Instructions for the *TREES* program, v 1.3

Jörgen Ripa, Lund, 2018

Introduction

The *TREES* program is a versatile tool for individual based simulations of large systems of evolving, competing species. It includes trait-based, eco-evolutionary feedbacks, full-blown genetics and evolving mating behavior. A main feature is the possibility to combine different modules to a large variety of models, which can be aimed at a multitude of scientific questions.

TREES provides a versatile platform for the analysis of macroevolutionary mechanisms and patterns, as driven by individual interactions and microevolution. The program offers ample realism, detail and possibilities for advanced analysis. At the same time, it is easy to use. A new model design can be put together in minutes, using a simple script. The program is written in C++ for maximal performance. It is fully object-oriented, which makes it easy to extend for future needs.

Contents

Installing the software

Installing is a basic procedure of downloading an executable.

Mac:

1. Follow this link: https://github.com/jorgenripa/TREES/releases and download TREES_1.3_Mac.

2. Save the file in a directory where you will run simulations. Rename the file TREES for convenience.

3. Open a Terminal window and change directory to the same directory as the TREES file. Use the `cd` command to change directory. Change the permissions of the TREES file like this:

```
chmod +x TREES
```

The program can now be run in a terminal window using

```
./TREES test.txt
```

The above command assumes there is a parameter file called `test.txt` in the same directory (an example can be found in the github repository: http://raw.githubusercontent.com/jorgenripa/TREES/master/test.txt).

The advanced user would know how to place the TREES executable in a more convenient place and create a symbolic link in the `/usr/local/bin` directory. This would make the somewhat awkward "./" unnecessary and make the TREES command available from anywhere.

UNIX:

1. Follow this link: https://github.com/jorgenripa/TREES/releases and download TREES_1.3_UNIX.

2. Follow the Mac instructions above (starting at point 2).

Windows:

A Windows version is unfortunately not available for version 1.3

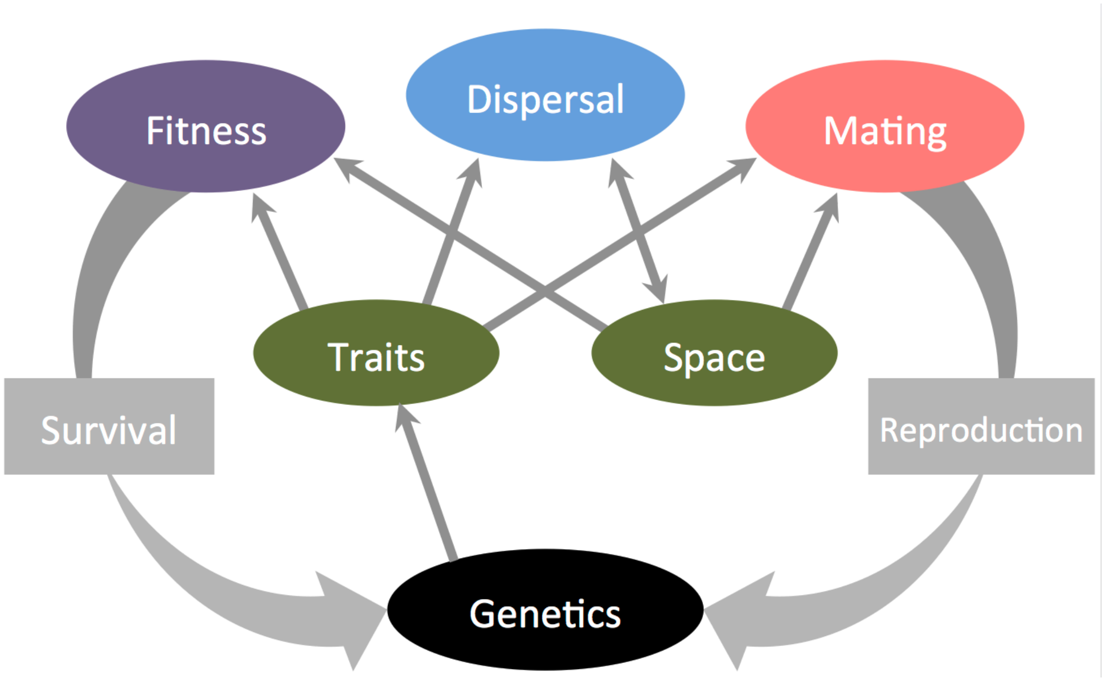
Other:

The complete code can be downloaded ('Source code') and compiled locally using gcc and the included build.sh script.

DISCLAIMER

Model design

The general model structure is shown in the figure below.



Each individual has a set of *traits* and a position in *space* (one or more coordinates). Each trait is controlled by a set of loci in the individual's genome.

The traits and the position in space of an individual determines its fitness and controls its dispersal and mating behavior. Individual fitness and mating success dictate the genetic composition of the next generation.

The content of each colored ellipse in the diagram above is controlled by the user. There are a number of *modules* that can be specified and combined to a large number of possible model configurations. The number of scientific questions that can be addressed is therefore also large.

All modules can be combined with each other, with few exceptions. It is thus possible to change just one part of a model, for instance the genetics module or a mating module, but leave the rest intact. It is in this way easy to make informative comparisons and investigate underlying mechanisms.

The model design is specified in a single text file, which is easy to read, edit and save for future reference. It can also be published as the exact definition of a particular model, making published results easy to reproduce.

Program flow

Each generation of the model comprises the following steps, in this order:

1. **Reproduction**. Each individual chooses a mate from the local *mating pool* according to its preferences and produces *F* offspring. Each offspring inherits the genes of the two parents recombined in standard manner, and the spatial position of the mother (the choosing mate). All individuals are diploid and hermaphroditic.
2. **Generation shift**. The parent generation is discarded, replaced by their offspring.

3. **Dispersal.** Each individual disperses with a given probability, which is a heritable trait. If no space module is defined, this step is skipped.
4. **Survival.** The fitness of each individual is determined depending on its traits, local interactions, and its position in space. Survival is random for each individual, but the probability to survive is proportional to its fitness.
5. **Sampling.** The population is sampled at fixed generation intervals. The samples are saved to a file at the end of the simulation.

The steps above are controlled by a set of *modules*, set by the user. The different module types are described in detail below, but we will start with an example model.

Building the model

A model is constructed through a single text file, where all modules and parameters are specified. The general syntax is

```
specifier : value
```

where `specifier` can be either a parameter name or a module type. The colon between `specifier` and `value` is required, but the spaces on each side are optional. The whole file is composed of such [`specifier` : `value`] pairs, separated by a new line or a comma. Please pay attention to the order of parameter specifications, since changing the order usually results in an error. The program is indifferent to upper or lower case, except for trait names.

The '#' character is used for comments and means the rest of a line is ignored. Tab characters and empty lines are ignored and can be added for structure. A sample model file is given below, with green-colored comments for clarity:

```
#Simulation parameters:
t_max : 4000 # the length of the simulation in generations
sample_interval : 200 # the number of generations between each sample
microsamples : means # some information saved every generation
checkpoint_interval : 1000 # how often to save checkpoints
keep_old_checkpoints : N # save all checkpoints, or only the latest?
seed : R                # this can be a fixed integer, or R for Random.
gene_tracking : N       # a Y/N variable
gene_sampling : N       # a Y/N variable

# Population parameters:
F : 2 # Fecundity parameter
n_0 : 100 # initial population size

# Specify a Genetics module:
Genetics : Continuous_Alleles, P_mutation : 1e-4

# Specify traits:
Trait : beak_size, dimensions: 1, loci_per_dim: 8, initial_value: 0
     Transform: linear, offset: 1, scale: 0.05

# Specify a Space module (optional)
# no space in this case

# Specify fitness module(s):
Fitness : Resource_landscape
     trait : beak_size # This is the evolving trait
     r : 1
     K_0 : 500
     s_K : 1
     s_a : 0.5
     s_space : 0 # spatial extent of competition
     k_space : 0 # spatial gradient of resource optimum

# Set the mating parameters:
Mating_pool : Global
Mating_trials : 100 # Trial matchings before a mating fails.

# Add mating preference modules (optional)
```

The parameter file above specifies a model of an evolving consumer feeding from a continuous (Gaussian) resource landscape. A one-dimensional trait `beak_size` controls the resource niche position of an individual. Mating is random.

The first few lines define a few necessary model parameters, described below. Next, the `Genetics` module is specified. There are three types to choose from – `Continuous_alleles`, `Diallelic` and `Omnigenic`.

After the Genetics module follows a list of one or several traits, in the example just one. Each trait is given a name ('`beak_size`' in the example). Each trait also has a number of dimensions and a number of loci per dimension. In the example, the trait `beak_size` is one-dimensional with 8 coding loci. Each trait is also followed by a (optional) list of transforms, which is part of the genotype-to-phenotype mapping.

After the list of traits follows the optional specification of a `Space` module, defining the spatial structure and dispersal parameters. If no `Space` module is defined, a panmictic population is assumed.

Next follows specifications of one or several `Fitness` modules, in arbitrary order. These modules determine the fitness of each individual, which in turn regulates individual survival. Fitness is applied in a multiplicative manner, i.e. the fitness of an individual is the product of the fitness values supplied by each fitness module.

Lastly follows a few mating parameters, described in detail below. The example has random mating, for simplicity.

To run the model above, save the specification as a text file, such as beak_size_model.txt, and run it from a command prompt by issuing the TREES command followed by the name of the parameter file:

Mac / UNIX:

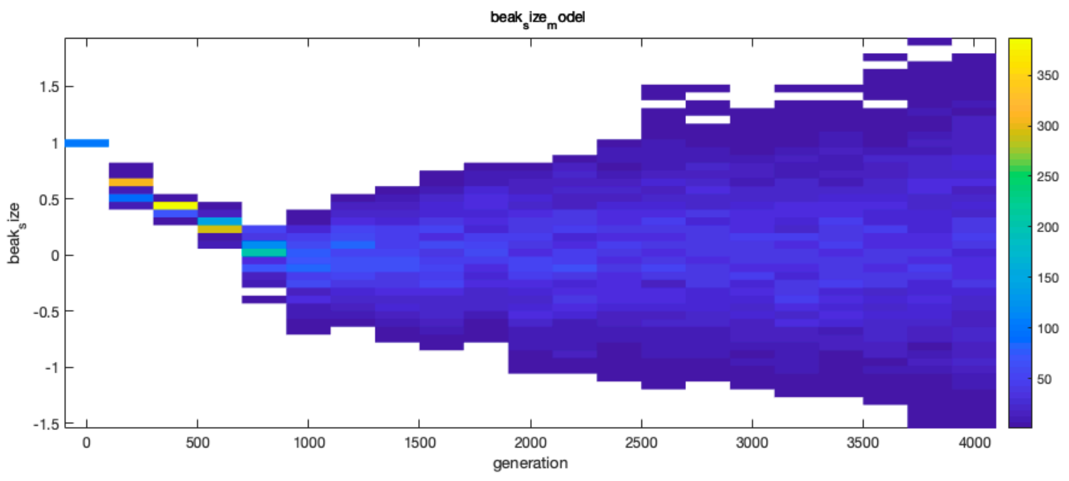
```
 ./TREES beak_size_model.txt
```

PC:

```
TREES beak_size_model.txt
```

The program output is a results file with extension `.sim`. It takes the same name as the parameter file, appended with `_results` and an index, defaulting to 1. The results file in this example is `beak_size_model_results_1.sim`. It can be imported to Matlab™ and plotted according to:

```
>> beak_sim = read_sim('beak_size_model.txt', 1);
>> plot_sim(beak_sim)
>> colorbar
```



The figure shows a heat map of the abundances of phenotypes over time. It can be seen that the evolving trait `beak_size` first converges to zero, the point of maximal resource abundance, and next diversifies to a large continuous distribution. Mating is random, so speciation is not possible.

Below follow detailed descriptions of all sections of the parameter file and all available modules. It is not necessary, and not even recommended, to read all descriptions before using the program. It is quite possible to instead cherry-pick the sections of one's personal interest and refer back to this document when necessary.

Simulation parameters

`t_max` is simply the length of the simulation (in generations). This has to be an above-zero integer, but can be specified using a scientific format: 1.5e5 is equivalent to 150000. The same format is allowed for all integer parameters.

`sample_interval` specifies how often the population will be sampled. All samples are saved in the output file. Frequent sampling gives high resolution output, but may generate very large data files. An advice is to first experiment with short simulations (a few hundred generations) to find an appropriate value. Remember to take possible increase in population size into account, since sample size scales with population size. The first generation (generation 0) is always sampled, followed by generations `sample_interval`, 2×`sample_interval`, and so on.

`microsamples` are small samples taken every generation. There are four options available:

`microsamples : none` (or just `n`) make no micro-samples

`microsamples : means` (or just `m`) save the population means of all traits and spatial positions (if applicable)

`microsamples : vars` (or just `v`) save means and variances of all traits and spatial positions (if applicable).

`microsamples : covars` (or just `c`) save means, variances, and covariances of all traits and spatial positions (if applicable)

Each microsample also contains a cpu-time, i.e. the cpu clock-time passed since the beginning of the simulation.

Note: Trait constants (see below) are never included in microsamples.

`checkpoint_interval` sets the interval between *checkpoints*, which are more extensive samples. One can set **`checkpoint_interval`** to zero in order to prevent checkpoints entirely and save file size. At each checkpoint, all samples and checkpoints are saved to the results file. Without checkpoints, the results will not be saved until the whole simulation is completed. Checkpoints can be used to *resume* a simulation at that point in time. See the resume option below.

keep_old_checkpoints is a Y/N variable. The value Yes (Y) means all checkpoints will be saved. The value No (N) means only the latest checkpoint will be saved. A No value will save disk space, but the simulation can only be resumed from the last checkpoint.

`seed` determines the seed for the random number generation. If `seed` is set to 'Random' (or just 'R'), as in the example, a seed is generated from the system clock with nanosecond resolution. The seed value (an integer) is later stored in the output file, which makes it possible to repeat simulations, perhaps with a smaller `sample_interval` or a larger `t_max`. This is a handy way to analyze specific events in more detail or investigate an interesting evolutionary scenario on a longer time-scale.

`gene_tracking` takes values Y (Yes) or N (No). If this option is on (Yes), individual genes are tracked throughout the simulation. This is described in a separate chapter below.

`gene_sampling` takes values Y or N, just like `gene_tracking`. If this option is on (Yes), the genotypes of all individuals will be sampled together with the phenotypes. This offers additional detail, such as the possibility to calculate F_{ST}-values, but may cause large output files.

Population parameters and initialization

Two population parameters have to be specified, in this order:

`F` (for fecundity) sets the fixed number of offspring produced by each individual in the reproduction stage. `F` has to be an integer larger than 1 and is an important fitness component. See the section on fitness modules below.

`n_0` is the number of individuals of generation 0.

Each simulation starts with n_0 identical individuals with identical spatial coordinates. The genes are initiated as described below in the Genetics section, and the initial spatial position is specified in the Space section. The phenotypes are derived directly from the genotypes, as described in the Traits section.

Genetics

Each individual has a its own diploid genome, inherited from both parents in a standard manner. Each locus recombines independently. The two alleles at each locus take values depending on the selected `Genetics` module:

Diallelic

This module implements an additive diallelic genetic model. Each locus has two possible alleles – a plus- and a minus-allele. The effect of each allele is ±1/2, i.e. the difference in allelic effects is 1. The phenotypic value of a trait (prior to transforms) is given by the sum of all allelic effects. As an example, consider a trait coded by 4 loci with the following genotype: [++, +–, –+, ++]. The corresponding phenotype equals 6×1/2 – 2×1/2 = 2. The range of possible trait values is in this case between –4 and +4. In general, a trait coded by *L* loci can take values between –*L* and +*L* (before transforms).

All traits listed in the model file are allocated the specified number of loci, all inherited independently.

Mutations occur at a rate `P_mutation` per allele per generation and transforms a plus-allele to a minus-allele and vice versa. The phenotypic effect of a single mutation is thus ±1. `P_mutation` is the single parameter of this module and has to be given a value. Example syntax:

```
Genetics: Diallelic, P_mutation: 1e-5
```

Continuous_Alleles

This module implements an additive continuum-of-alleles genetic model. Allelic effects can take any value. Mutations change the allelic effect according to a symmetric double exponential (Laplace) distribution with standard deviation 1, comparable to the `Diallelic` module.

In all other respects, the `Continuous_Alleles` module behaves as the `Diallelic` module. Example syntax:

```
Genetics: Continuous_Alleles, P_mutation: 1e-5
```

Omnigenetic

This module generates maximal pleiotropy, in the sense that all genes affect all traits. An *n*×*L* matrix, *B*, that maps the effects of *L* loci onto *n* traits (or trait components) is generated at simulation initialization. The elements of *B* are drawn independently from a double exponential distribution and standardized such that each row of *B* has a squared vector length equal to `loci_per_dim` / L, where `loci_per_dim` is set for each trait separately (see below). This standardization assures that an increase in *L* does not change the mutational variance added each generation, and the mutational increase of each trait remains relatively the same, compared to the other genetic modules.

The total number of loci, *L*, is set by the variable `Loci` in the module definition:

```
Genetics: Omnigenetic, P_mutation: 1e-6, Loci : 50
```

Choosing a large value for `Loci` (compared to the number of traits) leaves many degrees of freedom for evolution. A smaller value puts more constraints to evolutionary change. Another way of constraining evolution is to define a number of traits with strong stabilizing selection (see Traits and Fitness modules below).

Traits

The user specifies the number of heritable traits in the model. The traits are incorporated in the `Fitness`, `Space` and `Mating_preference` modules as parameters.

The traits are specified one after the other in the model file. Each trait has a name and three parameters – `dimensions`, `loci_per_dim`, and `initial_value`. Example:

```
Trait : A, dimensions : 3, loci_per_dim : 5, initial_value = 0
```

Trait *names* have to be unique, but can be any sequence of letters, digits and underscores, preferably chosen to describe the nature of the trait (such as 'size' or 'color'). Note that the trait name is the only case when the difference between upper and lower case is acknowledged by the program. It is thus possible, but not recommended, to have two separate traits labeled `a` and `A`.

The `dimensions` variable sets the number of dimensions. A value of 1 corresponds to a simple, real-valued trait. Higher integer values of `dimensions` generates vector-valued traits.

`loci_per_dim` sets the number of appointed loci per dimension. The example above generates a three-dimensional trait called A coded by in total 3×5 = 15 loci.

Finally, `initial_value` sets the starting value for the trait, given to all individuals in generation 0, in all dimensions. The starting population consists of identical, completely homozygote, individuals. Each trait is set to its `initial_value`, but the implementation varies between the genetics modules.

In the `Continuous_Alleles` genetics module, the first locus is set to be homozygote with allelic effects equal to `initial_value` / 2. All other loci have starting value 0.

In the `Diallelic` module, there are some constraints to the possible phenotypes. The program tries to find a homozygote genotype that matches the `initial_value`, and outputs an error if it fails. For instance, consider the case with just two loci (`loci_per_dim` = 2). The possible homozygote genotypes are [+ + / + +], [+ + / – –], [– – / + +], and [– – / – –], corresponding to the phenotypes 2, 0, 0 and –2 (each allele contributes +/–0.5). The only allowed values of `initial_value` are thus 2, 0 and –2. In general the possible values range from –`loci_per_dim` to +`loci_per_dim` in steps of 2.

In the `Omnigenic` module, all loci are initiated homozygous at value 0. The initial value of each trait is implemented as a constant added to the trait value in the genotype-phenotype mapping (prior to any transforms).

Constant Traits

It is possible to define constant, non-evolving, traits. This is useful if a trait can evolve in some simulations, but not in all. It is then sufficient to change the definition of the trait itself, without changing its usage.

The syntax to define a constant trait is:

```
Trait_constant : X, dimensions : 3, initial_value = 4.5
```

The only difference to an ordinary trait is the keyword `Trait_constant` and that the parameter `loci_per_dim` is missing. The trait is fixed at the value set by `initial_value`, in all dimensions. A constant trait is not saved in population samples. In all other respects, a constant trait functions as an ordinary trait.

Transforms

The genotype-to-phenotype mapping of each trait is completed by a (possibly empty) list of transforms. The transforms are applied in the order they are listed. Multi-dimensional traits are transformed one dimension at a time. **Note** that the `initial_value` parameter applies to the *untransformed* trait.

Linear transform

The `linear` transform is perhaps the most common. It transforms a phenotypic value x according to

$$y = offset + scale \cdot x$$

The `linear` transform is, among other things, used to set the size of mutations. By default, mutations have a standard variation of 1. A linear transform can rescale the phenotype such that mutations have an effect of arbitrary size. For example

```
Trait : color, dimensions : 3, loci_per_dim : 4, initial_value : 0
     Transform : linear, offset : 1, scale : 0.01
```

introduces a three-dimensional trait `color` with mutational effects of standard deviation 0.01. The `offset` parameter (here set to 1) determines the initial phenotype since the untransformed phenotypes start at zero (`initial_value` = 0).

Abs transform

The `abs` transform removes the sign of a phenotypic value. This may be useful to avoid negative values. There are no parameters. Example syntax:

```
Trait : size, dimensions : 1, loci_per_dim : 8, initial_value : 1
     Transform : abs
```

Range transform

The `range` transform conveniently transforms a trait to a given range [`min`, `max`]. This only applies to models with a `Diallelic` genetics module. It is implemented as a `linear` transform. Example syntax:

```
Trait : size, dimensions : 1, loci_per_dim : 8, initial_value : 1
     Transform : range, min : -10, max : 10
```

In the above example the trait size will be rescaled such that the lowest possible trait value (all alleles= –1/2) is -10 and the maximal trait value (all alleles = +1/2) is 10.

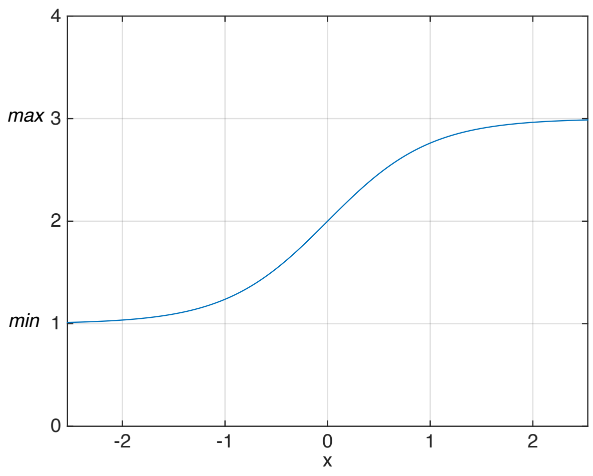
Logistic transform

The `logistic` transform is a more elaborate way of putting boundaries to phenotypic values. It has two parameters, `min` and `max`, which set the lower and upper boundaries, respectively. The functional relationship is given by

$$y = min + R/(1 + e^{-\frac{4x}{R}}) ,$$

where R = `max` – `min`. The function is scaled such that the slope at $x = 0$ is equal to 1. This means that mutational effects near $x = 0$ are in principle unaffected, but as x grows to large positive or negative numbers the transformed mutational effects become increasingly smaller. Example:

```
Trait : size, dimensions : 1, loci_per_dim : 8, initial_value : 0
     Transform : logistic, min : 1, max : 3
```



Normal_Deviate transform

This adds a normally distributed random deviate with zero mean to the phenotypic value, independently to all individuals. It can be used to introduce environmental noise to phenotypes, reducing the heritability. The single parameter is the standard deviation, `SD`. Example:

```
Trait : size, dimensions : 1, loci_per_dim : 8, initial_value : 5
     Transform : Normal_Deviate, SD : 0.5
```

Combined transforms

Finally, here follows a few examples of combined transforms.

```
Trait : size, dimensions : 1, loci_per_dim : 8, initial_value : 0
     Transform : linear, offset : 0, scale : 0.01
     Transform : logistic, min : 0, max : 2
```

Description: A trait `size` with mutational size 0.01, bounded to the interval (0, 2). The initial value can be calculated as the combined transform of the initial untransformed value 0. The first linear transform does not alter it (`offset` is 0), but the logistic transform maps a zero to the interval midpoint, which in this case is 1.

Space

Following the list of traits there is the possibility to specify a Space module. It can also be left out, which results in a totally panmictic population (complete mixing).

The space module defines the structure of space, but also controls the dispersal behavior of individuals.

Note that some fitness modules (below) require a particular space module.

Discrete Space

Space is implemented as a suite of discrete patches with this module. The position of an individual is an integer value from 0 to (`size` – 1), where `size` is an integer parameter > 0. The position can also be multi-dimensional, with one integer value per dimension. All individuals are initiated in the same place, specified by the variable `initial_position` (applied to all dimensions).

Each individual disperses independently with probability `P_disperse`, which takes values from a heritable trait. The trait has to be one-dimensional and specified in the preceding traits section, as described above. If the space module has more than one dimension, the direction (dimension) of dispersal is chosen at random.

There are three dispersal behaviors, or *types* to choose from:

- `Neighbor` (or just `N`) : Move to one of the two neighboring patches (position +/– 1).
- `Global` (or just `G`): Choose a patch randomly.
- `Distance` (or just `D`): Disperse a distance +/–X, where X is drawn from a geometric distribution with mean `dispersal_distance`.

There are also three boundary conditions to choose from:

- `Reflective` (`R`): dispersal across the boundaries is simply reflected
- `Circular` (`C`): classic circular boundaries
- `Absorbing` (`A`): individuals dispersing across the boundaries are removed

Note: The `Circular` boundary condition only applies to dispersal. Competition and mate choice do not occur across boundaries.

Syntax examples:

```
Trait_constant : PD, dimensions : 1, initial_value : 1e-3

…

Space : Discrete
size : 2            # number of patches
dimensions : 1      # number of dimensions
P_disperse : PD # dispersal probability, per individual
dispersal_type : Neighbor # one of Neighbor, Global, Distance
    #dispersal_distance : 1 # required if dispersal_type : Distance
boundary : Circular  # one of: Reflective, Circular, Absorbing
initial_position : 1  # the initial patch number of all individuals
```

A briefer but equivalent specification reads:

```
Space : Discrete, size : 2, dimensions : 1, P_disperse : PD
    dispersal_type : N, boundary : C, initial_position : 1
```

Here's a two-dimensional case, a 5x5 grid, and a specified dispersal distance:

```
Space : Discrete, size : 5, dimensions : 2, P_disperse : PD
     dispersal_type : D, dispersal_distance : 1.5
     boundary : A
     initial_position : 1
```

The first example has a constant `P_disperse` trait, but it can also evolve, if desired. A couple of trait transforms are then useful to rescale the size of mutations and keep the trait value between 0 and 1:

```
Trait : PD, dimensions : 1, loci_per_dim : 20, initial_value : -10
     Transform : linear, offset : 0, scale : 0.05
     Transform : logistic, min : 0, max : 1

…

Space : Discrete, size : 2, dimensions : 1, P_disperse : PD
     dispersal_type : N, boundary : C, initial_position : 1
```

Continuous Space

This is a continuous space implementation, but otherwise with much the same functionality as the `Discrete` module. An individual's spatial coordinate is a real number in the interval $0 \leqslant \text{position} < \texttt{size}$, one coordinate per dimension. The dispersal distance is drawn from an exponential distribution with mean `dispersal_distance`. The dispersal *direction* is drawn from a circular / spherical distribution, depending on dimensionality.

Syntax example:

```
Trait_constant : PD, dimensions : 1, initial_value : 1e-3

…

Space : Continuous
size : 100 # the size of the world in each dimension
dimensions : 1 # number of dimensions
P_disperse : PD # dispersal probability, per individual
dispersal_distance : 1 # average distance (exponential distribution)
boundary : Reflective # one of: Reflective, Circular, Absorbing
initial_position : 50 # initial position of all individuals
```

Fitness

Natural selection is implemented through differentiated survival of individuals. The survival stage consists of two steps: First, the fitness of all individuals is calculated by multiplying the fitness given by all *Fitness* modules. Second, each individual survives with a probability equal to its fitness divided by `F` (the fecundity). In this way, the total fitness is always survival multiplied with fecundity, as it should in a model with non-overlapping generations.

The survival probability (*fitness* / `F`) is truncated above at 1, which means that the realized fitness is never larger than `F`. A large value of `F` increases the demographic stochasticity and puts less limits to individual fitness. It also means a large number of offspring are produced each generation which may take time. An `F`-value of 2 means any individual fitness can never exceed 2, but may save computation time. Recall that the population is normally close to ecological equilibrium, which means most fitness-values are normally close to 1.

There are a number of fitness modules to choose from, and they can all be combined with each other.

Stabilizing_selection

This module puts a trait under stabilizing selection towards a given `optimal_value`. It can be used to keep trait evolution within reasonable bounds, to introduce a source of deleterious mutations (it the trait is otherwise neutral), or a mechanism for evolving incompatibilities. Fitness is calculated as

$$f = \exp(-c\sum_d \left|z_d - z^*\right|^e)$$

where z is the trait under selection, z^* is the optimal value, and the sum is across the dimensions of z. The positive parameters c (`cost_coefficient`) and e (`cost_exponent`) determine the shape and strength of the selection.

Syntax example:

```
Fitness : Stabilizing_selection
trait : z # the name of the trait under selection
optimal_value : 1
cost_coefficient : 0.5
cost_exponent : 2 # a value 2 gives a quadratic cost function
```

If the trait under stabilizing selection has no other function it is expected to quickly evolve to the `optimal_value` and stay there, save for drift. Disadvantageous mutations may become fixed by drift, but are expected to be canceled out by compensating mutations, possibly at other loci coding for the same trait. In this way, two isolated populations may evolve unique genetic solution to the same optimal phenotype, corresponding to co-adapted gene complexes. If the two populations do interbreed, the F2 offspring will vary extensively in phenotype and have low average fitness. It is in this way possible to introduce a mechanism of evolving genetic incompatibilities.

Resource_landscape

This module is based on the well-studied model of a continuous resource landscape, modeled as a Gaussian carrying capacity function. An evolving trait dictates the amount of resources available to an individual. The peak of the resource landscape, i.e. the maximal amount of available resources, corresponds to a trait value $k_s x_1$, where x_1 is an individual's position in space in *the first spatial dimension* and k_s is a spatial gradient parameter. If k_s is zero, the peak of the resource landscape always corresponds to a trait value of zero. If k_s is non-zero, the local resource peak Competition between individuals depends on their proximity in trait space as well as actual space (if specified). The fitness of individual *i* is given by

$$f_i = 1 + r\left(1 - \frac{\sum_j \exp\left(-\left(\frac{\sum_d\left(u_{jd} - u_{id}\right)^2}{2\sigma_a} + \frac{\sum_p\left(x_{jp} - x_{ip}\right)^2}{2\sigma_s}\right)\right)}{K_0 \exp\left(-\left(\frac{\sum_d\left(u_{id} - k_s x_{i1}\right)}{2\sigma_K^2}\right)\right) F}\right)$$

(truncated below at zero). The sum over *j* is taken across all individuals, the sums over *d* are across the dimensions of trait *u*, and the sum over *p* is across all dimensions of space (if any).

The fecundity parameter *F* is included to rescale the density dependence such that the equilibrium *adult*, post survival selection, population is controlled by K_0 and independent of *F*.

The parameters *r*, σ_a , K_0, and σ_K , σ_s (`s_space`) and k_s (`k_space`) have to be specified in the syntax:

```
Fitness : Resource_landscape
trait : beak_size # for example
r : 1
K_0 : 100
s_K : 1
s_a : 0.4
s_space : 1 # spatial extent of competition (all spatial dimensions)
k_space : 0 # spatial gradient of resource optimum
```

Density_dependence

This module introduces density dependence localized in space. Individual fitness is given by

$$f_i = \exp\left(r\left(1 - \frac{\sum_j e^{-\frac{d_{ij}^2}{2\sigma_s^2}}}{KF}\right)\right),$$

where the sum is across all individuals and d_{ij}^2 is the squared euclidian distance between individuals *i* and *j*, $d_{ij} = \sum_p\left(x_{ip} - x_{jp}\right)^2$ (x_{ip} is the position of individual *i* in spatial dimension *p*). Syntax:

```
Fitness : Density_dependence
r : 1
K : 500
s_space : 2 # s_s
```

The parameter σ_s (`s_space`) controls the spatial extent of competitive interactions. A large σ_s implies long-range competition and smaller local population sizes. The total population also scales directly with the carrying capacity parameter *K*.

The fecundity parameter *F* is included to rescale the density dependence such that the equilibrium *adult*, post survival selection, population is controlled by *K* and independent of *F*.

Note: In discrete space it is possible to set $\sigma_s = 0$, implying within-patch competition only. *It will also speed up computation times considerably*. A value $\sigma_s > 0$ implies at least some amount of competition between individuals in different patches and much longer computation times.

Discrete_resources

This is a model of n_R discrete resources with explicit (fast) dynamics according to a chemostat model. Individual consumption rate is controlled by an ecological trait. *The module is only available for discrete space models or models without space.*

Resource dynamics of resource *j* within a single patch occur on a fast time-scale, following

$$\frac{dR_j}{dt} = \mathrm{K} - R_j - \sum\nolimits_j a_{ij} R_j\,, \qquad j = 0..(n_R - 1),$$

where the sum is across all individuals in the patch. Notice that the index of the first resource is 0. The parameter *K* is a scaling parameter, setting the overall size of the system. The attack rate of individual *j* on resource *i* is

$$a_{ij} = \frac{a_0}{K} e^{-\frac{(u_j - i)^2}{2\sigma_a^2}}$$

where u_j is the assigned (one-dimensional) trait, a_0 dictates the maximal attack rate, and σ_a is the niche-width of the consumer. A trait value $u = i$ makes an individual specialized on resource i.

The equilibrium resource abundance evaluates to

$$R_i^* = \frac{K}{1 + \sum_j a_{ij}\,/F}\,.$$

The fecundity parameter *F* is included to rescale the density dependence such that the equilibrium *adult*, post survival selection, population is controlled by *K* and independent of *F*.

The fitness of an individual is evaluated in discrete time, assuming all resources are at equilibrium, according to

$$f_j = 1 + \sum\nolimits_i a_{ij} R_i^* - c_{min}\,,$$

where c_{min} is the minimal amount of resources required for an individual to replace itself.

The equilibrium population size of a resource specialist feeding on a single resource in a single patch can be calculated to

$$N^* = K\left(\frac{1}{c_{min}} - \frac{1}{a_0}\right),$$

which confirms the scaling property of *K* and sets bounds on c_{min} and a_0 for population persistence.

Syntax example:

```
Fitness : Discrete_resources
trait : X # any previously specified trait
n_R : 3 # the number of resources
K : 2000 # scaling constant
a_0 : 4 # maximal attack rate. Too high values leads to unstable
population dynamics
s_a : 0.6 # consumer niche width
c_min : 1 # minimal intake rate for self-replacement
```

Spatial_gradient

This module applies selection towards an optimal phenotype, that varies linearly in space. It operates only along the first spatial dimension and the first dimension of the associated trait. The fitness function is

$$f_i = e^{-\frac{(u_i - k_s x_i)^2}{2\sigma_s^2}},$$

where u_i is the trait value and x_i is the position of individual i, respectively. The parameter σ_s (`s_selection`) controls the strength of selection and k_s (`k_space`) is the steepness of the spatial gradient.

This module works in both discrete and continuous space. The discrete space position is simply the index of the patch (0, 1, 2, …).

Syntax example:

```
Fitness : Spatial_gradient
trait : B # trait for local adaptation
k_space : 1 # steepness of spatial gradient
s_selection : 1 # spatial niche width
```

Catastrophes

The `Catastrophes` module introduces global mortality events at random intervals. It has two parameters:

`P_catastrophe` is the probability there will be a catastrophe in any given year.

`P_survive` is the probability that an individual will survive a catastrophe.

Syntax example:

```
Fitness : Catastrophes
P_catastrophe : 0.001
P_survive : 0.01
```

Mating

Each reproduction event starts with mate selection. Each individual chooses a mate from the assigned *mating pool* according to its *mating preference* modules. More explicitly, a choosing individual is presented candidate mates randomly picked from the specified mating pool. It will accept a mate with a probability set by its mating preferences. If the candidate mate is rejected, a new candidate is picked randomly (with replacement), and so on until a mate is accepted or a maximal number of `mating_trials` candidates have been rejected. If all candidates are rejected, the choosing individual will not reproduce unless it is chosen as a mate by another individual.

The mating section of a model description includes at least two parameters

```
Mating_pool : [one of Selfing, Local, Global]
Mating_trials : [an integer > 0]
```

followed by an optional list of `mating_preference` modules.

The `Mating_pool` can be one of three choices:

`Selfing`: This is the simplest mating pool, and quite self-explanatory (non pun intended…). All individuals mate with themselves, but the mechanics of gamete formation, recombination and zygote formation still operate. No parameters are required. This module invalidates all specified `mating_preference` modules. Syntax:

```
Mating_pool : Selfing
```

`Local`: The random pick of candidate mates is weighted depending on their proximity in space, using a Gaussian weight function with width parameter σ_s (`s_space`). The weight of a partner at Euclidian distance *d* in space is $\exp\left(-\frac{d^2}{2\sigma_s^2}\right)$. This option requires a specification of the extra parameter σ_s. Syntax example:

```
Mating_pool : Local, s_space : 0.5
```

If space is discrete and $\sigma_s = 0$ (`s_space : 0`), mates are chosen randomly from the local patch, which is computationally more efficient than the general case.

`Global`: A global mating pool means that mating candidates are chosen randomly from the entire population, irrespective of proximity in space. Syntax:

```
Mating_pool : Global
```

The `Global` and `Local` mating pools are equivalent if no spatial module is defined.

The `Mating_pool` and `Mating_trials` parameters are followed by an optional list of `Mating_preference` modules. If more than one module is specified, they are applied in the order they are specified. The modules are multiplicative, such that the weight of any given mating candidate is the product of weights from all `Mating_preference` modules. The actual mate is then chosen randomly according to the combined weights. Selfing is allowed, i.e. an individual may choose itself as a mate.

There is currently only one type of `Mating_preference` module to choose from, the `target_selection` module, but it is highly versatile. The general idea (from

Thibert-Plante & Gavrilets 2013) is that each individual has a heritable *display* trait and a heritable *preference* trait. Mate selection is based on matching the preference with the display. The target and preference traits may be the same trait, which generates assortative mating – individuals tend to choose mates similar in phenotype to themselves. A third trait regulates the *strength* of partner selection, i.e. the choosiness. A negative strength trait implies disassortative mating, i.e. that mates dissimilar in phenotype are preferred. A strength of zero implies random mating. All three traits (*display*, *preference*, *strength*) can be multi-dimensional, but must have the same number of dimensions.

If we label the target trait *x*, the preference trait *y*, and the strength trait *c*, the probability that individual *i* will accept individual *j* is

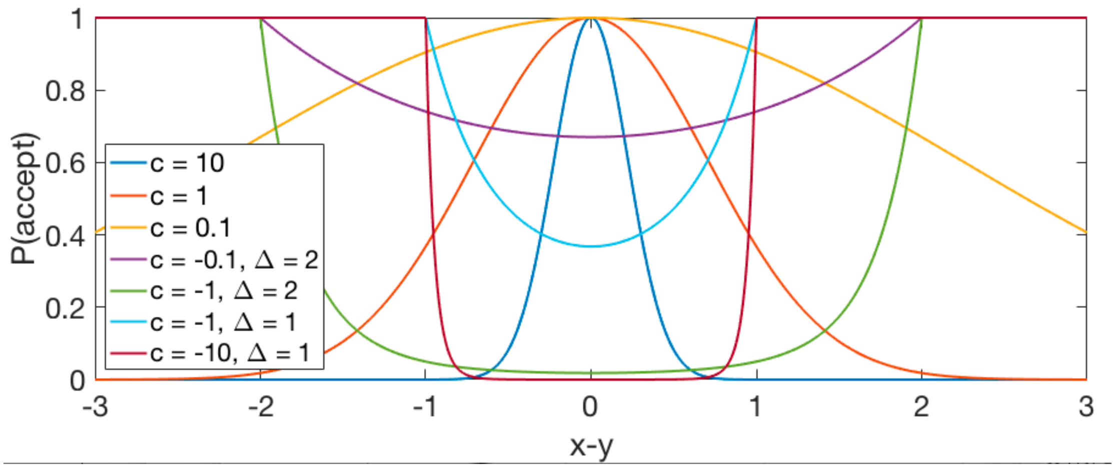
$$P_{ij} = \min\left(1, exp\left(-\sum_d p_d\right)\right),$$

where the sum is across the dimensions of *x*, *y* and *c* and

$$p_d = c_{id}\left(x_{jd} - y_{id}\right)^2, \qquad c_{id} > 0$$

$$p_d = \max\left(0, c_{id}\left(\left(x_{jd} - y_{id}\right)^2 - \Delta^2\right)\right), \qquad c_{id} < 0.$$

The additional parameter Δ (`Disassortative_limit`) is only used if there is disassortative mating, i.e. if the strength (c) is negative. It sets the phenotypic distance beyond which all mates are accepted. A few example preference functions are drawn below.



A couple of examples, that also illustrate the syntax, are given below.

```
Mating_preference : Target_selection
     Display : beak_size
     Preference : beak_size
     Strength : C
     Disassortative_limit : 1
```

If `beak_size` is associated with a fitness module, the above could be a magic trait model. The trait `C` could be an evolving trait or a constant trait.

Here's a more complete example with independently evolving target and preference traits, as well as an evolving strength:

```
Trait : Color
     dimensions : 3
     loci_per_dim : 10
     initial_value : 0
Trait : Color_Preference
     dimensions : 3
     loci_per_dim : 10
     initial_value : 0
Trait : Color_Choosiness
     dimensions : 3
     loci_per_dim : 10
     initial_value : 0

# other traits / modules ...

Mating_pool : Local
     s_space : 1 # set this to 0 for within-patch mating
Mating_trials : 1000

Mating_preference : Target_selection
     Display : Color
     Preference : Color_Preference
     Strength : Color_Choosiness
     Disassortative_limit : 1
```

The above creates a three-dimensional ‘color’ trait with associated preference and choosiness traits. If none of the traits is associated with a *Fitness* module, this is a pure sexual selection model.

More than one `Mating_preference` module can be specified. The probability to choose a mate is then simply the product of all probabilities generated by the different modules. A fixed strength of assortative mating on one trait can in this way be combined with sexual selection on another trait. Alternatively, assortativeness can evolve in parallel with sexual selection (cf. Thibert-Plante & Gavrilets 2013). The two examples above can thus be combined in the same model:

```
Mating_preference : Target_selection
     Display : beak_size
     Preference : beak_size
     Strength : C
     Disassortative_limit : 1

Mating_preference : Target_selection
     Target : Color
     Preference : Color_Preference
     Strength : Color_Choosiness
     Disassortative_limit : 1
```

Running simulations

Simulations are typically issued from a command prompt. The *TREES* executable should be in the search path or the current directory. The parameter file should be in the current directory, which is also where all output is saved. The command is simply `TREES` or `./TREES` (see installation above) followed by the name of the parameter file:

```
TREES myModel.txt
```

Optionally, two integers can be added to specify indices of repeated simulations:

```
TREES myModel.txt 11 20
```

The above command will run 10 replicate simulations which will be given indices from 11 to 20. The output of each replicate is saved in its separate output file. The file name consists of the name of the parameter file, appended with '_results_', the index of the simulation, and an extension '`.sim`'. The last example above would generate output files `myModel_results_11.sim, myModel_results_12.sim, myModel_results_13.sim`, etc. If no indices are specified, as in the first example, the single simulation is given index 1.

The TREES command also takes an option `-v` ('verbose') which gives extra output during the simulation run, including current population size at each sampling point as well as estimated finishing time. Example:

```
TREES -v test.txt
```

It is wise to make short test-runs, with perhaps a few hundred generations, to get a feeling for the time required. Remember that simulation time increases as the number of individuals increases, often at a quadratic rate. If total population size is expected to increase by a factor 10, the simulation may be 100 times slower towards the end! A simulation can at any point be interrupted with Ctrl+C.

It is also wise to let the name of the parameter file reflect the parameter values that may vary between simulations. This is to avoid over-writing earlier results, but also to assist sorting output files. Alternatively, different settings can be run in different directories.

Output

Each simulation is saved in a separate file with a hopefully unique name. The file name starts with the name of parameter file, followed by '_results_', the index of the replicate, and the extension '`.sim`'. For example, the command

```
TREES Model8.txt 1 2
```

would generate two output files: '`Model8_results_1.sim`' and '`Model8_results_2.sim`'.

File format

The output files are in binary format. Please refer to the Plotting and Analysis section below if all you need to do is analyzing results.

All integers are stored as 32 bit, unless otherwise specified, all floats are single precision (32 bit)). The sequence of data is:

1. File version (integer)
2. Simulation seed (64 bit unsigned integer)
3. A complete copy of the parameter file text, ended by a zero byte.
4. The number of loci (integer)
5. The number of microsamples (integer equal to 0 or the number of generations)
6. All micro-samples, in chronological order, if any (see below).
7. The number of samples (integer)
8. All samples in chronological order (see below)
9. Gene tracking data, if Gene_tracking is activated (see Gene Tracking below)
10. Checkpoints, if applicable.

Each micro-sample is structured as:

1. The microsample option, a single character, one of {'m', 'v', 'c'}.
2. The population generation (64 bit integer)
3. The cpu-time of the sample (floating point double, 8 bytes)
4. The population size (integer)
5. The actual data, depending on the chosen option (point 1):

    m: a *vector* (see below) of means, one per trait dimension and spatial dimension. Trait means are given in the order the traits are defined in the parameter file.

    v: first a *vector* of means (as above), followed by a *vector* of variances, one per trait dimension and spatial dimension.

    c: first a *vector* of means, followed by a *vector* of covariances, one per *pair* of trait dimension and spatial dimension. These come in the order (1,1), (1,2),…(1,*n*), (2,2), (2,3),…,(*n*,*n*), where *n* is the total number of trait and space dimensions.

    Each *vector* above is stored as a starting integer giving the number of elements, followed by the actual data as sequence of floats.

Each sample is structured as:

1. The generation number, starting at 0 (64 bit integer)
2. Population size (integer)
3. Complete genomes of all individuals, if Gene_sampling is on. For the Diallelic module, the alleles are saved as bits (mapping -1/2 to 0 and +1/2 to 1). For the Continuous_alleles model, these are saved as. The first half genome is saved for each individual before the second, analogous, half is saved for each individual.
4. Gene tracking sample, if Gene_tracking is activated (see Gene Tracking below).
5. All traits, one at a time. Each traits is saved simply as the phenotypic values of all individuals (floats). Multivariate traits are saved with all dimensions together ((individual 1, dimension 1), (individual 1, dimension 2), (individual 2, dimension 1), and so on).
6. Spatial information, if applicable. The spatial position (float) is saved analogous to a trait.

Gene Tracking

The program can keep track of all genes (or, more correctly, alleles) and their descendants. This is activated by setting the parameter `gene_tracking` to 'Y' or 'Yes' (see 'Simulation parameters' above). Gene tracking makes it possible to draw separate phylogenies of all genes present in the final population, or to calculate coalescence times of separate loci. The gene tracking data is continuously pruned of extinct, non-sampled, lineages to save computer memory and file size.

Each gene is given a unique ID number at its first appearance, which is either in generation 0 or following a mutation. The ID is stored in a table, together with the gene's phenotypic effect, the ID of the parent and the time of birth (in generations). There is one separate table per locus.

The gene tracking data is stored at the end of an output file as a sequence of lists, one list per locus. Each list contains

1. List length (integer)
2. Each gene in the list, containing:
    a. `id`: a unique number (64 bit integer)
    b. `parent`: the parent id (64 bit integer)
    c. `birth`: the generation of first appearance (64 bit integer)
    d. `death`: the generation of last appearance (64 bit integer)
    e. `effect`: its genetic effect (float)
    f. `children`: the number of (sampled) descendants (integer)
    g. `child_list`: all id:s (if any) of the immediate descendants (64 bit integers)

The first gene in the list is the root, from which all other descend. It has id = 1 and parent = 0. The generation of last appearance is constrained to sample events. It is only during sampling that gene presence/absence is checked in the population.

All population samples also contain gene tracking data. It occurs at the end of the sample, analogous in structure to the Gene_sampling data but with all entries 64 bit integers. This is simply the id:s of all genes currently present in the population.

Resuming simulations

The *resume* feature makes it possible to resume a simulation at saved checkpoints. This feature can be used to

- Continue a simulation from the last saved generation.
- Re-run a part of a simulation with increased sampling frequency
- Resume a simulation at a particular point in time with modified *fitness* or *mating* modules.

Syntax:

```
TREES -resume parameter_file.txt results_file.sim starting_generation
     new_results_file.sim
```

Example:

```
TREES -resume beak_size_model2.txt  beak_size_model_results_1.sim
     2000  beak_size_model2_results_0.sim
```

The parameter file does not have to be the same parameter file as was used to generate the results in the first place. You can thus re-run a simulation from any checkpoint with a new model specification *as long as the genetics and space modules and all traits are the same*. Any fitness or mating modules can be altered or removed and new ones can be added (as long as they use the available traits).

The `starting_generation` parameter is optional. If it is not specified, or set to -1, TREES will resume from the last saved checkpoint.

The `new_results_file.sim` is also optional. By default, TREES will save results to the specified `results_file.sim`, over-writing whatever was stored before.

The resumed simulation will continue until `t_max` generations, as specified in `parameter_file.txt`.

Resuming a simulation with the same model description will keep the random number generation consistent with the original seed. The result should thus be exactly the same as the first run. This can be used to re-run a particular part of a simulation with increased sampling frequency, for instance.

The `-resume` option can be combined with the `-v` (verbose) option, in any order.

Plotting and analysis (Matlab)

TREES includes a toolbox of Matlab functions for reading and analyzing simulation data. All functions described below can be found in the TREES_Matlab folder at github. Download the files to a suitable directory, preferably on your Matlab path.

Using these functions efficiently requires some basic knowledge of Matlab, but several examples are provided below.

read_sim

The basic Matlab program to read sim-files is `read_sim.m`. It takes two parameters – the name of the parameter file (not the results file) and the index of the simulation. The output is a struct containing all available information about the simulations. As an example, consider a parameter file m1.txt like this:

```
#Simulation parameters:
t_max : 5000, sample_interval : 100
microsamples : n
checkpoint_interval : 0, keep_old_checkpoints : N
seed : R, gene_tracking : N, gene_sampling : N
# Population parameters
F : 2, n_0 : 100

# Genetics
Genetics : Continuous_Alleles, P_mutation : 1e-4

# Traits
Trait : X, dimensions: 1, loci_per_dim: 10, initial_value: -20
     Transform: linear, offset: 0, scale: 0.05

# Fitness
Fitness : Stabilizing_selection, trait: X
     optimal_value : 0
     cost_coefficient : 0.1
     cost_exponent : 2
Fitness : Density_dependence, r : 1, K : 1000, s_space : 0

# Mating
Mating_pool : Global
Mating_trials : 100
```

The model has a single trait X coded by 10 continuous alleles loci and subject to stabilizing selection towards zero. Density dependence is added to control population size. Some sort of density dependence has to be included. Fitness will otherwise never exceed 1 and the population will rapidly go extinct.

On a standard computer, such a short simulation runs in a few of seconds. The simulation can be run using

```
TREES m1.txt 1
```

which generates an output file 'm1_results_1.sim' which can be read into Matlab using

```
>> sim1 = read_sim('m1.txt',1)
```

All data is stored in the struct sim1:

```
>> sim1

sim1 = 

  struct with fields:

                  name: 'm1'
                  seed: 1579790779407228
                 t_max: 5000
       sample_interval: 100
   microsamples_option: 'n'
   checkpoint_interval: 0
  keep_old_checkpoints: 'N'
         gene_tracking: 0
         gene_sampling: 0
                     F: 2
                   n_0: 100
              Genetics: [1×1 struct]
                Traits: [1×1 struct]
                 Space: [1×1 struct]
               Fitness: {[1×1 struct]  [1×1 struct]}
                Mating: [1×1 struct]
                  loci: 10
          microsamples: []
          sample_count: 51
               samples: [1×51 struct]
                 stats: [1×1 struct]
```

All modules are included in the struct, including all parameters. The `Traits` field is a vector of structs, one per trait (only one in this case). Each trait struct contains the parameters of that specific trait:

```
>> sim1.Traits

ans = 

            name: 'X'
            dims: 1
    loci_per_dim: 10
   initial_value: -20
      transforms: {[1x1 struct]}
```

which in turn contains a list of the transforms associated with that trait. In a similar manner, all modules are represented in the struct.

All population samples are contained in the `samples` field, which is a vector of structs, one per sample. A sample of this example model looks like this:

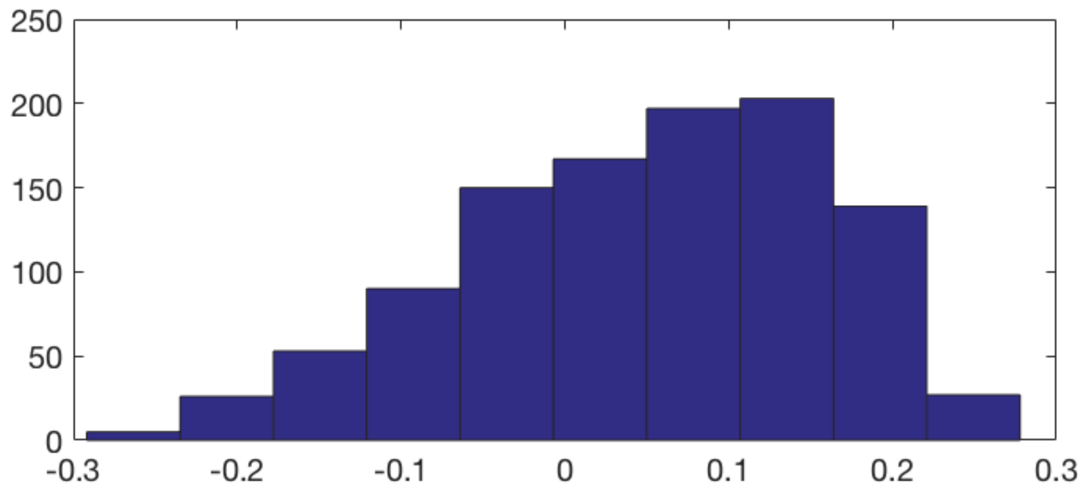
```
>> sim1.samples(1)

ans = 

           gen: 0
       cputime: 0
   sample_size: 100
             X: [1x100 double]
```

The first field is the generation number, followed by the cpu clock-time, followed by the current size of the population. Next, each trait is stored as a separate array, named after the trait name specified in the parameter file. The last sample of this simulation looks like this:

```
>> sim1.samples(51)

ans =

        gen: 5000
    cputime: 1.7991
       size: 972
          X: [1×972 double]
```

which means the population had grown to 972 individuals after 5000 generations. We can plot the final distribution of phenotypes in a histogram:

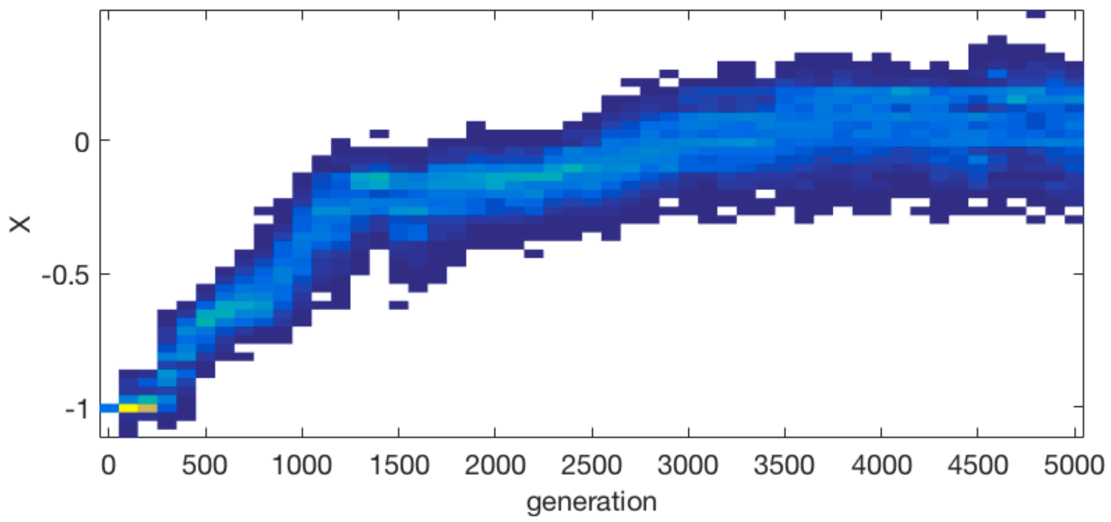
```
>> figure
>> hist(sim1.samples(51).X)
```



plot_sim

To plot the phenotypic evolution over time, us the `plot_sim.m` function instead:

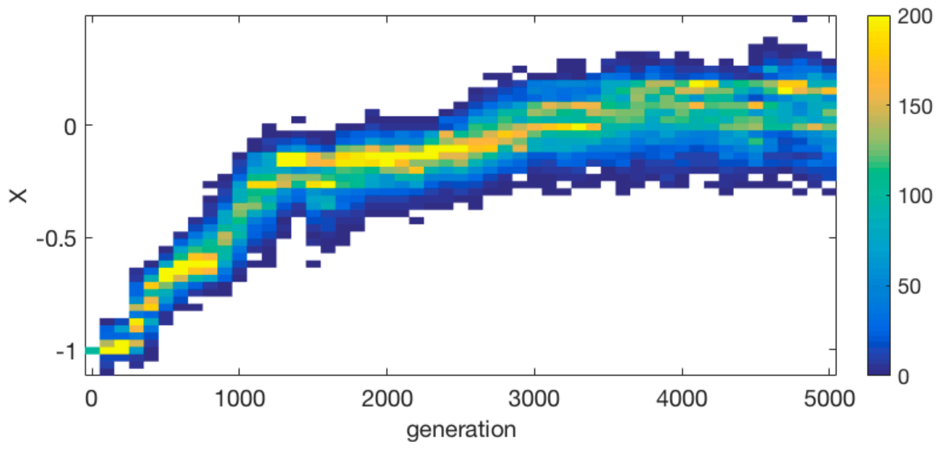
```
>> plot_sim(sim1)
```



The result is a heat map of the phenotypic evolution over time. The colors correspond to the number of individuals within each grid cell. Use `caxis` to change the color scale and `colorbar` to see the scale:

>> colorbar

```
>> caxis([0 200])
```



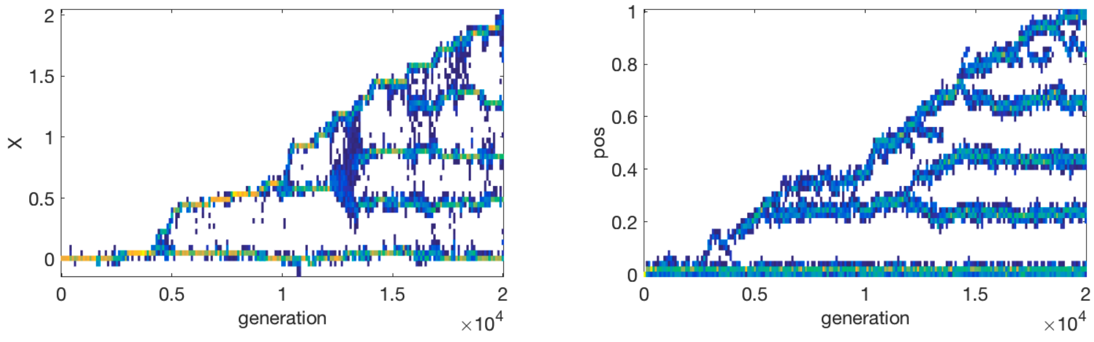
It can be seen that the X trait converges to the optimal phenotype X = 0. Genetic variation is maintained through continuous mutations and drift.

A more elaborate example is given below (m2.txt). The model is expanded with a one-dimensional continuous spatial structure and the stabilizing selection is given a gradually shifting optimum in space. Individuals compete and mate with neighbors in space.

```
#Simulation parameters:
t_max : 20000, sample_interval : 100
microsamples: n
checkpoint_interval : 0, keep_old_checkpoints: N

seed : R, gene_tracking : N, gene_sampling : N
# Population parameters
F : 2, n_0 : 100

# Genetics
Genetics : Continuous_Alleles, P_mutation : 1e-4

# Traits
Trait : X, dimensions: 1, loci_per_dim: 10, initial_value: 0
     Transform: linear, offset: 0, scale: 0.02
Trait_constant : PD, dimensions: 1, initial_value : 1

#Space
Space : Continuous, size: 1, dimensions : 1
     P_disperse : PD, dispersal_distance : .001
     boundary : A, initial_position : 0

# Fitness
Fitness : Spatial_gradient, trait : X
     k_space : 2
     s_selection : 0.3
Fitness : Density_dependence, r : 1, K : 100, s_space : .1

# Mating
Mating_pool : Local, s_space : .05
Mating_trials : 100
```

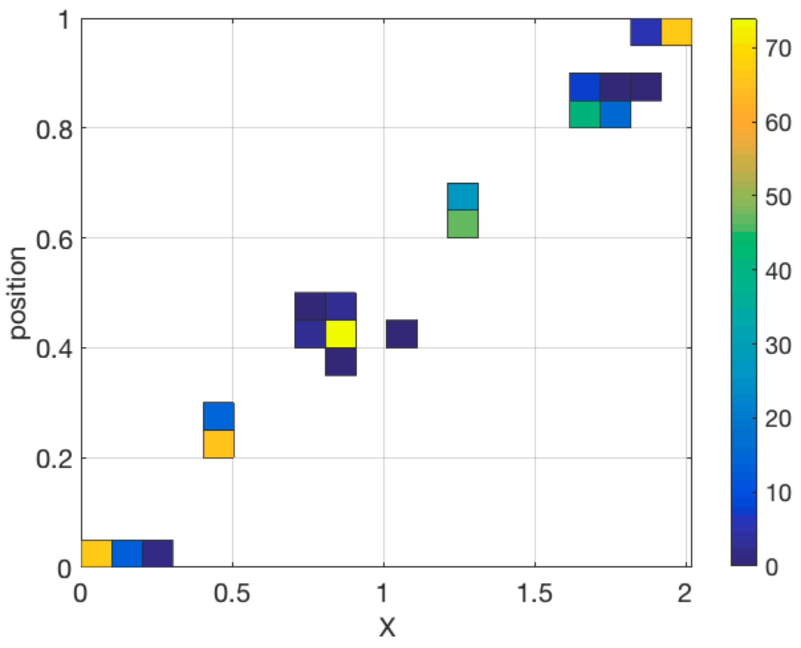
A sample simulation (`./TREES m2.txt 1`) looks like this:

```
>> sim2 = read_sim('m2.txt',1);
>> figure, plot_sim(sim2)
```



Both the trait X and the spatial position (‘pos’) are plotted across time. The population spreads and diversifies in space. One can easily identify clusters, both in phenotype space and actual space. The spread in space goes hand in hand with local adaptation in X. Clusters are fairly reproductively isolated due to spatial separation and selection against maladapted migrants. The structure of the last generation can be plotted like this to visualize the clustering and strong correlation between spatial position and trait value:

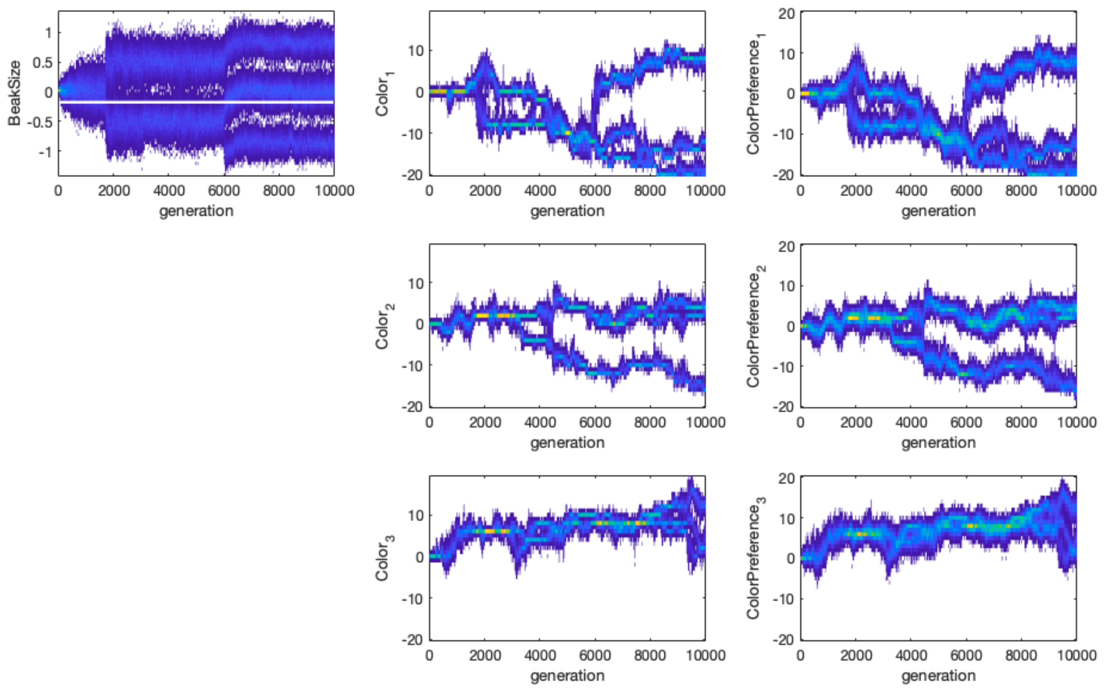
```
>> sa = sim2.samples(end);
>> figure
>> histogram2(sa.X, sa.pos, 20, 'DisplayStyle','Tile')
>> xlabel('X'), ylabel('position'), colorbar
```



A third example demonstrates the versatility of the mating modules. The model organism has one ecological trait (`BeakSize`) controlling its utilization of a continuously distributed resource (the Gaussian resource landscape model). It also has two ecologically neutral traits involved in sexual selection – a three-dimensional display trait (`Color`) and a likewise three-dimensional preference trait (`ColorPreference`). This is also a demonstration of the `Diallelic` genetics module.

```
#Simulation parameters:
t_max : 10000
sample_interval : 25
microsamples: n
checkpoint_interval : 0, keep_old_checkpoints: N
seed : R, gene_tracking : N, gene_sampling : N

# Population parameters:
F : 2 # Fecundity parameter
n_0 : 100 # initial population size

# Specify a Genetics module:
Genetics : Diallelic, P_mutation : 1e-4

# Specify traits:
Trait : BeakSize, dimensions: 1, loci_per_dim: 40, initial_value: 0
     Transform: linear, offset: 0, scale: 0.05

Trait : Color, dimensions: 3, loci_per_dim: 20, initial_value: 0
Trait : ColorPreference, dimensions: 3, loci_per_dim: 20,
initial_value: 0
Trait_constant : Choosiness, dimensions: 3, initial_value: 0.4

# Fitness:
Fitness : Resource_landscape, trait : BeakSize
     r : 1, K_0 : 500, s_K : 1, s_a : 0.4, s_space : 0, k_space : 0

# Set the mating parameters:
Mating_pool : Global
Mating_trials : 1e3
Mating_preference : target_selection
     Display : Color
     Preference : ColorPreference
     Strength : Choosiness
     Disassortative_limit : 1 # not used, since Choosiness is > 0
```

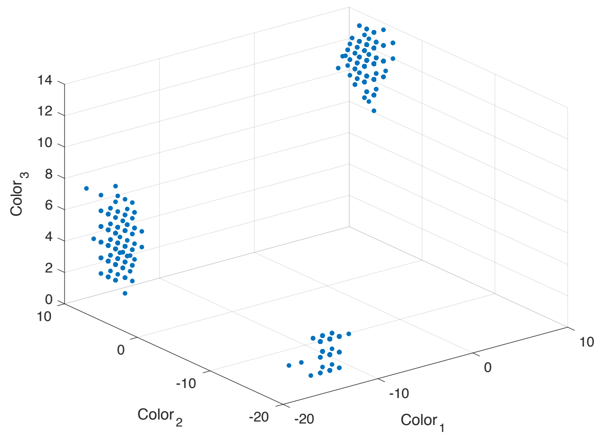
The resulting simulation can be studied using plot_sim:



The multi-dimensional traits are displayed with one panel per dimension, stacked on top of each other.

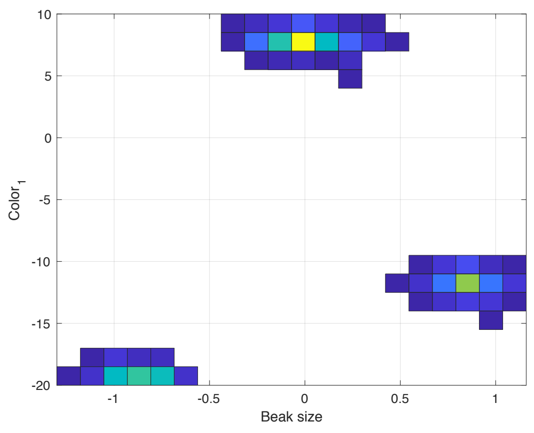
At the end of the simulation there are three reproductively isolated species, well separated in trait space. The final distribution of Color-morphs can be plotted as:

```
>> sa = sim3.samples(end);
>> plot3(sa.Color(1,:), sa.Color(2,:), sa.Color(3,:), '.',
'markersize',12)
>> grid on
>> xlabel('Color_1')
>> ylabel('Color_2')
>> zlabel('Color_3')
```



One can also plot the final distribution of BeakSize and Color$_1$:

```
>> histogram2(sa.BeakSize, sa.Color(1,:), 20, 'DisplayStyle','Tile')
>> xlabel('Beak size')
>> ylabel('Color_3')
```



It is apparent that the three color-morphs also correspond to three ecologically separated species.

Genetic analysis and seeding the random number generator

The last simulation in the previous section was run without saving any genetic data. If we want to analyze the genetics, we can re-run it using the same random number seed. This will reproduce exactly the same simulation. The seed is stored in the output file and can be obtained from the sim struct in Matlab:

```
>> uint32(sim3.seed)

ans =

           4042805153
```

We copy that seed to the parameter file and turn gene_tracking and gene_sampling on. To save disk-space, the sampling is also less frequent. All changes are highlighted in yellow below:

```
#Simulation parameters:
t_max : 10000
sample_interval : 500
microsamples: n
checkpoint_interval : 0
keep_old_checkpoints : N
seed : 4042805153, gene_tracking : Y, gene_sampling : Y

# Population parameters:
F : 2 # Fecundity parameter
n_0 : 100 # initial population size

# Specify a Genetics module:
Genetics : Diallelic, P_mutation : 1e-4

# Specify traits:
Trait : beak_size, dimensions: 1, loci_per_dim: 40, initial_value: 0
     Transform: linear, offset: 0, scale: 0.05

Trait : Color, dimensions: 3, loci_per_dim: 20, initial_value: 0
Trait : ColorPreference, dimensions: 3, loci_per_dim: 20,
initial_value: 0
Trait_constant : Choosiness, dimensions: 3, initial_value: 0.4

# Fitness:
Fitness : Resource_landscape, trait : beak_size
     r : 1, K_0 : 500, s_K : 1, s_a : 0.4, s_space : 0, k_space : 0

# Set the mating parameters:
Mating_pool : Global
Mating_trials : 1e3
Mating_preference : target_selection
     Display : Color
     Preference : ColorPreference
     Strength : Choosiness
     Disassortative_limit : 1 # not used, since Choosiness is > 0
```

The simulation result is exactly the same, albeit sampled less frequently. The sample of the final generation now looks like:

```
>> sa = sim3.samples(end)
```

```
sa =

                gen: 10000
            cputime: 109.0374
               size: 1045
                 G1: [160×1045 int8]
                 G2: [160×1045 int8]
               G1id: [160×1045 uint64]
               G2id: [160×1045 uint64]
          BeakSize: [1×1045 double]
             Color: [3×1045 double]
   ColorPreference: [3×1045 double]
```

The fields G1 and G2 contain the entire genomes of all individuals. There are in total 160 loci, associated to the traits in the order they appear in the parameter file. The first trait BeakSize is thus controlled by the first 40 loci, the following 60 (3×20) code for Color and the last 60 code for ColorPreference. The fields G1id and G2id contain id:s of the corresponding alleles.

Due to the separation in $Color_1$ and BeakSize at the end (see plot above) it is straightforward to single out the three ecomorphs of that sample as logical vectors:

```
>> morph1 = sa.BeakSize<0 & sa.Color(1,:) < 0;
>> morph2 = sa.Color(1,:) > 0;
>> morph3 = sa.BeakSize > 0 & sa.Color(1,:) < 0;
```

The morphs (or species) are numbered in BeakSize order, smallest to largest, which can also be confirmed:

```
>> mean(sa.BeakSize(morph1))
ans =
    -0.8776

>> mean(sa.BeakSize(morph2))
ans =
     0.0048

>> mean(sa.BeakSize(morph3))
ans =
     0.8516
```

F_ST

We can now calculate the F_{ST}-values of, for instance, the loci coding for BeakSize using the `F_ST.m` function:

```
>> F_ST(sa, morph1, morph2, 1:40)
ans =
     0.7010

>> F_ST(sa, morph1, morph3, 1:40)
ans =
    0.8005

>> F_ST(sa, morph2, morph3, 1:40)
ans =
     0.6959
```

The F_ST function returns an averaged F_{ST} following Weir & Cockerham (1984). A second output argument contains the F_{ST} for each locus, if required. Apparently, morphs 1 and 3 are most separated, whereas the other genetic separations are more equal. This pattern may be at least partly artificial since there are only two alleles possible at each locus (we use the Diallelic genetics module). Identical alleles does not necessarily mean identical-by-descent – it may also be due to back-mutations or convergent mutations. The Continuous_alleles genetics module does not generate identical mutations, but F_{ST} may still be afflicted with considerable noise and biases (Whitlock 2011).

F_ST_coal

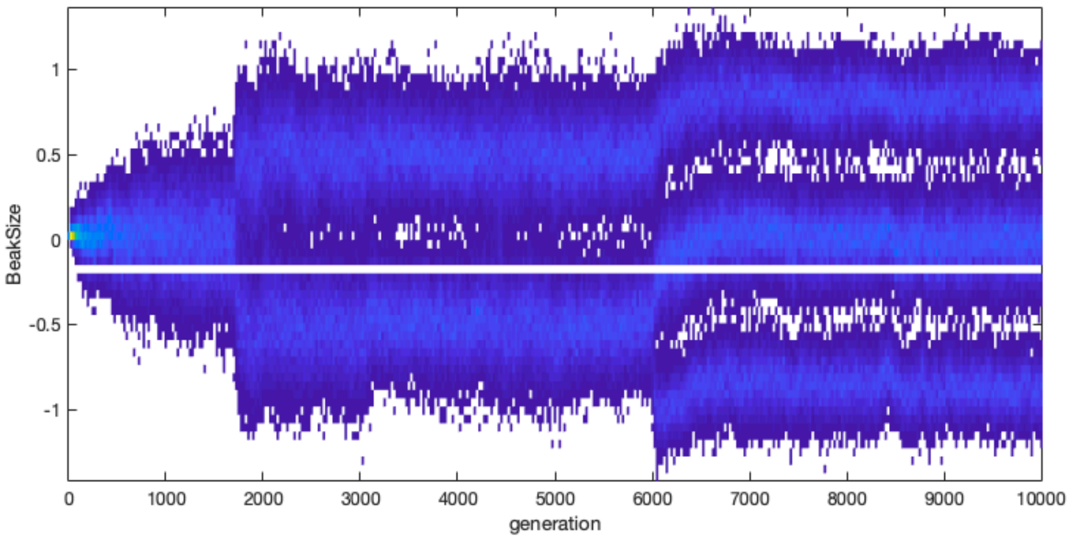
A better measure of evolutionary distance is the coalescence F_{ST} (Whitlock 2011). It compares the mean time to coalescence of two randomly sampled alleles within a group to the same measure between groups. A measure of 0 means there is no differentiation – coalescence times are the same within as between groups – and a measure of 1 implies all alleles are identical within a group but different between groups. Since gene-tracking is turned on, we can calculate it from the samples:

```
>> F_ST_coal(sa,sim3,morph1,morph2,1:40)
ans =
     0.6798

>> F_ST_coal(sa,sim3,morph1,morph3,1:40)
ans =
     0.9522

>> F_ST_coal(sa,sim3,morph2,morph3,1:40)
ans =
    0.8155
```

The results show that morphs 1 and 2 share more evolutionary history with each other than they do with morph 3 (with the largest beak). A close-up of the first, frequently sampled, simulation supports that conclusion (below), although the exact details can be hard to disentangle. A direct measure of the phylogenetic distance between morphs is given in the third output argument of `F_ST_coal`:

```
>> [~, ~, phylodist] = F_ST_coal(sa,sim3,morph1,morph3,1:40);
>> mean(phylodist)
ans =
6.3124e+03
```

This statistic points at a 6300 generation old separation of morph 1 and 3. The graph below shows that it is in fact older. The discrepancy can be due to gene flow as well as mutations occurring later than the demographic split of the populations. The coalescence F_{ST} as it is defined here only measures separation by mutation, not by demography.



References

Thibert-Plante, X. & Gavrilets, S. 2013. Evolution of mate choice and the so-called magic traits in ecological speciation. *Ecology Letters,* **16**:1004-1013

Weir, B. & Cockerham, C. 1984. Estimating F-statistics for the analysis of population-structure. *Evolution,* **38**:1358-1370

Whitlock, M. C. 2011. G'_{ST} and D do not replace F_{ST}. *Molecular Ecology,* **20**:1083-1091