N E M O

release 2.3

version 2.3.45

User Manual

February 3, 2016

authors
Frédéric Guillaume
frederic.guillaume@ieu.uzh.ch
Jacques Rougemont (MPI version)
jacques.rougemont@isb-sib.ch

contributors
Samuel Neuenschwander
Alistair Blachford
Sam Yeaman

availability
http://sourceforge.net/projects/nemo2

© 2006 – 2016 Frédéric Guillaume

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies. Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided also that the sections entitled Copying and GNU General Public License are included exactly as in the original, and provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one. Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by the Free Software Foundation.

Contents

1 INTRODUCTION **1**

1.1 Overview . . . 1

1.2 Main Features . . . 3

1.2.1 Population models . . . 5

1.2.2 The Individual . . . 5

1.2.3 Genetics . . . 6

1.2.4 Statistics and outputs . . . 7

1.3 Using Nemo . . . 7

1.3.1 Launching Nemo from the command line . . . 8

1.3.1.1 For Linux and Mac OS X users . . . 8

1.3.1.2 For Windows users . . . 9

1.3.2 Batch mode . . . 10

2 THE INIT FILE **12**

2.1 Parameter types . . . 12

2.2 Special characters . . . 13

2.3 Matrix parameters . . . 14

2.4 Sequential parameters . . . 14

2.5 External argument files . . . 17

2.6 Temporal arguments . . . 18

2.7 Output files and naming conventions . . . 19

3 SIMULATION COMPONENTS **21**

3.1 Simulation . . . 22

3.2 Population . . . 23
3.2.1 Loading a population from a file . . . 25

4 LIFE CYCLE EVENTS 30
4.1 Aging . . . 32
4.2 Breeding . . . 32
4.3 Breeding with Wolbachia . . . 34
4.4 Dispersal . . . 35
4.5 Seed dispersal . . . 38
4.6 Evolving Dispersal . . . 38
4.7 Selection . . . 39
4.7.1 Multi-trait selection . . . 40
4.7.2 Fixed selection model parameters . . . 41
4.7.3 Gaussian and quadratic model parameters . . . 41
4.8 Extinction and Harvesting . . . 43
4.9 Trait initialization . . . 44
4.9.1 Initialization of trait quant . . . 44
4.9.2 Initialization of trait ntrl . . . 45
4.9.3 Initialization of trait dmi . . . 45
4.10 Resize Population . . . 47
4.11 Cross Design (NCI) . . . 50
4.12 Population Regulation . . . 51
4.13 Save Stats . . . 51
4.14 Saving Files . . . 53
4.15 Store Data in Binary Files . . . 53
4.16 Composite LCE . . . 54
4.17 Breed with selection . . . 55
4.18 Breed-disperse (gametic migration) . . . 56
4.19 Breed with selection and backward migration . . . 58

5 TRAITS 60
5.1 The Genetic map . . . 60

5.2 Neutral markers . . . 63
5.3 Quantitative traits . . . 68
5.4 Deleterious mutations . . . 71
5.5 Dobzhansky-Muller Incompatibility loci . . . 74
5.6 Dispersal genes . . . 76
5.7 Wolbachia . . . 78

6 EXAMPLES **79**
6.1 Life cycles . . . 79
6.1.1 A basic life cycle . . . 79
6.1.2 Adding outputs . . . 80
6.2 Traits . . . 80
6.3 A complete example . . . 81

7 OUTPUT STATISTICS **84**
7.1 Stat Output Files . . . 84
7.2 Stat Options . . . 85
7.3 Population . . . 86
7.4 Neutral markers . . . 88
7.5 Quantitative traits . . . 90
7.6 Deleterious mutations . . . 92
7.7 Dobzhansky-Muller Incompatibilities (DMI) . . . 94
7.8 Selection . . . 94
7.9 Dispersal . . . 95
7.10 Wolbachia . . . 95

Chapter 1

Introduction

1.1 Overview

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

Availability: Nemo comes free of charges and is distributed under the GNU General Public License (GPL2). Binaries and source code are provided for the Linux, MacOSX, and Windows platforms. Nemo is coded in C++ and runs on any platform supporting a console-like environment and allowing it to be compiled with standard C/C++ compilers (GNU gcc being the default).

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

the documentation provided with the source package for instructions concerning the compiling process.

Using: The basic users' interface is a text file (a.k.a the 'init file') containing the input parameters and their argument in a key/value scheme. Nemo is then launched from the console 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 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 directions and chapter 3 for parameters description.

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

Acknowledgments: The parallel computing version (Nemo_MPI) has been developed in collaboration with Dr. Jacques Rougemont at the Swiss Institute of Bioinformatics using the Message Passing Interface (MPI) standard (`http://www.mpi-forum.org`) allowing to run simulations on cluster environments such as the Vital-IT cluster at SIB (`http://www.vital-it.ch`). That parallel version uses the Scalable Parallel Random Number Generators library (SPRNG; `http://sprng.cs.fsu.edu`) as a source of random numbers. The regular Nemo version implements a random number generator (i.e. the Mersene Twister) provided by the GNU Scientific Library (GSL; `http://www.gnu.org/software/gsl`) as well as several other mathematical routines defined in that library. Nemo was initially developed as part of the main author's PhD work at the Department of Ecology and Evolution at the University of Lausanne (`http://www.unil.ch/dee`). Alistair Blachford provided a first version of the bitwise recombination algorithm. Sam Yeaman helped with proofreading and debugging.

1.2 Main Features

Nemo is a forward-time simulation program. This means that the population state is evolved forward in time from generation 0 to generation T through successive (generational) iterations of the life cycle. The life cycle is also composed of a succession of events, chosen by the user. The individuals in the simulated population are run through this life cycle. They will do so only once during their lifetime as the kind of organism modeled so far is semelparous (i.e. reproduce only once and then die like pacific salmons for e.g.). The fate of an individual may depend on its traits value or phenotype. For instance, during viability selection, an individual will survive only if its viability trait (e.g., deleterious mutations) gives it a chance to win the viability lottery.

Nemo allows the updating of parameter values during a simulation by using temporal parameter arguments (see section 2.6). The population state (i.e. number and size of the patches) or any other model component can be changed through time. Patches can also be merged or split (see the resize life cycle event) to model population fusions/fissions.

Nemo offers many different kind of life cycle events (see below) that allow the user to set up many different population/evolution models or simply interact with the simulation data. For instance, Nemo can load simulation data in various format (see subsection 3.2.1) to start a new simulation or just perform some extra genetic analysis. Nemo can also use genetic markers data to seed a simulation. It is thus possible to run simulations based on real field/experimental data. The number of traits an individual can carry is also up to the user. Individuals without any trait can be used to simulate simple demographic models. The number of Life Cycle Events (LCE) composing the life cycle are only limited by their availability. These simulation components are added following the needs of the users/developers of the Nemo framework and we hope their number will increase with future versions. So far, the currently available components are as listed below:

- **Life Cycle Events** (LCE):
  - breeding (reproduction of dioecious or monoecious individuals)
  - viability selection (trait- and environment-dependent fitness values)
  - dispersal (forward and backward migration)
  - combinations of those LCEs (see section 4.16)
  - ageing (non-overlapping generation)
  - population regulation (ceiling model)
  - population growth (logistic, exponential, etc.) (section 4.18)
  - population extinction and harvesting (can be patch specific)

  - population modification (fission, fusion, addition of patches)
  - crossing design with half-sif/full-sib design (NCI)
  - and more...

- **Mating systems**: the breeding LCE allows for the following mating systems:
  - random mating (promiscuity)
  - polygyny (number of mating males can vary)
  - monogamy
  - hermaphroditism (monoecious organisms)
  - selfing (fusion of self-gametes)
  - cloning (no meiosis, suppresses recombination)

- **Dispersal models**: forward (zygotic) and backward (gametic) migration can be modelled with the following dispersal models:
  - sex-specific dispersal matrices fully describing any complex dispersal patterns as defined by the user
  - separate seed and pollen dispersal matrices (for monoecious organisms)
  - large migration matrices for simulations on large geographical grids can be simplified and passed as reduced dispersal and connectivity matrices

  or pre-defined dispersal models with:
  - 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, set as a torus or with reflective or absorbing borders

- **Traits**:
  - Universally deleterious mutations (di-allelelic mutations affecting fitness)
  - Neutral markers (from SNPs to microsatellites)
  - Pleiotropic quantitative trait loci (multiple correlated phenotypic traits)
  - Bateson-Dobzhansky-Muller incompatibility loci (pairs of epistatic loci)
  - Dispersal quantitative loci (male and female specific dispersal genes)
  - *Wolbachia* (endosymbiotic parasite causing cytoplasmic incompatibility)

1.2.1 Population models

Besides its flexibility in the types and number of components included in a simulation, Nemo provides a highly versatile population model. It ranges from the classical island model with evenly distributed patch sizes and dispersal rates to a spatially explicit population model with different sex-specific and patch-specific sizes and dispersal rates. This flexibility is achieved thanks to the matrix parameters that the user can pass to Nemo and which allow to design any kind of population model (see chapter 3 for more details). Extrinsic population extinction rate can also be added to model extinction/recolonization dynamics as well as stochastic variation of population sizes (i.e., harvesting). Furthermore, as the model is fully stochastic, patch sizes may vary during a simulation as a result of pure demographic stochasticity, up to the point of population extinction. Here, the mean female fecundity is key to set the level of population saturation and demographic stochasticity. The population regulation mechanism uses a ceiling model when migration is forward (as in the disperse LCE). That is, the total number of individuals present in a population at time of regulation is reduced to its carrying capacity for each sex. Specific growth rates can be used when backward migration is modeled with the breed_disperse. Population bottlenecks and other variation of the population model may also be modeled with temporal parameter values or the use of the resize LCE. In summary, Nemo allows for the following population features:

- patch-specific and sex-specific population sizes (patch sizes matrix)
- explicit pairwise dispersal rates, also sex-specific (dispersal matrix)
- demographic stochasticity (built-in)
- extrinsic extinction rate or patch size variation (harvesting)
- temporal change of the population parameters and/or dispersal rates
- pure Wright-Fisher population model with constant population sizes (with the breed_disperse LCE)

1.2.2 The Individual

An individual in Nemo is basically defined as a trait container. That means that the phenotypes of the individuals depend on which traits are modeled based on the parameters in the input file. By default, individuals don't carry any genetic information in absence of traits. The only pre-defined phenotypes are the individual's age and sex. Individuals also store information about their ancestry and demographics and have a unique ID and a pedigree class (informs if the two parents were a single individual, full-sib, half-sib, or unrelated individuals). Each individual also stores the

number of babies it had and the ID number of its mum, dad, and natal patch. These information tags are used to compute pedigree-based or age-/sex-specific statistics and are sometimes saved to file by the different simulation components.

1.2.3 Genetics

The genetic models implemented depend on the type of traits, but all types of loci can be placed on the same genetic map. That map is a recombination map, and not a physical map, in that it specifies the locus positions in units of recombination, the centimorgan [cM]. This means that bi-allelic neutral sites can be placed on the same map as QTL and sites under background selection (the deleterious mutations), where loci with the same map position will be physically linked. The traits available in Nemo are distinguished by the interpretation of their phenotypic value and their genetic architecture. At the coding level, this means that different data structures can be mixed together making trait implementation highly flexible and dependent on the specific need for the different traits. For instance, the neutral (ntrl) trait has no phenotypic value and is coded on one byte per locus, the deleterious (delet) trait has fitness as its phenotype and is coded on one bit per locus along with a single table relating mutation to their fitness value, while the quantitative (quant) trait has a continuous value as its phenotype coded on one double precision number (8 bytes) per locus per trait (all loci are pleiotropic when more than one trait is modeled). The data structure chosen obviously conditions the number of alleles available per locus, hence the use of bits for a bi-allelic trait like deleterious mutations, and the use of a single byte for neutral markers that can have from two (SNP), to four (nucleotide), to many alleles but not an indefinite number of alleles (maximum is 256 allelic states).

Simulation of large DNA sequences

Nemo has not been developed to model evolution of genetic polymorphisms of DNA sequences on large chromosomal regions spanning several million base pairs at the nucleotide level. The reason is that Nemo uses an explicit representation of each locus in each individual. This straightforward implementation is fine when modelling limited number of loci in the range of 100 to 10,000 loci, especially for neutral markers. The implementation capitalises on position-ordered arrays of loci, which makes access to locus values an efficient, constant-time operation. This is of particular interest for non-neutral traits which individual values must be read in each individual at each generation to determine fitness. The approach, however, has a huge computational cost when modeling large sequences of over 10^5 loci (in large populations) because of the intense memory usage it entails.

1.2.4 Statistics and outputs

Nemo provides several ways to record the ancestral population states. Summary statistics can be computed at different time periods during 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 among to monitor during a simulation. Here are examples of the summary statistics:

- **Neutral trait stats**: 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.
- **Deleterious mutations stats**: mutation frequency, heterozygosity, homozygosity, genetic load, heterosis, number of lethal equivalents, viability by pedigree classes, etc.
- **Dispersal trait stats**: mean male and female dispersal rates
- **Population stats**: patch saturation, female and male number per patch, sex-ratio, mean fecundity, variance of reproductive output, count of migrants, effective extinction rate, etc.

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

Alternatively, you can save the raw data of the ancestral population in either binary or text file formats. The various traits usually provide a way of saving the population genotypes in text files. A special binary file format is used to save the whole population information containing all the traits and individuals data and the simulation parameters. Binary files can then be used by Nemo to load a saved population and run a new simulation from it or use it as a source of individuals that have, for instance, reached a certain level of genetic stability (i.e. burn-in population).

1.3 Using Nemo

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

1.3.1 Launching Nemo from the command line

1.3.1.1 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 launch it. Then, whatever your operating system is, we assume you have installed the executable file `nemo2.x.y` in a folder somewhere on your file system and that you set your working directory to that place (using the `cd` command). The following commands will allow you to run a simulation.

First, lets have a glance at what is in the directory using the `ls` command:

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

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

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

The '`./`' characters in front of the executable filename simply means the program file is to be searched in the local directory rather than in one of the directories specified by the PATH environment variable. This command will produce the following output to your terminal window (or something approaching depending on the program's version):

```
> ./nemo2.1.0

  N E M O 2.1.0  [22 Jan 2009]

  Copyright (C) 2006-2009 Frederic Guillaume
  This is free software; see the source for copying
  conditions. There is NO warranty; not even for
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
  http://nemo2.sourceforge.net
------------------------------------------------
reading "Nemo2.ini"
setting random seed from input value: 213145234

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

```
    start: 23-01-2009 11:33:27
    mode: overwrite
    traits: delet, fdisp, mdisp, ntrl
    LCEs: breed_selection(1), store(2), save_stats(3), save_files(4)\
 , disperse_evoldisp(5), aging(6), extinction(7),
    outputs: test/{*.log, delet/*.del, fstat/*.dat, fstat/*.fsti, \
 binary/*.bin, data/*.txt}

    replicate 10/10 [11:34:32] 3000/3000

    end: 23-01-2009 11:34:40
--- done (CPU time: 00:01:11s)
setting random seed from input value: 213145234

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

    start: 23-01-2009 11:34:40
[...]
```

This output shows the progress of the simulation with the replicate and generation counters and prints the time when the current replicate started (in format hh:mm:ss) and ended, and the elapsed computing time (hh:mm:ss). This simulation was run on a MacBook 2.4 GHz Intel Core 2 Duo. The parameter file used in this example is the one present in the `example` directory of the distribution package.

1.3.1.2 For Windows users

You have two options to run Nemo under Windows. You may install CygWin (from http://www.cygwin.com), as Nemo has been compiled using this environment, this is the better option. Or, you can simply use the MS-DOS terminal (i.e. the command prompt). The latest option is explained here as using CygWin is like using any *nix environment (see previous section). So, launch the command prompt (cmd.exe) and `cd` to where you have installed Nemo. Assuming you have the following files in your current working directory after downloading the right archive (i.e. Nemo-*x.y.z*-Win-binaries.zip):

```
nemo2.x.y.exe cygwin1.dll Nemo2.ini
```

The cygwin1.dll file is required to run Nemo outside of CygWin and must be sitting in the same directory as the `nemo` executable. To launch Nemo, simply type the command:

```
> nemo2.x.y.exe
```

You should have the same output as previously under MacOS X / Linux.

Note about CygWin: when installing CygWin, check that you also install the GSL library by checking the `gsl:Runtime` option under the `Libs` section of the installer.

1.3.2 Batch mode

Nemo accepts only one type of argument on the command line, the name(s) of the init file(s) to run simulations from. For instance, if three init files are passed to Nemo, the program will initiate three simulations from those files, considering they don't incorporate any sequential parameters. Sequential parameters are parameters with more than one argument value (see section 2.4 below).

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

```
> ./nemo2.0 sim1.ini sim2.txt sim3
```

Here we have three init files called `sim1.ini`, `sim2.txt` and `sim3`, they are all text files, the extensions do not matter much here. Their parameters are the same as in the previous example. This command will produce the following output:

```
  N E M O 2.0.0  [25 Apr 2006]
[...]
------------------------------------------------
reading "sim1.ini"
reading "sim2.txt"
reading "sim3"

--- SIMULATION 1/3 -----------------------------

     replicate 10/10 [10:04:54] 100/100
--- done (CPU time: 00:01:26s)

--- SIMULATION 2/3 -----------------------------

     replicate 10/10 [10:06:36] 100/100
--- done (CPU time: 00:01:26s)
```

```
--- SIMULATION 3/3 ----------------------------

     replicate 10/10 [10:08:13] 100/100
--- done (CPU time: 00:01:26s)
```

Sequential parameters As an example of sequential parameters, let's assume 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:

```
> ./nemo2.0 sim1.ini sim2.txt sim3
[...]
reading "sim1.ini"
reading "sim2.txt"
reading "sim3"

--- SIMULATION 1/5 ----------------------------

     replicate 1/10 [10:19:22] 88/100 -> Pop extinction !
     replicate 3/10 [10:19:25] 74/100 -> Pop extinction !
     replicate 4/10 [10:19:26] 84/100 -> Pop extinction !
     replicate 7/10 [10:19:29] 97/100 -> Pop extinction !
     replicate 10/10 [10:19:33] 100/100
--- done (CPU time: 00:00:11s)

--- SIMULATION 2/5 ----------------------------

     replicate 10/10 [10:21:00] 100/100
--- done (CPU time: 00:01:26s)

--- SIMULATION 3/5 ----------------------------

     replicate 10/10 [10:24:00] 100/100
--- done (CPU time: 00:02:55s)

--- SIMULATION 4/5 ----------------------------
[...]
```

Chapter 2

The input parameters file

The configuration file (or init file) presented previously, 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 `'#'` is treated as a comment and is ignored. Parameters are character strings (with no whitespace character) that may be followed by one to several argument values separated by at least one white space character. 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); this is especially useful for temporal arguments (see below).
- **integer** : argument is a dot-less number value; a limit to the number of available values a parameter can take may be specified from case to case.
- **decimal** : argument may be a floating-point value. The following forms are equivalent: 0.0001, .0001 or 1e-4.
- **string** : argument is a character string that may 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 2.3 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** : {{*row1*}{...}} : any argument value starting and ending by two enclosing curly braces is considered as a matrix argument (see next section).
- **name expansion** : `%` : used in the character string of an argument to insert the value of another parameter when that parameter has multiple argument values (see sequential parameters in section 2.4).
- **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 2.5).
- **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 2.6). 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 2.6). Argument values are separated by commas within a block argument (e.g., `(@g0 0.02, @g5 0.5)`).

2.3 Matrix parameters

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

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

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

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

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

2.4 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.

2.5 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 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:

```
param0 1 2 3
param1 @filename1.txt @filename2.txt @filename3.txt
param2 @path-%1/to/filename-%'1[abc]'2.txt
```

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

2.6 Temporal arguments

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

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

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

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

```
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 patch to 0 before adding them at a latter generation by increasing their carrying capacity.

2.7 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 as above. 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 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 4.13). 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 4.15 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 the file manager of Nemo notifies the different simulation components at time of saving, you *must* include the `save_files` life cycle event (see section 4.14) in the life cycle, otherwise no files will be written for 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. Besides these two, it would make sense to select at least a basic sequence of life cycle events to run a basic simulation. Note that you can also use Nemo to simply load a previously saved population from a binary file (see the source_pop population parameter below) and compute statistics on it or extract genotypes and save them in a human-understandable format (usually text...).

Each component and its list of parameters are presented here. Some parameters are mandatory; they must be present in the init file in order to include a component to a simulation. Each component has at least one mandatory parameter. Optional parameters are marked as **(opt)** below and are used to add extra features needed to build a particular model. Nemo will not complain if a mandatory parameter is missing for a non-mandatory component (i.e., others than the simulation and population components) so you have to be careful while building the init file. The parameter type is given between two enclosing square-brackets '`[ ]`', see chapter 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, as defined by their traits' state. The action of an LCE may depend on the values of the individual's traits or not. For instance, selection will remove individuals by checking the phenotype of their fitness trait against a fitness function, or aging will remove all adult individuals independently of their traits' value to make room for the new generation.

The simulation components can also declare different output files and statistics. The file extensions and stat outputs are indicated for each component. For a discussion and a complete list of output statistics, have a look at chapter 7.

3.1 Simulation

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

replicates [integer]

Number of replicates to perform per simulation.

generations [integer]

Number of generations performed per replicate.

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 2.7).

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

logfile [string] (opt)

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

random_seed [integer] (opt)

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

postexec_script [string] (opt)

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

```
sh my_script.sh
```

postexec_args [string] (opt) This parameter is used to add an argument to the above script when executing it. Be aware that the expansion character '%' will not be expanded if present in the argument string and should thus be avoided.

3.2 Population

name: **population**
files: NA
stats: `pop`, `demography`, `migrants`, `kinship`, and more (see chapter 7)

patch_number [integer] (opt)

Number of patches in the population.

patch_capacity [integer/matrix] (opt)

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

patch_nbfem/patch_nbmal [integer/matrix] (opt)

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

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

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

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

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

This other example will also work fine:

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

Note however that the following will issue a fatal error:

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

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

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

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

3.2.1 Loading a population from a file

This section describes the set of parameters needed to load/read a population from a file. The type of data that can be loaded depends on the file format. The binary files, written by the store component (see 4.15), store the whole population state, that is, all individuals in the population are saved with their attributes and traits data (i.e. genetic data). Other simulation components may define an input function for the type of data they handle. Typically, the neutral markers trait (section 5.2) saves and loads its data in the FSTAT format (text) and the deleterious mutation trait (section 5.4) saves and loads from a text file (see respective trait's description for details about those files).

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 age class (offspring or adults, or both) that is used when loading a population depends on the one available in the source file and the one that is required by the life cycle events of the current simulation. The class to load is determined by finding the first event in the current life cycle that requires a specific age class (see chapter 4 on life cycle events). Nemo then tries to load that class from the source file. Independently of the loading mode, if that required age class is not available in the source population, the alternate one is used instead (i.e. offspring for adults, and vice versa). A warning message is displayed if that case happens.

Using compressed binary files: Finally, when loading populations from binary files, Nemo will automatically check whether the binary fill is compressed. If so, Nemo will decompress 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 description:

source_pop [string] (opt)

The path to the population file is given by this parameter. The path/filename of the source population may contain the special expansion character and format string (see chapter 2) to match the sequential parameter arguments values defined in the current configuration file. The replicate counter string may be automatically added when multiple replicate sources are used (see below). The file extension may also be specified below in the case the file format differs from the default binary one (i.e. ".bin"). Only one file can be used for a given simulation or replicate.

If every replicate of the current simulation is going to use the same binary source file as specified here, the full name of the file must be specified, i.e. including counters and extension strings. If the source population is to change during a simulation, the parameter source_replicates must be specified (see below). In this case, the path given here must not be terminated by the usual replicate counter string and any file extension.

source_file_type [string] (opt)

The argument here is the file extension string of the source file, including the dot (e.g. ".bin", ".dat", etc.). This will determine the file format of the source data. The default value is ".bin".

source_preserve [bool] (opt)

With this parameter, the deme structure of the source population is preserved. This means that Nemo will copy individuals from the source population into the current population deme by deme. If the source population has less demes than the receiving population, then the receiving population will have empty demes. Similarly, in 'preserve' mode, the receiving population will not be full if the source population does not contain enough individuals within demes to fill the receiving demes to their carrying capacity. The receiving population is filled sequentially, starting from the first patch to the last. No extra demes are added to match the number of demes in the source population. The population structure will be perfectly preserved if the source and the target population have same deme structure and sizes.

If not present, then, the individuals are randomly sampled from the source population, without replacement. In that case, the individuals of the source population are gathered together in a single container from which they are sampled until the whole receiving population is full or the source population is empty.

source_fill_age_class [adults, offspring] (opt)

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

source_generation [integer] (opt)

The generation to load from the binary source file. The population initialization will fail if that generation is not present in the binary file. The binary files may indeed store more than one generation (see section 4.15).

source_replicates [integer] (opt)

By specifying this parameter, you can tell Nemo how many replicates of the source population have to be used throughout the simulation to load the population from. If the value given here matches the number of replicates of the current simulation (see replicates above), each replicate will use a different source file as a source population. In the case this value is smaller than the current number of replicates, 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 the last generation saved in 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 copied by each replicate of that simulation.

Now, if we want to change that behavior and use a different source population for each replicate, we must specify the following set of parameters:

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

In that second example, each replicate loads a different population: `mysourcepop_001.bin` for replicate 1, `mysourcepop_002.bin` for replicate 2, etc.

If the simulation to run has a hundred replicates and we keep the same set of parameters for the source, the source population will be changed every four replicates only, starting from replicate 25. Replicates 1 to 4 will use data from the population in `mysourcepop_025.bin`, replicates 5 to 9 will use `mysourcepop_026.bin`, and so on until file `mysourcepop_049.bin`.

```
replicates 100
source_pop binarydir/mysourcepop
source_preserve
source_replicates 25
source_replicate_digit 3
source_start_at_replicate 25
```

Finally, loading a population from a trait file is also possible. This can be done from a single or different files, depending on the type of data. The simulation parameters should match the data structure in the source file for optimality. The following example loads neutral markers data (e.g. from a field study) from a single FSTAT file (see section 5.2 for more details) and use it to compute the F-statistics available in Nemo:

```
replicates 1
generations 1

patch_number 5
patch_capacity 50

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

## LIFE CYCLE ##
save_stats 1
```

```
save_files 2

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

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

Chapter 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 the course of a generation, at the rank it has been assigned in the stack of LCEs that constitutes the life cycle. This rank is given by the user in the init file. The life cycle is thus an ordered list of LCEs selected by the user. Most LCEs act on a per generation basis. Some may however have a different periodicity set by the parameters they declare.

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

The way to build the life cycle in the init file is to write the LCEs names (given below) followed by their rank number. Here is an example (see chapter 6 for more details):

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

This very simple life cycle starts with mating and breeding within the population that will generate a new offspring generation provided adults are present within patches. The statistics are then recorded and the simulation data is saved, at the

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

The Life Cycle Events described here are:

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

breed: mate and breed, create new offspring generation

breed_wolbachia: breed and *Wolbachia* transmission/infection

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

breed_selection: breed with selection (faster)

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

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

disperse: offspring dispersal

disperse_evoldisp: offspring dispersal with evolving dispersal rates

extinction: random patch extinction or harvesting

regulation: patch regulation (to carrying capacity)

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

save_files: write output files to disk

save_stats: record statistics

selection: perform viability selection on the offspring generation

store: save simulation data to binary files

The LCEs often act as modifiers of the population state. Most of the time, this simply consists of changing the content of various individual containers either by moving individuals between them or by adding/removing individuals to/from them. Individual containers are ordered by age class and by sex and are aggregated within patches. The two main age classes are the adult and the offspring age classes. A particular LCE will in general be associated with one or more age class. This information is given below by the age flag values associated with each LCE (see Table 4.1). These age flags tell which individual container will actually contain individuals after having executed the corresponding LCE during the life cycle and which age class is needed by an LCE. This will help you design a proper life cycle.

Table 4.1 Modification of the population age state caused by the LCEs in the basic life cycle. (+) means that age class is added to the population by the LCE while (−) means the LCE will remove all individuals of that age from the population. (x) means the LCE will modify the state of that age class. *required* means that age class is the required age class for the LCE, and will be loaded first whenever that LCE begins the life cycle.

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

4.1 Aging

name: **aging** [**integer**]
age flags: removes the **offspring** flag
files: NA

aging moves *all* individuals from their age class to the next and performs patch regulation at the same time. For now, only two age classes are present, the offspring and the adults. Therefore, `aging` moves the offspring to the adults age class and all the adults are removed, they die. No other LCE removes the adults from the population. It is thus very important to add this LCE to the life cycle. For each patch, the offspring individuals are randomly chosen to fill the adult containers until the patch carrying capacity is reached. **Note:** since the behaviour of this LCE has changed in version 2.0.7, be careful about its position in the life cycle. If placed before `disperse`, no offspring will be able to migrate in the population as they already aged. The regulation event is not useful anymore after aging but is still proposed, in a slightly different flavour (see below).

4.2 Breeding

name: **breed** [**integer**]
age flags: **adults** (required) and **offspring** (added)

files: NA
derived components: breed_wolbachia, breed_disperse, breed_selection, breed_selection_disperse

Performs mating and breeding of the new offspring generation following the mating system chosen. Adults are not removed here (see `aging` above). The number of offspring per female depends on the mean fecundity set by `mean_fecundity` below and may be a fixed number or a number drawn from different random distributions. The default distribution is Poisson.

mating_system [1 to 6]

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

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

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

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

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

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

6 : random mating with selfing This corresponds to what is called the Wright-Fisher model where individuals may self with probability 1/N (N = patch size). The individuals are considered hermaphrodites here, that is *only the females are used* (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 Breeding with Wolbachia

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

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

wolbachia_fecundity_cost [decimal]

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

wolbachia_incompatibility_cost [decimal]

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

wolbachia_inoculum_size [integer]

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

wolbachia_inoculum_time [integer]

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

4.4 Dispersal

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

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.

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.) ***ii)*** by specifying the full **migration matrix**, allowing for more flexibility in the type of migration modelled (e.g., allowing for long-distance dispersal on a landscape), ***iii) (new in 2.3)*** 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 for modelling large grids with limited dispersal among patches, and brings a large speed-up compared to the previous implementations. All migration matrices are now reduced internally.

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

The dispersal models implemented so far are:

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

2 : Propagule-pool Island model. In that modified version of the Island Model, 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 squared grid (or lattice) and dispersers can move to at least four adjacent patches (set by the dispersal_lattice_range parameter below). This option must be followed by the dispersal_lattice_model and dispersal_lattice_range parameters. The number of patches in the population must be a square number.

The dispersal_model parameter may be omitted when providing the dispersal matrix (or reduced matrix).

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_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.

dispersal_propagule_prob [decimal] (opt)

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

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 i to patch j. This parameter has precedence over the dispersal rate and model parameters. If too big, and especially when containing a large number of zeros, 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 the precedence as above.

dispersal_rate [decimal] (opt)

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

dispersal_rate_fem / _mal [decimal] (opt)

Replaces the previous parameter for the case of different males and females dispersal capabilities.

dispersal_reduced_matrix [matrix] (opt)

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

dispersal_connectivity_matrix [matrix] (opt)

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

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

4.5 Seed dispersal

name: **seed_disperse [integer]**
age flags: **offspring** (required)
files: NA

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

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

4.6 Evolving Dispersal

name: **disperse_evoldisp [integer]**
age flags: **offspring** (required)
files: NA
inherits from: disperse

This is a specialization of the previous LCE and thus inherits its parameters, though the rate parameters have no meaning here. In addition, it defines a couple more parameters used by the evolving dispersal models.

dispersal_cost [decimal]

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

dispersal_cost_fem /_mal [decimal] (opt)

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

dispersal_fixed_trait [female, male] (opt)

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

dispersal_fixed_rate [decimal] (opt)

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

4.7 Selection

name: **viability_selection [integer]**
age flags: **offspring** (required)
files: NA
derived components: breed_selection, breed_selection_disperse

Viability selection selectively removes individuals from a patch based on their survival probability given by their fitness trait. Currently, the fitness determining traits are delet (deleterious mutations), quant (quantitative traits), and dmi (Dobzhansky-Muller incompatibility loci), although any other trait may be used as long as the trait's phenotype is compatible with the fitness models implemented. Fitness can be either *absolute* (i.e., directly set from the individual's phenotype) or *relative* to the mean fitness value of the patch or of the whole population. For now, it only acts on the offspring age-class but can be placed anywhere in the life cycle. Future releases will extend this behaviour to selection on other age classes. This LCE also declares a set of fitness statistics that can be recorded during the simulation (see section 7.2). The parameters described here are the same as those used with the breed_selection and breed_selection_disperse composite LCEs (which inherit those parameters).

New in 2.3: selection can now 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 4.7.1 below.

selection_trait [string]

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

selection_model [fix, direct, gaussian] (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:
$W(\mathbf{z}) = exp\big[-\frac{1}{2}(\mathbf{z}-\boldsymbol{\theta})^T\boldsymbol{\omega}^{-1}(\mathbf{z}-\boldsymbol{\theta})\big]$,
where $\boldsymbol{\theta}$ is a vector of local optimal trait values, and $\boldsymbol{\omega}$ is the variance-covariance matrix of selection describing the individual fitness surface.

quadratic : A quadratic model of stabilising 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 [absolute, relative_local, relative_global] (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 mean fitness of other individuals in its patch (option **relative_local**) or in the whole metapopulation (option **relative_global**).

4.7.1 Multi-trait selection

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)). To specify a model with selection on the delet and quant traits, the following set of parameters would be necessary:

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

The fitness value of an individual is then given by the **product** of the fitness values of each trait.

4.7.2 Fixed selection model parameters

selection_base_fitness [decimal] (opt)

Base fitness of the population (W_0).

selection_lethal_equivalents [decimal] (opt)

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

selection_pedigree_F [matrix] (opt)

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

4.7.3 Gaussian and quadratic model parameters

selection_matrix [matrix] (opt)

This is the selection matrix $\boldsymbol{\omega}$ used to set the strength 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.

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 patch values are provided, the values will be recycled to fill the patch-specific selection matrices. Similarly for the trait values, although here only a single value is accepted (will copy the value to all traits).

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.

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. The population average of patch-specific phenotypic standard-deviations will be used if the parameter is not present in the init file (not set).

4.8 Extinction and Harvesting

name: **extinction [integer]**
age flags: **unchanged**
files: NA

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

extinction_rate [decimal/matrix] (opt)

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

extinction_size [decimal/matrix] (opt)

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

extinction_proportion [decimal/matrix] (opt)

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

extinction_threshold [decimal] (opt)

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

extinction_size_distribution [uniform, poisson, normal, exponential, log-normal] (opt)

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

extinction_size_dist_stdev [decimal] (opt)

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

4.9 Trait initialization

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

4.9.1 Initialization of trait quant

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

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

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 either the same number of traits modelled or one. If only one initial trait value is specified per 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 initialiser 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.

4.9.2 Initialization of trait ntrl

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

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

ntrl_init_patch_freq [matrix] (opt)

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

4.9.3 Initialization of trait dmi

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

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

dmi_init_freq [matrix]

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

Examples with 6 demes and 8 loci:

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

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

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

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

dmi_init_patch [matrix] (opt)

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

Example with 6 demes and 8 loci:

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

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

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

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

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

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

4.10 Resize Population

name: **resize [integer]**
age flags: **unchanged**
files: NA

The resize LCE modifies the state of the meta-population during a simulation but with more control than by using temporal arguments within the population parameters. In particular, it allows the user to merge or split existing patches without losing individuals or adding empty patches, which is what would happen when using temporal parameters.

resize_at_generation [integer/matrix]

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

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

resize_patch_number [integer] (opt)

Specifies the new number of patches in the population.

resize_patch_capacity [integer] (opt)

Specifies the new patch carrying capacity, also accepts a matrix argument as the population parameter (see 3.2).

resize_female_capacity [integer] (opt)

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

resize_male_capacity [integer] (opt)

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

resize_age_class [offspring, adults, all] (opt)

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

resize_do_flush [bool] (opt)

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

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

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

resize_do_fill [bool] (opt)

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

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

resize_do_regulate [bool] (opt)

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

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

resize_keep_patch [matrix] (opt)

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

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

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

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

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

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

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

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

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

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

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

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

4.11 Cross Design (NCI)

name: **cross** [**integer**]
age flags: **adults** (required); **offspring** (add)
files: NA

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

cross_num_sire [**integer**]

Number of sampled males per patch. Each male will be mated with num_dam females as many times as num_offspring.

cross_num_dam [**integer**]

Number of sampled females per sire. Each female produces num_offspring with one given male.

cross_num_offspring [**integer**]

Number of offspring produced per dam.

cross_at_generation [**integer**]

Generation at which crossing is performed.

cross_do_within_pop [**bool**] **(opt)**

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

cross_do_among_pop [**bool**] **(opt)**

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

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

cross_with_replacement [bool] (opt)

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

4.12 Population Regulation

name: **regulation** [**integer**]
age flags: **adults** (required)
files: NA

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

4.13 Save Stats

name: **save_stats** [**integer**]
age flags: **unchanged**
files: `".txt", "_bygen.txt"`

This LCE is used to tell the stat-services of the simulation to record the summary statistics specified with the stat parameters (see below). The statistics recorded depend on the age state of the population. The position of this LCE in the life cycle is thus important. Putting it after breeding will allow you to record stats on both offspring and adults while putting it after aging will allow you to record the stats on the adults only. The recorded stats are dumped to a text file at the end of each replicates and at the end of a simulation for the averaged stats, but only if the save_files LCE is present in the life cycle. See chapter 7 for a description of the different output files declared by this LCE. Note that no results will be saved if none of save_stats or save_files are present in the life cycle.

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

4.14 Saving Files

name: **save_files [integer]**
age flags: **unchanged**
files: varies

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

Some simulation components automatically upload their different file handlers to the file manager. For instance, the save_stat LCE defines two types of automatic output files, one ending with the `".txt"` and the other with the `"_bygen.txt"` extensions (see above and chapter 7) to save the statistics recorded during the simulation. Other component let the user chose what and when data must be saved on disk (see the trait components for e.g.).

4.15 Store Data in Binary Files

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

This LCE provides a way to dump all the traits and individual's 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. More than one generation of one replicate can be saved in one binary file but there always is one file per replicate. By default, binary files are compressed (with bzip2 by default) and put in a "tar" archive. This behaviour can be changed with the parameters described below.

store_dir [string] (opt)

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

store_generation [integer]

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

store_recursive [bool] (opt)

This option will tell the program to use the `store_generation` value as a generation logging time. The binary files will thus contain several generations.

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.

4.16 Composite LCE

Composite life cycle events are LCEs that inherit the properties (parameters) of other LCEs (the base LCEs) and extend, or sometimes, redefine their functionalities. For instance, `breed_selection` inherits the parameters of the `breed` and `viability_selection` LCEs and performs both breeding and viability selection in

one but doesn't add any new parameters. Other composite LCEs may also add new parameters (see below). Because the init file cannot have more than one copy of a parameter, the composite LCE and its base LCEs cannot have different parameters values; they share the exact same parameters. That behavior will change in future releases.

```
breed_selection
breed_disperse
breed_selection_disperse
```

4.17 Breed with selection

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

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

- Fitness is always absolute.
- The realised fecundity of a female or male is set accordingly to the survival of their offspring (allowing the correct computation of the values of the `heterosis`, `load`, and females/males realised fecundities and fecundity variances).
- This LCE may be faster than having `breed` followed by `viability_selection` in the life cycle when more than one trait are simulated, because mutation and recombination are performed on the selected trait before checking for survival. Therefore, mutation and recombination of the traits not under selection are performed on the surviving offspring only.

breed_selection_fecundity_fitness [bool]

If this parameter is set (present in the init file), the selection mode is changed from acting on offspring survival to act on the number of offspring produced by each female. In other words, with this mode, it is the fitness of the female that matters rather than that of the offspring. The mean value of the fecundity distribution is multiplied by each female's fitness when drawing its number of offspring produced. This works best when the mean fecundity is large because

only integer numbers of offspring can be produced, which is problematic when the mean of the Poisson distribution is too low (e.g. a fitness of 0.25 and a mean fecundity of < 4 will cause many more females to have no offspring than if the mean fecundity is 10). By having a too low mean fecundity, one looses precision in the selective process, and selection will be stronger.

4.18 Breed-disperse (gametic migration)

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

Note: **since version 2.3**, 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.

This LCE performs breeding and dispersal in a single step. It inherits the parameters of the breed and disperse LCEs. For an offspring, each parent is randomly taken from the local patch with probability $1 - m$ or from a different patch with probability m, where m is the dispersal rate. The dispersal rates are thus taken as *backward migration* or *immigration* rates in opposition to the *forward* emigration rates of the disperse LCE. This corresponds to the classical Wright-Fisher model if the mating system is hermaphroditism (mating_system 6). By default, exactly K offspring are produced per patch, if K is the patch capacity, unless the patch is extinct and the parameter breed_disperse_colonizers is specified, which limits the number of individuals grown locally from two 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 reads the immigration rates column-wise (element d_{ij} is the probability to get a migrant gamete *from* deme i *into* deme j, i being the row number and j the column number).
- There can be **no demographic stochasticity** (demes always at carrying capacity) if the growth model is set to 1 (instant growth, default value), and breed_disperse_colonizers is unset.
- Deme **extinctions** may cause the **program to hang indefinitely** if immigration into an extinct deme is impossible (e.g., because of source patch extinction or zero immigration set in the dispersal matrix).

- An extinct deme will be **instantly recolonised** (in a single generation) unless the number of immigrants is capped with breed_disperse_colonizers or a growth model is specified.
- Two dispersal matrices can be used for hermaphrodites to **model pollen migration** (i.e., fecundation of local ovules with immigrant pollen, without ovule migration), see breed_disperse_dispersing_sex.
- Mating systems 2 (polygyny) and 3 (monogamy) can not be used here.
- This LCE can be used to mimic the Wright-Fisher model when the mating system is set to 6 (random mating with selfing rate = $\frac{1}{N}$).
- This LCE is much faster than having breed followed by disperse in life cycle because exactly N offspring are produced and not $\frac{N}{2}\bar{f}$, $\bar{f}$ being the females mean fecundity. Usually, $\bar{f}$ should be greater than 2 to avoid too much demographic stochasticity, especially with small patch sizes.

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}.

breed_disperse_growth_rate [decimal] (opt)

The patch growth rate used in the logistic growth model.

4.19 Breed with selection and backward migration

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

This LCE aggregates the features of both previous composite LCEs. However, to perform selection and backward migration with populations of constant sizes, there must be some adjustments in the way selection is performed in the case where the mean fitness is too low to allow the patches to be filled with surviving offspring. The basic idea is therefore to define a minimum fitness threshold for the individuals. If the mean fitness of the adult (breeders) generation is below that threshold before mating, the offspring fitness is rescaled so that the mean patch fitness matches that threshold. In other word, the threshold is the minimum survival probability 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.

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).

Note: **for version 2.3**, since breed_selection_disperse inherits parameter definitions from breed_disperse, the dispersal parameters must also use the breed_disperse prefix instead of dispersal, see section 4.18 above.

Chapter 5

Traits

The traits described here are:

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

Each trait has an identifying name or type and may define different output files and stat options. For a complete description of the stat options, have a look at chapter 7.

5.1 The Genetic map

[New in version 2.3] The three sequence-based traits (*ntrl*, *quant*, and *delet*) share a common genetic map on which the loci of the different traits are placed. The genetic map in Nemo is a recombination map where the locus positions are specified in centi Morgan (cM), in opposition to the base-pair unit (bp) of physical maps. The genetic map may be composed of more than one chromosome, each with a different number of loci (although not always, see options below). 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 the three traits and 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.

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 specified recombination rates 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.

Limitations are that the number of chromosomes can not differ among traits (i.e. chromosomes without loci are not accepted), and the number of loci per chromosome on fixed map must be constant (see below).

***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 option lets one set the positions at equal distance between loci on a given 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 elements of 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 a single value is given and that value is 0.5, the loci are considered as 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 locus numbers 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. This option is not used when fixed maps are specified with *prefix*_genetic_map (see note above).

***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. 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 (i.e., a distance of 1 then corresponds to a recombination rate of 0.1% instead of 1% between two loci). 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.

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 4.9.2).

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 4.9.2 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: age (0 = offspring, 2 = adult), sex (0 = male, 1 = female), home (natal patch), ped (see p66), isMigrant (number of immigrant parents), father, mother, ID; father and mother contain unique ID's of the parents of the individual with identifier ID, for pedigree reconstruction.

- FSTAT (fstat)

  The file format is (almost) the same as that used by the FSTAT program (Goudet 1995). It adds information about each individuals (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 (Reymond & 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 to save the within patch F_{ST} values per-locus using the Weir & Hill (2002) estimates (see note below). Each line of the output text file contains the values of a specific locus and each column is for a different patch. The first line takes the column labels. The file extension is `".fsti"`.

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

ntrl_output_logtime [integer] (opt)

This is the 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 3.2.1, 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 does not include the `age`, `sex`, `ped`, and `origin` "loci". Here is an example of a neutral genotype output file, the file format is inherited from the FSTAT file format (Goudet 1995):

```
5 9 20 2
loc1
loc2
loc3
loc4
loc5
age
sex
ped
origin
1 1414 1019 2002 0820 0307 1 1 1 1
1 0814 0219 2002 2020 0307 1 1 1 1
```

```
1 0808 0217 1902 0820 0907 4 1 2 1
1 0820 0209 1902 0805 0918 4 0 0 4
[...]
4 0307 1308 0220 0401 0115 1 1 1 4
4 0905 1213 0302 0312 0506 4 1 2 2
[...]
5 2017 1010 2013 1812 1505 4 0 1 5
5 2017 1008 2013 1811 1505 4 1 2 3
```

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

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

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

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

Note about the statistics: Nemo lets the user choose between various estimates of gene diversity and genetic differentiation both within and between populations. The classical F-statistics are available by using the `'fstat'` stat option (see section 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.

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

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

$$F_{ST} = \frac{\theta-\alpha}{1-\alpha},\ F_{STii} = \frac{\theta_{ii}-\alpha}{1-\alpha},\ F_{STij} = \frac{\theta_{ij}-\alpha}{1-\alpha},$$

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

References: Goudet, J. 1995. "FSTAT (Version 1.2): A computer program to calculate F- statistics." *Journal of Heredity* 86: 485-486.

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

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

Weir, B. S., and C. C. Cockerham. 1984. Estimating F-Statistics for the analysis of population structure. *Evolution* 38:1358–1370.

Weir, B. S., and W. G. Hill. 2002. Estimating F-Statistics. *Annu. Rev. Genet.* 36:721–750.

5.3 Quantitative traits

name: **quant**
files: `".quanti"` (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 muiltiple 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 modelled.

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 trait can be set here. It is 0 by default. This parameter is valid for the whole metapopulation. The LCE quanti_init can be used to set patch-specific initial values. The value at each locus is set by dividing the initial value by two times the number of loci. The initial population will then be monomorphic for this trait value, unless specified otherwise

quanti_init_model [0,1] (opt)

If the initialisation model is set to "0", the initial population will be monomorphic for the initial trait values specified previously. If set to "1", a random mutational effect is added to each locus, on top of the initial value. Model "1" is the default.

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 age sex home ped isMigrant father mother ID
```

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

If 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]

The timing at which phenotypes should be saved, or the 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).

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 4.7). Deleterious mutations are coded by bi-allelic loci, with value of 0 for the wild-type, healthy form, and 1 for the deleterious form. The strength of the deleterious effect of each mutation (i.e. strength of selection) and its dominance 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_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_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.

delet_effects_mean [decimal]

Mean effect of the deleterious mutations. Also known as the selection coefficient 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_dom_coef [decimal]

Equivalent to `delet_dominance_mean`, kept for backward compatibility.

delet_sel_coef [decimal]

Equivalent to `delet_effects_mean`, kept for backward compatibility.

delet_continuous_effects [bool] (opt)

Deprecated since version 2.0.7.

delet_fitness_model [1,2]

Sets the fitness model used to compute the individual viability from the deleterious genome (the trait phenotype):

1 : Multiplicative model. The individual fitness (or viability) is computed as the product of the fitness of each locus: $W = 1 - (1 - s)^{n_1} - (1 - hs)^{n_2}$ where n_1 is the number of homozygote loci and n_2, the number of heterozygote loci. s is the selection coefficient and h the dominance coefficient.

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 has same meaning as previously. W is truncated at 0, fitness can never be negative here.

delet_fitness_scaling_factor [integer] (opt)

This parameter's value is used as a scaling factor for the individual's phenotype, i.e. its viability is multiplied by this value.

delet_save_genotype [bool] (opt)

Parameter used to save the population genotypes in a text file with the `".del"` extension. The first line holds the column labels. Each line starts with the population identifier followed by one column per locus plus the age, sex, pedigree class, and patch of origin of each individual. The allelic values are 0 for the wild type allele and 1 for the deleterious allele. For the cases where

mutational effects are continuously distributed, the second row holds the selection coefficient (homozygous effect) of each locus, and the third one holds the heterozygous effects of each locus.

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

delet_genot_logtime [integer] (opt)

This is the generation periodicity of the genotype files or the generations at which the files should be saved if provided as multiple values in an array. If the number is greater than the total number of generations, no data will be saved.

5.5 Dobzhansky-Muller Incompatibility loci

name: **dmi**
files: ".dmi"
phenotype: a real value in $[0, 1]$, interpreted as the fitness value of the individual

The DMI trait codes for so-called (Bateson-)Dobzhansky-Muller Incompatibilities that occur between pairs of loci when both loci are heterozygotes for diploids or for "heterozygous" pairs for haploids. In the latter case, loci in repulsion usually decrease fitness (i.e. aB or Ab have lower fitness than AB or ab). The trait is bi-allelic, with allele 0 representing the wild-type (A, B, C, ...) and 1 the 'mutant' (a, b, c, ...).

The fitness effects of each incompatible pair must be set using a matrix argument (see dmi_genot_table below). The fitness values of all possible genotypes must be specified. The fitness model used is multiplicative:

$$W = \prod (1 + w(pair_i)),$$

where $w(pair_i)$ is the fitness value of the locus pair i.

A specific initializer has also been added to set patch-specific initial frequencies (see dmi_init).

dmi_loci [integer]

Number of incompatible locus. The trait is haploid by default. Incompatibilities come by pair, and pairs of locus are contiguous on the chromosome(s). The recombination rate between each *locus* is set below.

dmi_is_haploid [bool] (opt)

Can be used to change the ploidy of the trait. Is set to 'true' by default. The trait will be diploid is this is set to 0 ('false').

dmi_mutation_rate [decimal]

Per-locus mutation rate. Mutations are both-way ($0 \rightarrow 1$ & $1 \rightarrow 0$).

dmi_recombination_rate, dmi_genetic_map, dmi_random_genetic_map (opt)

Recombination is handled by the genetic map. All genetic map parameters apply. See section 5.1.

dmi_genot_table [matrix]

This table sets the fitness of each pair of locus relative to the wild type. It must be set for each pair explicitly (no repetition of patterns, for now). The structure is: one row per incompatible pair and one column per genotype. There is here a slight difference for the haploid and diploid versions. In the haploid case, the fitness of all 4 genotypes must be given. There are two incompatible pairs, aB and Ab. The fitness associated with each genotype is written in the following order:

- for haploids (and one pair)

{{AB, aB, Ab, ab}}

- for diploids (and one pair):

{{AABB, AABb, AAbb, AaBB, AaBb, Aabb, aaBB, aaBb, aabb}}.

For the diploids, 9 genotypic values must be given. We do not distinguish between single- locus heterozygotes (i.e. $Aa == aA$). The incompatible pair is the middle one, AaBb (element number five in the genotype array). The table below shows the ordering of the genotypes in the array:

	BB	Bb	bb
AA	1	2	3
Aa	4	5	6
aa	7	8	9

The values should be given relative to maximum fitness (= 1). Wild-type genotypes should thus get value 0 and incompatible genotypes should get negative values. Otherwise, be sure to set the fitness model of the selection LCE to relative_local to get relative fitness values (see parameter selection_fitness_model).

dmi_save_genotype [bool] (opt)

Used to tell Nemo to write the genotypes to file.

dmi_logtime [integer] (opt)

Tells every what generation the genotypes should be saved to a text file.

dmi_output_dir [string] (opt)

Tells where (relative to root_dir) to save the genotype files.

STATS

adlt./off.dmi Records the average frequency of allele '1' (the mutant) and the average frequency of incompatibility over all loci/pairs. The output (in the stat files) include patch-specific averages and overall means for both quantities (adlt./off.dmi.freq, adlt./off.dmi.p#)

The incompatible genotype is AaBb in the diploid case (or 01 01 as in the output genotype file) and Ab or aB in the haploid case. The frequency of these genotypes is recorded in the output stat file as adlt./off.dmi.icmp for the overall average, or adlt./off.icmp.p# for the per-deme frequencies.

5.6 Dispersal genes

name: **fdisp, mdisp**
files: NA
phenotype: a real value in $[0, 1]$

If the following parameters are added to the init file, two quantitative traits will be added to the individuals. One codes for the female dispersal rate and is expressed in females. The second codes for the male dispersal rate and is expressed in males only. Both traits are continuous quantitative traits coded by a single diploid locus whose allele values are real numbers ranging from 0 to 1. The two loci are co-inherited. The dispersal probability of an individual (i.e. the trait's phenotype) is the mean of the two allele values at the corresponding locus.

disp_mutation_rate [decimal]

Mutation rate of the dispersal alleles. This is the probability to change the allele value by an amount drawn from an inverse-exponential distribution with the mean set below.

disp_mutation_mean [integer]

This parameter is the mean of the exponential distribution used to draw the mutation step added to the genotype value.

disp_init_rate_fem [decimal] (opt)

Initial genotype (both alleles) of the female dispersal locus.

disp_init_rate_mal [decimal] (opt)

Initial genotype (both alleles) of the male dispersal locus.

disp_init_rate [decimal] (opt)

Initial genotype of both the male and female dispersal locus.

5.7 Wolbachia

name: **wolb**
files: NA
phenotype: a boolean representing the infection status of the individual

The Wolbachia trait is used to simulate the dynamics of an endosymbiotic parasite causing cytoplasmic incompatibility. Its transmission is vertical, through females only and is not perfect, the zygote may loose its parasite ("mutation" process represented by the transmission rate presented below). Zygotes issued from the mating between an infected male and an uninfected female must pay the cost of incompatibility that decreased their chance of survival at birth by a given amount (parameter incompatibility cost of the `breed_wolbachia` LCE). Being infected by *Wolbachia* also induces a cost that translates into a reduced fecundity of the infected females (parameter fecundity cost of the `breed_wolbachia` LCE). See the `breed_wolbachia` LCE for details on the breeding and infection parameters.

wolbachia_transmission_rate [decimal]

This is the rate of transmission of *Wolbachia* from a mother to its offspring. If different from one, the parasite may be lost during gamete formation.

Chapter 6

Examples

6.1 Life cycles

6.1.1 A basic life cycle

To start with, lets exemplify what a basic life cycle looks like:

```
breed 1
disperse 2
aging 3
```

It starts with the reproduction of the population (`breed`), thus adding offspring individuals to it. Then the offspring migrate within the population (`disperse`) before getting older and replacing the previous adult generation that will die because of `aging` (non overlapping generations). The new adult generation is also regulated to not exceed the patches carrying capacities.

Writing this life cycle in a different order would produce exactly the same result, given the sequence of LCEs is conserved (see the following examples). The only change is the population state at the beginning and the end of the cycle.

```
aging 1                 disperse 1
breed 2                 aging 2
disperse 3              breed 3
```

Writing the life cycle as above does not ensure that these LCEs will all be loaded into the life cycle as some of them define additional mandatory parameters that must be present in the init file as well. The `breed` and `disperse` LCEs define such

mandatory parameters. The following example will allow to completely build the life cycle.

```
breed 1
disperse 2
aging 3

mating_system 3        # monogamy
mean_fecundity 3
mating_proportion 0.8  # 20% of extra-pair matings

dispersal_model 2   # Island Model with propagule pool migration
dispersal_propagule_prob 0.3 # 30% of propagule dispersers
dispersal_rate 0.125
```

6.1.2 Adding outputs

The previous basic life cycle misses two important features. It does not record statistics and does either not write any output files. To do so, you have to add the following LCEs, `save_stats` and `save_files`.

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

This way, both the adults and offspring statistics are computed and the various files declared by the simulation components are saved to disc. Which age classes are present in the population at the time of statistics recording and file writing will determine the content of output files (especially the stats output files), the ranks of these LCEs are thus important in that perspective. A third output LCE could have been added here, it is the `store` LCE. Its rank in the life cycle will also determine the age-class content of the binary files.

6.2 Traits

To add a trait to a simulation, it is sufficient to add the mandatory parameters of that trait to the init file. Here is an example with three of the traits currently implemented in Nemo.

```
## NEUTRAL MARKERS ##
ntrl_loci 20
ntrl_all 20
ntrl_mutation_rate 0.0001
ntrl_mutation_model 1    # SSM model

## GENETIC LOAD ##
delet_loci 1000
delet_mutation_rate 0.0001
delet_effects_mean 0.05
delet_dominance_mean 0.36
delet_fitness_model 1    # multiplicative model

## DISPERSAL GENES ##
disp_mutation_rate 0.001
disp_mutation_mean 0.2
```

Each individuals in the simulation will thus carry four sets of genes. One coding for neutral markers with 20 loci, one with 1000 loci carrying deleterious mutations and two coding for female and male dispersal. The genotypes can be saved in binary files using the `store` LCE or by adding the trait-specific output parameters and the `save_files` LCE somewhere in the life cycle.

6.3 A complete example

The next example shows a complete init files with all the mandatory parameters and all the trait output parameters.

```
## SIMULATION ##
filename example
logfile logfile.log
root_dir test
random_seed 988889
run_mode overwrite

replicates 10
generations 1000

## POPULATION ##
patch_number 50
```

```
patch_capacity 20

## LIFE CYCLE ##
breed_selection 1
save_stats 2
save_files 3
disperse_evoldisp 4
aging 5
store 6
extinction 7

# breed and selection parameters #
selection_trait delet
selection_model direct
mating_system 3        #monogamy
mean_fecundity 15      #high enough to resist inbreeding depression
mating_proportion 0.8  #20% of extra-pair mating

# extinction parameter #
extinction_rate 0.05

# disperse parameters #
dispersal_model 2
dispersal_propagule_prob 0.3
dispersal_rate 0.125

# save_stats parameters #
stat off.fstat off.delet viability disp demography extrate
stat_log_time 10
stat_dir stat

# store parameters #
store_dir binary
store_generation 1000
store_noarchive

## NEUTRAL MARKERS ##
ntrl_loci 20
ntrl_all 256
ntrl_mutation_rate 0.0001
ntrl_mutation_model 1
# ouput #
ntrl_save_genotype
```

```
ntrl_output_dir ntrl
ntrl_output_logtime 1000

## GENETIC LOAD ##
delet_loci 100
delet_init_freq 0
delet_mutation_rate 0.0001
delet_effects_distribution exponential
delet_effects_mean 0.05
delet_dominance_mean 0.36
delet_fitness_model 1
# ouput #
delet_save_genotype
delet_genot_dir delet
delet_genot_logtime 1000

## DISPERSAL GENES ##
disp_mutation_rate 0.001
disp_mutation_mean 0.2
dispersal_cost 0.2
```

This example will produce the following files (with # representing the replicate number from 01 to 10).

```
logfile.log
test/example.log
test/stat/example_bygen.txt
test/stat/example.txt
test/ntrl/example_#.dat
test/delet/example_#.del
test/binary/example_#.bin.bz2
```

More elaborate examples can be found in the `example/` folder of the installation package.

Chapter 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 4.13). 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
off.demography	off.nbr	total number of offspring in the metapopulation
	off.nbfem	mean number of female offspring per extant patch
	off.nbmal	mean number of male offspring per extant patch
	off.density	average offspring density
	off.dvar	variance of the offspring density of extant patches
adlt.demography	adlt.nbr	total number of adults in the metapopulation
	adlt.nbfem	mean number of females per extant patch
	adlt.nbmal	mean number of males per extant patch
	adlt.density	average adult density
	adlt.dvar	variance of the adult density of extant patches
demography		the above demographic stats for offspring and adults
extrate	extrate	proportion of extinct patches in the population
fecundity	adlt.femfec	mean assigned females fecundity
	adlt.femrealfec	mean effective females fecundity, discounting offspring that do not survive, different from the previous one only when viability selection occurs with breeding
	adlt.femvarfec	mean variance in effective fecundity of females
	adlt.malrealfec	mean effective males fecundity
	adlt.malvarfec	mean variance in effective fecundity of males
kinship	off.fsib	mean proportion of full-sib

Table 7.1 continued on next page

Stat option	Output name	Description
	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
	residents	mean number of residents per patch
	immigrate	effective immigration rate computed as $(\frac{immigrants}{immigrants+residents})$
	colonisers	mean number of immigrants per extinct patch
	colonrate	effective colonisation rate of extinct patches
migrants.patch	emigr.pi	number of emigrants from patch i
	resid.pi	number of residents in patch i
	imrate.pi	effective immigration rate into patch i computed as $(\frac{immigrants}{immigrants+residents})$
	colo.pi	number of colonizers of patch i; is -1 if patch wasn't extinct. A value of 0 means the patch was extinct but not recolonized.

Table 7.1 continued on next page

Stat option	Output name	Description
pop		same as "demography", off/adlt.sexratio, and "extrate" together
pop.patch	off./adlt.fem.p*i*	number of females in patch *i*
	off./adlt.mal.p*i*	number of males in patch *i*
	age.patch*i*	time since last extinction of patch *i*
	patch.avrg.age	mean time (generation) since last extinction of a patch
	extrate	proportion of extinct patches in the population
off/adlt.fem.patch	off./adlt.fem.p*i*	number of females in patch *i*
off/adlt.mal.patch	off./adlt.mal.p*i*	number of males in patch *i*
adlt.sexratio	adlt.sexratio	see above
off.sexratio	off.sexratio	offspring sex ratio

Table 7.1: Population stat options 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.] coa	*age*.theta	mean within deme coancestry
	age.alpha	mean between demes coancestry
adlt.coa.persex	adlt.thetaFF	mean within deme, within females coancestry
	adlt.thetaMM	mean within deme, within males coancestry
	adlt.thetaFM	mean within deme, between sexes coancestry
adlt./off. coa.within	adlt./off.theta	as above
adlt./off. coa.between	adlt./off.alpha	as above
[adlt./off.] coa.matrix	*age*.theta	as above
	age.alpha	as above
	age.coa*i*.*i*	deme specific mean coancestry within deme *i*, for all demes.

Table 7.2: Neutral markers continued on next page

Stat option	Output name	Description
	age.coa*i*.*j*	deme specific mean coancestry between demes i and j, for all pairwise comparisons.
[adlt./off.] coa.matrix.within	*age*.theta	as above
	age.coa*i*.*i*	deme specific mean coancestry within deme i, for all demes.
sibcoa	prop.fsib	mean proportion of full-sib
	prop.phsib	mean proportion of paternal half-sib
	prop.mhsib	mean proportion of maternal half-sib
	prop.nsib	mean proportion of non-sib
	coa.fsib	mean coancestry within full-sib
	coa.phsib	mean coancestry within paternal half-sib
	coa.mhsib	mean coancestry within maternal half-sibs
	coa.nsib	mean coancestry within non-sib
[adlt./off.] ntrl.freq	*age*.ntrl.l*i*.a*j*	frequency of allele j at locus i in the whole population
	age.ntrl.l*i*.Het	mean heterozygosity of locus i in each patch
[adlt./off.] fstat	*age*.allnb	mean number of alleles per locus in the whole population
	age.allnbp	mean number of alleles per locus within demes
	age.fixloc	mean number of fixed loci in the whole population
	age.fixlocp	mean within demes number of fixed loci
	age.ho	observed heterozygosity
	age.hsnei	expected demic heterozygosity (Nei & Chesser 1983)
	age.htnei	expected total heterozygosity
	age.fis	F_{IS} (Nei & Chesser 1983)
	age.fst	F_{ST} (G_{ST}; Nei & Chesser 1983)
	age.fit	F_{IT} (Nei & Chesser 1983)
[adlt./off.] weighted.fst	*age*.fst.WH	the Weir&Hill (2002) F_{ST} estimate

Table 7.2: Neutral markers continued on next page

Stat option	Output name	Description
[adlt./off.] weighted.fst.matrix	*age*.fst.WH	the Weir&Hill (2002) F_{ST} estimate
	age.fst*i*.*i*	deme specific F_{ST} within deme i, for all demes.
	age.fst*i*.*j*	deme specific F_{ST} between demes i and j, for all pairwise comparisons.
[adlt./off.] weighted.fst.within	*age*.fst.WH	the Weir&Hill (2002) F_{ST} estimate
	age.fst*i*.*i*	deme specific F_{ST} within deme i, for all demes.
[adlt./off.]fstWC	*age*.fis.WC	the Weir&Cockerham (1984) F_{IS} estimate (f)
	age.fst.WC	the Weir&Cockerham (1984) F_{ST} estimate (θ)
	age.fit.WC	the Weir&Cockerham (1984) F_{IT} estimate (F)
[adlt./off.] mean.NeiDistance	*age*.D	mean b/n demes Nei's genetic distance (D).
[adlt./off.] NeiDistance	*age*.D*i*.*j*	pairwise Nei's genetic distance b/n demes i and j, for all pairs.
[adlt./off.] Dxy	*age*.Dxy	average pairwise sequence divergence between all pairs of patches
[adlt./off.] Dxy.patch	*age*.Dxy.p*i*p*j*	average pairwise sequence divergence between patch i and patch j

Table 7.2: Neutral markers stat options continued

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)

Table 7.3: Quantitative traits continued on next page

Stat option	Output name	Description
	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)
[adlt./off.] quanti.covar.patch	*age*.cov.q*ij*.p*k*	genetic covariance between trait i and j in patch k
[adlt./off.] quanti.eigen.patch	*age*.qeval*i*.p*j*	eigenvalues of the G-matrix in patch j (genetic covariance matrix)
	age.qevect*ij*.p*k*	loadings of trait j on eigenvector i of the G-matrix in patch k
[adlt./off.] quanti.eigenvalues.patch	*age*.qeval*i*.p*j*	eigenvalues of the G-matrix patch j
[adlt./off.] quanti.eigenvect1.patch	*age*.qevect1*i*.p*j*	loadings of trait i on the first eigenvector of the G-matrix in patch j

Table 7.3: Quantitative traits continued on next page

Stat option	Output name	Description
[adlt./off.] quanti.skew.patch	*age*.Sk.q*i*.p*j*	skew of the phenotypic distribution of trait i in patch j
[adlt./off.] quanti.patch		adds the stats from quanti.mean.patch, quanti.var.patch, quanti.covar.patch, and quanti.eigen.patch

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 demic number of fixed mutation
	age.delsegr	mean number of segregating mutation in the whole population
	age.delsegrp	mean demic number of segregating mutation
	age.delfst	Fst of the deleterious mutations
	age.lethequ	mean number of lethal equivalents
	age.heterosis	heterosis computed as: $H = 1 - \frac{b_g}{b_p}$ b_g : the effective fecundity of within deme matings (mating partners are from the same patch) b_p : the effective fecundity of between deme matings (mating partners are from different patches)

Table 7.4: Deleterious mutations continued on next page

Stat option	Output name	Description
	age.load	mean demic mutational load computed as: $L = 1 - \frac{\bar{W}}{W_{max}}$ where W_{max} is the maximum number of surviving offspring produced by a female in a patch
		Note: heterosis and load are computed from the female fecundities which are updated according to the offspring survival in the breed_selection LCE only, and are thus null when viability selection is performed differently. In that case, they can be inferred from the fitness stats.
[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 continued on next page

Stat option	Output name	Description
survival		now part of the selection LCE's stats.

Table 7.4: Deleterious mutations stat options continued

7.7 Dobzhansky-Muller Incompatibilities (DMI)

Table 7.5 DMI stat options

Stat option	Output name	Description
[adlt./off.] dmi	*age*.dmi.freq	overall average frequency of mutant alleles, across loci
	age.dmi.p*i*	patch-specific frequency of mutant alleles, across loci
	age.dmi.icmp	overall average frequency of the incompatible genotype(s) (AaBb for diploids, Ab and aB for haploids)
	age.dmi.icmp.p*i*	patch-specific frequency of the incompatible genotype(s), across loci

Table 7.5: DMI stat options continued

7.8 Selection

Table 7.6 Selection stat options

Stat option	Output name	Description
fitness	*age*.fitness.mean	mean of the within patch **offspring** fitness *before* viability selection, i.e., including all offspring
	fitness.outb	fitness of b/n demes outbred offspring
	fitness.outw	fitness of w/n demes outbred offspring
	fitness.hsib	fitness of half-sib crosses
	fitness.fsib	fitness of full-sib crosses
	fitness.self	fitness of selfed crosses
fitness.prop	prop.outb	proportion of b/n demes outbred **offspring**

Table 7.6: Selection stats continued on next page

Stat option	Output name	Description
	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 7.6: Selection stat options continued

7.9 Dispersal

Table 7.7 Dispersal stat options

Stat option	Output name	Description
[adlt./off.] disp	*age*.disp	mean dispersal rate
	age.fdisp	mean female dispersal rate
	age.mdisp	mean male dispersal rate

Table 7.7: Dispersal stat options continued

7.10 Wolbachia

Table 7.8 Wolbachia stat options

Stat option	Output name	Description
wolbachia	off.fwoinf	mean infection frequency of offspring females

Table 7.8 continued on next page

Stat option	Output name	Description
	off.mwoinf	mean infection frequency of offspring males
	off.incmating	mean number of incompatible matings
	adlt.fwoinf	mean demic infection in extant demes for adult females
	adlt.mwoinf	mean infection frequency of adults males in the whole population
	wolb.infvar	inter-demic variance in adult female infection
	wolb.extrate	proportion of demes having lost infection in adult females
wolbachia_perpatch	off.p*i*fwoinf	mean infection frequency of offspring females in patch i
	off.p*i*mwoinf	mean infection frequency of offspring males in patch i

Table 7.8: Wolbachia stat options continued