weight. Figure 6.3 illustrates this process.

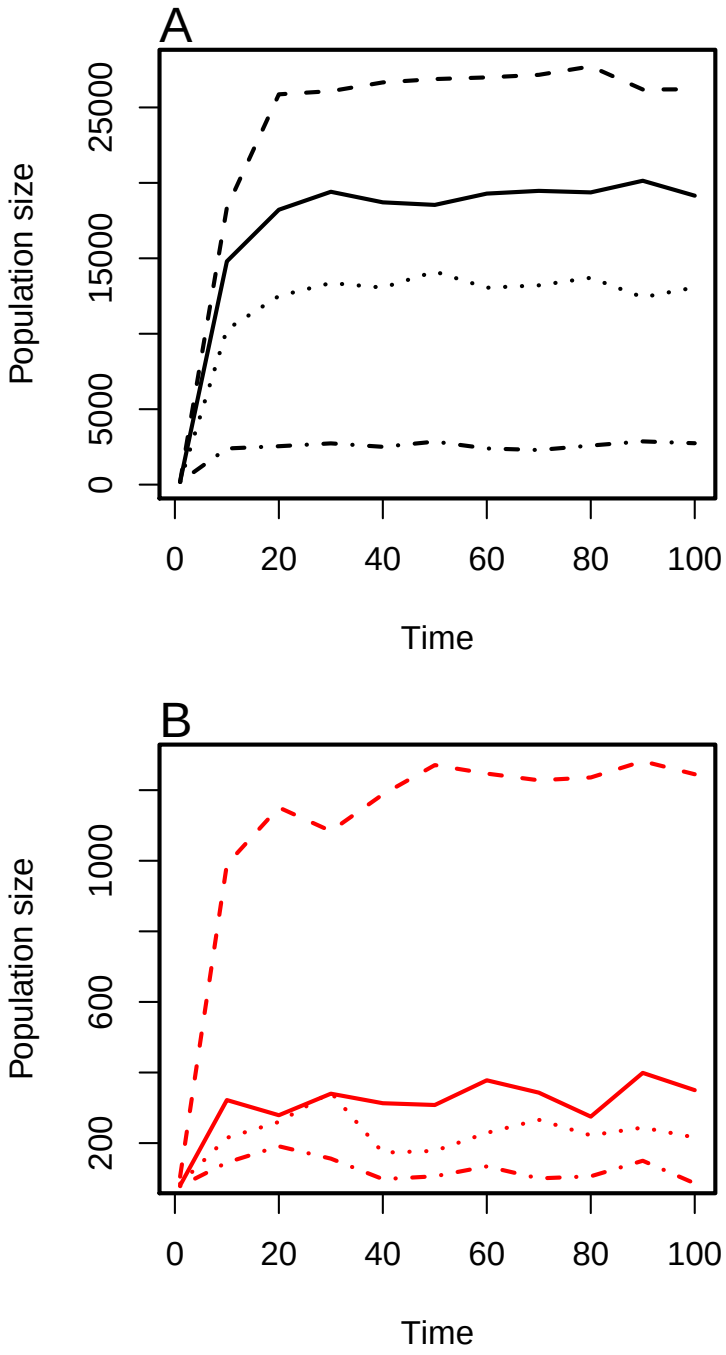

Figure 6.3 Population size as a function of time for different competitive weights for *Campanula* (A) and *Dianthus* (B). Full lines: Competitive weights kept for the simulation. Values used for NEMO-AGE parameter `regulation_by_competition`, for *Campanula* = 0.001 (dashed), 0.0014 (full), 0.002 (dotted), 0.01 (dotdash), for *Dianthus* = 0.01 (dashed), 0.037 (full), 0.05 (dotted), 0.1 (dotdash).

Seed Dispersal

We used the dispersal kernels of the seeds determined by Dullinger et al. (2012) (and used in Cotto et al. (2017)) to build NEMO-AGE dispersal parameters (i.e., the patch connectivity matrix and the dispersal probability matrix). The reduced dispersal matrices are stored in separate files within the `examples/param/` folder and passed to the dispersal parameters as arguments:

```
## valid for all simulations with Campanula
## DISPERSAL ##
dispersal_connectivity_matrix &param/pConnectedcampanula.txt
dispersal_reduced_matrix      &param/pDispcampanula.txt
```

Space and environment variables

Once the biology of the species is determined, the next step is to define the study area and the relevant environmental variables. For our example, we selected a 64km^2 area within the Austrian Alps. The simulated area is divided into a grid of 32×32 patches (i.e. 250m scale; Figure 6.4). We then used the species distribution modeling (SDM) framework BIOMOD2 (Thuiller et al., 2016) to obtain a map of the probability of occurrence of the focal species, based on presence-absence observations and environmental variables (see Figure 6.4, 2010). We used two WorldClim bioclimatic variables (http://www.worldclim.org) representing the mean amount of precipitation (BIO12) and the mean annual temperature (BIO1), and the abundance of carbonate in the soil (strongly structuring plant populations) as a non-climatic environmental variable. We considered that the species was present in a given patch when its probability of occurrence was above a threshold value (provided by the BIOMOD2 output, see Cotto et al., 2017). The SDM was run over the whole species range in the Austrian Alps at the 250m scale. We used that output to set the initial distribution of occupied patches in our grid, setting carrying capacity to 0 in patches predicted as un-occupied. The carrying capacities are passed as an array of 1024 values, saved in text files: `examples/param/proj_(campanula/dianthus)_I_1.txt` (for grid 1). Patch sizes are initialized at 100 individuals at generation 0 before setting the carrying capacity to > 3000 at generation 1. The values of the environmental variables in each grid cell (patch) are stored as (1024×3) matrices in the file `examples/param/currentE_grille1.txt` (ordered as {BIO1, BIO12, Carbonate}).

```
## burn-in simulation for Campanula
## POPULATION ##
patch_number 1024
patch_nbfem ( @g0 &param/proj_campanula_I_1.txt, \ #init at 100
```

```
              @g1 &param/proj_campanula_K_1.txt )
patch_nbmal 0
```

Stage-specific selection

Climate affects survival of the seedlings in our model (see Cotto et al., 2017). The following step thus requires to define the relationship between selection, seedling survival and the environment. In the current example this relationship is unknown because seedling survival in different environment was not investigated. The approach chosen proposes that the phenotype of an individual is its climatic niche and that the optimum of the Gaussian selection function is the current environment. Therefore, an individual performs best when its niche, i.e. its phenotype, matches the current environment. We thus considered a three-trait phenotype, each trait corresponding to one of the three environmental variables we used in the niche modeling framework (see previous paragraph). To complete the selection model, we used a baseline value for the width of the Gaussian selection function (inversely related to the strength of selection) obtained from the environmental variation contained within the species' range as predicted by the niche modeling framework. This variation corresponds to a lower bound to the strength of selection acting at the scale of individuals. A sensitivity analysis, investigating several stronger selection intensities than this baseline is then required (see below). Lastly, we assumed that each trait in the multivariate phenotype is controlled by ten unlinked additive loci:

```
## burn-in simulation for Campanula
## SELECTION ##
selection_trait           quant
selection_trait_dimension 3
selection_model           gaussian
selection_at_stage        {{1}}
selection_matrix {{15.21,8179,37.94}}  #{{Bio1, Bio12, Carbonate}}
selection_local_optima    &param/currentE_grille1.txt

## QUANTITATIVE TRAITS ## valid for all simulations with Campanula
quanti_traits             3
quanti_loci               10
quanti_mutation_rate      0.001 0.0001  # 'sequential' parameter #1
quanti_mutation_variance  0.05
quanti_environmental_variance {{2, 45, 1}} # sets h^2 < 1
```

Sensitivity analysis

Not parameter values of a complex eco-evolutionary model can be known for every study case. To palliate this problem, it is advised to use a sensitivity analysis to evaluate the importance of a given parameter. This way, the uncertainty associated with the parameter can be evaluated from the range of simulation outcomes. In the present example, the amount of genetic variation at the phenotypic traits was not known. We thus ran simulations for two different mutation rates (`quanti_mutation_rate 0.001 0.0001`), a key parameter conditioning the potential evolutionary response fueled by standing and de novo genetic variation. Further sensitivity analysis related to this example can be found in Cotto et al. (2017) (e.g., selection strength, adult survival).

Climate change

The last step for performing eco-evolutionary projections of the change in species' range is to determine the change of the climatic conditions into the future. This is done by using climate model projections of the environmental variables Bio1 and Bio12 and integrating them into the simulation framework. We use those climate projections to determine the future phenotypic optima of the phenotypic traits. Such climate projection are available from the International Panel on Climate Change (Moss et al., 2010) and need in general to be downscaled to the spatial scale of the study. We used here climatic projection as formatted for Cotto et al. (2017). Climatic projections are often available for time intervals larger than a single year (the timescale of simulations), so that climatic projections need to be extrapolated within these intervals. NEMO-AGE allows linearizing climatic projections by implementing a linear rate of environmental change between two projections and by allowing this rate to be updated at any given year. The rate of per-cycle change in the local trait optima is specified with parameter `selection_rate_environmental_change` which receives a (1024×3) matrix with each per-deme, per-trait rate value. The matrix is updated every 10 years with new rates computed from the climate change scenarios. We used projections from scenarios RCP 2.6 and 8.5 between years 2020 and 2090 (correspond to "years" 10 and 80 of the simulaitons). The climate change simulations are run as a separate set of simulations, which use the populations from the burnin simulations as starting populations.

```
## Climate change simulation for Campanula
## Change of selection parameters for trait 'quant'
## First, the local optima are set to their initial values

selection_local_optima &param/currentE_grille1.txt

## Then, local optima start to change at generation 10 at given rates
```

```
## The rate matrices are saved in external files

selection_rate_environmental_change (@g0 &param/constantE.txt, \
 @g10 &param/rate_2020_s1_1.txt, \ # first set of values
 @g21 &param/rate_2030_s1_1.txt, \ # for RCP 2.6
 @g31 &param/rate_2040_s1_1.txt, \
 @g41 &param/rate_2050_s1_1.txt, \
 @g51 &param/rate_2060_s1_1.txt, \
 @g61 &param/rate_2070_s1_1.txt, \
 @g71 &param/rate_2080_s1_1.txt, \
 @g81 &param/rate_2090_s1_1.txt, \
 @g91 &param/constantE.txt) \
 (@g0 &param/constantE.txt, \      # second set of values
 @g10 &param/rate_2020_s3_1.txt, \ # for RCP 8.5
 @g21 &param/rate_2030_s3_1.txt, \
 @g31 &param/rate_2040_s3_1.txt, \
 @g41 &param/rate_2050_s3_1.txt, \
 @g51 &param/rate_2060_s3_1.txt, \
 @g61 &param/rate_2070_s3_1.txt, \
 @g71 &param/rate_2080_s3_1.txt, \
 @g81 &param/rate_2090_s3_1.txt, \
 @g91 &param/constantE.txt)
```

Simulation setup

At this point, we have set all the elements to perform species' range projection with Nemo-age. The simulations are processed in two phases. First, we perform burn-in simulations for 5000 generations (years) prior to modeling the environmental change. Burn-in allows populations to reach both genetic and demographic equilibrium. In this example, burn-in simulations are initiated with the spatial distribution predicted from the niche modeling framework (SDM), where patches predicted as unoccupied are set with a carrying capacity of zero. This avoids colonization of patches outside the initial range during burn-in. The initial phenotype of each individual is set equal to the current environment within their cell, with some random variation. We thus assume that the populations are initially almost perfectly adapted to their local environment. The phenotypic and genetic distribution within cells (patches) around the trait optimum values will then reach their migration-selection-mutation-drift equilibrium during burn-in. A single replicate is performed for each parameter set in the burn-in simulations. The replicates are then used as start population for the second phase, the climate change simulations. These simulations start at an hypothetical year 2000 and assume that the environment starts to change in 2010 (the first climate projection is for 2020) until 2090. After 2090, the environment

stays constant for 60 year, for a total of 150 years. We save summary statistics, monitoring the demographic and evolutionary state of the whole population along the way. We can then visualize the species' range dynamics over time (Figures 6.4, 6.5) and the changes of the populations genetic and phenotypic distributions (Figure 6.5).

```
# Burn-in simulation parameters for Campanula
## SIMULATION ##
filename    campanula_sa07_mu%'.4'1
replicates  1
generations 5000

## saving the population in binary files requires the 'store' LCE
## STORE ##
store_generation 5000
store_dir        popini
store_noarchive          #do not produce a tarball file
```

```
# Climate change simulation parameters for Campanula
## SIMULATION ##
filename campanula_s%'1[13]'2_sa07_mu%'.4'1
# we have two 'sequential' parameters with two values each:
#  %1 = quanti_mutation_rate
#  %2 = selection_rate_environmental_change

replicates  10
generations 150

## POPULATION ##
patch_number 1024
patch_nbfem  3000
patch_nbmal  0

## SOURCE POPULATION ## source pop from burn-in binary file
source_pop popini/campanula_sa07_mu%'.4'1
source_preserve
source_replicates 1
source_replicate_digit 1

[...]
# remaining parameters are as those in the burn-in sims
# except for the rate of change of the environment (see above)
```

The full init files for *Campanula* and *Dianthus* and their respective external parameter files are available with the distribution of NEMO-AGE in folder `examples/Example1-*` and in folder `examples/param/`.

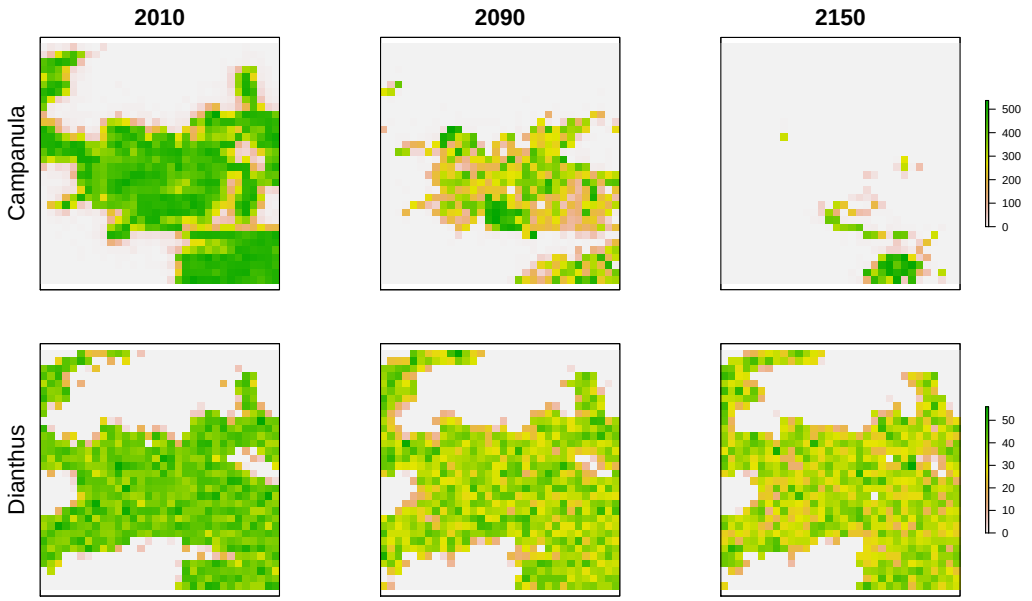

Figure 6.4 Distribution and size of the populations within the studied area at different year for the two species *Campanulla pulla* (top) and *Diathus alpinus* (bottom). Projections for future climate start in 2011 and finish in 2090. Therefore, distributions in 2010 and 2090 correspond to those prior and after (projected) climate change. Illustration for a single replicate for adult survival $s_a = 0.7$, mutation rate $u = 0.0001$, climate change scenario RCP 2.6. Color scales are on the right represent number of individuals per cell/patch.

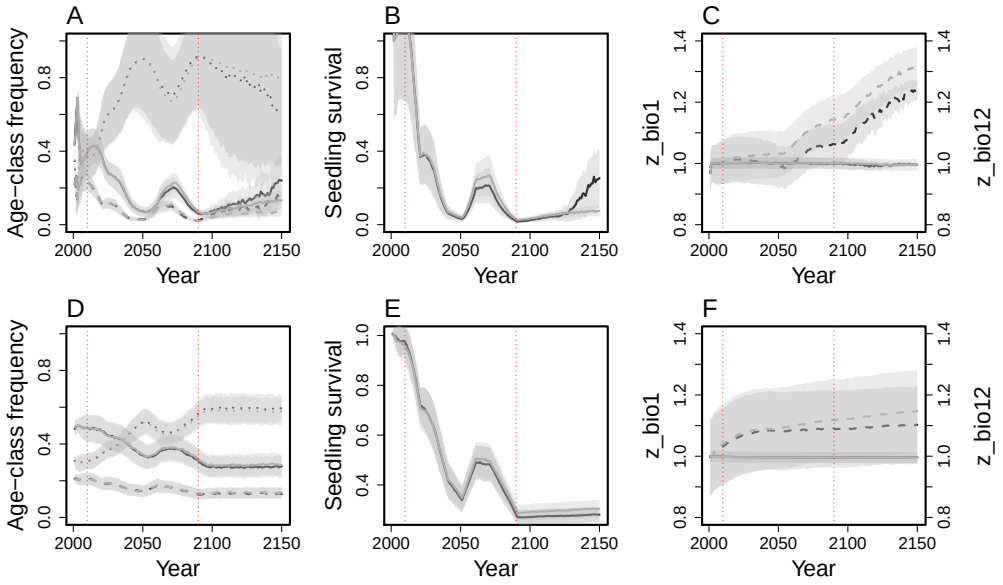

Figure 6.5 Age structure (A, D), seedling survival (B, E) and quantitative trait value (C, F) as a function of year for *Campanula pulla* (A, B, C) and *Dianthus alpinus* (D, E, F). Panels A and D, dotted lines: seedlings; dashed lines: pre-adults; full lines: adults. Panels C and F, dashed lines: z_bio1 (mean annual temperature); full lines z_bio12 (annual amount of precipitation). Projections for future climate start in 2011 and finish in 2090 (vertical red dotted lines). Illustration for a single replicate for adult survival $s_a = 0.7$, mutation rate $u = 0.0001$ (black) and $u = 0.0010$ (gray), climate change scenario RCP 2.6. Lines represent the mean across all populations (occupied cells in the landscape), and the shaded area represents the standard deviation.

Chapter 7

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 4.8). 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 7.2. 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 !
```

7.1 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.

7.2 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.

7.3 Population

Table 7.1 Population stat options

Stat option	Output name	Description
demography	pop.tot	total number of individuals in the whole population
	a*i*.tot	total number of individuals in stage *i* in the whole population
	a*i*.age	mean age of the individuals in stage *i*
	a*i*.fem.patch	mean number of females in stage *i* per extant patch
	a*i*.mal.patch	mean number of males in stage *i* per extant patch
extrate	extrate	proportion of extinct patches in the population
kinship	off.fsib	mean proportion of full-sib
	off.phsib	mean proportion of paternal half-sib
	off.mhsib	mean proportion of maternal half-sib
	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

Table 7.1 continued on next page

Stat option	Output name	Description
	`residents`	mean number of residents per patch (only stage 0-offspring)
	`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 (only stage 0-offspring)
	`imrate.p`i	effective immigration rate into patch i computed as $(\frac{immigrants}{immigrants+residents})$
	`colo.p`i	number of colonizers of patch i
`pop`		same as "demography", "off/adlt.sexratio", and "extrate" together
`pop.patch`	`off./a`i`.fem.p`j	number of females in patch j and stage 0 (`off`) or $i > 0$ (`adlt`)
	`off./a`i`.mal.p`j	number of males in patch j and stage 0 (`off`) or $i > 0$ (`adlt`)
`off/adlt.fem.patch`	`off./a`i`.fem.p`j	number of females in patch j and stage 0 (`off`) or $i > 0$ (`adlt`)
`off/adlt.mal.patch`	`off./a`i`.mal.p`j	number of males in patch j and stage 0 (`off`) or $i > 0$ (`adlt`)
`adlt.sexratio`	`adlt.sexratio`	ratio of the total number of females over the total number of males in the wole population and all stages
`off.sexratio`	`off.sexratio`	same as for the adult sex ratio for the offspring in stage 0

Table 7.1 continued

7.4 Neutral markers

Table 7.2 Neutral markers stat options.

Stat option	Output name	Description
Note: More details about the stats are given in section 5.2.		
[adlt./off.] ntrl.freq	*age*.ntrl.l*i*.a*j*	frequency of allele j at locus i in the whole population for stage 0 (off) or all stages > 0 (adlt)
	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.]fstatWC	*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)

7.5 Quantitative traits

Table 7.3 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)
	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)

Table 7.3: Quantitative traits continued on next page

Stat option	Output name	Description
[adlt./off.] quanti.covar.patch	*age*.cov.qij.pk	genetic covariance between trait i and j in patch k
[adlt./off.] quanti.eigen.patch	*age*.qeval$i$.pj	eigenvalues of the G-matrix in patch j (genetic covariance matrix)
	age.qevectij.pk	loadings of trait j on eigenvector i of the G-matrix in patch k
[adlt./off.] quanti.eigenvalues.patch	*age*.qeval$i$.pj	eigenvalues of the G-matrix patch j
[adlt./off.] quanti.eigenvect1.patch	*age*.qevect1i.pj	loadings of trait i on the first eigenvector of the G-matrix in patch j
[adlt./off.] quanti.skew.patch	*age*.Sk.qi.pj	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

Table 7.3: Quantitative traits stat options continued

7.6 Deleterious mutations

Table 7.4 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 within patch number of fixed mutation
	age.delsegr	mean number of segregating mutation in the whole population
	age.delsegrp	mean within patch number of segregating mutation

Table 7.4: Deleterious mutations continued on next page

Stat option	Output name	Description
	age.delfst	Fst of the deleterious mutations
[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

Table 7.4: Deleterious mutations stat options continued

7.7 Selection

Table 7.5 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

Table 7.5: Selection stat options continued

Bibliography

Barfield, M., Holt, R. D., and Gomulkiewicz, R. C. M. (2011). Evolution in stage-structured populations. *Am. Nat.*, 177(4):397–409.

Berryman, A. A. (1973). Population dynamics of the fir engraver, Scolytus ventralis (Coleoptera: Scolytidae): I. analysis of population behavior and survival from 1964 to 1971. *The Canadian Entomologist*, 105(11):1465–1488.

Beverton, R. J. H. and Holt, S. J. (1957). *On the dynamics of exploited fish populations*, volume 11. Springer Science & Business Media.

Caswell, H. (2001). *Matrix Population Models*. Sinauer Associates, Inc., 2nd edition.

Cotto, O., Schmid, M., and Guillaume, F. (2020). Nemo-age: spatially explicit simulations of eco-evolutionary dynamics in stage-structured populations under changing environments. *Methods Ecol Evol*, 11(10):1227–1236.

Cotto, O., Wessely, J., Georges, D., Klonner, G., Schmid, M., Dullinger, S., Thuiller, W., and Guillaume, F. (2017). A dynamic eco-evolutionary model predicts slow response of alpine plants to climate warming. *Nat Commun*, 8:15399.

Dullinger, S., Gattringer, A., Thuiller, W., Moser, D., Zimmermann, N. E., Guisan, A., Willner, W., Plutzar, C., Leitner, M., Mang, T., Caccianiga, M., Dirnbock, T., Ertl, S., Fischer, A., Lenoir, J., Svenning, J.-C., Psomas, A., Schmatz, D. R., Silc, U., Vittoz, P., and Hulber, K. (2012). Extinction debt of high-mountain plants under twenty-first-century climate change. *Nature Clim. Change*, 2(8):619–622.

Gilbert, O. M., Foster, K. R., Mehdiabadi, N. J., Strassmann, J. E., and Queller, D. C. (2007). High relatedness maintains multicellular cooperation in a social amoeba by controlling cheater mutants. *Proc. Natl. Acad. Sci. U. S. A.*, 104(21):8913–8917.

Goudet, J. (1995). Fstat (version 1.2): A computer program to calculate f- statistics. *J. Hered.*, 86:485–486.

Goudet, J. (2005). Hierfstat, a package for r to compute and test hierarchical f-statistics. *Mol Ecol Notes*, 5:184–186.

Hülber, K., Wessely, J., Gattringer, A., Moser, D., Kuttner, M., Essl, F., Leitner, M., Winkler, M., Ertl, S., Willner, W., et al. (2016). Uncertainty in predicting range dynamics of endemic alpine plants under climate warming. *Global change biology*, 22(7):2608–2619.

Matthey-Doret, R. (2020). SimBit: A high performance, flexible and easy-to-use population genetic simulator. *bioRxiv*, page 2020.05.12.086884. Publisher: Cold Spring Harbor Laboratory Section: New Results.

Moss, R. H., Edmonds, J. A., Hibbard, K. A., Manning, M. R., Rose, S. K., Van Vuuren, D. P., Carter, T. R., Emori, S., Kainuma, M., Kram, T., et al. (2010). The next generation of scenarios for climate change research and assessment. *Nature*, 463(7282):747.

Nei, M. and Chesser, R. K. (1983). Estimation of fixation indices and gene diversity. *Ann. Hum. Genet.*, 47:253–259.

Raymond, M. and Rousset, F. (1995). GENEPOP (version 1.2): population genetics software for exact tests and ecumenicism. *J. Heredity*, 86:248–249.

Ricker, W. E. (1954). Stock and recruitment. *Journal of the Fisheries Board of Canada*, 11(5):559–623.

Salguero-Gómez, R., Jones, O. R., Archer, C. R., Bein, C., de Buhr, H., Farack, C., Gottschalk, F., Hartmann, A., Henning, A., Hoppe, G., Römer, G., Ruoff, T., Sommer, V., Wille, J., Voigt, J., Zeh, S., Vieregg, D., Buckley, Y. M., Che-Castaldo, J., Hodgson, D., Scheuerlein, A., Caswell, H., and Vaupel, J. W. (2016). COMADRE: a global data base of animal demography. *Journal of Animal Ecology*, 85(2):371–384.

Thuiller, W., Georges, D., Engler, R., Breiner, F., Georges, M. D., and Thuiller, C. W. (2016). Package 'biomod2'. *Species distribution modeling within an ensemble forecasting framework https://CRAN. R-project. org/package= biomod2*.

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.