Network evolution Documentation

Release 1.1

Paul François, Mathieu Hemery, Adrien Henry

Mar 29, 2018

Contents

CHAPTER 1

Install ϕ-evo

ϕ-evo relies on python>=3.5, pip, and c.

The software has been successfully tested on the three main operating systems(windows,mac OSX, and GNU-linux) but **we recommend using a GNU-linux distribution(ubuntu)** as it has been tested more thoroughly and more regularly on this platform.

1.1 install Anaconda

The _phievo package depends on python>=3.5. If python is not already installed on your computer, we recommend to install it by using the anaconda distribution.

Among other things, anaconda provides the standard package manager of python *pip*. Before anything, it is good to check that you are working with the most recent version of pip:

```
pip install --upgrade pip
```

Note: When multiple versions of python are installed on the same computer, you may need to specify the version of python or pip you are using: `python` (`pip`) for python2 and `python3` (`pip3`) for python3. Make sure that the `which pip` and `which python` return the right pip (and python) installation path. For simplicity we will use `pip` in the following instructions.

Note: If you install packages for all the users of your computer, you need to have admidistrator rights and use `sudo` before the pip command. It can happens that your global and your local pip are not the same. To make sure the administrator uses the right pip, run `sudo which pip`. The installation instructions assume you do not need to add `sudo` before `pip`.

1.2 install the package

With pip installed, the installation is straight forward, run:

```
pip install https://github.com/phievo/phievo/blob/master/dist/phievo-1.1.zip?raw=true
```

1.3 Install gcc on windows

Windows does not come with the `gcc` compiler installed but the free software foundation provides a minimal distribution of the gnu softwares for windows, it is called MinGW.

Once you have downloaded `mingw-get-setup.exe`, run it. A selection panel will open. We recommend you to install at least the two following packages(the others are not relevant for ϕ-evo): - mingw-developper-toolkit - mingw32-base

Choose the default directory.

After the installation is finished, update windows `PATH` so that it knows where to look for the `gcc` command. Open a the command prompt and run:

```
setx PATH "%path%;C:\MinGW\bin"
```

Note: gcc is distributed by other packages such as code blocks or visual basics. In such case, you do not need to install MinGw. Just upload you `PATH` so that windows knows where is the gcc compiler.

1.4 Install gcc on mac osx

OSX does not have the gcc compiler installed by default either. There are different ways to install it. The fastest is probably via homebrew:

```
brew install gcc
```

If `gcc` is not already installed on you system (via macports or Xcode), *homebrew*'s *gcc* should be automatically in the system's `PATH`.

1.5 Install pygraphviz

pygraphviz is not included in the default dependencies of *phievo* because it does not exist natively on windows and we wanted to publish a version that that runs on all the systems. *pygraphviz* is used only to display network layouts. If it is not installed, *phievo* will print a warning and use *networkx* spring layout instead.

On max OSX, you have to use homebrew to install graphvix first :

```
brew install graphviz pkg-config
pip install pygraphviz
```

On GNU/linux, installing the dependencies varies depanding on the distribution. We tested the following on debian and ubuntu

```
sudo apt-get install graphviz graphviz-dev pkg-config
sudo pip install pygraphviz
```

On other distributions, you want to find the equivalent of *graphviz*, *graphviz-dev*, and *pkg-config*.

We found that sometimes on ubuntu the C linking to the graphviz library does not work properly. The fix is to be more explicit on the linking for the pip command:

```
sudo pip install pygraphviz --install-option="--include-path=/usr/include/graphviz" --
↪install-option="--library-path=/usr/lib/graphviz/"
```

1.6 run_evolution.py script

An extra script (run_evolution.py) needs to be downloaded with the phievo package to start an evolution. It is stored in the root of the phievo repository.

You can either manually download it or open a python terminal and run

```
>>> import phievo
>>> phievo.download_tools()
```

The former utility also downloads a jupyter notebook that can be used to analyse the results of a simulation in current directory.

1.7 Analyse notebook

We provide a jupyter notebook at the root of the github repository to help with the analysis of the runs. If you wand to run it, you will need to install several extra python libraries, to help with this, they are writen in extra.txt.

```
pip install -r https://raw.githubusercontent.com/phievo/phievo/master/extra.txt
```

Similarly to the (run_evolution.py) script, Analyse Run.ipynb is downloaded when you call the `phievo.download_tools()` function.

The jupyter kernel is started with the following command

```
jupyter notebook
```

Usually it autmotically opens a new windows in your terminal in which you need to select `Analyse Run.ipynb`. If the windows does not open, it can be open manually by copy-pasting the url printed in your shell after you ran the command in a wer browser.

When using the plotly package, you may find that the plots do dot display well in the notebook (white square), the solution to this problem is to increase the io rate allocated to the notebook by using the `NotebookApp.iopub_data_rate_limit` option when starting jupyter:

```
jupyter notebook --NotebookApp.iopub_data_rate_limit=10000000000
```

1.8 Test your installation

To test that everything works properly, we recommend that you run an example simulation. Several examples of simulations are stored in the github repository Examples directory. You can download all the simulations by cloning the repository with git:

```
git clone https://github.com/phievo/phievo.git
```

This will also download phievo's code.

To download a single example there is a built-in tool that can be run in a python shell:

```
>>> import phievo
# Downloads run_evolution.py and Analyse Run.ipynb in  the current directory
>>> phievo.download_tools()
# Downloads an example project directory
>>> phievo.download_example("adaptation")
```

The function `download_example` allows to download one of the following examples:

- adaptation
- somite
- hox
- hox_pareto
- lac_operon
- immune
- seed_adaptation
- seed_adaptation_pruning
- seed_somite
- seed_somite_pruning
- seed_lacOperon
- seed_lacOperon_pruning
- seed_hox_pareto_light

The examples starting with "seed_" keyword also contain the results of the simulations. The results can directly be visualized in the Analyse notebook.

After downloading an example project directory and the *run_evolution.py* script you are all set to start an evolution.

```
|-- run_evolution.py
|-- Analyse Run.ipynb
`-- example_adaptation/
    |-- initialization.py
    |-- fitness.c
    |-- init_history.py
    `-- input.c
```

To launch the evolution, simply run

```
python run_evolution.py -m example_adaptation
```

Note: You can add the -c option (`./run_evolution.py -cm example_adaptation`) to delete a Seed that was created by a former run and prevents a new run to start. Be careful, a deleted seed cannot be recovered.

If everything works correctly you should see the evolution starting. When an evolution is running it displays regularly updates of its current state in the terminal and a `STOP.txt` file is created at the root of the project. The purpose of the STOP file is to have a quick method to check on the current state of a run when it is launched as a background task. When the *STOP* file is deleted, the run stops.

1.9 Create a new project

To start a new project, the best is to use an existing example as a template and to modify the relevant parameters.

Similarly to the `Analyse notebook`, we also propose the Project Creator.ipynb notebook to help with the creation of a new project.

```
jupyter notebook Project\ Creator.ipynb
```

CHAPTER 2

Presentation

This section presents the basics element to understand the structure of the algorithm and the role of the various python modules.

2.1 An algorithm overview

There is three main blocs in the algorithm that correspond to the three libraries present at the root of the project.

- *Networks*: gathers all the elements to represent, modify and simulate the evolving biological networks that are the indidividual level of our population.
- *Population_types*: implements the so-called genetic algorithm and its several variants through a *Population* class and its subclasses.
- *AnalysisTools*: gathers the tools used after the simulation to analyse, represent and study the results.

2.2 Network components

Network (and its sub-class *Mutable_Network*) represents the individual level of our evolutionary algorithm. Apart of the methods used to implement the different operations, the main attribute is *graph*, a networkx.MultiDiGraph object that stores the biochemical network as a bipartite graph of *Species* (and *TModule*) on one side and *Interaction* on the other. The organisation of the graph thus relies on the networkx package.

The subclass called *MutableNetwork* handles the mutations in the *Network*

The *deriv2* module is responsible for reading a *Network*'s interactions and to generate a C file that will integrates the differential equations simulating the species and then computes the fitness of the network that will be used at the genetic algorithm level.

2.2.1 Species

Species is one of the two major components of a network. A species is a protein that can have different types (Degradable, Phosphorylable, etc.). Most of the time, those species will be added automatically by the algorithm when handling the various interactions. For example when two species are chosen to be part of a new protein-protein interaction, a new species will be added to simulate the complex thus formed.

However, to manually build the initial network, you may want to add species with some fixed properties. For this, you need to build a list of lists containing the type name as a first element and the different parameters (if any) must complete the list in a pre-defined order (`l_types` in the example below). For instance a degradable species comes with its degradation rate. Note that adding a single Species is actually quite rare as they often came as a whole gene with a CorePromoter and a TModule (see the TModule picture below).

```
l_types = [["Degradation",0.5],["Complexable"],["Output",0]]
mySpecies = my_Network.new_Species(l_types)
mySpecies = my_Network.new_gene(rate, delay, l_types, basal_rate)
```

But see the `initiation.py` file of an example to a complete construction of a Network object.

2.2.2 Interaction

The Interactions, as suggested by its name, accounts for how species and TModules interact. Examples of interactions are protein-protein interactions, transcription factor regulations, etc. See the sections below for various type of preimplemented interactions.

Note also that it is often necessary to implement new interactions tailored for a specific task. (See `Examples/immune/` for an example of such new interactions.)

2.2.3 TModule

A TModule is the last type of network component. It corresponds to the transcription part of a gene and is connected to the species it controls via a CorePromoter interaction. In the ϕ-evo's framework, a gene is thus represented by three components: *TModule*, *CorePromoter*, and *Species*.

Uphill, the transcription factor species that regulate a gene are connected to the Tmodule through TFHill interactions.

2.3 Population & Evolution

The evolution algorithm mimics Darwinian selection by simulating a population where the individuals are in competition to pass their genome to the next generation.

It first generates an initial population the size of which is defined by the user by cloning an initial Network and from then follow cycles of mutation, fitness computation and selection. Each cycle thus defines our time step of evolution and will subsequently be called a generation.

2.3.1 Elite strategy

By default we choose to use the *elite strategy* because of its robustness and its cheap computationnal cost. Thus, during the selection step, the worst part of the population is deleted, while the fittest half of the individuals are directly passed to the next generation. Then, each of them is copied and this copy is mutated.

Note that this scheme automatically keeps the population size constant. Moreover, it relies only on the rank of the individuals in the population and not on the quantitative fitness. This makes it very robust to the possible difficulties and failures of the fitness implementation.

2.3.2 Pareto evolution

In the case where the fitness is composed of multiple components, it is not obvious how to balance the different modules in the global fitness. It may be interesting to have a multiple objective optimization where all the components of the fitness have the same importance; only changes improving one component without decreasing the others are considered as an improvement.

For a fitness splited in N components: $F = \{f_1, f_2, ..., f_N\}$. We say that individual i dominates (strictly) j if and only if the fitnesses F^i and F^j are such that:

$$\forall k \quad f_k^i \geq f_k^j, \quad (\exists k \quad f_k^i > f_k^j)$$

Clearly multiple objective optimisation does not result in one best network in the end but to a population of highest rank networks called the Pareto front. More information can be found on Wikipedia.

From a practical standpoint, the algorithm works similarly to the genetic algorithm with a modified selection process. As in the genetic algorithm, half of the population is passed to the next generation and duplicated. Because the only classification criterion is the network's rank, the cutoff may occur in the middle of a set of equivalent network since they have the same rank. In such a case the algorithm selects randomly the networks with the cutoff rank to complete the set of individuals passed to the next generation.

2.3.3 Results

During the evolution, the results are stored in separate folder for each seed soberly called _Seed*/_, this folder contains three main type of elements:

- *log_?* — are brute copy of the files used as input for this seed (the correspondance should be obvious).
- *Bests_?.net* — is a pickle of the *Network* object with the best fitness at the corresponding generation, this allows you to trace back the evolution of the individuals in the population
- *data.?* — contains various data about the seed (mean fitness, times, etc.)
- *Restart_file.?* — this shelve object contain a copy of the whole population in case you want to restart the evolution after the termination of the first run of the program.

2.4 Modelization & Integration

To simulate the dynamics of a species the program first needs to explore the nodes and the interactions that are connected to it. Then it builds the equations that govern the dynamic of its concentration. These equations are then written as C code and integrated.

The following sections presents the predefined networks interactions and there corresponding ordinary differential equations.

2.4.1 TModule and gene production

There exists two types of TF actions: activition and inhibition. Both types are modelled using Hill functions but there their effects is included differently to the global regulation. Only the maximum of all the activations is accounted for

whereas the inhibitions are multiplicative. In some extend activation and repression work respectively as OR and NOR logical operations.

Next the CorePromoter interaction adds a delay τ_P to accounts for the protein synthesis time. Practically, the algorithm considers the state of the system at time $t - \tau_P$ to estimate the production of P at time t.

The following configuration

leads to the following equation:

$$\frac{dS}{dt} = \left(\max \left\{ PR_S \times \max \left\{ \frac{A_1^{n_{A1}}}{A_1^{n_{A1}} + h_{A1}^{n_{A1}}}, \frac{A_2^{n_{A2}}}{A_2^{n_{A2}} + h_{A2}^{n_{A2}}}, \dots \right\}, B_S \right\} \times \frac{h_{R1}^{n_{R1}}}{R_1^{n_{R1}} + h_{R1}^{n_{R1}}} \times \dots \right)_{(t-d_S)}$$

—

In the above equation, the h and n parameters correspond respectively to the Hill constant and coefficient. The PR is the production rate of the protein in optimal conditions and B is the basal rate(in case no activator is present). The overall production is modulated by the repression.

2.4.2 Degradation

Every protein P labelled as *degradable* is degraded over time with a rate δ_P. This

$$\frac{dP}{dt} = -\delta_P P$$

2.4.3 Phosphorylation

The phosphorylation is the addition of a phosphate group to a Species by a kinase. It creates a new phophorylated species. The dynamics of this mechanism is controlled by a hill function that accounts for the use of the kinase by all the different species. In the case of of kinase that catalyses the phosphorilation of two species S_1 and S_2.

$$\frac{dS_1}{dt} = -\frac{dS_1^*}{dt} = k_p^1 \frac{K \left(\frac{S_1}{h_1}\right)^{n_1}}{1 + \left(\frac{S_1}{h_1}\right)^{n_1} + \left(\frac{S_2}{h_2}\right)^{n_2}} - k_d^1 S_1^*$$

$$\frac{dS_2}{dt} = -\frac{dS_2^*}{dt} = k_p^2 \frac{K \left(\frac{S_2}{h_2}\right)^{n_2}}{1 + \left(\frac{S_1}{h_1}\right)^{n_1} + \left(\frac{S_2}{h_2}\right)^{n_2}} - k_d^2 S_2^*$$

Note that by default, there is no mechanism implemented for active dephosphorylation so that they hapen with constant rates k_d^1 and k_d^2.

2.4.4 Protein-Protein-Interaction (PPI)

The PPI interaction accounts for the complexation of two single proteins into one complex.

The rate is obtained from a mass-action dynamics:

$$\frac{dP_1}{dt} = \frac{dP_2}{dt} = -\frac{dC}{dt} = -\text{rate} = -k^+ P_1 P_2 + k^- C$$

with k^+ and k^- being respectively the forward and backward rate constants

CHAPTER 3

Create a new project

This tutorial lists a series examples on how to perform common tasks with ϕ-evo.

3.1 Build a network manually

Before even starting a simulation, let us build a network manually in order to get familiar with the way they are encoded in the program. Most of the code is written in python[1], let us call our first file **HowTo_manualNetwork.py** is provided in the example directory.

```
# Import libraries
from phievo.Networks import mutation
import random

# Create a random generator and a network
seed = 20160225
g = random.Random(seed) # This define a new random number generator
L = mutation.Mutable_Network(g) # Create an empty network
```

We have created a first network, **L**, that can be used as a container for the species and insteractions. For now **L** is still empty, we can add a new species as follows

```
parameters=[['Degradable',0.5]] ## The species is degradable with a rate 0.5
parameters.append(['Input',0]) ## The species serves as an input referenced by the␣
↪index 0 in the evolution algorithm.
parameters.append(['Complexable']) ## The species can be involved in a complex
parameters.append(['Kinase']) ## The specise can phosphorylate another species.
parameters.append(['TF',1]) ## 1 for activator 0 for repressor

## Create a species and add it to the network
## S1 is a reference to access quickly to the newly created species latter in the code
S1 = L.new_Species(parameters)
```

[1] The front interface is coded in **python** (version >3.5). But for efficiency reason, the core integration is coded in **C**.

All the characteristics we want to associate with the species are listed followed by their parameters. The list is then sent to the *new_Species* function to create the species. This methods is used when adding a external species (such as an input) that is not produced by the network itself.

In most cases a species comes with its transciptional machinery (*Species* + *CorePromoter* + *TModule*). The species and its related componant are added via the *new_gene* function.

Similarly a *PPI*(protein-protein interaction) is added with the complexation reaction and a phosphorylated species is added with the phosphorylation interaction.

Adding these functions to a code would look like this

```
parameters=[['Degradable',0.5]]
parameters.append(['TF',1])
parameters.append(['Complexable'])
TM0,prom0,S0 = L.new_gene(0.5,5,parameters)

parameters=[['Degradable',0.5]]
parameters.append(['TF',0])
parameters.append(['Complexable'])
TM1,prom1,S1 = L.new_gene(0.5,5,parameters)

parameters=[['Degradable',0.5]]
parameters.append(['TF',1])
parameters.append(['Phosphorylable'])
TM2,prom2,S2 = L.new_gene(0.5,5,parameters)


parameters=[['Degradable',0.5]]
parameters.append(['TF',0])
TM3,prom3,S3 = L.new_gene(0.5,5,parameters)

## Add complexation between S0 and S1.
parameters.append(['Kinase'])
ppi,S4 = L.new_PPI(S0 , S1 , 2.0 , 1.0 , parameters)

## Add a phosphorylation of S2 by S4
S5,phospho = L.new_Phosphorylation(S4,S2,2.0,0.5,1.0,3)
S5.change_type("TF",[1]) # Note this is already the default value for a␣
↪phosphorilated species

## Regulate the production of S1 by S3 and S5
tfhill1 = L.new_TFHill( S3, 1, 0.5, TM1,activity=1)
tfhill2 = L.new_TFHill( S5, 1, 0.5, TM1,activity=1)
```

To display the layout of the former network, the program provides draw function :

```
L.draw()
```

3.2 Run a simulation

A ϕ-evo project is stored in a directory named as the project.

```
mkdir lac_operon
```

It contains all the configuration files of the project

- initialization.py (name must start with "init"): Contains the initialization parameters, the path to the C files and optionally an inial network. If the former is not described in the initialyzation file, it will be generated randomly.
- a fitness **C** file code used to compute the fitness. After an integration, the dynamics is stored in an array `history[SPECIES][TIME][CELL]`. You need to create a custom set function that analyse this array. In the end, the function *treatment_fitness* should print the fitness of the network.
- An init history file that contains the code that sets `history[SPECIES][t=0][CELL]` wrapped in a function called *init_history*.
- An init input file creates an *input* function. The input function is called at every time step to modify the `history` if necessary.

3.2.1 initialization.py

This file stores the informations about the evolution such as the ranges of variation for the parameters, the mutation rates, the paths to the C files, or the algorithm parameters.

The dictionary *dictionary_ranges* sets the range of values a parameter can take. If only one value Max is given, then the the range is [0,Max]. To specify the the minimal value for a parameter, you have to provide an array [Min,Max]

```
## The hill coefficient of a TFhill can varry between 1 and 5.
dictionary_ranges['TFHill.hill']= [1., 5.0]
## The rate of a TModule can varry between 0 and 2.
dictionary_ranges['TModule.rate']= 2
```

The dictionary *cfile* contains the path of the C files

```
cfile['fitness'] = fitness.c
cfile['init_history'] = init_history.c
cfile["inputc] = input.c
```

The dictionary *dictionary_mutation* contains the rates at which a mutation in the network appears. Note that the alorithm gathers the rates provided and normalizes them in order to have an average of one mutation per new generation during the evolution.

```
## Rate of appearance of the new transcription factor
dictionary_mutation['random_gene(\'TF\')']=0.02
```

The *prmt* dictionary contains the parameters related to the functioning of the program and the algorithm.

```
## Number of integration step in the Euler integrator
prmt['nstep'] =3000
## time step during the integration
prmt['dt'] = 0.05
## Setting prmt['restart']['activated'] to False allows to start a fresh simulation
prmt['restart'] = {
  "activated": False,
  "freq": 50 # Generation frequency for saving the complete population
}
## Define the compiler (gcc by default)
prmt["compiler"] = "g++"

prmt['langevin_noise'] = 0 # Intensity of the langevin noise for stochastic simulation
prmt['multipro_level'] = 1 # Use multiprocess if one 1. If 0, singlethread.
##
```

You may also specify the type of output you want and to prevent deleting species with a specific tag:

```
list_unremovable=['Input','Output']
list_types_output=['TF']
```

We can choose an intial network to start the simulation with. This is done through the *init_network* function. The construction of the initial network follows the steps presented in *Build a network manually*.

3.2.2 fitness.c

This file contains two required C functions *fitness* and *treatment_fitness*. The first function function computes the fitness each individual trials. Once all the trials have been analysed by *fitness*, the *treatment_fitness* function combines the different fitnesses (ex: taking an average, sum, etc.) and prints the summary fitness to the shell. The former fitness is read by the python algorithm and used to classify the networks among the other networks of the population.

You may add more analysis functions and to redefine *fitness* and *treatment_fitness* as long as it prints the network's fitness and has the following prototype:

```
static double result[NTRIES];

void fitness( double history[][NSTEP][NCELLTOT], int trackout[],int trial)
    {
        result[trial] = 0;
    }

void treatment_fitness(double history[NGENE][NSTEP][NCELLTOT], int trackout[])
    {
        for(trial=0;trial<NTRIES;trial++)
            total_fitness += result[trial];
        printf("%f",total_fitness)
    }
```

The `trackout` lists the indexes of the outputs in the networks. You can also decide to use the global list `trackin` which contains the indexes of the ouputs.

3.2.3 init_history.c

Before every integration, the algorithm reads the array `history[NGENE][0][NCELLTOT]` to set the initial conditions of the run. You can use the *init_history.c* file to edit the first time step, this way it will be used as a initial condition.

Note that you can be more specific by using the two lists `trackin` and `trackout` that contain the indexes for the inputs and outputs respectively.

```
void init_history()  {
 int ncell,n_gene;
   for (ncell=0;ncell<NCELLTOT;ncell++){
     for (n_gene=0;n_gene<SIZE;n_gene++){
       history[n_gene][0][ncell]=0;
     }
   }
 }
```

3.2.4 input.c

Sometime it is necessary to add artificial inputs during an integration. This is done via the *input* function. The *input* function is called at every time step and for every cell before computing the species derivatives. Since the derivatives for the species at time *t* are computed based on the values `history[NGENE][t][NCELLTOT]`, you can use *input* to modify the `history` array.

```
void inputs(int time,int cell,int trial){
    ...
}
```

To get more precise informations, we recommand you to have to look at how *Examples/lac_operon/* project is built.

3.2.5 Launching a run

The program is launched with the *run_evolution.py* script:

```
python run_evolution.py -m lac_operon/
```

The script loads the parameters and launches the run.

run_evolution.py should be placed in the same project directory as the project directory:

```
|
 --- run_evolution.py
 --- (Analyse Run.ipynb)
 --- example_project/
              |
               --- initialization.py
               --- fitness.c
               --- init_history.c
               --- input.c
```

Note: *run_evolution.py* is not installed with phievo and must be downloaded manually from here or by running the command `phievo.download_tools()` in a python shell.

To restart a new run, one must provide the # of the run (or seed index). By default, the run number is 0. To prevent errasing a run by mistake, the code will not start if you do not provide a new run number in the initialization file. You can also tell the program explicitly to clear the Seeds with the "-c" or "–clear" option.

```
python run_evolution.py -cm lac_operon/
```

3.3 Restart an evolution

Every *k* generations, the algorithm saves a complete generation in a file called *Restart_file* in the Seed's directory. If interrupted, you can use this *Restart_file* to restart from a backup generation. You can set the restart generation in the initialization file:

```
prmt['restart'] = {
  "activated": True, ## Activate restart
  "seed": 0, ## Index of the restart seed
  "kgeneration": 50, # Generation where to restart the algorithm
  "same_seed": True,
  "freq": 50 # Keep the same saving frequency
}
```

When the seed and the generation is not set or `None`, ϕ-evo will uses the last backup-ed generation in the seed with highest index.

3.4 Pareto evolution

To start a pareto(multi-objectives) optimization with ϕ-evo, extra paremeters need to be defined in the initialization file:

```
prmt['pareto']=True ## Activates pareto evolution
prmt['npareto_functions']=2 ## Number of fitness components
prmt['rshare']=0 ## Radius under which networks are penalysed for being too
                 ## close on the pareto front
```

CHAPTER 4

Simulation parameters

This sections presents the different parameters that can be set in the initialization file.

4.1 Kinetic parameters (`dictionary_ranges`)

The kinetic parameters are specific to a type of interaction or a type of species. They are stored in the `dictionary_ranges` dictionary. One can define the range over which they can vary by setting the its range with a size 2 list (if the minimum is 0, the range can be set with a float corresponding to the maximum).

In the example of a `Degradation` interaction, the range of variation of the `rate` of degration is set with

```
dictionary_ranges["Degradation.rate"] = 0.0
```

4.2 Mutation parameters (`dictionary_mutation`)

The mutation parameters define the rate a which a given mutation occurs. Note that the evolution rescale the generation time so that a network undergoes an average of one mutation per generation. The mutation parameters are defined in the `dictionary_mutation` dictionary.

A new mutation function is defined when creating a new interaction. Each new mutation can have its own rate defined.

Examples:

- `dictionary_mutation["random_gene()"]`: Rate at which `random_gene()` mutation is executed (with default settings).
- `dictionary_mutation["random_gene(type = 'TF')"]`: Rate at which `random_gene()` mutation is executed with the parameter *type* equal to *"TF"* (it creates a species with a tag "TF" corresponding to a transcription factor).

4.3 General simulation parameters (`prmt`)

The general simulation parameters are stored in a dictionary called `prmt`:

- Number of seeds (`nseed`): Number of independent evolution to simulate.
- First seed (`firstseed`): Index of the first seed. This index is also used to seed the random number generator.
- Number of generations (`ngeneration`): Number of generation to simulate in each independent evolution.
- Number of cells (`ncelltot`): Number of cells in the organism.
- Population size (`npopulation`): Number of network in the population.
- Number of neighbors (`nneighbor`): Number of neighbors cell has.
- Fraction mutated per gen (`frac_mutate`): Fraction of networks in the population to mutate at every generation.
- Number of Inputs (`ninput`): Number of species with an inputs (with an input tag) a network should have.
- Number of Outputs (`noutput`): Number of species with an outputs (with an output tag) a network should have.
- Number of trials (`ntries`): When a fitness depends of a network's initial conditions or in the presence of Langevin's noise, it is useful to run several independent kinetic integrations. `ntries` determines the number of integrations to run. Note that in the case the initialization of each trial should be done with the init_history function and the agregation(e.g. averaging) of the fitnesses coresponding to each integration is done with the treatment_fitness function.
- Time step dt (`dt`): Size of an integration time step in the Euler algorithm.
- Number of time steps (`nstep`): Number of integration time step in the Euler algorithm.
- Langevin noise value (`langevin_noise`): Level of the langevin noise in a stochastic simulation. When 0, the integrations are deterministic.
- Gillespie generation time (`tgeneration`): The computation of the next mutation follows a Gillespie algorithm. `tgeneration` defines the initial time, then the time `tgeneration` is updated to have roughly one mutation in `frac_mutate` of the networks.
- Recompute networks (`redo`): Should the networks that do not change in from a generation to the other be re-integrated ti compute the fitness?
- Pareto simulation (`pareto`): Should we run a Pareto integration?
- Number of pareto functions (`npareto_functions`): Number of pareto functions defined?
- Pareto penalty radius (`rshare`): This parameter prevents a network from being dominated by a networks with fitnesses that fall too close to it current position in the fitness space. Increasing `rshare` helps to explore a larger portion of the fitness space. Warmflash et al 2012.
- Multiple threads (`multipro_level`): Should the algorithm run in parallel?
- Generation printing frequency (`freq_stat`): During a simulation the algorithm regularly prints informations about its current state. `freq_stat` defines the number of generations between two prints.

4.4 Restart parameters (`prmt["restart"]`)

To restart a simulation either after it has been stopped or from a specific seed and generation one can configure the *restart* parameters. The parameters are hosted in a sub-dictionary or `prmt`, `prmt["restart"]`:

- `prmt["restart"]["activated"]`: Activate restart

- `prmt["restart"]["freq"]`: Frequency at which a complete generation is saved.
- `prmt["restart"]["kgeneration"]`: Generation at which to restart the algorithm
- `prmt["restart"]["seed"]`: Seed at which to restart the algorithm
- `prmt["restart"]["same_seed"]`: Restart with the same seed

More information is available on the (restart an evolution section)[create_new_project.html#restart-an-evolution].

CHAPTER 5

Results and Analysis Tools

ϕ-evo has a module dedicated to the analysis of the results. The results are stored in a *Simulation* object that contains a set of method that give a quick access to the most relevant observables of a run. To start analyzing the `evo_dir` project, you need to create a *Simulation* object associated to it.

```
from phievo.AnalysisTools import Simulation

sim = Simulation("evo_dir")
```

From there it is pretty straight forward to explore the architecture of the results. A simulation contains Seeds which themselves contain Networks. In order not to overload the memory, the Seeds only store a link to the networks. As an example, here is how you would load the best network for generation 350 in the seed number 2:

```
sim = Simulation("evo_dir")
best_net_2_350 = sim.Seeds[2].get_best_net(350)
# Equivalent to
sim = Simulation("evo_dir")
best_net_2_350 = sim.get_best_net(2,350)
```

5.1 Organization of the results

If you want to understand why the *Simulation* object is organized the way it is and how to go beyond its possibilities, you need to have an idea of how ϕ-evo stores the results of a simulation.

By default, for every generation g only one *Network* is stored using pickle in a file labelled `Bests_g.net`. When the simulation has only one fitness objective, this network is the one with the best fitness in the population. However when the evolution is run using a multiobjective criterium (like pareto optimisation), the best net is chosen randomly among the network of rank 1.

The former storing method limits the disk space usage. However you might want to store the whole population either for restarting the algorithm from a given generation or to analyze every member of the generation. To add this feature, you can specify a storing period by setting the `prmt['restart']['freq']` parameter in the initialization file

before launching the simulation. For example, if you set it to 50, the complete population will be stored every 50 generations in a python *shelve* named `restart_file`.

Other files created:

- `data` is a quick access shelve file to certain informations stored as lists at the following keys:
  - *generation*: index of the generation
  - *fitness*: fitness of the best network
  - *n_species*: number of species in the best network
  - *n_interactions*: number of interaction in the best network
- `parameters` is a copy of the parameter dictionnaries (defined for the non default in the initialization file) that were used during the simulation.
- `log_#.c` Copy of the input, fitness, history, etc. **c** files used for the simulation.
- `log_init_file.py` Copy of the init file used for the simulation

5.2 Analysis Tools

In this section we will explore the built-in functions that are bound to a *Simulation* object.

5.2.1 custom_plot

Plots two observables one against each other for a given seed. The available observables are the ones contained in the `data` file ("generation", "fitness", "n_species", "n_interactions").

```
sim.seeds[1].custom_plot("generation","fitness")
# Similarly you can use the shortcut
sim.custom_plot(1,"generation","fitness")
```

5.2.2 plot_fitness

There also exists a method to plot the fitness directly:

```
sim.seeds[1].show_fitness()
# or
sim.show_fitness(1)
```

5.2.3 get_best_net

Get the best net found in a given generation (the function reads the `Bests_g.net` file and return the Network object)

```
bestnet_g5_seed3 = sim.seeds[3].get_best_net(5)
# or
bestnet_g5_seed3 = sim.get_best_net(3,5)
```

5.2.4 get_backup_net

If you want to extract a network from a entirely stored generation, you can use *get_backup_net*. Be careful though, not every population is stored in the `restart_file`. You can use the `stored_generation_indexes` to check which generation has been stored.

```
net8_g50_seed3 = sim.seeds[3].get_backup_net(50,8)
# Or
net8_g50_seed3 = sim.get_backup_net(3,50,8)
```

5.2.5 stored_generation_indexes

The *stored_generation_indexes* is method that returns the list of stored generations.

```
list_stored = sim.seeds[1].stored_generation_indexes()
# Or
list_stored = sim.stored_generation_indexes(1)
```

5.2.6 Read a network from the pickle file

The simulation stores the best networks of every generation in the name `Bests_#.net`. This is only a pickle file and can be read manually using the pickle library:

```
import pickle

with open("Bests_#.net","rb") as net_file:
    net = pickle.load(net_file)
```

Or using the ϕ-evo function:

```
import phievo

phievo.read_network("Bests_#.net")
```

5.2.7 Running a network's dynamics

By construction ϕ-evo does not allow to quickly run the dynamics of a network. Because the dynamics is computed in C (for performance reason), a python Network object does not have a method that directly returns the derivative at a given state of gene quantities. However ϕ-evo has the method `run_dynamics` to symplify the run of a dynmics for a given network based on the history and inputs defined in *init_history.c* and *input.c* respectively.

```
net = sim.get_best_net(3,5)
dyn_buffer = sim.run_dynamics(net=net,trial=1)
```

You can specify the number of trial you want to run (if the dynamics is stochastic for example). The buffer returned by the function is a dictionary where the "time" and "net" keys give you access to the time vector and the network used for the run respectively. The other keys are the index of the trial for which you want to access the data. Note that the buffer is also stored in the *Simulation.buffer_data*, the latter is erased every time you run a new set of dynamics for *Simulations*.

5.2.8 Plotting the results of a dynamics

The simulation object allows you to plot the two results you would like to see after running a dynamics:

1. The time course of the genes in a given cell with *Plot_TimeCourse*
2. The evolution of the genes along the system at a given time point with *Plot_Profile*

```
sim.Plot_TimeCourse(trial_index=1,cell=1)
sim.Plot_Profile(trial_index=1,time=1)
```

5.2.9 Draw a network's layout

The network object contains a function to draw the layout of its gene interactions:

```
net = sim.get_best_net(3,5)
net.draw()
```

the option *edgeLegend* makes appear all the ids of the different species and interactions:

```
net.draw(edgeLegend=True)
```

5.2.10 Modifying an existing network

You can easily delete an interaction or a species from an existing network once you know its id through calling *delete_clean* and specifying the id and the type of the node to remove:

```
net.delete_clean(id=2,target='interaction')
net.delete_clean(id=5,target='species')
```

delete the interaction 2 and species 5 respectively.

To modify a precise node, you can access it with the function *get_node* and then modify it

```
my_species = net.get_node(id=2,target='species')
my_species.degradation = 1.0
```

will set to 1 the degradation rate of species 2.

5.2.11 Storing and retrieving network

Once modified, you can store the resulting network in a pickle with:

```
net.store_to_pickle('my_file.net')
```

and read it later with:

```
net = phievo.read_network('my_file.net')
```

Note that the net extension is present only for readibility.

5.3 Notebook

To facilitate the use of the former functions, ϕ-evo as a class *Notebook* that is used to run them in a jupyter notebook.

All the functions described previously can be used directly in a jupyter notebook but the *Notebook* class improves the usability by handling the dependencies between widgets. For instance you want the module in charge of plotting a network's layout to be disabled as long as a Seed and a Network have not been selected.

A Notebook object serves as a container for all the available modules you can use in the jupyter notebook. A module contains the material to handle a cell: its widgets, some update functions and a display function that displays the widgets in the jupyter notebook. In the end, the user only needs to run `myNotebook.myModule.display()` to create a jupyter elementary app in a cell. Then the module should be able to handle the expected inputs from the user.

5.3.1 Creating a custom module

Every module of contained in the *Notebook* inherits from the *CellModule* class. The latter is a minimal template used to constrain the requirements a module must have:

- `__init__(self,Notebook)` : The init function takes the *Notebook* it is contained in as an argument.
- `update(self)` : If the module has dependencies, this function must be defined. When dependency is updated, this function is called.
- `display(self)`: The function must be redefined to display the widgets and to handle the relation between them.

__init__

This is the function where you define the different widgets for the module. It is also here that you define the dependencies of the module or create a new ones. The dependencies system allows communication between different *CellModules*.

```
## Inform the notebook that MyModule depends on the Seed
self.notebook.dependencies_dict["seed"].append(self)
## Creates a dependencies
self.notebook.dependencies_dict["dep_name"] = []
```

Note that if you create a new dependency, you should make sure that you also handle the updates when the dependency changes:

```
for cell in self.notebook.dependencies_dict["dep_name"]:
    cell.update()
```

update

Every module, particularly those with dependencies, should have an update function. This is the function to call when the dependency is changed. The update function can do whatever you want but mostly its purpose is to unable/disabled the widgets when a dependency is changed or to reset their options.

In Addition to the *self.notebook.dependencies_dict*, a module can access the dictionnary *self.notebook.extra_variables* to pass values between *CellWidgets*.

display

The display function is here to contain the interaction and display code you would normally put in a jupyter notebook to handle the communication of the widgets with the functions.

The philosophy of the *CellModule* is to create an elementary app in charge of one action (plotting a curve, setting the seed, etc.). Using a module's display method in a cell gives access to the app at this location.

Other functions

The *update* and *dispay* functions are usually not enough to run the *CellModule*. You will need to define custom methods for your module to handle the widget interactions(for instance, what happens when a widget is clicked?).

Example: DisplayFitness

Here is a little practical example on how to include a custom *CellModule* that displays the best fitness of the selected generation when the button is clicked.

Create a module file *NB_Module.py* and import the Notebook module and some widget libraries:

```
from  phievo.AnalysisTools.Notebook import Notebook,CellModule
from ipywidgets import interact, interactive, widgets
from IPython.display import display
```

Then create the *CellModule* object:

```
class DisplayFitness(CellModule):
    def __init__(self,Notebook):
        super(DisplayFitness, self).__init__(Notebook)
        self.button = widgets.Button(description="Display fitness",disabled=True)
        self.display_area = widgets.HTML(value=None, placeholder='<p></p>',
↪description='Fitness:')
        self.notebook.dependencies_dict["seed"].append(self)
        self.notebook.dependencies_dict["generation"].append(self)
        self.notebook.dependencies_dict["project"].append(self)
    def update_display(self,button):
        """
        Custom function that handles the button click and wrtie the fitness in the␣
↪HTML widget.
        """
        seed = self.notebook.seed
        gen = self.notebook.generation
        fit = str(self.notebook.sim.seeds[seed].generations[gen]["fitness"])
        self.display_area.value = "<p>{0}</p>".format(fit)
    def update(self):
        """
        Clear the HTML text and when the seed or the generation is updated.
        """
        if self.notebook.sim is None or self.notebook.seed is None or self.notebook.
↪generation is None:
            self.button.disabled=True
        else:
            self.button.disabled=False
        self.display_area.value="<p></p>"
    def display(self):
        """
```

```
        Display the button and the display area on one row.
        """
        self.button.on_click(self.update_display)
        display(widgets.HBox([self.button,self.display_area]))
```

Save the file and open the notebook to associate the newly created module to a notebook object.

```
...
from  phievo.AnalysisTools.Notebook import Notebook
import NB_Module

notebook = Notebook()
setattr(notebook,"display_fitness",NB_Module.DisplayFitness(notebook))
```

Now the display_fitness module can be used as any other *CellModule* by creating a new cell and running:

```
notebook.display_fitness.display()
```

A copy of the *NB_Module.py* file is available in the *Examples/* directory.

CHAPTER 6

Examples

ϕ-evo provides a series of examples of project and already run seeds.

6.1 Examples of projects

The example of projects are stored in the Example directory of the *phievo* package:

The function `download_example` allows to download one of the following examples:

- adaptation[1]
- somite[2]
- hox[3]
- hox_pareto
- lac_operon
- immune[4]
- minimal_project

```
import phievo
phievo.download_example("adaptation")
```

This command creates a project directory *example_adaptation* at your current path. The project contains all the configuration files required to start an evolution.

[1] François P, Siggia ED. A case study of evolutionary computation of biochemical adaptation. Physical Biology. 2008;5(2):26009.
[2] François P, Hakim V, Siggia ED. Deriving structure from evolution: metazoan segmentation. Molecular Systems Biology. 2007 Dec;3:9.
[3] François P, Siggia ED. Predicting embryonic patterning using mutual entropy fitness and in silico evolution. Development (Cambridge, England). 2010;137(14):2385–2395.
[4] Lalanne JB, François P. Principles of adaptive sorting revealed by in silico evolution. Physical Review Letters. 2013 May;110(21):218102.

6.2 Examples of seeds

Because some simulation can take some time to run, we provide the result seeds we used to generate the figure of the paper:

- seed_adaptation
- seed_adaptation_pruning
- seed_somite
- seed_somite_pruning
- seed_lacOperon
- seed_lacOperon_pruning
- seed_hox_pareto_light

To download the result of a simulation on your computer, you can use phievo:

```
import phievo
phievo.download_example("seed_adaptation")
```

The project downloaded can be analysed using the *Analyse Run.ipynb* notebook.

6.3 Hox pareto

The complete simulation for the Hox Genes takes a lot of space, only a portion of the original results is accessible through phievo.

You can manually download the complete simulation here.

6.4 References

CHAPTER 7

A simple example: the lactose operon

7.1 Description of the biological problem

The lactose operon is one of the most studied example in the regulation of proteins production. In *Escherichia coli*, the operon[1] encodes three different genes named lacZ, lacY and lacA from which the two firsts are the most importants. LacZ codes for a protein that hydrolizes lactose to produce glucose and galactose, which are themselves used by the cell as carbon sources. LacY encodes a permease, a protein which pumps the lactose into the cell. Both of these proteins need to be synthesized by the cell to use the lactose as an energy source, but as this is costly, and less efficient than using the glucose directly, the cell manages to produce them only in presence of lactose and in absence of glucose.

Cells have thus designed a logical gate, schematically shown in Figure 7.1 , to compute the binary function: *lactose and no glucose* that controls the expression of the whole operon. The biological strategy is the following: near the operon, the gene lacI encodes a repressor of the operon which is constitutively expressed so that by default, the operon is turned off. When lactose is present in the medium, a closed form, the allolactose is also present and will bind to the lacI repressor, thus impeding it to block the operon. It is now possible to expressed the operon but there is still no activation. The activator, the CAP protein, is indeed in an active form only in the presence of cAMP which is produced in absence of glucose[2]. As long as glucose is present, the operon is still silent and it is only when glucose become rare that cAMP goes high, thus activating the CAP protein which activate the operon and thus the production of the needed proteins.

Hereafter, we will run our genetic algorithm to optimize a function close from the logical gate corresponding to the lac operon, that is: $x, y \mapsto x \,\&\, \neg\, y$ (x nand y), the link with the biology of the real lac operon would nonetheless ask more work than will be presented here.

[1] In genetics, an operon is a functioning unit of DNA, it designates a cluster of genes under the control of a single promoter.

[2] For curious reader, the reason why, when energy tends to rarify, the cell suddenly produces an extraodinary amount of seemingly useless proteins is still an active question!

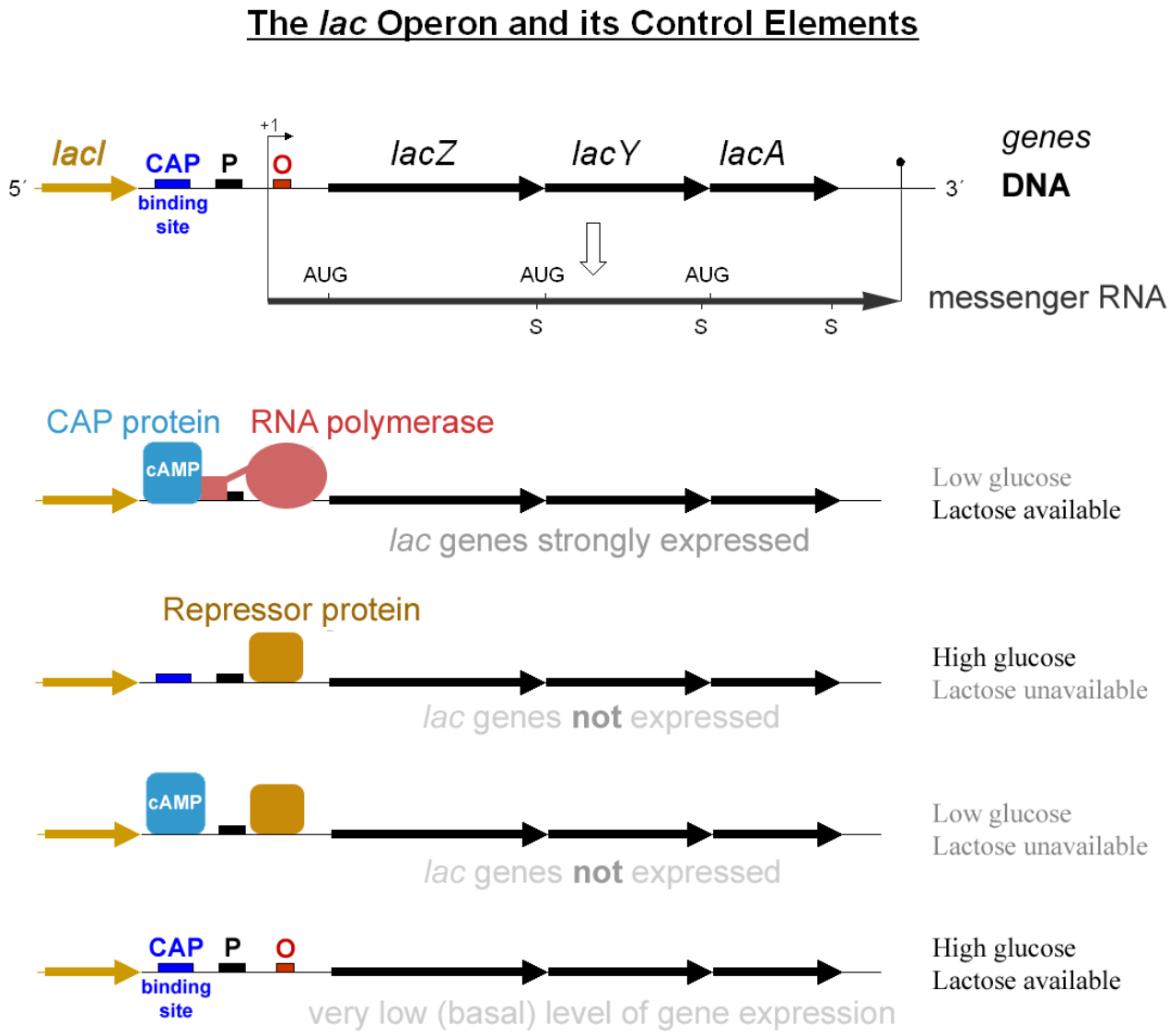

Fig. 7.1: Scheme presenting the main elements of the lactose operon along the DNA strain (top), and the state of the operon through several external conditions (bottom). Published on Wikimedia by G3pro and Tereseik.

7.2 Implementation in the algorithm

7.2.1 Remark

All files, functions and variables names along with terminal commands will be printed using the LaTeX environment verbatim and display with `this particular font`.

Two mains questions need to be answered in order to configure the algorithm for a particular problem. What? and How? : What is the precise function we need to optimize in order to describe the problem? and How the solution is allowed to be found by the algorithm? The first will be mainly described by the C code files like `init_history.c` and `fitness.c` while the second will be solved through the tuning of the various parameters in the so called `init*.py` file.

The `init_history.c` file describes the form of the input(s) that will be feed into the network. This is done through the construction of the double array `isignal[time][n_cell][n_input]` which indicates the concentration of the various input with respect to the time and cell.

In our case, we have two inputs that will represent the concentration of glucose and lactose and will be taken as binary functions (each sugar has a concentration of 0.0 or 1.0) which follow a random sequence of presence and absence, the time being spent in each state uniformly drawn between 10 and 60 seconds (see figure [fig:response_lo]).

The `fitness.c` file intend to process the output of the integrator which is rounded up in the double array named history indexed in the following way: `history[Species][Time][Cell]`. The variables `trackin` and `trackout` keeps in memory the label of the inputs and outputs species. The fitness is directly printed out by the `treatment_fitness` function. (Note however, that `treatment_fitness` is a void, fitness is passed with the `printf("%f",fitness);` statement.)

For lac operon simulation, each try of the integrator is treated independantly and follow the time course of the input and output to determine the times at which production is needed (that is when there is lactose and no glucose) and the concentration of the output at that time. We then have chosen to compute the mutual information[3] between lactose & ¬glucose and the concentration of the output.

Finally, the `init*.py` file indicate the mutation rates of the different interactions, the number of networks in the population, the number of generation of the simulation, the initial network from which we want to start and so on.

In the case of the lac_operon, we will ask the algorithm to use only protein-protein interaction (PPI) and repression/activation of gene (TFHill) and put to zero the parameters indicating the appearance of other interactions, for example:

```
random_Interaction('Degradation') = 0
random_Interaction('Phosphorylation') = 0
```

which control the rate at which new degradations and phosphorylations are added to the network to be probed by the evolution.

Each of this file has to be put in a single folder (in our case `lac_operon/`) in order to be found by the algorithm. Evolutionary procedure is now simply launched by running the

```
python run_evolution.py -m lac_operon
```

command line while in the main folder. The algorithm will now display a lot of more or less important stuff in your terminal. The most interesting are the generation number which indicate at which point of your simulation you are. When accustomed to it, the `Best_fitness` is an interesting variable to look at to know if the condition you defined actually allow the algorithm to find valid solution for the problem. Finally, every line starting by `ERROR` needs of course your special attention.

[3] The mutual information of two random variables is a way to quantify the information I can extract about one variable by measuring the second.

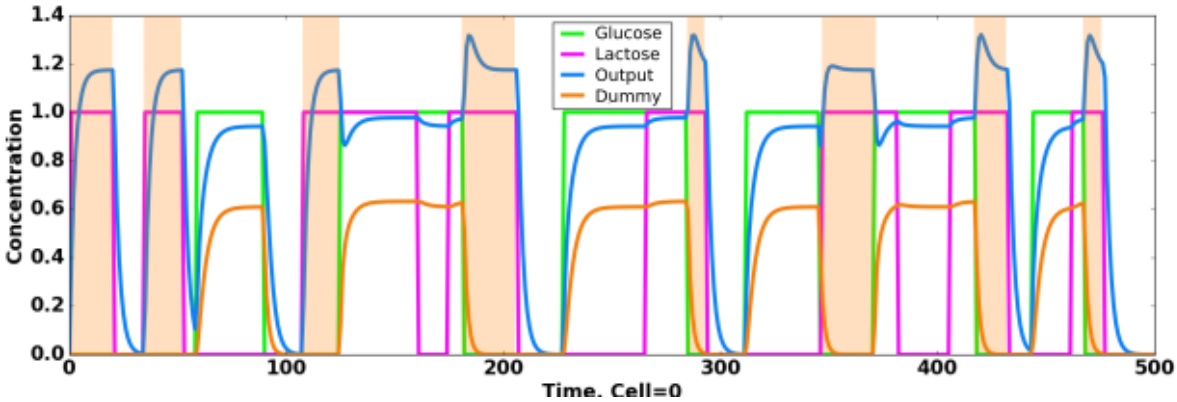

Fig. 7.2: Detailed response of the network presented in Figure 7.3 **A**, colors correspond between the two figures. Orange shades indicate the time at which response is waited.

7.2.2 A word about fitness

In order for the evolutionary procedure to give meaningful results, a special attention need to be given to design a proper fitness function. There is several reasons for this particular importance but the main one is that the algorithm will only try to solve the exact problem you have defined – i.e. minimize the fitness function you have provided – which is usually different from the actual task you have in mind.

For example, one of the solution proposed by the algorithm for the lac_operon fitness proposed earlier (the mutual information between the output concentration and the lactose & ¬glucose function) was to use lactose as a weak activator of the output and glucose as... a strong activator of the output! When looking at the time course of the output concentration, it makes plain sense because the concentration is near zero when there is no sugar, goes to one when there is only lactose and saturate around two when there is either glucose only or when both sugare are presents. Thus if the concentration is around one you know that you have lactose and no glucose. You can extract the whole information about the lactose & ¬glucose function from the output concentration which is the task we ask for, even if the answer was quite surprising.

This also mean that you will often want to modify your fitness function after a first bunch of runs to be more explicit or to try a different fitness function. To avoid being rapidly lost between your different simulation, you can look at the `Seed*/log_fitness.c` file for a reminder of the fitness used at this time.

A second remark about fitness is that the function should goes smoothly from the low fitness landscape to the region you want to explore, that is the fitness function should already rewards the first steps toward the solution. Otherwise, the algorithm will be stuck in the low level region and cannot even start to optimize. This question covers a broad range of litterature both in evolutionary biology and genetic algorithm computer science around the fitness-landscape shape question with suggestive names such as mount Fuji, house of cards or golf-course. It is usually not a big deal but could bring you some surprise if you don't keep it in mind.

7.3 How to read and interpret results

Now that your computer has run several simulations it is time to analyse them to decipher the output of the evolutionary algorithm. The first thing to look at is the time course of the fitness for several runs, to show the fitness of the first run, you can either use the `Analyse Run` notebook or use the Simulation class.

Make sure to check several runs to know the typical fitness of a successful or failed run, this will discard the cases where the evolutionary algorithm has been stuck and doesn't have enough time to converge.

To study a particular network, you can now type `network(500)` if you want to display the state of the best network in the population at generation 500 (the end of the simulation given our `init*.py` files). It may be small and concise but usually it's not, evolutionary procedure tends to accumulate a lot of uninteresting interactions and species – the famous DNA junk? –that may be ignored. Anyway, this is the raw result of the evolution. It will print out the file directory where the network has been saved for later analysis.

You can from there read and write network (with the read and write function), compute the fitness (with the `fitness` function) and even look at the time course of the species for a particular realisation of the fitness computation. If net is your network, just type `fitness(net, plot=True)`. You can also plot a network using `net.draw()`.

Finally, you can also add homebrew function to analyse your evolutionary result by adding a `analyse.py` file in the project folder. It will be imported with `analyse_network` through the name `spec`.

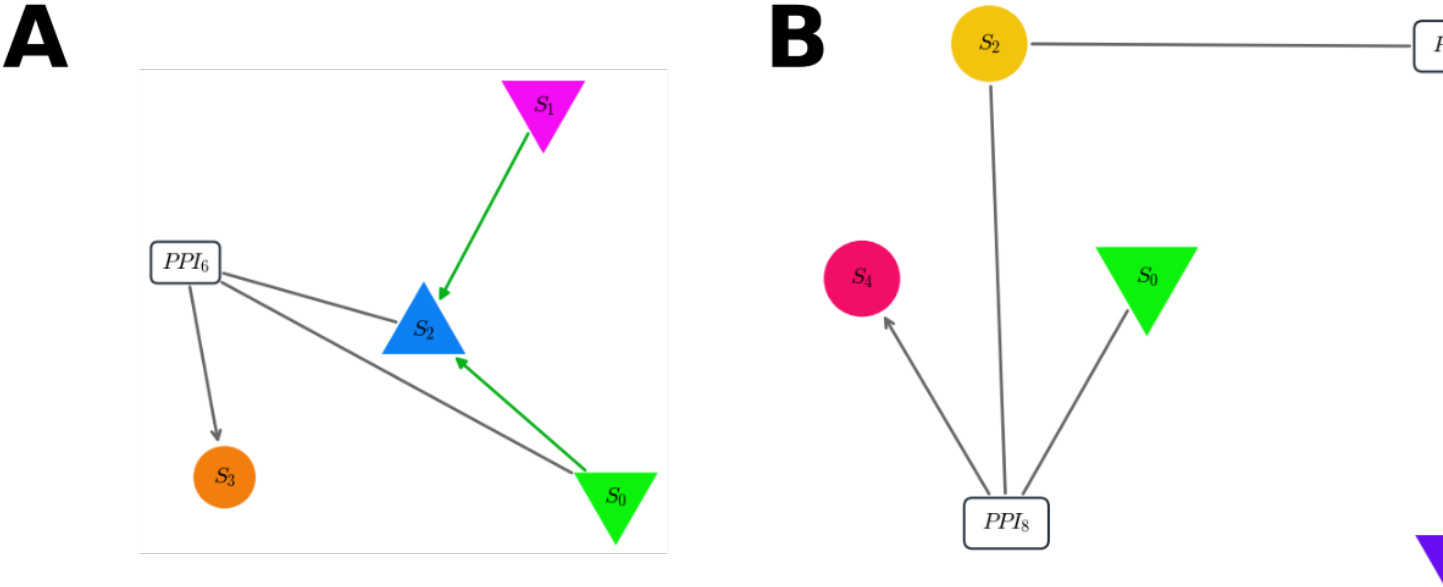

Fig. 7.3: Pannels **A.** and **B.** shows two typical topologies of the final result of the algorithm trying to optimize our mutual-information fitness. In both pannel, inputs are species 0 (glucose) and 1 (lactose) (down-triangle) and output is the up-triangle. **A.** Both sugars regulate positively the output, but the glucose also form a dimer with it thus impeding the response. The time course of this network is displayed in Figure 7.2. **B.** Here a single species (S2) can form two complexes, one very strongly with the glucose (S4), and another weaker with the lactose (S3). The former complex being the output.

In our case, out of 10 runs, 80% ended on 2 main different topologies (after pruning) both performing correctly, that is the fitness plateau around -0.8 on a scale of 0 to -1. Four correspond to the network of Figure 7.3 -**A** while four other looks like the one in Figure 7.3 - **B**. I let up to you the biological interpretation of these results[4] but the first obvious feature is the uniformity of the solution. Nearly all the successfull runs show very similar patern indicating that the biological grammar available actually imposes strong constraints on the possibles solution to a particular problem.

7.3.1 Geometry

7.3.2 New interactions

[4] Just a hint, for case **B** it seems to me that species 2 should be considered as the DNA strain!

CHAPTER 8

Create a new interaction

ϕ-evo allows you to add a custom interaction that is not available in the default list.
To do so you need to write an interaction file.

To make make the explanation clearer, we will explain how to build a new interaction on a real example of a methylation interaction.

The methylation adds a methyl group to a species S. The methylated species is denoted with a * symbol:

$$S \leftrightarrow S^*$$

We choose the simplest kinetics for this reaction:

$$\frac{dS^*}{dt} = -\frac{dS}{dt} = k_f S - k_b S^*$$

Let us start by creating the *Methyl.py* in a project directory.

8.1 Imports

Every interaction depends on the following ϕ-evo modules:

- classes_eds2: for the core structure of the intaction
- mutation: to handle mutation
- deriv2: to explain how to generate the C code associated to the new mutation

```
# In Methyl.py

from phievo import __silent__,__verbose__
if __verbose__:
    print("Execute Methyl (Interaction Template)")
```

```
from phievo.Networks import mutation
from phievo.Networks import deriv2
from phievo.Networks import classes_eds2
import copy
```

8.2 Define a new type of species

Only methylable species can be methylated. For now ϕ-evo does not know how to create a methylable species and what are its characteristics. There should be a few line telling how to do it:

```
# In Methyl.py
mutation.species_types["Methylable"] = lambda random_generator:[
    ["Methylable"],
    ['Diffusible',mutation.sample_dictionary_ranges('Species.diffusion',random_
↪generator)]
]
classes_eds2.Species.Tags_Species["Methylable"] = []
```

In the above lines, we tell ϕ-evo that a Methylable species has two characteristics:

- Methylable: obviously
- Diffusable: An extra characteristic is added to show how one would add a characteristics that comes with a parameter. A lambda function allows the program to generate new parameters when a new species is created.

Note: You can use the `mutation.sample_dictionary_ranges` to sample a random variable whose range has been define in `dictionary_ranges` in the *init* file. ### Set the default ranges for the parameters

```
# In Methyl.py

### Define the default dictionary_range
mutation.dictionary_ranges['Methyl.methyl'] = 0.0/(mutation.C*mutation.T)
mutation.dictionary_ranges['Methyl.demethyl'] = 0.0/mutation.T
```

8.3 Define the *Methyl* class

Every interaction in ϕ-evo inherits from the *classes_eds2.Interaction*:

```
# In Methyl.py
class Methyl(classes_eds2.Interaction):
    """
    Methylation interaction

    Args:
        Methyl.methyl(float): binding rate of a methyl group
        Methyl.demethyl(float): unbinding rate of a methyl group
        label(str): Methylation
        input(list): Type of the inputs
        output(list): Type of the outputs
    """
    def __init__(self,methyl=0,demethyl=0):
        classes_eds2.Node.__init__(self)
        self.methyl=methyl
        self.demethyl=demethyl
```

```
        self.label='Methylation'
        self.input=['Methylable']
        self.output=['Species']

    def __str__(self):
        """
        Used by the print function to display the interaction.
        """
        return "{0.id} Methylation: methyl. = {0.methyl:.2f}, demethyl = {0.demethyl:.
↪2f}".format(self)

    def outputs_to_delete(self,net):
        """
        Returns the methylated form of the species to delete when the reaction is␣
↪deleted.
        """
        return net.graph.successors(self)
```

The interaction's methods are the following:

- `__init__`: Creates the interaction object
- `__str__`: Produces the string used by the print function
- `outputs_to_delete`: Function that tells what are the species that were added to the network when the interaction was built and that need to be deleted when the interaction is removed.

8.4 Handling the mutation

The program needs five functions to tell ϕ-evo how to add the mutation via a mutation

8.4.1 number_Methyl

Evaluate the number of possible interactions of type *Methyl* that can be added to the network. This number is used to verify that the actual number of possible mutation found in `random_Methyl` is consistant with our intuition.

```
# In Methyl.py

def number_Methyl(self):
    """
    Returns the number of possible methylation in the current network.
    Note: this function is optional, it is used to check the consistency of
    the random_Methyl function.
    """
    n = self.number_nodes('Methylable')
    n_Methyl = self.number_nodes('Methyl')
    return n-n_Methyl
```

8.4.2 new_Methyl

This is the function that adds the *Methyl* interaction to the Network. It creates both a *Methyl* interaction and a *methylated species*.

```
# In Methyl.py
def new_Methyl(self,S,methyl,demethyl,parameters):
    """
    Creates a new :class:`Networks.Methyl.Methyl` and the species methylated for in␣
↪the the network.

    Args:
        S: species to methylate
        methyl(float): binding rate of a methyl group
        demethyl(float): unbinding rate of a methyl group
        parameters(list): Parameters of the methylated species
    Returns:
        [methyl_inter,S_methyl]: returns a Methyl interaction and a methylated␣
↪species.
    """

    S_methyl = classes_eds2.Species(parameters)
    meth_inter = Methyl(methyl,demethyl)
    assert meth_inter.check_grammar([S],[S_methyl]),"Error in grammar, new Methylation
↪"
    self.add_Node(S_methyl)
    self.add_Node(meth_inter)
    self.graph.add_edge(S,meth_inter)
    self.graph.add_edge(meth_inter,S_methyl)
    return [meth_inter,S_methyl]
```

Note: Then function needs a list of characteristics for the methylated species created. It is provide via `parameters`.

8.4.3 new_random_Methyl

Wrapping of the `new_Methyl` function. It generates randomly the rate of the methylation and the parameters of the methylated species created.

```
# In Methyl.py

def new_random_Methyl(self,S):
    """
    Creates a methylation with random parameters.

    Args:
        S: Species to methylate
    Returns:
        [methyl_inter,S_methyl]:returns a Methyl interaction and a methylated species.
    """
    parameters = {}
    if S.isinstance("TF"):
        parameters['TF'] = self.Random.random()*2
    for tt in S.types:
        if tt not in ["TF","Methylable","Input","Output"]:
            parameters[tt] = [mutation.sample_dictionary_ranges('Species.{}'.
↪format(attr),self.Random) for attr in S.Tags_Species[tt]]

    # Transform to fit phievo list structure
    parameters = [[kk]+val if val else [kk] for kk,val in parameters.items()]
    methyl = mutation.sample_dictionary_ranges('Methyl.methyl',self.Random)
    demethyl = mutation.sample_dictionary_ranges('Methyl.demethyl',self.Random)
    return self.new_Methyl(S,methyl,demethyl,parameters)
```

8.4.4 random_Methyl

Function called by the ϕ-evo to add a new Methylation interaction to the network during the evolution. It chooses a methylable species randomly and calls `new_random_Methyl` to add a methylation to this species.

```
# In Methyl.py

def random_Methyl(self):
    """
    Evaluates the species that can be phosphorilated, picks one an create a random
    methylation. The random mutation is made using :func:`new_random_Methyl <phievo.
↪Networks.classes_eds2.new_random_Methyl>`.

    Returns:
        [methyl_inter,S_methyl]: returns a Methyl interaction and a methylated␣
↪species.
    """
    try:
        list_methylable=self.dict_types["Methylable"]
    except KeyError:
        print("\tThe network contain no Methylacble species.")
        raise
    list_possible_methylable = []
    for S in list_methylable:
        if not self.check_existing_binary([S],"Methyl"):
            list_possible_methylable.append(S)
    n_possible = len(list_possible_methylable)
    assert n_possible==self.number_Methyl(),"The number of possible new methylation␣
↪does not match its theoretical value."
    if n_possible==0:
        if __verbose__:
            print("No more possible methylation.")
        return None
    else:
        S = list_possible_methylable[int(self.Random.random()*n_possible)]
        return self.new_random_Methyl(S)
```

8.4.5 Methyl_deriv_inC

Function that generates the C code string of the interaction kinetics.

```
# In Methyl.py

def Methyl_deriv_inC(net):
    """
    Function called to generate the string corresponding to in a methylation in C.
    """
    func_str = "\n/************** Methylations ****************/\n"
    methylations = net.dict_types.get("Methyl",[])
    for methyl_inter in methylations:
        S = net.graph.predecessors(methyl_inter)[0]
        S_meth = net.graph.successors(methyl_inter)[0]
        f_rate = "{M.methyl}*{S.id}".format(M=methyl_inter,S=S)
```

```
        b_rate = "{M.demethyl}*{S_m.id}".format(M=methyl_inter,S_m=S_meth)

        func_str += deriv2.compute_leap([S.id],[S_meth.id],f_rate)
        func_str += deriv2.compute_leap([S_meth.id],[S.id],b_rate)
    return func_str
```

8.5 Bind the code to ϕ-evo

The last step is to add all the functions written previously to the default `Mutable_Network`.

```
# In Methyl.py
setattr(classes_eds2.Network,"number_Methyl",number_Methyl)
setattr(classes_eds2.Network,"new_Methyl",new_Methyl)
setattr(classes_eds2.Network,"new_random_Methyl",new_random_Methyl)
setattr(classes_eds2.Network,"random_Methyl",random_Methyl)
deriv2.interactions_deriv_inC["Methyl"] = Methyl_deriv_inC
```

You can download Methyl.py from ϕ-evo's examples ### Edit the init file to load Methyl

The top of the init file should now be able to load the Methyl module with an import if the two files are in the same directory:

```
# In initialization.py
import Methyl
```

Now the new mutation settings are made similarly to any of the default interaction:

```
# In initialization.py

mutation.dictionary_ranges['Methyl.methyl'] = [0.1,1]
mutation.dictionary_ranges['Methyl.demethyl'] = [0.1,0.5]

dictionary_mutation['random_gene(\'Methylable\')'] = 0.1
dictionary_mutation['random_Interaction(\'Methyl\')']=0.1
dictionary_mutation['remove_Interaction(\'Methyl\')']=0.01
....
```

CHAPTER 9

Known Bugs

9.1 Disabling scrolling bar in Analyse Run.ipynb

cell>All Output>Toggle Scrolling

CHAPTER 10

phievo package

10.1 Networks module

10.1.1 classes_eds2

Defines the main class used to describe the evolved networks The class hierarchy is the following:

Network

- ***Node*:**
  - *Species*
  - *TModule*
  - ***Interaction*:**
    - *CorePromoter*
    - *TFHill*
    - *PPI*
    - *Phosphorylation*
    - other interactions

Types: Should be just the class, but for Species we have mutliple types (eg TF, Complex, Kinase, Phosphatase, Input, Output), several of which can apply at once, so the class defn in python not general enough.

IO: species of type = 'Input' has a defined time course supplied within the integrator.c. Type = 'Output' are the species whos time course is used by the fitness function, they are numbered and created as genes ie with TModule and CorePromoter nodes attached to them.

Grammar: the rules as for what can interact with what, depends on the type of interaction and the types of inputs and outputs. All the data for checking grammar given in the class defn of interaction. The grammar also enters the function, Network.remove_Node().

Class Network: defines a bipartite graphs with adjacent nodes either 'physical-objects, segments of the genome (eg Species or TModule) or interactions. Network class then has lots of methods to add and remove nodes and edges, check the grammar rules, and output the network either as C-code or as dot diagram .

Caps: Classes begin as caps, abbreviations eg TF in CAPS. Functions within classes lc, '_' to separate names, retain Caps for embedded class names.

Arguments to functions are in order implied by directed graph, eg check_grammar(nodes_in, node_tested, nodes_out) add_interaction(upstream_species, interaction, downstream species)

---

class phievo.Networks.classes_eds2.**Interaction**
Bases: *phievo.Networks.classes_eds2.Node*

Interaction class derived from Node, defines interaction between Species or TModule

check_grammar(*input_list*, *output_list*)
checks the grammar for the interactions

Parameters

- **input_list** (*list*) – nodes to be checked
- **output_list** (*list*) – nodes to be checked

Returns Boolean for the consistency of up and downstream grammar

class phievo.Networks.classes_eds2.**Network**
Bases: object

Complete description of a network of interactions.

It is represented as a bipartite graph between the biochemical species and the interactions. The very description is stored in the *graph* attribute.

Note that each interaction import add new methods to the Network class.

graph
networkx.MultiDiGraph – the network properly speaking

order_node
int – index to keep track of the order of the nodes

dict_types
dict – a dictionary indicating the Nodes of a given type (types are the keys)

hash_topology
int – to index the topologies (see __hash_net_topology__)

title
str – for graphing network and to hold misc info

Cseed
int – random seed for the integration in C

remove_output_when_duplicate
bool – if you want to remove Output tag when duplicating genes

activator_required
bool – if an activator is required to get any gene product

fixed_activity_for_TF
bool – if a TF either an activator or repressor (if False, they can do both)

Main functions: add_* methods just add objects to the graph new_* create and add objects (usually by calling add_* method)

add_CorePromoter2Species(*inter*, *output*)
Add a CorePromoter Interaction and its output to the network

Parameters

- **inter** (*CorePromoter*) – the CorePromoter to be added
- **output** (*Species*) – the CorePromoter output

add_Node(*node*)
add_node to graph unless already present

Parameters **node** (*Node*) – The node to be added

Returns boolean indicating if the node has effectively been added

add_TFHill(*tf*, *inter*, *module*)
Add a TF, a TModule and a *TFHill* interaction to the network

Parameters

- **tf** (*Species*) – with the 'TF' tag
- **inter** (*TFHill*) – will link tf and module
- **module** (*TModule*) – TModule to link the TFHill to

add_TModule2CorePromoter(*module*, *inter*)
Add a CorePromoter Interaction and its TModule to the network

Parameters

- **module** (*TModule*) – the CorePromoter module
- **inter** (*CorePromoter*) – the CorePromoter to be added

catal_data(*interaction*)
Find the reactants, catalysors, products for a catalytic interaction

Parameters **interaction** – the Interaction you are interested in

Returns list of the form [catalyst,reactants,products]

check_Node(*node*, *list_nodes_loop*)
Check if a Node can be removed from the network

Delegate to Node.isremovable check if node is not an input/output or a node uniquely and directly upstream of a nonremovable species (eg part of output gene)

Parameters

- **node** (*Node*) – the node to be checked
- **list_node_loops** (*list*) – to handle non tree like network

Return: Boolean indicates if node can be safely removed

check_existing_Degradation(*i1*, *i2*)
Check if a Degradation exists between species i1 and i2

Parameters

- **i1** (*Species*) – the 'enzyme'
- **i2** (*Species*) – the species degraded

Returns True if i1 is known to degrade i2

check_existing_Phosphorylation(*signature*)
check if a particular phosphorylation exists in the network

Parameters **signature** (*list*) – The signature of the phospho in the form [Kinase,Input]

Return: True if this phosphorylation exist

check_existing_binary(*list*, *Type*)
Check if a specific binary interaction of type Type already exists

typically used for PPI

Parameters

- **list** (*list*) – the reactants (Nodes) you are looking for
- **Type** (*str*) – the type of Interaction you are looking for

Return: bool

check_existing_link(*list*, *Type*)
Check if a specific interaction of type Type already exists between the elements of list

Parameters

- **list** (*list*) – the reactant/product couple (Nodes) you are looking for
- **Type** (*str*) – the type of Interaction you are looking for

Return: bool

clean_Nodes(*verbose=False*)
remove nodes from the network until all nodes pass the check_grammar test

Parameters **verbose** (*bool*) – Flag to activate the prolix mode

Return: Boolean indicating the completion of the process

Delete any node with incorrect grammar until all remaining nodes pass test Currently implemented to check grammar on interaction nodes only, thus need remove_Node function that kills species and other phys objects that are not defined in absence of interaction

delete_clean(*id*, *target='interaction'*, *verbose=False*)
Remove a node according to its id and clean the network Warning: This operation renames all the nodes (and changes the id)

Parameters

- **id** – integer id of the node
- **target** – string either interaction or species, the type of the node to delete

draw(*file=None*, *edgeLegend=False*, *extended=False*, *display=True*, *return_graph=False*)
Draw the network in a matplotlib framework

Delegate to *pretty_graph*

Parameters

- **file** (*str*) – save the picture in file, or print it on screen if file is None
- **edgeLegend** (*bool*) – Label the graph edges
- **extended** (*bool*) – Display inner modules (ex: TModules)
- **display** (*bool*) – Display the figure

- **return_graph** (*bool*) – Returns a graph object rather than a figure

Examples

my_Network.draw('my_lovely_network.pdf')

duplicate_PPI(*species*, *D_species*, *interaction*, *module*, *D_module*)
function to duplicate a PPI interaction

Parameters

- **species** (*Species*) – the original species
- **D_species** (*Species*) – the new species
- **interaction** (*PPI*) – the interaction you want to duplicate
- **module** (*TModule*) – the original module
- **D_module** (*TModule*) – the new module

duplicate_TFHill(*D_species*, *interaction*, *module*, *D_module*)
duplicate a TFHill interaction

Parameters

- **D_species** (*Species*) – the new species
- **interaction** (*TFHill*) – the interaction you want to duplicate
- **module** (*TModule*) – the original module
- **D_module** (*TModule*) – the new module

duplicate_downstream_interactions(*species*, *D_species*, *module*, *D_module*)
Called in case of gene duplication to copy the downstream interactions

Parameters

- **species** (*Species*) – the 'mother' species
- **D_species** (*Species*) – the 'daughter' species
- **module** (*TModule*) – the 'father' module
- **D_module** (*TModule*) – the 'son' module

duplicate_gene(*species*)
Duplicate a gene, i.e. a triplet Tmodule/CorePromoter/Species

Parameters **species** (*Species*) – Species to duplicate

Returns

list of the form [*new_TModule*, *new_CorePromoter*, *new_Species*, *old_TModule*]

- *new_TModule*: *TModule*
- *new_CorePromoter*: CorePromoter
- *new_Species*: *Species*
- *old_TModule*: *TModule*

or None if an error occured

duplicate_species_and_interactions(*species*)
Called to duplicate a species with its interactions

Right now only duplicates downstream TFHills and PPI and upstream TFHills. The input&output tags are removed from duplicate gene (see self.remove_output_when_dulicate)

Parameters **species** (*Species*) – the mother species

Returns A list [D_module,D_promoter,D_species] D_module (*TModule*): the duplicate TModule D_promoter (*CorePromoter*): the duplicate CorePromoter D_species (*Species*): the duplicate Species

get_node(*id*, *target='interaction'*)
Return the node correspoding to the specified id and target

Parameters

- **id** – integer id of the node
- **target** – string either interaction or species, the type of the node to search

list_possible_Degradation()
Return the list of all possible new degradations

new_Degradation(*Input1*, *Input2*, *rate*)
Create a new Degradation and add it to the network

Parameters

- **Input1** (*Species*) – the 'enzyme'
- **Input2** (*Species*) – the species degraded (have to be Degradable)
- **rate** (*float*) – the degradation rate

Returns list of the form [Degradation] or None if an error occured

new_PPI(*P1*, *P2*, *assoc*, *dissoc*, *types*)
Create a new `Networks.PPI.PPI`, its associated complex and add then to the network.

Parameters

- **P1** (*Species*) – First Protein
- **P2** (*Species*) – Second Protein
- **assoc** (*float*) – the association rate
- **dissoc** (*float*) – the dissociation rate of the complex
- **types** (*list*) – the types of the complex species

Returns

- *ppi*: *PPI*
- *complex created*: *Species*

Return type list of the form [*ppi*,'complex created'] with

new_Phosphorylation(*kinase*, *species*, *rate*, *threshold*, *hill*, *dephospho*)
Create a new Phosphorylation, its associated product and add them to the network.

Parameters

- **kinase** (*Species*) –
- **species** (*Species*) –

- **rate** (*float*) – the association rate
- **threshold** (*float*) – the Michaelis-Menten constant
- **hill** (*float*) – the hill coefficient of the reaction
- **dephospho** (*float*) – the dephosphorylation rate of the product

Returns list of the form [*Phosphorylation* , *Species*] or None if an error occured

new_Species(*types*)
Create a new Species instance and add it to the network

Parameters **types** (*list*) – the list types of the Species (see Species.__init__)

Returns The *Species* which have been created

new_TFHill(*tf*, *hill*, *threshold*, *module*, *activity=0*)
Create a new TFHill with given parameters and link it to the network.

Parameters

- **tf** (*Species*) – the upstrem Species
- **hill** (*float*) – the hill coefficient of the reaction
- **threshold** (*float*) – the Michaelis-Menten constant
- **module** (*TModule*) – the downstream TModule
- **activity** (*int*) – if fixed_activity_for_TF is True, always use the activity of tf

Returns return the new interaction or None if an error occured

Return type *TFHill*

new_custom_random_gene(*ltypes*)

new_enhancer(*species*, *rate*, *delay*, *parameters*, *basal=0.0*)
Create a complete new gene (TModule, CorePromoter and Species)

Parameters

- **species** (Species) –
- **rate** (*float*) – the rate production of the TModule
- **delay** (*int*) – the delay of the CorePromoter
- **parameters** (*list*) – the species parameter (see Network.new_Species)
- **basal** (*float*) – the basal production of the TModule (default to 0.)

Return: list of the form [Networks.classes_eds2.TModule, Networks.CorePromoter.CorePromoter]
or None if an error occured

new_gene(*rate*, *delay*, *parameters*, *basal=0.0*)
Create a complete new gene (TModule, CorePromoter and Species)

Parameters

- **rate** (*float*) – the rate production of the TModule
- **delay** (*int*) – the delay of the CorePromoter
- **parameters** (*list*) – the species parameter (see Network.new_Species)
- **basal** (*float*) – the basal production of the TModule (default to 0.)

Return: list of the form [Networks.classes_eds2.TModule, Networks.CorePromoter.CorePromoter,
or None if an error occured

number_Degradation()
Computes the number of possible Degradations

number_PPI()
Return the number of possible PPI in network

number_Phosphorylation()
Return the number of possible Phosphorylations

number_TFHill()
Return the number of possible TFHill

number_nodes(*Type*)
count the number of Nodes of type Type

Parameters **Type** (*str*) – the type you are looking for

Return: The number of Nodes of types Type in dict_types

propagate_activity_TFHill()
Ensure that TFHill activity correspond to the one of their predecessor - done for compatibility with older versions

remove_Node(*Node*)
remove node from the network graph

In case of interactions, also remove any phys objects (eg species, TModule) that are no more defined in absence of this interaction In the course of evolution, only interactions should be explicitly removed, then the other nodes are managed with the help of clean_nodes

Parameters

- **node** (*Node*) – The node to be removed
- **verbose** (*bool*) – Flag to activate the prolix mode

Return: Boolean indicating the completion of the process

store_to_pickle(*filename*)
Save the whole network in a pickle object named filename

Parameters **filename** (*str*) – the directory where the object is saved

verify_IO_numbers()
Redetermine all the input/output index

label_them run through the list and give the correct index to all the items

write_id()
Update all indexations of the network

Return: a number comprise between 0 and sys.maxint

class phievo.Networks.classes_eds2.**Node**
Bases: object

Superclass for all nodes object

id = 'None'

int_id()
extract the integer identifer computed in Network.write_id()

Returns int - the identifier of the Node None when valid int not found eg if write_id not called

isinstance(*name*)
check the type of a node

Customed the builtin isinstance(derived_class, base_class) for the general case

Parameters **name** – the type to be tested

Returns returns True if self is of type name

isremovable(*net*, *list_nodes_loop*, *verbose=False*)
Check if a Node can be removed from the network

Parameters

- **net** (Network) – the network self belongs to
- **list_nodes_loop** (*list*) – to handle non tree like network
- **debug** (*bool*) – Flag to activate a prolix version

Returns Boolean removable or not

list_types()
Return the list of types associated to a node

outputs_to_delete(*net*)
Indicates a list of objects to delete when removing the node from the network

Needs to be tuned specifically by all derived classes.

Parameters **net** (Network) – the network self belongs to

Returns list - default empty list

print_node()
print a full description of the current node

rand_modify(*random_generator*)
modify every parameters of the node self

This subroutine is then export to the *Node* class and used as a method Called the sample_dictionary_ranges subroutine when needed

Parameters

- **self** (*Node*) – the node you want to modify
- **random_generator** – a random number generator (.random() called here)

Returns in place modification

Return type None

string_param()
Returns a function with parameters for the nodes

Mainly here to be customized in subclasses

Returns string , default '.'

class phievo.Networks.classes_eds2.**Species**(*listtypes=[]*)
Bases: *phievo.Networks.classes_eds2.Node*

Class for any type of species, or list of species of various types Input list of lists eg [[Degradation, degradation], [TF, activity], [Complex,], [Kinase],.. [Output, n_put], [Input, n_put]] where n_put is an integer enumerating IO The first tag of ['Species'] is assumed and should not be input

Tags_Species = {'Phosphorylable': [], 'Phosphatase': [], 'Linear_Producer': [], 'Co

add_type(*Type*)
add Type and its corresponding parameters

Several layer of check are done before the core function to insure that Type is correctly added Also used to add the output/input tag to species. e.g.:species.add_type(['Output',n_put])

Parameters

- **Type** (*list*) – must be provided in a list of the form
- **as defined in Tags_Species** (*['Tag',parameter1,parameter2]*) –

Returns 1 if everything is done properly None if an error occur during the process

change_type(*Type*, *parameters*)
Change the parameters of a type

Parameters

- **Type** (*string*) – name of the type to modify
- **parameters** (*list*) – list of the new parameters as defined in the Tag_Species

clean_type(*Type*)
Removes a type and corresponding attributes from a species

def_label()
Function to write labels for graphical representation

default_tags = ['Degradable', 'Phosphorylable', 'Diffusible']

isinstance(*name*)
check the type of a node

Customed the builtin isinstance(derived_class, base_class) for the Species class

Parameters **name** – the type to be tested

Return: return True if self is of type name

label = 'Generic Species'

list_types()
Return the list of types associated to a node (custom for Species)

parameters = ['Degradable', 'Phosphorylable', 'Diffusible']

class phievo.Networks.classes_eds2.**TModule**(*rate=0*, *basal=0*)
Bases: *phievo.Networks.classes_eds2.Node*

A TModule regulate the production of a Species, it generally binds upstream to a CorePromoter (direct production) or a TFHill (regulation) and downstream to another TFHill which point to the product Species.

Parameters

- **rate** (*float*) – the production rate to be regulated
- **basal** (*float*) – the basal production rate

string_param()

phievo.Networks.classes_eds2.**check_consistency**(*lTypes*, *lNodes*)
Check the consistency between a list of types and a list of nodes

Typically used when constructing an interaction to check the biochemical grammar. For each type, it recursively checks if there is a corresponding node in list_nodes.

Parameters

- **lTypes** – the desired type of each node
- **lNodes** – the list of nodes

Returns Boolean indicating if the consistency is OK

phievo.Networks.classes_eds2.**compare_node**(*x*)
Used to order nodes in arbitrary but deterministic order when needed

Definition of CorePromoter Interaction. The CorePromoter is part of a *gene* system and binds a *TModule* to a *Species*.

---

class phievo.Networks.CorePromoter.**CorePromoter**(*delay=0*)
Bases: *phievo.Networks.classes_eds2.Interaction*

Core promoter for transcription of one species

The CorePromoter serve as a link between the TModule and the Species to preserve the bipartite nature of the network.

delay
int

label
str – 'transcription' by default

input
list – list of input types: ['TModule']

output
list – list of output types: ['Species']

outputs_to_delete(*net*)
indicate the Nodes to remove when deleting the CorePromoter

Parameters **net** (Network) – The network to which the CP belongs

Return: list of all the predec. and succ. of self in net

string_param()
Self description of the Interaction

phievo.Networks.CorePromoter.**add_CorePromoter2Species**(*self*, *inter*, *output*)
Add a CorePromoter Interaction and its output to the network

Parameters

- **inter** (*CorePromoter*) – the CorePromoter to be added
- **output** (*Species*) – the CorePromoter output

phievo.Networks.CorePromoter.**add_TModule2CorePromoter**(*self*, *module*, *inter*)
Add a CorePromoter Interaction and its TModule to the network

Parameters

- **module** (*TModule*) – the CorePromoter module
- **inter** (*CorePromoter*) – the CorePromoter to be added

phievo.Networks.CorePromoter.**duplicate_gene**(*self*, *species*)
Duplicate a gene, i.e. a triplet Tmodule/CorePromoter/Species

Parameters **species** (*Species*) – Species to duplicate

Returns

list of the form [*new_TModule*, *new_CorePromoter*, *new_Species*, *old_TModule*]

- *new_TModule*: *TModule*
- *new_CorePromoter*: CorePromoter
- *new_Species*: *Species*
- *old_TModule*: *TModule*

or None if an error occured

phievo.Networks.CorePromoter.**new_custom_random_gene**(*self*, *ltypes*)

phievo.Networks.CorePromoter.**new_enhancer**(*self*, *species*, *rate*, *delay*, *parameters*, *basal=0.0*)

Create a complete new gene (TModule, CorePromoter and Species)

Parameters

- **species** (*Species*) –
- **rate** (*float*) – the rate production of the TModule
- **delay** (*int*) – the delay of the CorePromoter
- **parameters** (*list*) – the species parameter (see Network.new_Species)
- **basal** (*float*) – the basal production of the TModule (default to 0.)

Return: list of the form [Networks.classes_eds2.TModule, Networks.CorePromoter.CorePromoter]
or None if an error occured

phievo.Networks.CorePromoter.**new_gene**(*self*, *rate*, *delay*, *parameters*, *basal=0.0*)

Create a complete new gene (TModule, CorePromoter and Species)

Parameters

- **rate** (*float*) – the rate production of the TModule
- **delay** (*int*) – the delay of the CorePromoter
- **parameters** (*list*) – the species parameter (see Network.new_Species)
- **basal** (*float*) – the basal production of the TModule (default to 0.)

Return: list of the form [Networks.classes_eds2.TModule, Networks.CorePromoter.CorePromoter, Netw
or None if an error occured

phievo.Networks.CorePromoter.**random_enhancer**(*self*, *Type='TModule'*)

Create a new random enhancer. It includes a TModule and a CorePromoter.

Parameters **Type** (*list*) – following the traditional template [‘type’, param]

Returns

- *tModule*: *TModule*
- *core_promoter*: *CorePromoter*

Return type list of the form [*tmodule*, *core_promoter*] with

phievo.Networks.CorePromoter.**random_gene**(*self*, *Type='Species'*)
Create a new random gene with a species of type Type

Parameters **Type** (*list*) – following the traditional template ['type', param]

Returns

- *tModule*: *TModule*
- *core_promoter*: *CorePromoter*
- *species*: *Species*

Return type list of the form [*tmodule*, *core_promoter*, *species*] with

10.1.2 Mutation

This module adds a layer to the elements defined in *classes_eds2* and creates an extended version of *Species* called *Mutable_Network*. The addon adds a set of tools for node mutations. For mutation/removal, effective mutation rate will be the reference mutation rate times the number of instances of the considered Type.

Attributes

- C,L,T (float):
- dictionary_mutation (dict): referenced mutation rates and associated command as key
- dictionary_ranges (dict): referenced parameters that can change and their ranges
- list_create (list): list of Nodes subject to creation
- list_mutate (list): list of Nodes subject to mutation
- list_remove (list): list of Nodes subject to removal
- list_types_output (list): list of the possible types for the output

---

class phievo.Networks.mutation.**Mutable_Network**(*generator=<random.Random object at 0x36b0878>*)
Bases: *phievo.Networks.classes_eds2.Network*

Expand the Network class with all functions related to mutation

the random_Type() routines are the only ones called by evolution to sample all possible links on graph that can give rise to given interaction Type and then choses one. The assignment of random interaction parametes and types of output, packaged in separate routines new_random_Type(), that can be used independently to generate specific topologies with random parameters .. attribute:: fitness

float – the fitness of the Network, None by default (worst than everyother number)

dlt_fitness
float – the change of fitness at the last generation

data_evolution
list – keep various information such as fitness variance, average...

data_next_mutation
list – field to keep the data on the next mutation

Random
Random – defines the local random generator number

Main functions:

build_mutations()
builds a dictionary with relative mutation rates for a specific network

This method is based on dictionary_mutation
Returns dict with the rates of each events for the network

compute_Cseed()
Return a random integer to determine the integrator seed

compute_next_mutation()
determine the time and type of next mutation for the gillespie algo.
Returns float time to next mutation
given a network, computes the time of the next mutation and the command to execute to perform the mutation for the gillispie algorithm

mutate_Node(*Type*)
randomly selects then mutates a Node of a given Type
Parameters **Type** (*str*) – the Type to mutate (e.g. Species: *Species*, TFHill: *TFHill*, Node: *Node <phievo.Networks.classes_eds2.Node>*...)
Returns boolean if something is mutated

mutate_and_integrate(*prmt*, *nnetwork*, *tgeneration*, *mutation=True*)
function to mutate, integrate and update the fitness

Note that compile_and_integrate is defined in Networks/deriv2.py
Parameters
- **prmt** (*dict*) –
- **nnetwork** (*int*) – an id for the C-file
- **tgeneration** (*float*) – the time before the next gen.
- **mutation** (*bool*) – if False, no mutation will be made

Returns
- n_mutations (int): the numbre of mutation performed
- nnetwork (int): same as args
- self (*Mutable_Network*): the Mutable_Network object itself
- result (list): output of treatment_fitness (see compile_and_integrate)

Return type List [n_mutations,nnetwork,self,result] where

new_random_Degradation(*Input1*, *Input2*)
Creates a Degradation with random parameters between the Species
Parameters
- **Input1** (*Species*) – the 'enzyme'
- **Input2** (*Species*) – the species degraded (have to be Degradable)

Returns list of of the form [Degradation]

new_random_PPI(*P1*, *P2*)
Creates a PPI with random parameters between the Species
Parameters
- **P1** (*Species*) – First protein
- **P2** (*Species*) – Second protein

Returns
- *ppi*: PPI
- *complex created*: *Species*

Return type list of the form [*ppi*,'complex created'] with

new_random_Phosphorylation(*kinase*, *species*)
Creates a Phosphorylation of species by kinase with random parameters
Parameters

- **kinase** (*Species*) – the kinase
- **species** (*Species*) – the species to Phosphorylate

Returns list of the form [*Phosphorylation* , *Species*] or None if an error occured

new_random_TFHill(*tf*, *module*)

Creates a TFHill between tf and module with random parameters

Parameters

- **tf** (*Species*) – must have the 'TF' tag
- **module** (*TModule*) – TModule associated to the TFHill

Returns return the new interaction or None if an error occured

Return type *TFHill*

random_Degradation()

Create new random Degradation among all possible ones

Returns of the form [Degradation] or None if an error occured

Return type list

random_Interaction(*Interaction_Type*)

create a new (and unique) interaction

Parameters **Interaction_Type** (*str*) – the type of interaction you want

Returns None

random_PPI()

Create new random PPI among all those possible

Returns

- *ppi*: *PPI*
- *complex created*: *Species*

Return type list of the form [*ppi*,'complex created'] with

random_Phosphorylation()

Creates a new Phosphorylation among all possibles

Returns list of the form [*Phosphorylation* , *Species*] or None if an error occured

random_Species(*Type='Species'*)

Create a new random species instance of a given type

Parameters **Type** (*str*) – the desired type of the new Species instance

Returns note that it is automatically added to the network

Return type *Species*

random_TFHill()

Creates a new *TFHill* among all possibles

Returns return the new interaction or None if an error occured

Return type *TFHill*

random_add_output()

Randomly adds an output tag to a random species

random_change_output()

Function that changes one output by adding then removing a TAG output

random_duplicate()

Routine to duplicate gene and its interactions

Currently the classes_eds2.duplicate_* only implemented for TF & PPI interactions If duplicating an output gene, add a new output tag to duplicated species, irrespective of other dictionary_mutation['*output*'] values in initialization.

Returns boolean indicating if a duplication has been finally done

random_enhancer(*Type='TModule'*)
Create a new random enhancer. It includes a TModule and a CorePromoter.
Parameters **Type** (*list*) – following the traditional template ['type', param]
Returns
- *tModule*: *TModule*
- *core_promoter*: *CorePromoter*

Return type list of the form [*tmodule*, *core_promoter*] with

random_gene(*Type='Species'*)
Create a new random gene with a species of type Type
Parameters **Type** (*list*) – following the traditional template ['type', param]
Returns
- *tModule*: *TModule*
- *core_promoter*: *CorePromoter*
- *species*: *Species*

Return type list of the form [*tmodule*, *core_promoter*, *species*] with

random_remove_output()
Removes at random an output tag on some species

Outputs are always index 0,1,2...; not possible to have 0,1,3 for instance

remove_Interaction(*Type*)
Randomly removes a Node of a given Type
Parameters **Type** (*str*) – the type you want to remove (e.g. '*Interaction*', *Species*, ...)
Returns boolean indicating if something is effectively removed

phievo.Networks.mutation.**build_lists**(*mutation_dict*)
Construct the index of Species types subject to various operation

Parameters **mutation_dict** (*dict*) – the dictionary listing the various operation (typically inits.dictionary_mutations)

phievo.Networks.mutation.**ligand_fct**(*random_generator*)

phievo.Networks.mutation.**rand_modify**(*self*, *random_generator*)
modify every parameters of the node self

This subroutine is then export to the *Node* class and used as a method Called the sample_dictionary_ranges subroutine when needed

Parameters

- **self** (*Node*) – the node you want to modify
- **random_generator** – a random number generator (.random() called here)

Returns in place modification

Return type None

phievo.Networks.mutation.**random_parameters**(*Types*, *random_generator*)
Create a set of new random parameters for a Species instance of type Types

This used only for initialization and adds attributes to various types. Some of which may not be mutable later

Parameters

- **Types** (*str*) – a species type
- **random_generator** – a random number generator (.random() called here)

Returns parameters a list of random parameters that can create a new Species or None if an error occured

phievo.Networks.mutation.**sample_dictionary_ranges**(*key*, *random_generator*)

Draw a random value for a parameter of type key

Look on dictionary_range, if the attribute to key is: a real number, a list or tuple of two reals defining min-max of range, and sample accordingly.

Parameters

- **key** (*str*) – the type of parameter you want
- **random_generator** – a random number generator (.random() called here)

Returns float a random value or int if key is CorePromoter.delay or None if an error occured

10.1.3 TFHill

Definition of TFHill interaction TFHill are mainly a convenient link between TModule and their regulating species. It is used to conserve the bipartite nature of the network.

---

class phievo.Networks.TFHill.**TFHill**(*hill=0*, *threshold=0*, *activity=0*)

Bases: *phievo.Networks.classes_eds2.Interaction*

Implement the link between TModule and the TF

Parameters

- **hill** (*float*) – the hill coefficient of the reaction
- **threshold** (*float*) – the Michaelis-Menten constant
- **activity** (*int*) – flag for activation (1) or repression (0)
- **label** (*str*) – 'transcription' by default
- **input** (*list*) – list of input types: ['TModule']
- **output** (*list*) – list of output types: ['Species']

string_param()

Self description of the Interaction

phievo.Networks.TFHill.**add_TFHill**(*self*, *tf*, *inter*, *module*)

Add a TF, a TModule and a *TFHill* interaction to the network

Parameters

- **tf** (*Species*) – with the 'TF' tag
- **inter** (*TFHill*) – will link tf and module
- **module** (*TModule*) – TModule to link the TFHill to

phievo.Networks.TFHill.**compute_transcription**(*net*, *module*)

Determine the transcription rate of a given module

Used for integration in transcription_deriv_inC

Parameters **module** (*TModule*) – TModule to compute .

Returns string the algebraic transcription rate of module

phievo.Networks.TFHill.**duplicate_TFHill**(*self*, *D_species*, *interaction*, *module*, *D_module*)
duplicate a TFHill interaction

Parameters

- **D_species** (*Species*) – the new species
- **interaction** (*TFHill*) – the interaction you want to duplicate
- **module** (*TModule*) – the original module
- **D_module** (*TModule*) – the new module

phievo.Networks.TFHill.**new_TFHill**(*self*, *tf*, *hill*, *threshold*, *module*, *activity=0*)
Create a new TFHill with given parameters and link it to the network.

Parameters

- **tf** (*Species*) – the upstrem Species
- **hill** (*float*) – the hill coefficient of the reaction
- **threshold** (*float*) – the Michaelis-Menten constant
- **module** (*TModule*) – the downstream TModule
- **activity** (*int*) – if fixed_activity_for_TF is True, always use the activity of tf

Returns return the new interaction or None if an error occured

Return type *TFHill*

phievo.Networks.TFHill.**new_random_TFHill**(*self*, *tf*, *module*)
Creates a TFHill between tf and module with random parameters

Parameters

- **tf** (*Species*) – must have the 'TF' tag
- **module** (*TModule*) – TModule associated to the TFHill

Returns return the new interaction or None if an error occured

Return type *TFHill*

phievo.Networks.TFHill.**number_TFHill**(*self*)
Return the number of possible TFHill

phievo.Networks.TFHill.**propagate_activity_TFHill**(*self*)
Ensure that TFHill activity correspond to the one of their predecessor - done for compatibility with older versions

phievo.Networks.TFHill.**random_TFHill**(*self*)
Creates a new *TFHill* among all possibles

Returns return the new interaction or None if an error occured

Return type *TFHill*

phievo.Networks.TFHill.**transcription_deriv_inC**(*net*)
gives the string corresponding to transcription for integration

Return: A single string for all transcriptions in the network

10.1.4 PPI

Definition of Protein-Protein-Interaction Creation: unknown Last edition: 2016-10-26

class phievo.Networks.PPI.**PPI**(*association=0*, *disassociation=0*)
Bases: *phievo.Networks.classes_eds2.Interaction*

Protein-protein interaction between two species

Parameters

- **association** (*float*) – the association rate
- **disassociation** (*float*) – the dissociation rate fo the complex
- **label** (*str*) – 'PP Interaction' by default
- **input** (*list*) – list of input types: ['Complexable','Complexable']
- **output** (*list*) – list of output types: ['Species']

check_grammar(*input_list*, *output_list*)
checks the grammar for the interactions (custom for PPI)

Parameters

- **input_list** (*list*) – nodes to be checked
- **output_list** (*list*) – nodes to be checked

Returns Boolean for the consistency of up and downstream grammar

outputs_to_delete(*net*)
Return the complex to delete when removing the LR

phievo.Networks.PPI.**PPI_deriv_inC**(*net*)
gives the string corresponding to Networks.PPI.PPI for integration

Returns str a single string for all Networks.PPI.PPI in the network

phievo.Networks.PPI.**duplicate_PPI**(*self*, *species*, *D_species*, *interaction*, *module*, *D_module*)
function to duplicate a PPI interaction

Parameters

- **species** (*Species*) – the original species
- **D_species** (*Species*) – the new species
- **interaction** (*PPI*) – the interaction you want to duplicate
- **module** (*TModule*) – the original module
- **D_module** (*TModule*) – the new module

phievo.Networks.PPI.**new_PPI**(*self*, *P1*, *P2*, *assoc*, *dissoc*, *types*)
Create a new Networks.PPI.PPI, its associated complex and add then to the network.

Parameters

- **P1** (*Species*) – First Protein
- **P2** (*Species*) – Second Protein
- **assoc** (*float*) – the association rate
- **dissoc** (*float*) – the dissociation rate of the complex
- **types** (*list*) – the types of the complex species

Returns

- *ppi*: *PPI*
- *complex created*: *Species*

Return type list of the form [*ppi*,'complex created'] with

phievo.Networks.PPI.**new_random_PPI**(*self*, *P1*, *P2*)
Creates a PPI with random parameters between the Species

Parameters

- **P1** (*Species*) – First protein
- **P2** (*Species*) – Second protein

Returns

- *ppi*: PPI
- *complex created*: *Species*

Return type list of the form [*ppi*,'complex created'] with

phievo.Networks.PPI.**number_PPI**(*self*)
Return the number of possible PPI in network

phievo.Networks.PPI.**random_PPI**(*self*)
Create new random PPI among all those possible

Returns

- *ppi*: *PPI*
- *complex created*: *Species*

Return type list of the form [*ppi*,'complex created'] with

10.1.5 Phosphorylation

Definition of Phosphorylation interaction

! WARNING: IF USING THIS CLASS PUT config.multiple_phospho to 0, otherwise you might have bugs (for now) TODO: in New Phosphorylation, test on n_phospho; if it is 1 (or higher than something) then remove Phosphorylable. Also update n_phospho accordingly when phosphorylated

phievo.Networks.Phosphorylation.**Phospho_deriv_inC**(*net*)
gives the string corresponding to Phosphorylation for integration

Returns A single string for all Phosphorylations in the network

class phievo.Networks.Phosphorylation.**Phosphorylation**(*rate=0*, *threshold=1*, *hill_coeff=1*, *dephospho_rate=1*)
Bases: *phievo.Networks.classes_eds2.Interaction*

Phosphorylation interaction

rate
float – the phosphorylation rate

threshold
float – the Michaelis-Menten constant

hill
float – the hill coefficient of the reaction

dephosphorylation
float – the dephosphorylation rate

label
str – 'Phosphorylation' by default

input
list – list of input types: ['Kinase','Phosphorylable']

output
list – list of output types: ['Kinase','Phospho']

check_grammar(*input_list*, *output_list*)
checks the grammar for the interactions (custom for Phosphorylation)

Parameters

- **input_list** (*list*) – nodes to be checked
- **output_list** (*list*) – nodes to be checked

Returns Boolean for the consistency of up and downstream grammar

outputs_to_delete(*net*)
Return the phosphorylated species to delete when deleting a Phosphorylation

phievo.Networks.Phosphorylation.**check_existing_Phosphorylation**(*self*, *signature*)
check if a particular phosphorylation exists in the network

Parameters **signature** (*list*) – The signature of the phospho in the form [Kinase,Input]

Return: True if this phosphorylation exist

phievo.Networks.Phosphorylation.**new_Phosphorylation**(*self*, *kinase*, *species*, *rate*, *threshold*, *hill*, *dephospho*)
Create a new Phosphorylation, its associated product and add them to the network.

Parameters

- **kinase** (*Species*) –
- **species** (*Species*) –
- **rate** (*float*) – the association rate
- **threshold** (*float*) – the Michaelis-Menten constant
- **hill** (*float*) – the hill coefficient of the reaction
- **dephospho** (*float*) – the dephosphorylation rate of the product

Returns list of the form [*Phosphorylation* , *Species*] or None if an error occured

phievo.Networks.Phosphorylation.**new_random_Phosphorylation**(*self*, *kinase*, *species*)
Creates a Phosphorylation of species by kinase with random parameters

Parameters

- **kinase** (*Species*) – the kinase
- **species** (*Species*) – the species to Phosphorylate

Returns list of the form [*Phosphorylation* , *Species*] or None if an error occured

phievo.Networks.Phosphorylation.**number_Phosphorylation**(*self*)
Return the number of possible Phosphorylations

phievo.Networks.Phosphorylation.**random_Phosphorylation**(*self*)
Creates a new Phosphorylation among all possibles

Returns list of the form [*Phosphorylation* , *Species*] or None if an error occured

10.1.6 Degradation

Definition of catalysed degradations.

---

class phievo.Networks.Degradation.**Degradation**(*rate=0.0*)
Bases: *phievo.Networks.classes_eds2.Interaction*

Catalyse the degradation of a given species

rate
float – the degradation constant

label
str – 'Degradation' by default

input
list – list of input types: ['Species','Degradable']

output
list – list of output types: ['Species']

check_grammar(*input_list*, *output_list*)
checks the grammar for the interactions (custom for degradation)

Parameters

- **input_list** (*list*) – nodes to be checked
- **output_list** (*list*) – nodes to be checked

Returns Boolean of the consistency of up and downstream grammar

outputs_to_delete(*net*)
indicate the Nodes to remove when deleting the Degradation

Parameters **net** (*Mutable_Network*) – The network to which the interaction belongs

Returns here an empty list

Return type list

phievo.Networks.Degradation.**Degradation_deriv_inC**(*net*)
gives the string corresponding to degradations for integration

Return:A single string for all degradation in the network

phievo.Networks.Degradation.**check_existing_Degradation**(*self*, *i1*, *i2*)
Check if a Degradation exists between species i1 and i2

Parameters

- **i1** (*Species*) – the 'enzyme'
- **i2** (*Species*) – the species degraded

Returns True if i1 is known to degrade i2

phievo.Networks.Degradation.**list_possible_Degradation**(*self*)
Return the list of all possible new degradations

phievo.Networks.Degradation.**new_Degradation**(*self*, *Input1*, *Input2*, *rate*)
Create a new Degradation and add it to the network

Parameters

- **Input1** (*Species*) – the 'enzyme'
- **Input2** (*Species*) – the species degraded (have to be Degradable)
- **rate** (*float*) – the degradation rate

Returns list of the form [Degradation] or None if an error occured

phievo.Networks.Degradation.**new_random_Degradation**(*self*, *Input1*, *Input2*)
Creates a Degradation with random parameters between the Species

Parameters

- **Input1** (*Species*) – the 'enzyme'
- **Input2** (*Species*) – the species degraded (have to be Degradable)

Returns list of of the form [Degradation]

phievo.Networks.Degradation.**number_Degradation**(*self*)
Computes the number of possible Degradations

phievo.Networks.Degradation.**random_Degradation**(*self*)
Create new random Degradation among all possible ones

Returns of the form [Degradation] or None if an error occured

Return type list

10.1.7 deriv2

Here are the tools to convert a Network object to a C-file that will be compiled and run. The C-file goes into workplace_dir/built_integrator*.c along with executable The C-file is assembled with several pieces:

- header, utilities, geometry, integrator and main: see initialization_code.init_deriv2
- for each interaction: see interaction.interaction_deriv_inC (bottom of file)
- see also Networks.interaction.py and the cfile dictionary

All these pieces are assembled by compute_program(), and then compiled with compile_and_integrate().

The c-code files passed only once in form of dictionary cfile. The numerical parameters need to find dimensions of arrays, integration steps, input as arguments to functions

phievo.Networks.deriv2.**workplace_dir**
str – the directory where build_integrator*.c will go

phievo.Networks.deriv2.**Ccompiler**
str – 'gcc' by default

phievo.Networks.deriv2.**cfile**
dict – where the generic c-code are found (can be reset to fit problem)

phievo.Networks.deriv2.**noise_flag**
bool – flag to know if we integrate or not with noise

TODO: it would be nice to include in header.h declaration of all C functions used so that they can then be loaded in any order, currently order constrained by declare before use.

phievo.Networks.deriv2.**all_params2C**(*net*, *prmt*, *print_buf*, *Cseed=0*)

Collect all the numerical constants and format them to C like

neelocalneig,diff,index_ligand,ded

Parameters

- **net** (*Mutable_Network*) –

  –

- **prmt** (*dict*) – dictionary from initialization file
- **print_buf** (*bool*) – control printing of time history by C codes
- **Cseed** (*int*) – seed for the integrator random number generator

Returns A C formated string of parameters

phievo.Networks.deriv2.**compile_and_integrate**(*network*, *prmt*, *nnetwork*, *print_buf=False*, *Cseed=0*)

Compile and integrate a network

Wait for process completion before launching another integration See https://www.python.org/dev/peps/pep-0324/ for interface to run C code

Parameters

- **network** (*Mutable_Network*) –

  –

- **prmt** (*dict*) – dictionary from initialization file
- **nnetwork** (*int*) – an id to separate the different C-file
- **print_buf** (*bool*) – control printing of time history by C codes to a file
- **Cseed** (*int*) – seed for the integrator random number generator

Returns list of corresponding to the different line of the output of treatment_fitness (see your fitness.c file) or None if an error occured

phievo.Networks.deriv2.**compute_leap**(*list_input_id*, *list_output_id*, *rate*)

Routine to compute strings for derivative in C associated to an interaction

if noise_flag, adds a Langevin noise term which scaled with concentration

Parameters

- **list_input_id** (*list*) – contains id of the input, i.e. the depleted species
- **list_output_id** (*list*) – contains id of the created species
- **rate** (*str*) – the rate, should be positive

Returns a C-formatted string

phievo.Networks.deriv2.**degrad_deriv_inC**(*net*)

gives the string corresponding to the degradation integration

Returns A single string for all degradations in the network

phievo.Networks.deriv2.**track_changing_variable**(*net*, *name*)
Return a list of the indices of the species with type name

Use this function when Output or Input may be added (we do not care about their order)

Parameters

- **net** (*Mutable_Network*) –

  –

- **name** (*str*) – a Species tag, usually 'Input' or 'Output'

Returns list of the id species list ordered by growing n_put

phievo.Networks.deriv2.**track_variable**(*net*, *name*)
Return a list of the indices of the species with type name

This is way of keeping track of fixed IO variables. Use this function only if the output or input are fixed in the algorithm, otherwise, use track_changing_variable

Parameters

- **net** (*Mutable_Network*) –

  –

- **name** (*str*) – a Species tag, usually 'Input' or 'Output'

Returns list of the id species list ordered by growing n_put

phievo.Networks.deriv2.**write_program**(*programm_file*, *net*, *prmt*, *print_buf*, *Cseed=0*)
Write the built_integrator of the network in the C file

Collect python encoded C and the stored files selected via cfile dictionary and write them in the correct order.

Parameters

- **programm_file** (*TextIOWrapper*) – the built_integrator file
- **net** (*Mutable_Network*) –

  –

- **prmt** (*dict*) – passed to all_params2C
- **print_buf** (*bool*) – passed to all_params2C
- **Cseed** (*int*) – passed to all_params2C

Returns The C programm as a python string

10.1.8 lovelyGraph

The lovelyGraph modules contains a set of utilities to plot a network. It uses the homemade package *PlotGraph*.

phievo.Networks.lovelyGraph.**gettype**(*node*, *type_list*)

phievo.Networks.lovelyGraph.**pretty_graph**(*net*, *extended=True*, *layout='graphviz'*)
Creates a ready-to-plot graph object from a network.

Parameters **net** (*Mutable_Network*) –

Returns returns a *PlotGraph graph*

phievo.Networks.lovelyGraph.**produce_CorePromoter_name**(*node_reac*)

phievo.Networks.lovelyGraph.**produce_Degradation_name**(*node_reac*)

phievo.Networks.lovelyGraph.**produce_PPI_name**(*node_PPI*)

phievo.Networks.lovelyGraph.**produce_Phospho_name**(*node_reac*, *cat=False*)

phievo.Networks.lovelyGraph.**produce_TFHill_name**(*node_reac*)

phievo.Networks.lovelyGraph.**produce_TModule_name**(*node_species*)

phievo.Networks.lovelyGraph.**produce_species_name**(*node_species*)

phievo.Networks.lovelyGraph.**short_label**(*species*)

10.2 PlotGraph

10.2.1 Graph

class phievo.Networks.PlotGraph.Graph.**Graph**(*layout*)

Bases: object

Container of a directed graph. It contains mainly two types of objects: nodes and edges.

add_edge(**argv*, ***kwargs*)

Add an edge to the graph.

Parameters

- **argv** (*list(str)*) – Is handled if it contains only two elements corresponding to the edge's starting and ending nodes.
- **kwargs** (*dict*) – The function handles only the keys **style**, **label** that respectively correspond to the edge's style and its label. It can also deal with **nodeFrom** and **nodeFrom** if it was not defined in argv. The other keys are passed for latter use by the plotting function.

Returns Networks.PlotGraph.Components.Edge:The edge reference.

add_node(**argv*, ***kwargs*)

Add a node to the graph.

Parameters

- **argv** (*list(str)*) – Is handled if it contains only one element corresponding to the node label.
- **kwargs** (*dict*) – The function handles only the keys **size**, **marker** that respectively correspond to the node's area and its shape. It can also deal with **label** if it was not defined in argv. The other keys are passed for latter use by the plotting function.

Returns Networks.PlotGraph.Components.Node: The node reference.

draw(*file=None*, *edgeLegend=False*, *display=True*)

Draw the graph in a matplib framework. The node and edges are generated using patches.

Parameters **file** (*str*) – Optional. When defined, the figure will be saved under the **file** name. Otherwise the program pops up a window with the graph.

Returns None

edge_list()

Generate a list of the node edges

Returns Each tuple in the list contains the starting and ending node labels.

Return type list((str,str))

get_networkx()

layout(*recursion=500*)
Compute a layout for the node and set the node positions.

node_list()
Generate a list of the node labels

Returns of the labels for the node contained in the graph

Return type list(str)

set_node_size(*size*)
Homogenise the node area in the network.

Parameters **size** (*float*) – Relative node area as compare to the default area.

Returns None

10.2.2 Graph components

class phievo.Networks.PlotGraph.Components.**Arrow**(***kwargs*)
Bases: *phievo.Networks.PlotGraph.Components.Edge*

The class arrow is inherited from Networks.PlotGraph.Components.Edge. It adds extra fonctionalities to generate Matplolib patches.

get_autoPatch(*offsets=(0, 0)*, *num=0*)
Generates a matplotlib patch for the arrow between two nodes. It takes into account the offset to keep between the ends of the arrow and the node given the node shape. This is an implementation of get_vector for a edge that start and ends at the same node.

Parameters **offsets** (*float*, *float*) – offset between node and the start of the arrow and offset between node and the end of the arrow

Returns Matplotlib.Patches

get_patch(*offsets=(0, 0)*, *angle=0.2*)
Generates a matplotlib patch for the arrow between two nodes. It takes into account the offset to keep between the ends of the arrow and the nodes given the node shapes.

Parameters **offsets** (*float*, *float*) – offset between node1 and the start of the arrow and offset between node2 and the end of the arrow

Returns Matplotlib.Patches

class phievo.Networks.PlotGraph.Components.**BarB**(*widthB=0.4*, *angleB=None*)
Bases: matplotlib.patches._Bracket

An arrow with a bar(|) at the B end. The class is added to matplotlib to allow "-|" style of arrow.

phievo.Networks.PlotGraph.Components.**Bezier**(*P0*, *P1*, *P2*)

class phievo.Networks.PlotGraph.Components.**Circle**(**args*, ***kwargs*)
Bases: *phievo.Networks.PlotGraph.Components.Node*

Circle is inherited from Networks.PlotGraph.Components.Node and represents a node with a circular shape ().

get_patch()
Draw of a matplotlib patch to be added to the graph plot.

Returns Matplotlib.Patch

radius(*theta*)
Every point on the node's boundary is refered to by an angle in rad. Given the shape of the node, compute the radius of the boundary for a angle.

$$\theta \rightarrow R$$

Parameters **theta** (*float*) – Angle a which to compute the distance between the center and the boundary.

Returns corresponding to the radius.

Return type float

class phievo.Networks.PlotGraph.Components.**Edge**(*nodeFrom*, *nodeTo*, *label*, ***kwargs*)
Bases: object

Directed graph edge between two nodes.

compute_center(*A*, *B*, *angle*)

get_vector(*offsets=(0, 0)*, *angle=0*)
Generate a starting and ending point of the edge's arrow that accomodates the desired space and between the arrow and the nodes given the node shapes.

Parameters

- **offsets** (*float*, *float*) – offsets between the arrow and the two nodes
- **angle** (*float*) – If angle is 0, the arrow follows a straigh line between two nodes. Otherwise it is a curved line starting and arriving to the node with two opposite angles with respect to the freeAngle value

Returns

tuple containing:

- start (numpy.array): Start of the arrow
- end (numpy.array): End of the arrow

Return type (tuple)

get_vector_auto(*offsets=(0, 0)*, *num=0*)
Generate a starting and ending point of the edge's arrow that accomodates the desired space and between the arrow and the node given the node shapes. This is an implementation of get_vector for a edge that start and ends at the same edge.

Parameters

- **offsets** (*float*, *float*) – offsets between the arrow and the two nodes
- **angle** (*float*) – Here the angle cannot be 0. The arrow is a curved line starting and arriving to the node with two opposite angles with respect to the freeAngle value.

Returns

tuple containing:

- start (numpy.array): Start of the arrow
- end (numpy.array): End of the arrow

Return type (tuple)

radius(*theta*)

record_angle(*angle*)

setReceiveEdge()

set_center(*center*)

class phievo.Networks.PlotGraph.Components.**HouseDown**(**args*, ***kwargs*)
Bases: *phievo.Networks.PlotGraph.Components.Node*

Node with a pentagon shape ().

get_patch()
Draw of a matplotlib patch to be added to the graph plot.

Returns Matplotlib.Patch

radius(*theta*)
Every point on the node's boundary is refered to by an angle in rad. Given the shape of the node, compute the radius of the boundary for a angle.

$$\theta \to R \times \frac{\cos \pi/5}{\cos((5\theta - 3\pi/2)\%(2pi)/5 - \pi/5)}$$

Parameters **theta** (*float*) – Angle a which to compute the distance between the center and the boundary.

Returns corresponding to the radius.

Return type float

class phievo.Networks.PlotGraph.Components.**HouseUp**(**args*, ***kwargs*)
Bases: *phievo.Networks.PlotGraph.Components.Node*

Node with a pentagon shape ()

get_patch()
Draw of a matplotlib patch to be added to the graph plot.

Returns Matplotlib.Patch

radius(*theta*)
Every point on the node's boundary is refered to by an angle in rad. Given the shape of the node, compute the radius of the boundary for a angle.

$$\theta \to R \times \frac{\cos \pi/5}{\cos((5\theta + 3\pi/2)\%(2pi)/5 - \pi/5)}$$

Parameters **theta** (*float*) – Angle a which to compute the distance between the center and the boundary.

Returns corresponding to the radius.

Return type float

class phievo.Networks.PlotGraph.Components.**Interaction**(*node1*, *node2*)
Bases: object

In the module Graph, an iteraction between node A and node B stands for at least one edge between those two node. It is a mean to keep tracks of all the edges that exist between A and B.

add_edge(*edge*)
Add an edge to an the existing interaction

Parameters **edge** (*Edge*) – edge to be added to the list of edge references

Returns None

get_patches(*offsets=(0, 0)*)
Run through the interactions edges to create a Matplotlib patch for each of them

Parameters **offsets** (*float, float*) – Size 2 tuple containing the offset to leave between the edges an the node1 and node2.

Returns list of Matplotlib patches

Return type [Matplotlib.Patches]

class phievo.Networks.PlotGraph.Components.**Line**(***kwargs*)
Bases: *phievo.Networks.PlotGraph.Components.Edge*

class phievo.Networks.PlotGraph.Components.**Node**(*label*, *size*, **args*, ***kwargs*)
Bases: object

Directed graph node or vertex.

find_freeAngle()
Searches for the best position where to add a new edge to the node. It is used only for looping edges. It tries to increase the angle between the new angle and the already plotted edges.

Parameters **angle** (*float*) – Value between 0 and 2π where an new edge arrives or leaves the node.

Returns the function returns the optimal angle

Return type float

plot_label()
Write the node label on the plot a the node's center.

record_angle(*angle*)
Every point on boundary of the Node is refered to by an angle. This function records the postition each time a new edge is drawn. The list of angle is used to choose the optimal position where to add looping edges.

Parameters **angle** (*float*) – Value between 0 and 2π where an new edge arrives or leaves the node.

Returns None

set_center(*pos*)
Set the coordinates of the node's center.

Parameters **pos** (*list(float)*) – Coordinates of the node's center

Returns None

class phievo.Networks.PlotGraph.Components.**RoundedRectangle**(**args*, ***kwargs*)
Bases: *phievo.Networks.PlotGraph.Components.Node*

Node with a RoundedRectangle shape ().

get_patch()
Draw of a matplotlib patch to be added to the graph plot.

Returns Matplotlib.Patch

radius(*theta*)
Every point on the node's boundary is refered to by an angle in rad. Given the shape of the node, compute the radius of the boundary for a angle.

Parameters **theta** (*float*) – Angle a which to compute the distance between the center and the boundary.

Returns corresponding to the radius.

Return type float

class phievo.Networks.PlotGraph.Components.**Square**(**args*, ***kwargs*)
Bases: *phievo.Networks.PlotGraph.Components.Node*

Node with a square shape ().

get_patch()
Draw of a matplotlib patch to be added to the graph plot.

Returns Matplotlib.Patch

radius(*theta*)
Every point on the node's boundary is refered to by an angle in rad. Given the shape of the node, compute the radius of the boundary for a angle.

$$\theta \to R \times \frac{\cos \pi/4}{\cos((4\theta + 2\pi/2)\%(2pi)/4 - \pi/4)}$$

Parameters **theta** (*float*) – Angle a which to compute the distance between the center and the boundary.

Returns corresponding to the radius.

Return type float

class phievo.Networks.PlotGraph.Components.**TriangleDown**(**args*, ***kwargs*)
Bases: *phievo.Networks.PlotGraph.Components.Node*

Node with a triangle shape ().

get_patch()
Draw of a matplotlib patch to be added to the graph plot.

Returns Matplotlib.Patch

radius(*theta*)
Every point on the node's boundary is refered to by an angle in rad. Given the shape of the node, compute the radius of the boundary for a angle.

$$\theta \to R \times \frac{\cos \pi/3}{\cos((3\theta + 3\pi/2)\%(2pi)/3 - \pi/3)}$$

Parameters **theta** (*float*) – Angle a which to compute the distance between the center and the boundary.

Returns corresponding to the radius.

Return type float

class phievo.Networks.PlotGraph.Components.**TriangleUp**(**args*, ***kwargs*)
Bases: *phievo.Networks.PlotGraph.Components.Node*

Node with a triangle shape ().

get_patch()
Draw of a matplotlib patch to be added to the graph plot.

Returns Matplotlib.Patch

radius (*theta*)

Every point on the node's boundary is refered to by an angle in rad. Given the shape of the node, compute the radius of the boundary for a angle.

$$\theta \to R \times \frac{\cos \pi/3}{\cos((3\theta - 3\pi/2)\%(2pi)/3 - \pi/3)}$$

Parameters **theta** (*float*) – Angle a which to compute the distance between the center and the boundary.

Returns corresponding to the radius.

Return type float

10.2.3 Layout

phievo.Networks.PlotGraph.Layout.**hierarchical_layout**(*node_list*)

phievo.Networks.PlotGraph.Layout.**layout**(*node_list*, *interaction_list*, *radius=1*, *layout='graphviz'*)

Use networkx layout function to compute the node centers

Parameters

- **node_list** (*list*) – List of all the nodes in the nework
- **interaction_list** (*list*) – List of tuple describing the nodes in interaction
- **radius** (*float*) – Order of magnitude for a node radius. used to scale the minimal distance.
- **layout** (*str*) – Use a networkx layout. Choose between: - circular - spring - shell - random - spectral - circular - fruchterman_reingold - pygraphviz

Returns indexed by nodes names and containing their (x,y) position (for use with draw_networkx pos argument typically)

Return type dict

10.3 Populations

10.3.1 Default evolution

Defines the Class Population with her principal method, evolution, which evolve a set of networks. All initialization done from an initialization.py file. All the modules are initialized through run_evolution.py.

The initial networks to evolve, can be built from just the input/output genes, a predefined newtork, or restarted from any saved population from a previous run. (See initialization file for details)

The time between generations is variable, and about the same for all species, we sample the mutation rates with a gillespie like algorithm, hence the name

The evolution method will write the following files in the namefolder given as argument to Population.__init__ stdout basic info each generation: * Bests = for generation, the network with best fitness in text form to edit or process with stat_best_net.py * Restart* = binary dbm type file with data to restart evolution at selected generation numbers * graphic files with time course and best network diagram at selected generations

class phievo.Populations_Types.evolution_gillespie.**Population**(*namefolder*)
Bases: object

Define a population as a list of networks called Population. Genus and a principal method evolution. object means it is a newstyle class ! See the web for distinction between new and olds style class, important for inheritance

best_fitness
float – keep trace of the best fitness in the population

genus
list – the list of individuals(*Network*) of the population

same_seed
bool – indicate if the file is a restart or not

tgeneration
float – starting hop time for the gillespie algorithm

npopulation
int – size of te population

bests_file
str – directory to save the data of evolution

Main methods: evolution: launch the evolutionary algorithm pop_mutate_and_integrate: update the whole population

evolution(*prmt*)
Main method to evolve population

Returns None

genus_mutate_and_integrate(*prmt*, *nnetwork*, *mutation=True*)
mutate, and update the fitness of one individual

Parameters

- **prmt** (*dict*) – the inits parameters for integration
- **nnetwork** (*int*) – the index of the network in the population
- **mutation** (*bool*) – a flag to activate mutation

Returns the number of mutation int: the index of the network in the population Network: The resulting network after mutation

Return type int

increment_identifier(*network*)
Test whether the network was mutated. If so the network identifier is updated with a new index.

initialize_identifier()
Set an unique index to every network of the initial population an set the max_network_identifier value. If the run restarts an existing simulation, only max_network_identifier is computed.

pop_mutate_and_integrate(*initial*, *first_mutated*, *last_mutated*, *prmt*, *net_stat*)
Recompute the fitness for half the population and mutate/compute the fitness for the rest. Save all the data in net_stat

Parameters

- **initial** (*int*) – index of the first individual in population

- **first_mutated** (*int*) – index of the first mutated individual in population
- **last_mutated** (*int*) – index of the last mutated individual in population
- **prmt** (*dict*) – the inits parameters for integration
- **net_stat** (*NetworkStat*) – to store the population data

Returns in place modification

Return type None

pop_sort()
Sort the population with respect to fitness

save_restart_file(*kgeneration*, *header*, *tgeneration*)
Save a dbm file, keyed by the generation number (a string!) and with value a [parameter dictionary, genus]. Might be more transparent to write out Poulation instance and forget header, and be sure to update tgeneration

storing(*t_gen*, *net*)
Store the work and various data for later analysis

Network object are stored in individual pickle file in Seed{}/data Data are stored in a shelve called the Seed{}/Bests_{}.net

Parameters

- **t_gen** – the key (normally the generation number)
- **net** (*Network*) – the object to be saved

Returns None

update_fitness(*nnetwork*, *integration_result*)
Update (in place) the fitness and the dlt_fitness

Parameters

- **nnetwork** (*int*) – the index of the network in the population
- **integration_result** (*list*) – the output of compile_and_integrate

Returns in place modification

Return type None

phievo.Populations_Types.evolution_gillespie.**fitness_treatment**(*population*)
default function for fitness treatment

If necessary, should be implemented in the init*.py file

phievo.Populations_Types.evolution_gillespie.**init_network**(*mutation*)
Default function to create network

It must be overwritten with function from the init*.py file otherwise stop the programm

phievo.Populations_Types.evolution_gillespie.**restart**(*directory*, *generation*, *verbose=True*)
Allow the user to restart an old run

Parameters

- **directory** (*str*) – the directory of the restart file
- **generation** (*int*) – the generation number

Returns the parameters of the run genus (list): the list of individuals(*Network*) of the population

Return type rprmt (dict)

10.3.2 Pareto evolution

This module provide a pareto_Population class to perform a Pareto evolution, that is, a general frame to evolve Networks according to more than one fitness function.

See: Warmflash, A., Francois, P., & Siggia, E. D. (2012). Pareto Evolution of Gene Networks: An Algorithm to Optimize Multiple Fitness Objectives. Physical Biology, 9(5), 56001.

Coder: A. Warmflash, P. François

phievo.Populations_Types.pareto_population.**compdist**(*x*, *y*, *n_functions*)
Compute the distance between the fitness of x and y

class phievo.Populations_Types.pareto_population.**pareto_Population**(*namefolder*, *nfunctions*, *rshare*)
Bases: *phievo.Populations_Types.evolution_gillespie.Population*

Update the Population to manage a Pareto evolution

Note that we dynamically change the fitness of the individuals to give them a list-like fitness.

nfunctions
int – number of functions taken into account by pareto

rshare
float – parameter for the fitness sharing

pop_fitness_share()
Use fitness sharing to increase the diversity of the population.

That is, it augment the rank of inidividual to close from each other to promote diversity in the population. The implementation is a variant on the basic fitness sharing algorithm in section II of Cioppa et al. IEEE Trans. Evol Comp. 11:453

pop_print_pareto(*f_pop*, *f_best*)
Write various information about population in files f_pop and

Parameters

- **f_pop** (*str*) – short description of all individuals
- **f_best** (*str*) – complete description of the first rank only

pop_sort(*verbose=False*)
Perform a pareto sorting of the population using the Goldberg algorithm.

See Van Velhuizen and Lamont. Evol Computation. 8:125 (2000) for details To avoid having population dominated by 0,0 function assigns lowest rank to networks with this score.

update_fitness(*nnetwork*, *integration_result*)
Update (in place) all the fitnesses and the corresponding dlt_fitness

Parameters

- **nnetwork** (*int*) – the index of the network in the population
- **integration_result** (*list*) – the output of compile_and_integrate

class phievo.Populations_Types.pareto_population.**pareto_thread_Population**(*namefolder*, *nfunctions*, *rshare*)

Bases: *phievo.Populations_Types.pareto_population.pareto_Population*, phievo.Populations_Types.thread_population.thread_Population

Update the pareto_Population class to allow threading

Note, when looking for inherited method, python always choose the right most first (here pareto_Population).

pop_mutate_and_integrate(*initial*, *first_mutated*, *last_mutated*, *prmt*, *net_stat*)

Recompute the fitness for half the population and mutate/compute the fitness for the rest. Save all the data in net_stat

Parameters

- **initial** (*int*) – index of the first individual in population
- **first_mutated** (*int*) – index of the first mutated individual in population
- **last_mutated** (*int*) – index of the last mutated individual in population
- **prmt** (*dict*) – the inits parameters for integration
- **net_stat** (*NetworkStat*) – to store the population data

Returns in place modification

Return type None

phievo.Populations_Types.pareto_population.**pcompare**(*x*, *y*, *n_functions*)

Perform a pareto comparison of two networks based on their different fitness

Parameters

- **x,y** (*Network*) – the object to compare
- **n_functions** (*int*) – the number of function taken into account

Returns the comparison of x & y (1 if x>y), 0 indicates that they are pareto equivalent

phievo.Populations_Types.pareto_population.**single_comparison**(*x*, *y*)

Compare two numbers and return 1 if x>y, -1 if x<y and 0 otherwise

10.4 Analysis tools

10.4.1 Simulation

class phievo.AnalysisTools.Simulation.**Genealogy**(*seed*)

Bases: object

compare_ss_wrt_parent(*sim*, *child*, *parent*)

get_network_from_identifier(*net_ind*)

load_sort_networks()

Loads an existing network classification

plot_compare_multiple_networks(*sim*, *indexes*, *cell=0*)

Print a svg figure of the cell profile,time series and the network layout in the seed folder.

plot_front_genealogy(*generations*, *extra_networks_info=[]*, *filename=''*)
Uses the seed plot_pareto_fronts function to display the pareto fronts. In addition, the function allows to plots extra networks in the fitness plan

Parameters

- **generations** – list of generation indexes
- **extra_networks_indexes** – list of extra network informatino dictionaries.

plot_lineage_fitness(*line*, *formula='{}'*, *highlighted_mutations=[]*)

plot_mutation_fitness_deviation(*only_one_mutation=True*, *networks=None*, *ploted_ratio=1*)
Plot the deviation of fitness in the fitness space caused by a generation's mutation.

Arg: only_one_mutation (bool): If True, plot only the networks that undergone only a single mutation durign a generation.

scatter_pareto_accross_generations(*generation*, *front_to_plot*, *xrange*, *yrange*, *step=1*)

search_ancestors(*network*)

sort_networks(*verbose=False*, *write_pickle=True*)
Order the networks, by the label_ind, in a dictionary. The dictonary contains the most useful information but takes last space. The information dictionaries is easier to handle than the actual networks.

Parameters

- **verbose** – print information during sorting
- **write_pickle** – backup the sorting information in a pickle file

Returns dictionary. A key is associated to each network

class phievo.AnalysisTools.Simulation.**Seed**(*path*)
Bases: object

This is a container to load the information about a Simulation seed. It contains mainly the indexes of the generations and some extra utilities to analyse them.

compute_best_fitness(*generation*)

custom_plot(*X*, *Y*)

Plot the Y as a function of X. X and Y can be chosen in the keys of self.observables.

Parameters

- **seed** (*int*) – number of the seed to look at
- **X** (*str*) – x-axis observable
- **Y** (*list*) – list (or string) of y-axis observable

get_backup_net(*generation*, *index*)
Get network from the backup file(or restart). In opposition to the best_net file the restart file is note stored at every generation but it contains a full population. This funciton allows to grab any individual of the population when the generation is stored

Parameters

- **generation** – index of the generation (must be a stored generation)
- **index** – index of the network within its generation

Returns the selected network object

get_backup_pop(*generation*)
Cf get_backup_net. Get the complete population of networks for a generation that was backuped.

Parameters **generation** – index of the generation (must be a stored generation)

Returns List of the networks present in the population at the selected generation

get_best_net(*generation*)
The functions returns the best network of the selected generation

Parameters **seed** (*int*) – number of the seed to look at

Returns the best network for the selected generation

Return type Networks

show_fitness(*smoothen=0*, ***kwargs*)
Plot the fitness as a function of time

stored_generation_indexes()
Return the list of the stored generation indexes

Returns list of the stored generation indexes

class phievo.AnalysisTools.Simulation.**Seed_Pareto**(*path*, *nbFunctions*)
Bases: *phievo.AnalysisTools.Simulation.Seed*

pareto_generate_fit_dict(*generations*, *max_rank=1*)
Load fitness data for the selected generations and format them to be understandable by plot_pareto_fronts

plot_pareto_fronts(*generations*, *max_rank=1*, *with_indexes=False*, *legend=False*, *xlim=[]*, *ylim=[]*, *colors=[]*, *gradient=[]*, *xlabel='F_1'*, *ylabel='F_2'*, *s=50*, *no_popup=False*)
Plot every the network of the selected generations in the (F_1,F_2) fitness space.

Parameters

- **generations** (*list*) – list of the selected generations
- **max_rank** (*int*) – In given population plot only the network of rank <=max_rank
- **with_indexes** (*bool*) – NotImplemented
- **legend** (*bool*) – NotImplemented
- **xlim** (*list*) – [xmax,xmin]
- **ylim** (*list*) – [ymax,ymin]
- **colors** (*list*) – List of html colors, one for each generation
- **gradient** (*list*) – List of colors to include in the gradient
- **xlabel** (*str*) – Label of the xaxis
- **ylabel** (*str*) – Label of the yaxis
- **s** (*float*) – marker size
- **no_popup** (*bool*) – prevents the popup of the plot windows

Returns matplotlib figure

show_fitness(*smoothen=0*, *index=None*)
Plot the fitness as a function of time

Parameters

- **seed** (*int*) – the seed-number of the run

- **index** (*array*) – index of of the fitness to plot. If None, all the fitnesses are ploted

Returns Matplolib figure

class phievo.AnalysisTools.Simulation.**Simulation**(*path*, *mode='default'*)
Bases: object

The simulation class is a container in which the informations about a simulation are unpacked. This is used for easy access to a simulation results.

PlotData(*data*, *xlabel*, *ylabel*, *select_genes=[]*, *no_popup=False*, *legend=True*, *lw=1*, *ax=None*)
Function in charge of the call to matplotlib for both Plot_TimeCourse and Plot_Profile.

Plot_Profile(*trial_index*, *time=0*, *select_genes=[]*, *no_popup=False*, *legend=True*, *lw=1*, *ax=None*)
Searches in the data last stored in the Simulation buffer for the time course corresponding to the trial_index and plot the gene profile along the cells at the selected time point.

Parameters

- **trial_index** – index of the trial you. Refere to run_dynamics to know how
- **trials there are.** (*many*) –
- **time** – Index of the time to select
- **select_genes** – list of gene indexes to plot
- **no_popup** – False by default. Option used to forbid matplotlib popup windows Useful when saving figures to a file.

Returns figure

Plot_TimeCourse(*trial_index*, *cell=0*, *select_genes=[]*, *no_popup=False*, *legend=True*, *lw=1*, *ax=None*)
Searches in the data last stored in the Simulation buffer for the time course corresponding to the trial_index and the cell and plot the gene time series

Parameters

- **trial_index** – index of the trial you. Refere to run_dynamics to know how
- **trials there are.** (*many*) –
- **cell** – Index of the cell to plot
- **select_genes** – list of gene indexes to plot
- **no_popup** – False by default. Option used to forbid matplotlib popup windows Useful when saving figures to a file.

Returns figure

clear_buffer()
Clears the variable self.buffer_data.

custom_plot(*seed*, *X*, *Y*)
Plot the Y as a function of X. X and Y can be chosen in the list ["fitness","generation","n_interactions","n_species"]

Parameters

- **seed** (*int*) – number of the seed to look at
- **X** (*str*) – x-axis observable
- **Y** (*str*) – y-axis observable

get_backup_net(*seed*, *generation*, *index*)
Get network from the backup file(or restart). In opposition to the best_net file the restart file is note stored at every generation but it contains a full population. This funciton allows to grab any individual of the population when the generation is stored

Parameters

- **seed** – index of the seed
- **generation** – index of the generation (must be a stored generation)
- **index** – index of the network within its generation

Returns The selected network object

get_backup_pop(*seed*, *generation*)
Cf get_backup_net. Get the complete population of networks for a generation that was backuped.

Parameters

- **seed** – index of the seed
- **generation** – index of the generation (must be a stored generation)

Returns List of the networks present in the population at the selected generation

get_best_net(*seed*, *generation*)
The functions returns the best network of the selected generation

Parameters

- **seed** (*int*) – number of the seed to look at
- **generation** (*int*) – number of the generation

Returns The best network for the selected generation

get_genealogy(*seed*)

load_Profile_data(*trial_index*, *time*)
Loads the data from the simulation and generate ready to plot data. :param trial_index: index of the trial you. Refere to run_dynamics to know how :param many trials there are.: :param time: Index of the time to select

run_dynamics(*net=None*, *trial=1*, *erase_buffer=False*, *return_treatment_fitness=False*)
Run Dynamics for the selected network. The function either needs the network as an argument or the seed and generation information to select it. If a network is provided, seed and generation are ignored.

Parameters

- **net** (*Networks*) – network to simulate
- **trial** (*int*) – Number of independent simulation to run

Returns

data (dict) dictionnary containing the time steps at the "time" key, the network at "net" and the corresponding time series for index of the trial.

- net : Network
- time : time list
- outputs: list of output indexes
- inputs: list of input indexes
- **0** [data for trial 0]

– **0** [array for cell 0:]

g0 g1 g2 g3 ..

t0 . t1 . t2 . . .

show_fitness(*seed*, *smoothen=0*, ***kwargs*)

Plot the fitness as a function of time

Parameters **seed** (*int*) – the seed-number of the run

Returns matplotlib figure

stored_generation_indexes(*seed*)

Return the list of the stored generation indexes

Parameters **seed** (*int*) – Index of Seed, you want the stored generation for.

Returns list of the stored generation indexes

10.4.2 Palette

phievo.AnalysisTools.palette.**HSL_to_RGB**(*h*, *s*, *l*)

Converts HSL colorspace (Hue/Saturation/Value) to RGB colorspace. Formula from http://www.easyrgb.com/math.php?MATH=M19#text19

Parameters

- **h** (*float*) – Hue (0...1, but can be above or below (This is a rotation around the chromatic circle))
- **s** (*float*) – Saturation (0...1) (0=toward grey, 1=pure color)
- **l** (*float*) – Lightness (0...1) (0=black 0.5=pure color 1=white)

Returns Corresponding RGB values

Return type (r,g,b) (integers 0...255)

Examples

```
>>> print HSL_to_RGB(0.7,0.7,0.6)
(110, 82, 224)
>>> r,g,b = HSL_to_RGB(0.7,0.7,0.6)
>>> print g
82
```

phievo.AnalysisTools.palette.**color_generate**(*n*, *colormap=None*)

Returns a palette of colors suited for charting.

Parameters

- **n** (*int*) – The number of colors to return
- **colormap** (*str*) – matplotlib colormap name http://matplotlib.org/examples/color/colormaps_reference.html

Returns A list of colors in HTML notation (eg.[‘#cce0ff’, ‘#ffcccc’, ‘#ccffe0’, ‘#f5ccff’, ‘#f5ffcc’])

Return type list

Example

```
>>> print color_generate(5)
['#5fcbff','#e5edad','#f0b99b','#c3e5e4','#ffff64']
```

phievo.AnalysisTools.palette.**floatrange**(*start*, *stop*, *steps*)

Computes a range of floating value.

Parameters

- **start** (*float*) – Start value.
- **end** (*float*) – End value
- **steps** (*integer*) – Number of values

Returns A list of floats with fixed step

Return type list

Example

```
>>> print floatrange(0.25, 1.3, 5)
[0.25, 0.51249999999999996, 0.77500000000000002, 1.0375000000000001, 1.3]
```

phievo.AnalysisTools.palette.**generate_gradient**(*values*, *seq*)

Generates a desired list of colors along a gradient from a custom list of colors.

Parameters

- **values** – list of values that need to ba allocated to a color
- **seq** – sequence of colors in the gradient

phievo.AnalysisTools.palette.**make_colormap**(*seq*)

Return a LinearSegmentedColormap seq: a sequence of floats and RGB-tuples. The floats should be increasing and in the interval (0,1).

phievo.AnalysisTools.palette.**update_default_colormap**(*colormap*)

Update the color map used by the palette modules

Arg:

colormap (str): name of the matplotlib colormap http://matplotlib.org/examples/color/colormaps_reference.html

10.4.3 extra_functions

phievo.AnalysisTools.main_functions.**download_example**(*example_name*, *directory=None*)

Download an example seed or project.

phievo.AnalysisTools.main_functions.**download_tools**(*run_evolution='run_evolution.py'*, *AnalyseRun='AnalyseRun.ipynb'*, *ProjectCreator='ProjectCreator.ipynb'*)

phievo.AnalysisTools.main_functions.**download_zip**(*dir_name*, *url*)

Download and extract zip file to dir_name.

phievo.AnalysisTools.main_functions.**load_generation_data**(*generations*, *restart_file*)

Searches in the restart file the the informations that has been backed up up about the individuals at a given generations.

Parameters

- **generations** (*list*) – index of the generations to load_generation_data
- **restart_file** – path of the restart_file

Returns dictionary where each key contains the informations about one generation.

phievo.AnalysisTools.main_functions.**read_network**(*filename*, *verbose=False*)

Retrieve a whole network from a pickle object named filename

Parameters **filename** (*str*) – the directory where the object is saved

Returns The stored network

phievo.AnalysisTools.main_functions.**smoothing**(*array*, *param*)

Smoothen an array by averaging over the neighbourhood

Parameters

- **array** (*list*) – the to be smoothed array
- **param** (*int*) – the distance of the neighbourhood

Returns list of same size as array

CHAPTER 11

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W