Ecole Supérieure de Biotechnologie de Strasbourg

Université de Strasbourg

ICube UMR 7357

pygenets

AN EVOLUTIONARY ALGORITHM TO MODEL AND SIMULATE GENETIC NETWORKS

Instructions for use

Michaël Pierrelée

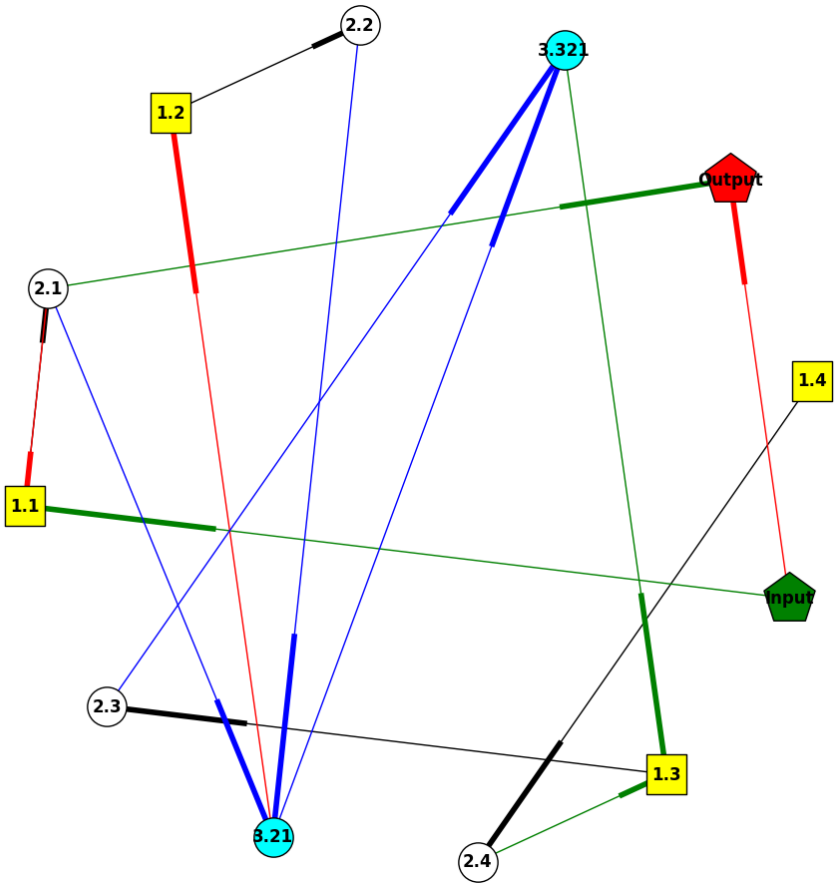
Internship done from 11/07/2016 to 02/09/2016

Under the supervision of Elise Rosati and Morgan Madec

michael.pierrelee@gmail.com

Table of contents

Introduction



pygenets provide a package to model and simulate a population of genetic networks under an evolutionary algorithm in order to figure out an optimal network according to inputs and expected outputs, on Python. A network is modeled with a graph where nodes are inputs, outputs, genes, proteins and complexes, and where edges are interactions (activation or repression) or productions (expression of a protein or formation of a complex) from a node to another.

Figure 1 An example of a genetic network. Yellow nodes are genes, white nodes are proteins and blue nodes represent protein-protein interaction. Green, red, blue and black edges represent activations, repressions, formations of complexes and productions of proteins by a gene, respectively. A negative feedback can be noticed between the gene 1.1 and its protein 2.1. Another interesting point is the function of the complexes 3.21 and 3.321 as pumping complexes, modulating indirectly the activity of the protein 2.1 on the output by reducing its available quantity.

Structure of networks

Some basic rules for network construction can be highlighted:

1. there is at least one input, and all inputs have to be directly or indirectly connected to the only output;
2. a gene or a set of genes cannot be strictly independent from the main network of the output;
3. a gene produce only one protein and a protein has always at least one function, such as an interaction with a gene or with another protein to form a complex;
4. a complex always has two predecessors;

5. a complex always has a function, like a protein;
6. but if the rule 5 is not respected, it means that the complex it is in a pumping complex-like configuration. In other words, one of its predecessors has at least 2 functions (the complex formation and another).
7. A node can have only one interaction with a same node and it cannot have an interaction on itself.
8. It can have some limitations: number of proteins interacting with a same gene and maximal number of genes in a network, these parameters are fixed by the user.

These rules involve special behaviors after addition or deletion of a node or an edge:

1. If a node is added, its protein has to have a function;
2. if a complex is added and it is in a pumping complex-like configuration, then the script randomly choose whether it creates an interaction between the complex and another node;
3. after the addition of a complex, the choice of the targeted node (the destination of the edge from the complex) cannot be another complex to avoid the formation of a grid of complexes in one round;
4. a protein cannot be directly deleted by the script, it is always removed after that its gene is gone;
5. of course, an input or an output cannot be added or deleted by a mutation;
6. if a gene or a complex is removed, every predecessor of the node makes a link with every direct successor of the node and the parameters of these new interactions are copies from the old node-children interactions;
7. if a sub network becomes independent after a deletion, it is removed if it is not possible to make a link with the successors of the deleted node or if this last has no children;
8. if a complex is removed and if this complex is involved in the formation of other complexes, the indirect successors (the successors of successors) which are not complexes are found out and a link is created between them and the predecessors of the deleted complex.
9. The behavior 8 is not followed if a complex is removed because one of its predecessors is removed. In this particular case, the remaining predecessor forms a link with the children, even if they are complexes.
10. If there are creations or modifications of a node during a deleting mutation, the parameters which have to be defined will be copies from the deleted node or edge.

All functions must have these behaviors. It is important to understand that the nodes called “complexes” are not biological complexes but a representation of the interaction of two other nodes. So a “complex” could be a complex or the modification of a protein by another, like a phosphorylation or an ubiquitination.

Parameters

For a network and its differential equations, several parameters were identified:

- a quantity for nodes, in µMol, for inputs, proteins, complexes and the output, knowing the quantities of inputs doesn’t change over time;
- the maximal and minimal productions of a protein by its gene, in $\mu Mol.s^{-1}$, for genes and the output;

- the effecter-promoter binding affinity (activation or repression), in µMol, for edges from proteins, complexes and inputs;
- the effecter-promoter binding Hill's number, for the same edges as previously;
- the stoechiometric coefficients `a` and `b` for the formation of a complex such as `a*A + b*B = AB`, for complexes;
- the reaction constants `kon` and `koff` for a complex such as `A + B -> AB (kon) -> A + B (koff)`, also for complexes;
- and the degradation constant, in s^{-1}, for proteins, complexes and the output, but not for the input.

So mRNAs are not modeled here. By identification of terms of equations and to determine the domain of parameter values, production thresholds are calculated from $P_{max} = k_{tr} * k_{tl}/d$ and $P_{min} = \alpha * k_{tr} * k_{tl}/d$ with α the promoter leakiness, k_{tr} the transcription rate, k_{tl} the translation rate and d the degradation rate (representing mRNA and protein degradation rates) for a given gene and its protein (Madec et al., 2016), whereas the domain of values for kon and koff are proposed by Corzo and Santamaria (2006). These domains are:

Parameters	Variation domains
P_{max}	*[10e-7, 10e-1]*
P_{min}	*[10e-9, 10e-3]*
K	*[10e-4, 10e+3]*
Hill	*[1, 4]*
Coeff. Stoech.	*[1, 5]*
k_{on}	*[10e+4, 10e+8]*
k_{off}	*[10e-6, 10e+5]*
degradation	*[10e-3, 10e-1]*

The differential equations are defined by Madec et al. (2016) for a protein Y with N activators and M repressors acting on its gene and forming C complexes with other proteins P:

$$Act_Y = \sum_{i=1}^{N} \left(\frac{[A_i]}{K_{A,i,Y}} \right)^{n_{A,i,Y}}$$

$$Rep_Y = \sum_{i=1}^{M} \left(\frac{[R_i]}{K_{R,i,Y}} \right)^{n_{R,i,Y}}$$

$$Com_Y = \sum_{i=1}^{C} \left(koff_{i,Y} * [YP_i] \right) - \sum_{i=1}^{C} \left(kon_{i,Y} * [Y] * [P_i] \right)$$

$$\frac{d[Y]}{dt} = Pmin_Y + (Pmax_Y - Pmin_Y) * \frac{Act_Y}{1 + Act_Y} * \frac{1}{1 + Rep_Y} + Com_Y - d_Y * [Y]$$

For a complex:

$$\frac{d[YP_i]}{dt} = -Com_Y - d_{YP_i} * [YP_i]$$

Evolutionary algorithm

Following the work of P. François and V. Hakim (2004), a population of N networks evolves during successive generations. At the generation 0, there are N identical networks with one or more inputs acting on a gene, producing a protein which actives the output. This last is a gene coding for a protein which is not represented in the graph and so the output node must be considered like a set of two nodes (the gene and its protein).

At each generation, a copy of each network is mutated and added in the pool with the other networks, giving a population of 2N elements. They are ranked according to their scores and the best half is kept to give the population for the next round, the other part is removed. At the end, the population should have converged into a local optimum, but not necessary the best optimum.

Six mutations were identified according to François and Hakim (2004):

1. **m1:** change randomly a parameter;
2. **m21:** remove a gene or a complex;
3. **m22:** remove an interaction of a drawn node;
4. **m31:** add a gene and give it a link with a chosen node;
5. **m32:** create a new interaction of a protein/complex on a gene;
6. **m33:** create a new complex between two proteins and/or complexes.

The order is important because the probabilities must be: `P(m1) > P(m21) > … > P(m33)` with the sum of probabilities of mutations equal to be 1. In the optimization step, occurring for the last generations in order to only keep the core nodes and optimize the parameters, the probabilities m31 to m32 are set to 0.

The number of generation could be 1, to follow the Drake's rule (François, 2014, Shadrin & Parkhumchuk, 2014) or 2 (François & Hakim, 2004). Here, number of mutations per network per generation was equal to the number of mutations per node per network per generation (fixed by the user) times the number of nodes. That number is limited between 1 and a value given by the user.

Moreover, a node to mutate is chosen according to its age which is increased at each generation. The nodes are sorted by ages, the different ages are get back and divided into 5 parts. A part is randomly drawn according to an exponential law, then an age and a node of its age is randomly chosen. The equation of probability is, for M parts whit a part Gk and N the number of nodes involved in the part Gk:

$$P(G_k) = \frac{N_k}{\sum_{i=1}^{M} N_i} * \gamma * e^{\gamma \cdot k}$$

Obviously, in a mathematical point of view, it doesn't make sense because $\sum_{i=1}^{M} P(G_i) \neq 1$. But each probability is then corrected by $P'(G_k) = P(G_k) / \sum_{i=1}^{M} P(G_i)$ and so $\sum_{i=1}^{M} P'(G_i) = 1$.

The exponential decay law was also been implemented:

$$P(G_k) = \frac{N_k}{\sum_{i=1}^{M} N_i} * \gamma * e^{-\gamma \cdot k}$$

This equation is then corrected following the same principle than above.

Currently, only two scores are implemented:

- **The output score**, which is the mean squared error: $\sum_i((cal_i - th_i) * w_i)^2$ with *cal*, *th* and *w* the calculated output value, the expected output value and the weight of its value respectively, for one or more fixed inputs *i*.
- **The complexity score** equals to $\sum_{i=1}^{N} 1.6^{A_i+R_i-1} * 1.25^{A_i} * 1.25^{C_i}$ for N nodes, with A, B and C the number of activating edges, repressing edges and protein-protein interactions from the node *i* respectively. Thus, the genes and the output are not taken into account because they don't have these types of edges.

The ranking of networks is done according to the Pareto's algorithm NDS (Fonseca and Fleming, 1995, Warlflash et al., 2012) where the rank of an individual is equal to the number of dominated solutions plus 1. A sharing function was also added from the equation described in Van Veldhuizen and Lamont (2000). Let f the Pareto's rank of the solution i:

$$f_s(i) = f(i) + \frac{1}{N}\sum_{j \neq i}\left(1 - \frac{d_{ij}}{r_s}\right)$$

$$r_s = \frac{\sqrt{\sum_{k=1}^{p}\left(x_{k,max} - x_{k,min}\right)^2}}{\sqrt[p]{2q}}$$

with `dij` the distance between the solutions `i` and `j`, `x` a score, `N` the number of individuals, `p` the number of axis (2) and `q` the number of solutions to figure out. Practically, the implemented sharing function is:

$$f_s(i) = f(i) + \frac{1}{N}\sum_{j \neq i}\left(1 - \frac{d_{ij} * \sqrt{2q}}{\sqrt{(com_{max} - com_{min})^2 + (mse_{max} - mse_{min})^2}}\right) * C$$

The constant C was added to increase the value of the sharing term, but this last is not used by default.

The growth step happens in the first generation to ignore the network complexity during the selection step. To avoid a Pareto on 1D, a Tournament algorithm was implemented following the explanation of Collet in the MOOC *"Optimisation Stochastique Evolutionnaire Problématique"* (p80-82). It's a n-aire stochastic tournament: the set of networks is divided into `P` parts and `S` elements are selected from each part. The probability to select the best element of a group is defined by `t`. If it is not selected, the second best element is selected with the same probability and so on. When the script reaches the end of the list, it starts a new round until S elements are drawn.

After the growth step, a NDS Pareto's algorithm is used as described previously.

Installation

Packages

The algorithm works on Python 2.7.12, with especially the packages: scipy 0.17.1 or +, matplotlib 1.5.2 and network 1.11. It was run with the manager Anaconda (continuum.io) and the interactive shell IPython notebook, on Windows 10 x64, but these two elements are not required. The package multiprocessing is also used to manage the RAM memory. The list of packages can be found in __init__.py.

The installation procedure on the command terminal of Anaconda is:

```
> conda create -n python27 python=2.7
> activate python27
> conda install -c anaconda scipy=0.17.1
> conda install -c conda-forge matplotlib=1.5.2
> conda install -c anaconda networkx=1.11
```

If necessary, pygraphviz 1.3.1 can be also installed but it is not used in the script, whereas pyyaml is used to import/export network structures:

```
> conda install -c rmg graphviz=2.38.0
> conda install -c marufr pygraphviz=1.3.1
> conda install -c conda-forge pyyaml=3.11
```

If a package is not setup, it can be found on https://anaconda.org/.

In fact, the basic environment of Anaconda Python 2.7 version is fully functional, downloadable on https://www.continuum.io/downloads. Because Networkx also works on Python 3, the algorithm could be run on it with few minor modifications.

Package "pygenets"

pygenets is the name of the folder containing the algorithm. It has to be used like a package and the easiest way is to copy the directory pygenets in the same folder than the work file, in the case of an interactive shell (ipython, spyder…).

Test

To verify if the script is functional, this code can be run in a work file, executed with a command from a shell or a terminal:

```
# coding: utf8
import multiprocessing
from pygenets import *
def run_algo(state):
    c = cts()
    genomes = [init(c, ["Input"]) for i in range(5)]
    networks =  []
    for i in range(len(genomes)):
        for j in range(5): g = mutation(genomes[i], c, False)[0]
        mse = scoring_output(g, {"Input": [0]}, [0], [1])
        com = scoring_complexity(g)
        networks += [[g, com, mse, 1]]
    pareto(networks)
    networks = sorted(networks, key = operator.itemgetter(3,2,1))
    for i, net in enumerate(networks):
        scores = net[1:]
        pos = nx.fruchterman_reingold_layout(net[0])
        drawing(net[0], pos, "network_" + str(i) + ".png", 10,
    10, [-0.05,1.05], [-0.05,1.05], scores = scores)
    state["result"] = True
if __name__ == '__main__':
  manager = multiprocessing.Manager()
  state = manager.dict(result = False)
  p = multiprocessing.Process(target = run_algo, args = (state,))
  p.start()
  p.join()
  if state["result"]: print "simulation done"
  else: print "error"
```

5 various networks will be generated and displayed in the folder of the work file. The first should have the best rank and the last the worst, according to Pareto's algorithm.

Comment: with a copy/paste, an error should occur because " is the right ascii character in a code and " is the wrong. Moreover, `multiprocessing` doesn't directly work in an interactive shell. The code has to be put in a file which will be executed with a command such as `%run test.py` in `ipython`.

How to do

cts function

The parameter set has to be defined by the user in the package file starter.py. Below, the keys represent the properties of the network, whereas the parameters of simulations are defined in another file such as simulation.py.

"input_qty"

- It defines the quantity of input acting on each gene linked to it. The default value is fixed to 0, but this will be changed during the script runtime and so it is not necessary to modify it. The unit of measurement is in µMol.

"output"

- It gives a degradation constant to the protein coded by the output and which cannot be "mutated". Technically, this attribute is given to the output node because there is not a protein node for the output. It is in second^{-1}.

"pmax", "pmin", "K", "hill", "stoech", "kon", "koff", "degrad"

- Respectively, the maximal and minimal rate thresholds of genes, given in µMol.s^{-1},
- the promoter binding affinity for repressors and activators in µMol,
- the promoter binding Hill's number for repressors and activators,
- the stoechiometric coefficient of the complexation reaction aA + bB = AB,
- whereas kon and koff are respectively the binding and the dissociation constants in the reactions A + B -> AB (kon) and AB -> A + B (koff),
- and the protein decay rate in s^{-1}, this constant tallies with the mRNA and the protein decay rates.
- "default": it is the default value given to each node or edge, especially for the output where "degrad" is fixed because there are no mutations on it. The values are randomly chosen when a new node is added for example, but the creation of a new network by the function init takes the values of the keys default.
- "domain": it is the variation domain of a parameter, which is modified by a mutation.
- "variation": it defines how a new random value is affected to a parameter. A new type of variation has to be described in the function mut in mutating.py.

"max_nb_of_genes"

- This parameter defines the maximal number of genes (output not included) of a network. It seems important to keep in mind that a network needs a bigger number before the evolution process leads to an optimal network. So this definition doesn't mean that it is the maximal number of genes in the final genome.

"TFBS_limit"

- It is the maximal number of effectors (activators or repressors) which can be linked to a same gene, so it is the maximal number of different transcription factor binding sites in relation with a gene.

"proba"

- It defines the probability that each type of mutation occurs. The sum of values below has to be 1 and the order has to be respected, such as P(m1) > P(m21) > P(m22) > P(m31) > P(m32) > P(m33) (it is not strictly superior). If this last condition is not respect, the function mutation in mutating.py has to be adapted.
- "m1": probability to modify a parameter,
- "m21": probability to remove a protein-gene interaction,
- "m22": probability to remove a gene or a complex,
- "m31": probability to add a new gene,
- "m32": probability to give a new function to a protein, as a transcription factor,
- "m32": to create a new protein-protein interaction and so create a new complex.

Create a first network

Import the python package:

```
from pygenets import *
```

The constants are defined by the function cts in the file starter.py and the function init will create a basic graph with one gene and one protein between the input(s) and the output:

```
cts = cts()
genome = init(cts, ["Input"])
```

Several inputs can be added to the graph, they will be linked to the only gene:

```
genome = init(cts, ["Input1", "Input2"])
```

genome is a Networkx object (precisely a digraph) and so Networkx functions can be used to manipulate it. For example, the network structure can be displayed with:

```
genome.nodes(data = True)
genome.edges(data = True)
```

Respectively, the nodes and the edges with their properties are returned.

A network can be saved in a file such as a YAML file:

```
nx.write_yaml(genome, 'network.yaml')
```

And a graphical representation can be displayed:

```
pos = layout(genome)
pos = nx.fruchterman_reingold_layout(genome, pos = pos)
pos = nx.fruchterman_reingold_layout(genome, k = 100, pos = pos,
iterations = 100)
drawing(genome, pos, 'network.png', 10, 10, [-0.05, 1.05], [-0.05,
1.05])
```

The different pos are used to improve the quality of the representation. layout is useful to sort the levels of nodes and avoid a random positioning by the Fruchterman-Reingold algorithm. A level is the nodes forming one path between the output and a specified node. The second line can be removed in the case of simple networks. Some networks generate bugs and the first line could be removed if that happened.

Test a network

A first property of a network is its response to an input. A dictionary of values which have to be tested for each input has to be defined. For example:

```
inputs = { "Input1": [0.01, 0.01, 1, 1], "Input2": [0.01, 1, 0.01, 1] }
```

There are 2 inputs and 2 values to test: 0.01 and 1. A list of expected outputs has to be also defined for the sets (0.01, 0.01), (0.01, 1), (1, 0.01) and (1, 1):

```
exp_outputs = [0, 0, 0, 50]
```

The function get_output from solver.py returns the output of a network, so it is required to use the function set_input from misc.py in order to modify "input_qty" of the genome. set_input takes a list such as inputs, but with only one value for each key.

```
out = []
for i in range(len(exp_outputs)):
    inp = { k: inputs[k][i] for k in inputs.keys() }
    set_input(genome, inp)
    out += [ get_output(genome) ]
```

In fact, the function test_genome from testing.py does the same job:

```
out = test_genome(genome, inputs)
```

The user has to modify solver.py if he wants to have another output. Currently, the function integrater dynamically computes the final steady state reached by a network, whereas solver resolves differential equations in a static way. In both cases, they use hill_eqt which defines the differential equation of each concentration for proteins, complexes and for the output, the input quantity being stable.

The mean squared score can be straightforward computed with scoring_output from scoring.py. A weight for each expected output is also given:

```
weight = [1, 1, 1, 1]
mse = scoring_output(genome, cts, inputs, exp_outputs, weight)
```

A weight in the domain [-1, 1] decreases the MSE.

The complexity score of a network is returned by scoring_complexity, it is the square of the sum of nodes and edges: com = scoring_complexity(genome)

Change a network

The file adding.py contains functions to add new nodes and interactions between nodes. The names of genes are floats: at the left of the dot it is a 1, and after the dot it is its id. For a complex, it is a 3 before the dot and 2 for a protein. A protein coded by a gene has the same index as it and the id of a complex is the concatenation of ids of two proteins producing it. For example, the gene 1.1 expresses the protein 2.1 which makes a complex 3.12 with the protein 2.2. If 2.1 and 2.2 forms two other complexes, they will be labeled 3.012 and 3.0012 respectively, and so on.

Start with the canvas:

```
from pygenets import *
cts = cts()
genome = init(cts, ["Input"])
```

add_gene is used to add a gene and its protein. Then, an interaction between this protein and another node has to be added (gene 1.2 because 1.1 is the first generated by init):

```
add_gene( genome, 1.2, pmax, pmin, degrad, 1 )
add_PGI( genome, 2.2, "Output", effect, K, hill )
add_PGI( genome, "Input", 1.2, effect, K, hill )
```

The choice of parameters is not important in this example. A gene 1.2 was linked to the input and its protein 2.2 was linked to the output. degrad is the decay rate of 2.2 and effect is "activator" or "repressor". Next, a complex can be formed with 2.1 and 2.2:

```
label_complex = add_complex( genome, 2.1, 2.2, a, b, degrad, kon, koff, 1 )
add_PGI( genome, label_complex, "Output", effect, K, hill )
```

a and b are the stoechiometric coefficients of the reaction a*[2.1] + b*[2.2] = [label_complex].

The functions of removing.py are more automatic. Only a gene or a complex can be targeted and functions manage the different cases which can happen (for example by removing predecessor nodes if they are become useless).

To compare different cases, the previous network is a little bit modified:

```
add_gene( genome, 1.3, pmax, pmin, degrad, 1 )
genome.remove_edge( "Input", 1.2 )
add_PGI( genome, "Input", 1.3, effect, K, hill )
add_PGI( genome, 2.3, 1.2, effect, K, hill )
```

The function remove_edge from the package Networkx is used to reconfigure the interaction between the input and the gene 1.2, added above, in order to add the gene 1.3 between them. For the next step, several cases will be tested to see the behavior of removing functions.

Case 1: remove the complex

```
remove_complex(genome, label_complex, cts["TFBS_limit"])
```

The complex is removed and one of its predecessors, the node 2.2, is now linked to the output with the same edge as the old interaction complex-output. But the interaction 2.1-output is unchanged. Indeed, when a node is deleted from the network, its predecessors are linked to its successors (here, 2.1/2.2 and the output resp.) according to the attributes of the old edges between the removed node and its successors, as far as possible. For example, it is not possible when there is already a direct interaction predecessor -> successor.

Instead of delete the complex, the node 1.1 will be deleted (a protein must not be targeted by a deletion procedure).

Case 2: remove the node 1.1

```
remove_gene(genome, 1.1, cts["TFBS_limit"])
```

An interaction is not created between the input and the complex because an input is not considered as a protein. Thus, the complex is also deleted because a complex without 2 parents is not tolerated. Even if the input would be a protein, the complex would not be kept. So the script searches a successor of the complex and makes an interaction between it (the output) and the predecessor (the input).

Case 3: remove the node 1.2

```
remove_gene(genome, 1.2, cts["TFBS_limit"])
```

It is the same situation as above: there is a new link with the parent (2.3) and the child of the complex (output). To go further, solitary genes will be created in the next case.

Case 4: remove the interaction input <-> 1.1 and remove the node 1.1

```
genome.remove_edge("Input", 1.1)
remove_gene(genome, 1.1, cts["TFBS_limit"])
```

Again, the effect is the same as for the cases 2 and 3 but this time, there are no predecessors and so no new interactions are added.

Case 5: remove the interaction complex <-> output and remove the node 1.2

Here, the complex is a configuration called "pumping complex-like". That means it is conserved because it have an indirect effect on a successor of one of its predecessors. In other words, 3.12 acts on the output through 2.1 by regulating the quantity of 2.1. This configuration can only happen after than the edge complex -> child is deleted. And if its predecessor is also removed in a next round:

```
genome.remove_edge(3.12, "Output")
remove_gene(genome, 1.2)
```

So the genes 1.2 and 1.3 are deleted because 2.2 has no successors anymore just like 2.3 after the deletion of 1.2. The initial network returned by `init` is thus get back.

To conclude, there are many other situations far more complicated, for example with a grid of complexes or when a gene cannot take a new predecessor because the TFBS limit is reached or almost reached for this gene. But the previous cases show the major principles, such as a complex and a protein will be deleted if they don't have 2 predecessors and at least 1 successor respectively. If the other possibilities have to be studied, it is easy to play with a manually created network or with one randomly generated.

Mutate a network and random generations

A network is randomly modified by the functions in `mutating.py`, moreover each node has an age which can be increased at each round by using `update_age()` and the random selection of nodes to mutate follows the functions `choose_node` and `proba_mut`.

Choose a node

The nodes of a network are split up into `P` groups according to their ages, such as the first group has the youngest nodes and the last the oldest nodes, then a probability is given to each group and a node of the drawn group is chosen. This process is done by `choose_node` and the probabilities are returned by `proba_mut`.

Let a `genome` with several nodes, `P` the number of groups of ages (not necessary homogeneous) and `gamma` the constant of the exponential law. `choose_node` requires a list of nodes which can be drawn with their age:

```
nodes = genome.nodes(data = True)
to_choose = [[n, dic["age"]] for n, dic in nodes]
choose_node(to_choose, decay, P, gamma)
```

If decay is equal to `False`, the probabilities follow an exponential law. In other words, the youngest group has the best chance to be drawn. If decay is `True`, so it is an exponential decay law and if it is `None`, the probabilities are equivalent to a uniform law. But regardless of the value of `decay`, a `gamma` equal to 0 involves a uniform law. Last point, the probability is also depending on the total number of nodes in a group of ages. So, for example, under a Uniform law, a group could have a better probability than another to keep a constant probability for each node.

Mutate a network

There are 6 mutating functions corresponding respectively to the probabilities m1 to m33 defined in `cts()`:

1. `mutate_param(genome, cts, False)`: change randomly a parameter (m1).
2. `mutate_remove(genome, tf_lim, False)`: remove a gene or a complex (m21).
3. `mutate_removeInteraction(genome, tf_lim, False)`: remove an interaction of a drawn node (m22).
4. `mutate_addgene(genome, cts, False)`: add a gene and give it an link with a chosen node (m31). The function doesn't verify the TFBS limit.

5. `mutate_adddinteract(genome, tf_lim, False, com = False)`: create a new interaction of a protein/complex on a gene.
6. `mutate_addinteract(genome, tf_lim, False, com = True)`: create a new complex between two proteins and/or complexes.

`False` is the type of law used by `proba_mut` (see above), so here it is the exponential law, and `tf_lim` is the value of the TFBS limit defined in `cts()`. The function `add_complex` is also defined in the file, but it is not presented above because it has no a specified probability. Nevertheless, it can used with `mutate_addcomplex(genome, cts, False)` and one or two of its predecessors can be fixed with `mutate_addcomplex(genome, cts, False, protein1 = prot1, protein2 = prot2)`. In fact, it is used by the functions `_addgene` and `_addinteract`.

For example, to only modify the parameters of genome and optimize them, a solution can be:

```
cts = cts()
genome = init(cts, ["Input"])
save = None
mse = 0
for i in range(100):
    save = genome.copy()
    for j in range(10): mutate_param(save, cts, False)
    score = scoring_output(genome, {"Input": [0]}, [0], [1])
    if mse > score: genome, mse = save, score
```

So, 100 times, a copy of the network is made by `copy` and is mutated 10 times by `mutate_param`. The mean squared output is computed and if it is better, the copy replaces the old genome and so on. The function `scoring_output` will be presented in the next subpart.

As previously mentioned, the key `"variation"` in `cts` is encoded in the function `mut`. For example, if the user wants to add a specific type of variation, he has to add a new condition such as:

```
elif variation == "myVariation":
    choosen_value = …
    return choosen_value
```

Finally, it is the function `mutation` which is the most used in a common practice. It draws a type of mutation according the probabilities in `cts` and executes it, then it returns the mutated genome (it makes a copy of the previous network before the mutation) and a logging message. In the case where the mutated genome is wrong, the function makes a new try until 4 times at most. `mutation` must be used like that:

```
mutated = mutation(genome, cts, decay)[0]
```

`decay` is equal to `True`, `False` or `None`. `[0]` must not be forgotten.

Get the scores of a network

To compare several networks, two opposing scores are determined from their properties: the error between the expected and measured output values for a fixed input, and the network complexity. Then the ranking of networks is computed, that will be described in the next part. Because the Pareto's algorithm is used to rank, it is possible to have more than two scores, but the functions of scoring.py would need some adjustments.

Scoring a network

scoring_output and scoring_complexity are the two functions giving the scores for a network. So this last is represented on a figure and according the logic of pygenets, the X-axis is the complexity and the Y-axis the output score, also called the mean squared error or *MSE*.

scoring_complexity(genome) returns the complexity score. So for a basic genome, such as one generated by init, with a gene linked to the input and the output (through its protein) by activating edges:

```
genome = init(cts(), ["Input"])
complexity = scoring_complexity(genome)
```

The complexity is equal to 2.5 (1.25 + 1.25).

The function scoring_output has already been seen, more generally:

```
genome = init(cts(), ["Input1", "Input2"])
inputs = {"Input1": [0, 0, 1, 1], "Input2": [0, 1, 0, 1]}
exp_out = [0, 0, 100, 100]
weight = [1, 1, 1, 1]
scoring_output(genome, inputs, exp_out, weight)
```

So inputs is a dictionary where each value for a key is a list of as long as exp_out (the expected output value) and weight (the weight for an output value).

That function uses another: set_inputs, changing the values of inputs in the network attributes. For example:

```
set_inputs(genome, {"Input1": 0, "Input2": 0})
```

So the key "input_qty" of each input node is set according to the new parameters. Then, the output value can be calculated by a function of solver.py.

Solver and output values

solver.py integrates differential equations to determine the converged output value. Its functions are largely based on Scipy. A first method was implemented to use the solver of differential systems, scipy.optimize.fsolve:

```
solver(genome)
```

It returns a tuple of names and the set of solutions for a network, corresponding to concentrations of proteins such as `([2.1, "Output"], [1000, 910])` (the concentration 1000 is for the protein 2.1). Nevertheless, it gives bad results and so equation resolution is now using the function `integrater` (`scipy.integrate.odeint`) or `integrater2` (`scipy.integrate.ode` with the integrator `dopri5`). For example:

```
integrater(genome, address="images/concentrations.png")
```

The two integrators have a set of parameters which is slightly different between them because the working is not the same. The address is optional and saves the plot of protein concentrations over the time.

The final output value for a fixed input quantity is currently the only data which has to be extracted from resolution methods: `get_output` will do this job.

```
genome = init(cts(), ["Input"])
set_inputs(genome, {"Input": 0})
get_output(genome)
```

Moreover, a major update of the code could involve a modification of differential equations governing the protein concentrations. These ones are defined by `hill_eqt`, requiring arguments (interaction forces, stoechiometric coefficients…) defined by the function `prepare`. It returns the variables `proteins`, `factors`, `names` and `qty`. They are described directly in the code (in the function `hill_eqt`).

Scoring a list of networks

`scoring.py` also provides functions to compute directly a list of networks. Currently, it takes a data structure such as:

```
net1 = [<networkx object>, complexity score, output score, rank]
networks = [net1, net2, …]
```

Adding new scores will require an updating of functions using the list `networks`.

`new_network` adds a network to the list and then, `update_scores` determines the scores for each new network (recognized by a complexity score equal to 0):

```
g_init = init(cts(), ["Input"])
networks = []
for i in range(10):
    g = g_init.copy()
    for j in range(20): g = mutations(g, cts(), False)[0]
    new_network(networks, g)
networks = update_scores(networks, inputs, exp_out, weight)
```

So a population of 10 networks is created, every one being a copy of a basic network, and mutated, then `update_scores` updates the values, according to the same parameters as for

scoring_output seen above, excepted the rank. It is chosen by special functions presented in the next part.

Select networks

A sample of networks from a population is selected according to its rank, representing a choice between opposed scores.

Ranking

Two functions are available to make the ranking: pareto and tournament, defined in scoring.py. The first follows the Pareto's algorithm and the second the tournament selection. They take both as argument the list networks defined above.

pareto has several features:

- pareto(networks): it is the basic algorithm when a network has a rank increased of 1 at each time that it dominates another. In case of equality on one score, there is still an incrementation depending on the domination of the second score.
- pareto(networks, ignored_com = True): if the argument is True, the complexity score is ignored. So the ranking is done only on 1D, with a perfect elitism.
- pareto(networks, biased = True): here the ranking score is biased in favor of the output score. If a network is dominated on the Y-axis (the MSE), the rank is increased of 2 instead of 1, but it doesn't change for the X-axis (the complexity). This argument is not compatible with ignored_com.
- pareto(networks, avg = avg): if avg is a 2D list (avg[0] and avg[1] are the means of complexity and output scores respectively of the current population), a network get a penalty of 1 if one of its score overreaches the mean of this score and a penalty of 2 if the two scores in the bad halves. This argument, like the next, is compatible with the previous points.
- pareto(networks, sharing = True, C = 1, nb_sol = 10): ranks get a sharing function to select only few networks in a cluster of close scores in order to increase the diversity. The parameter C, by default its value is 1, multiplies the sharing values added to ranks, and nb_sol is number of solutions that the script has to figure out.

This function can also display a Pareto's plot where each dot is a network, with the complexity score on X-axis and the MSE on Y-axis:

```
pareto(networks, address = "img/pareto_plot.png", rk = 5, *args)
```

The blue dots are the networks with a rank less than the argument rk and the red dots have a rank higher.

However, the Tournament algorithm seems more adapted to rank networks with 1D instead of ignored_com for pareto. Here, a stochastic tournament is used with 3 arguments: t, the probability to select a network in a group (t > 0) and P the number of groups dividing the list of networks into lower groups where a number S of elements will be selected.

```
tournament(networks, score, t, P, S)
```

score is the position of the score in the list net1 of networks (according to the same notation as previously) taken into account by the function. For the output score, it is equal to 2.

To make the selection, it will be required to return a dictionary of networks by rank, because several genomes can have the same rank. This is done by:

```
r = rank_list(networks)
```

Selection

For a population of 2N individuals, the N elements with the best ranks are selected and the other half is rejected. Nevertheless, the number of chosen networks is not fixed and the function selection from selecting.py will manage the cases when N is not pair or when the number of elements to select is lower than the sample size:

```
selection(networks, ranks, N, C = 0)
```

ranks is the dictionary generated by rank_list, N the number of elements to select from networks and C the number of elements to reject if 2*N > len(networks). So C must not be higher than the length of networks. Moreover, this function can randomly draw a sample from the population if, for example, the number of elements with a rank = 1 is higher than N.

Evolutionary algorithms may lead to the convergence of a population toward an optimal network or at least a sample fills of a same type of network. So cleanup can be useful to eliminate the redundancies: networks with the same scores and the same structure (but it doesn't verify the parameters of the network).

```
cleanup(networks)
```

It returns a new list of networks.

An example

In this example, 4 networks are chosen from a randomly generated population of 10 elements, according to the Pareto's algorithm.

```
from pygenets import *
g_init = init(cts(), ["Input"])
networks = []
#generate networks
for i in range(10):
    g = g_init.copy() #create a basic network
    for j in range(20): #make 20 mutations on the same network
        g = mutation(g, cts(), False)[0] #follow an exp law
        update_age(genome) #increase of 1 the ages
    new_network(networks, g) #add a new network to the list
#compute the scores and the ranks
networks = update_scores(networks, {"Input": [0]}, [100], [1])
```

```
networks = cleanup(networks) #clean the redundanciess
pareto(networks)
#select networks
r = rank_list(networks)
selected = selection(networks, r, 4)
```

This script is the core of the evolutionary algorithm. It is now quick to add an evolutionary optimization of networks:

```
from pygenets import *
g_init = init(cts(), ["Input"])
networks = []
N = 10
#generate networks
for i in range(N): new_network(networks, g_init.copy())
for n in range(5):
    for i in range(N):
        g = networks[i][0].copy()
        for k in range(20): g = mutation(g, cts(), False)[0]
        update_age(g)
        new_network(networks, g)
    #compute the scores and the ranks
    networks = update_scores(networks, {"Input": [0]}, [100], [1])
    pareto(networks)
    #select
    r = rank_list(networks)
    networks = selection(networks, r, N)
#networks = cleanup(networks) #clean the redundanciess
print networks
```

In this example, the number of generations is fixed to 5. The 3 main steps are:

1. make a copy of every genome, mutate them 20 times, increment node ages and add them to the list
2. compute the scores and then the ranks of 20 genomes
3. select it the 10 genomes with the best ranks

But the main problem of this script is the memory. Indeed, the integrators generate a huge quantity of data consuming too much memory and so, it is not possible to run several simulations without freeze Python. The native package `multiprocessing` brings a solution.

Multiprocessing

Few rules have to be followed to assure a good run. First, the evolutionary script has to be put in a function named `run_algo`. It can takes several parameters such as a classical function but it returns nothing and the standard output (from a print function) will be only displayed in the terminal, contrary to the interactive shell for `IPython` (but there are no modifications for `Spyder` for example). Moreover, also in the case of IPython, the code must not be executed from the interactive shell but only from a file which will be run with `%run myfile.py`. This script takes this look:

```
from myfile import *
param1, param2 = param() #load parameters contained in it
if __name__ == '__main__':
    manager = multiprocessing.Manager()
    state = manager.dict(result = False)
    p = multiprocessing.Process(
          target = run_algo,
          args = (param1, param2,)
    )
    p.start()
    p.join()
    if state["result"]: print "simulation done"
    else: print "error"
```

The initial condition `if __name__` doesn't have to be forgotten according to the Python docs, neither the comma after `param2` in `Process`.

Briefly, `multiprocessing.Manager().dict()` creates a special object permitting to return some results from the function `run_algo`, it is the only way to communicate with it. The key `"result"` will be modified by the core function at the end of the simulation to confirm the success of the operation (so its value is changed to `True`). Other keys can be added if it is needed, following the principle below:

```
state = manager.dict(key1 = defaultValue1, key2 = defaultVal2,  …)
```

And after `p.join()`, the new value of a key can be get back by:

```
print state["key1"], state["key2"], …
```

It has to be noticed that for IPython, the standard output of the print just above will be the interactive shell and not the terminal because it is not in `multiprocessing` anymore.

Next, for this code:

```
p = multiprocessing.Process(
    target = run_algo,
    args = (param1, param2,)
)
p.start()
p.join()
```

`target` is the name of the function to execute by the package and `args` are the arguments of the function targeted. Then, a new object is created and the kernel, where the function `run_algo` will be executed, can be started by using `p.start()` immediately followed by `p.join()`. Several run could potentially be started in the same time, thanks to the parallelization which is offered by the package `multiprocessing` (this is its first purpose), but that was not approached.

Final script

The final script is in the file `simulation.py` which has to be placed in the same folder than `pygenets`. It contains 4 parts:

- The function parameters which defines the parameters of simulations,
- the function `new_set` generating a new set of parameters for one simulation,
- and `run_algo` which is the core for the evolutionary algorithm.
- At the end, the `if` condition which executes the code.

A succession of simulations can be run and the code will continue without any problems. All generated data will be saved.

These different parts will be presented in the next parts.

IF __NAME__ CONDITION

The code from simulation.py is here cut in several parts to facilitate the lecture. It is required to follow in the same time the file.

1. `param, simul = parameters()` this function returns the parameters and the total number of simulations, that will be explained latter.
2. `repeat = 1` here the number of repeats per simulation can be fixed.
3. `g_init = init(cts(), ["Input"])` the variable `g_init` contains the basic network which will be used to start every simulation. It can also be a network loaded from a `yaml` file by using the function `nx.read_yaml`.
4. Loops
    - Part `#names`: a folder named simulations is created to contain the folder of each simulation. A folder of a simulation collects the files generated from the current simulation such as Pareto's figures, curves, networks… Its name is generated from the date, followed by the ids of the simulation and of the repeat. For example: _1_3 means *"Simulation 1, Repeat 3"*.
    - Part `#run`: see the part multiprocessing.
    - Part `#print`: display a message to inform whether a simulation is successful. For IPython users, it will be displayed in the interactive shell.

PARAMETERS FUNCTION

The function returns a dictionary of parameters. Each key corresponds to a parameter of simulations and the value associated to the key has to be a list of elements; for the parameters of a network, they are defined in the function `cts` as previously seen. And for each element of a list is a parameter for one simulation. For example, with the parameter `"gen_nb"`:

```
"gen_nb": [10, 20, 30, 10, 20, 30]
```

So, 6 simulations will be executed. Indeed, the function gets the length of the longest list to determine the number of simulation. The element 0 (10) is the parameter for the simulation 1, the element 1 (20) for the simulation 2 and so on. Now, for 2 parameters:

```
"gen_nb": [10, 20, 30, 10, 20, 30],
"pop": [5, 5, 5, 10, 10, 10]
```

In the same idea, the simulation 1 will take the elements in position 0 (10 and 5), the simulation 2 will take the elements in position 1 (20 and 5), and so on. But it is not necessary to complete every list of every parameter like this. If a list of a parameter is incomplete, the script will take the last element of a list and will copy it as many times as required. Thus, a script with this: `"gen_nb": [10, 20, 10, 20], "pop": [5]`, and other this `"gen_nb": [10, 20, 10, 20], "pop": [5, 5, 10]`, will respectively returns: `"gen_nb": [10, 20, 10, 20], "pop": [5, 5, 5, 5]` and `"gen_nb": [10, 20, 10, 20], "pop": [5, 5, 10, 10]`. The user has to be careful for the elements of list which are themselves lists or dictionaries, such as `"inputs"` or `"proba_optim"`.

So, parameters will return a tuple with the parameters in first position and the length of the longest list in second. This last value will give the total number of simulations to run.

The parameters of simulations are described below:

- `"gen_nb"`: is the number of generations for the evolution of a population,
- `"pop"`: is the population size,
- `"optim"`: is the generation where the optimization step of networks will start (see also `"proba_optim"`),
- `"growth"`: is the duration (in number of generations) of the growth step, where the complexity score is ignored and the tournament algorithm is used (after this step, it will be a standard Pareto),
- `"nb_mut"`: is the mean of the number of mutations per node and per generation (so its domain is]0, +[,
- `"max_m"`: is the maximum number of mutations per generation,
- `"proba_optim"`: is a dictionary containing the probabilities for each mutation to occur during the optimization step (by default, only mutations on parameters and deletions can happen),
- `"decay"`: is the law followed by `proba_mut` to select a node according to its age (`False` for the exponential law, `True` for the exp decay law and `None` for the Uniform law),
- `"t"`: is the probability to keep a good ranked network in the tournament algorithm (t > 0 and the elitism is maximal for t = 1),
- `"score"`: is the position of the ignored score in the tournament algorithm, by default it is the complexity score (1 = complexity score, 2 = output score),
- `"nb_groups"`: is the number of groups for the tournament algorithm (`nb_groups <= pop/2`),
- `"inputs"`: is a dictionary where each key is the name of an input associated to the values of this input which will be tested (more there are values, more the execution time will be high),

- "exp_outputs": is a list of expected values of the output according to each value of the inputs (see the function scoring_output),
- "weight": is the list of weights for each expected output (an output value with a high weight will have a bigger importance).

NEW_SET FUNCTION

For a simulation i and a dictionary of parameters generated by the function parameters, new_set(i, parameters()[0]) returns the set of parameters for that simulation. In the previous example:

```
"gen_nb": [10, 20, 10, 20], "pop": [5, 5, 10, 10]
```

The function will return for the simulation 3/4:

```
"gen_nb": 10, "pop": 10
```

If a new parameter is defined in the first function, so a new line has to be added in this function and another in run_algo.

RUN_ALGO FUNCTION

This function has 5 arguments:

- state: the special dictionary to return to the main environment the results of the simulation;
- param: the set of parameters generated by new_set;
- g_init: the basic network used to start each simulation;
- prefix: an index for the current simulation;
- loc: the address of the folder for the current simulation.

The main parts of run_algo will be presented below.

Parameters and Variables

Each parameter defined in the previous function has to have a line here.

The variables pg and ct are required to display the progress bar and have to be initialized to 0; just like cpt (this last counts the number of executed loops).

avg and std contain the means and the standard deviations per score per generation, whereas best contains the scores of the best network per generation. In this script, avg[0], std[0] and best[0] are for the complexity score and [1] for the MSE.

Loop

`for i in range(pop): new_network(networks, g_init.copy())` fill up `networks` with basic networks. So this list has a length equal to the expected population size (`pop`).

In the main loop of the evolutionary algorithm, the steps presented in the examples can be found.

1. **optimization:** after a fixed number of generations, the probabilities of mutations are changed according to those defined in the parameters;
2. **mutations:** a new list of networks will be created with a copy of each genome in the old list and another mutated copy
    a. the number of mutations per network is defined as the result of the average number of mutations per node per generation times the number of nodes in the network, but the number of mutations is limited by `[1, max_m]`;
    b. `new_network` is repeated two times to add the parent (a copy of the genome which was not be modified) and the child (a mutated copy);
3. **progress bar:** to monitor the progress of a simulation, the function progress takes in argument the number of loops done (`cpt`), the maximal number of loops to do (`gen_nb - 1`) and the variables `pg` and `ct`;
4. **scores, ranking and selection:** the principle was already explained, but here, the Tournament algorithm is executed in the first generations before growth to ignore the complexity score and after that, the Pareto's algorithm take the lead;
5. **Monitoring:** the main properties of the current generation are saved to be displayed after the main loop.

Display

Several data are saved by the script:

- the evolution of means of scores on the generations: `score_evolution(…, gen_nb, avg, std = std)`;
- the evolution of scores of the best network on the generations: `score_evolution(…, gen_nb, best)`;
- the structure of the 5 best networks of the last generation
    - a graphical representation of the structure: `drawing(genome, …)`,
    - a backup of the Networkx object into a `yaml` file: `nx.write_yaml(genome, …)`,
    - and the dose-response curve for this genome: `plt.savefig(…)`;
- and the scores of all various networks generated by the script into a file, a special mention ("`displayed`") is written for the networks which was plotted.

A specific line must not be forgotten:

```
  selected_net = sorted(selected_net, key =
operator.itemgetter(3,2,1))
```

Without it, the plotted networks will not be the best. `operator.itemgetter(3,2,1)` means a sort according to the rank (in position 3 in the list) and in case of an equality, the output score (in position 2) and finally, the complexity score.

Moreover, it is possible to compute a dose-response curve with `test_genome` from `testing.py` taking in argument the genome to test and the values of the input to test. Currently, the script only works with one input, but the function can return the output values for several inputs and so only thee plotting has to be modified.

At the end, the "log" with the details about the various networks is saved in the file and the `state` variable is changed into `True`.

Discussion

No analyses were done on the last version of the script, but some remarks can be exposed from observations on the preliminary results of an old version and results justifying some values of parameters.

1. Probability law used to chose a node (proba_mut)

The probabilities generated by proba_mut were plotted according to the different groups of ages and the gamma constant:

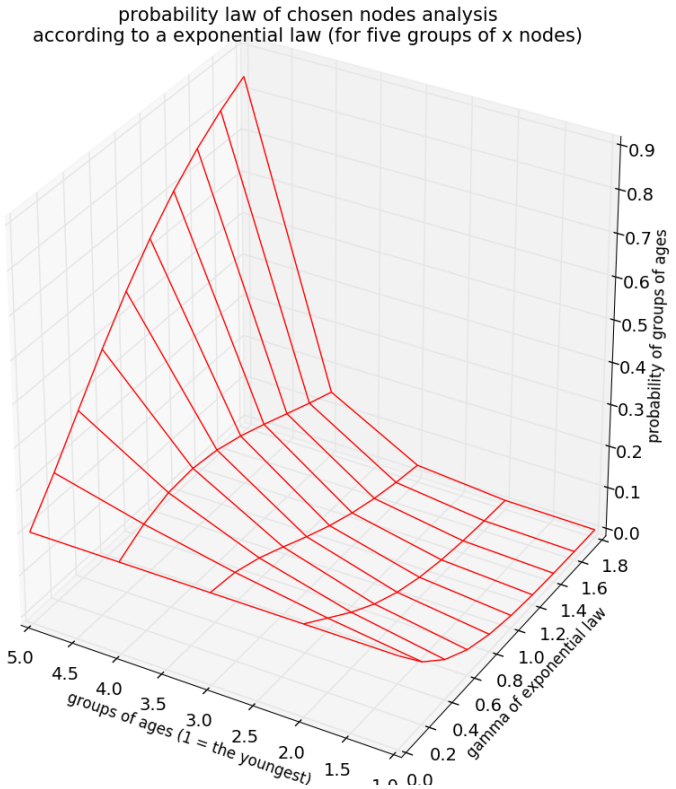


Figure 2 Probabilities for a group of ages according to exponential law in a division of ages into 5 groups

For gamma = 0, the probabilities follow a uniform law. So a gamma of 0.8 seems a good compromise between a too high probability for the oldest and a uniform probability, the probability of the oldest is 0.5 and second 0.25.

The plot is exactly the same for an exponential decay law, but the X-axis has to be inverted.

2. tournament

For a population of 1000 elements with 500 repeats, the curve of the elitism (the % of best elements get back from the initial shuffled population) according to the size of a group in relation to the size of population (because the population is divided into P groups) was plotted:

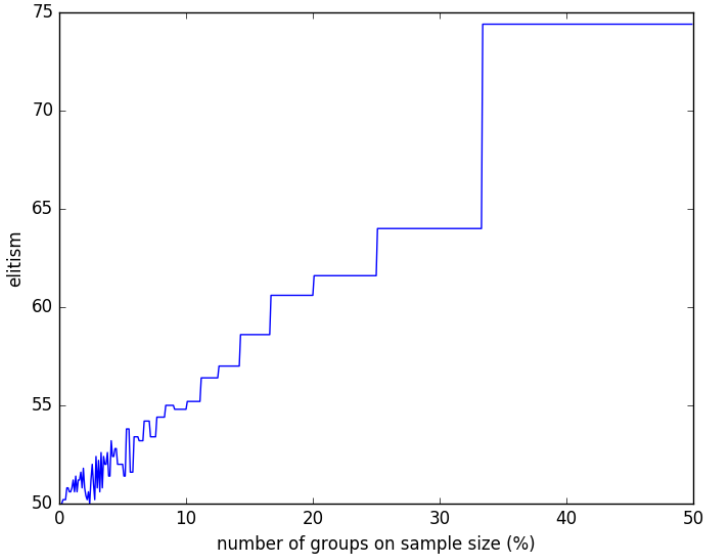


Figure 3 % elitism according to the group size normalized with the population size

Even if the number of repeats is high, the plot can still vary. Whatever, groups representing 20 to 30% could be a good choice to keep a good elitism without exclude 1/3 of diversity, compared to a priori an elitism of 100% for the Pareto's algorithm.

3. integration

The integrator cannot work if the step time is to high, so the user has to be careful with it. Moreover, the number of rounds of integration must be limited because some networks can be instable. For example the red curve in this plot (after the generation 400):

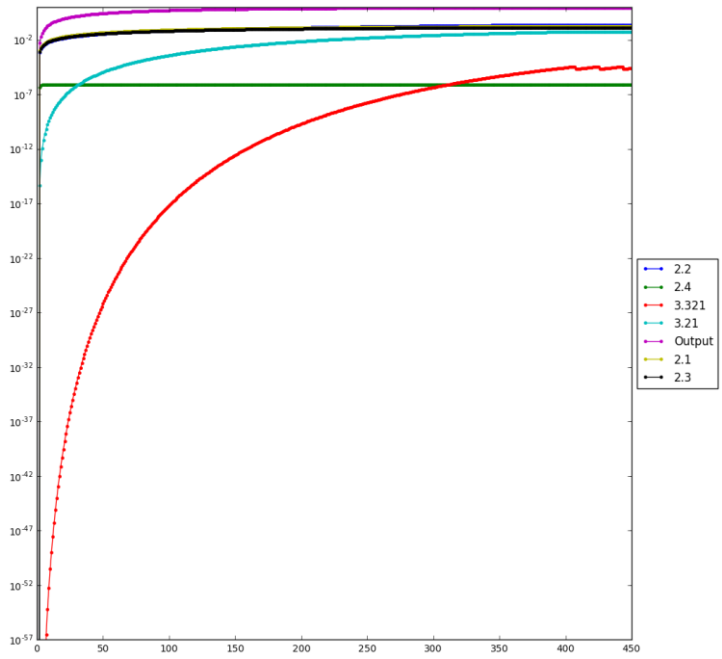


Figure 4 Evolution of concentrations on time for a network which can bug if the integration step

But even if a concentration cannot converge, the script will return the mean on the last points. Moreover, a very precise concentration is not needed for the selection. So E = 100 or 1000 for `integrater` or `integrater2` respectively are largely sufficient.

4. complexity repartition

A population of 200 independent networks was generated and mutated 2000 in order to observe the frequencies of values of complexity:

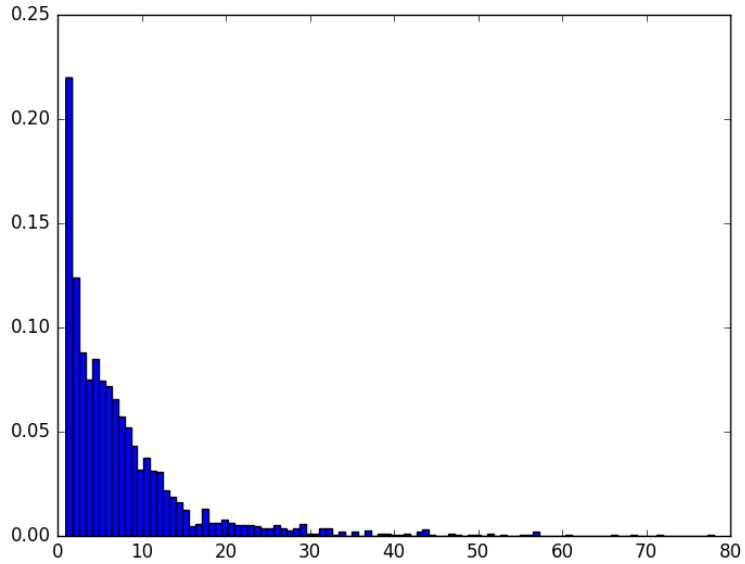


Figure 5 Frequencies of complexity score for 200 networks mutated 2000 times

So complexity is mainly lower than 30 units. With a higher probability to add genes, complexity scores will be more spread out.

5. examples of networks generation for an old version of the script

With an old version, a band filter was generated using especially these parameters:

```
inputs = {"Input1": [1e-5, 0.03, 1e3]}
exp_outputs = [0, 30, 0]
gen_nb = 50
pop  = 40
optim = gen_nb*3/4
growth = gen_nb*3/4
2 mutations per generation (gave good results)
```

Especially, the complexity score of a network was the square of the number of nodes and edges.

For example, one of best result is:

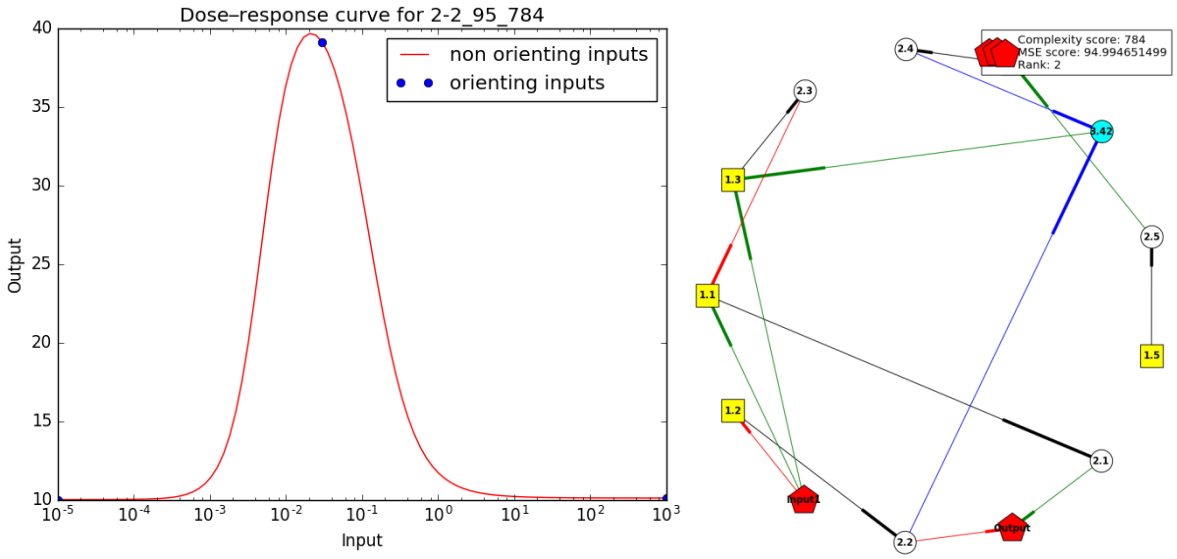


Figure 6 A band filter using the values of Basu (Basu et al., 2005)

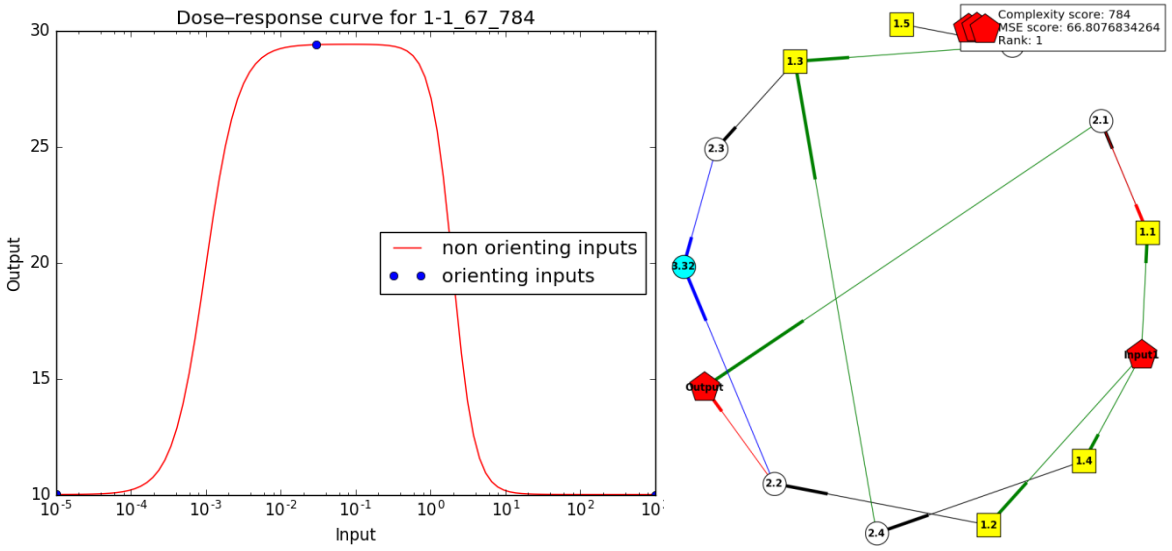


Figure 7 Another band filter

For this old script, some points have to be highlighted :

a. It was very difficult to generate band filters, the convergence was fast but with the good set of parameters, band filters are often generated;
b. there were are great diversity in the networks: in their structure and in their response;
c. if the orienting input points (in blue on the curve above, /!\ which are not the expected outputs) were too close, the script cannot generated an optimal network or the pic is too small;

d. the algorithm liked to generate networks with a “plat” response, fixed to 10 (which was the minimal reached by all networks);
e. for the plot with the means of complexity and outputs scores on generations, there were always a correlation between a good convergence and a sligh decrease of the MSE after the start of the optimization step (see figure 8);
f. for the figure 8, the huge decrease in the first generations is highly correlated with the number of mutations per generation (more there are mutations, earlier is the drop), it could be caused by the very inoptimality of the basic network;
g. the non-diversity of the output score could be a problem in the research of optimality;
h. have too many generation was not a very good solution because the script didn’t generated any optimal network, and increase the population size didn’t help.
i. These observations let to know that there is a precise good set of simulation parameters permitting to figure out an optimal network.
j. With this old script, the basic network was replaced by another a little bit optimized (such as the figure 6), but it was not concluant: the small optimization was lost. It sould be tested again with the last version of the code.

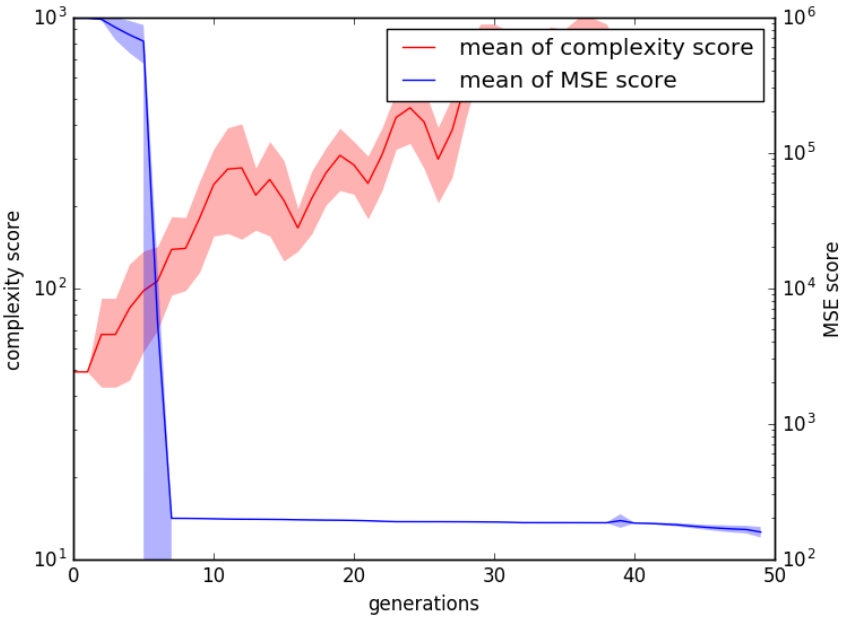


Figure 8 Evolutions of scores for the population of the network in the figure 6

Conclusion

pygenets perform an evolutionary algorithm on genetic networks in order to determine the optimal network for an expected response. But it is highly conditioned by the choice of a good set of simulation parameters and so it needs to highlight a domain of variation for them.

The huge quantities of memory and time required are also a problem. Parallelization and execution of Python on better computers will help as well as develop a way to resolve differential equations without the Python integration, generating some working bugs (they are caught in the function integrater and are not displayed). Nevertheless, the code can be easily left to run without actions from the user.

The functions of scoring and selections should be also studied to know their importance in the script. Tournament could help to resolve the convergence issue but it is not necessary the ultimate solution.

Sources

- Basu, S., Gerchman, Y., Collins C. H., Arnold F. H., Weiss, R. A synthetic multicellular system for programmed pattern formation. *Nature* **434**, 1130-1134 (2005)
- Corzo, J. & Santamaria, M. Time, the forgotten dimension of ligand binding teaching. *Biochem. Mol. Biol. Educ.* **34,** 413–416 (2006).
- Collet, P. – The MOOC "Optimisation Stochastique Evolutionnaire Problématique" (p80-82).
- François, P. Evolving phenotypic networks in silico. *Seminars in Cell and Developmental Biology* **35,** 90–97 (2014).
- François, P. & Hakim, V. Design of genetic networks with specified functions by evolution in silico. *PNAS* **101,** 580–5 (2004).
- Madec, M., Pecheux, F., Gendrault, Y., Rosati, E., Lallement, C., Haiech, J. GeNeDA: an open-source workflow for design automation of gene regulatory networks inspired from microelectronics. *Journal of Computational Biology* (2016).
- Shadrin, A. A. & Parkhomchuk, D. V. Drake's rule as a consequence of approaching channel capacity. *Naturwissenschaften* **101,** 939–954 (2014).
- Van Veldhuizen, D. A. & Lamont, G. B. Multiobjective evolutionary algorithms: analyzing the state-of-the-art. *Evol. Comput.* **8,** 125–147 (2000).
- Warmflash, A., Francois, P. & Siggia, E. D. Pareto evolution of gene networks: an algorithm to optimize multiple fitness objectives. *Phys. Biol.* **9,** 56001–56007 (2012).