UNIVERSITÀ DEGLI STUDI DI PISA

Corso di Laurea Magistrale in Biologia Molecolare e Cellulare

Tesi sperimentale di Laurea

CREATION AND OPTIMISATION OF A YEAST WHOLE-CELL NETWORK

Relatore Interno:
Dr. Roberto Marangoni

Candidato:
Lorenzo Ficorella

Relatore esterno:
Dr. Giorgio Favrin

Anno Accademico 2013/2014

INDEX

Brief Summary

My Master's thesis work has been conducted in the Department of Biochemistry at the University of Cambridge, in the laboratory of Professor Steve Oliver, under the supervision of Dr Giorgio Favrin. The subject of my work has been the creation and optimization of a yeast network, using data from literature and experiments and therefore it has required both a computational and an experimental part.

The purpose of this work has been the creation of a network model that can be employed to predict the results of real experiments regarding the few pathways I have focused on. Its purpose has also been the demonstration of the validity of this approach, i.e. that this approach can be extended to other sections in order to obtain more detailed networks with an ever increasing predictive power.

The steps of this Master's thesis work have been:

- Creating a virtual model of a yeast cell (*S. cerevisiae*), which has been drawn as a Petri Net so that its behaviour over time could be simulated using the appropriate algorithms. The final network has multiple levels; at the topmost level lays a simple whole-cell network while in the lower layers pathways can be described in more detail, and some of them actually are (i.e. real genes and gene interactions are displayed). In facts, the main purpose is not creating a network that contains all yeast genes and their interaction, rather creating a network that can be expanded in the following months and years, but that can be used even in this "unfinished" state. The main pathway that has been chosen for further analysis is the protein folding in the ER. I have chosen this area partly because it is relatively easy to model and to experiment with, partly because I believe that it would be a useful attachment to the whole network.
- Writing scripts that are capable of acquiring data from the model, simulating its behaviour and finally training and testing the network. The simulation script implements the general Petri Net rules, to decide how the network can change at each step of the simulation, and the Gillespie algorithm, to decide which action is taking place at each step and therefore how the network actually changes. The training and testing part is carried out using the simulation script in a broader framework that implements a Monte Carlo approach, aiming to reduce the difference from the simulation and the experimental results. I have written those and other ancillary scripts in the R language, partly because it would eventually be easier to perform other statistical analysis on the results, partly because R is the most common language in biology and therefore it would be easier for other users reading and using my scripts.
- Performing experiments to gather data that are needed for the training and testing stage of the process. As data, I have decided to use the concentration of a reporter in several single mutant strains; therefore, I had to choose a reporter, i.e. a measurable quantity whose values could be simulated in the network and measured during experiments.

I have decided to use GFP as reporter because it is an extremely renowned reporter, namely there are already many data available from the literature regarding both its usage and the expected results: on 15/7/2014, there were 43764 results in PubMed Central and 5324 in PubMed using "GFP+reporter" as key words.

I have employed the normal versions (which folds in the cytoplasm) and the ER-addressed GFP; this latter has been created by fusing GFP and GPCR, which naturally goes through the endoplasmic reticulum in order to be sent to the membrane; these GFP variants have been added to the model as generic cytoplasmic-processed and ER-processed proteins, respectively.

In an interesting paper (Jonikas et al., 2009) GFP is used as a reporter of the Unfolded Protein Response, being transcribed by a transcription factor activated during the UPR; I have decided to use those data for the training of the model, therefore I have added this reporter (a generic UPR-induced protein) in the network.

- Training and testing the network using experimental and literature data, thereby changing network parameters. In facts, the first parameters of the network are made up with the sole purpose of generating a stable model, i.e. a model in which the concentration of the molecular species remain constant (or oscillate periodically) over time; this kind of network, though, is far from behaving as a real cell. The training stage has aimed to change those parameters in order to obtain a network that is stable in all the conditions (e.g. when deleting a node that represents a mutated gene) and in which the ratio of the simulated concentrations of the reporter in wild type and mutated conditions is equal to the actual ratio measured during the experiments. This final trained and tested model is not a faithful representation of a real cell, all the parameters still having no biological meaning, but it should behave as a real one; therefore, it could be used to make prediction about the behaviour of real yeast cells in different conditions such as mutations, oxidative stress, and changes in the culture conditions.

The network has been created using esyN (Bean et al., 2014), a web-based tool to build and share stochastic Petri Nets and generic graphs. It has been developed in our laboratory and it is available at www.esyn.org; I have contributed to its development by generally helping in designing the many features on the website itself and creating some of the example networks. I have also provided all the accessory tools to simulate and analyse Petri nets, i.e. the script that I have employed in my work, which are now available at github.com/esyN/esyN-simulation.

The final network is available both on esyN.org and on eyeast.org, a web repository of yeast Petri Net models, comprising all the yeast networks created using the esyN web-based tool.

Breve Sommario

Il mio lavoro di tesi è stato condotto nel dipartimento di Biochimica dell'Università di Cambridge, nel laboratorio del Professor Steve Oliver, sotto la supervisione del Dr Giorgio Favrin; il soggetto del lavoro è consistito nella creazione ed ottimizzazione di una rete virtuale di lievito, usando dati ottenuti sia dalla letteratura che da esperimenti, e pertanto ha richiesto sia una parte computazionale che una sperimentale

Lo scopo del lavoro è stato la creazione di un modello che possa essere impiegato per predire i risultati di esperimenti riguardanti mutazioni di geni coinvolti nei processi cellulari su cui mi sono focalizzato. Scopo più generale è stato la validazione dell'approccio impiegato, cosicché esso possa essere esteso ad altre sezioni ed ottenere così reti ancora più dettagliate e con un crescente potere predittivo. I passi seguiti nello svolgimento di questo lavoro di tesi sono stati i seguenti:

- Creazione di un modello virtuale di cellula di lievito (*S. cerevisiae*), impiegando il linguaggio delle Petri Nets affinché il suo comportamento nel tempo possa essere simulato utilizzando algoritmi appropriati. La rete definitiva è multilivello: nel livello più alto si trova una semplice schematizzazione della cellula, mentre negli strati inferiori i singoli processi sono descritti in maggiore dettaglio, sia utilizzando nodi generici sia utilizzando vere proteine ed interazioni.
  Quest'ultimo livello di dettaglio è stato applicato solo per alcuni processi; infatti lo scopo non è stato la creazione di una rete contenente tutti i geni e le loro interazioni, quanto piuttosto la creazione di una rete che possa essere ingrandita ed integrata nei prossimi mesi ed anni, e che comunque possa essere impiegata anche in questo stato di parziale incompletezza.
  Il processo che è stato descritto in dettaglio è il ripiegamento ed la maturazione delle proteine nel reticolo endoplasmatico; ho scelto quest'area perché ho ritenuto sia che fosse relativamente facile da modellare ed investigare per via sperimentale, sia che potesse essere un'aggiunta molto utile alla rete di lievito che è in via di costruzione.
- Creazione di scripts che siano capaci di acquisire dati dal modello, simularne il comportamento ed anche effettuare procedure di "*training and testing*" su di esso. Tutti gli script sono stati scritti nel linguaggio di R, sia perché in questo modo sarebbe stato più semplice condurre analisi statistiche dei risultati, sia perché tale linguaggio è molto comune in area biologica e quindi dovrebbe essere più semplice per altri utenti leggere, modificare ed usare i miei scripts.
  Lo script di simulazione unisce le regole generali delle Petri Nets, per decidere come la rete possa cambiare ad ogni passo della simulazione, e l'algoritmo di Gillespie, per decidere quale azione stia effettivamente avvenendo e pertanto come la rete stia effettivamente cambiando.
  Il "*training and testing*" sono svolti impiegando lo script di simulazione all'interno di una cornice più ampia che, implementando un metodo Monte Carlo, mira a ridurre le differenze tra i dati ottenuti dalla simulazione e i risultati sperimentali.

- Conduzione di esperimenti per acquisire i dati necessari per la fase di training della rete. Come dati ho deciso di utilizzare la concentrazione di un osservabile in diversi ceppi mutanti, pertanto ho dovuto scegliere tale osservabile, cioè una quantità misurabile i cui valori possano essere sia simulati nella rete sia misurati negli esperimenti.

  Ho scelto come osservabile la fluorescenza della GFP per via del suo grandissimo utilizzo nella comunità scientifica. Ho impiegato sia la versione normale, che si ripiega nel citoplasma, sia una versione che si ripiega nel reticolo endoplasmatico; quest'ultima è stata creata fondendo la GFP ed la proteina GPCR, che naturalmente passa attraverso il RE per poter essere espresso in membrana. Queste due varianti sono state aggiunte al modello come generiche proteine processate rispettivamente nel citoplasma e nel reticolo endoplasmatico

  In un articolo interessante (Jonikas et al, 2009), la GFP è utilizzata come reporter trascrizionale della Unfolded Protein Response, essendo trascritta da fattori di trascrizioni attivati durante la UPR stessa; avendo deciso di utilizzare anche questi dati per il training del modello, ho aggiunto questo osservabile nella rete come una proteina generica indotta dalla UPR

- *"Training and testing"* della rete usando sia i dati sperimentali che quelli derivati dalla letteratura, in modo da poter cambiare (ottimizzare) i parametri della rete. Infatti, i primi parametri della rete sono stabiliti con l'unico obiettivo di generare un modello stabile, cioè un modello in cui la concentrazione delle specie molecolari resti globalmente costante nel tempo; questo tipo di rete, però, è ben lontano dal comportarsi come una vera cellula. Il training è pertanto necessario per cambiare quei parametri in modo da ottenere una rete che sia stabile in tutte le condizioni e in cui il rapporto delle concentrazioni dell'osservabile, simulate nei ceppi mutanti e wild type, sia uguale al vero rapporto ottenuto dai dati sperimentali.

  La rete definitiva non è comunque una rappresentazione fedele di una cellula vera, dal momento che nessun valore dei parametri ha un reale corrispettivo biologico, ma dovrebbe essere in grado comportarsi come se lo fosse; pertanto, tale rete potrebbe essere impiegata per effettuare predizioni riguardo il comportamento di reali cellule di lievito in svariate condizioni quali mutazioni, stress ossidativo, cambiamenti del mezzo di coltura etc.

La rete è stata creata usando esyN (Bean et al, 2014), un sito per la costruzione e condivisione sia di Petri Nets stocastiche sia di grafici generici. Tale sito, disponibile all'indirizzo www.esyn.org, è stato sviluppato nel nostro laboratorio; ho contribuito al suo sviluppo aiutando nella progettazione delle sue caratteristiche e nella creazione di alcune reti di esempio. Ho inoltre fornito tutti gli strumenti accessori necessari per la simulazione e l'analisi di Petri Net in ambiente R, cioè gli script che ho impiegato nel mio lavoro e che sono ora disponibili all'indirizzo github.com/esyN/esyN-simulation.

La rete finale è (o sarà a breve) consultabile sia in esyN.org stesso sia nel sito eyeast.org, che funge da vetrina per Petri Nets riguardanti il lievito, comprese tutte quelle prodotte in esyN.org

1. Introduction

1.1. General background

1.1.1. Biological networks

Pathways, maps and networks

The graphical representation of biological pathways is becoming increasingly common: maps and networks allow for an easy display of biological data and a quick confront of results from different experiments conditions or species. Large curated networks help researcher in interpreting their data and suggesting new potential experiments, e.g. the results of a transcriptomic experiments, by showing and suggesting hints regarding the real and putative interactions among molecular species.
Moreover, new methods are being invented to analyse those maps, e.g. by identifying the most important nodes and relations of the network, thereby obtaining new findings (e.g. Li et al., 2014; Beurton-Aimar et al., 2014; Vera-Licona et al., 2014). In particular, the topology of any network can be studied looking for distinctive feature such as motifs and modules, i.e. aggregations of elements that have a definite architecture and can be found in several different and even unrelated networks.

Some of these maps work as descriptive network, i.e. they show molecular species (genes, proteins, metabolites etc.) and their dependencies; therefore, they are useful for a visual inspection of a certain pathway but they are not built to be simulated. Each node of these networks can be described in detail, or can have links to other resources that allow the user to retrieve all the information he needs; clear examples of this kind of maps can be found on the KEGG website (http://www.genome.jp/kegg,).
Some other network models are created so that their behaviour over time could be simulated and generate quantitative predictions. There are many different ways to write an executable network model; I have decided to use the Petri Net formalism and modelling language, but it must be noticed that there are few others formalisms and many other modelling languages available (Modelling language, 2014). Moreover, once the modelling language has been chosen, there are still many ways to employ it in the definition and construction of the network; for instance Petri Net models could be written using the Petri Net Markup Language (Billington et al., 2003), whose purpose is providing a common format for allowing for a faster and easier exchange of those models.

Finally, once the formalism and all its features have been set, the experimenter has to choose between stochastic and continuous simulations, between matrices or differential equations etc.; these choices helps in identifying the best approach and algorithm to be employed in the simulation phase.
This large amount of possibilities allows for a great versatility of the networks, and their hundreds of related archives, but it tends to cause compatibility issues and make their merging complicated.

Executable networks (with predictive power)

When creating an executable network, three main factors must be taken into account: the elements, i.e. which products and molecular species are going to be studied over time; the architecture, i.e. how these elements are linked together; the parameters, i.e. the abundance of each element and the "weights" of their links. The choice of the first two elements depends on whether a complete network is needed or not and, if so, whether it is available at the present time (which varies from subject to subject); the choice of the third element is a bit more tricky.

When creating a faithful network, real parameters should be employed, namely kinetic and stochastic parameters measured in several repeated experiments; the advantage of this kind of network is obvious, namely that they can be used to perform real and quantitatively exact simulations. Therefore, it should be possible to evaluate how good a network is (i.e. its potential predictive power) by verifying if it does reproduce the available experimental results and in a second stage if it predicts new properties (e.g. interactions, protein levels, changes in equilibriums, etc.)

The disadvantages are obvious too, though, the first of them being that a huge effort must be carried out to obtain all the parameters (some of whom could be tricky to measure). Second, it must be taken into account that those parameters could change when varying conditions, so that they eventually must be measured under several conditions, or a general rule must be found and applied to change them accordingly. Third, it might be difficult to join and tune together network related to separate areas, whose parameters are calculated and expressed in different ways, e.g. metabolic and genetic networks.

Another possible approach to the construction of executable network is choosing to employ purely made-up parameters, thus solving many of the disadvantages of the previous approach; the problem is that the simulation results of this kind of networks have no direct biological meaning per se.

Those results could still be meaningful if a relationship between simulation values and experimental data were calculated; moreover, those results could be useful if they were used not as absolute values, rather as relative ones when confronting the results of two different conditions of the network (e.g. wild type and mutated, normal and perturbed).

This approach requires an accurate choice of the parameters and their careful improvement, in order to obtain a network that behaves as a real cell even if it is not faithful representation of it, thus producing meaningful simulation results. There are several ways to perform that improvement phase, for instance setting the parameters so that the simulated ratio of the abundance of reporter places in different conditions is equal to the ratio measured in the experiments.

Both approaches should generate executable networks that could be employed to predict the quantity of molecular species in determinate conditions after a set amount of time. This feature would be useful for suggesting hypothesis, driving experiments and in the interpretation of data analysis; in theory, "faithful" networks could also be employed to obtain data without performing actual experiments.

1.1.2. Biological background

Saccharomyces cerevisiae

Yeasts comprise more than 1500 species of eukaryotic microorganisms belonging to the Fungi domain; (Kurtzman and Fell, 2006); they do not form a single phylogenetic grouping because they actually belong to two separate phyla (Ascomycota and Basidiomycota). Yeasts are unicellular, although some species may behave as multicellular through the formation of strings of connected budding cells (pseudohyphae). *Saccharomyces cerevisiae* is likely the most famous yeast because it is being used since centuries in baking, brewing and winemaking. *S. cerevisiae* cells are round, about 5-10 micrometres in diameter; they live both as haploid and as diploid.

Haploid cells can only undergo mitosis, and die when facing stressful conditions; they reproduce via budding, i.e. a bud grows from the mother cell and mitosis only occurs when the bud reaches the dimension of a normal cell. When enough nutrients are provided, budding is a continuous process, meaning that a new bud is produced from a daughter cell even before she divides from the mother cell; this way, yeast cells can double their population every 100 minutes.

Diploid cells undergo mitosis too, and undergo meiosis when facing stressful conditions, thus producing four haploid spores; haploid cells can reform a diploid organism by mating. For the mating to occur, two cells belonging to different mating types are needed; *S. cerevisiae* mating types, i.e. primitive aspects of sex differentiation, are MAT-α and MAT-A.

All *S. cerevisiae* cells can break sugars (e.g. glucose, maltose, and trehalose) thus producing ethanol by fermentation or CO_2 by respiration (only in anaerobic conditions); if enough nutrients are provided, cells show the Crabtree effect (De Deken, 1966), i.e. they tend to perform fermentation no matter whether O_2 is available or not. As nitrogen sources, yeast cells can use ammonia, urea and amino acids (or small peptides), but they cannot use nor nitrate (they cannot reduce it) nor whole proteins (they do not secrete proteases).

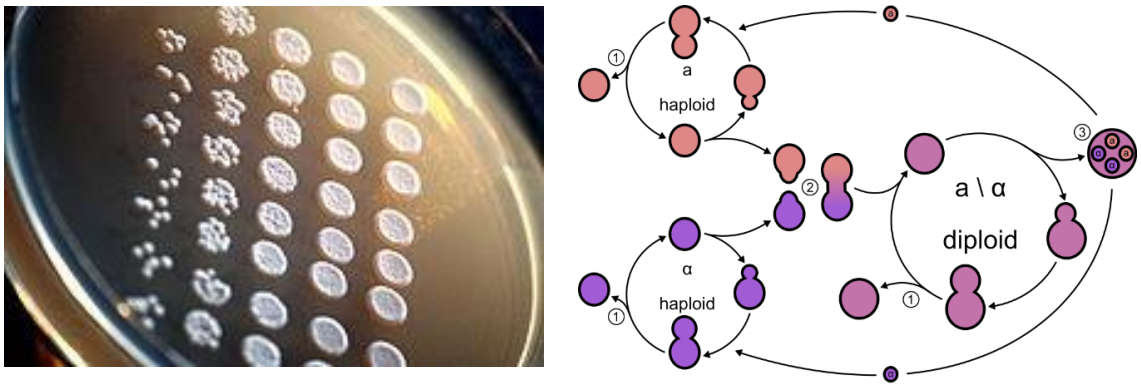


Image 1.1. On the left: *S. cerevisiae* cells on a solid plate (Rainis Venta, http://commons.wikimedia.org/). On the right: cell cycle of a yeast cell.

Yeast as a model organism

Standard values, units of measure and concepts are of capital importance in each field of science; they ensure that experiments can be repeated and results confronted, thus allowing proving or invalidating theories. Repeating experiments in biology is not a trivial issue, though, because it is almost impossible to take into account all the variables that influence the reproducibility of the results; a standardization in the experimental conditions and in the kinds of organisms studied is therefore required.

This way, some "model organisms" have emerged, i.e. organisms that are studied as a paradigm for all the other ones, meaning that the findings made in these models should explain (or provide hints about) the same phenomena occurring in other organisms. There are several models because each one shows a clear example of one or more specific phenomena of study; e.g., embryo development can easily be studied in sea urchin, because its embryo is transparent.

The other main reason for which a certain organism is chosen as a model is that it is amenable to experimental manipulation: it lives, grows and reproduces quickly and cheaply; moreover, its genome can be "easily" manipulated and specific strains or races can be easily produced, selected and stored. Finally, nowadays biologists can analyse and modify the genome of their model organisms, thus producing and using fully characterized strains; this causes a reduction in the unexplained and unaccounted variability, thus allowing for an easier and more accurate comparison of experimental results.

Saccharomyces cerevisiae is a valid model organism because it possess all the qualities aforementioned: it is easy to study because it is the simplest eukaryotic organism, yet it possess many of the features of higher organisms; it is easily cultured, cheap and has a short generation time; its genome can be altered easily (e.g. homologous recombination) and strains selected. Moreover, some of its features are known and it has been used in industry since centuries.

An extensive effort is being carried out for a complete characterization of *S. cerevisiae* cells by sequencing the genome (Goffeau et al., 1996) and performing systematic gene annotation, which requires not only identifying all the genes but also their functions. This step is being carried on in many ways, for instance by producing single mutant and double mutant strains (Giaever and Nislow, 2014). Since many of these genes and gene products have homologs in other organisms, their characterization is being helpful to discover the function and the interactions of many cell cycle proteins, signalling proteins, channel proteins, protein-processing enzymes.

Its main application so far has been in the study of cell cycle: meiosis and mitosis, namely their stages and their related checkpoints (Uhlmann et al., 2011); DNA damages and DNA repair mechanisms; aging and senescence of cells, and the effects of the caloric restriction and replicative aging, and so on. It has also been employed in astrobiology (Warmflash and Ciftcioglu, 2007) to test whether organisms could survive in deep space, which is a requirement of the transpermia hypothesis (namely, the idea that life came to the Earth through the deep space, protected inside rocks that hit our planet).

Protein processing in endoplasmic reticulum

As it is well known, the translation of mRNA into proteins is performed by ribosomes in the cytoplasm. Proteins whose function must be carried out in the Endoplasmic Reticulum (ER) or the Golgi apparatus, or must be secreted outside the cell, are translated by ribosome attached to the external membrane of the (rough) ER; all the other proteins are translated by free ribosomes.

Briefly speaking, the first kind of proteins (from here on "ER-targeted proteins) has a tagging sequence at its N-terminus, thus it is the first part of the proteins to be translated by ribosomes, while they are still free in the cytoplasm. This sequence is bound by several factors that momentarily block the translation, anchor the ribosome to the ER membrane, translocate the nascent protein inside the ER and then allow the translation to continue; therefore, the nascent protein accumulates inside the ER.

This simple scheme varies when dealing with membrane proteins, especially if they have multiple membrane domains; they possess one or more anchoring and targeting sequences, in order to allow the protein to grow inside and outside the ER (i.e. by starting and stopping the translocation several times).

Inside the ER, proteins are in the proper environment for their folding and their post-translational modifications: several specific chaperones and PDIs (protein disulphide isomerases) help the protein folding; some enzymes trim the aminoacidic sequence (e.g. by recognising and eliminating the ER-targeting sequence, which is useless after the translocation stage); some other enzymes add and modify lateral sugar chains (N-linked glycosylation).

This N-glycosylation step consists of three main phases: the synthesis of a precursor oligosaccharide (anchored to the membrane by a dolichol molecule), the transfer en bloc of the whole precursor oligosaccharide to the protein, and the processing of this precursor (which is a species-specific stage).

These chains have multiple roles: they might be needed for the protein folding (e.g. by hiding, or separating two sticky regions) or they might be a fundamental part of the final protein (e.g. membrane glycoproteins). Moreover, they also signal the maturation stage of the protein, namely whether all the steps have been completed and the protein is correctly folded, and thus can be exported to the Golgi; they also signal whether some errors have occurred during the folding stage and thus the protein must be unfolded and refolded, or rather sent to be degraded (ERAD). In this case, misfolded proteins are recognized, unfolded and retrotranslocated into the cytoplasm, where they are bound by other components that target them to the ubiquitin-proteasome system.

In some conditions (e.g. heat stress), unfolded proteins can accumulate inside the endoplasmic reticulum, engulfing its mechanisms of folding and addressing to degradation; in such cases, unfolded proteins start a signalling cascade which generate and sustain a clever system named UPR, i.e. Unfolded Protein Response. It consists in a global re-modulation of the protein production, both at the transcriptional and translational stages; generally speaking, it increases the number and efficiency of chaperones and degrading systems, and it slows down the translation of all the other proteins.

1.1.3. Reporters

Growth phenotype

Growth rate μ defines how quickly a certain cell population increases over time; it is inversely proportional to the doubling time of that cell population. Therefore:

$$N = N_0 * 2^n \qquad N = N_0 * 2^{\frac{t}{t_d}} \qquad N = N_0 * e^{\frac{\ln(2)}{t_d}t} \qquad N = N_0 * e^{\mu t} \qquad \mu = \frac{\ln(2)}{t_d}$$

It is a very complex phenotype because it is affected by many variables (Black, 1996)

- Growth conditions, namely temperature, abundance of nutrients, kind of sugar and nitrogen sources, exposure to selecting or mutating agents etc.
- Condition of the colony, i.e. number and density of cells, stage of the colony (lag phase, exponential growth, steady state) etc.
- Condition of the single cell, namely its age (i.e. how many replications it has already made), the accumulation of ROS and DNA damages etc.
- Possible presence of mutations of cell cycle related genes, or mutations of other unrelated genes

The results of these factors could be an increase or decrease of the growth rate, depending on their interplay: e.g. if a cell maintained its usual replication rate but it went into a senescent (non-replicative) stage earlier than usual, the global growth rate would be affected nevertheless.

On the other hand, it is quite easy to calculate the growth rate by measuring the number of cells over time: the number of cells can be retrieved using sophisticated methods that allow counting only live cells, or simply measuring the optical density of a sample. In facts, the optical density is proportional to the cell density, and by confronting the OD in different time points, it is possible to calculate the growth rate μ:

$$\frac{N}{V} = \frac{N_o}{V} * e^{\mu t} \qquad \frac{N}{V} = a * OD$$

$$a * OD = a * OD_0 \qquad OD = OD_0 * e^{\mu t} \qquad \mu = \frac{\ln(OD_2) - \ln(OD_1)}{t_{(2-1)}}$$

The OD measured is not caused by absorption at a specific wavelength (cells are almost transparent) rather by scattering, which happens using any wavelength, although with different values. Therefore, it does not actually matter which wavelength is chosen for measuring the OD, but it must be the same for all the samples of the experiment; usually, OD_{600} is used.

Moreover, most of the variables affecting growth rate can be monitored and maintained at constant level through all the experiments, thus simplifying the global framework; this way, growth rate could be employed as a global reporter of the effects of mutations, by confronting the absorbance of mutant and wild type strains grown separately but in the same conditions (Blomberg, 2011).

It must be noticed that there is another way to confront the growth rate among different strains, namely the competitive growth: a certain amount of cells of each "labelled" strain grow in the same medium and compete for the same nutrients. After a certain amount of time, the number of cells of each strain is measured and the ratio among those abundances is calculated, thus defining which strains grow more quickly and which ones grow slower than the wild type (Bell, 2010).
Finally, it must also be considered that non-viable mutations, or mutations that totally impair cell growth or division, require a more complex experimental design: for instance, instead of having them in null-mutant strains, they could be in temperature sensitive strains and the temperature of the experiment could be changed accordingly to turn the mutation on/off.

For sake of simplicity, I have performed only non-competing experiments, using viable mutants.

Green Fluorescent Protein

The GFP (Prendergast and Mann, 1978) is a protein isolated from the Atlantic jellyfish *Aequorea victoria*; it exhibits bright green fluorescence (emission peak at 509nm) when exposed to light in the blue-ultraviolet range (a major excitation peak at 395nm and a minor one at 475nm). In *A. victoria*, it is coupled with the aequorin, a photoprotein that emits blue light after binding Ca^{2+}ions, thus exciting GFP; the blue light emission is due to the oxidation of its prosthetic group coelenterazine (i.e. luciferin) into coelenteramide

It is composed of 238 amino acid residues, for a final weight of 26900 Dalton, arranged in 11 β-sheets and 2 α-helices; the β-sheets form a β-barrel, whereas one of the helices is located along the central axis of the barrel and contains the fluorophore (Ormö et al, 1996). The fluorophore is a heterocyclic ring composed by three amino acids (Ser 65, Tyr 66 and Gly 67) that undergo posttranslational modification (cyclisation, dehydration and oxidation); the barrel protects the fluorophore and determines its environment, thus influencing its properties (e.g. excitation and emission wavelengths).

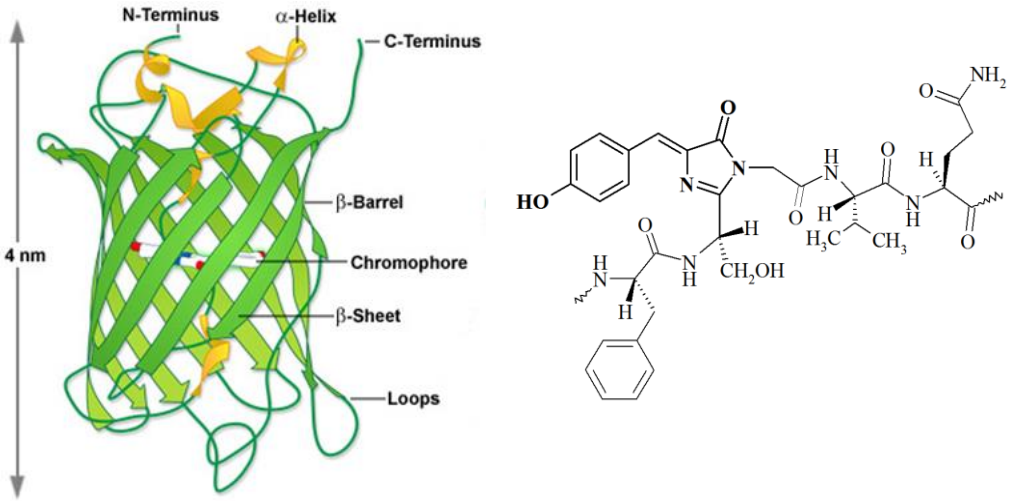


Image 1.2. On the left: drawing based on the 3D structure of the GFP (Day and Davidson, 2009). On the right: the fluorophore.

By mutating single residues of the protein, it has been shown (Shaner et al., 2005; Olenych et al., 2007) that the glycine 67 is essential for the formation of the fluorophore, whereas the tyrosine 66 could be changed in any aromatic amino acids. Serine 65 is not essential, but if it were mutated into Threonine, the resulting fluorophore would be more stable; on the other hand, the overall stability of the protein would be increased by mutating the phenylalanine 64 (which is outside the fluorophore) into a leucine.

Several variants of the GFP have been created by random or site-directed mutations, in order to increase its stability but also to change its properties. These variants are catalogued in seven different classes:

- The first class consists of the native GFP
- The second and third classes are composed of GFPs that have only one excitation peak.
- The fourth class is composed of YFP, i.e. proteins that emit yellow light instead of green; the most relevant mutation is Thr203Tyr, which changes the environment of the fluorophore
- The fifth class is composed of CFP, i.e. proteins that emit in the cyan wavelength; they are created by changing the tyrosine 66 into tryptophan, and then changing other residues in the β-barrel in order to maintain high quantic yields.
- The sixth class is composed of BFP, i.e. proteins emitting blue light thanks to the substitution Tyr66His and other minor mutations.

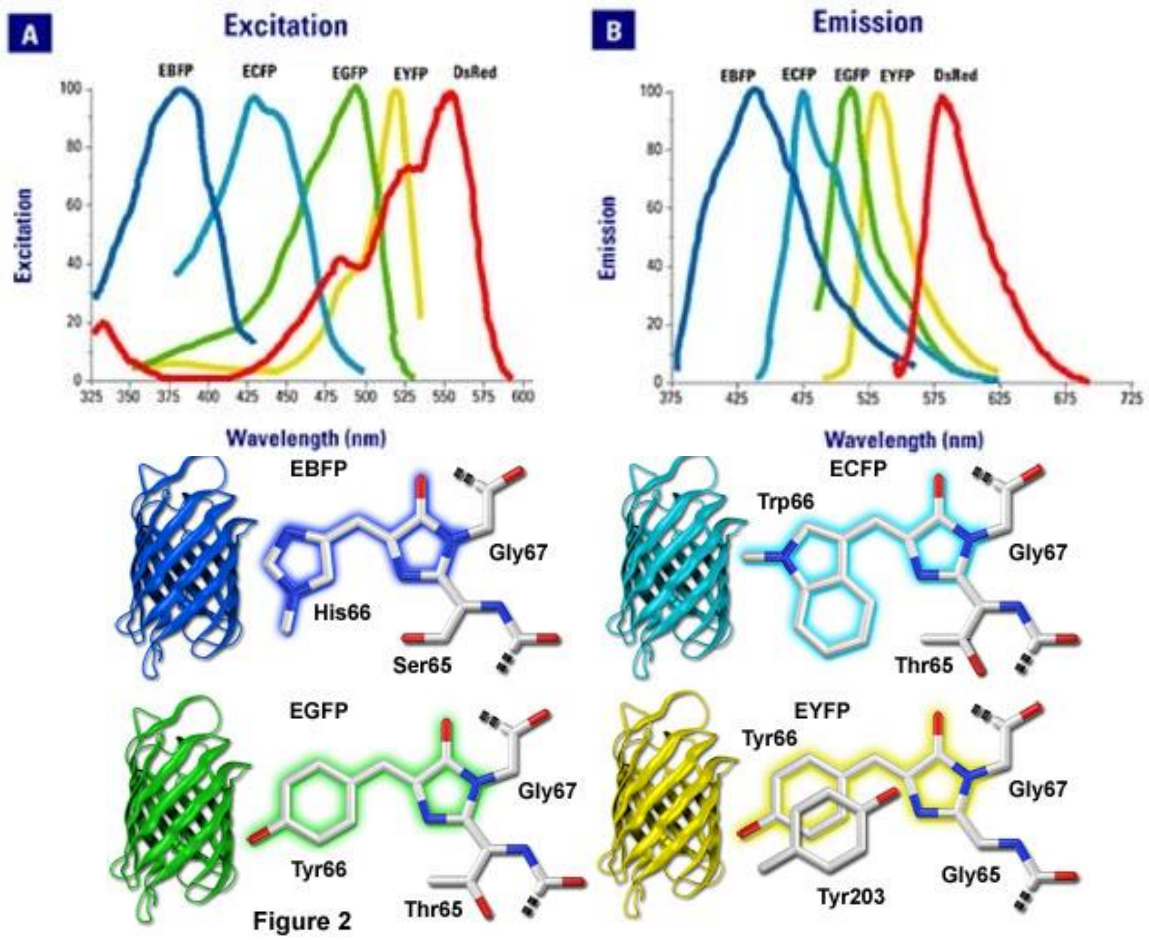


Image 1.3. Upper image: spectra of GFP variants. (ClonTech Labs). Lower image: fluorophores of GFP variants.

GFP fluorescence is an intrinsic phenomenon, i.e. it does not require additional molecules, and therefore it is frequently used as a reporter. There are many possible uses of the GFP and its variants (Tsien, 1998), limited only by the imagination of the researchers; for instance, it can be employed:

- Proving that an exogenous gene can be expressed in a transfected cell, e.g. when testing new transfection methods or protocols, or dealing with new kind of cells;
- Proving that an exogenous gene can be transfected into an oocyte or a zygote, so that all the cells of the organism express that gene (from the first or second generation, respectively);
- Measuring the expression of a gene that is under the control of an inducible promoter; this is accomplished by using that promoter to transcribe GFP gene;
- Studying the strength of a promoter, again using that promoter to produce GFP;
- Measuring the production and degradation rate of a certain protein, by fusing GFP and target protein together (some consideration needed, though).
- Studying the localization of a certain protein, again by creating a fusion protein.
- Performing assays such as the Two-Hybrid one, i.e. creating two fusion proteins bearing half GFP each and then measuring the eventual fluorescence: there is fluorescence only if the other two halves of the fusion proteins interact somehow.

Of course, qualitative experiments (e.g. localization) require fluorescence microscopes, whereas quantitative experiments (e.g. measurement of gene expression levels) require fluorescence spectrophotometers.

Using two or more GFP variants is useful for two main reasons. First, they allow studying multiple gene products at the same time and in the same cell, e.g. their localization or their expression rates, which is particularly useful in interaction studies. Second, they allow for a cross-validation of the data: fluorescent proteins could behave differently from the gene products that is the object of study; using more fluorescent proteins, each one behaving slightly different from the others, could help in reducing this issue, for instance verifying whether a certain fluorescence localization is meaningful or not.

In my work, I have used the wild type GFP to test the transformation protocol of the mutant strains I had selected; then I have employed the normal cytoplasmic variant of the Venus Yellow Fluorescent Protein as a reporter of the translation efficiency. Venus (F46L, F64L, M153T, V163A, S175G) is an YFP whose fluorophore assembles more quickly than usual due to the mutation of phenylalanine 46 into a leucine (Nagai et al., 2002).
Later on, I have used the Sapphire BFP (Q69M, C70V, S72A, Y145F, V163A, S175G, T203I) fused with the secreted protein GPCR, mainly because a plasmid containing the fusion protein gene was already available in the laboratory; therefore, I have repeated the previous experiment with the normal cytosolic variant of the Sapphire Blue Fluorescent Protein (Zapata-Hommer and Griesbeck, 2003).

1.2. Computational background

1.2.1. Petri Nets

General description

The term "Petri Net" refers both to a mathematical modelling language for the description of distributed systems (Blatke, 2001) and to its graphical representation; it derives from its inventor, Carl Adam Petri, who ideated them in August 1939 for describing chemical processes.

A Petri Net is composed of nodes, namely places and transitions, and arcs (i.e. edges) joining places and transitions (but not two nodes of the same kind; it is a bipartite graph); chosen a certain transition, places upstream of it are called "input places", whereas places downstream are called "output places". Places are populated by discrete quantities of *tokens*, whereas arcs are labelled using *weights*.

Petri Net graphs can be simulated, meaning that the tokens can be moved among the places thus changing the marking of the network, i.e. the overall distribution and number of tokens; the simulation is performed using specific firing rules (see the appropriate section in the following pages). Petri Nets are usually stochastic; in this case, there is no predetermined sequence of firing transitions and therefore, if multiple transitions are enabled at the same time, any one of them may actually fire.

According to the kind of simulations performed, one or more transitions can fire at the same time; this or these transitions must be chosen from the set of all the enabled transitions by employing other simulation algorithms. The algorithm is chosen according to the kind of network that is going to be simulated; for instance, even though Petri Nets are born to simulate discrete models, they can also be employed to describe continuous processes, thus requiring a completely different set of algorithms.

Petri Nets have a solid mathematic background: proper mathematical syntax is provided to describe all the rules and the possible states (markings) of a network; graphs can be written as vector and matrices, thus allowing for matrix calculus and other analysis etc.

Moreover, they can be extended by adding new features, some of which I have used in my work: marked tokens (coloured Petri Nets), nested network (hierarchical P.N.), inhibitory and modifier arcs; Petri Nets can also be transformed in semi-deterministic models by using timed and prioritised transitions.

Finally, Petri Nets can be analysed to identify their mathematical properties:

- the reachability of a marking, i.e. whether a certain configuration can be reached in a finite number of steps starting from the present marking;
- the liveness of the network, i.e. how often and how many times transitions can fire (see also the section regarding dead states);
- the boundness of the network, namely how the tokens can distribute among the places (see also the section regarding the accumulation of tokens)

Arcs

Normal edges are oriented edges connecting places and transitions; pre-arcs are edges going from places to transitions, whereas post-arcs are edges going from transitions to places. These edges have weights; the weight of a pre-arc determines how many tokens are removed from the (pre)place when the transition fires, whereas the weight of a post-arc determines how many tokens are produced to the (post)place. They are the simplest kind of edges and they could be employed to represent most of the metabolic reactions.

Read-only edges are non-oriented edges; their weight s represent how many tokens must be in a certain place for the firing of the linked transition, but those tokens are not removed from the place. They could be represented as two normal edges going from a place to the transition and vice versa: the firing of the transition consumes and produces the same amount of tokens in that place. They could be employed to represent the action of a molecular species that intervenes in a reaction without changing its concentration, e.g. an enzyme: it is needed, but it is not consumed nor produced.

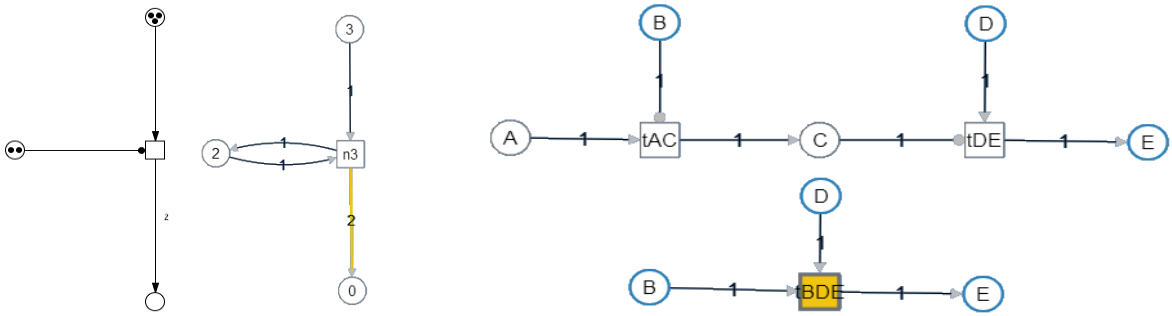


Image 1.4. On the left: read-only edge, redrawn as two normal edges.
On the right: inhibitory arcs and alternative network construction to substitute them.

Inhibitory edges are edges connecting places to transitions. They work exactly in the opposite way of a normal edge: if the (pre)place has a number of tokens equal or greater than the weight of the inhibitory arc, then the transition is disabled. It is impossible to represent those arcs as a combination of simple normal arcs, although it might be possible to obtain very similar results using an appropriate architecture of the network and parameters. Inhibitory edges could be employed to represent the action of inhibitory molecules, such as miRNA.

Modifier edges are edges connecting places to transitions. They work in a dissimilar way from all the other edges; their weights are only needed to determine whether these arcs are playing a role in the transition or not. If the number of tokens of a (pre)place is equal or greater than the weight of the linking edge, then the edge can act by changing some parameters of the transition; otherwise, the transition can still happen using its original set of parameters. Being unique, those arcs cannot be represented in any other way; they could be employed to represent changes in the speed of a reaction due to changes in the enzyme (e.g. a transient phosphorylation) or the reaction environment (e.g. pH changes).

Nodes

Places represent the agents of the network, i.e. the nodes that cause and are affected by the events that happen in the network. They may represent genes, proteins and metabolites, different states of the same molecular species (e.g. repressed and transcribing genes, phosphorylated and unphosphorylated enzyme) or even different localization of the same species (e.g. a transcription factor inside and outside the nucleus); they may also represent generic concepts such as "catabolites" or "proteins".
Each place owns tokens, which represent the quantity of that place in the network. This quantity could have a biological meaning (e.g. molar concentration) when creating uniform networks, namely networks that focus on specific pathways or consist of the same kind of species (and have no "generic" places); in non-uniform networks instead, this quantity must be necessarily invented, unrelated to biology measurements.

Transitions represent the actions that take place in the network, i.e. the nodes that are responsible for the modification of the tokens amount in each place. They may represent all the processes that happen in a cell: specific reactions, both chemical and enzymatic ones; signal transduction cascades, e.g. binding of ligands, translocation of factors; general processes such as "translation", "transcription" or "degradation"; changes in the molecular species due to the environment, e.g. protein denaturation.
Each transition is characterized by a Mass Action parameter; it does not affect the firing of the transition itself, rather it affects the likelihood that its transition actually fires if enabled. When dealing with the "uniform" network abovementioned, these parameters could actually have a biological meaning; for instance, they could be (or be derived from) the kinetic constants of reactions. When dealing with "non-uniform" networks instead, these parameters could be invented in order to obtain networks showing the desired behaviour; for instance, in a network showing two pathways, parameters could be set so that one pathways is usually chosen (e.g. normal metabolic pathways versus salvage pathways).

Coarse nodes are a particular kind of nodes and transitions, namely nodes that contain something else inside them; in the model I have drawn, course places contain other places while coarse transitions contain whole networks. Those nodes do not actually exist, i.e. they are not part of the matrices summarizing the network, so they must be employed very carefully.
Coarse places are useful for representing generic places, i.e. groups of real places (e.g. "chaperones" instead of all the specific chaperones), or representing the protein complexes instead of all their subunits. If a coarse place is linked to a real transition, this link is interpreted as all the children places (i.e. the places contained in it) were connected to that transition.
Coarse transitions are useful for setting the layers of a multilevel network: the lower layers are represented as coarse transition in the upper layer. Therefore, they are useful to organize the network, but they do not have any functions; if a coarse transition is linked to a real place, this link does not actually exist and it is not shown in any matrix.

The firing rule

The standard firing rule is very simple (Murata, 1989): a transition is enabled if all the read-only edges and the pre-arcs are enabled (i.e. their corresponding places have a number of tokens greater than the weights of the arcs), and the inhibitory arcs are disabled. More the one transition could be enabled at the same time, but only one actually fires; the choice of the transition depends on the algorithm implemented (see the following pages), but it surely influenced by the Mass Action parameter.

When a transition fires, tokens are moved among places accordingly to the weights and kinds of the edges and a new state (i.e. overall disposition of tokens) is reached. It must be noticed this is the standard rule, but custom rules could be set for each transition; this is not particularly relevant in biological networks but it has some application in other fields such as engineering.

The outcome

The repetition of these rules for many steps, the architecture of the network itself, and the sequence of firing of the transitions determine the overall outcome of the simulation: tokens could accumulate, diminish or remain constant; the network could "live" indefinitely or it could reach a dead state, i.e. a state in which no transition is enabled.

The decrease of the overall number of tokens could be due to wells, i.e. transitions that consume tokens without producing them, or it be simply due to unbalanced weights of the edges, so that more tokens are consumed than produced. Of course, the increase of the tokens can happen for the opposite reasons, namely because more tokens are produced than consumed, or because of "source" transitions, i.e. transitions that produce tokens without consuming them.

Whether tokens remain constant or increase, they may be evenly distributed among places or they may accumulate in only few ones. This outcome could be intentional, due to the network architecture, or it could be unforeseen: e.g., a certain transition could be randomly chosen in the firsts steps and then it could be preferred in the following ones because of the mechanisms of the algorithm, thus causing tokens to accumulate in its post-places.

Therefore, a dead state could be reached not only in networks whose number of tokens decrease but also in networks whose tokens accumulate in a dead-end place, i.e. a place that is not consumed in any transitions. Dead states, as well as uncontrolled increases of tokens, are usually unwelcome features of the network, meaning that its architecture is somewhere, somehow flawed.

It must be noticed that this is not a general rule: for instance, a network representing the growth of cells in culture medium should show an increase and accumulation of the tokens; a network representing the whole cell should reach a dead state when all the nutrients (inside and outside the cell) are consumed.

1.2.2. Algorithms

Totally stochastic simulation algorithm

As mentioned before, the simulation of Petri Nets is performed using other simulation algorithms, the simplest one being the totally stochastic algorithm (a general description of Petri Nets modelling is available at (Haas, 2002).

The functioning of this algorithm is very simple: each enabled transition has the same chance to be chosen during a simulation step; therefore, transitions are randomly chosen. When using specific Mass Action parameters, they influence the likelihood of each transition; therefore, transition are still randomly chosen, but each one has a different chance.

In the first case, $P_1 = P_2 = P_3 = P_i$

In the second case, $P_1 = \frac{k_1}{\sum k_i}$ $P_2 = \frac{k_2}{\sum k_i}$ $P_3 = \frac{k_3}{\sum k_i}$ $P_i = \frac{k_i}{\sum k_i}$

Since each transition happens instantly, there is no actually way to define and calculate the duration of each step of the simulation; actually, it might be stated that the whole simulation happens instantly (sum of infinite zeroes).

A possible solution could be considering a single step as time unit, so that the global duration of the simulation would be equal to the number of its steps. This method would have no physical meaning, but it could be useful to study and observe changes of the markings over time; it could be adopted to simulate networks in which each action has the same duration.

In this case, $T_1 = T_2 = T_3 = T_i = 1$ $T_{tot} = N_{steps}$

When using Mass Action parameters, another solution could be calculating the duration of each step as the multiplicative inverse of the parameter of the chosen transition; this way the global duration of the simulation would depend both on the number of steps and on which steps have actually happened. This method could be adopted to simulate networks in which transitions have different durations due to their own properties only.

In this case, $T_1 = \frac{1}{k_1}$ $T_2 = \frac{1}{k_2}$ $T_3 = \frac{1}{k_3}$ $T_i = \frac{1}{k_i}$ $T_{tot} = \sum_1^{N_{steps}} t_j$

Even though these methods could be applied to Petri Nets in many fields, it is almost impossible to employ them to simulate biological networks and pathways.

The first method cannot be applied at all, because biological transitions last differently. The second method cannot be usually applied because it does not take into account many additional factors, such as the abundance of the species involved in the transition. Anyway, it could be employed in some limited cases; e.g., it fits the representation of an enzymatic reaction when the substrate saturates the enzyme, i.e. the reaction proceeds at its highest speed possible.

Gillespie algorithm

The Doob-Gillespie algorithm (Doob, 1942; Gillespie, 1976) is a powerful tool in computational systems biology for simulating reactions involving small quantities of reagents, i.e. tens of molecules rather than molar concentrations.

If fact, traditional modelling of reactions is performed considering bulk reactions, involving the interaction of millions of molecules; these conditions allow the creation of continuous and deterministic models, expressed as ordinary differential equations. Instead, Gillespie algorithm grants the ability to perform discrete and stochastic simulation of scarcely populated systems; therefore, it can be applied to cellular processes that involve few molecules and could not be modelled using ODEs.

The physical basis of the algorithm is that, even though cells are very populated systems, and therefore random collisions among "agents" are frequent, proper fruitful collisions (e.g. collisions that happen with the right orientation and energy of the "agents") are not that frequent.

The algorithm works by generating a statistically correct trajectory (possible solution) of a stochastic equation, i.e. "a random walk that exactly represents the distribution of the master equation". It is composed by three steps:

- Start: The number of molecules in the system and the values of reactions constants are set.
- Choice: The next reaction to occur and its duration are determined; the duration and the likelihood of each reaction is proportional to the number of molecules involved, but it also depends on a random number generated at each step.
- Update: The overall time is increased by the amount of the duration of the chosen reaction; the molecular distribution (i.e. the number of each kind of molecules) is changed accordingly to the chosen reaction.

The algorithm is repeated (steps 2-3) until the final time-point is reached or all the reactions are disabled.

The Gillespie algorithm can be employed in the simulation of Petri Nets (Haas, 2002). It is computationally more expensive than the purely stochastic algorithm abovementioned, but it is more versatile: the dependence on a random generated number accounts for the stochasticity; the dependence on the number of molecules involved accounts for its rigour and versatility.

Therefore, it can be employed to simulate almost any network, or even a whole cell; in facts, it can correctly distinguish very frequent reactions from others that happen very rarely (even if they had faster kinetics) and it can help determining/displaying how certain pathways come to be preferred over others.

Of course, when adapting this algorithm to the Petri Nets, some considerations must be made:

- "Sources" transitions have no input places, therefor the abundance of input places is null; these transitions would have zero chance to be chosen by the Gillespie algorithm
- If tokens of different places represented different quantities, they might even have different orders of magnitude; the transitions linked to smaller places would be neglected, even though they were not supposed to be.

Of course, the algorithm workflow must be slightly modified; there are a few variants, and it is important to define which one has being used because it would change the results of the simulations. In the following paragraph, I will introduce the variant I have used in my work (Blatke, 2011; Feres, 2007):

0) The number of tokens in places and MA parameters of transitions are set
1) The rate of each enabled transition is calculated: $R_i = k_i * \sum M_{ji}$ where R_i is the rate of the "i" function, k is its MA parameter and $\sum M_{ji}$ is the overall sum of the tokens of input places.
2) An exponential distribution is calculated for each transition using the correspondent rate, i.e. the mean of the distribution is $\lambda_i = \frac{1}{R_i}$; a random number S is generated from each distribution.
3) The transition that possess the smallest S (i.e. $S_i = \min(S)$) is the actually firing transition. Its S is the "time of the transition"; it can be considered as a waiting time before the firing itself, which happens in no time.
4) Update: The overall time is increased by the amount S; tokens are moved among the accordingly to the chosen reaction.

The algorithm is repeated until a dead state (no transitions enabled) or the final time-point are reached.

Again, the elapsed time has no biological meaning, i.e. it cannot be easily converted in seconds; nevertheless, it is more meaningful than the time calculated in the totally stochastic algorithm because it takes into account that different transitions last differently.

For instance, the total time could be used to confront different condition of the same network: given a certain number of steps, a condition producing a shorter final time means that the single transitions are quicker and the network is more "active". This analysis could be performed also on different networks, if the overall time were normalized for the global tokens of the network.

It must be noticed that the duration of a transition could be an arbitrarily small number; in a positive feedback loop, tokens would increase in the input places thus making the transition faster, its time tending to zero $\lim_{M\to\infty} S_i = 0$. It could be argued that this is biologically impossible, because the highest speed of a reaction usually cannot exceed the diffusion speed; therefore, some changes in the algorithm have been introduced in order to resolve this issue, for instance by setting a minimum duration.

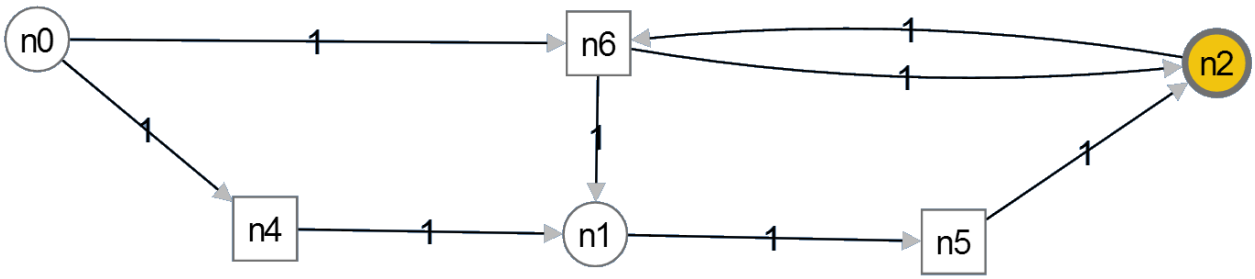


Image 1.5. Petri Net graph of an indirect positive feedback: N4 and N6 represent the same reaction, respectively without and with the positive feedback of the product N2.

Optimization algorithms

Machine learning (Baştanlar and Özuysal, 2014; Sommer and Gerlich, 2013) has been defined as the "field of study that gives computers the ability to learn without being explicitly programmed" (Arthur Samuel, 1959); it is composed by training and testing. There are many algorithms performing machine learning approaches (List of machine learning algorithms, 2014) and many common applications, such as the OCR (Optical Character Recognition) and the spam messages filters.

Training is the learning step, in which the machine acquires experience; it require the *representation* of the data and the creation/evaluation of functions on these data. That means, the machine deals with known data in order to build a model to interpret them.

It should be possible to *generalize* the training of the machine, meaning that these models should perform well when applied on other sets of data. Even if it is impossible to predict how well a machine will perform (that is the key object of study in the computational learning theory), some preliminary checks can be done; this is the *testing step*, which is accomplished using another set of known data that has not been employed in the training session.

As I have explained at the beginning of this introduction, I have not employed a real machine learning approach in my work, but I have applied the same concepts of training and testing to my network: the *generalization* principle of the machine learning approach represents the predictive power I have been looking for in my "virtual cell".

The algorithm used for the training of the network belongs to the vast class of Monte Carlo methods. Those methods employ repeated random sampling to obtain the distribution of a probabilistic entity (Raeside, 1976; Monte Carlo method, 2014).

The kind of the results and the input of those algorithms largely depend on their field of application and purposes, such as optimization and numerical integration.

I have used a Monte Carlo optimization (minimization) algorithm, very similar to a method that is used to optimize the structure of a protein (i.e. minimizing its free energy); it is composed by three steps:

- Random sampling step: creation of a new conformation by randomly changing one or more attributes of the previous conformation. In the protein folding, the starting point is an invented conformation; in networks, the starting conformation is the original set of parameters.
- Calculation of the energy of the new conformation. In networks, the "energy" could be the quantity of a certain place that should be maximized, a ratio that should be minimized etc.
- Choosing or rejecting the new conformation: in the simplest version, a conformation is rejected if its energy is more than the energy of the previous conformation. In more elaborated versions such as the. Metropolis criterion, (Beichl and Sullivan, 2000) some of these otherwise rejected conformation could be randomly accepted; this more flexible algorithm is useful to escape from only apparently good solutions (e.g. local energetic minimum during protein folding).

These steps can be repeated for any number of times; theoretically, the iteration should terminate when an equilibrium is reached, that is when the same conformation and energy is maintained for infinite runs; practically, a number of runs must be chosen, whether for the whole process or the sole equilibration stage.

Regarding the Monte Carlo algorithm applied to networks, some more considerations must be made:

- The function employed for calculating the "energy" values is the aforementioned Gillespie algorithm; this algorithm can be considered as a form of Monte Carlo (kinetic Monte Carlo), therefore the whole optimization process would consist in two nested Monte Carlo algorithms.
- The formula I have employed for choosing the destiny of a conformation is derived from Monte Carlo optimization of protein conformations; therefore, some of the values must be adapted to the new purpose of the formula.
- It makes no sense applying any kind of optimization algorithm to a network whose parameters are known and measured; it is only useful if parameters are invented, such as this case.
- Even if parameters are invented, it does not mean that they do not follow any rules; for instance, some parameters could be unchangeable, others could be intertwined each other (e.g. two parameters that must have the same value) etc. Those rules must be taken into account when randomly choosing and changing parameters values, thus limiting the optimization process.

2. Materials and methods

2.1. Computational part

2.1.1. Snoopy

Snoopy (Rohr et al., 2010) is a software tool to build, animate and simulate many types of Petri Nets. It is a powerful tool because it can handle all the different kinds of Petri Nets, be they stochastic, continuous, coloured or other minor species; it is therefore employed in many different fields of application of Petri Nets (Marwan et al., 2012; Blatke et al., 2013). Depending on the kind of the network, different types of output files are available, to conduct further analysis (e.g. simulation) of the network; for instance the user can export stochastic network, written as matrices, as MatLab files.

The building tool allows the user to employ all the kinds of Petri Net edges (normal, inhibitory, read only, modifiers), although some of them are disabled in certain types of networks, and nodes. User can also choose to use coarse places or coarse transitions; the difference between them is very subtle and resides in which kind of nodes (places or transitions) lay at the edge of the nested network, i.e. at the interface between the lower and the upper layer.

User can change all the basic parameters: i.e. tokens, mass action parameters, weights of the edges,; it's also possible to change the rule according which a specific transition behaves (although this feature is more useful in informatics or engineering rather than biology) or is picked during the simulation stage.

The animation tool is a graphical representation of the tokens game: a certain enabled transition is picked randomly (mass action parameters and tokens number are not considered when choosing the transition), then one or more tokens are moved accordingly among places. The user can follow the path of the tokens and see whether they are disappearing (wells), increasing (sources), or simply moving, whether there are some pathways disconnected from the network or other which are chosen more frequently, whether some transitions are always disabled or always enabled etc.

The simulation tool allows the user to simulate the behaviour of the network (all the layers together), choosing the rules to be applied and the duration of the simulation; the output of this tool is an interactive graph of number of tokens over time, in which each line represent one place, which can be exported as an image. Unfortunately, there is no way to export the values of the places population at the end of the simulation; therefore this tool is only useful as support of the visual inspection provided in the animation tool, to understand which places are accumulating tokens and which places are losing them, but it cannot be employed for more detailed analysis.

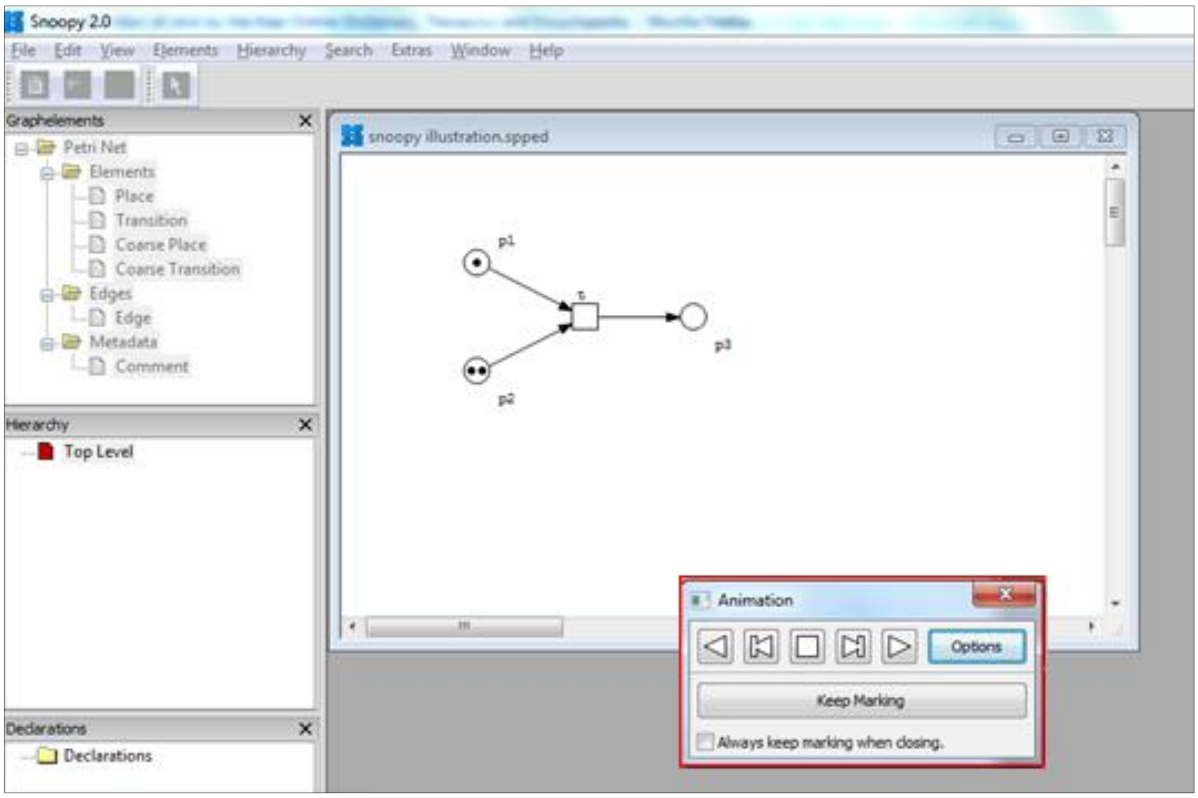


Image 2.1. Screenshot of Snoopy main page: on the upper left, there is editing box; on the lower left, there is the navigation box; on the upper right, there is the network-drawing box; on the lower right, there is the animation toolbox.

On the other hand, the software is quite computationally heavy, which could potentially slow down or freeze the computer when dealing with large networks. Moreover, despite its many functions, Snoopy does not allow the user to customize it according to his wishes; this is likely not a major issue for most of the users because its many functions cover all the principal needs, but it could become a serious issue if the user required functions that are not provided in the program.

For instance, despite the graphical view being customizable, it is difficult to handle with coarse transitions and multiple layers, connect nodes that lay in different layers, finding a certain node in the network, tuning together coarse places and coarse transitions (they cannot be joined by an edge, so some workaround is needed if that connection were required in the network.

In the simulation phase too, even if the user can change some parameters (especially regarding the output), he cannot change nor check the simulation rules; so, it is impossible to know which operations are being carried on during the simulation, or even stop the script in the middle of a simulation. Because of this and other (hidden) problems, it is almost impossible to simulate any large network, which makes this feature useless for my work.

2.1.2. esyN

The esyN web-based tool is based on Cytoscape.js (Lopes et al., 2010), which has been modified and build upon in order to create a tool for building simple "diagram" networks and more complex multi-layer stochastic Petri Nets. It is simpler than Snoopy is, given that it does not have all the features needed for other kinds of networks (e.g. coloured, continuous etc.), nor it allows the user to simulate or animate any networks; moreover, only two kinds of edges (direct and inhibitory ones) are allowed in this web-based tool.

On the other hand, none of the lacking features are necessary when building stochastic Petri Nets, and it is lighter than the pc-based software, which means it can handle easily bigger networks (i.e. a larger amount of edges and nodes) without freezing or crashing; that is why esyN has proven to be very useful for my work.

Moreover, it has some features that are useful for drawing clean networks and navigate among layers in an easier way than in Snoopy:

- Coarse transitions work as usually, meaning that they do not really exists as a node but they have another network nested inside them; you can navigate among the layers by clicking on a coarse transition.
- Coarse places have a different function, meaning that they contain other places rather than containing a whole network; this way, a hierarchy of places is generated, which can be accessed using a dedicated toolbox. Coarse places are existing places in the network, and all the edges reaching them automatically reach their "children" places; for instance, this is useful when representing multimeric enzymes.
- Dispersed places are a novel kind of places that allows the user to add the same place many times in the network (in the same layer or in different ones); they are existing places, where all the copy of a single dispersed place work as a whole. This way, this feature could be useful for reducing the number of overlapping edges in any networks, making them cleaner and clearer; the localization of all the places can be found using a dedicated toolbox.

Finally, esyN provides another useful feature for finding a chosen place of the network in the InterMine databases (http://intermine.github.io/intermine.org/ (Kalderimis et al., 2014) and retrieve all its genetic or physical interaction from the *Homo sapiens*, *Drosophila melanogaster or Saccharomyces cerevisiae* databases. The same toolbox also allows inserting those interacting genes inside the network: in Petri Nets, they are only located near the chosen places; in simple non Petri Net network, they are linked to the chosen places.

2.1.3. R programming language and software

R (R core team, 2014) is a programming language and software environment to organize, show, manipulate and analyse statistical data. The language itself is an implementation of the lexical scoping semantics and S programming language, while the source code for the R environment is written in R, FORTRAN and C. Although R language can be written on any O.S. (e.g. as a text file), it is not platform independent, meaning that an O.S.-specific environment is needed to interpret it; inside the environment, a command line interface is adopted.
As a programming language, R allows the user not only to run simple statistical analysis but also to write and run complex scripts, handling many data at once and performing many manipulations and analysis in the same time. That means the user can assign values to variables, use the main basic programming features (e.g. loops, condition checks etc.) and perform common statistical analysis (recorded as "functions" in the R repository), but he can also define his own functions to perform customized analysis.

R can virtually handle any kind of data, be them characters or numbers (integer and non-integer), but of course most of the functions only apply to numeric variables. There are many "classes" of variables (e.g. vectors, matrices, tables and lists), each one handled by its dedicated functions; in R user can also define its own class and functions, creating S3 or S4 kinds of objects.

The R interpreter can read data inserted using command lines, but it can also acquire data from files stored in the PC, for instance from simple text files or CSV (comma separated values) tables. This also applies to the output, which can be printed out in the screen, both as numeric results and graphs, but also written in a file, be it a table, a matrix, a vector or even text lines.

Undoubtedly, one of the most important features of R environment is that its capabilities can be extended using external packages containing more advanced and specialized formulas, functions, graphical tools, external files handling, statistical techniques etc. Being so popular, there are thousands of packages available, which cover almost all the aspects of statistical analysis; of course, most of them are developed in R language, but some of them are developed in C and FORTRAN, or even Java.
Some of the most popular packages are already included in the installation files of the R interpreter, all the others (more than 5.800 as of July 2014,) can be retrieved from CRAN (Comprehensive R Archive Network, http://cran.r-project.org). More packages are available in other repositories; for instance, Bioconductor (Gentleman et al., 2004) is the main archive of R packages applied to the manipulation and analysis of biological data. The broadness of those repositories is because any user can create a package containing his own functions, objects and classes (and any supporting material) and submit it to those repositories for evaluation (and eventually integration in their archives).

Available CRAN Packages By Name

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

CRAN
Mirrors
What's new?
Task Views
Search

About R
R Homepage
The R Journal

Software
R Sources
R Binaries
Packages
Other

Documentation
Manuals
FAQs
Contributed

A3	A3: Accurate, Adaptable, and Accessible Error Metrics for Predictive Models
abc	Tools for Approximate Bayesian Computation (ABC)
abcdeFBA	ABCDE_FBA: A-Biologist-Can-Do-Everything of Flux Balance Analysis with this package
ABCExtremes	ABC Extremes
ABCoptim	Implementation of Artificial Bee Colony (ABC) Optimization
ABCp2	Approximate Bayesian Computational model for estimating P2
abctools	Tools for ABC analyses
abd	The Analysis of Biological Data
abf2	Load Axon ABF2 files (currently only in gap-free recording mode)
abind	Combine multi-dimensional arrays
abn	Data Modelling with Additive Bayesian Networks
abundant	Abundant regression and high-dimensional principal fitted components
accelerometry	Functions for processing minute-to-minute accelerometer data
AcceptanceSampling	Creation and evaluation of Acceptance Sampling Plans
ACCLMA	ACC & LMA Graph Plotting
accrual	Bayesian Accrual Prediction
accrued	Visualization tools for partially accruing data
ACD	Categorical data analysis with complete or missing responses
Ace	Assay-based Cross-sectional Estimation of incidence rates
acepack	ace() and avas() for selecting regression transformations
acer	The ACER Method for Extreme Value Estimation
aCGH.Spline	Robust spline interpolation for dual color array comparative genomic hybridisation data
acm4r	Align-and-Count Method comparisons of RFLP data
ACNE	Affymetrix SNP probe-summarization using non-negative matrix factorization
acopula	Modelling dependence with multivariate Archimax (or any user-defined continuous) copulas
aCRM	Convenience functions for analytical Customer Relationship Management

Image 2.2. List of the first packages available from CRAN repository (in alphabetical order)

Some of those packages allow the user to integrate R with other programming languages (e.g. Python or Java), meaning that they "translate" the files which are produced by (or eventually go to) scripts written in other languages. For instance, "rjson" package (Couture-Beil, 2014) can be used to read and extrapolate data from JSON files (i.e. files written in java).

Other integrating packages actually integrate those languages in R by translate the whole functions, so that there is no need to use another interpreter at all; finally, it must be pointed out that the user can actually apply the opposite strategy, meaning that he can also use packages to integrate R functions in other languages' interpreters (e.g. PyR).

Some other packages such as Ggplot2 (Wickham, 2009) address the plotting device of the R environment, improving the plotting of data and functions and increasing the customizable options (e.g. position of the legend, dimensions of all the elements). They also add new features as three-dimensional graphs and adjacent plots; this latter feature is very useful for a visual comparison of two sets of data, which is very difficult to do using the standard plotting device because it shows only one graph at time.

One example of packages that enhance the statistical capabilities of the R interpreter is the Zoo package (Zeileis and Grothendieck, 2005); it is mainly used in financial analysis, but it can be applied in any field. This package is made to deal with irregular time series, which are data collected in different times (or time intervals): e.g., the user can extrapolate or interpolate data at specific time points, which is useful for calculating mean values from multiple time series. Zoo works by converting those irregular series (be they vectors or matrices) in a S3 class called "zoo"; it can only deal with totally indexed series, but it has some functions to deal with duplicated times in a time series.

2.1.4. Web sites

GitHub (https://github.com/) and **BitBucket** (https://bitbucket.org/) are two web-based hosting services that work both as cloud storage and as public archive for practically any kinds of codes and scripts; in facts, they allow the user to store and share its code with selected users or the public, but also to create teams and cooperate in writing a code.

This latter feature consists of a web environment where the users can write the code itself (without testing, though), that is a text editor that can recognize many different programming languages; the code can also be created locally and then uploaded to the website in the correct folder. All the different versions of the code are stored and can be browsed and confronted; the authors can pursue their own "branch" of changes, which can be later confronted and merged in order to obtain the definitive version.

KEGG (Kyoto Encyclopaedia of Genes and Genomes) (Kanehisa et al., 2014) is a database of biological data; it articulates itself in several sub-databases regarding data about health (e.g. Disease and Drug databases), the genomic information (e.g. Genome and Genes), data about chemical compounds (e.g. Compound and Reaction) and systems information (e.g. Module and Pathway).

KEGG PATHWAY is an archive of manually drawn maps aiming to represent our knowledge on metabolism and cellular functions. Those maps contain networks of physical interactions among gene products, which are linked to the corresponding coding gene, and can be browsed to look for interactions and gene functions, to compare organisms or different states of the same one (e.g. diseased vs healthy).

Gene Ontology (GO) (Ashburner et al., 2000) is a controlled vocabulary of gene/gene products attributes, that is a list of generic, species non-specific interdependent concepts that should suffice to univocally describe a certain molecular species. This vocabulary has been created by the Gene Ontology Consortium, whose current aims are maintaining the vocabulary, using it to annotate all the gene and gene products and providing tools to use those annotations. The GO file is available at the GO website (http://www.geneontology.org) and can be downloaded as a whole or browsed using AmiGO.
Each gene or gene product can be described according to its localization, function and the biological process in which it participates; therefore the GO vocabulary cover all those three domains. It is structured as a directed acyclic graph, in which each term has relationships with others: e.g. "is a", "regulates" and "occurs in", but also "broader synonym of", "narrower synonym of" etc.

YeastMine (Balakrishnan et al., 2012) is a tool to search and retrieve data available in the Saccharomyces Genome Database (Cherry et al., 2009); it is powered by InterMine, which means it provides a very powerful, flexible and customizable way to search and organize multiple types of data. It is available from http://yeastmine.yeastgenome.org and can be used to retrieve protein sequences and features, gene localization and sequences, phenotypes and interaction data (both physical and genetic ones); a single gene as well as a list of genes can be searched in the same time.

2.2. Experimental part

2.2.1. Yeast culturing

Yeast media

Yeast can be cultured both in liquid and solid (i.e. with agar) media, according to the purposes of the culturing: yeasts in liquid media grow generally faster, which is useful if we needed a large number of; yeasts in solid media grow more slowly and organize themselves in distinct colonies, which is useful if a single colony is needed. Several kinds of media exist, and each kind can be made using different "ingredients" and varying their proportion according to specific needs.

Generally speaking, media can be classified as synthetic (i.e. defined) or complex (i.e. undefined), according to whether composition of the medium is known or not; synthetic media are produced by adding known amounts of known chemical species, whereas complex media are produced using substances whose chemical species might be known but their abundance is unknown.

Complex media are usually complete, meaning that they let any yeast to grow; synthetic media, instead, usually lack one or more elements (e.g. a certain amino acid), therefore selecting against yeast that are auxotrophic for those elements; synthetic media having only sugars and inorganic nitrogen are called "minimal media" and allow the growth of solely non-auxotrophic yeasts.

Finally, a "selective medium" can be created adding a toxic substance (e.g. antibiotics); in this medium, no wild type yeast can survive, therefore selecting for strains bearing a resistance gene. This is particularly useful for selecting yeasts that have acquired a plasmid bearing the resistance gene.

YEPD (yeast extract peptone dextrose) is a complete medium made of yeast extract, peptone, glucose (or dextrose) and distilled water; yeast extract (i.e. concentration of dead autolyzed yeasts) and peptone (proteolyzed milk) provide all the kinds of amino acids and the vitamins, albeit their concentration is unknown, thus making YEPD a complex (complete) medium.

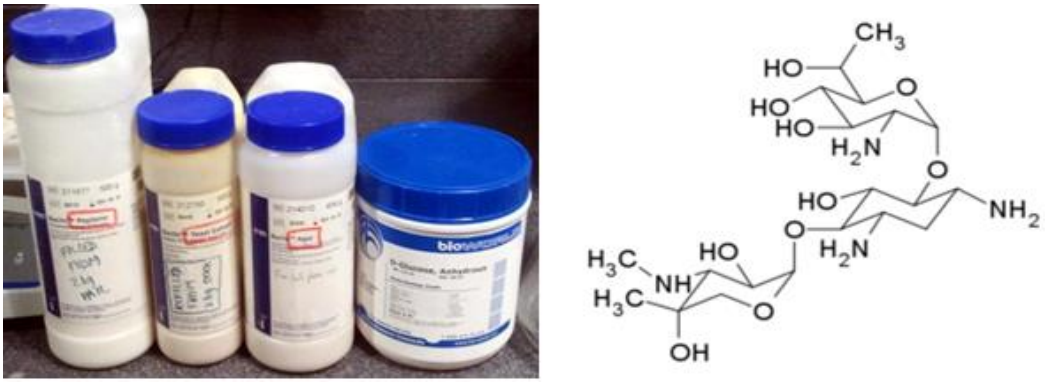


Image 2.3. On the left: constituents of the YEPD (Mark McCormick at http://getyourscienceon.wikia.com). On the right: structure of the Geneticin

YEPD can be transformed in a selective medium by adding an antibiotic, for instance G418 (i.e. Geneticin). This is an aminoglycoside antibiotic produced by *Micromonospora rhodorangea*, which acts by blocking the elongation step of the polypeptide biosynthesis, thus acting both on eukaryotes and prokaryotes. Therefore, this medium selects those yeasts that bear the correspondent resistance gene, the aminoglycoside phosphotransferase encoded by a gene from the Tn5 transposon.

YNB-SC (yeast nitrogen base, synthetic complete) is a complete medium made of YNB, glucose (or dextrose), SC and distilled water; YNB is a mix which provides all the essential vitamins and salts, whereas SC is a mix which provides all the amino acids, thus making YNB-SC a complete medium.
This medium is not very useful per se, but it is very useful if one or more amino acids are removed from the SC mixture (creating the so called "dropout mix"); in fact, this medium has selective power, impairing the growth of yeast cells which are auxotroph for the selected amino acids.

Yeast deletion strains library

The mutant strains I have used during my work derive from the results of a big international project, the Saccharomyces Genome Deletion Project (Giaever et al., 2002), aimed to create a library of yeast strains harbouring all the possible null mutations and then identify the function of all mutated genes.

This project generated four mutant collections: MAT-A haploid, MAT-α haploids, homozygous mutant diploids and heterozygous diploids (the only collection whose strains can carry non-viable mutations). Genes were mutated from the start to the stop codon, replacing them with a KanMX module (resistance gene against the Geneticin G418) and a specific tag, so that the strains can be identified using an array screening. To date, 90% of the genes have been mutated and the correspondent strains have been created, thus producing over 20.000 mutant strains.
It must be noticed that the genetic background (i.e. the genotype shared by all the mutant strains) is not wild type, rather all the strains share some mutations and are auxotroph for the same elements. At the present time, the whole library can be bought from several laboratories (ATCC, Invitrogen, Open Biosystems or EUROSCARF) and it is available in five different genetic backgrounds; only one background harbour all four collections, the other four harbour only the MAT-A or MAT-α collection.

Our laboratory owns a copy of the whole library; the mutant strains I have used in my work derive from the haploid MAT-A collection, whose genetic background is BY4241 his3Δ0, leu2Δ0, ura3Δ0. Yeast cells have been frozen while adhering to beads, and using glycerol as cryoprotectant; therefore, they are kept frozen in a -80°C freezer to preserve them and impede their growth. First time a mutant strain is needed, a single bead can be extracted from its tube and then plated in liquid or solid medium; next time that strain is needed, the researcher should use cells from the liquid culture or single colonies from the solid plate rather than using another bead.

2.2.2. Plasmids

Plasmids are small DNA molecule that can replicate independently and are separate from chromosomal DNA. They are very common in bacteria, but they can also be found in Archaea and simple eukaryotic organisms such as yeasts; usually they can be transferred between hosts (not necessarily belonging to the same species) via horizontal gene transfer (e.g. via bacterial conjugation).

There are several kinds of plasmids, which differ for many aspects such as sequence length, preferred host, DNA conformation (e.g. linear, circular and supercoiled and functions of genes they carry. Usually plasmids enhance their hosts' survival chances: resistance plasmids provide resistance genes, to avoid or destroy harmful compounds; Col plasmids and virulence plasmids provide genes which code for toxic compounds attacking bacteria and superior organisms (e.g. humans), respectively; degradative plasmids provide metabolic genes that are capable of digesting usually indigestible substances etc.
Natural plasmids, together with cosmids, viruses, artificial plasmids and chromosomes (e.g. BAC and YAC) are used as vectors in genetic engineering; they are commonly manipulated to alter their sequence, replacing naturally occurring genes with other genes of interest, adding new features etc. These plasmids are then transferred inside the host cells via transformation (natural uptake of naked DNA, only suitable for certain hosts), transduction (requires the use of viruses) or transfection (DNA inserted inside the cell by damaging cell wall and membrane). This way, cells can accomplish many different functions: producing large amounts of the desired protein or (normal amounts of) novel engineered molecules, amplifying specific gene sequences (e.g. useful when creating libraries) etc.; when using integrating plasmids that disrupt a chromosomal gene sequence, mutant strains can be obtained (see the Yeast Deletion Library above).

Independently from the specific sequence of each plasmid, all the plasmid used for the expression of recombinant proteins share some common features. First, non-integrating plasmids need a replication origin, namely a site where DNA polymerases can bound, whereas integrating plasmids need a specific sequence to integrate themselves inside a host chromosome. Second, any plasmid need to harbour at least one gene that allows for the selection of transfected cells; it could be a resistance gene that allows the cells to live in a selective medium, or a metabolic gene that compensate an auxotrophy and allows the cells to live in a non-complete medium.
Plasmids that are going to be used in yeast cells are often amplified (i.e. replicated) inside bacterial cells; that means that those plasmids need two replication origins, one for the bacterial polymerases and the other for the eukaryotic ones, and they often harbour two resistance genes (e.g. if bacteria and yeast are going to be selected using two different antibiotics).
Plasmids that possess all the aforementioned features (and some more) but do not carry the gene of interest are called "empty" plasmids; they are the starting point for creating recombinant plasmids.

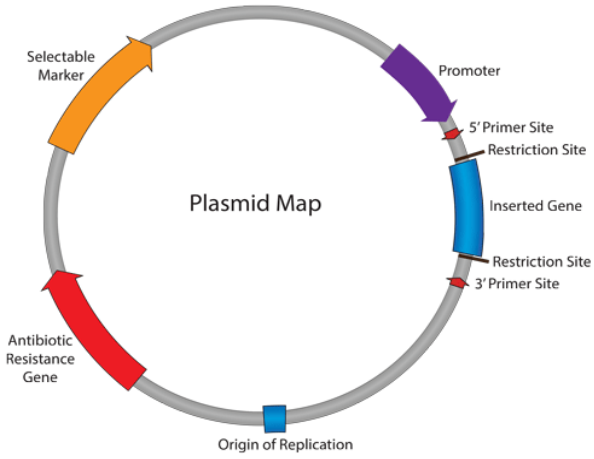


Image 2.4. Main elements of a "complete" plasmid (http://blog.addgene.org/topic/plasmids-101)

In other words, empty plasmids are capable of replicating themselves and rescuing the hosting cell from selective conditions, but they do not harbour the desired gene; rather they carry a counter-selecting gene, i.e. a gene whose product can harm the hosting cell (e.g. apoptosis proteins, prodrug activators). The gene of interest is then cloned inside the toxic gene, disrupting it; this way, only cells that have acquired the recombinant (i.e. not empty) plasmid can survive, thus increasing the yield of successfully transfected cells.

This is a sample protocol for creating yeast strains containing a non-integrated, non empty plasmid:

1) Amplifying the sequence to be transfected, e.g. using a PCR machine. If the sequence is not available in nature (e.g. genes codifying for tagged proteins or fusion proteins), further steps must be carried out. For instance, fusion proteins require the amplification of the single pieces, which are joined using restriction enzymes; tagged protein instead can be produced more easily by an accurate design of the primers, so that they already contain the tagging sequence.
2) Transforming *E. coli* cells with the empty plasmid and then selecting the transformed cells using a selective medium (e.g. LB and ampicillin). Since *E. coli* is not naturally competent to perform transformation, cells must be pre-treated (e.g. heat shock) to induce the transformation.
3) Extracting the empty vector from those cells and cloning the amplified sequence into it. This requires using restriction enzymes that recognize and cut the same short sequence both on the plasmid and the amplified sequence, and then ligases that repair those DNA cuts thus inserting the sequence of interest inside the plasmid; this insertion should disrupt the "toxic" gene.
4) Transforming *E. coli* cells with the newly produced plasmid and then selecting the transformed cells that bear the complete plasmid using an appropriate medium: e.g. LB and ampicillin (to select transformed cells) and a substance that can be converted in a toxic compound by the "toxic" gene (to counter-select the cells transformed with empty plasmids). Transformed cells can be stored, or the plasmid can be extracted and preserved in a -20°C freezer.
5) Cultivating yeast cells on a complete liquid medium (e.g. complex YEPD), and then transfect them with the extracted plasmid; in order to do so, electroporation or solvents (e.g. PEG) can be employed to weaken cell membrane. Transfected yeast cells can be cultivated on selective medium, be it a complex medium with antibiotics or a synthetic non-complete medium (when using auxotrophic strains and plasmids that allow recovering from that deficiency).

2.2.3. Spectrophotometric measurements

Whereas a standard spectrophotometer is useful for performing biochemical assays and measure instant absorbance or fluorescence of a relatively small number of samples, it is not enough when dealing with live cultures whose absorbance or fluorescence must be read over a long period (e.g. 48-72 hours). In those cases, a plate reader is much more useful; the device I have used in my work is the FLUOstar Optima reader (BMG LabTech).

It is a device that allows measuring absorbance and fluorescence directly from the plate wells (e.g. 96 or 384) in which yeast cells are growing; therefore, it combines the features of a spectrophotometer with those of an incubator (e.g. shaking the plates, keeping a constant temperature thorough the experiment).
The device is connected to a pc where all the data collected are recorded. The software running in the pc allows the user to define all the parameters:
- The general settings, namely which kind of measurement are carried out (absorbance, fluorescence, FRET etc.), the size and type of the plate used, the global duration of the experiment and the duration of each stage of the process etc.
- The specific parameters such as temperature, rounds per minutes, orientation of the shaking etc., but also absorbance or excitation/emission wavelengths.

In the software interface, each well can be renamed (this name will be used to flag the resulting data from that well) or flagged as "blank well", which is relevant for the following data analysis. The user can load previously saved parameters or protocols, and he can write and run homemade scripts, which are useful when combining multiple operations in the same experiments (e.g. measuring absorbance and fluorescence, or fluorescence in different wavelengths, in the same cycle).

Finally, the data recorded can be exported as csv (comma separated values) files for processing and statistical analysis (e.g. in an R environment), but they can also be accessed and analysed using the correspondent analysis software. For instance, recorded values of each sample of a 96-well plate can be represented as graphs nested inside a 96-boxes table, thus allowing for a quick comparison of all the samples. When analysing the results, the user can choose to deal with unmodified recorded values or with normalized values, which are the original values minus the mean values of the blank-flagged wells.

It must be noticed that the device can perform instant measurements; these are particularly useful for calculating the concentration of cells in each sample (i.e. well of a plate). Instant cell concentration measurements should be performed before any longer analysis in order to check that the correct amount of cells has been added to each well, namely a number of cells that can sustain an exponential growth.

3. Results

3.1. Network modelling

3.1.1. Drawing the network

First step: general layers

The very first step of my work has been the drawing of the network. We decided to use a complete network, comprising all the cellular processes, rather than limiting it to a single process; nevertheless, we did not want to (and we could not) characterize all the processes in detail, therefore I have focused on some processes and used some approximations to model the other ones.

The choice of creating a whole-cell network was driven by the idea that this work is not self-conclusive; rather it could be used as a starting point for future development in order to achieve a complete network, showing each process in detail. As modelling language, we choose Petri Net: it is relatively easy to learn and put into practice, and its features allows for creating multi-scale, multi-level networks that can be simulated.

At the very first level, I have drawn a network showing all the main cellular processes, common to all the living organisms: transitions such as “transcription”, “translation”, “metabolism”, and places such as “RNA”, “proteins”, “DNA” etc.
The main purpose of this level is providing a scaffold for the lower layers, where processes are investigated in more details. Therefore, it is made mostly of coarse nodes (both places and transitions), which are not part of the final matrices that I have employed to simulate the multi-level network.
This layer also hosts a simplified sketch of the cellular replication process, which culminates in a place that works as a cell counter.

In the second level, I have drawn networks that are slightly more detailed; nodes are still generic, i.e. they do not correspond to specific molecular species or reactions, but each network show a (simple) representation of the processes that constituted the transitions of the first level. This way, I have added a network that describes metabolism, another one showing transcription and another one representing translation. At this stage, the network can be employed to describe all the eukaryote cell, but it does not apply to bacteria; in fact, the translation network explicitly distinguish processes that happen in the cytoplasm from processes happening in the endoplasmic reticulum (which is absent in bacteria).
From this stage onward, I have deepened the description of translation processes only, leaving the other processes at this level of detail.

Second step: detailed layers

The second step consisted in the creation of the layers describing the translation process. This broad term include: translation; modification of proteins in the cytoplasm; translocation and modification of proteins in the endoplasmic reticulum, their following processing through the Golgi apparatus and their possible secretion or expression in membrane); degradation of protein via the ubiquitin-proteasome system and via the autophagy mechanisms.

The translation events that take place inside the cytoplasm, as well as the degradation processes, have been explained by using generic nodes instead of real gene products, but increasing the details provided by the network; therefore, some insights on the different stages of the translation, folding, posttranslational modification and eventually degradation are provided.

Differently, the "translation events" that take place inside the endoplasmic reticulum have been described in detail, using a set of intertwined networks; real gene products and their interaction have been employed in the drawing of these networks. In order to build them I have begun looking for known genes and their relationships; main sources of data have been the KEGG pathway website and the Saccharomyces Genome Database repository.

First, I have retrieved the pathway map of protein processing inside the endoplasmic reticulum; this map has provided me the name of the most important genes involved in protein processing, from the translocation of the nascent polypeptide chain to the export to the Golgi apparatus, as well as those genes involved in the ERAD and the UPR. This list can be found in the supplementary materials (6.4.2)

Second, I have looked for these genes in the SGD repository, aiming to understand their function, their interplay with other factors and the physical interaction occurring in the ER. This way, the original list has slightly changed by adding some genes whose functions could be useful in modelling the process and deleting some others that would have likely been useless in the model.

I could have kept them, but my aim was creating the simplest network possible, thus avoiding all the unnecessary agents that would have only increased the size and complexity of the network without improving it. In particular, I was interested in genes whose deletion gives rise to viable mutant strain, which I needed for the experimental part of my work.

The final list consists of:

- Proteins involved in the protein folding, such as chaperones and disulfide-isomerases;
- Enzymes involved in the N-glycosylation and subsequent changes in the sugar chain;
- Signalling proteins and downstream signals (outside the ER too);
- Channel constituents for protein translocation, retrotranslocation etc.;
- Proteins involved in the vesicles formation and protein export;
- Ubiquitinating enzymes and ancillary proteins involved in the ERAD.

Third, I have used those genes to draw a network showing the ER processes and another that focuses on the ERAD system. It must be noticed that these networks occupy the same level of the hypothetical hierarchy of the whole network, meaning that they could be merged in a single network; I have preferred keeping them separate simply for clarity reasons.

I have "translated" KEGG pathway maps to Petri Net models to draw these networks; since those maps only show agents (i.e. places), this has required the explicit definition of the actions (i.e. transitions) and the molecular species that are subject of these agents and actions (i.e. different stages of protein maturation). Of course, I have also had to determine how to represent relationships and interaction among genes in the Petri Nets; again, I have tried to use the minimum amount of edges (and transition), in order not to overload the network itself.

The next step of this "translation" process has been deciding the names of the nodes; since these projects are made to be public, and maybe extended by other authors, I had to find and use unambiguous names for the nodes of the network. At this stage, I have also changed the temporary names of all the nodes in the rest of the global multi-level network, in order to write them using the same criteria.

- For places representing real gene product, I have adopted the standard name (i.e. the official Gene Symbol), retrieved from the SGD; I have chosen these names because they are short (they fit well in the network) and yet unambiguous.
- I had to invent the names of the general places (be they coarse or not),, trying to employ terms that could easily describe the molecular species/category; for instance, I have used the term "G3M9 glycosylated intermediate" to define a specific stage of the protein processing inside the ER, where "G3 M9" refers to a specific stage of the sugar chain processing)
- For all the transitions, I have decided to employ names that are recorded in the Gene Ontology vocabulary, in its domain of functional terms. Using these terms in a network is useful not only for improving its clarity but also for helping in the hierarchical organization of the transitions, which should follow the organization of GO terms; therefore, it is also useful in organizing layers, since each lower layer corresponds to a coarse transition in the upper layer.

Fourth, I have connected all the single networks together. This means joining in the same project all different layers and ensuring that the hierarchy is maintained, but also linking "sibling" networks, i.e. networks that lay in the same layer; coarse transitions and dispersed places have been instrumental in linking layers and sibling networks, respectively.
Once the network has been completed, I have specified all the necessary parameters, namely weights of the edges, mass action parameters of the transitions and tokens of the places (the rationale I have used is explained in the next paragraph). Finally, I have exported the final matrix and manually checked it in order to verify that all the interactions had been correctly written.

Third step: reporters in the network

The third step consisted in adding reporters in the network, i.e. places that could be employed to summarize the "phenotype" of the network, thus allowing for comparison between model and experimental data.

As I mentioned in the introduction, a possible reporter could have been the growth rate, namely the number of cells divided by the simulation time; the time depends on the algorithm adopted, whereas the number of cells can be read in the "cell counter" place located in the uppermost layer.
Some changes might be needed on the simulation script when using this reporter. For instance, the network may represent a cell that can only live by acquiring nutrients from the external environment, and those nutrients are limited and must be shared by all the cells in the culture. In this case the finite number of tokens in the "external nutrient" place must be divided for the number of cells generated, so that the network have less nutrients and could reach a dead state before expected.

Another possible reporter could have been the number of dead cells due to apoptosis. Apoptosis could be modelled as a reduction in the aforementioned "cell counter", or it could be drawn as a pathway leading to a dead state of the model (e.g. by removing tokens from fundamental places, such as the place corresponding to the DNA).

The use of GFP as reporter (normal and fused with an ER-targeted protein) does not require any other additional reporter in the network, since its abundance can be studied using generic places that are already in the network. The idea is that the level of the GFP corresponds to the mean levels of all the other proteins, or at least changes in the GFP level are proportional to changes in the levels of other proteins.
Therefore, normal GFP levels can be evaluated looking at the amount generic cytoplasmic proteins, whereas ER-targeted GFP levels and fusion protein levels can be obtained from the amount of generic ER-processed proteins. It must be noticed that the simulated values refer to a single cell only whereas the measured values refer to many cells, and therefore must be divided by the number of cells.

The last reporter considered is GFP transcription dependent on the Unfolded Protein Response.
I could have added to the network a gene that is specifically transcribed and translated after UPR; this way, levels of its gene product could have been evaluated and confronted to the amount of UPR-dependent GFP produced after UPR events in real cells.
However, due to the design of the whole network, I have envisioned that the abundance of the activated transcription factor could be used in place of the protein produced by the action of that transcription factor. This way I could "recycle" a place that is already in my network rather than adding a completely new pathway, thus simplifying the following stages of simulation and optimization of the network.

3.1.2. Focus on: modelling problems

Choice of the parameters

In theory, all the parameters of a biological network could be retrieved from literature; in practice it could not be made because the network is not homogeneous, i.e. some places are specified at gene level, others represent generic "agents". First, some of the parameters are not known, and many are only been studied in standard conditions, so that we would have no idea of their values in all the other conditions. Second, even if we knew the all the values of all the parameters in all conditions, we could not know whether these parameters could be employed in simulating an incomplete network.

Therefore, some parameters must be estimated; it is important to choose these values properly because if the simulation starts from a not biological plausible network, it might be impossible to reach a working network (i.e. a network that fit the experimental data) using the algorithms I have developed. This is because this multidimensional parameter optimisation has a very rugged solution space, which is typical explained using the Travelling Salesman Problem (Applegate, 2006). Therefore, it would require very sophisticated Monte Carlo techniques and a large amount of CPU time to be solved. In order to reduce the complexity of the problem, I have decided to start from a "biological plausible network", showing those properties:

- Its simulation tends to reach a dead state, that is, the catabolic reactions in the network must destroy more tokens than those created in the anabolic reactions. In other words, since the network represent a closed system, it must be avoided the creation of self-sustaining networks, which would violate the second principle of thermodynamics.
- It must be possible to avoid dead states by supplying nutrients (i.e. tokens) from external sources; that is, network must not reach dead states before nutrients are finished (unless apoptosis intervenes) because of the action of some errors in the architecture of the network. External sources could be unlimited or not, thus allowing for simulating cultures growing in media whose nutrients level is maintained constant or depletes during the cell growth.
- During the simulation, the amount of tokens in each place can oscillate but it must remain within some boundaries. More generally, tokens must not accumulate in few places leaving the others empty. It should also be avoided that some transitions fire not even once during the simulation, whereas some others are always firing; more generally, even if the simulation must go through some pathways very rarely, it should be avoided the creation of short circuits that cause the simulation going through very few pathways.
- Last but not least, since this network has been created by substituting read-only edges with coupled normal edges, edges forming a couple must have the same weights. On the other hand, when a place is both read and used in the firing of a transition, the weights of the edges in the couple must diverge, the pre-arc weighting more than the post-arc.

These rules apply to the architecture of the network and the weights of the edges, but they also apply to the other two kinds of parameters, namely tokens in each place and Mass Action parameters.

I have assigned tokens to places in order to create a configuration of the network that, when simulated, could show the aforementioned properties; in particular, the amount of tokens greatly impact which transitions are chosen and therefore it determines whether tokens accumulate in some places.
Therefore, I have assigned tokens to each place in order to enable all the transitions, except those inhibiting processes, and then I have manually modified some of these amounts of tokens in order to obtain the desired feature. These markings have no biological meaning, but they can be employed for our purposes in the following optimization stages.

Mass Action parameters have been set to "1" as default; the idea is that network properties should emerge independently from this kind of parameters, which are the real parameters that are going to change during the optimization process. Nevertheless, some parameters had to be decided at this stage; for instance all those transitions that have no biological counterpart but are required for the network functioning, have parameters set to infinite, so that their duration (calculated using the Gillespie algorithm) is zero; these parameters will not be changed during the optimization stage. Besides, in my network there are no concurring pathways, i.e. a classic metabolic pathway and a salvage pathway producing the same molecular species, but if they had been in the model, I would have used Mass Action parameters to insure that the first pathway would have been preferred over the second one.

Moreover, in order to avoid unwanted dead states, catabolic processes must be kept under control, so that tokens are not consumed too quickly. At first, I had thought of drawing these processes as transitions requiring many tokens in the input place to fire (read-only edges), even if consuming only one token. This solution would have solved this problem, but it would have caused a limitation to the stochasticity of these processes: the architecture would have not allowed the transition to fire when the number of tokens in the input place is low. Then, I have decided to employ mass action parameters instead of weights to model these processes; I have set these parameters to 0.1, so that the corresponding transitions would happen less frequently. This solution solves the issue and, at the same time, it lets the model be fully stochastic: these transition can always happen (if the input places are populated), even if that cause a place to be totally emptied of its tokens.

Image 3.1. On the left: first solution (as drawn in esyN and Snoopy). On the right: second solution

Logic gates

By the term "logic gates" I refer to transitions that do not have any biological counterparts or meanings, rather they are necessary for the network architecture; as I mentioned before, they all have parameter values equals to infinite. I have drawn the templates of all these gates, which are now available inside the esyn.org website and can be added as "modules" to Petri Nets by any users.

They are created and applied with the purpose of performing simple logic operation. For instance: AND (standard operation of Petri Nets) means that all the input places must be populated in order to enable the firing of the transition; OR means that at least one input place must be populated; XOR means that not all the input places must be populated; NOT is the standard action of the inhibitory edges etc.

As I mentioned in the introduction, coarse places are considered as implicit AND gates; since a coarse place linked to a real transition could be written as if all its child places were linked to the same transition, this means that all the children places must be populated to enable the firing of the transition. This conversion is automatically performed when writing the network as matrices.

This feature is useful when the coarse place represents a molecular complex: all its subunits must exist in order to produce a complex that can perform its activity. On the other hand, it would be detrimental when the coarse place represents a generic category that comprises its child places but it is not limited to them. In this case, it would be a mistake interpreting the link between coarse place and real transition as an AND logic gate; rather, it should be converted into an OR logic gate.

It would be a useful extension of the drawing tool if allowed the user to choose which kind of conversion must be applied to each coarse place; however, currently this conversion towards OR logic gates must be done manually:

- First, the real transition must be converted into a coarse transition; this way the automatic conversion to AND gates cannot be trigged when obtaining the matrices. The problem is that coarse transition and coarse places are non-existing nodes, thus the link between them is lost.
- Second, a logic network is created inside the newly made coarse transition. It is a network showing only the "OR" logic gate: each children place is linked to a different copy of the original transition, and they all produce the same output place; therefore, if one children place is populated, that is enough to produce tokens in the output place.

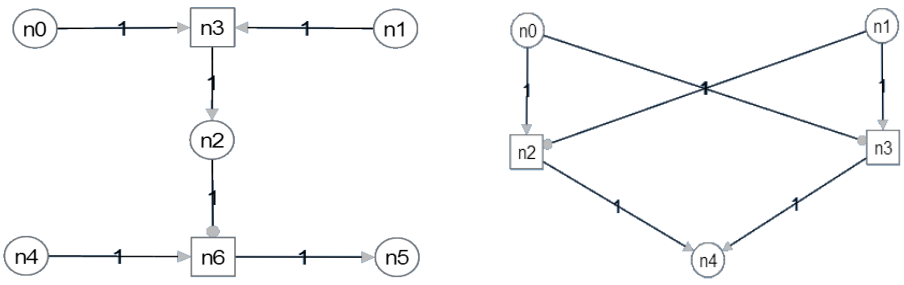


Image 3.2. On the left: NAND logic gate. On the right: XOR logic gate. Source: esyn.org

3.1.3. The final network

All the single networks that compose the final network can be found in the supplementary materials; hierarchy of places and occurrence of dispersed places in the network is also noted there. Here, I will comment some other interesting features and descriptions.

The first level is occupied by a network called "whole cell"; it is made of

- coarse places: CYT-Functional proteins (i.e. all the proteins that work in the cytoplasm) and ER-Functional proteins (i.e. working in the endoplasmic reticulum)
- coarse transitions: Transcription, Translation and Metabolism
- non-coarse transitions (for now): Cell division, DNA replication, Apoptosis (it is a "well" transition that consumes DNA tokens without producing anything)
- non-coarse places: Apoptosis factors, CYT-Polymerases (comprising both DNA and RNA polymerases), DNA, RNA, Amino acids, Nucleotides, Cells counter.

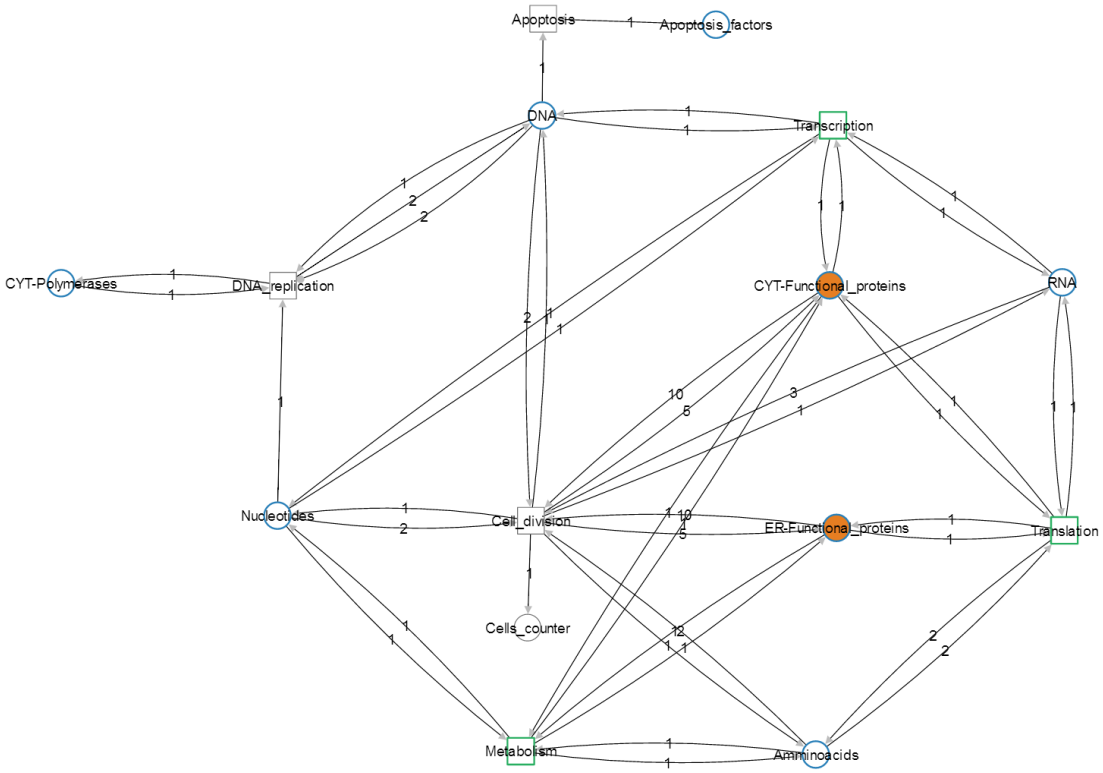


Image 3.3 Example of a non-detailed network: "whole cell" network.

The second level is occupied by three networks corresponding to the coarse places of the first level. Transcription network is made of:

- coarse transitions: none, because this is already the highest level of detail for this field;
- non-coarse transitions: gene transcription, RNA processing and preRNA and RNA destruction;
- non-coarse places: DNA, RNA, Nucleotides, CYT-Polymerases, preRNA.

Metabolic network is made of:

- coarse transitions: none, because this is already the highest level of detail for this field;
- non-coarse places: Nucleotides, Amino Acids, Energy, Metabolites, External metabolites, membrane transporters, CYT-Enzymes;
- non-coarse transitions: nucleotide and amino acid catabolism, i.e. destruction of nucleotides/amino acids, generating generic "metabolites"; nucleotide and amino acids anabolism, i.e. the opposite reactions, which require "metabolites" and energy; metabolic reactions, i.e. using metabolites to produce energy; uptake, which transfers tokens from the "external metabolites" place to the internal "metabolites" place.

Translation network is made of:

- coarse transitions: CYT-processing and ER-processing;
- coarse places: CYT-Functional proteins and ER-Functional proteins (linked to coarse transitions);
- non-coarse transitions: "translation starts", i.e. the first step of translation, which is common for proteins that mature in the cytoplasm and in the endoplasmic reticulum;
- non-coarse places: RNA, Amino Acids, Ribosomes, Primed translation (the output of the non-coarse transition), eIF2α (working as inhibitor of the translation, activated by UPR events).

The third level consists of four real network and four logic networks. I have already mentioned how logic networks are created and how they work; therefore, I will describe their purposes only:

- "ER retaining" and "CYT localization" networks are required for modelling the production of a generic ER-processed and CYT-processed proteins;
- "hidden transition 4" and "vesicle formation" are required for modelling the opposite reactions, i.e. the degradation of a generic CYT-processed or ER-processed protein.

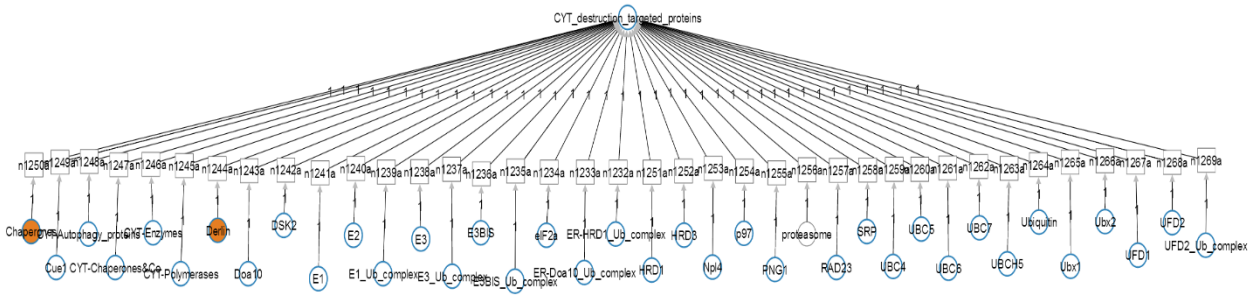


Image 3.4 Example of a logic network: the same transition is repeated many times but there is one output only

The "ERAD retrotranslocation" network describes the steps from the recognition of misfolded proteins to their addressing to the proteasome, thus including the ubiquitination steps. Some of the most interesting nodes are the coarse places representing molecular complexes: Chaperones, CYT-Hsp40, Sec61, PDI and Derlin.

The “CYT processing” network describes the protein processing inside the cytoplasm. It is made of:

- coarse places: CYT-Functional proteins; it has required the introduction of coarse transitions containing logical networks, i.e. “logical hidden transition 4” and “logical CYT-localization”;
- real coarse transitions: catabolic process, another network of the third level, although it might be speculated that it forms another independent level;
- non-coarse places: Primed translation, Amino acids, Ribosomes, CYT-Chaperones, unfolded protein, folding intermediate, folded protein, destruction targeted protein;
- Non-coarse transitions: “translation continues”, CYT folding, Posttranslational Modifications, and hidden transitions 1, 2, 3. The first three are required for the protein processing, the other three are required for the creation of a OR logic gate inside this network; this gate is required to model the degradation of proteins and protein intermediates.

The “ER processing” network is very vast, describing all the steps of protein folding, modification and targeting toward the Golgi or toward the destruction (ERAD), and the subsequent steps of processing in the Golgi etc.; here are described the most important nodes:

- “ER-Functional proteins” is a generic coarse place, which has required the introduction of a coarse transitions called “logical ER retaining”
- “ERAD retrotranslocation” is a real coarse transition, containing another network of the third level (same consideration applies as before).
- Sec13/31, Sec23/24, ER hsp40, ERAD-factors, OST, Sec61 are coarse places that represent molecular complexes; therefore, they do not require any logic gate and network. Their children places are located in the same network, encircling their correspondent parent places; they are easily spotted because they have no connections to any nodes.

The “catabolic process” network contains both the autophagy and the ubiquitin-proteasome pathways, expressed as generic processes, i.e. there are not real gene products. Cytoplasmic processed proteins (both native and misfolded) targeted for destruction can enter both pathways; ER processed proteins can only enter the autophagy pathway; ER-misfolded proteins can only enter the ubiquitin-proteasome pathway (at the end of the pathway because the ubiquitination steps have already been carried over in the “ERAD retrotranslocation” network).
In this network the most interesting nodes are the “ER-Functional_proteins” coarse place and the correspondent “logical vesicle formation” coarse transition: the coarse places represent a generic place, which therefore has required the use of a coarse transition containing the fourth and last logic network.

3.2. Coding the scripts

3.2.1. Simulation script

Input files

As I have mentioned, networks can be written as matrices and vectors, which represent the input files for any simulation script; when creating the matrices, all the networks levels are collapsed into one.
Vectors are employed to represent the names of the places, the name of the transitions, the marking of each place and the Mass Action parameters for each transition.
Matrices are employed to represent the weights of the edges; their rows consist of transitions whereas their columns consist of places. Therefore, matrices can summarize inhibitory edges, read only edges, inward edges (i.e. edges going towards transitions), outward edges (i.e. edges going toward places) etc.

Snoopy software allows the user to export matrices as a single MatLab files; then, an in-house developed python script converts this file into multiple text files, each bearing one vector or matrices. These text files are the input of the R script.
There are several functions in R to read external files and save the read data into a variable; examples of these functions are "read.csv()", "read.delim()", "read.table" and so on. They differ for the kind of input they can read (e.g. numeric matrices, generic tables, table written as comma separated values files, incomplete tables and so on) and the class of the variable they produce (e.g. matrix, vector, list etc.). When calling these functions in a script, other features can be set; for instance, a function could be written so that it does not count the first line of a file, which is particularly useful when dealing with files that have a header.
Therefore, this is the command I have written: *matrixVariable <- as.matrix(read.delim("file path.txt")).*
It must be noticed that the matrices representing input edges have positive values even if those edges actually subtract tokens from places; therefore, their sign must be changed into negative before employing those matrices.

esyN tool allows the user to export matrices as a single JSON file. This file can be read in the R environment, provided that the "rjson" package is installed and working, and its content can be assigned to a variable. Matrices and vectors can be extrapolated directly from this variables and assigned to the correspondent variables; the prototype command is *matrixVariable <- filevariable$part_of_the_file* for vectors and *matrixVariable <- do.call(rbind, filevariable$part_of_the_file*) for matrices.

I have employed the first method when trying to draw and simulate Snoopy networks, but I have hastily moved to the second one since I have started using esyN tool for network creation.

Other inputs

In both cases, during the input reading stage some more variables must be set.

The number of steps ("stepsNumber" variable) determine how accurate the simulation is, meaning that a certain amount of steps is required for all the network properties to emerge, and if a lower amount of steps is chosen, some properties could be lost. Unfortunately, this number of steps is unknown and therefore an arbitrarily high number must be set instead; even better, the whole simulation should be repeated using different numbers of steps and thus obtaining the minimum amount required.

The number of iteration of the whole simulation ("iterNumber variable) determines how many times the simulation is repeated; the purpose of these iterations it will be explained in the next paragraphs.

The number of points (spotsN) and the threshold (sensSd) variables are important in the analysis of the simulation data. The idea is that each simulation output is an irregular time series and therefore, when calculating mean values among the simulations, these time series must be interpolated to some specific common time points: spotsN determines the number of this spots. If some simulation lasted sensibly longer or shorter than the mean duration, they should not be considered; the threshold variable determines how distant (in standard deviation units from the mean) a duration must be in order to cause the discarding of the correspondent simulation.

These parameters are set to default values that, based on my experience, should allow for a relatively quick but reliable simulation; they can be changed in the script or during the simulation, writing them in the user interface when prompted to do so.

Moreover, during the simulation, it could be possible to analyse a specific place of the network rather than the whole network. In order to do so, its name must be assigned to the "choicePlaceName" variable, so that its corresponding column in the matrices would be assigned to the "choicePlaceNumber" variable. By default, there is no place assigned to these variables (each network differs and has different place names), rather they are written so that all the places of the network are considered.

Finally, in the input setting stage, some other variables must be initialized (even if no value is assigned to them at this stage), libraries must be loaded and possible external scripts must be called. In fact, since all the scripts I have written share many of their functions (e.g. reading input files), I have written those functions in external scripts, which must be called for the scripts to work.

It must be noticed that another version of the simulation script exists; in that version, the script is complete, meaning that there is no need to call any external scripts, but it is actually split into two (simulations and the analysis of the simulation. This latter version is that one available on GitHub, accompanying the esyN tool.

Totally stochastic core

The core of the simulation consists of a complex function, which could be written in the script code or in another file; both versions are available in the supplementary materials (6.2). Its functioning is quite simple: the simulation is repeated for the selected number of steps, unless a dead state is reached beforehand; the total amount of steps performed corresponds to the duration of the simulation. Therefore, the function has the following workflow:

1) A probability vector is created from the vector containing the Mass Action parameters, and a transition vector is created, containing numbers from one to the number of transitions.
2) A transition is randomly chosen, taking into account the likelihood of each transition
3) Some "if clauses" check whether the network contains inhibitory edges and, if so, whether the chosen transition is inhibited or not.
4) If it is not inhibited, another "if clause" is employed to check whether it is enabled or not, i.e. whether each place involved in that transition has enough tokens for its firing.
5) If it enabled, then the marking of the network is updated by subtracting the consumed tokens and adding the produced ones; this new marking is recorded in a global matrix containing the markings of each step. Then, another step is performed starting from point 1.
6) If the transition is inhibited or not enabled,
   then the probability and the transition vectors are updated by deleting that transition; the step is then repeated from point 2, without updating the marking of the network. If all the transitions are deleted, then a dead state is reached; in this case, the simulation stops.
7) When the simulation end (or is stopped), the global matrix (containing all the markings at each step) is returned to the script and stored in another variable.

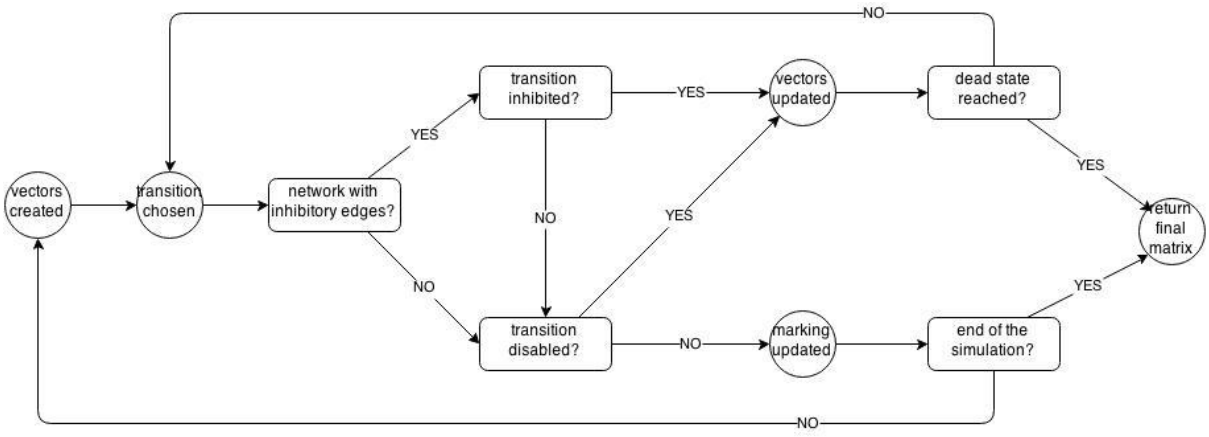


Image 3.5 Core functions workflow (both stochastic and Gillespie cores)

I have also tried to employ a different approach, namely calculating which transitions are enabled before choosing one; this way, I could have avoided picking disabled transitions. I have abandoned that path because it was clear that the script would have been computationally heavier, not lighter, than the previous one.

Gillespie core

The core of the Gillespie simulation is almost identical to the core of the totally stochastic simulation; the only features that differ are the formula employed in the choice of the firing transition and the method employed for calculating time. The final matrix contains the markings reached and the elapsed time at each step of the simulation. Therefore, the function has the following workflow:

1) "sweep()" function is employed to sum, for each transition, all the tokens inside its input places: the input matrix signals which places are involved, the tokens vector define the number of tokens in each place. Therefore, a vector is created; by multiplying it and the vector containing the Mass Action parameters, the final probability vector is obtained. Moreover, a transition vector is created, containing numbers from one to the number of transitions.
2) For each transition, an exponential distribution is calculated using the correspondent value in the probability vector as rate of the function (i.e. inverse number of the function mean value); a random number is generated from each distribution, thus creating a time vector.
3) The minimum value of this vector is the "waiting time" of the simulation step; the firing transition is its corresponding transition (i.e. that one that has generated the exponential distribution and the random number).
4) Some "if clauses" check whether the network contains inhibitory edges and, if so, whether the chosen transition is inhibited or not; if it is not inhibited, another "if clause" is employed to check whether it is enabled or not, i.e. whether each place involved in that transition has enough tokens for its firing.
5) If it enabled, then the marking of the network is updated by subtracting the consumed tokens and adding the produced ones; total elapsed time is updated by adding the newly calculated "waiting time". This new marking and the elapsed time are recorded in a global matrix containing the markings of each step; then, another step is performed starting from point 1.
6) If the transition is inhibited or not enabled, then the time and the transition vectors are updated by deleting that transition; the step is then repeated from point 3, without updating the marking of the network and the total elapsed time. If all the transitions are deleted, then a dead state is reached; in this case, the simulation stops.
7) When the simulation end (or is stopped), the global matrix (containing all the markings at each step) is returned to the script and stored in another variable. If the simulation had stopped before its natural end, the last line of the matrix is repeated but the amount of time is made negative, in order to flag that state as a "dead state".

I have also tried to employ a different approach, namely calculating which transitions are enabled before choosing one; this way, I could have avoided picking disabled transitions. I have abandoned that path because it was clear that the script would have been computationally heavier, not lighter, than the previous one.

Iterating the simulation

As I have mentioned before, the simulation scripts I have creates allows for the reiteration of the simulation; the number of times a simulation is repeated is set in the "iterNumber" variable.

Reiterating is useful to calculate mean values and to reduce the intrinsic variability of the simulation output, i.e. that variability that is not due to the stochasticity of the cell processes, rather to the stochasticity of the algorithm itself.

Moreover, when reiterating new properties emerge, i.e. the properties of a group of cells (a colony, for instance) rather than the properties of a single cell. As example, I have reported the behaviour of a feedback loop (Blatke, 2011, p. 50); the three following images represent the Petri Net model (upper diagram) and the simulation outputs when iterating the simulation 100 times (graph on the left) or when no reiteration is performed (graph on the right).

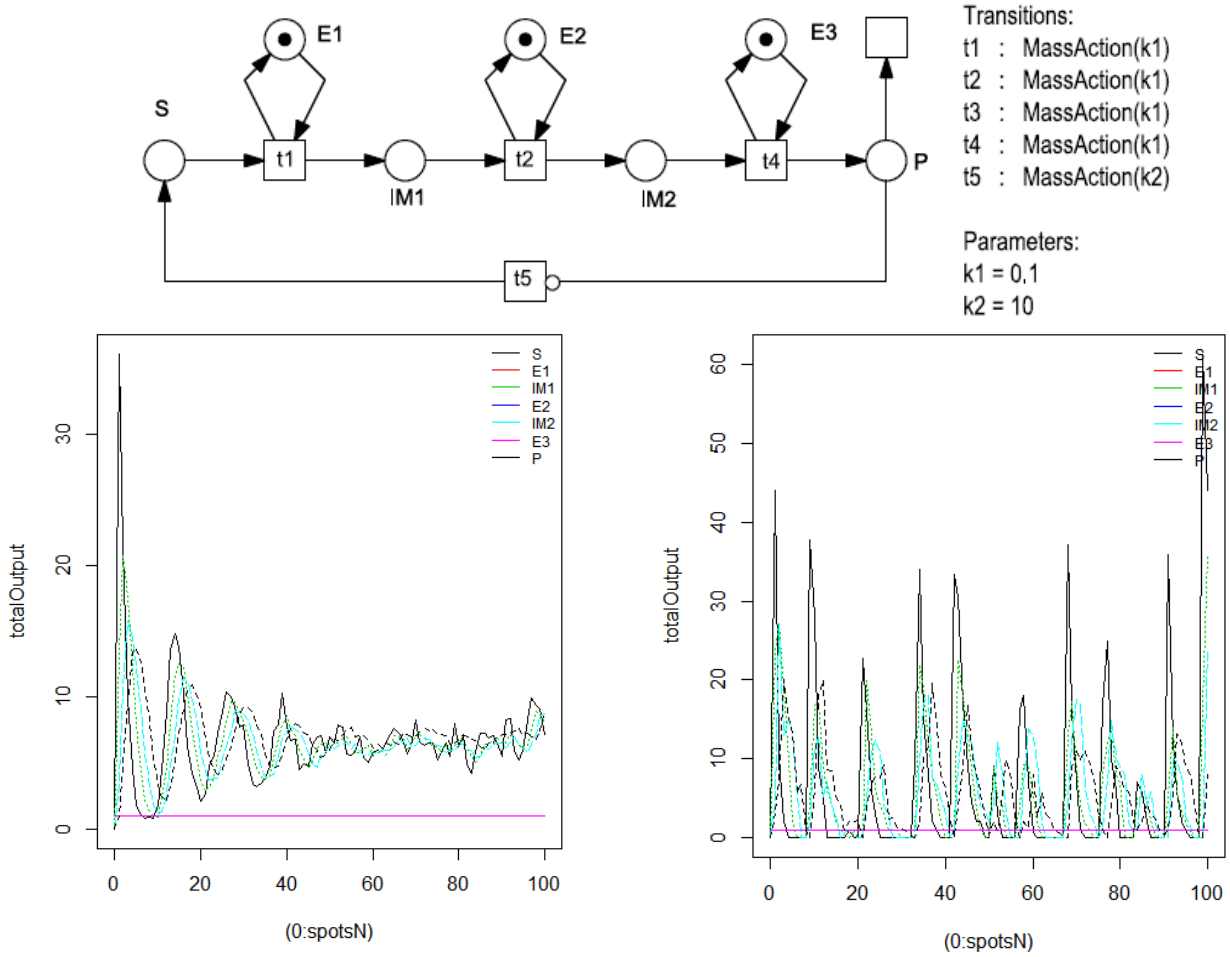


Image 3.6 Petri Net diagram (upper picture) and graphs representing the output of its simulation.

The oscillatory behaviour produced by the negative feedback loop can be easily noticed when performing a single simulation; when iterating it, though, another feature emerges: peaks are attenuated during the simulation, 'till all the places that can accumulate tokens (i.e. not the places representing enzymes, such as E1, E2, E3) tend to have the same mean amount of tokens. This feature could not be guessed without repeating the simulation and calculating the mean of the results.

It must be noticed that many statistical analyses can be performed on the results of the simulation; calculating the mean among them is one of the simplest and more widely adopted, and that is the method I have employed in my scripts.

In order to calculate mean values, some steps must be performed:

1) Simulation is repeated many times and the output of each simulation is recorded in independent variables; a time vector is created, containing the duration of each simulation
2) Mean and standard deviation are calculated from the distribution of those durations, and a graph of this distribution is plotted. Outliers, i.e. simulation that lasted sensibly more or sensibly less than the mean duration, are removed; the exclusion distance is set in the "sensSd" variable.
3) Among the remaining simulations, the shortest duration of the simulation is chosen as the time of the mean simulation; that means that all the other simulations are trimmed to that time, discharging the markings recorded in the steps that had followed that time.
4) A time vector is created by dividing the minimum time calculated into a number of time points; this number is set in the "spotsN" variable and determines the accuracy of the following steps. Each simulation, which is an irregular time series, is converted into a "zoo" object and markings (i.e. tokens in each place) are interpolated for each time point; this way, each transition is converted into a regular time series.

   It must be noticed that this step can be skipped when employing the totally stochastic core of the simulation; in facts, in that case the simulation are already regular time series, and the markings are recorded at the same time points, therefore interpolating is not necessary.
5) A mean time series is calculated using all the non-discarded simulations, i.e. calculating the mean amount of tokens in each place in each time point; moreover, a summarizing table is created by joining the final markings of each simulation.

Those matrices are exported and saved as text files, in the same folder that contained the input file, so that they can be accessed anytime outside the R environment; the mean time series is useful for studying changes of mean markings during the simulations, whereas the summarizing table is useful for comparing the results of different simulations.

Moreover, a graph is plotted by using data from the mean time series; the x-axis represent the duration of the time series, i.e. the duration of the shortest non-discarded simulation, whereas the y-axis represent the amount of tokens in each place. If a specific place had been set, the graph is created using the "plot()" function and it only shows the trend of that place. If no specific place had been chosen, the graph is created using the "matplot()" function and it shows the trend of all the places; the graphs employed in the description of the iteration stages have been created using this function.

3.2.2. Optimization script

The starting point 1

In my work, the training of the network has been performed by optimizing the network, i.e. changing its parameters so that the ratio between the final value of a reporter in wild type and mutated networks is the same of the ratio measured in the experimental part (or found in the literature). As I have mentioned, it employs a Monte Carlo method; inside this method the values of the reporter are calculated by performing a simulation with the aforementioned Gillespie algorithm. This is why I have written all those scripts as pieces that can assemble in different ways, so that the main formulas could be employed in several circumstances; therefore:

- The input files are the same files that are employed in the simulation scripts, and are called/read/assigned to variables by using the same external scripts; however, since this optimization phase has been performed only on networks drawn in esyN.org, I have only employed the script that reads the JSON file.
- The simulation is performed by calling the same functions employed in the simulation scripts, the only feature missing is the iteration of the simulation; each simulation stage is repeated twice, once using the parameters of the "wild type" network and once using the parameters of the "mutated" one.
- The output files are text files that can be read as input files of the simulation scripts, so that the simulation can be performed without changing the network; they can also be read by the user and applied to change the network, thus obtaining its final version. As I will explain later, they contain the vector representing the (changed) set of Mass Action parameters.

A mutated network is a wild type network with a place missing, which corresponds to the mutated gene. The problem is that deleting a place alters the equilibrium of the network: upstream transitions, i.e. transitions that produced tokens in that place, are transformed into "well" transitions, which consume tokens without producing them; on the opposite, downstream transitions, i.e. transition that required tokens, are transformed into "source" transitions.

Therefore, in order to prevent upstream and downstream transitions from firing, they can be deleted from the network or their Mass Action parameters can be set to zero, so that they cannot ever happen. Instead, setting the amount of tokens of the mutated place to 0 prevents downstream transitions but do not affect the upstream ones; finally, changing the weights of edges, does not solve the problem at all.

These changes are not performed in the network drawing, rather they are carried on in the optimization script itself; that makes easier deciding which place should be mutated, and simulating the resultant outcome. I have written a few variants of the script, differing in the way mutated networks are created: setting the Mass Action to zero is the easiest way, whereas deleting the transitions seems to be the most efficient one from a computational point of view.

The starting point 2

Practically speaking, this script changes the Mass Action parameters of the network, thus modifying the rates of the transitions rather than the architecture of the network; the final output is therefore a new vector of these parameters. Before starting the simulation, some parameters, specific of the optimization stage, must be chosen:

- The number of Monte Carlo steps ("mcNumber" variable) determines how many changes are applied to the original set of parameters before the optimization ends; as I have mentioned in the introduction, this number should be the smallest number possible that allows for reaching the optimized configuration.
- The number of total iterations of the optimization process ("totIteration" variable) determines how many times the whole script is repeated; the purpose of these iterations it will be explained in the next paragraphs.
- The minimum and the maximum weights ("minWeight" and "maxWeight" variables) determine the smallest and the highest value that can be assigned to a Mass Action parameter during the phase of random changes of the parameter set; by default, they are set to 0.1 and 100 respectively.
- The inverse temperature ("invTemp" variable), usually set to 0.5, influences how likely is that an otherwise rejected conformation is actually accepted; by "rejected conformation" I mean a set of parameters that increase the difference between the simulated and measured ratio instead of reducing it. Its name derives from the value assigned to this parameter when using Monte Carlo methods for structure optimization: in those cases, it is equal to $\frac{1}{K_b T}$.

Therefore, the first steps of the optimization algorithm can be reassuming in the following way:

1) The input files are read, the external script are called and all the script parameters are set (both the optimization and the simulation parameters); in this step, users are prompted to set the real measured ratio of reporter values from wild type and mutant cells.
2) Selection of the "reporter" place, whose levels will be "measured" and used in the optimization step. It is handled as the place that is specifically analysed in the simulation script: first, it can be changed in the script or the user can interactively assign it; second, it is assigned to the same "choicePlaceName" variable and translated into the "choicePlaceNumber" variable.
3) Mutation of the network, i.e. creation of matrices and vectors representing the mutated network; it can be done in any of the methods discussed above; the place to be mutated is written by the user when prompted to do so. It must be noticed that if the selected methods involved the deletion of places and/or transitions, the size of the matrices and the length of the vectors change; in this case, it might be necessary to "retune" all the formulas (e.g. the choicePlaceNumber) in order to be sure that they refer to the same item in the variables.

The workflow

Once the script has been initialized, the proper optimization can begin; as the corresponding script, it is divided into three main stages:

- One value of the vector containing the Mass Action parameters (from here on "vectorPar") is randomly chosen and then changed into a random number, extracted from a uniform distribution extending from minWeight and maxWeight.
- Gillespie simulations are performed on the wild type and the mutant networks using the corresponding function, and the resulting matrices are stored into variables; from this matrices, the final value of the reporter place and the final duration of the simulation are obtained. It must be noticed that the simulation is performed only once per network; this is necessary in order to keep the computational cost under control, even if this might cause loss of information.
- An "if clause" check whether the mutation introduced has produced networks that reach dead states; this is performed by simply evaluating the sign of the duration of the simulation, since negative values are employed to flag dead states. If not, another "if clause" checks whether it has produced networks that are incapable of increasing the amount of tokens in the reporter place (i.e. its value is 0).
- If all those clauses are false, then a ratio is calculated between the values of the simulated reporter in the mutant and wild type networks, and it is subtracted from the value of the experimental ratio; this way, a delta value is calculated. This delta value is compared with the delta value calculated in the previous Monte Carlo step (in the first step, infinite is employed as the old delta value) using the following statement: $e^{(invTemp*(Delta_i - Delta_{i-1})} > runif(1)$, where "runif(1)" is a function to extract a random number from a uniform distribution (range 0 to 1).
- This way, if the new Delta (i. e. $Delta_i$) is smaller than the old Delta (i.e. $Delta_{i-1}$), that statement is always true, otherwise its validity depends on the difference among the deltas and the random number generated.
- If the statement is true, then the changes made to the "vectorPar" variable are kept; if the statement is false, or any of the previous "if clauses" have prevented the computation of the simulated ratio (and the following stage), then those changes are discarded. Then, another Monte Carlo step begins, using the new or the old set of Mass Action parameters.

Some other "if clauses" could be employed before calculating the ratio and the delta values. For instance, it could be improper comparing reporter values reached in different times; therefore, it should be checked that both simulations of the wild type and the mutant network lasted a comparable amount of time. Moreover, it could be checked that both networks are stable, i.e. they do not tend to accumulate tokens in some places because of the mutation of the network or the changes in the parameters. These clauses are not implemented in the script yet, but they will surely be added in the nearest future.

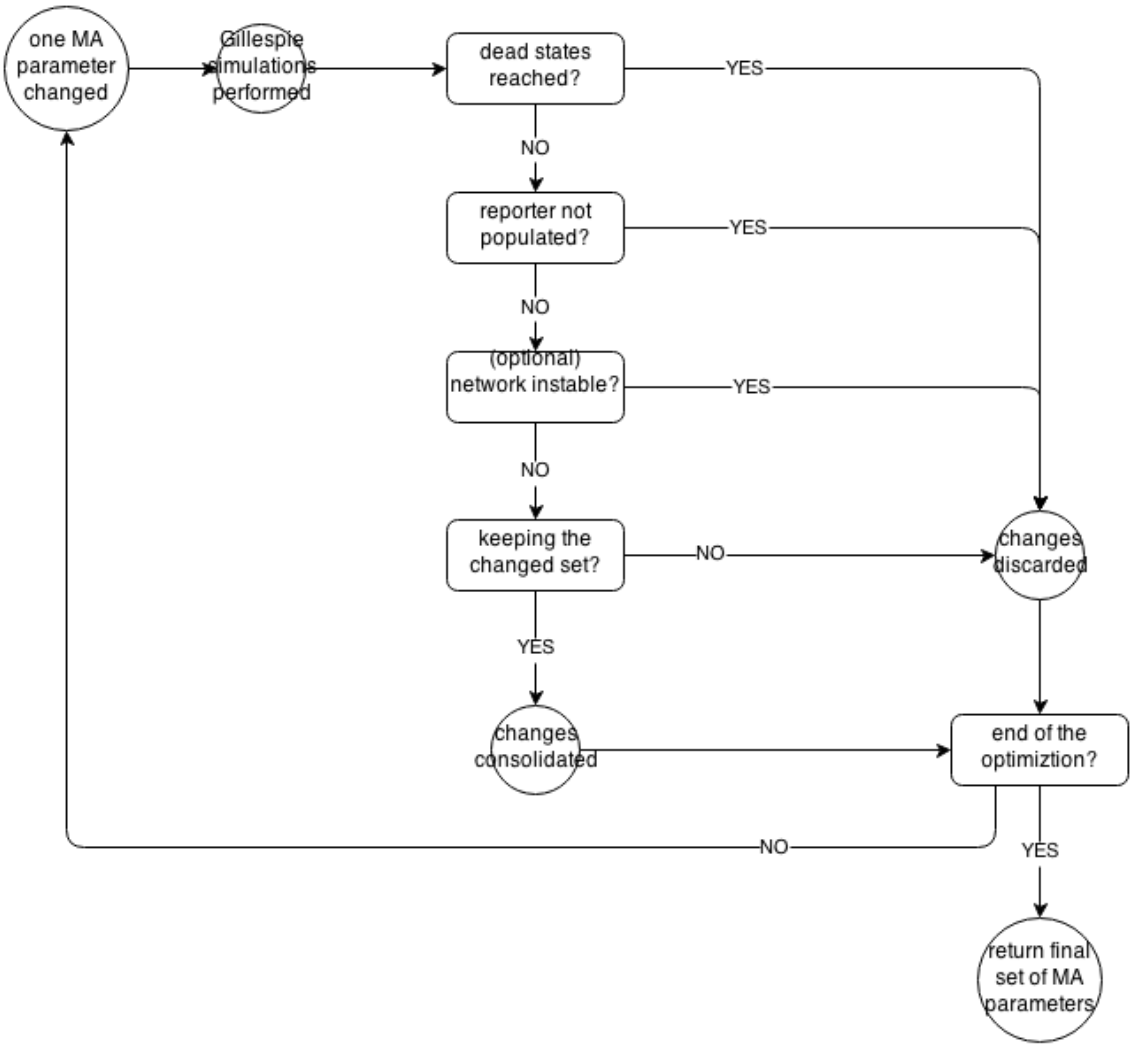


Image 3.7 Workflow of the whole optimization script (iteration excluded)

Possible extensions of the script

In order to train the network, two conditions (one mutant and one wild type) are not enough; rather many single mutations must be employed. It is pointless, however, to use them in series, i.e. optimizing the network using the ratio between reporters in a mutant and wild type and then repeating the optimization for each mutation: the final set of Mass Action parameters could not be applied successfully for the first mutation analysed.

Therefore, all the mutation must be employed in parallel, i.e. in the same time; this requires some changes in the script, first of all that a whole set of mutated vectors and matrices must be generated for each mutation. Then, one delta value (simulated ratio minus experimental ratio) is calculated for each mutation; the square values of all the deltas are summed together, as if a vector distance were calculated, thus generating the global new Delta. The formula is: $Delta_i = \sum_j^{mutant\ Nr} (Delta_{i,j})^2$

This new Delta ($Delta_i$) is finally compared to the global old Delta ($Delta_{i-1}$, i.e. the delta generated in the previous Monte Carlo step), and the algorithm proceeds normally.

Another detail must not be overlooked: some Mass Action parameters must not change. For instance, those "locked" parameters could be related to logic transitions, or they could be the parameters set to zero when "creating" mutant network (if that is the chosen method). Luckily, there is a quite simple method in order to accomplish that:

- First, user is prompted into writing the names of the transitions corresponding to the unchangeable parameters (parameters set to zero are locked by default); those names could also be written in the script, or they could be read from an external file. A possible implementation of the script could allow the automatic creation of this vector of parameters by reading the name of the transitions and "locking" all those parameters related to transition that have a special tag (e.g. "logic") in their name.
- Second, this vector of unchangeable parameters is removed from the vector containing all the mass action parameters; the resulting vector will provide the pool of parameters that can be chosen when selecting which transition should change.

The only side effect of using locked parameters is that it might happen that no transition is left free to change; in that case, the sole solution is loosening those blocks (if they do not apply to logic transition) or reducing the number of conditions (i.e. mutants) that are employed in the optimization process.

Last, as I mentioned before, the whole optimization process can be repeated for "totIteration" number of times; each time it is repeated, a new output file containing the last set of Mass Action parameters is produced. Repeating the optimization could be useful to verify whether the set of parameters generated at each iteration are identical or differ; in case they differed, confronting them could be useful to check how different they are and whether some key features are preserved. For instance, some parameters could maintain almost the same values in each set of parameters, thus indicating that these values are very important for the creation of an optimized set of parameters, i.e. their corresponding transitions are the main transitions in the network.

When multiple output files are produced, they must also be investigated in order to understand which one should be chosen as final set of Mass Action parameters of the network, and therefore employed in the following testing stage. The easiest way to perform this choice is simulating the network using all the available sets (i.e. output files): the set that generates the best results (simulated ratio nearest to experimental one) is chosen as the definitive one. In order to do so, simulation parameters should be changed so that they generate results that are more precise; e.g., the number of simulation steps could be increased (doubled?) compared to the number of those employed in the optimization stage, or the simulation could be iterated.

3.3. Network optimization

3.3.1. First experiment: growth rate as reporter

General explanation

As I have mentioned in the previous sections, at first I have tried to use growth rate as reporter in my network, even before defining all the lower layers and specifying the ER pathways. I have chosen growth rate because it is a well-known reporter, many literature being already available, and it is quite easy to put into practice, i.e. to experiment on it; moreover, it could be added to the network without compromising its architecture. Therefore, I have measured the growth rate of several mutant strains, whose mutations were not obviously related to the cell cycle (e.g. cyclins, CDKs, checkpoint proteins); however, after obtaining the experimental results I have decided to give up on this idea, partly because of problems in the modelling, partly because of problems in the experimental stage.

The main experimental issue was that the variability among the replicas I had performed convinced me that I could not rely on the data I had obtained. This point could have been overcome only by repeating the experiments many more times in order to yield reliable mean values but of course, I had no idea how many times should I have performed these experiments to reach the goal.
Moreover, experimental conditions, settings and criteria change in each growth experiment of yeast mutant strains that can be found in the literature, and some these parameters are not even known or available; therefore, it is difficult to tune the results together, i.e. using those data from the literature to drive and interpret the experimental data collected

The main issues concerned the modelling part, though. Since growth rate can be affected in many ways, and many possible mutations can be studied, I had selected those mutations yielding great changes in the growth rate phenotype. Therefore, since these mutations belong to many different pathways and are largely uncorrelated, it would have been essential modelling many pathways in detail and then training all them together. The problem is that this tremendous increase in the number of parameters would have yield to a proportional increase in the computational time required for the training stage.
Moreover, even if I could have managed to finish the training stage in an acceptable time, I could not have been sure that the final network would have been properly trained. In facts, increasing the number of parameters without increasing the number of conditions causes the increase of the number of possible solutions (i.e. sets of values that can be employed in the network to obtain the required growth rates). In other words, I would have obtained a network that behaved well in the selected conditions (the mutant strains I had employed in the training stage) but that would have had no predictive power at all.

The workflow

Despite the use of this reporter not being optimal, I have nevertheless decided to describe the steps I have performed, as some of them are common to the other experiments I have conducted.

First, I have chosen the mutant strains to be analysed, selecting those genes that are not involved in the cell cycle but whose mutation is viable and yields significant changes in the growth rate (Giaever et al., 2002, suppl. mat.),. Then, I have located these genes in the -80°C freezer containing the library of single-mutated MAT-A yeast strains, thus ignoring those genes whose correspondent strains were not available; the final list can be found in the supplementary materials (6.4.1).

Second, I have plated one single bead of each strain into non-selective solid complex medium (YPD+ agar) in order to create a stock of cells that could be employed in the following stages, without consuming any more beads. From this stock, I have retrieved the cells I have cultured overnight on liquid selective medium (YPD+G418); it must be noticed that I could grow them on this medium because all the library strains carry the correspondent resistance gene integrated in the genome (substituted to the mutated gene). All the strains have been cultivated together in a 96-well plate

Third, I have taken an aliquot of this liquid culture in order to replicate the plate into another 96-well plate. In order to do so, I have measured the OD of each well of the first culture, and then I have taken one specific aliquot for each well, so that the second culture had a starting OD equal to 0.2.
This second liquid culture has been employed for the data gathering step; therefore, it has grown overnight inside the cell-counter, which has kept the temperature at 31°C and has shaken the culture at 300 rounds per minute. This way I could measure the absorbance (i.e. the amount of cells) thorough the experiment. I have repeated this measurement two more times starting from the solid culture, thus creating biological replicas (the same strains are repeated but using different cells); it must be noticed that each 96-well plate already contained all the strains repeated twice, thus creating a technical replica (the same cells are repeated).

As I have explained before, the results were quite disappointing because both the lag phase duration and the growth rate in the exponential phase were remarkably different in the biological replicas (but not in the technical ones). One way to explain this is that some important factors, distinguishing cells belonging to the same strain, had been neglected and therefore they had to be considered in order to explain the results; since those factor are unknown, this would have proven to be impossible. The other explanation is that there was a great intrinsic variability, due to the interaction of many small non-important factors, and an undisclosed number of replicas should have been employed in order to obtain meaningful mean results.

3.3.2. Second experiment: GFP as reporter

General explanation

The second run of experiments has involved the measurement of the reporter values in different strains of yeast, bearing mutations in gene involved in the protein processing inside the endoplasmic reticulum. The reporter that has been employed in these experiments is the GFP (actually the Sapphire variant of the BFP), both as normal cytoplasmic variant and the fusion protein GFP-GPCR (G-protein coupled receptor), which is targeted to the ER by default.

Using these reporters and focusing on a smaller area has allowed me to avoid the modelling issues that had emerged in the previous run: the final model has a reasonable number of parameters, so that the network can be easily simulated and optimized. Moreover, since all the mutants belong to the same area (i.e. sub-network), it could have been possible to restrict the optimization process to that area only (by locking all the parameters of transition not related to the protein processing), thus speeding up the optimization process further more. Finally, even if it is impossible to determine whether a trained network will behave as planned, it is evident that the network trained using this reporter is more reliable than the network I would have obtained by using the growth rate as reporter.

Therefore, I have conducted two experiments using the two variants of the GFP; these experiments have the same starting point, i.e. the choice of the genes that need to be mutated. This list had already been retrieved when building the network, specifically when building the layers related to the protein processing inside the endoplasmic reticulum; therefore, I only had to check that our library contained all the selected strains, and eventually removing from the list all the missing ones. Of course, I could have bought the missing strains, but we have considered that there was no need, since we were not interested in those mutations in particular and we had enough strains to work on; the final list, which can be found in the supplementary materials (6.4.3), comprises 52 strains.
Then, as in the previous experiments, I have plated one single bead of each strain into non-selective solid YPD medium in order to create a stock of cells that could be employed in the following stages.

Then, I have plated *E. coli* cells carrying the plasmid I had to transfect into the yeast mutant strains.
In the case of the normal Sapphire protein, it has just meant retrieving the cells from the frozen liquid culture (i.e. without beads) in the -80°C freezer and plating them into a complex selective medium (LB plus ampicillin, that is the selection marked employed in the creation of these lines).
In the case of the fusion protein, it has meant cultivating *E. coli* cells, retrieving the plasmid from a -20°C freezer, transforming the cultivated cells with this plasmid and then cultivating in liquid medium and selecting the transfected *E. coli* cells. Generic explanation of this method can be found in the "Materials and Methods" section, whereas protocol can be found in the supplementary materials (6.3.2).

The workflow

Once I have obtained colonies of the yeast strains and *E. coli* cells carrying the plasmids of interest, I could start the transformation step. The whole process has been repeated twice, once for each plasmid, meaning that I have performed those experiments in series rather than in parallel; anyway, the protocol I have employed is almost the same in both cases. The detailed protocol can be found in the supplementary materials (6.3.1); here, I will enounce the workflow and some of its key points:

1) Yeast cells are cultured overnight in liquid medium (YPD + G418) in a 96-well plate. Unlike the previous experiments, in this case it is impossible to fit two replicas inside one plate, therefore, replicas are going to be created in a later stage; it also reduce the number of manipulation that must be done, thus quickening the protocol.
2) Plasmid is retrieved from the bacterial culture by lysing the cells and then purifying the mixture. In our lab, E.Z.N.A.® Plasmid Mini Kit II are employed; ; their name is due to the fact that they can be only used to deal with small amount of cells, thus retrieving small amount of plasmid DNA. They consist of reagents, buffers and purification columns that are needed in the whole process, from cell lysis to DNA precipitation and plasmid purification.
3) The solution containing the plasmid is analysed in order to measure the final concentration achieved; this measurement is performed by a device called "NanoDrop". It is a spectrophotometer that is designed to do instant measurement of tiny amounts of solutions (usually, 1µL of the target solution); user can set the wavelength (280nm for nucleic acids) and the solution volume, and the concentration of the selected species is printed out in the output.
4) The plate containing yeast cells is centrifuged in order to remove most of the culture medium and a mixture is added to make cell membrane more permeable to DNA; this mixture contains Lithium Acetate, DTT (dithiothreitol) and PEG (polyethylene glycol). Then a precise amount of plasmid is added in each well (that means, different volumes of the solution but the same amount of micrograms); it is accompanied by carrier DNA (e.g. single strand salmon sperm DNA), which is useful for preventing plasmid degradation by cytosolic DNases.
5) Cells are cultured for a short period (less than 1h) in a water bath, then they are plated into solid plates containing selective medium (YNB-SC); one petri capsule can only fit half of a 96 well plate, so two plates are needed. In my case, since all the strains are auxotroph that cannot produce histidine, uracil and leucine, and since the plasmid reverts the auxotrophy for the latter amino acid, the selective medium must contain all the amino acids except leucine. At this stage, more replicas can be created; this is useful to increase the likelihood that a certain transformed strains grows and forms visible colonies.
6) All the plates are put into an incubator set to 31°C for a couple of nights to allow all the strains to grow; in facts, it must be noticed that the transformation process stresses the cells, increasing the duration of the lag phase, and the mere presence of the plasmid slows down the growth rate.

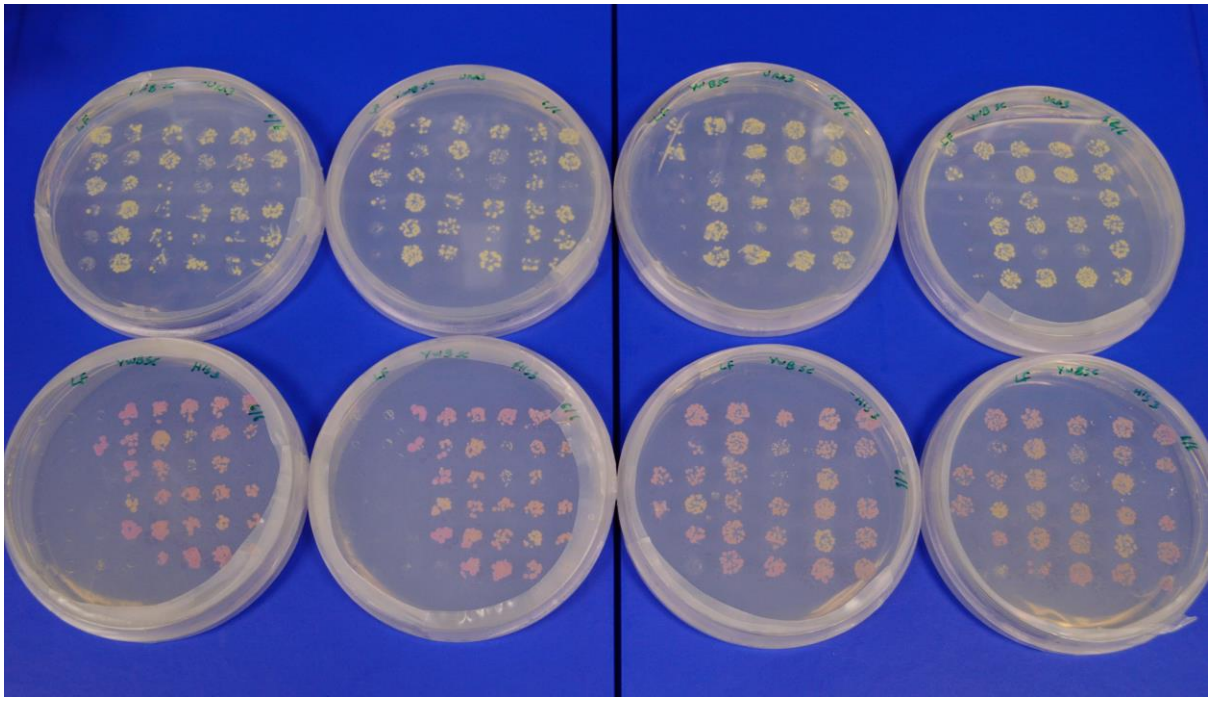


Image 3.8 Some of the yeast strains cultivated and transfected with the plasmid containing the GFP gene

Final step

Once I have obtained the culture of all the mutant yeast strains, I could start the main part of the experiment, i.e. measuring the fluorescence and the absorbance of each strain growing in a liquid culture. In order to do that, first I had plate the strains in liquid medium (YNB-SC –Leu in a 96-well plate), by picking one colony from each strain; then, this culture has grown overnight and an aliquot from each well has been transferred into a new 96-well plate containing the same medium, thus yielding a dilution factor of 1:20.

This plate has been employed in the measuring phase: it has been placed in the spectrophotometer and about 200 cycles has been automatically performed over a period of two days; each cycle is composed of shaking, absorbance measurement and fluorescence measurement, and it is performed keeping temperature constant (30°C). It must be noticed that the 96-well plates I have used in the machine are black, only the bottom being transparent; this is required for an exact fluorescence measurements, in order to avoid that fluorescence from other wells could be recorded as coming from the well that is being measured.

The entire measuring phase has been repeated two times for each reporter (i.e. cytoplasmic and ER-targeted GFP); each time a new 96-well plate has been employed, using the same methods and the same volume of medium and transferred aliquots. Therefore, the whole measuring stage has required 2(reporters)*3(replicas)*2(days) = 12 days to be completed; the data, analyses and results are illustrated in the "results" subsection, whereas their application to the network optimization can be found in the "network optimization" section.

Other considerations

As I have mentioned in the introduction, at first I had employed Venus YFP as cytoplasmic reporter and the transformed yeast cells were selected on a –Ura medium rather than on a –Leu medium. Then, since the fusion protein had been obtained using the Sapphire variant, for homogeneity reasons I decided to use that variant as cytoplasmic reporter too; therefore, I had to repeat this experiment using Sapphire variant and an YNB-SC –Leu medium.

Moreover, I had some difficulties when applying the protocol to the second transformation, namely that one performed using the plasmid that carried the gene codifying for the fusion protein.

The first problem was due to the fact that the plasmid had been mistakenly annotated, so that it seemed that the selective marker was URA3 rather than LEU2: the obvious result is that the first attempt of creating transformed strains has miserably failed because I had employed the wrong medium (YNB-SC –Ura). Therefore, I had to repeat the transformation protocol and then plating the transformed cells using the right medium, i.e. YNB-SC –Leu.

The second problem is that the growth rate of the cells transformed with this plasmid is very low, so that cells had to be cultivated for more than three days before showing typical yeast colonies; even so, some strains have not grown at all, therefore I had to remove them from the list of employed strains.

The last consideration that must be done before analysing the results concerns the localization of the fusion protein GFP-GPCR, because fluorescence levels might not suffice in explaining the phenotype. For instance, let us imagine a strain that is mutant for a protein involved in the translocation of the nascent protein inside the ER. In that strain the folding of the fusion protein could still happen and the fluorescence levels could be equal to those measured in a wild type strain, but the localization of the native protein would surely differ. In other word, microscopic visualization is needed to gain a more complete understanding of the data.

I have employed a microscope to look at the cell (bright field) and to localize the fluorescence, trying to determine whether fluorescence came from the membrane or the cytoplasm. Some of its features are:

- It mounts a set of oil immersion lenses, whose highest magnification power is 100X
- It can be used both in bright field and in fluorescence mode because it is associated to a laser
- it has many filters for choosing the wavelength observed; roughly speaking, they correspond to the emission wavelength of the most common fluorescent proteins (e.g. GFP, mCherry, YFP)
- The image can be seen through the oculars or can be recorded by a camera, and therefore seen and stored using a computer.
- The camera software also allows for modifying image capture parameters (e.g. exposition time), thus allowing for identification of weak and otherwise invisible fluorescence signals.

3.3.3. Second experiment: data

Cytoplasmic-processed GFP

A first analysis of the results can be made using the software of the cell-counting machine. For instance, by observing absorbance over time it is possible to determine which strains have grown and which strains have not; it is also possible to verify that non-charged wells are truly empty, and using the absorbance measured in those wells as blank measure for all the other measurements.

On the other hand, analysing absorbance data on R environment offers more information; for instance, it is possible to fit a growth curve on the data recorded in each well, and then studying the dispersion of data (i.e. the deviation between estimated and measured values). The "grofit" package (Kahm and Hasenbrink, 2010) implements many solutions, i.e. it allows employing many kinds of growth curves and choosing the curve that fits best.

By these analyses, I have discarded those mutant strains that had not grown in both experimental replicas, i.e. mutant that had not reached the stationary phase, or those strains whose distribution fitted very poorly on any growth curve.

Analysing the fluorescence intensity values, some other consideration could be made, the easiest being which strain had the highest intensity and therefore produced the highest levels of GFP. One informative data comes from the analysis of fluorescence over time: as it can be seen in the graphs and data produced, some strains reached the highest levels of fluorescence at the end of the exponential phase, whereas other reached this level in the mid of the exponential phase. These data can be interpreted in many different ways, and should be subject to further analyses; in the meantime, though, I have decided to employ all the strains whose behaviour was confirmed, observed in both replicas.

After these preliminary considerations, I could compare absorbance and fluorescence levels among the non-discarded strains; for each strain, I have calculated the fluorescence intensity in the middle of the exponential growth phase (F_{50}), and then I have employed that value for further comparisons.

In particular, by dividing the value calculated in mutant strains for the value calculated in the wild type strain, I have obtained the experimental ratios that I needed for the optimization phase. If the ratio of a certain mutant strain could be measured in both the replicas, I have employed the mean among those ratios as the final ratio for that strain; F_{50} values can be found in the supplementary materials (6.4.3).

At the end of these analyses, I am confident that the cytoplasmic-processed GFP can be employed as reporter because meaningful levels of fluorescence can be read in almost any strain, and because its expression does not impair cell growth. This reporter could seem too far from the processes we want to observe and evaluate (i.e. protein processing in the ER), but the differences on F_{50} levels state otherwise: cytoplasmic processing of proteins is indirectly affected by mutations in genes expressing ER-related proteins. Therefore, this reporter can be used in the network optimization.

ER-processed GFP

The same consideration applies on this case too.

First, I have employed both the software of the cell-counting machine and the "grofit" package in R to evaluate the absorbance data of each strains; almost all the strains have grown normally, reaching the stationary phase before the end of the measurement process.

Second, I have analysed the fluorescence intensity data, which has proven to be much more complicated. In facts, it seems that the levels of fluorescence of almost all the strains (in any replica) were lower than the blank measure obtained from the empty wells. It is not due to contamination of the blank wells, because absorbance measurements in those wells indicate that nothing is growing in there; therefore, it must be due to very low levels of fluorescence in almost any wells.

Another possible explanation could be that all the wells were contaminated, so that bacteria have grown instead of transformed yeast; however, microscopy observations of some aliquots from these wells have shown that no significant bacteria contaminants were present in those wells.

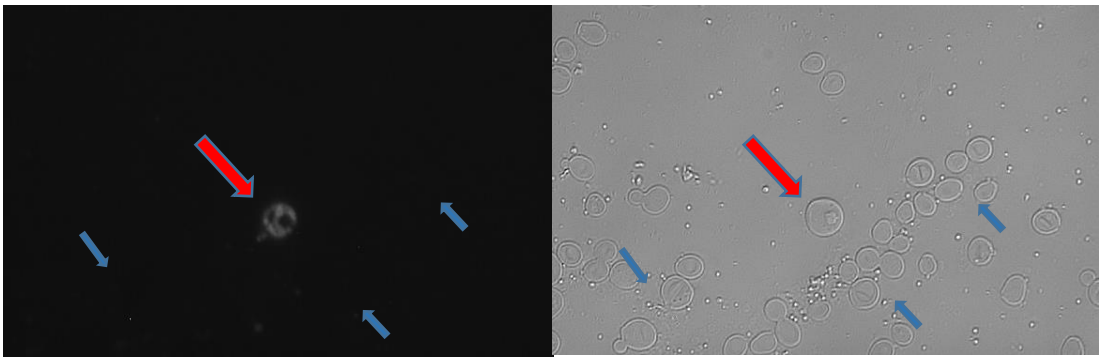


Image 3.9 Images of the transformed yeast: very few cells (red arrow) have measurable fluorescence levels.

It could also be argued that the medium was not selective, thus allowing for the growth of any yeast and impeding the selection and growth of transformed yeast; however, wild type yeast cells plated in the same medium have not grown, thus confirming that the plate wells really contained transformed yeasts.

Finally, it could be hypothesized that the fluorescent protein employed was not the Sapphire variant as it was supposed to be, but it is impossible since the plasmid has been sequenced to verify its content before employing it.

Therefore, the remaining options are that the fusion protein did not work as planned and failed in obtaining its correct folding, or cells required much more time to express this construct and show appreciable levels of fluorescence; in any case, it would have been much more complicated obtaining meaningful and usable results.

In conclusion, ER-targeted GFP would have been a very good reporter for the optimization of the network, being very near to the processes directly affected by the mutants that I am using in the network. In practice, though, it was not essential and its level have proved too difficult to obtain experimentally; therefore I have optimized the network using literature data and fluorescence levels of normal GFP.

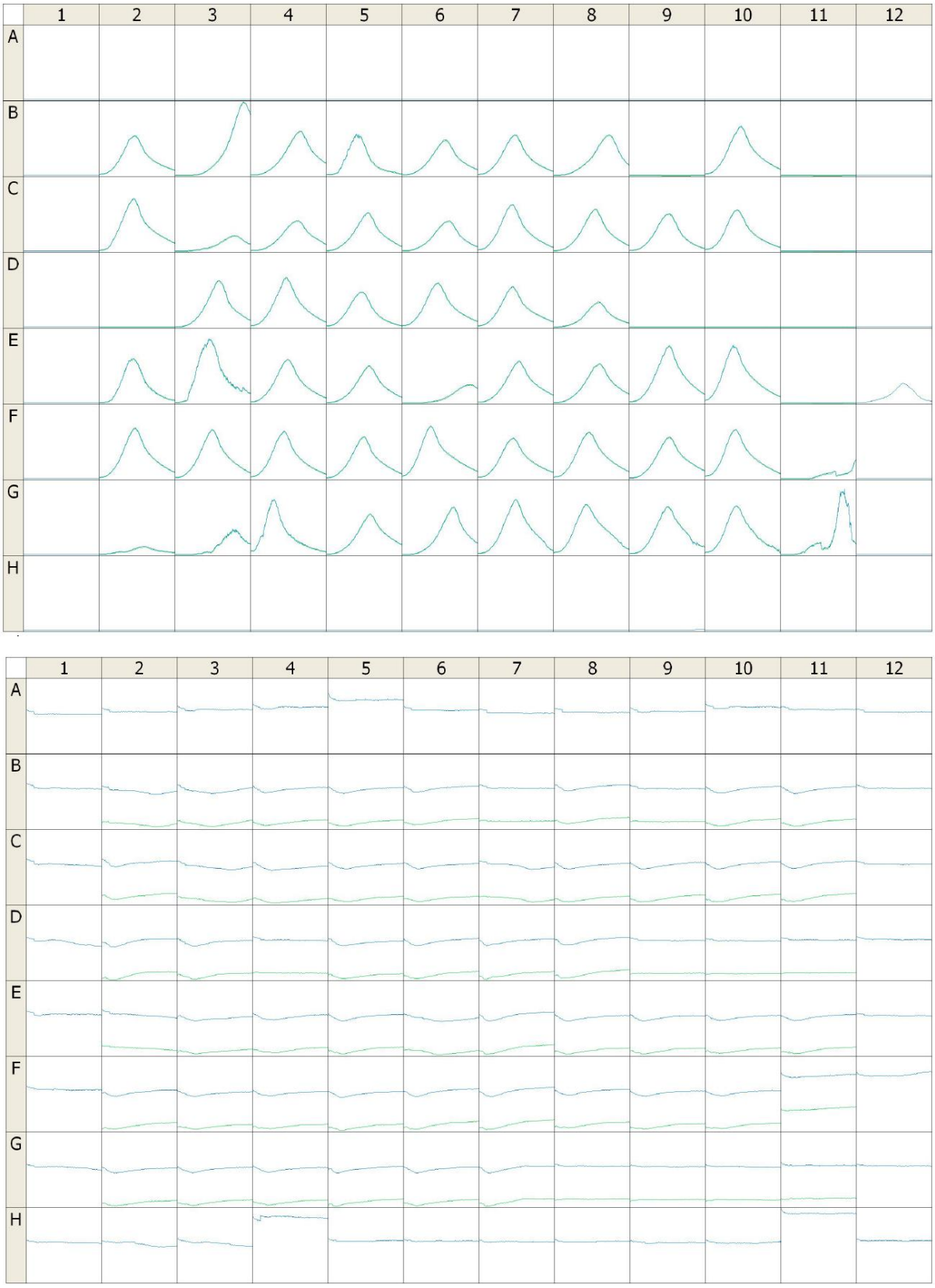


Image 3.10 Graphs of the fluorescence intensity of each well (i.e. each strain) over the duration of the experiment: fluorescence of the cytoplasmic GFP (upper image) and the ER-processed GFP (lower image). Absolute values are depicted in blue, blanked values are depicted in green; external wells only contain blue lines because they are the blank wells. In the lower image, it can be noticed that the all wells (both empty and full ones) have the same level, i.e. zero fluorescence.

3.3.4. Testing the optimization script

Initialization

Before optimizing the network, it is necessary to test the script itself:

- To verify that the script works as planned, i.e. it tends to produce sets of the Mass Action parameter that reduce the difference between the experimental and the simulated ratios of certain reporters in set conditions, while keeping the original features of the network.
- To confront different k parameters obtained iterating the optimization multiple times (starting from the same set of parameters), evaluating how much they differ and whether the amount of these differences is related to the accuracy of the optimization process itself (i.e. whether these differences tend to diminish when the difference between the ratios tend to zero).
- To study the behaviour of the script when using different initial markings. In facts, it is positive that markings influence the sequence of transitions that are chosen to fire, but it is not clear whether they would determine which set of MA parameters is obtained at the end of the optimization.

This testing could be performed directly on the whole-cell network, but it is very large and the optimization process takes much time; therefore, I have decided to employ a smaller network, being confident that the findings could be extended to any conditions of usage of the script. I have therefore employed the following workflow:

1) I have retrieved a Petri Net from the public database of esyN.org, choosing a model representing the role of TDP43 in healthy condition; I have slightly modified the architecture of the network so that it fitted better with my purpose. The network can be found at http://www.esyn.org/builder.php?publishedid=198&type=PetriNet
2) I have then chosen the reporter and the mutant places in this network, and I have simulated the behaviour of the wild type and mutated network using its original markings and MA parameters. The ratios between the tokens in the reporter place in each mutant and in the wild type represent the "experimental" set that I have employed in the following stages.
3) I have created the first initial marking by assigning ten tokens to each place. In addition, I have created one hundred initial markings by randomly choosing the abundance of each place. For all those initial markings, I have set to one all the MA parameters that need to be modified during the optimization.
4) I have run the optimisation script once for each randomly generated initial marking, and then one hundred times using two randomly generated markings. Each run has generated a different set of MA parameters and a correspondent final delta value i.e., in this case, the difference between the ratios calculated using the native and the mutated set of MA parameters.

Testing and results

As expected, since the optimization is performed using a stochastic approach, many different results are obtained from each iteration of the optimization script, whether the same or different initial markings are employed; the results differ both in the set of MA parameters created and in their final delta value. Some analyses of the results have been performed; explanations, example graphs and considerations can be found in the text below, whereas all the plots can be found in the supplementary materials (6.5).

First, by plotting the frequency of occurrence of the starting delta values (obtained at the beginning of the optimization stage), it can be observed that they are approximately normally distributed. On the other hand, by plotting the frequency of occurrence of the final delta values (obtained at the end of the optimization stage), it can be observed that values near to zero are the most populated.
These findings can be observed using delta values obtained both by employing different markings and by iterating the script using the same marking; therefore, it is proved that the script does minimize effectively the final delta value.

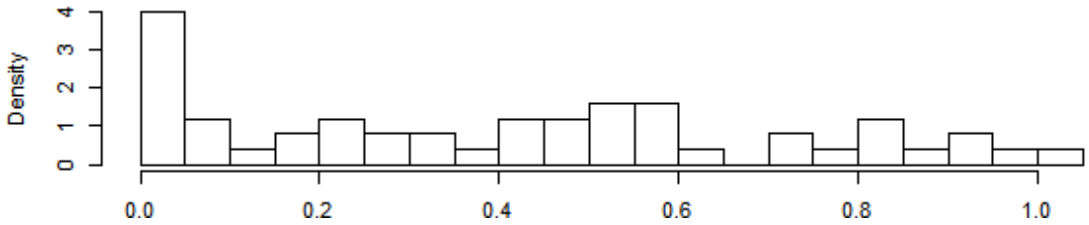


Image 3.11 Histogram representing the final delta values obtained by using 100 different starting markings.

Second, each random marking can be compared to the mean of all the random markings, thus calculating a "distance" from that mean marking. By plotting these distances and their corresponding final delta values, it can be observed that starting markings do not influence the final delta values obtained, i.e. they all have the same chance to achieve a low delta value, i.e. yielding a better MA parameters set.

Third, it can be observed that all the final MA parameters sets differ from each other and from the native one. For each set, a "distance" from the original set can be calculated and plotted against their related final delta values; this way, it can be noticed that smaller deltas associate with smaller distances. This is true for sets obtained both from random markings and from iterations of the same optimization; therefore, it is proved that MA parameters sets tend to the native one when their delta tends to zero.

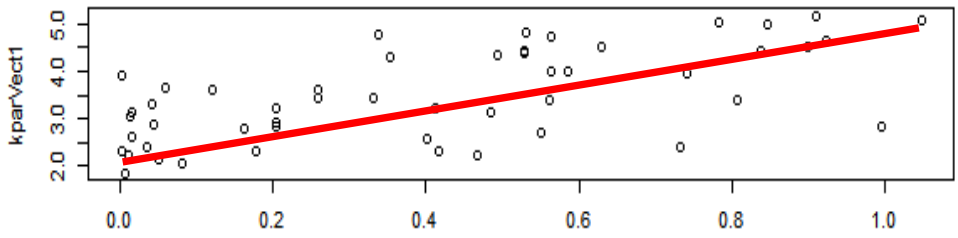


Image 3.12 Graph representing the distances between final MA parameters sets and the native set; distances have been plotted against their related delta values.

Fourth, heat maps can be drawn by comparing the final MA parameters sets, indexed according their corresponding delta values. Those maps are useful to highlight whether some parameters are more important than others are in determining a final good or bad delta value; their functioning is explained in the supplementary materials (6.5).

By observing the values of the highlighted parameters, it can be appreciated that they tend to the values they had in the native MA parameters set when final delta values tend to zero; on the other hand, this finding does not hold true for all the others parameters.

In these optimizations performed, three parameters were more important than all the others; by observing the Petri Net, it could be observed that they do not all belong to transitions directly linked to the reporter place. Therefore, it can be stated that the heat maps highlight the most important parameters, i.e. those parameters that are mainly responsible for the final delta values obtained, and that this information could not be easily retrieved otherwise.

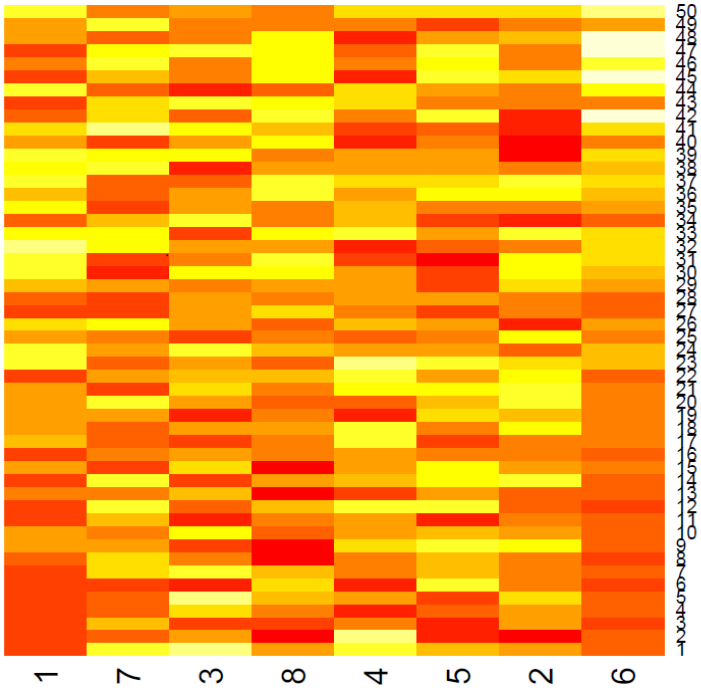


Image 3.12 Heat map created by comparing the values of each parameter among all the MA parameters sets; sets obtained iterating the optimization 50 times, employing one starting markings.

Fifth, it could be argued that the values of the most important parameters are determined by the random starting markings employed only when associated to bad (high) final delta values. In that case, it could be stated that the starting markings have a major impact in determining the final MA parameter set only if the set is obtained from an incomplete optimization.

Finally, it must be noticed that one or more mutated sets could be different from the native one and yet it could yield a final "delta" of 0; in this case, it would simply mean that all those sets are solutions of the network, i.e. the same result can be reached using any of them. However, when increasing the number of conditions or the number of reporters employed, it should become progressively more difficult to obtain multiple solutions.

3.3.5. Training and testing the network

Initialization

Once I have proved that the optimization script could be applied to my network, I could start the proper optimization phase. It has required some preliminary steps:

1) Obtaining the set of mutations that must employed in the script; the optimization is performed twice using two different reporters, each one needing a specific set of conditions (i.e. mutations) in which it has been measured and it must be simulated.

   Experimental data have supplied the set of mutants to be used with the cytoplasmic-processed GFP reporter, whereas literature data (a subset of the list found in the supplementary materials of Jonikas et al., 2009) have supplied the set of mutants accompanying the UPR reporter; this set can be found in the supplementary materials (6.4.4).

   Both sets have been randomly split into two subsets each, one for the training stage and another one for the following testing stage.

2) Setting MA parameters values. As I have mentioned in the previous sections, almost all the parameters have been set to one; only logic transitions have MA parameter set to 10000, i.e. a number that is sufficiently high so that they can be considered as immediate transitions.

   The script has been modified so that it automatically recognize logical transitions and add their corresponding parameters to the list of those that cannot not be changed; it also recognize transitions belonging to the same logical layer, which should have the same MA parameter values, so that their parameters changes together whenever one of them is chosen.

3) Finally, an external list is loaded, which contains the parameters of all the transitions not directly involved in the protein processing in the endoplasmic reticulum (and subsequent processes); this way, the number of changing parameters is decreased, and it should be easier to reach good values in the optimization process. Moreover, keeping external parameters unchanged allows for creating a more flexible network, i.e. it should be relatively easy to add additional detailed processes and optimize only those new parts of the network

4) I have finally made some changes in the script to add those optional checks described before (e.g. checking that tokens do not accumulate too much). I have also decided to modify the way reporter values are considered, preferring to employ their absolute values rather than the ratio between their values and the duration of the simulation; that was made to reduce the variability, considering that the absolute values oscillate much less than the aforementioned ratio

   Other minor adjustments consisted in changing the network eliminating the "External metabolites" place, thus introducing a source transition instead; this change does not affect the optimization stage, rather it prevents some problems that could have arisen when simulating the network. I have also changed some names of the transitions (e.g. logic and coupled ones) so that the script could recognize them automatically.

Running the process

Once the script and the network were complete, I have trained the network using those two subsets of mutants I had generated, in two different optimization processes.

I have chosen a number of optimization steps that could yield a mean of ten changes in each of the considered transitions, and a number of simulation steps (for each optimization step) that could yield a mean of ten firing events for each transition of the network. Therefore:

- For the UPR-related set: $mcSteps = 300,\ stepsNumber = 5000$
- For the cytoplasmic GFP-related set: $mcSteps = 500,\ stepsNumber = 5000$

I have run each process four times, changing the "inverse temperature", that influences how likely a configuration that increases the energy of the system is accepted rather than rejected: too restrictive temperatures may prevent the system from reaching its energy minimum, whereas too relaxed temperature may prevent the system from remaining in its energy minimum once it is reached. I have also employed a simulated annealing approach, i.e. is I have changed those temperatures during the optimization process so that they were progressively more restrictive.

For each set of mutants, I have obtained four sets of MA parameters and I have chosen the set that yielded the smallest "delta" value during the optimization process, thus completing the **training** stage.

Then, I could start the **testing** stage; I have followed the same protocol for both the sets of mutants, therefore I will describe just one of them:

1) I have simulated the network using the testing subset of mutants, and employing the starting set of MA parameters. I have employed the same formulas from the optimization script to calculate the ratios of the reporter values in the mutant and wild type conditions, thus obtaining a $\Delta_{starting}$
2) I have simulated the network using the testing subset of mutants, and employing the optimized set of MA parameters; using the same formulas described before, I have obtained a $\Delta_{optimized}$
3) I have simulated the network using the testing subset of mutants, and employing N randomly generated sets of MA parameters; this way, I have obtained a N $\Delta_{random,i}$

It must be noticed that all those simulations have been performed using the same number of steps employed during the optimization stage, but they have been iterated at least ten times in order to obtain mean values of the final deltas.

The testing stage confirms that the final delta obtained using any optimized set of MA parameters is smaller than all the other deltas calculated (from randomly generated set and from the starting set); therefore, it can be stated that optimization was successful using both experimental and literature data. The set of MA parameters yielding the lowest delta has been employed to update the parameters of the whole-cell network on the eysN.org website.

Further possibilities

There are more analyses that could be performed if optimization were repeated several times. Unfortunately, due to the complexity and the size of the network, each run of the simulation requires several days to be completed on a standard hardware (2GHz, 4GB of RAM), and repeating the script many time would require months of computations or the use of a supercomputer. Therefore, here I will simply describe some of the analyses that could be performed, without actually performing them.

First, the whole training and testing could be repeated using a greater number of simulation and optimization steps; this way, it could be possible to obtain a final set of MA parameters that would generate an even smaller delta between the simulated and the experimental results. Moreover, the optimization could be repeated starting from different markings of the network, all preserving the desired features of the network; this way it could be possible to confirm the finding that markings do not influence the final set, if an adequate number of steps is employed during the optimization.

Second, it might be argued that optimization would yield a better set of MA parameters if a lower number of parameters were changed during the process, i.e. if only the key parameters were changed while leaving the others unaffected. In order to test this idea, this workflow should be employed:

1) Repeating optimization many times, each time preventing changes on one value from the set of changeable MA parameters.
2) If a better set were created, then the transition corresponding to the blocked parameters could be added to the set of unchangeable transitions; the first point could be repeated again preventing changes on one more MA parameter
3) If no better set were created, then the set of changeable MA parameters represent the minimum pool required for optimizing the network.

It this idea were confirmed, it could also be useful to analyse which are the transitions whose parameters need to change, because it is highly likely that they represent the most important transitions of the network, i.e. those that happen more often or somehow affect more deeply the outcome of a simulation.

It could also be possible to demonstrate this idea using a very different approach:

1. By repeating the optimization many times, it should be possible to obtain many different solutions, i.e. sets of parameters that minimize the delta values. By confronting those sets together, it could be possible to find which parameters are similar and which change greatly among the sets: parameters with similar values are likely more important than parameters that can assume any value
2. Optimization could be then repeated by changing only the most important parameters, while setting the others to one: if the final delta were smaller than all the others measured changing the whole set of parameters, then the tested idea would be confirmed.

Third, by confronting the final sets of parameters obtained using the sets of ratios from the literature and the experiments, it can be observed that they are very different, and therefore they are not interchangeable: they are different solutions of different problems.

It order to obtain a final set that could be employed for both conditions (the experimental and the literature sets of ratios), the only possible approach is optimizing the network using both conditions simultaneously. In other words, that means:

1) Finding a set of mutants whose ratios are available in both conditions; this set must be divided into a training and a testing subsets.
2) Optimizing the network using a slightly different formula, that considers both the conditions at the same time: $Delta_i = \sum_j \sum_k^{mutant\ Nr} (Delta_{i,j,k})^2$, where *"i"* is the current optimization step, *"j"* is the condition (experimental vs literature ratios) and *"k"* is the mutant.

The resulting set of MA parameters might perform worse than the starting sets when applied to a single condition, because of the reduced number of mutants employed in the optimization; on the other hand, its overall performance (on both conditions) should be much better than the starting sets obtained using a single condition. If this hypothesis were proved, it might be useful deciding to implement many more reporters at the same time, in order to have an accurate representation of the network in more conditions.

4. Conclusions

In conclusions, the outcome of My Master's thesis work consists of:

1) A whole-cell Petri Net model of yeast *S. cerevisiae* that is available to the public, and therefore it can (and will be) extended in the future
2) Two scripts for the simulation and analysis of Petri Nets models downloaded from esyN.org; as such, they are an integral part of the first release of the esyN.org website, which resulted in a paper that has been accepted for publication by PLoS One (Bean et al., 2014).
3) A script that allows the optimization of the MA parameters using experimental data.
4) Predictive power obtained optimizing the abovementioned model, which validates the approach employed and can be applied for driving experimental design.

In this section, I will briefly summarize the main features and future perspective, of each of these points.

4.1. Whole-cell yeast network

4.1.1. The network

I have built a multi-level whole-cell yeast network: the upper levels contain a coarse-grained representation of all the cellular processes, whereas the lower levels contain a more detailed representation of translation-related processes (e.g. translation, folding, post-translational modifications, targeting to organelles or to the membrane). In particular, protein processing inside the ER is described at the highest level of details, using real genes and metabolic reactions, whereas all the other layers contain generic ("coarse") places and transitions. Three main features of this network are:

1) It is written using the Petri Net formalism, therefore it can be written as matrices and its behaviour can be simulated over time combining Petri Net firing rules and other algorithms (e.g. Gillespie algorithm)
2) Some of its parameters are set in order to yield a final network showing the desired features:
    a. Uniformity in the distribution of tokens, i.e. proteins do not accumulate
    b. Accessibility of all the transitions, i.e. all the reactions that are really occurring in a cell must be able to occur in the network too;
    c. Avoidance of self-sustaining states, i.e. the network/cell must die when running out of external metabolites.
3) Some other parameters are esteemed, therefore I had to optimize them using a "training and testing" approach and implementing a Monte Carlo method; this approach has required some experimental data to be used for comparison with the simulated values during the optimization.

It must be noticed that optimization generates predictive power in the network, meaning that the results obtained simulating the network in different conditions could be employed as expected results of a real experiment conducted in the same conditions.

4.1.2. Future expansions

Another key feature of this network is that it can be extended quite easily: in some cases, processes are already in the network and they just need to be described in details, whereas in some other cases the upper layers can supply a framework for easily adding new detailed processes.

Expanding the network would be useful to create a complete yeast network that can be employed to visualize reactions and physical interaction among molecular species. It would be a Petri Net version of networks already available to the public, but it could be particularly useful because Petri Net require explicitly indicating and showing how molecular species interact, and because they allows for describing different states of the same species (e.g. phosphorylated/dephosphorylated, folded/unfolded, inside/outside a compartment).
An expanded network could be optimized yielding a better (more accurate) set of parameters, even if the computational time required for its optimization would increase. Moreover, expanding the network would increase the pathways and processes that can be studied, and the conditions whose results can be predicted by simulating the network.

4.1.3. Web repositories

The whole-cell network has been built in the esyN.org; it is a website recently developed in our lab, which implements the cytoscape.js tool, thus allowing for building Petri Net and standard networks.
Its main feature, though, is that users can both save and export their networks offline in several formats and they can save them online in the esyN.org database. The networks can be saved as private project, so that only the author and other collaborators (invited by him) can see and modify the project, or they can be saved as public project, so that everybody can see, use, copy and modify them.

Therefore, esyN.org works as database of Petri Networks but it also allows multiple users to share networks, cooperate in the creation of a project, and to publish the final drawing.
The whole-cell network I have created is still a private project, but it will be made public soon; this way, the expansion process of that network could be performed not only by me but also by many other users, independently working on different layers at the same time. That would dramatically reduce the time required to build the network, even if it would require some additional work to tune together those different parts, which are likely going to be written using different "styles" (e.g. names assigned to the nodes).

It should also be possible to use eyeast.org to access this unfinished network and all the pieces that will compose the whole picture: this website, currently under construction, should work as a showcase of all yeast network stored in the esyN database.

4.2. Scripts for Petri Nets

4.2.1. Simulation script

I have created a script that allows simulating Petri Nets by implementing the Gillespie algorithm. Unlike some other programs already available, this script allows for changing almost all the simulation parameter, thus yielding a great flexibility; moreover, it is written in R, therefore it is completely verifiable and customizable by the user. I have also created some other variants implementing less sophisticated algorithms, in case a simpler but quicker script was required.

This script accepts as inputs file produced both by Snoopy and by the esyN.org web tool. Indeed, this tool accompanies the website, meaning that all the users of esyN.org can download the script from the GitHub public repository at https://github.com/esyN/esyN-simulation and run it on their computers. I am still working on this script in order to make it faster and more stable, and the copy on the repository is updated frequently to reflect these changes.

The simulation script goes with an analysis script that allows transforming each irregular time series (representing the simulation output) into a regular time series, so that it can be compared with other time series and a mean time series can be calculated. The analysis script also allows plotting the mean time series in order to have an immediate, graphical representation of the behaviour of the network (or some places within it) during the simulation steps: the amount of tokens is on the y-axis, whereas the time of the simulation (as calculated in the Gillespie algorithm) is on the x-axis.
It must be noticed that meaning the time series is just one of the possible analyses that can be performed; indeed, the simulation and analysis scripts are divided so that any user can write its own analysis script and replace mine.

At first, some "qualitative" tests have been performed using bigger networks, i.e. verifying that the general simulated behaviour of the model was similar to the expected one: e.g. tokens accumulating in the right place, network reaching a dead state after few steps and so on.
Then, the simulation and the analysis scripts have been "quantitatively" tested using some Petri Nets found in the literature, which had already been built and simulated (Blatke, 2011, pp. 48-52):

1) The networks have been built on esyN.org, setting the appropriate tokens and MA parameters; then, they have been simulated using the same parameters found in the literature.
2) The final outputs (i.e. graphs) have been confronted with the results already available, thus showing that both the simulation and the analyses had worked as planned. This way I could compare not just trends, but also quantitative values.

4.2.2. Optimization script

Then, I have created another script that allows optimizing Petri Nets by implementing a Monte Carlo method. This script too allows for changing almost all the parameters and, being written in R, it is highly customizable; this is particularly interesting because almost each network has different requirements, different conditions that must be checked during the optimization process and, therefore, that must be added to the optimization script.
At the moment, two versions exist: one can (must) be applied for optimizing the whole-cell network I have created; the other one is a simpler version without any "if condition" checks, so that it can be employed as the starting point for building customized optimization scripts. These versions are not available to the public yet, but it is planned that they will be made public in the near future.

This script employs the following workflow:

- Reading the input files (the same as the simulation script described before)
- Creating mutant networks by modifying parameters of transitions linked to the mutated places
- Randomly changing a MA parameter; rules can be implemented to set which parameters should not be changed in this step
- Simulating the wild type and the mutant networks, thus obtaining the ratio of reporter values in mutant and wild type conditions; simulation is performed using the abovementioned script.
- Comparing the simulated and the experimental ratio, thus obtaining the "energy" of the system
- Comparing the new and old values of the system "energy", and accepting the changed parameter if the energy decreases.

As it has been demonstrated by testing the network, the number of simulation steps and the number of optimization steps greatly influence the outcome of the optimization script, especially its ability to produce a final set of MA parameters that gives a final "energy" very near to 0. It has also been demonstrated that, when employing a numbers of steps that allows the minimization to be completed, the starting markings of the network do not influence the outcome of the optimization process.

This script is computationally heavy, because it repeats several times another script that perform some calculi several times. Pruning the network, i.e. removing useless nodes, could help in speeding up the script: it reduces the size of the matrices, and therefore the time required for the calculi, and it reduces the number of steps (i.e. calculi) that are required to simulate the behaviour of the network.
Moreover, the number of required Monte Carlo steps is reduced when using a smaller number of parameters, thus speeding up the process. It is also useful for generating a more versatile network: if the model were extended in other regions using the appropriate parameters, the parameters obtained optimizing the starting region should continue behaving correctly.

4.3. Predictive power

4.3.1. Possible uses

The network is optimized so that the ratio of the values of a certain reporter place, obtained simulating the wild type and the mutated network, should be equal to the ratio that could be measured experimentally. It should be possible to extend this feature to other conditions, whether they were experimentally known or not: these extensions generate the predictive power, i.e. the ability to predict the outcome of an experiment in a semi-quantitative fashion.
This predictive power could be employed to perform a preliminary check of experimental results, or for suggesting which range of values should be expected when performing an experiment. It could also be applied to perform *in silico* experiments and to drive the experimental design, e.g. by determining which experiments would be more informative for proving/rejecting a hypothesis.

The current network can be employed to predict the behaviour of yeast strains harbouring mutations in the genes involved in the protein processing (especially in the processing that occurs inside the endoplasmic reticulum), the Unfolded Protein Response or the ERAD. These mutant genes could already be in the network or they could be added if needed, although it might require repeating the whole optimization phase.
The predictive power obtained so far could also be employed for simulating the behaviour of the network when overexpressing a yeast gene or when expressing heterologous genes. For instance, it could be employed to simulate what happens in a yeast cell when an unstable or aggregation-prone protein (e.g. Tau protein from Alzheimer's disease) is expressed, e.g. by measuring the activation of the UPR or the production/degradation rates of cytoplasmic and ER-targeted proteins.

It must be noticed that an optimized set of MA parameters with a very low energy is not necessarily a good set for predicting the outcome of the network; in facts, a certain set might just be a good solution of the optimization problems without being usable for performing good predictions. Therefore, the testing stage is essential for verifying that the optimized set of parameters shows predictive power, and even in that case the predictions obtained must be considered and used with caution.

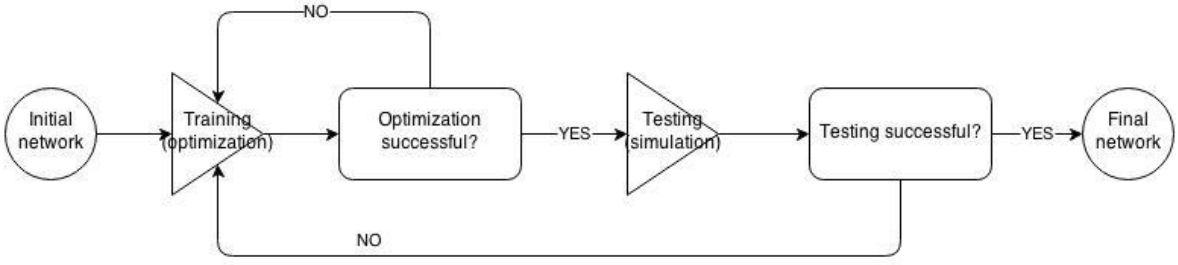


Image 4.1 Workflow summarizing the "training and testing" approach required for obtaining predictive power

4.3.2. Increasing the predictive power

Predictive power can be increased by increasing the number of reporters and conditions employed in the optimization stage. So far, I have employed two different reporters, each one studied using a set of about 40-50 mutants; the employed reporters are the GFP expressed upon the induction of Unfolded Protein Response and the cytoplasmic levels of the native GFP.
Unfortunately, I could not use the third planned reporter, i.e. the levels of the GFP in the endoplasmic reticulum measured as fluorescence of the fusion protein GFP-GPCR, because the reporter did not work as planned. The next step could therefore be repeating those experiments, or employing an ER-targeted GFP (fusing the fluorescent protein with a translocation signal sequence) as reporter of the GFP levels in the endoplasmic reticulum.

Moreover, predictive power is limited to the processes that have been described in details and subsequently optimized; therefore, extending the network allows for obtaining predictive power on many more processes. When completing the network and optimizing it (using supercomputers), in theory it should be possible to obtain a set of parameters that show predictive power in all the processes, thus allowing for simulating the behaviour of any mutant strains.
Finally, a complete network will allow using global reporters such as growth rate. These reporters might be particularly useful to optimize the whole network rather than single pieces of it, thus "tuning" all the processes together; they would also allow for optimizing and then employing the network to simulate different experimental conditions (e.g. changes in the metabolite sources) rather than mutant strains.

5. Bibliography

Applegate, D. L. (2006). The Traveling Salesman Problem: A Computational Study (p. 593). Princeton University Press.

Ashburner, M., Ball, C., Blake, J., & Botstein, D. (2000). Gene Ontology: tool for the unification of biology. Nature Genetics, 25(May), 25–29.

Balakrishnan, R., Park, J., Karra, K., Hitz, B. C., Binkley, G., Hong, E. L., Sullivan, J., Micklem, G. & Cherry, J. M. (2012). YeastMine--an integrated data warehouse for *Saccharomyces cerevisiae* data as a multipurpose tool-kit. Database. The Journal of Biological Databases and Curation, 2012, bar062. doi:10.1093/database/bar062

Baştanlar, Y., & Özuysal, M. (2014). Introduction to Machine Learning. In M. Yousef & J. Allmer (Eds.), miRNomics: MicroRNA Biology and Computational Analysis SE - 7 (Vol. 1107, pp. 105–128). Humana Press. doi:10.1007/978-1-62703-748-8_7

Bean, D. M., Heimbach, J., Ficorella, L., Oliver, S. G., & Favrin, G. (2014.). esyN: Network Building, Sharing and Publishing. *PLoS One*, (in publishing).

Beichl, I., & Sullivan, F. (2000). The metropolis algorithm. Computing in Science & Engineering, 65–69.

Bell, G. (2010). Experimental genomics of fitness in yeast. Proceedings. Biological Sciences /The Royal Society, 277(1687), 1459–67. doi:10.1098/rspb.2009.2099

Beurton-Aimar, M., Nguyen, T.-N., & Colombié, S. (2014). Metabolic Network Reconstruction and Their Topological Analysis. In M. Dieuaide-Noubhani & A. P. Alonso (Eds.), Plant Metabolic Flux Analysis SE - 2 (Vol. 1090, pp. 19–38). Humana Press. doi:10.1007/978-1-62703-688-7_2

Billington, J., Christensen, S., & Hee, K. Van. (2003). The Petri net markup language: concepts, technology, and tools.

Black, J.G. (1996). Microbiology. Principles and Applications. Third Edition. Prentice Hall. Upper Saddle River, New Jersey. pp. 144-148.

Blatke, M. A., Dittrich, A., Rohr, C., Heiner, M., Schaper, F., & Marwan, W. (2013). JAK/STAT signalling - an executable model assembled from molecule-centred modules demonstrating a module-oriented database concept for systems and synthetic biology. Molecular BioSystems, 9(6), 1290–1307. doi:10.1039/C3MB25593J

Blatke, M. A. (2011). Petri Nets in Systems Biology

Blomberg, A. (2011). Measuring growth rate in high-throughput growth phenotyping. Current Opinion in Biotechnology, 22(1), 94–102. doi:10.1016/j.copbio.2010.10.013

Cherry, J. M., Hong, E. L., Amundsen, C., Balakrishnan, R., Binkley, G., Chan, E. T., Christie, K. R., Costanzo, M. C., Dwight, S. S., Engel, S. R., Fisk, D. G., Hirschman, J. E., Hitz, B. C., Karra, K., Krieger, C. J., Miyasato, S. R., Nash, R. S., Park, J., Skrzypek, M. S., Simison, M., Weng, S. & Wong, E. D. (2012). Saccharomyces Genome Database: the genomics resource of budding yeast. Nucleic Acids Research, 40(Database issue), D700–5. doi:10.1093/nar/gkr1029

Couture-Beil, A. (2014). rjson: JSON for R. Retrieved from: http://cran.r-project.org/package=rjson

Day, R. N., & Davidson, M. W. (2009). The fluorescent protein palette: tools for cellular imaging. Chemical Society Reviews, 38(10), 2887–921. doi:10.1039/b901966a

De Deken, R. H. (1966). The Crabtree effect: a regulatory system in yeast. Journal of General Microbiology, 44(2), 149–56.

Doob, J. (1942). Topics in the theory of Markoff chains. Transactions of the American Mathematical Society, 1(I), 35–37.

Feres, R. (2007). Notes for Math 450, lecture 6. Stochastic Petri nets and reactions Petri nets. Available from http://www.math.wustl.edu/~feres/Math450Lect06.pdf [14/07/2014]

Gentleman, R. C., Carey, V. J., Bates, D. M., Bolstad, B., Dettling, M., Dudoit, S., Ellis, B., Gautier, L., Ge, Y., Gentry, J., Hornik, K., Hothorn, T., Huber, W., Iacus, S., Irizarry, R., Leisch, F.,

Giaever, G., & Nislow, C. (2014). The Yeast Deletion Collection: A Decade of Functional Genomics. Genetics, 197(2), 451–465. doi:10.1534/genetics.114.161620

Giaever, G., Chu, A. M., Ni, L., Connelly, C., Riles, L., Véronneau, S., … Johnston, M. (2002). Functional profiling of the *Saccharomyces cerevisiae* genome. Nature, 418(6896), 387–91. doi:10.1038/nature00935.
Supplementary Materials available from: http://genomics.lbl.gov/YeastFitnessData/slow_table.html

Gillespie, D. T. (1976). A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. Journal of Computational Physics, 22(4), 403–434. doi:10.1016/0021-9991(76)90041-3

Goffeau, A., Barrell, B. G., Bussey, H., Davis, R. W., Dujon, B., Feldmann, H., Galibert, F., Hoheisel, J. D., Jacq, C., Johnston, M., Louis, E. J., Mewes, H. W., Murakami, Y., Philippsen, P., Tettelin, H., & Oliver, S. G. (1996). Life with 6000 Genes. Science, 274(5287), 546–567. doi:10.1126/science.274.5287.546

Haas, P. J. (2002). Stochastic Petri Nets: Modelling, Stability. Simulation, 19–24.

Jonikas, M., Collins, S., Denic, V., & Oh, E. (2009). Comprehensive characterization of genes required for protein folding in the endoplasmic reticulum. *Science*, (March), 1693–1697.

Kahm, M., & Hasenbrink, G. (2010). grofit: fitting biological growth curves with R. Journal of Statistical Software, 33(7)

Kalderimis, A., Lyne, R., Butano, D., Contrino, S., Lyne, M., Heimbach, J., Hu, F., Smith, R., Stěpán, R., Sullivan, J., & Micklem, G. (2014). InterMine: extensive web services for modern biology. Nucleic Acids Research, 42(Web Server issue), W468–72. doi:10.1093/nar/gku301

Kanehisa, M., Goto, S., Kawashima, S., & Nakaya, A. (2002). The KEGG databases at GenomeNet. Nucleic Acids Research, 30(1), 42–6.

Kurtzman, C., & Fell, J. (2006). Yeast Systematics and Phylogeny — Implications of Molecular Identification Methods for Studies in Ecology. In G. Péter & C. Rosa (Eds.), Biodiversity and Ecophysiology of Yeasts SE - 2 (pp. 11–30). Springer Berlin Heidelberg. doi:10.1007/3-540-30985-3_2

Li, M., Zheng, R., Zhang, H., Wang, J., & Pan, Y. (2014). Effective identification of essential proteins based on priori knowledge, network topology and gene expressions. Methods (San Diego, Calif.), 67(3), 325–33. doi:10.1016/j.ymeth.2014.02.016

Li, C., Maechler, M., Rossini, A. J., Sawitzki, G., Smith, C., Smyth, G., Tierney, L., Yang, J. Y. H. & Zhang, J. (2004). Bioconductor: open software development for computational biology and bioinformatics. Genome Biology, 5(10), R80. doi:10.1186/gb-2004-5-10-r80

List of machine learning algorithms [Internet]. Wikipedia, The Free Encyclopedia; 2014 May 20, [cited 2014 Aug 14]. Available from: http://en.wikipedia.org/wiki/List_of_machine_learning_algorithms

Lopes, C. T., Franz, M., Kazi, F., Donaldson, S. L., Morris, Q., & Bader, G. D. (2010). Cytoscape Web: an interactive web-based network browser. Bioinformatics (Oxford, England), 26(18), 2347–8. doi:10.1093/bioinformatics/btq430

Marwan, W., Rohr, C., & Heiner, M. (2012). Petri Nets in Snoopy: A Unifying Framework for the Graphical Display, Computational Modelling, and Simulation of Bacterial Regulatory Networks. In J. van Helden, A. Toussaint, & D. Thieffry (Eds.), Bacterial Molecular Networks SE - 21 (Vol. 804, pp. 409–437). Springer New York. doi:10.1007/978-1-61779-361-5_21

Modelling language [Internet]. Wikipedia, The Free Encyclopedia; 2014 Mar 27, 06:42 UTC [cited 2014 Aug 14]. Available from: http://en.wikipedia.org/wiki/Modeling_language

Monte Carlo method [Internet]. Wikipedia, The Free Encyclopedia; 2014 Jun 22, [cited 2014 Aug 14]. Available from: http://en.wikipedia.org/wiki/Monte_Carlo_method

Murata, T. (1989). Petri nets: Properties, analysis and applications. Proceedings of the IEEE.

Nagai, T., Ibata, K., Park, E. S., Kubota, M., Mikoshiba, K., & Miyawaki, A. (2002). A variant of yellow fluorescent protein with fast and efficient maturation for cell-biological applications. Nat Biotech, 20(1), 87–90.

Olenych, S. G., Claxton, N. S., Ottenberg, G. K., & Davidson, M. W. (2007). The fluorescent protein color palette. Current Protocols in Cell Biology, Chapter 21, Unit 21.5. doi:10.1002/0471143030.cb2105s36

Ormö, M., Cubitt, A. B., Kallio, K., Gross, L. A., Tsien, R. Y., & Remington, S. J. (1996). Crystal Structure of the *Aequorea victoria* Green Fluorescent Protein. Science, 273 (5280), 1392–1395. doi:10.1126/science.273.5280.1392

Prendergast, F. G., & Mann, K. G. (1978). Chemical and physical properties of aequorin and the green fluorescent protein isolated from *Aequorea forskalea*. Biochemistry, 17(17), 3448–3453. doi:10.1021/bi00610a004

R Core Team. (2014). R: A Language and Environment for Statistical Computing. Vienna, Austria. Retrieved from: http://www.r-project.org/

Raeside, D. E. (1976). Monte Carlo principles and applications. Physics in Medicine and Biology, 21(2), 181.

Rohr, C., Marwan, W., & Heiner, M. (2010). Snoopy--a unifying Petri net framework to investigate biomolecular networks. Bioinformatics (Oxford, England), 26(7), 974–5. doi:10.1093/bioinformatics/btq050

Shaner, N., Steinbach, P., & Tsien, R. (2005). A guide to choosing fluorescent proteins. Nature Methods, (December).

Sommer, C., & Gerlich, D. W. (2013). Machine learning in cell biology - teaching computers to recognize phenotypes. Journal of Cell Science, 126(Pt 24), 5529–39. doi:10.1242/jcs.123604

Tsien, R. Y. (1998). The green fluorescent protein. Annual Review of Biochemistry, 67, 509–44. doi:10.1146/annurev.biochem.67.1.509

Uhlmann, F., Bouchoux, C., & López-Avilés, S. (2011). A quantitative model for cyclin-dependent kinase control of the cell cycle: revisited. Philosophical Transactions of the Royal Society of London. Series B, Biological Sciences, 366(1584), 3572–83. doi:10.1098/rstb.2011.0082

Vera-Licona, P., Jarrah, A., Garcia-Puente, L. D., McGee, J., & Laubenbacher, R. (2014). An algebra-based method for inferring gene regulatory networks. BMC Systems Biology, 8, 37. doi:10.1186/1752-05

Warmflash, D., & Ciftcioglu, N. (2007). Living Interplanetary Flight Experiment (LIFE): an experiment on the survivability of microorganisms during interplanetary transfer. Workshop on the exploration of Phobos and Deimos, 34, 84001.

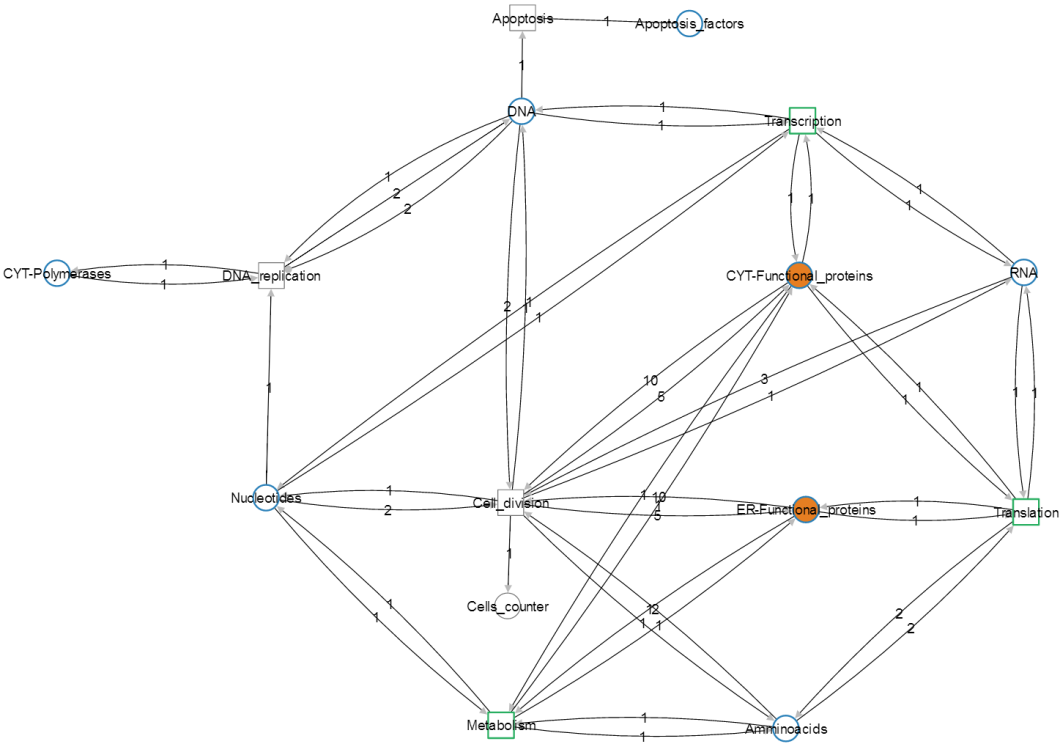
Wickham, H. (2009). ggplot2: elegant graphics for data analysis. Springer New York.

Zapata-Hommer, O., & Griesbeck, O. (2003). Efficiently folding and circularly permuted variants of the Sapphire mutant of GFP. BMC Biotechnology, 3, 5. doi:10.1186/1472-6750-3-5

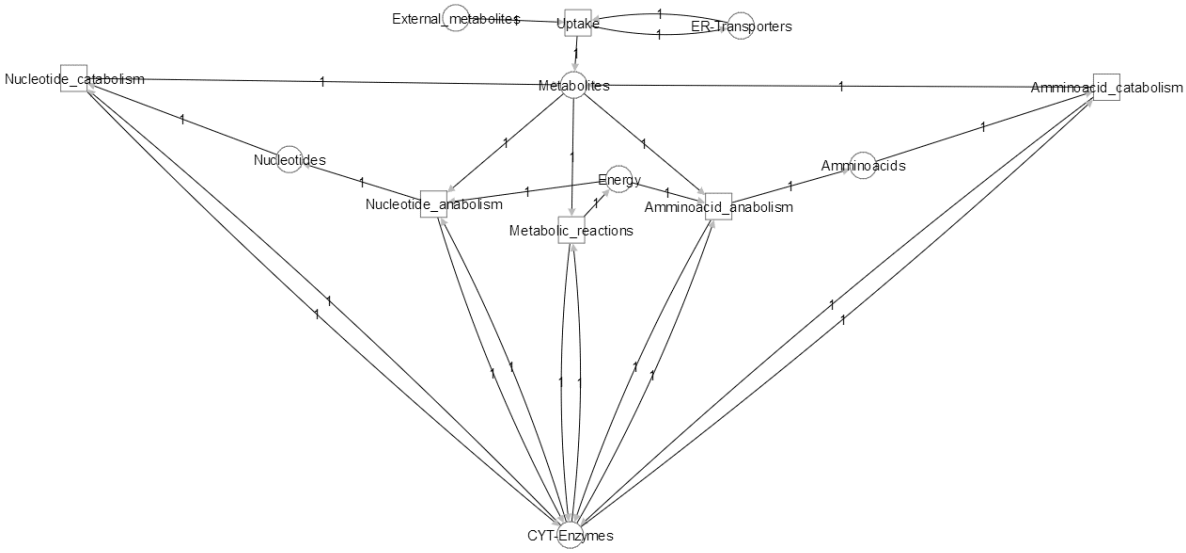
Zeileis, A., & Grothendieck, G. (2005). zoo: S3 Infrastructure for Regular and Irregular Time Series. Journal of Statistical Software, 14(6), 1–27.

5. Supplementary materials

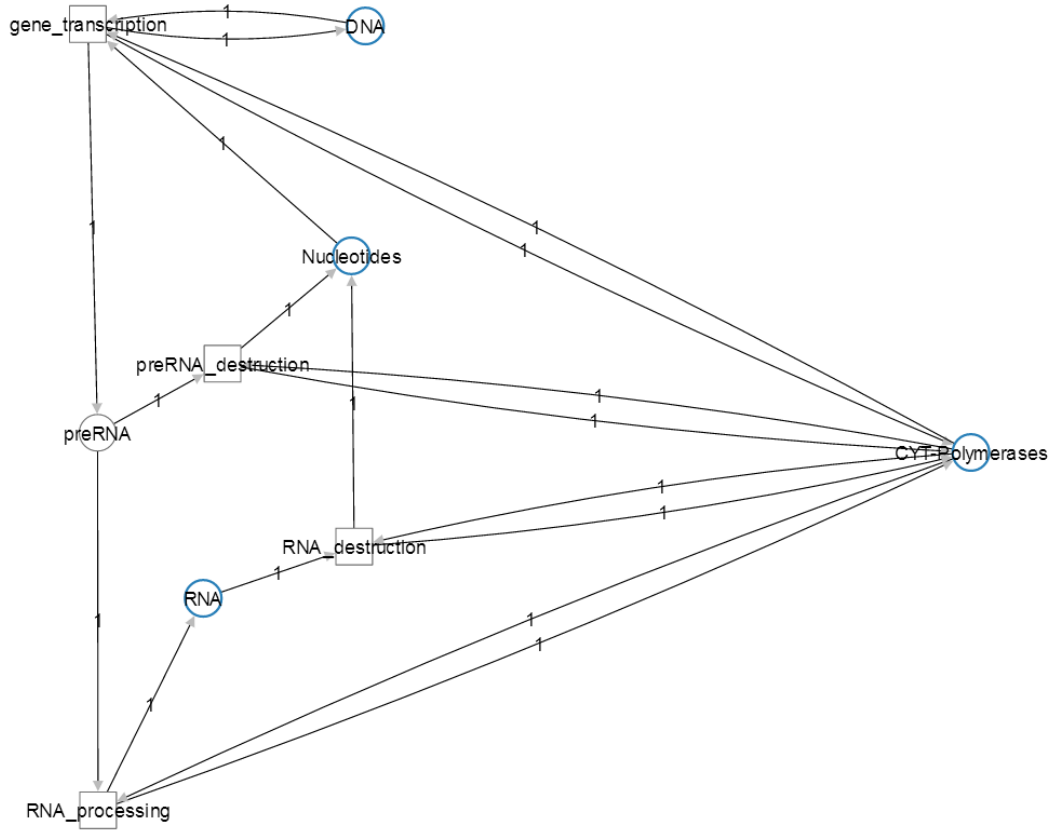
5.1. Petri Networks



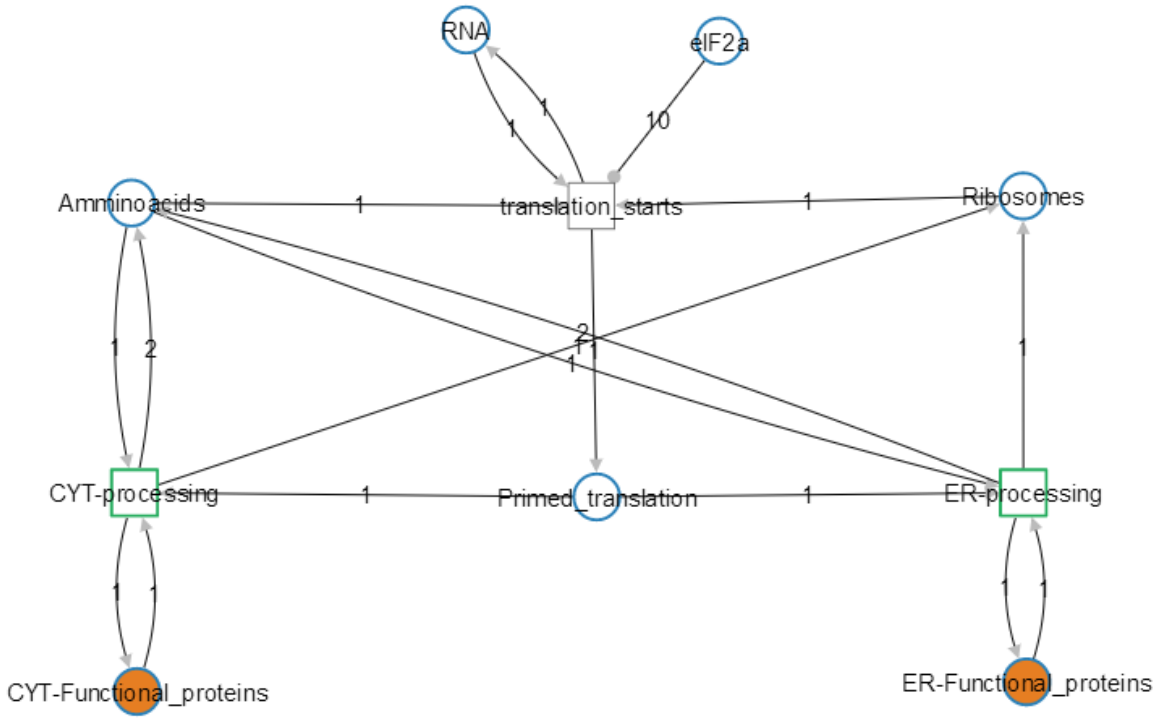
First layer: whole-cell network



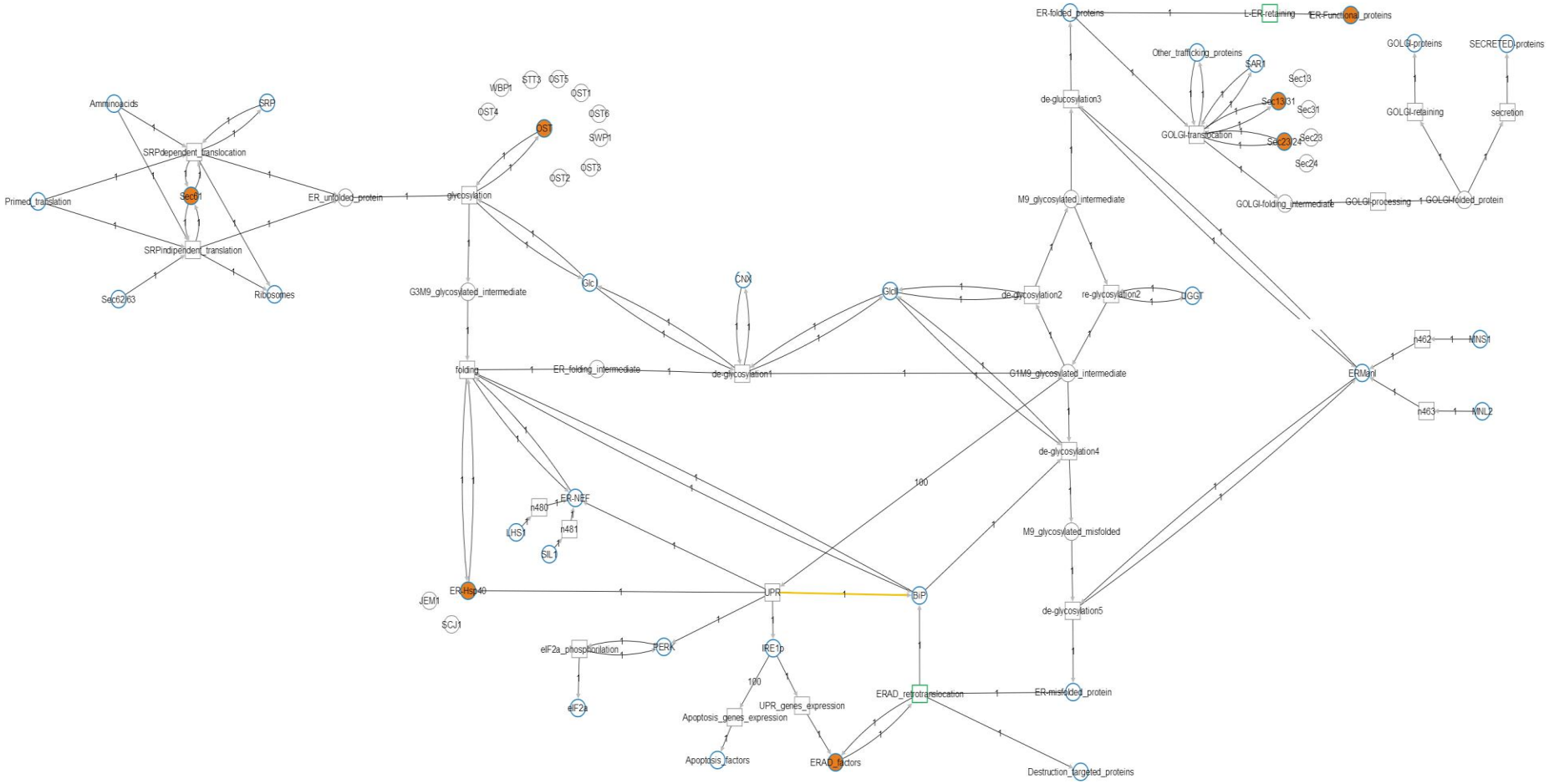
Second layer: metabolism



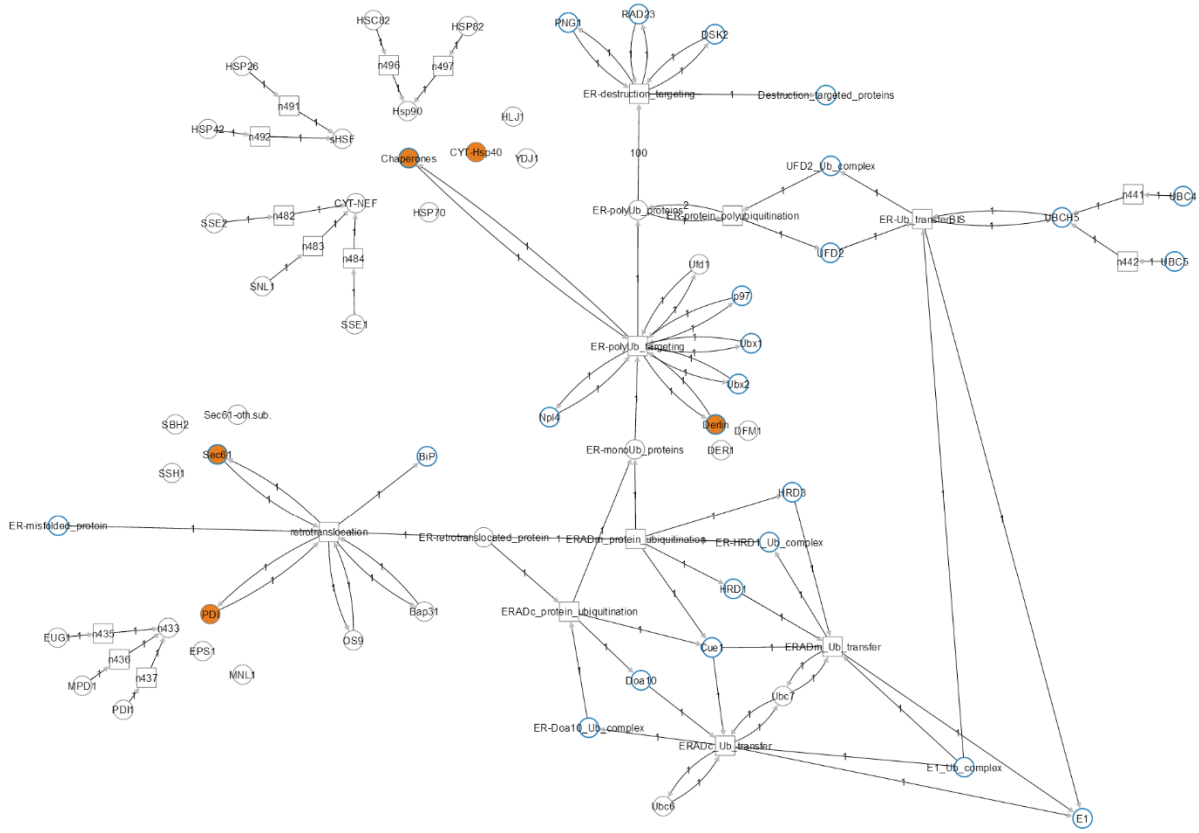
Second layer: transcription



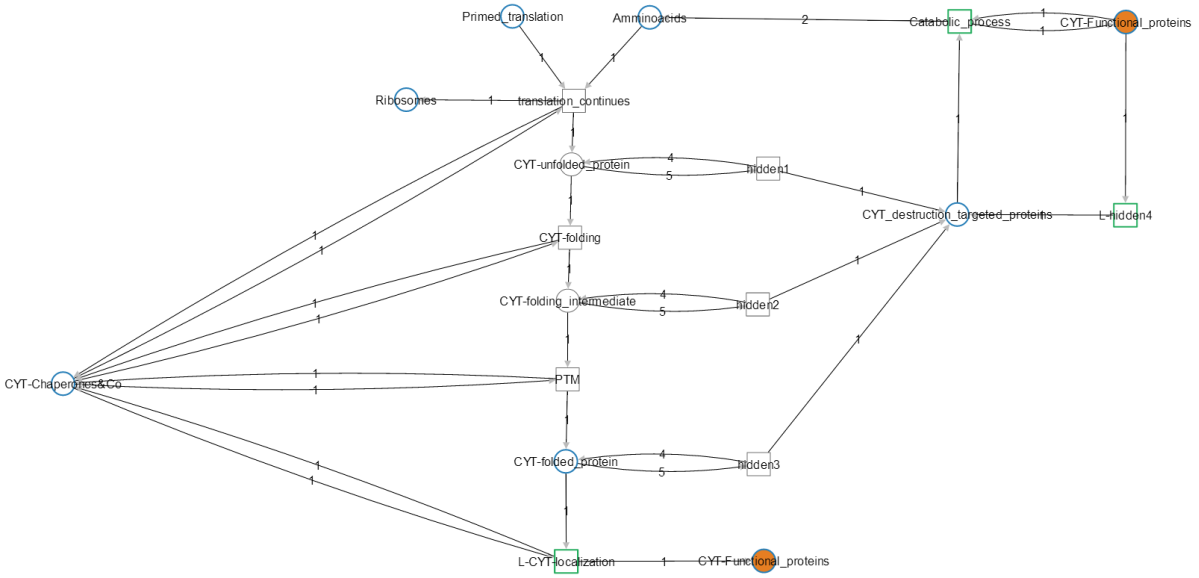
Second layer: translation



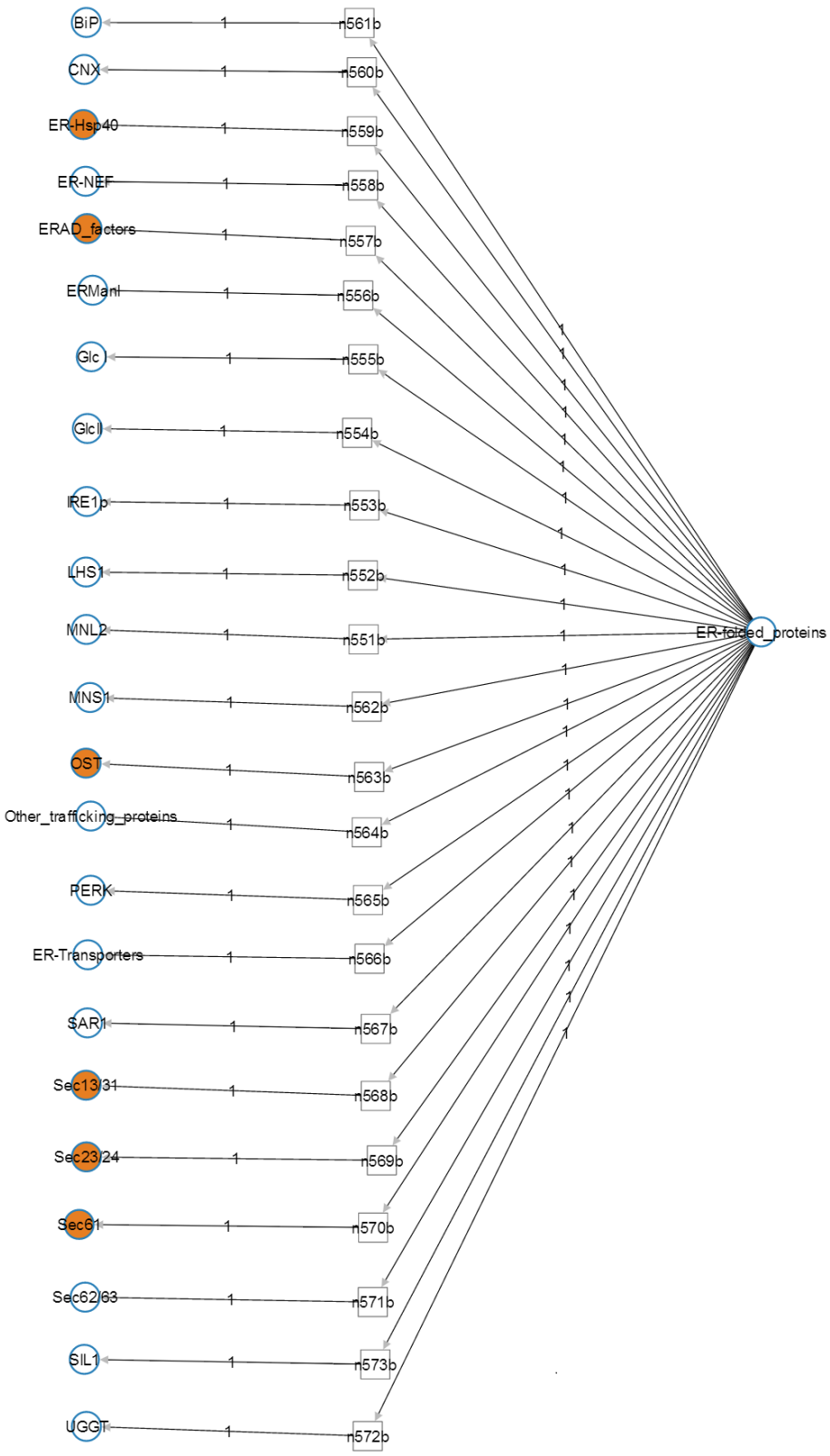
Third layer: protein processing in the ER



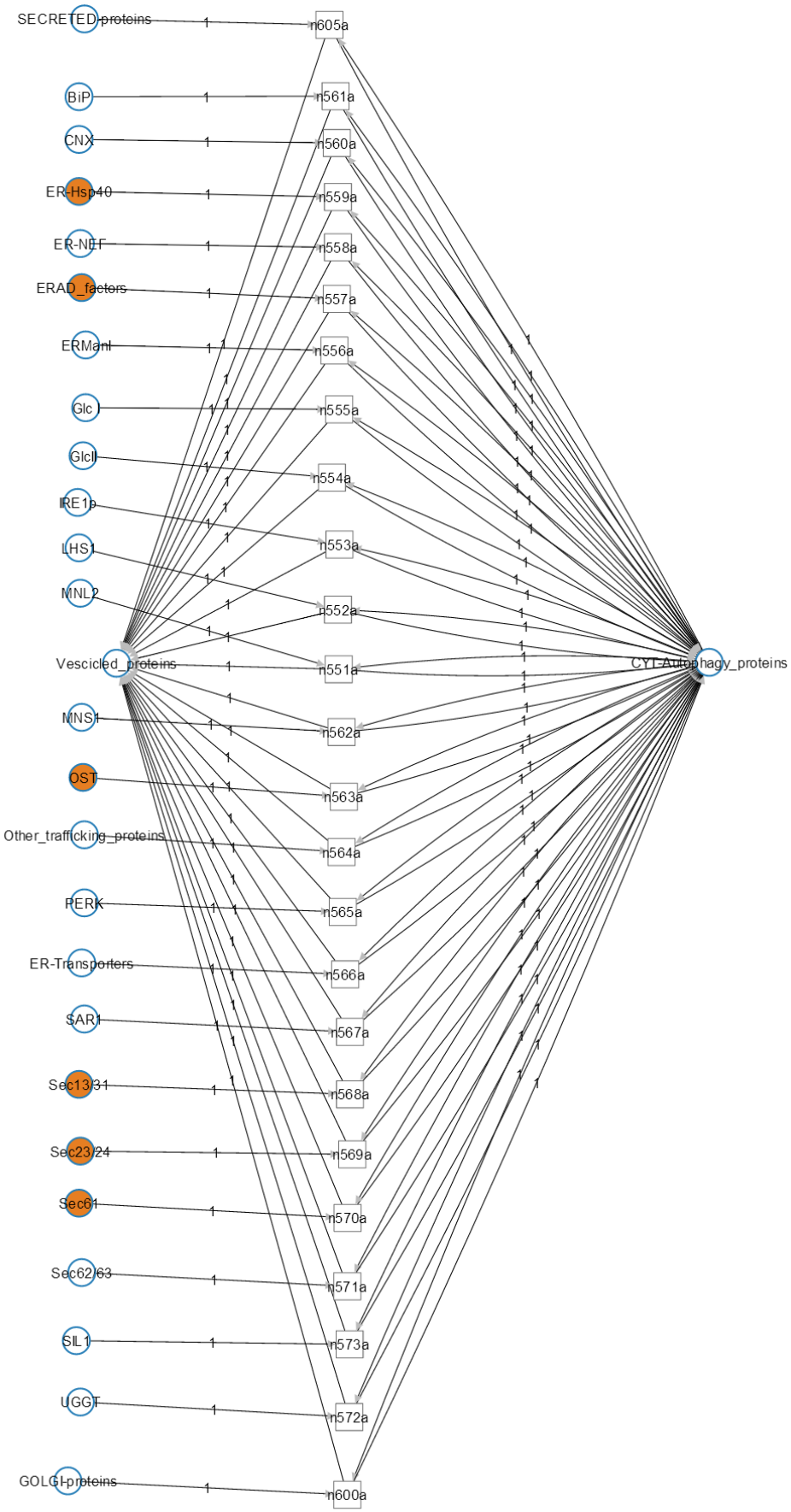
Third layer: Endoplasmic Reticulum Associated Degradation



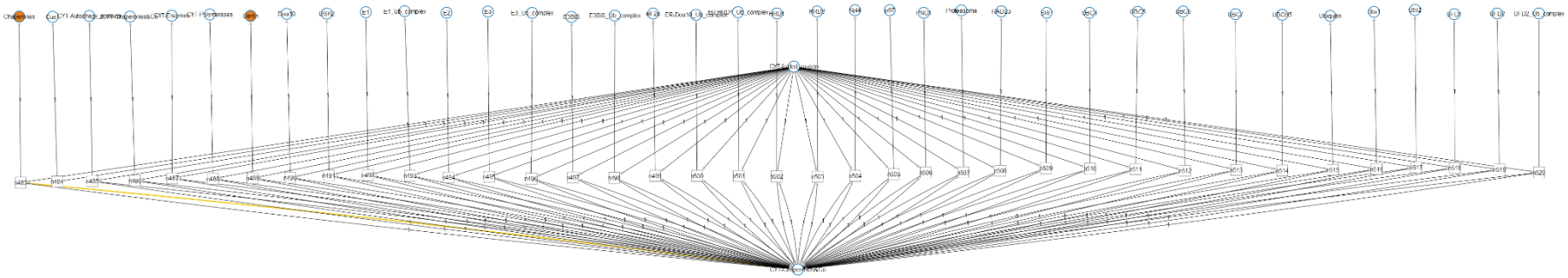
Third layer: protein processing in the cytoplasm



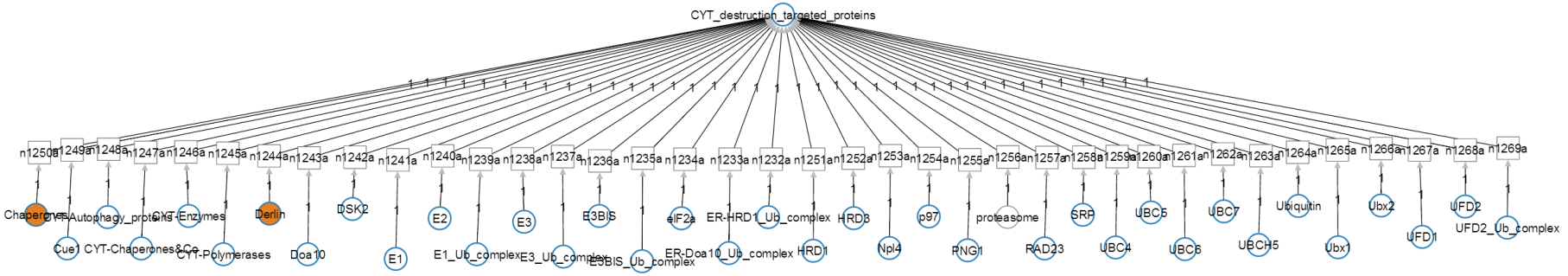
Logic layer: ER retaining



Logic layer: vesicles formation



Logic layer: cytoplasmic localization



Logic layer: hidden transition (4)

5.2. Scripts

Script locating lines

It is needed by all the split scripts in order to locate the input files and the subscripts.

```
if (!(file.exists(scriptFile))) {
  repeat {
    choiceInput  <- readline ("\n Press 1 to enter the path manually, or choose the
directory you want to search in ")
    if   (choiceInput == 1) {
      scriptPath    <- readline("Please enter the path to the script file: \n")
      if (file.exists(paste (scriptPath, scriptFile, sep = ""))) {break}
    } else {
      if (.Platform$OS.type == "windows")
        diskPath    <- paste(choiceInput, ":/", sep ="")
      else diskPath <- "/"
      inputList     <- list.files(diskPath, pattern= "3-Simulator.R", full.names =
TRUE, recursive = TRUE)
      if    (length(inputList) != 0) {
        print (dirname(inputList))
        repeat {
          choiceInput3 <- type.convert(readline ("\n please enter the number of the
line you want to use "), as.is = TRUE)
          if (choiceInput3 %in% c(1:length(inputList))) {break
          } else  cat ("please enter a valid value \t")
        }
        scriptPath <- dirname(inputList[choiceInput3])
        break
      }
    }
  }
  setwd(scriptPath)
}
```

Variable script

It is needed for reading the input files and setting simulation and optimization parameters.

```
## This section is needed to read the input files
  if  (!("rjson" %in% rownames(installed.packages())))
    install.packages("rjson")
  if  (!("zoo" %in% rownames(installed.packages())))
    install.packages("zoo")
  library('rjson')
  library('zoo')

  inputData       <- fromJSON(paste(readLines("../Materials/merge_matrices.txt"),
collapse="")) #you can give a file path where I have "merge_matrices.txt"
  matrixInhibit  <- do.call(rbind, inputData$inhib)     #matrix of the weights of
inhibitory arcs, always going FROM places TO transitions
  inhibIndex      <- (which(matrixInhibit>=1, arr.ind=TRUE))    #see where are
inhibitions in the matrix
  matrixInward  <- do.call(rbind, inputData$post)
  matrixOutward <- -1*(do.call(rbind, inputData$pre))
  matrixDelta   <- matrixInward + matrixOutward
  matrixTokens  <- inputData$marking    #tokens in all the places at time 0 (i.e.
the tokens you've written in the network)
  vectorPar     <- inputData$k                  # Mass Action parameters

  transitNames  <- inputData$tnames
  placesNames     <- inputData$pnames   #you'll need those variables later to run
the simulation
  kplaceN           <- length(placesNames)  # Number of places
  ktransitN       <- length(transitNames) #Numb of Transitions
```

```
  colnames(matrixInhibit) <- colnames(matrixInward) <- colnames(matrixOutward) <-
colnames(matrixDelta) <- placesNames
}

## This section is needed to call most of the required variables
  totIteration  <- 20
  mcNumber      <- 100
  iterNumber    <- 100
  stepsNumber   <- 2000
  minWeight     <- 0.001
  maxWeight     <- 4
  invTemp         <- 10     #Inverse Temperature


## Creating new vector and matrices representing the mutated network
if ((scriptFile != "3. Simulator.R") && (scriptFile != "0. Starting.R")) {
  if (interactive == "Y") {
    source ("Subscripts/Variables_ask.R")
  } else {source ("Subscripts/Variables_set.R")}
} else {source("Subscripts/Variables_start.R")}

if (scriptFile != "3. Simulator.R") {
  ## matrices and vectors must be created for each mutation considered
  allallIndexM        <- NA
  for (mutCounter in 1:cPMNnumber) {
    nam1  <- paste ("allIndexM",mutCounter,sep = "")
    assign (nam1, unique(c(which(matrixInward [,choicePlaceMutantN[mutCounter]]!=
0), which(matrixOutward [,choicePlaceMutantN[mutCounter]]!= 0))))
    nam2  <- paste ("inhibIndexM",mutCounter,sep = "")
    assign (nam2, which(matrixInhibit[,choicePlaceMutantN[mutCounter]]!= 0))
    allallIndexM <- unique(c(allallIndexM,get(nam1)))

    tempInhibit <- matrixInhibit
    tempInhibit[get(nam2),choicePlaceMutantN[mutCounter]] = 0
    nam3  <- paste ("matrixInhibitM",mutCounter,sep = "")
    assign (nam3, tempInhibit)
  }
  allallIndexM <- allallIndexM[-1]
  simulcoreTot <- matrix(nrow = cPMNnumber+1, ncol= kplaceN+1)


## Ancillary function
  vectorParStart<- vectorPar
  tokenProduct   <- vector(length=ktransitN)
  prova = which(matrixOutward != 0, arr.ind = TRUE)
  prova3 = which(matrixInward !=0, arr.ind = TRUE)
  temp = list()
  temp2 = list()
  temp3 = list()
  for(i in 1:ktransitN){
    # the product of the number of tokens in the input places for each transition
    temp[[i]]  <- prova[(prova[,1] == i),2]
    tokt   <- matrixTokens[temp[[i]]]
    tot    <- prod(tokt)
    tokenProduct[i] <-tot
    temp2[[i]] <- unique(c(prova[(prova[,1] == i),2], prova3[(prova3[,1] == i),2]))
    if (length(temp2[[i]]) == 0) {
      temp3[[i]] <- unique(c(prova[(prova[,1] == i),2], prova3[(prova3[,1] ==
i),2]))
    } else if (length(temp2[[i]]) == 1) {
      temp3[[i]] <- unique(which(matrixOutward[,temp2[[i]]] != 0, arr.ind = TRUE))
    } else {
      temp3[[i]] <- unique(which(matrixOutward[,temp2[[i]]] != 0, arr.ind =
TRUE)[,1])
    }
  }
  vectorProb   <- tokenProduct*vectorPar        # probability vector completed
```

Variable "ask" subscript

It is needed for setting further parameters employed in the optimization and pre-testing stage.

```
if (file.exists("../Materials/kparametersBEST.txt")) {
  choiceSettingsD = readline ("\n Do you want to use new optimized mass action
parameters? press 1 for yes " )
  if (choiceSettingsD == 1) {
    vectorPar <- readLines("../Materials/kparametersBEST.txt")
    vectorPar   <- type.convert(unlist(strsplit(vectorPar, "\t")))
  }
}

## This section is needed to set the optimization parameters
cat ("\Total repetitions= ", totIteration, "\n MonteCarlo steps = ", mcNumber)
cat ("\n duration of the simulation = ", stepsNumber, "\n Min value of k parameter
= ", minWeight, "\n Max value of k parameter = ", maxWeight)
repeat {
  choiceSettingsB   <- readline("\n Press ENTER to use default values, or insert
new values divided by space \n")
  if (choiceSettingsB == "") {break
  } else {
    simulParametersB  = unlist(strsplit(choiceSettingsB, " "))
    if ((length(simulParametersB) == 5) && (min(simulParametersB) >0)) {
      simulParametersB1 = type.convert(simulParametersB[1:3])
      simulParametersB2 = type.convert(simulParametersB[4:5])
      if ((is.integer(simulParametersB1)) && (is.numeric(simulParametersB2))) {
        totIteration  <- simulParametersB1[1]
        mcNumber      <- simulParametersB1[2]
        stepsNumber   <- simulParametersB1[3]
        minWeight     <- simulParametersB2[1]
        maxWeight     <- simulParametersB2[2]
        break
      }
    }
  }
}

## This section is needed to set the reporter and the mutating places
while (!(exists("cPMNnumber"))) {
  cat   ("the places in this network are: \n")
  print (placesNames)

  while (!(exists("choicePlaceNumber"))) {
    choicePlaceName      <- readline ("insert the exact name of the place you want
to optimize: ")
    if (choicePlaceName %in% placesNames) {
      choicePlaceNumber  <- match (choicePlaceName, placesNames)
    }
  }
  while (!(exists("choicePlaceMutantName"))) {
    choiceSettingsE <- readline ("Press ENTER to load the training set (mutants
ratio) from the external file, anything else to write them manually: ")
    if (choiceSettingsE == "") {
      trainingMatrix <- as.matrix(read.csv("../Materials/TrainingSet.csv", header =
FALSE, sep = " "))
      choicePlaceMutantName <- trainingMatrix[1,]
      ratioReal             <- type.convert(trainingMatrix[2,])
    } else {
      tempInput <- unlist(strsplit(readline ("insert the exact name of the places
you want to delete, divided by space: "),," "))
      if (all(tempInput %in% placesNames)) {choicePlaceMutantName <- tempInput}
        tempRatio <- type.convert(unlist(strsplit(readline ("insert the ratios
between the fluorescence measured in wt and mut, divided by space: "),," ")))
        if ((is.numeric(tempRatio)) && (length(tempRatio) ==
length(choicePlaceMutantName)))
          ratioReal <- tempRatio
      }
```

```
    }

    if (scriptFile == "1. MLA 2.1.R") {choicePlaceMutantName <-
unique(c(choicePlaceMutantName, "ER-Transporters"))}

    # checking that the reporter is different from the mutating place(s)
    if (!(choicePlaceName %in% choicePlaceMutantName)) {
      choicePlaceMutantN    <- match(choicePlaceMutantName, placesNames)
      cPMNnumber            <- length(choicePlaceMutantN)
    } else {
      choiceSettingsD <- readline ("you're trying to mutate the place you have
chosen to evaluate. Press 1 to discard this mutation, 2 to modify the optimized
places, 3 to modify the mutated places, anything else to change everything")
      if  (choiceSettingsD == 1) {
        choicePlaceMutantName <- choicePlaceMutantName[choicePlaceMutantName !=
choicePlaceName]
        if (!(length(choicePlaceMutantName > 1))) {
          cat ("\n It seems you have deleted all the mutated places; please, insert
new ones")
          rm("choicePlaceMutantName")
        }
      } else if (choiceSettingsD == 2) { rm("choicePlaceNumber")
      } else if (choiceSettingsD == 3) { rm("choicePlaceMutantName")
      } else {rm("choicePlaceNumber", "choicePlaceMutantName")}
    }
  }

 ## This section is needed to decide which rules should be employed
    forbiddenPar         <- 0
    allowedTransit       <- c(1:ktransitN)
    choiceSettingsC1     <- readline ("Press 1 to prevent changes of the logic
transitions parameters, anything else to skip this rule: ")
    if (choiceSettingsC1 ==1) {forbiddenPar    <- which(grepl("logic",
transitNames))}
    choiceSettingsC2     <- readline ("Press 1 to prevent changes of the mutated
transitions parameters, anything else to skip this rule: ")
    if (choiceSettingsC2 ==1) {forbiddenPar    <- unique(c(allallIndexM,
forbiddenPar))}
    choiceSettingsC4     <- readline ("Press 1 to couple duplicated transitions
together, anything else to skip this rule: ")
    if (choiceSettingsC4 ==1)   {
      duplPos    <- which (grepl("dupl", transitNames))
    } else {duplPos = duplNames <- ""}


    if (file.exists("../Materials/forbiddenPar.txt")) {
      choiceSettingsC3   <- readline ("Press 1 to read from an input file, 2 to
write the names of forbidden transition now, anything else to skip this rule: ")
      if (choiceSettingsC3 ==1) {
        forbiddenList  <- readLines("../Materials/forbiddenPar.txt")
        forbiddenTemp  <- unlist(strsplit(forbiddenList, " "))
        forbiddenPar   <- unique(c(forbiddenPar,which(transitNames %in%
forbiddenTemp)))
      }
    } else {choiceSettingsC3   <- readline ("Press 2 to write the names of
forbidden transition now, anything else to skip this rule: ")}
    if  (choiceSettingsC3 == 2) {
      forbiddenList   <- readline ("\n Write the names of the transitions that
cannot change, divided by spaces")
      forbiddenTemp   <- unlist(strsplit(forbiddenList, " "))
      forbiddenPar    <- unique(c(forbiddenPar,which(transitNames %in%
forbiddenTemp)))
    }
    if ((length(forbiddenPar) >1 ) || ((length(forbiddenPar) ==1 ) && (forbiddenPar
!=0))) {allowedTransit  <- allowedTransit[-forbiddenPar]}
  }
```

Variable "start" subscript

It is a subscript for setting further parameters that are only employed in the starting and simulation stage.

```
if (file.exists("../Materials/kparametersBEST.txt")) {
  choiceSettingsD = readline ("\n Do you want to use new optimized mass action
parameters? press 1 for yes " )
  if (choiceSettingsD == 1) {
    vectorPar <- readLines("../Materials/kparametersBEST.txt")
    vectorPar   <- type.convert(unlist(strsplit(vectorPar, "\t")))
  }
}

## This section is needed to set the optimization parameters
cat    ("\n The default values are: \n Duration of the simulation \t = ",
stepsNumber, "\n Iterations of the simulation = ", iterNumber)
repeat {
  choiceSettingsB   <- readline("\n Press ENTER to use default values, or insert
new values divided by space \n")
  if (choiceSettingsB == "")
    break
  else {
    simulParametersB  = type.convert(unlist(strsplit(choiceSettingsB, " ")))
    if ((length(simulParametersB) == 2) && (is.integer(simulParametersB)) &&
(min(simulParametersB) >0)) {
      stepsNumber   <- simulParametersB[1]
      iterNumber    <- simulParametersB[2]
      break
    }
  }
}

## This section is needed to set the reporter and the mutating places
if (scriptFile  == "0. Starting.R") {
  while (!(exists("choicePlaceNumber"))) {
    cat    ("the places in this network are: \n")
    print (placesNames)
    tempInput      <- readline ("insert the exact name of the place you want to
optimize: ")
    if (tempInput %in% placesNames) {
      choicePlaceName       <- tempInput
      choicePlaceNumber     <- match (choicePlaceName, placesNames)
      choicePlaceMutantName <- placesNames[-choicePlaceNumber]
      choicePlaceMutantN    <- match(choicePlaceMutantName, placesNames)
      cPMNnumber            <- length(choicePlaceMutantN)
    }
  }
} else {
  tableGlobal             <- matrix(ncol= kplaceN+1, nrow= iterNumber)
  colnames(tableGlobal) <- c(placesNames, "Dead State?")
  #Table to summarize the results of all runs and the mean value of the runs (for
each place)
  spotsN      <- 1000          # Number of sampling points in which you interpolate
  sensSd      <- 2             # Simulations farther than "sensSd" Standard
Deviation won't be considered
  while (!(exists("choicePlaceNumber"))) {
    cat    ("the places in this network are: \n")
    print (placesNames)
    choicePlaceName = readline ("insert the exact name of the place you're
interested in, or press ENTER if you don't want to consider a particular place ")
    if (choicePlaceName == "") {
      cat ("\n \t All places will be considered \n")
      choicePlaceNumber <- c(1:kplaceN)
    }else if (choicePlaceName %in% placesNames) {choicePlaceNumber  = which
(colnames(matrixInward) ==  choicePlaceName)}
  }
}
```

Simulcore subscript

It is the core of the simulation that is only employed in the simulation stage.

```
Simulcore <- function(vectorPar, vectorProb,inhibIndex, matrixInhibit){
    matrixMatrix     <- matrix(ncol =kplaceN+1, nrow= stepsNumber)
    matrixMatrix[1,] <- c(matrixTokens,0)
    totTime            <- 0

    for (matrixRow in 1:stepsNumber) {
      # the following lines implement the core of the Gillespie algorithm;
      vectorTrans  <- 1:ktransitN
      vectorTime   <- 0
      for (x in 1:ktransitN) {
        if (vectorProb[x] <= 0) {vectorTime[x] = Inf}
        else {vectorTime[x] <-rexp(1,vectorProb[x])}
      }
        repeat {
            parTime <- min(vectorTime)
            if (parTime == Inf) {
                cat("\t  you reached a dead state!! \t")
              matrixMatrix[matrixRow,] <- c(matrixTokens, -totTime)
              return(matrixMatrix[c(1:matrixRow),])
            }
            chance  <- which(vectorTime == parTime)
            if (length(chance) >1)
              chance    <- sample(chance,1)
            rn        <- vectorTrans[chance]

            if (length(inhibIndex) == 0) {
                if (all(matrixTokens >= -matrixOutward[rn,])) {
                matrixTokens <- matrixTokens + matrixDelta[rn,]
                    totTime      <- totTime + parTime
                    matrixMatrix[matrixRow,] <- c(matrixTokens, totTime)
                    break
                }
            } else {
                indexIndex  <- inhibIndex[which(inhibIndex[,1] ==rn),2]
                if ((length(indexIndex) == 0) || (all(matrixTokens[indexIndex] <
matrixInhibit[rn,indexIndex]))) {
                    if (all(matrixTokens >= -matrixOutward[rn,])) {
                    matrixTokens <- matrixTokens + matrixDelta[rn,]
                        totTime      <- totTime + parTime
                        matrixMatrix[matrixRow,] <- c(matrixTokens, totTime)
                        break
                    }
                }
            }
            vectorTrans <- vectorTrans[!vectorTrans == rn]
            vectorTime  <- vectorTime[-chance]
            if  (length(vectorTrans) == 0){
                cat("\t  you reached a dead state!! \t")
                matrixMatrix[matrixRow,] = c(matrixTokens, -totTime)
                return(matrixMatrix[c(1:matrixRow),])
            }
        }

        importantTemp <- temp3[[rn]]
        important <- importantTemp[which(vectorPar[importantTemp] !=0)]
    for (i in important) {
      vectorProb[i] <-prod(matrixTokens[temp[[i]]]) * vectorPar[i]
    }
  }
  return(matrixMatrix[c(1:matrixRow),])
}
```

Simulcore Opt subscript

It is the core of the simulation that is employed in all the other stages.

```
Simulcore <- function(vectorPar, vectorProb, inhibIndex, matrixInhibit){
    totTime          <- 0

    for (matrixRow in 1:stepsNumber) {
      # the following lines implement the core of the Gillespie algorithm;
      vectorTrans  <- 1:ktransitN
      vectorTime   <- 0
      for (x in 1:ktransitN) {
        if (vectorProb[x] <= 0) {vectorTime[x] = Inf}
        else {vectorTime[x] <-rexp(1,vectorProb[x])}
      }
        repeat {
            parTime <- min(vectorTime)
      if (parTime == Inf) {
        cat("\t  YOU reached a dead state!! \t
        matrixMatrix = c(matrixTokens, -totTime)
        return(matrixMatrix)
      }
            chance  <- which(vectorTime == parTime)
            if (length(chance) >1)
              chance    <- sample(chance,1)
            rn        <- vectorTrans[chance]

            if (length(inhibIndex) == 0) {
                if (all(matrixTokens >= -matrixOutward[rn,])) {
                  matrixTokens <- matrixTokens + matrixDelta[rn,]
                  totTime      <- totTime + parTime
                break
                }
            } else {
                indexIndex  <- inhibIndex[which(inhibIndex[,1] ==rn),2]
                if (((length(indexIndex) == 0) || (all(matrixTokens[indexIndex] <
matrixInhibit[rn,indexIndex]))) && (all(matrixTokens >= -matrixOutward[rn,]))) {
                  matrixTokens <- matrixTokens + matrixDelta[rn,]
                  totTime      <- totTime + parTime
                  break
                }
            }
            vectorTrans <- vectorTrans[-chance]
            vectorTime  <- vectorTime[-chance]
            if  (length(vectorTrans) == 0){
                cat("\t  you reached a dead state!! \t")
                matrixMatrix = c(matrixTokens, -totTime)
                return(matrixMatrix)
            }
        }
        importantTemp <- temp3[[rn]]
        important <- importantTemp[which(vectorPar[importantTemp] !=0)]
        for (i in important) {
          vectorProb[i] <-prod(matrixTokens[temp[[i]]]) * vectorPar[i]
        }
    }
    matrixMatrix <- c(matrixTokens, totTime)
    return(matrixMatrix)
}
```

Starting script

It is only useful for creating the mutant set during the TESTING of the optimization script.

```
## here it starts the "declaration" part"
scriptFile  <- "0. Starting.R"
if (!(file.exists(scriptFile))) {...}
source ("Subscripts/Variables.R")
source ("Subscripts/simulcoreOpt.R")


## here it starts Montecarlo Simulation
meanMatrix <- matrix(ncol = kplaceN+1, nrow = iterNumber)
vectorResults <- vector(length = (cPMNnumber+1))

for (iterCounter in 1:iterNumber) {
  meanMatrix[iterCounter,]  <- Simulcore(vectorPar, vectorProb, inhibIndex,
matrixInhibit)
}
cat ("\t  1")
if (any(meanMatrix[,kplaceN+1] <= 0)) {
  stop("\n Error! Your starting network is not viable!!")
}
vectorResults[1] <- mean(meanMatrix[,choicePlaceNumber])


for (mutCounter in 1:cPMNnumber) {
  allIndexM      <- get(paste ("allIndexM", mutCounter,sep=""))
  inhibIndexM    <- get(paste ("inhibIndexM",mutCounter,sep = ""))
  matrixInhibitM <- get(paste ("matrixInhibitM",mutCounter,sep = ""))

  vectorParM <- vectorPar
  vectorParM[allIndexM] = 0
  vectorProbM <- vectorProb
  vectorProbM[allIndexM] = 0

  for (iterCounter in 1:iterNumber) {
    meanMatrix[iterCounter,]  <- Simulcore(vectorParM, vectorProbM, inhibIndexM,
matrixInhibitM)
  }
  loopCheck <- mutCounter+1
  cat ("\t", loopCheck)
  if (any(meanMatrix[,kplaceN+1] <= 0)) {
    cat ("\n This mutant is not viable and therefore it will be discharged")
    vectorResults[loopCheck] = NA
  } else {vectorResults[loopCheck] <-mean(meanMatrix[,choicePlaceNumber])}
}


simulcoreMat <- vectorResults[c(2:(cPMNnumber+1))]/vectorResults[1]
testDeath    <- which(is.na(simulcoreMat))
if (length(testDeath) != 0) {
  if (length(testDeath) == length(simulcoreMat)) {
    stop("\n Error! All the mutants are not viable!!")
  }
  choicePlaceMutantName <- choicePlaceMutantName[-testDeath]
  simulcoreMat          <- simulcoreMat[-testDeath]
}
trainingSet <- matrix(nrow =2, ncol= length(simulcoreMat))
trainingSet[1,] = choicePlaceMutantName
trainingSet[2,] = simulcoreMat
tempName  <- "../Materials/TrainingSet.csv"
write.table(trainingSet, tempName, quote = FALSE, sep= " ", row.names = FALSE,
col.names = FALSE)
```

Optimization script

It is the proper optimization script to be used for training the whole-cell network; it employs most of the needed condition checks.

```
## here it starts the "declaration" part"
scriptFile  <- "1. MLA 2.2.R"
interactive <- "Y"
if (!(file.exists(scriptFile))) {..}
source ("Subscripts/Variables.R")
source ("Subscripts/simulcoreOpt.R")
dir.create ("../Materials/MontecarloResults", showWarnings = FALSE)

## here it starts Monte Carlo Simulation
for (totCounter in 1:totIteration) {
  vectorParOld = vectorPar <- vectorParStart
  countMcReal <- 0
  # first simulations to simulate the original network
  simulcoreTot[1,] <- Simulcore(vectorPar, vectorProb, inhibIndex, matrixInhibit)
  cat ("\t", 1)
  if ((simulcoreTot[1,choicePlaceNumber] == 0) || (simulcoreTot[1,kplaceN+1] == 0))
{
    stop("\n Error! Your starting network is not viable!!")
  }
  for (mutCounter in 1:cPMNnumber) {
    allIndexM      <- get(paste("allIndexM", mutCounter,sep=""))
    inhibIndexM    <- get(paste ("inhibIndexM",mutCounter,sep = ""))
    matrixInhibitM <- get(paste ("matrixInhibitM",mutCounter,sep = ""))

    vectorParM <- vectorPar
    vectorParM[allIndexM] = 0
    vectorProbM <- vectorProb
    vectorProbM[allIndexM] = 0

    simulcoreTot[mutCounter+1,] <- Simulcore(vectorParM, vectorProbM, inhibIndexM,
matrixInhibitM)
    cat ("\t", mutCounter+1)
    if (simulcoreTot[mutCounter+1,kplaceN+1] == 0) {
      stop("\n Error! One or more mutants are not viable; you need you use another
script.")
    }
  }
  valuesNew   <- simulcoreTot[, choicePlaceNumber]
  valuesDeltaVec  <- abs((valuesNew[c(2:(cPMNnumber+1))]/valuesNew[1]) - ratioReal)
  valuesDeltaOld  <- sum(valuesDeltaVec^2)
  valuesDeltaBegin = valuesDeltaBest <- valuesDeltaOld
  cat ("\n")

  # other simulations to optimize the original network
  for(mcCounter in 1:mcNumber) {
    rnp <- sample(allowedTransit,1)
    if (rnp %in% duplPos) {
      duplChosen  <- substr(transitNames[rnp],5,5)
      rnp         <- which(grepl(paste ("dupl", duplChosen, sep = ""),
transitNames))
    }
    vectorPar[rnp] <- runif(1,minWeight,maxWeight)

    simulcoreOutput <- Simulcore(vectorPar, vectorProb, inhibIndex, matrixInhibit)
    cat ("\t", 1)
    if ((simulcoreOutput[kplaceN+1] == 0) || (simulcoreOutput[choicePlaceNumber] ==
0) || ((simulcoreOutput[] > stepsNumber) && (sample(2,1) !=1)) ||
((simulcoreOutput[kplaceN+1] < 0) && (sample(2,1) !=1))) {
      vectorPar  <- vectorParOld

    } else {
      simulcoreTot[1,] <- simulcoreOutput
      loopCheck        <- 0
```

```
      for (mutCounter in 1:cPMNnumber) {
        allIndexM      <- get(paste("allIndexM", mutCounter,sep=""))
        inhibIndexM    <- get(paste ("inhibIndexM",mutCounter,sep = ""))
        matrixInhibitM <- get(paste ("matrixInhibitM",mutCounter,sep = ""))

        vectorParM <- vectorPar
        vectorParM[allIndexM] = 0
        vectorProbM <- vectorProb
        vectorProbM[allIndexM] = 0

        simulcoreOutput <- Simulcore(vectorParM, vectorProbM, inhibIndexM,
matrixInhibitM)
        if ((simulcoreOutput[kplaceN+1] == 0) || ((simulcoreOutput[] > stepsNumber)
&& (sample(2,1) !=1)) || ((simulcoreOutput[kplaceN+1] < 0) && (sample(2,1) !=1))) {
          vectorPar  <- vectorParOld
          break
        }
        loopCheck <- mutCounter + 1
        simulcoreTot[loopCheck,] <- simulcoreOutput
        cat ("\t", loopCheck)
      }
      if (loopCheck == cPMNnumber+1) {
        valuesNew   <- simulcoreTot[, choicePlaceNumber]
        valuesDeltaVec   <- abs((valuesNew[c(2:(cPMNnumber+1))]/valuesNew[1]) -
ratioReal)
        valuesDeltaNew   <- sum(valuesDeltaVec^2)
        valuesDelta        <- valuesDeltaOld - valuesDeltaNew
        if  (valuesDeltaNew < valuesDeltaBest) {
          valuesDeltaBest <- valuesDeltaNew
          vectorParBest   <- vectorPar
        }
        if  (exp(invTemp*valuesDelta)> runif(1)) {
          vectorParOld   <- vectorPar
          valuesDeltaOld <- valuesDeltaNew
          countMcReal    <- countMcReal +1
          cat("\t !!", countMcReal,"\t")
        } else {vectorPar<- vectorParOld}
      }
    }
    cat ("\t", mcCounter, "\n")
  }
  cat("\n", totCounter, " rounds completed of ", totIteration, "\n")

  # final configuration and markings obtained is stored in external files; each
iteration of the script produces 2 files
  tempName1 <- sprintf("../Materials/MontecarloResults/kparameters%03d.txt",
totCounter)
  tempName11<- sprintf("../Materials/MontecarloResults/kparametersMIN%03d.txt",
totCounter)
  tempName2 <- sprintf("../Materials/MontecarloResults/markings%03d.txt",
totCounter)
  tempName3 <- sprintf("../Materials/MontecarloResults/Deltas%03d.txt", totCounter)
  write.table(t(vectorPar), tempName1 , quote = FALSE, row.names=FALSE,
col.names=FALSE, sep = "\t")
  write.table(t(vectorParBest), tempName11 , quote = FALSE, row.names=FALSE,
col.names=FALSE, sep = "\t")
  write.table(t(simulcoreTot[nrow(simulcoreTot),]), tempName2 , quote = FALSE,
row.names=FALSE, col.names=FALSE, sep = "\t")
  write.table(c(valuesDeltaBegin, valuesDeltaOld, valuesDeltaBest), tempName3 ,
quote = FALSE, row.names=FALSE, col.names=FALSE, sep = "\t")
}

tempName3 <- sprintf("../Materials/MontecarloResults/kparameters%03d.txt",
totIteration + 1)
write.table(t(vectorParStart), tempName3 , quote = FALSE, row.names=FALSE,
col.names=FALSE, sep = "\t")
save (totIteration, stepsNumber, choicePlaceNumber, file =
"../Materials/MontecarloResults/parameters.R", ascii = TRUE)
```

Pre-testing script

It is needed to choose the best set of parameters among those generated in the optimization.

```
## here it starts the "declaration" part"
scriptFile  <- "2. optGrowth 1.R"
interactive <- "Y"
if (!(file.exists(scriptFile))) {...}
source ("Subscripts/Variables.R")
load ("../Materials/MontecarloResults/parameters.R")
source ("Subscripts/simulcoreOpt.R")

## here it starts Montecarlo Simulation
totalDelta    <- vector(length=totIteration)
valuesDelta   <- vector(length=iterNumber)
simulcoreTot  <- vector(length = cPMNnumber+1)

for (totCounter in 1:totIteration) {
  cat ("starting set ", totCounter , ", iteration ")
  vectorPar   <-
readLines(sprintf("../Materials/MontecarloResults/kparameters%03d.txt", totCounter
))
  vectorPar   <- type.convert(unlist(strsplit(vectorPar, "\t")))
  vectorProb  <- tokenProduct*vectorPar

  for (iterCounter in 1:iterNumber) {
    simulcoreOutput <- Simulcore(vectorPar, vectorProb, inhibIndex, matrixInhibit)
    cat ("\t", 1)
    if (((simulcoreOutput[] > stepsNumber) && (sample(2,1) !=1)) ||
((simulcoreOutput[kplaceN+1] <= 0) && (sample(2,1) !=1))) {
      valuesDelta[iterCounter] = Inf

    } else {
      simulcoreTot[1] <- simulcoreOutput[choicePlaceNumber]
      for (mutCounter in 1:cPMNnumber) {
        allIndexM      <- get(paste ("allIndexM", mutCounter,sep=""))
        inhibIndexM    <- get(paste ("inhibIndexM",mutCounter,sep = ""))
        matrixInhibitM <- get(paste ("matrixInhibitM",mutCounter,sep = ""))

        vectorParM <- vectorPar
        vectorParM[allIndexM] = 0
        vectorProbM <- vectorProb
        vectorProbM[allIndexM] = 0

        simulcoreOutput <- Simulcore(vectorParM, vectorProbM, inhibIndexM,
matrixInhibitM)

        if (((simulcoreOutput[] > stepsNumber) && (sample(2,1) !=1)) ||
((simulcoreOutput[kplaceN+1] <= 0) && (sample(2,1) !=1))) {
          valuesDelta[iterCounter] = Inf
          break
        }
        simulcoreTot[mutCounter+1]  <- simulcoreOutput[choicePlaceNumber]
      }
      if (valuesDelta[iterCounter] != Inf) {
        valuesDeltaVec <- abs((simulcoreTot[c(2:(cPMNnumber+1))]/simulcoreTot[1]) -
ratioReal)
        valuesDelta[iterCounter] <- sum(valuesDeltaVec^2)
      }
    }
  }
  totalDelta[totCounter] = mean(valuesDelta[which(valuesDelta != Inf)])
  cat("\n", totCounter, " rounds completed of ", totIteration, "\n")
}
totalDelta[which(is.na(totalDelta))] = Inf
print (totalDelta)
bestResult = which.min(totalDelta)
```

```
cat ("\n the best k set is the ", bestResult, "°, which gives a final delta of ",
totalDelta[bestResult])

if (bestResult == length(totalDelta)) {
  cat ("\n Optimization failed!!")
} else {
  tempName  <- sprintf("../Materials/MontecarloResults/kparameters%03d.txt",
bestResult)
  file.copy (tempName, "../Materials/kparametersBEST.txt")
}
```

Optimization script v.2

It is employed during the TESTING of the optimization script.
It is identical to the Optimization script v.1, apart from the conditions checked; the previous and the new conditions checks may be found below.

```
OLD
if ((simulcoreOutput[kplaceN+1] == 0) || (simulcoreOutput[choicePlaceNumber] == 0)
|| ((simulcoreOutput[] > stepsNumber) && (sample(2,1) !=1)) ||
((simulcoreOutput[kplaceN+1] < 0) && (sample(2,1) !=1)))

NEW
if ((simulcoreOutput[choicePlaceNumber] == 0) || (simulcoreOutput[kplaceN+1] == 0)
|| ((simulcoreOutput[kplaceN+1] < 0) && (sample(2,1) !=1)))
```

Moreover, in this script, the variable "matrixTokens" is not read from the input file, rather it is randomly generated at each iteration of the optimization process

```
NEW:   matrixTokens[-choicePlaceNumber] <- round(runif((kplaceN-1), min=0,max=20))
```

Pre-testing script v.2

It is employed during the TESTING of the optimization script.
It is identical to the pre-testing script v.1, apart from the conditions checked; the previous and the new conditions checks may be found below.

```
OLD
if (((simulcoreOutput[] > stepsNumber) && (sample(2,1) !=1)) ||
((simulcoreOutput[kplaceN+1] <= 0) && (sample(2,1) !=1)))

NEW
if ((simulcoreOutput[kplaceN+1] <= 0) && (sample(2,1) !=1))
```

Simulation script

Simulation and analysis of the data are joined in this script, but the simulation core is located in another script

```
### HERE STARTS THE DECLARATION PART
scriptFile  <- "3. Simulator.R"
if (!(file.exists(scriptFile))) {...}
dir.create ("../Materials/Results", showWarnings = FALSE)
dir.create ("../Materials/Analysis", showWarnings = FALSE)
inputPath   <- "../Materials/Results/"
outputPath  <- "../Materials/Analysis/"
outputFile1 <- paste (outputPath, "finalmarkings.txt", sep = "")
outputFile2 <- paste (outputPath, "timeseries.txt", sep = "")
source ("Subscripts/Variables.R")
source ("Subscripts/SimulcoreMod.R")


### HERE STARTS THE SIMULATION PART
valMat = allMat   <- c(1:iterNumber)

if (iterNumber == 1) {
  tempOutput   <- Simulcore(vectorPar, vectorProb, inhibIndex, matrixInhibit)
  #outputFile3 <- paste (inputPath, "simulation1.txt", sep = "")
  #write.table (t(tempOutput), outputFile3, quote = FALSE, sep = "\t", row.names =
FALSE, col.names = c(placesNames,"Time"))
  simulcoreOutput1   <- tempOutput
  maxTime = minTime  <- abs(tempOutput[stepsNumber, kplaceN+1])

} else {
  maxTime <- 0
  for (iterCounter in 1:iterNumber) {   #the entire simulation is repeated
"iterNumber" number of times
    cat ("\n", iterCounter, "iterations started of ", iterNumber)
    tempOutput   <- Simulcore(vectorPar, vectorProb, inhibIndex, matrixInhibit)
    #outputFile3 <- paste (inputPath, "simulation", iterCounter,".txt", sep = "")
    #write.table (tempOutput, outputFile3, quote = FALSE, sep = "\t", row.names =
FALSE, col.names = c(placesNames,"Time"))
    assign (paste ("simulcoreOutput",iterCounter, sep=""),tempOutput)
    maxTime[iterCounter] <- abs(tempOutput[stepsNumber, kplaceN+1])
  }

  meanTime <- mean(maxTime)
  sdTime   <- sd(maxTime)
  #hist(maxTime, col="red")
  #abline   (v=c(meanTime, (meanTime - 2*sdTime), (meanTime + 2*sdTime)),col =
"blue")
  #abline   (v=c((meanTime - sdTime), (meanTime+sdTime), (meanTime + 3*sdTime),
(meanTime - 3*sdTime)),col="green")
  #readline("\n Distribution of the duration of each iteration; press enter to
continue")
  invMat   <- c(which(maxTime > (meanTime+sensSd*sdTime)), which(maxTime <
(meanTime -sensSd*sdTime)))
  bornDead <- which(maxTime == 0)
  if (length(bornDead) != 0) {
    invMat <- unique(c(valMat,invMat))
    cat ("\n One or more iteration ignored (the simulation reached a dead state at
the very beginning)")
  }
  if (length(invMat) != 0) {
    if (length(invMat) != length(allMat)) {
      valMat <- allMat[-invMat]
      } else {stop ("\n All the transitions have been ignored. No data available to
calculate a time series")}
  }
  minTime  <- min(maxTime[valMat])
  cat ("\n", 100*length(valMat)/length(allMat), "% of the iteration considered")
}
```

```
### HERE STARTS THE ANALYSIS PART
tableGlobal     <- matrix(ncol= kplaceN+1, nrow= length(valMat))    # Table to
summarize the results of all runs and the mean value of the runs (for each place)
colnames(tableGlobal)   <- c(placesNames, "Dead State?")
totalOutput    <- matrix(ncol= kplaceN, nrow=(spotsN+1))
totalOutput[,] <- 0

for (matCounter in valMat) {
  timeLine      <- seq(0.0, minTime, length.out = spotsN + 1)
  timeAxis      <- zoo(0, timeLine)
  nam2          <- paste("simulcoreOutput", matCounter, sep = "")
  assign("tempInput2",get(nam2))
  timeSeries      <- zoo(tempInput2[,c(1:kplaceN)], tempInput2[,(kplaceN+1)])
  #aggregate(timeSeries, index(timeSeries), mean)              # Only useful if a
transition happens so quickly that R cannot measure its duration

  mergedSeries  <- merge(timeSeries,timeAxis)
  mergedSeries[,c(1:kplaceN)] <- na.approx(mergedSeries[,c(1:kplaceN)], rule=2)
  timeIndex       <- which (index(mergedSeries) %in% index(timeAxis))
  simulcoreTemp <- as.matrix(mergedSeries[timeIndex,c(1:kplaceN)])
  totalOutput   <- totalOutput + simulcoreTemp[c(1:(spotsN+1)),]

  if (tempOutput[stepsNumber,kplaceN+1] < 0) {
    tableGlobal[which(valMat == matCounter),]<-
c(round(simulcoreTemp[spotsN+1,],3), "YES")
  } else  tableGlobal[which(valMat == matCounter),]<-
c(round(simulcoreTemp[spotsN+1,],3), "NO")
}


if (length(valMat) > 1) {
  totalOutput <- totalOutput/length(valMat)
  tableMean     <- totalOutput[spotsN+1,]
  tableGlobal   <- rbind(tableGlobal, c(round(tableMean,3), ""))
  rownames(tableGlobal)     <- c(valMat, "Mean")
}

if (choicePlaceName != "") {
  cat ("\n therefore, the (mean) value of ", choicePlaceName, " is ",
tableMean[choicePlaceNumber], "for each iteration")
  plot ((0:spotsN), totalOutput[,choicePlaceNumber], type="l")
} else  {
  matplot ((0:spotsN), totalOutput, type="l")
  legend('topright', placesNames , col=1:6, lty=1, bty='n', cex=.75)
}

totalOutput <-cbind(totalOutput, timeLine)
write.table (tableGlobal, outputFile1, quote = FALSE, sep = "\t",row.names = FALSE,
)  #print the summary table and the mean value of the selected place
write.table (round(totalOutput,3), outputFile2, quote = FALSE, sep = "\t",row.names
= FALSE, )  #print the summary table and the mean value of the selected place
```

Simulation script – on GitHub

Simulation script as found on GitHub; it does not contain the analysis part, but it does not require any additional subscript for reading the input files and performing simulations.

```
### HERE STARTS THE INPUT PART
## General input
  library('rjson')            # rjson package is loaded
  stepsNumber   <- 3000       # Number of steps of each simulation
  iterNumber    <- 100        # Number of times each simulation is repeated
  inputPath     <- ""         # by default, the input file is read from the w.d.
  outputPath    <- "Results/" # by default, output files are in this folder
  inputFileName <- "merge_matrices.txt"
  inputFile     <- paste (inputPath, inputFileName, sep = "")
  inputData     <- fromJSON(paste(readLines(inputFile), collapse=""))

  matrixTokens  <- inputData$marking
  transitNames  <- inputData$tnames        # Name of transitions
  placesNames   <- inputData$pnames        # Name of places
  kplaceN       <- length(placesNames)     # Number of places
  ktransitN     <- length(transitNames)    # Number of Transitions
  vectorPar     <- inputData$k             # Mass Action parameters

  matrixInhibit <- do.call(rbind, inputData$inhib)
  inhibIndex      <- which(matrixInhibit>=1, arr.ind=TRUE)
  matrixInward  <- do.call(rbind, inputData$post)
  matrixOutward <- -1*(do.call(rbind, inputData$pre))
  matrixDelta   <- matrixInward + matrixOutward
  colnames(matrixInhibit) <- colnames(matrixInward) <- colnames(matrixOutward) <-
colnames(matrixDelta) <- placesNames

## Ancillary function
  tokenProduct   <- vector(length=ktransitN)
  prova = which(matrixOutward != 0, arr.ind = TRUE)
  prova3 = which(matrixInward !=0, arr.ind = TRUE)
  temp = list()
  temp2 = list()
  temp3 = list()
  for(i in 1:ktransitN){
    # the product of the number of tokens in the input places for each transition
    temp[[i]]  <- prova[(prova[,1] == i),2]
    tokt   <- matrixTokens[temp[[i]]]
    tot    <- prod(tokt)
    tokenProduct[i] <-tot
    temp2[[i]] <- unique(c(prova[(prova[,1] == i),2], prova3[(prova3[,1] == i),2]))
    if (length(temp2[[i]]) == 0) {
      temp3[[i]] <- unique(c(prova[(prova[,1] == i),2], prova3[(prova3[,1] ==
i),2]))
    } else if (length(temp2[[i]]) == 1) {
      temp3[[i]] <- unique(which(matrixOutward[,temp2[[i]]] != 0, arr.ind = TRUE))
    } else {
      temp3[[i]] <- unique(which(matrixOutward[,temp2[[i]]] != 0, arr.ind =
TRUE)[,1])
    }
  }
  vectorProb   <- tokenProduct*vectorPar        # probability vector completed


### HERE STARTS THE SIMULATION FUNCTION
Simulcore <- function() {
  matrixMatrix     <- matrix(ncol =kplaceN+1, nrow= stepsNumber)
  matrixMatrix[1,] <- c(matrixTokens,0)
  totTime          <- 0

  # each step is repeated for stepsNumber times (x=1 is the starting condition)
  for (matrixRow in 1:stepsNumber) {
    vectorTrans  <- 1:ktransitN
    vectorTime   <- 0
```

```
    for (x in 1:ktransitN) {
      if (vectorProb[x] <= 0) {vectorTime[x] = Inf}
      else {vectorTime[x] <-rexp(1,vectorProb[x])}
    }

    repeat {
      # the shortest transition is chosen and its time recorded
      parTime   <- min(vectorTime)
      if (parTime == Inf) {
        matrixMatrix[matrixRow,] <- c(matrixTokens, -totTime)
        return(matrixMatrix[c(1:matrixRow),])
      }
      chance <- which(vectorTime == parTime)
      if (length(chance) >1) {chance    <- sample(chance,1)}
      rn       <- vectorTrans[chance]

      if (length(inhibIndex) == 0) {
        # option 1: there is no inibitory edge in the network
        if (all(matrixTokens >= -matrixOutward[rn,])) {
          matrixTokens <- matrixTokens + matrixDelta[rn,]
          totTime      <- totTime + parTime
         matrixMatrix[matrixRow,] <- c(matrixTokens, totTime)
          break
        }
      } else {
        # option 2: there are some inibitory edges in the network
        indexIndex <- inhibIndex[which(inhibIndex[,1] ==rn),2]
        if ((length(indexIndex) == 0) || (all(matrixTokens[indexIndex] <
matrixInhibit[rn,indexIndex]))) {
          if (all(matrixTokens >= -matrixOutward[rn,])) {
            matrixTokens <- matrixTokens + matrixDelta[rn,]
            totTime      <- totTime + parTime
            matrixMatrix[matrixRow,] <- c(matrixTokens, totTime)
            break
          }
        }
      }

      vectorTrans <- vectorTrans[!vectorTrans == rn]
      vectorTime  <- vectorTime[-chance]
      if    (length(vectorTrans) == 0){
        cat("\t  dead state!!")
        matrixMatrix[matrixRow,] <- c(matrixTokens, -totTime)
        return(matrixMatrix[c(1:matrixRow),])
      }
    }
    importantTemp <- temp3[[rn]]
    important <- importantTemp[which(vectorPar[importantTemp] !=0)]
    for (i in important) {
      vectorProb[i] <-prod(matrixTokens[temp[[i]]]) * vectorPar[i]
    }
  }
  return(matrixMatrix[c(1:matrixRow),])
}

### HERE STARTS THE SIMULATION PART
dir.create ("Results", showWarnings = FALSE)

for (iterCounter in 1:iterNumber) {
  tempOutput    <- Simulcore()
  outputFile3  <- paste (outputPath, "simulation", iterCounter,".txt", sep = "")
  if (is.matrix(tempOutput)) {
    write.table  (tempOutput, outputFile3, quote = FALSE, sep = "\t", row.names =
FALSE, col.names = c(placesNames,"Time"))
  } else {write.table  (t(tempOutput), outputFile3, quote = FALSE, sep = "\t",
row.names = FALSE, col.names = c(placesNames,"Time"))}
  cat ("\n", iterCounter, "iterations completed of ", iterNumber)
}
```

Analysis script – on GitHub

Analysis script as found on GitHub; it does not contain the simulation part, but it does not require any additional subscript for reading the input files and performing simulations.

```
### HERE STARTS THE INPUT PART
library('zoo')              # zoo package is loaded; it must be installed
before running the script

spotsN      <- 100          # Number of sampling points in which you
interpolate from all the time series to build a new averaged one
sensSd      <- 2            # Simulations whose duration is farther than
"sensSd" Standard Deviation from the Mean won't be considered
choicePlaceName  <- ""          # The places you're interested in; as
default, all the places are considered
inputPath   <- "Results/"   # by default, input files are read from this
folder inside the working directory
outputPath  <- "Analysis/"  # by default, output files will be placed in
this folder inside the working directory
outputFile1 <- paste (outputPath, "finalmarkings.txt", sep = "")
outputFile2 <- paste (outputPath, "timeseries.txt", sep = "")
fileNames   <- list.files(inputPath)
iterNumber  <- length(fileNames)


# each matrix retrieved from a file is assigned to a specific variable and
its final time is stored in a vector
if (iterNumber == 1) {
  inputFile <- paste(inputPath, fileNames, sep = "")
  tempInput <- read.table(inputFile, header = TRUE, sep="\t")
  assign("simulcoreOutput1",tempInput)
  maxTime = minTime  <- abs(tempInput[nrow(tempInput),ncol(tempInput)])
  valMat    <- 1

# the script changes a little when dealing with more than one input files
(a for loop is required)
} else {
  maxTime     <- 0
  for (iterCounter in 1:iterNumber) {
    inputFile <- paste(inputPath,"simulation", iterCounter,".txt", sep =
"")
    tempInput <- read.table(inputFile, header = TRUE, sep="\t")
    nam       <- paste ("simulcoreOutput",iterCounter, sep="")
    assign(nam,tempInput)
    maxTime[iterCounter]  <-
abs(tempInput[nrow(tempInput),ncol(tempInput)])
  }

  # mean value and stardard deviation of the simulation durations are
calculated
  meanTime <- mean(maxTime)
  sdTime   <- sd(maxTime)
  # simulation durations are plotted in a graph indicating mean value and
1x, 2x, 3x sd distances from the mean
  hist(maxTime, col="red")
  abline (v=c(meanTime, (meanTime - 2*sdTime), (meanTime + 2*sdTime)),col =
"blue")
  abline (v=c((meanTime - sdTime), (meanTime+sdTime), (meanTime +
3*sdTime), (meanTime - 3*sdTime)),col="green")
  readline("\n Distribution of the duration of each iteration; press enter
to continue")
```

```
  valMat = allMat   <- c(1:iterNumber)

  # simulation that lasted too differently from the mean are flagged; if a
simulation has been flagged, it is removed from the pool
  invMat   <- c(which(maxTime > (meanTime+sensSd*sdTime)), which(maxTime <
(meanTime -sensSd*sdTime)))
  # other simulations are removed if the reached a dead state in the very
first stage of the simulation
  bornDead <- which(maxTime == 0)
  if (length(bornDead) != 0) {
    invMat <- unique(c(valMat,invMat))
    cat ("\n One or more iteration ignored (the simulation reached a dead
state at the very beginning)")
  }
  if (length(invMat) != 0) {
    if (length(invMat) != length(allMat)) {
      valMat <- allMat[-invMat]
      } else {stop ("\n All the transitions have been ignored. No data
available to calculate a time series")}
  }
  #the shortest simulation determines the duration of the mean time series
  minTime  <- min(maxTime[valMat])
  cat (100*length(invMat)/length(allMat), "% of the iteration ignored")
}

# this part is needed if you are interested in a specific place and have
specified it at the beginning of the script
kplaceN          <- ncol(simulcoreOutput1) - 1
placesNames      <- colnames(simulcoreOutput1)[-(kplaceN+1)]
if (all(choicePlaceName == "")) {
  choicePlaceNumber <- c(1:kplaceN)
} else  choicePlaceNumber <- which (colnames(simulcoreOutput1) ==
choicePlaceName)


###################################
### HERE STARTS THE ANALYSIS PART
###################################

dir.create ("Analysis", showWarnings = FALSE)

tableGlobal    <- matrix(ncol= kplaceN+1, nrow= length(valMat)) # Table to
summarize the results of each simulation (tokens in each place)
colnames(tableGlobal)   <- c(placesNames, "Dead State?")
totalOutput    <- matrix(ncol= kplaceN, nrow=(spotsN+1))        # Table to
mean amount of tokens at each time point, i.e. the mean of all the
simulations
totalOutput[,] <- 0

for (matCounter in valMat) {
  timeLine      <- seq(0.0, minTime, length.out = spotsN + 1)   # time line
created by subsetting the selected total duration for the chose number of
time points.
  timeAxis      <- zoo(0, timeLine)
  nam2          <- paste("simulcoreOutput", matCounter, sep = "")
  assign("tempInput2",get(nam2))
  timeSeries      <- zoo(tempInput2[,c(1:kplaceN)],
tempInput2[,(kplaceN+1)])  #the original irregular time series is converted
into a zoo object
```

```
  #aggregate(timeSeries, index(timeSeries), mean)
# Extremely slow step; to be used only if you suspect that a transition
happens so quickly that R cannot measure its duration

  # the original irregular time series is interpolated at specific time
points, thus creating a regular time series
  mergedSeries  <- merge(timeSeries,timeAxis)
  mergedSeries[,c(1:kplaceN)] <- na.approx(mergedSeries[,c(1:kplaceN)],
rule=2)
  timeIndex       <- which (index(mergedSeries) %in% index(timeAxis))
  simulcoreTemp <- as.matrix(mergedSeries[timeIndex,c(1:kplaceN)])
  # the regular time series can be summed to the others because it has been
created by interpolating in the same time points
  totalOutput   <- totalOutput + simulcoreTemp[c(1:(spotsN+1)),]

  # the results of each simulation, modified during the interpolation
phase, are recorded into the summarizing table
  if (tempInput[nrow(tempInput),kplaceN+1] < 0) {
    tableGlobal[which(valMat == matCounter),]<-
c(round(simulcoreTemp[spotsN+1,],3), "YES")
  } else  tableGlobal[which(valMat == matCounter),]<-
c(round(simulcoreTemp[spotsN+1,],3), "NO")
}

# the actual mean is calculated in the mean matrix by diving the total
amount of tokens for the number of simulations
if (length(valMat) > 1) { # there's no need to do that if only one
simulation has been analysed.
  totalOutput <- totalOutput/length(valMat)
  tableMean     <- totalOutput[spotsN+1,]
  tableGlobal   <- rbind(tableGlobal, c(round(tableMean,3), ""))
  rownames(tableGlobal)     <- c(valMat, "Mean")
}

# a graph is plotted, showing the amount of tokens during all the
simulation; a specific place or all the places are plotted according to the
first settings.
if (choicePlaceName != "") {
  cat ("\n therefore, the (mean) value of ", choicePlaceName, " is ",
tableMean[choicePlaceNumber], "for each iteration")
  plot ((0:spotsN), totalOutput[,choicePlaceNumber], type="l")
} else  {
  matplot ((0:spotsN), totalOutput, type="l")
  legend('topright', placesNames , col=1:6, lty=1, bty='n', cex=.75)
}

# the mean table and the summarizing table are exported into two text files
totalOutput <-cbind(totalOutput, timeLine)
write.table (tableGlobal, outputFile1, quote = FALSE, sep = "\t",row.names
= FALSE, )  #print the summary table and the mean value of the selected
place
write.table (round(totalOutput,3), outputFile2, quote = FALSE, sep =
"\t",row.names = FALSE, )  #print the summary table and the mean value of
the selected place
```

5.3. Protocols

5.3.1. Transformation of yeast cells

Culture

- Distribute 100µl of YPD in a 96-multi well plate (U bottom)
- Add 10µl of a stationary phase culture in each well
- Incubate overnight at 30°C *without agitation* (to avoid cross-contamination) under a wet atmosphere (for instance, humidity supplied by a water bath)

Transformation

- Centrifuge cells 5 minutes at 3000rpm *(remember to balance the centrifuge!)*
- Remove the supernatant by quickly flipping the plate (over a sink)
- Resuspend the pellet by smoothly vortexing the plate (there is enough liquid left to do so).

- Add 100 µl in each well of the following solution (note that Carrier DNA must be denatured 5 minutes at 95°C and then immediately put on ice).

Substance	Final concentration	Final volume = 50 ml	Final volume = for 25 ml
Lithium Acetate	0,2 M	10 ml of 1 M	5 ml of 1 M
PEG 3350	40%	40 ml of 50%	20 ml of 50%
DTT	0.1M	5 ml of 1 M	2.5 ml of 1 M
Carrier DNA	5%	2.5 mL of 10mg/ml	1.250 mL of 10mg/ml

-Add 1 µg of the replicative plasmid to each well (the volume in µl must be calculated using the concentration read using the NanoDrop machine)

NOTE: You may want to test the optimal concentration for your plasmid; it should not be needed to use more than 1µg, because the rate between transformed cells and µg of plasmid quickly reaches a plateau.

Incubation

- *Carefully mix* by pipetting.
- Incubate 30 minutes at 45°C (e.g. using a water bath)
- Make 10µl drops on the appropriate media (we use large square plates) *(2-4 repeats)*

NOTE: This protocol can be applied for transforming many strains in the same time, but the final yield is not excellent; on the other hand, when dealing with a few strains, other protocols may be employed in order to obtain higher yields (i.e. number of transformed cells).

5.3.2. Transformation of *Escherichia coli* cells

Protocol

1. Thaw competent cells **on ice**. Once thawed they will start losing viability and cannot be reused.
2. Pre-chill all Eppendorf tubes on ice *(it is important!)* and thaw plasmid DNA on ice.
3. Prepare Eppendorf tubes with 50ul competent cells, and then add 5ul plasmid DNA.
4. Mix by flicking *(do not pipette nor vortex)*
5. Incubate on ice for 30 min
6. Heat shock 42°C 30 sec, no shaking
7. Add 500ul LB (do not select for the plasmid because the cells have not had time to express the resistance marker)
8. Incubate 37°C, 750 rpm, 60 min. Do not extend the incubation time of the plate if you do not see colonies: if any correct colonies are going to form, they will appear by the morning.
9. Plate 100-150ul on selective medium
10. Incubate 37°C overnight

Notes

- We prepare a large batch of competent cells using the Rubidium chloride method, and then rapidly freeze the cells in 200ul aliquots for storage at -80°C. Only take as many tubes as you need from the freezer *(competent cells are fragile)*
- It is important not to incubate the plate for too long. This is particularly important when using Amp as a marker, because it is degraded by the transformed cells and satellite colonies will begin to form around them.
    - Therefore, plates must be taken from the incubator and checked in the morning, cooled to room temperature and then parafilmed and put in the fridge.
    - Similarly, if a long-term storage is required, then a glycerol stock must be prepared, because transformed *E. coli* cannot be reliably stored on plates for long.
- If a very low concentration of plasmid DNA is employed, or the transformation is supposed to be inefficient, SOC medium can be used instead of LB for step 7.
    - SOC (a version of Super Optimal Broth with added glucose) is more complicated to prepare than LB but can increase transformation efficiency; once prepared, it must be stored in the fridge to help preventing contamination.

5.3.3. Plasmid extraction

E.Z.N.A.® Endo-Free Plasmid DNA Mini Kit II Protocol - Spin Protocol. This protocol is designed to isolate plasmid DNA from *E. coli* grown in an overnight 1-5 mL LB culture.

Materials and Equipment to be supplied by user:

- 100% ethanol
- Nuclease-free 1.5 mL or 2 mL microcentrifuge tubes
- Culture tubes and sterile deionized water
- Water bath or incubator capable of 70°C
- Microcentrifuge capable of at least 13,000 x g

Kit materials

- Solution I with RNase A (RNase must be added to the solution before starting the experiment)
- Solution II and Solution III
- Equilibration buffer
- HiBind® DNA Mini Columns
- DNA wash buffer (it must be diluted with 100% ethanol prior to use)

Protocol

1. Isolate a single colony from a freshly streaked selective plate, and inoculate a culture of 1- 5 mL LB medium containing the appropriate selective antibiotic. Incubate for ~12-16 hours at 37°C with vigorous shaking (~300 rpm). Use a 10-20 mL culture tube or a flask with a volume of at least 4 times the volume of the culture.
2. Centrifugation at 5,000g for 10 minutes at room temperature.
3. Decant or aspirate the medium and discard.

4. Add 250 µL Solution I/RNase A. Vortex or pipet up and down to mix thoroughly. Complete suspension of cell pellet is vital for obtaining good yields.
5. Transfer suspension into a new 2 mL microcentrifuge tube.
6. Add 250 µL Solution II. Invert and gently rotate the tube several times to obtain a clear lysate. A 2-3 minute incubation may be necessary. Note: Avoid vigorous mixing, as this will shear chromosomal DNA and lower plasmid purity.
7. Add 350 µL Solution III. Gently invert several times until a flocculent white precipitate forms.
8. Centrifuge at maximum speed (≥13,000 x g) for 10 minutes. A compact white pellet will form.

9. Insert a HiBind® DNA Mini Column II into a 2 mL Collection Tube.

Optional Protocol for Column Equilibration:

1. Add 100 µL of equilibration buffer the HiBind® DNA Mini Column.
2. Centrifuge at maximum speed for 30-60 seconds.
3. Discard the filtrate and reuse the collection tube.

10. Transfer the cleared supernatant from Step 8 by CAREFULLY aspirating it into the HiBind® DNA Mini Column. Be careful not to disturb the pellet and that no cellular debris is transferred to the HiBind® DNA Mini Column
11. Centrifuge at maximum speed for 1 minute; discard the filtrate and reuse the collection tube.

12. Add 500 µL HB Buffer.
13. Centrifuge at maximum speed for 1 minute; discard the filtrate and reuse collection tube.
14. Add 700 µL DNA Wash Buffer. Note: DNA Wash Buffer.
15. Centrifuge at maximum speed for 1 minute; discard the filtrate and reuse the collection tube. Optional: repeat steps 14-15 for a second DNA Wash Buffer wash step.

16. Centrifuge the empty HiBind® DNA Mini Column for 2 minutes at maximum speed to dry the column matrix. Note: It is important to dry the HiBind® DNA Mini Column matrix before elution because residual ethanol may interfere with downstream applications.
17. Transfer the HiBind® DNA Mini Column to a clean 1.5 mL microcentrifuge tube.
18. Add 40µL sterile deionized water directly to the centre of the column membrane. Note: the efficiency of eluting DNA from the HiBind® DNA Mini Column is dependent on pH. If using sterile deionized water, make sure that the pH is around 8.5.
19. Let sit at room temperature for 1 minute, then centrifuge at maximum speed for 1 minute. Optional: Repeating steps 18-19 will yield any residual DNA, though at a lower concentration.

20. Store plasmid DNA at -20°C

5.4. Gene lists

1. List of the viable mutant strains employed in the first experiment, i.e. measurement of the growth rate of slow growing yeast colonies.

Systematic name	Standard name	Connections	Network size	SHELF	PLATE	R	C
YER044C	ERG28	45	117	57	312	D	12
YEL044W	IES6	21	189	57	312	A	3
YLR403W	SFP1	34	139	60	320	C	1
YGR006W	PRP18	15	113	61	322	C	2
YDR300C	PRO1	22	65	62	327	C	8
YLR435W	TSR2	23	197	64	333	E	11
YNL054W	VAC7	11	110	66	338	B	2
YGR036C	CAX4	45	225	66	337	D	9
YJL184W	GON7	30	67	67	342	B	10
YJL189W	RPL39	20	40	67	342	B	11
YCR047C	BUD23	11	48	67	342	G	5
YNL133C	FYV6	11	63	67	341	A	12
YPR067W	ISA2	16	41	68	344	F	11
YLR382C	NAM2	12	75	68	343	A	10
YPL268W	PLC1	15	198	68	344	E	10
YNL138W	SRV2	85	268	68	343	E	6
YGR272C	EFG1	15	49 (59)	69	348	E	12
YLR244C	MAP1	18	93	71	372	A	9
YPL050C	MNN9	62	116	71	372	B	10
YLR240W	VPS34	20	82	71	372	B	8
YGL038C	OCH1	20	82	71	372	C	5
YOR202W	HIS3	//	//	59	318	E	6

“Connections” is the number of interacting partners of a gene retrieved in YeastMine. “Network size”, instead, is the number of all the physical interactions among the genes connected to the query gene; it is calculated by retrieving the number of total interactions in YeastMine and then removing duplicate interactions.

Shelf, Plate, R (Row) and C (Column) refer to the location of the strains in the -80°C freezer. It may be noticed that this list has been sorted according to the location of the strains rather than the alphabetical order of the genes; it has been made this way in order to quicken the step of strains retrieval from the fridge, thus reducing the damages to the frozen samples.

2. List of genes involved in the ERAD and the UPR (retrieved from KEGG pathways).

They might not exactly correspond to the places in the network due to modelling reasons.

Systematic Name	KEGG name
YDL072C	Bap31
YJL034W	BiP
YAL058W	CNX
YMR264W	Cue1
YBR201W	Derlin
YDR411C	Derlin
YIL030C	Doa10
YMR276W	DSK2
YJR007W	eIF2a
YJR131W	ERMan I
YLR057W	ERMan I
YGL027C	Glc I
YBR229C	Glc II
YOL013C	Hrd1
YLR207W	Hrd3
YJL073W	Hsp40
YMR214W	Hsp40
YMR161W	Hsp40*
YNL064C	Hsp40*
YAL005C	Hsp70
YBL075C	Hsp70
YER103W	Hsp70
YLL024C	Hsp70
YNL209W	Hsp70
YMR186W	Hsp90
YPL240C	Hsp90
YHR079C	IRE1
YHR204W	MNL1
YKL073W	NEF
YOL031C	NEF
YBR169C	NEF*
YIL016W	NEF*
YPL106C	NEF*
YBR170C	Npl4
YDR057W	OS9
YDL232W	OST
YEL002C	OST
YGL022W	OST

Systematic Name	KEGG name
YGL226C-A	OST
YJL002C	OST
YML019W	OST
YMR149W	OST
YOR085W	OST
YOR103C	OST
YDL126C	p97
YCL043C	PDI
YDR518W	PDI
YIL005W	PDI
YOR288C	PDI
YDR283C	PERK
YPL096W	Png1
YEL037C	RAD23
YPL218W	SAR1
YDL195W	Sec13/31
YLR208W	Sec13/31
YIL109C	Sec23/24
YPR181C	Sec23/24
YBR283C	Sec61
YDR086C	Sec61
YER019C-A	Sec61
YER087C-B	Sec61
YLR378C	Sec61
YPL094C	Sec62/63
YOR254C	Sec62/63
YBR072W	sHSF
YDR171W	sHSF
YER100W	Ubc6/7
YMR022W	Ubc6/7
YBR082C	UbcH5
YDR059C	UbcH5
YBL058W	Ubx
YML013W	Ubx2
YGR048W	Ufd1
YDL190C	Ufd2
YOR336W	UGGT

3. List of the mutant strains employed in the second experiment, i.e. fluorescence measurement using a cytoplasmic GFP as reporter.

Systematic name	KEGG Name	F. Ratio	Set
YDL072C	Bap31	1.071438	TR
YAL058W	CNX	1.450026	TE
YBR201W	Derlin	1.271776	TR
YDR411C	Derlin	1.372556	TE
YIL030C	Doa10	1.146694	TR
YMR276W	DSK2	0.680619	TE
YJR131W	ERMan I	1.016127	TR
YLR057W	ERMan I	0.96801	TE
YBR229C	Gcl II	1.133449	TR
YGL027C	Glc I	1.205143	TE
YOR202W	HIS3	1	TR
YOL013C	Hrd1	1.320354	TE
YLR207W	Hrd3	1.181487	TR
YJL073W	Hsp40	0.997621	TE
YMR214W	Hsp40	1.07017	TR
YMR161W	Hsp40	1.173455	TE
YNL064C	Hsp40		
YAL005C	Hsp70	1.063484	TR
YBL075C	Hsp70	0.903656	TE
YER103W	Hsp70	1.209884	TR
YLL024C	Hsp70	1.125516	TE
YMR186W	Hsp90	1.213468	TR
YPL240C	Hsp90	1.223061	TE
YHR079C	IRE1	0.992711	TR
YHR204W	MNL1	1.062126	TE
YKL073W	NEF	0.927122	TR

Systematic name	KEGG Name	F. Ratio	Set
YOL031C	NEF	1.137503	TE
YBR169C	NEF	2.0225	TR
YIL016W	NEF	1.210837	TE
YPL106C	NEF		
YBR170C	Npl4	1.308627	TR
YDR057W	OS9	0.976991	TE
YDL232W	OST	0.861588	TR
YGL226C-A	OST	1.088746	TE
YML019W	OST	1.091643	TR
YOR085W	OST	0.957667	TE
YDR518W	PDI	0.963519	TR
YIL005W	PDI	0.66542	TE
YOR288C	PDI	1.076834	TR
YDR283C	PERK	0.935403	TE
YPL096W	Png1	1.011833	TR
YEL037C	RAD23	1.122692	TE
YBR283C	Sec61	1.234497	TR
YER019C-A	Sec61	0.836368	TE
YBR072W	sHSF	1.159579	TR
YDR171W	sHSF	1.332989	TE
YMR022W	Ubc6/7	0.586871	TR
YBR082C	UbcH5	1.41485	TE
YDR059C	UbcH5	1.146073	TR
YBL058W	Ubx		
YML013W	Ubx2	0.967074	TE
YDL190C	Ufd2	1.139287	TR

The table also shows the calculated fluorescence ratio when available, and the final subsets (TE = testing, TR = training). It may be noticed that more genes have the same name in KEGG: in some cases, that means that any of them could perform the activity reported in KEGG pathways; in other cases, that means that all the genes associated to a certain name are required for that action to take place. Therefore, retrieving the list from KEGG was not enough, since each place in those pathways had to be investigated in details.

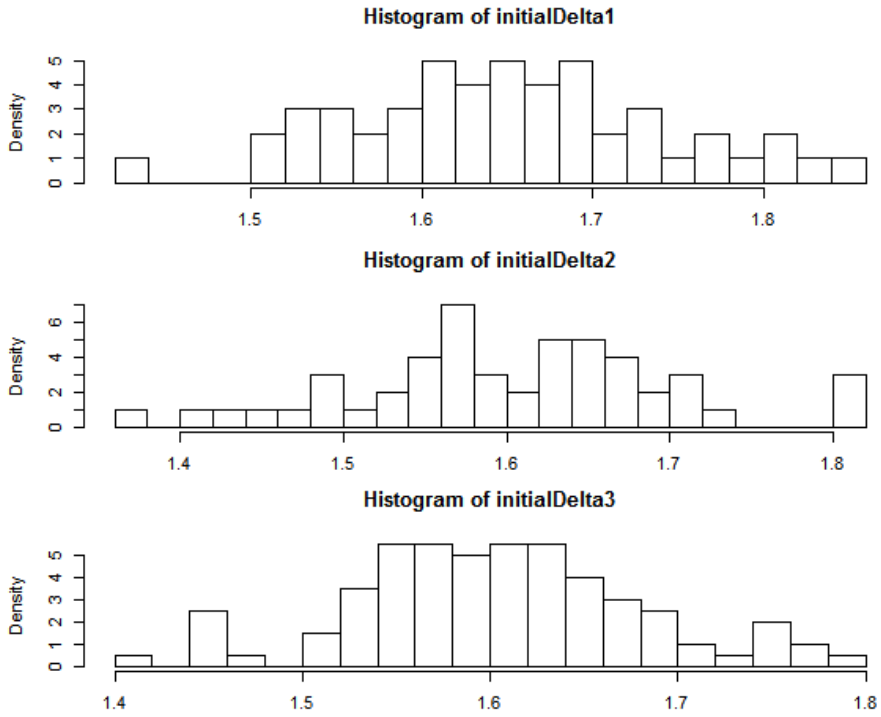
4. List of the mutant strains employed in one of the two optimization processes.

Systematic Name	Name in Network	F. Ratio	Set
YAL058W	CNX	0.795919	TE
YMR264W	Cue1	2.690238	TR
YBR201W	DER1	0.983099	TR
YDR411C	DFM1	1.028758	TE
YMR276W	DSK2	1.256507	TR
YIL005W	EPS1	1.212764	TR
YDR518W	EUG1	1.045429	TE
YMR161W	HLJ1	1.593492	TE
YOL013C	HRD1	1.027215	TE
YLR207W	HRD3	1	TE
YMR186W	HSC82	1.610198	TR
YBR072W	HSP26	0.941434	TE
YPL240C	HSP82	1.812730	TR
YHR079C	IRE1p	0.992055	TE
YJL073W	JEM1	1	TR
YKL073W	LHS1	0.986476	TE
YHR204W	MNL1	1.092884	TR
YOR288C	MPD1	1.594510	TE
YBR170C	Npl4	0.832199	TR
YOR085W	OST3	0.942065	TR
YDL232W	OST4	1.002485	TE

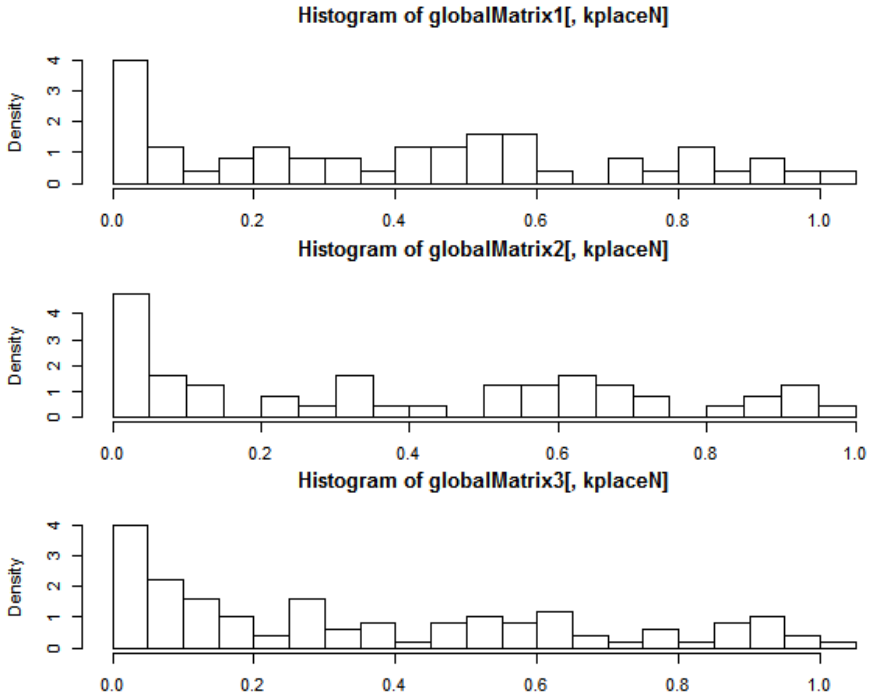
Systematic Name	Name in Network	F. Ratio	Set
YGL226C-A	OST5	1.669784	TR
YML019W	OST6	1.849899	TE
YPL096W	PNG1	0.496856	TE
YMR022W	UBC7	1.100814	TR
YEL037C	RAD23	0.943032	TE
YER019C-A	SBH2	1.035770	TR
YMR214W	SCJ1	1.209995	TR
YML013W	Ubx2	0.914934	TR
YOL031C	SIL1	5.093371	TR
YIL016W	SNL1	0.683595	TE
YPL106C	SSE1	3.132344	TR
YBR169C	SSE2	2.978349	TE
YBR283C	SSH1	0.854054	TR
YIL030C	Doa10	18.70455	TE
YBR082C	UBC4	2.575009	TE
YDR059C	UBC5	0.854729	TR
YDL190C	UFD2	1.394445	TR
YNL064C	YDJ1	2.258177	TE
YDR057W	OS9	7.541905	TR
YBL058W	Ubx1	1.705164	TE
YDR171W	HSP42	1.149413	TR
YLR057W	MNL2	0.916348	TE

It derives from the supplementary materials of Jonikas et al. (2009), i.e. the measurement of the fluorescence levels induced upon activation of the UPR; it only contains those genes that were already shown in my whole-cell network. The table also shows the fluorescence ratio, and the final subsets.

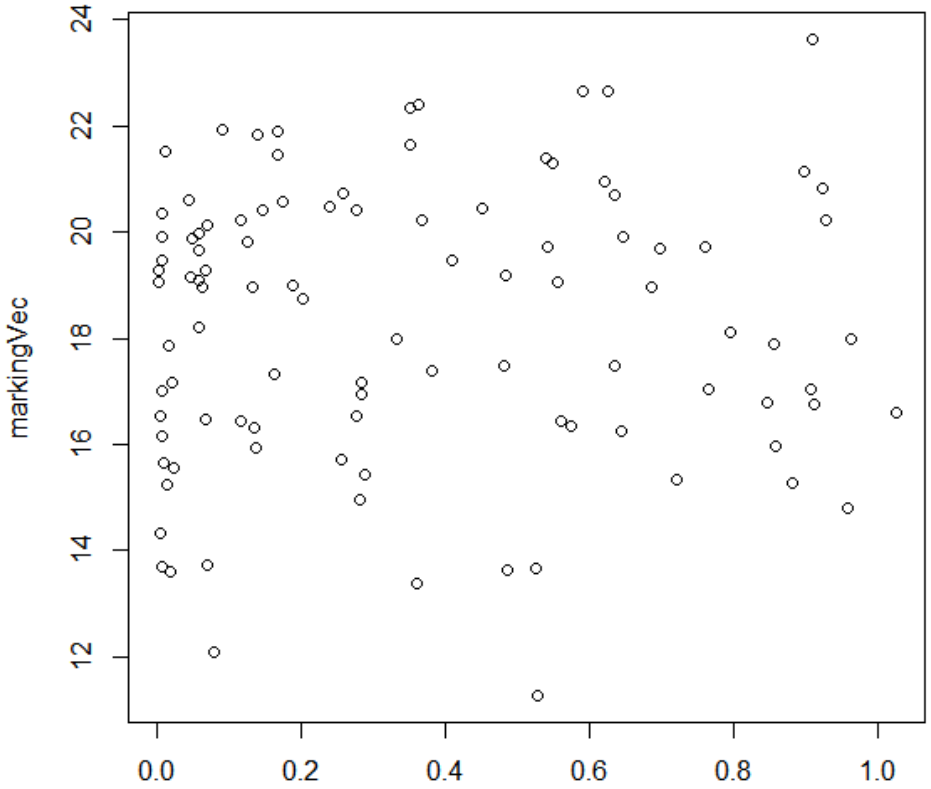
5.5. Analyses from the testing the optimization script



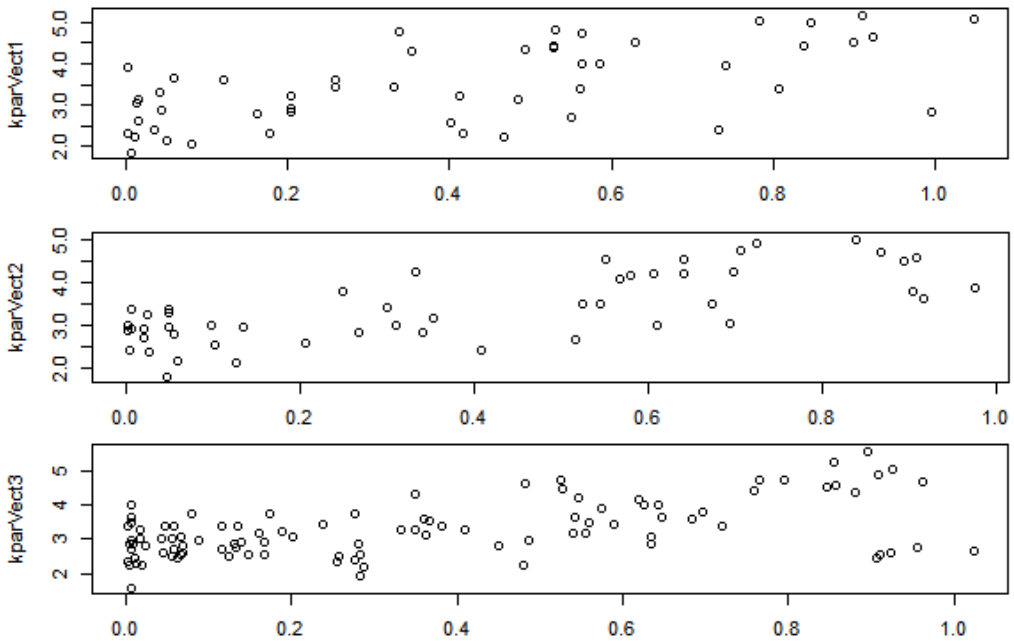
Distribution of starting delta values: plots obtained using 100 random markings (lower picture) and using 50 iteration starting from the same marking (middle and higher picture)



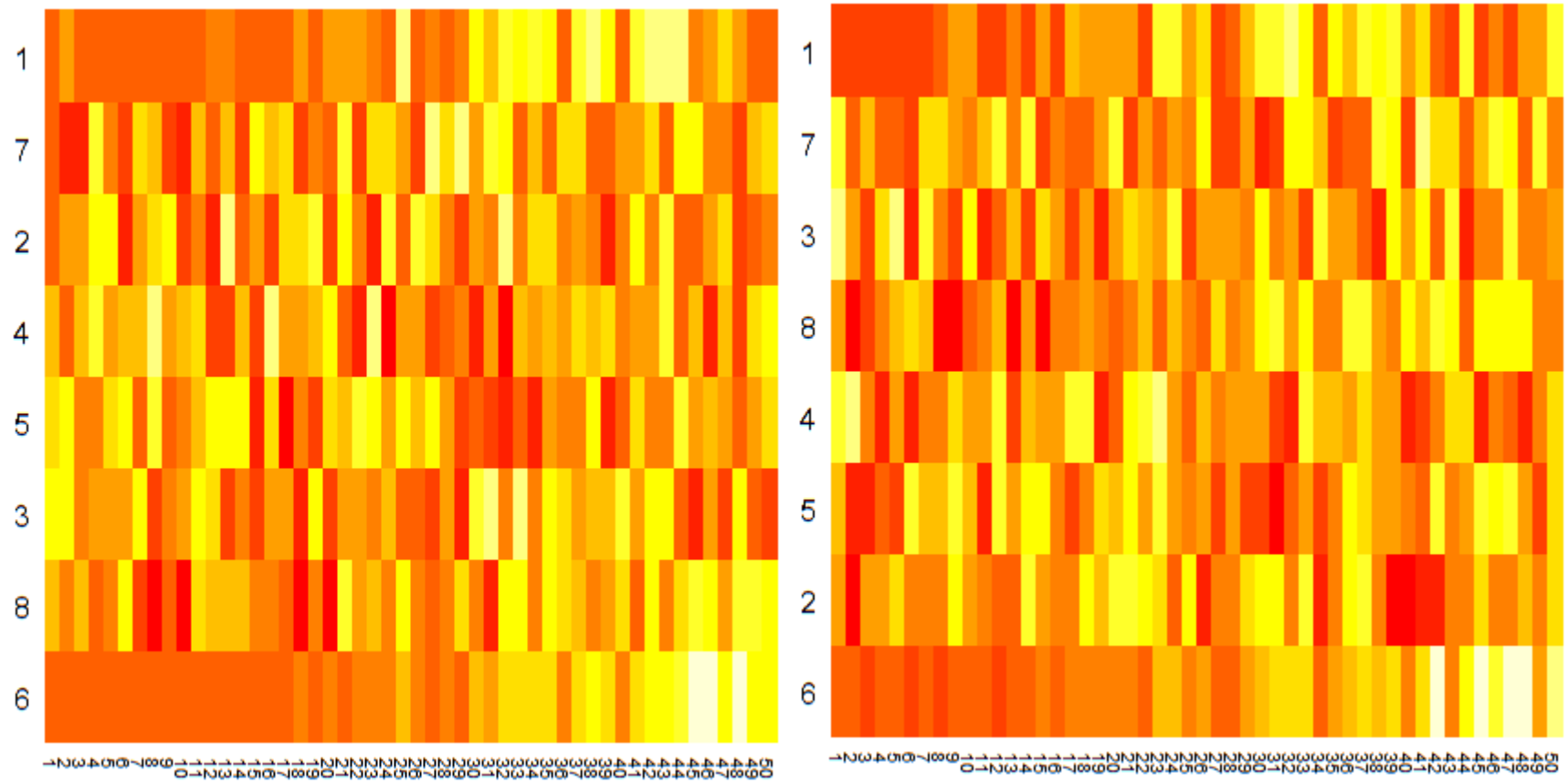
Distribution of final delta values: plots obtained using 100 random markings (higher picture) and using 50 iteration starting from the same marking (middle and lower picture)

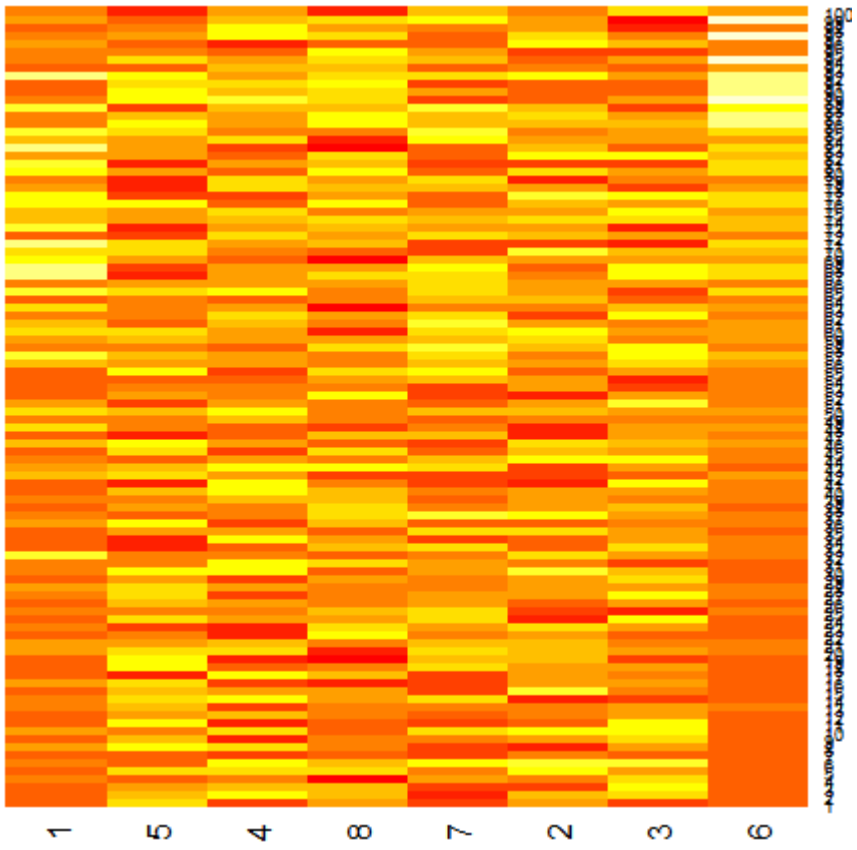


Scatterplot showing the distribution of markings distances (y-axis) as function of the final delta values they have produced (x-axes); given a marking "j" and a mean marking "mean", a marking distance is calculated as $Distance = \sqrt{\sum((T_{i,j} - T_{i,mean})^2)}$



Scatterplot showing the distribution of MA parameters sets distances (y-axis) as function of their corresponding final delta values; distances are calculated as explained above, but using the real set of MA parameters instead of a mean set. Plots obtained using 100 random markings (lower picture) and using 50 iteration starting from the same marking (middle and higher picture)


1
7
2
4
5
3
8
6
1
7
3
8
4
5
2
6



Heat Maps: sets are indexed according to their corresponding final delta value (higher values on top of each map, lower values at the bottom); 1-6 represent the MA parameters composing the set. Each parameter is coloured according to its difference from the real parameter (as found in the network): brighter colours indicate greater differences, whereas redder colours indicate smaller differences.

Plots obtained using 100 random markings (lower picture) and using 50 iteration starting from the same marking (middle and higher picture). It can be appreciated that the parameters 6, 1 and 8 follow a clear colouring pattern, becoming redder as they reach the bottom of each plot; that means, they become more similar to the native parameter as the final delta value is reduced, thus indicating that they are likely responsible for the reduction of the corresponding delta value.