ENHANCED BOTERO MODEL DOCUMENTATION

DION HÄFNER

September 29, 2015

CONTENTS

1 FIRST STEPS

1.1 *Installation*

To start, just clone the git repository containing this project to your local hard drive.

You will need to have Python 2.7 or 3.x installed, along with the usual scientific packages like NumPy, Matplotlib and Pandas. Additionally, parts of the project are written in Cython, so you will need to have this library installed as well. A convenient way to get all this is Anaconda Python, which is available for all major OS, and free of charge for non-commercial use. Another highly recommended package that is not included with Anaconda Python is Seaborn, which provides high-end data visualization. You can get this either from source or run

```
$ pip install seaborn
```

from your favorite Unix-style command line. Without Seaborn, all graphical output will be much less pretty, and violin plots are replaced by simple box plots.

If you cannot or do not want to install Anaconda Python, but have Python >= 2.7 and the package manager pip installed, you may run (with `sudo` as needed)

```
$ pip install -r requirements
```

in the main folder of this project. This should automatically install all the necessary requirements.

After you have installed the prerequisites, you will probably have to compile the Cython parts of the module once. To do this, execute

```
$ python setup.py build_ext --inplace
```

If none of the commands threw any errors you are ready to start simulating!

1.2 *Usage*

The goal with this package was to create a software that is both easy to use and easy to modify. The simplest work flow for running simulations with this code would be:

1. Customize the model parameters in the file `constants.py`, as defined in the array `_PARAMETERS` (or just keep the default values).
2. Call `python main_constant.py` (simulation with constant population size), and grab a cup of coffee.

3. The output will be found in a new folder `output` and will contain the mean genes of the populations and their standard deviation in every time step, and a detailed plot every few time steps (may be specified in the `constants.py`).

For runs with variable population size, you also need to specify two `.csv`–files, containing the mean genes of the starting population and their standard deviation (output of a run with constant population size). You have to pass the name of these files via the command line:

```
$ python main_variable.py <mean-file>.csv <std-file>.csv
```

Output is then being printed to the folder `output_var`.

You will probably want to run many different simulations while varying some parameters, and keeping the rest constant. To automate this task, it is more convenient to pass parameters from command line (e.g. via a shell script) than to change them in the `constants.py` file. Luckily, *every* parameter that can be specified in the `_PARAMETERS` array can be overwritten from command line, e.g. by calling

```
$ python main_constant.py --population_size 10000 --mu 0.0001
```

You can receive a full list of options by calling

```
$ python main_constant.py --help
```

1.3 *Making changes to the code*

Most of the code of the project is just regular Python code, which should be pretty easy to modify. However, there currently is also one file written in Cython (a module that allows to mix C and Python code), `animal.pyx`, for performance reasons. To have any of the changes to this file take effect you will have to compile it again by running

```
$ python setup.py build_ext --inplace
```

The following table gives a short overview on the purpose of each source file:

File	Description
constants.py	Sets model constants and parameters for other modules to import, and parses the command line. Change the default values here.
main_constant.py	Main wrapper to be called for a run with constant population size.
main_variable.py	Main wrapper to be called for a run with variable population size. Has two additional required command line parameters (mean and standard deviation of the start genes).
setup.py	Compiles the cythonized parts of the project. Needs to be called to have changes to .pyx files take effect.
example.py	Demonstrates the usage of the project's classes.
src/iterate_population.py	Called by the main wrapper. Operates on given Population and Environment instances. Main time step controller.
src/output_population.py	Handles file output and plotting.
src/environment.py	Implements Environment class.
src/animal.pyx	Implements Animal class (cythonized).
src/population.py	Implements Population class.

2 MODEL DESCRIPTION

This section is supposed to give a quick introduction to the model as proposed by Botero et al, and the modifications of the model that were done in this code (typeset in red, full list of modifications in 3). For a much more complete and detailed consideration refer to their original paper, [1].

2.1 *General purpose*

The Botero model simulates a population of a species and their genetic adaptation to their environment. A population consisting of N animals (usually about 5,000), each with their own set of genes, is generated and put into a periodic environment with certain parameters like frequency R, predictability P, amplitude A, and random noise B (for the additional parameter O see 3). Of special interest is the predictability P of the environment: The animals can always measure the current state of the environment, E, and a *cue* C, telling them how a future state may develop (which does not need to be true for $P < 1$ – or may even be anti-correlated for $1 \leq P < 0$). Each animal may develop a certain *strategy* concerning how to adapt their phenotype to the environmental cue, if at all.

Newly born animals inherit their parents' genes, with a chance of random mutation. In a first step, the total number of animals in the population is held constant – if too many children are born, some of them randomly die, and if too few are born, random entities are cloned. This has the purpose to quickly find the 'best' gene composition without risking the populations extinction with every evolutionary misstep. To make sure that the best genetic composition is eventually adapted throughout the population, the number of offspring of each animal is based on how well the animal has been adapted to its environment throughout its lifetime.

As soon as a set of *good* genes is found, it can be used as a starting condition for runs with a variable population size. In these runs, the environment the animals are put into may change. This is the main thing that Botero et al. examined: By which amount may the environment change in order to allow the population to continue its existence?

This modified version also features *multiple* environments, and allows for migration between them. This way, it can be examined whether migration may be a feasible solution for a species to prevent extinction, and how different families of the same species develop genetically in different environments.

2.2 *The genome*

The Botero species carries a set of seven (nine) genes. This genome encodes one phenotype of the species, called *insulation* I, and how this phenotype is adjusted depending on the environment the species lives in. Those genes, and the properties they encode, are:

s The plasticity of an animal – i.e., whether it is able to react to environmental clues at all ($s > 1/2$) or is ignoring them entirely ($s \leq 1/2$).

I_0 The baseline insulation of the animal (a constant offset when calculating the insulation I).

b How much a certain animal adjusts its insulation I after receiving an environmental clue C (is set to 0 if animal is not plastic, i.e., if $s \leq 1/2$).

a The probability of an animal to adjust its insulation I at each time step during its life time (is set to 0 if animal is not plastic, i.e., if $s \leq 1/2$).

h The probabiltiy of using I_0' and b' instead of I_0 and b to calculate the insulation (is set to 0 if animal is not plastic, i.e., if $s \leq 1/2$). This decision is only done at birth, and stays constant througout the lifetime of the animal.

I_0', b' Alternative set of genes determining the insulation I of the animal, chosen randomly at birth with probability h to allow for bet-hedging strategies.

m The likelihood that the animal roams to a different, randomly chosen environment (different from the one that it is currently in). This takes place right after birth or when m_a is triggered.

m_a The likelihood that the animal has a *chance* of roaming in every step of its lifetime (reversible migration).

The handling of genes is done in the file `animal.pyx`, e.g. through the method `Animal.mutate()`, which randomly mutates each gene with a probability μ by a mutation step drawn from a gaussian distribution with mean 0 and standard deviation 0.15 (currently hard-coded in the function).

2.3 *Animals*

An animal is created by passing a list of genes to the `Animal` class defined in the source file `animal.py` (or leaving the argument blank, resulting in random genes). The `Animal` class has multiple public methods:

`react(E, C, evolve_all=False)` Lets the animal react to the current environment state E / C (array-likes for multiple environments): lets the animal migrate, updates their insulation (based on m_a, or always right after birth), and calculates their mismatch.

`lifetime_payoff(positions)` Calculates the animal's lifetime payoff and returns it. `positions` is an array containing the number of animals in each environment, and the lifetime payoff is decreased linearly with increasing number of animals in the environment.

`mutate()` Mutates the animal's genes with a chance of μ (constant) for each gene. The mutation step is drawn from a Gaussian with mean 0 and standard deviation 0.15.

The animal's genes can be read or written by using the properties `Animal.genes` or `Animal.gene_dict` (as list or dictionary, respectively).

2.4 *Populations*

Currently, all animals of a run are contained in one single Population, a class defined in `population.py`. It provides some wrappers for easy data access and operations on the whole population (i.e., all animals inside the population), and, most importantly, the two breeding controller methods `breed_constant()` and `breed_variable()`. These query the lifetime payoff of each animal, calculate the number of offspring for each parent, create the new animals with their parent's genes, call the `mutate()` method, and correct the population size by killing / cloning animals.

2.5 *Time step structure*

In each time step, the following processes take place:

1. The current state of the population is printed to the output files.
2. Environment states E and cue C are computed.
3. The population (i.e., each animal) reacts to the changed environment / migrates, if they are able to do so (gene $a > 0$, $m_a > 0$).
4. The current time is increased by 1.
5. Repeat steps 2–4 L times.

6. Let the population breed, and correct population size.
7. Let the newly born animals react to their environment once, regardless of their reversibility.

3 MODIFICATION OVERVIEW

An exhaustive list of all modifications made to the original Botero model in this code package:

- New genes m and m_a were introduced modeling (reversible) migration.
- Environments now have an offset parameter O, since the invariance to absolute values is destroyed as soon as multiple environments are introduces (one might e.g. want to model a *warmer* and a *colder* environment).
- A new parameter containing the *cost* of migration, k_m, was introduced. The default value is $k_m = 0.2 = 10k_d$ (k_d: cost of plasticity).
- To prevent the population from stacking all animals in one environment, the animals' lifetime payoff is decreased by a factor of $1 - N_{env}/N$ (N_{env}: number of animals in current environment, N: total number of animals in population). This essentially models some sort of rudimentary competition between the animals in the same environment.
- In this version, it is possible to limit certain genes to the range $[0, 1]$. This is to make sure that e.g. migration does not become so unlikely to be developed that it is never observed during a run with variable population size.
- In the original model, the insulation I of an animal is calculated as (analogous with primed quantities I_0', b'):

  $$I(C) = I_0 + b \cdot C$$

  Since we are now dealing with environments that have an offset O, it was necessary to limit the animals' ability to adapt equally easily to any absolute value of the environments. Thus, a scale function f was introduced, such that

  $$I(C) = f(I_0) + f(b) \cdot C$$

  Currently, the scale function implemented in `animal.pyx` is

  $$f(x) = \operatorname{sgn}(x) \frac{\log_{10}(|x| + 1)}{\log_{10}(3)}$$

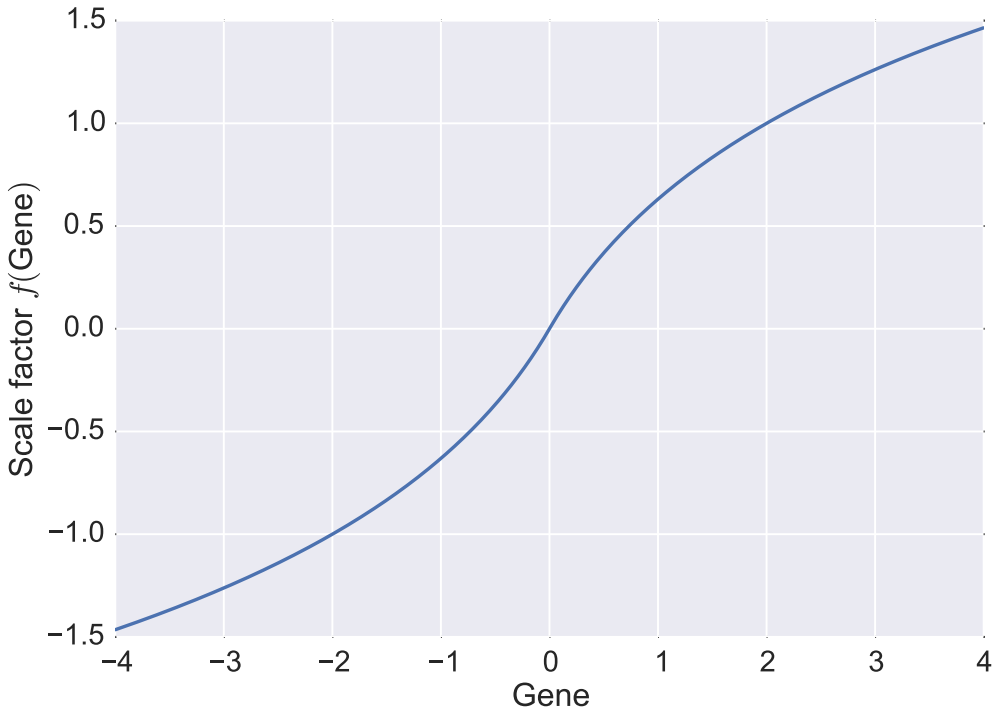


Figure 1: Scale function f

which runs linearly for small values of $|x|$, but then proceeds to grow logarithmically, making it increasingly harder to adapt for extreme environments, but not impossible (cf. Fig. 1).

4 PARAMETER OVERVIEW

Here, the parameters of the model as defined in the file `constants.py` are explained:

Parameter	Type	Meaning
population_size	int	Number of animals per population.
generations	int	Number of generations per run.
L	int	Life time of each animal in time steps.
kd	float	Constant cost of plasticity.
ka	float	Cost of each adaptation.
tau	float	Coefficient of lifetime payoff exponential.
q	float	Controls expected number of offspring in variable scenario.
mu	float	Mutation rate of the genes.
environments	[[5×float]]	Parameters of each environment in the order R, P, A, B, O.
environment_names	[string]	Displayed name of each environment.
km	float	Cost of migration.
limit	[string]	Names of genes that should be limited to $[0, 1]$.
populations	int	Number of identical populations per run.
plot_every	int	Detailed output is plotted every N generations (0 = never).
verbose	bool	Triggers verbose output to command line.

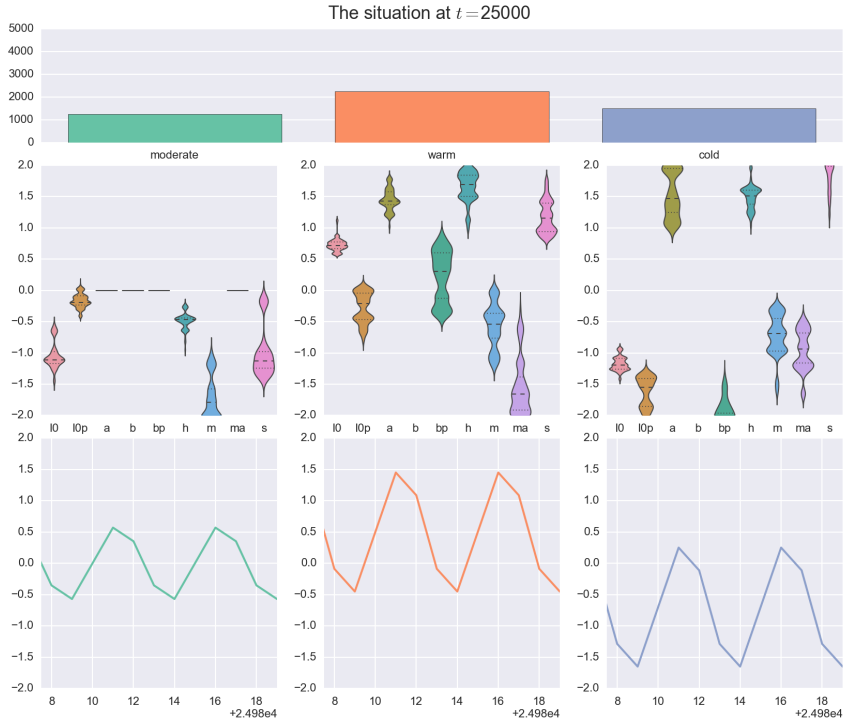
For more information, refer to the original paper, [1]. All parameter names are unchanged.

5 EXAMPLARY RESULTS

5.1 *Final states*

Some final states for the standard parameters, but with different values of R (the same for all environments):

$\mathbf{R = 1}$:

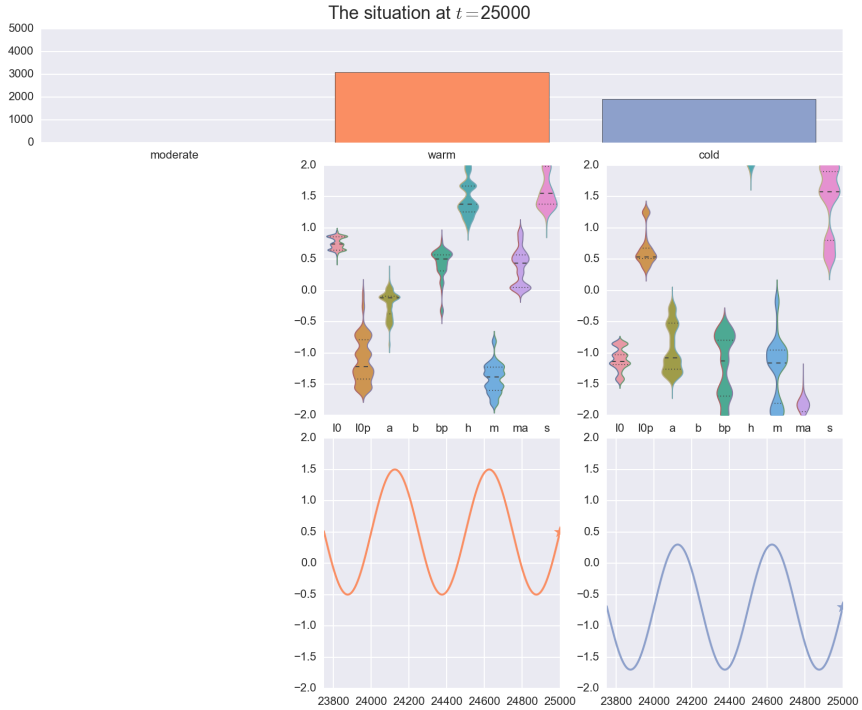


The animals in the moderate environment (which has low predictability) develop a conservative bet-hedging strategy, while the animals in the other, more extreme (but more predictable) environments develop reversible plasticity.

R = 10:

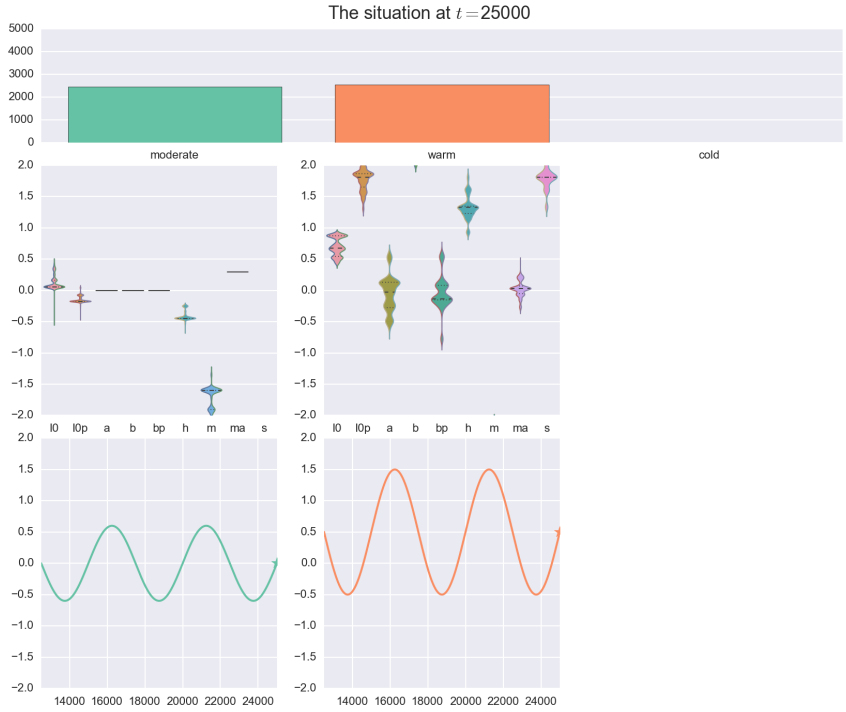
The moderate environment still shows conservative bet-hedging. The warm and cold environments show reversible plasticity of a lesser degree (smaller a).

R = 100:



The animals in the moderate environment die out, the other environments show irreversible plasticity. The animals in the warm environment are more successful, since it is more predictable.

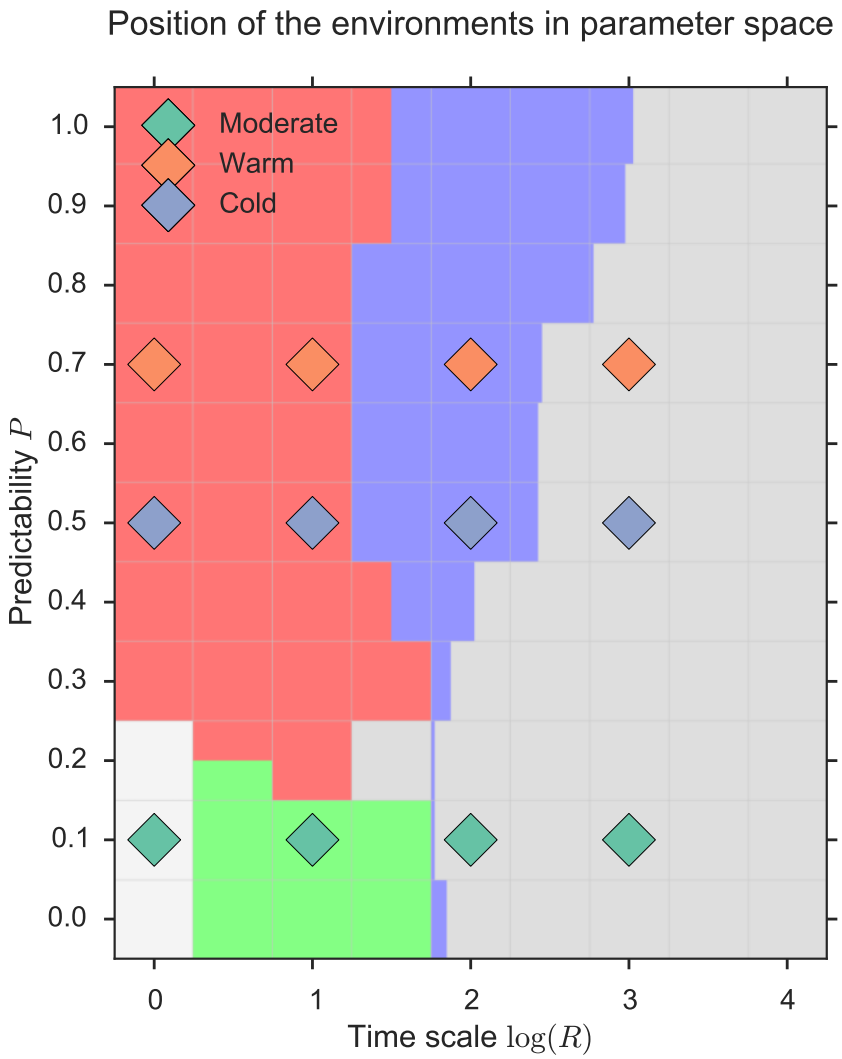
R = 1000:



The moderate environment shows adaptive tracking, while the animals in the warm environment develop a mixed strategy (adaptive tracking and irreversible plasticity). The animals in the cold environment die out. This is probably caused by the scale function f, which punishes adaptive tracking for extreme climates.

5.2 *Observed strategies*

The following figure shows the position of the standard environments in parameter space, and the strategies found in the final states during test runs for various combinations of (P, R):



Color-coding:

WHITE: conservative bet-hedging,

GREEN: diversifying bet-hedging,

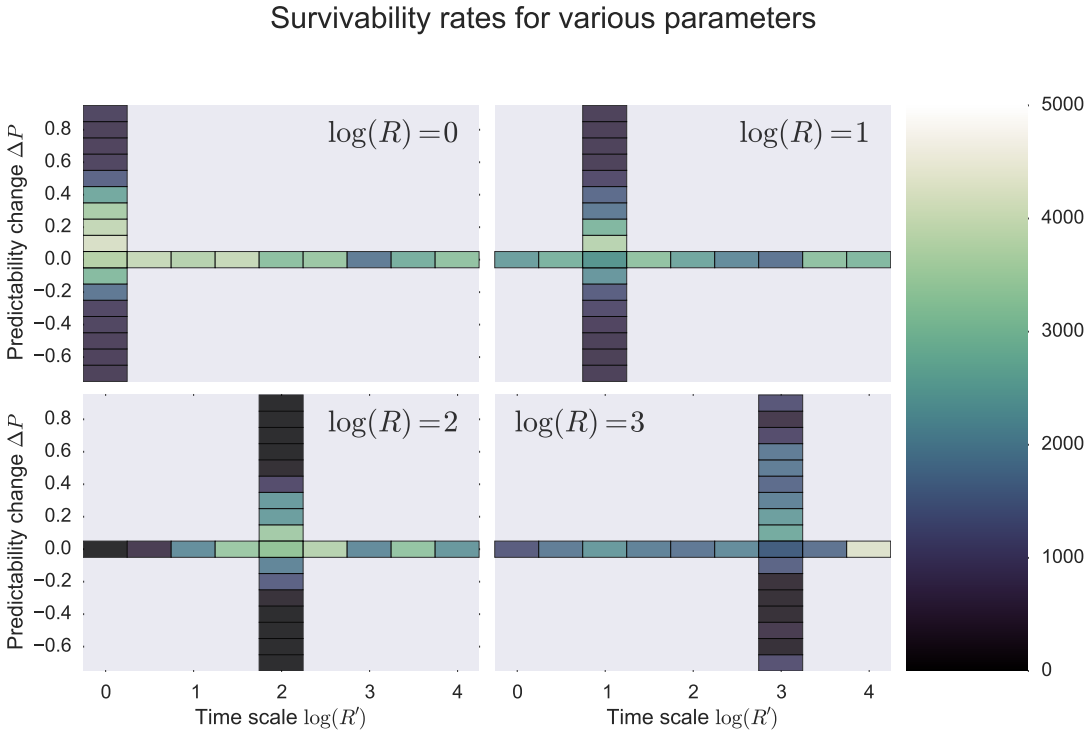
RED: reversible plasticity,

BLUE: irreversible plasticity,

GREY: adaptive tracking.

5.3 *Tipping points*

The following figure shows statistics on how many animals of a population (coming from environments with parameters $(P_i,\ R),\ i \in \{1,2,3\}$) survived when put into environments with parameters $(P_i + \Delta P,\ R')$ during a run with variable population size:



REFERENCES

[1] Carlos A. Botero, Franz J. Weissing, Jonathan Wright, and Dustin R. Rubenstein: Evolutionary tipping points in the capacity to adapt to environmental change. *PNAS*, 112 (1): 184-189, 2015.
http://www.pnas.org/content/112/1/184

LICENSE