Simulation and Constraint Satisfaction of Gene Regulatory Networks — Final Report

Ilkut Kutlar (u1621364) — Year 3   Supervisor: Dr Sara Kalvala

April 29, 2019

Keywords: Computational Biology, Gene Regulatory Network, Systems Biology, Synthetic Biology, Software Development, Simulation

Abstract

In recent years, the field of synthetic biology has become more important as several breakthroughs have been made showing the huge potential that synthetic gene networks possess, one example being the Repressilator. This gave rise to new research looking into modifying existing networks or building new ones to serve various useful functions.

The challenge facing researchers is building networks with the right properties to produce proteins in desired amounts for research or medical purposes. Many possible gene network combinations need to be considered to find a suitable one, a time-consuming task.

It is possible to create computer models of networks and simulate them to observe gene products even before building the network in real life. This speeds up the task of designing networks to get desired outcomes. However, a better tool could automate this task by going through all possible networks and looking for a suitable match. This project aims to build a tool enabling the creation and simulation of computer models for gene networks, as well as the non-trivial task of modifying gene regulatory networks to satisfy a set of given constraints, such as ones related to the amount of proteins produced.

Contents

1 Introduction

1.1 Introduction

Synthetic Biology is a relatively new field concerned with creating new biological constructs not found in nature with the aim of serving a useful purpose [1]. The field involves an engineering process and therefore a researcher may need to go through a design stage before actually implementing the construct in a real organism.

This project aimed to make a CAD and simulation software for biological constructs called gene regulatory networks to help with this design process. The key and original aspect of the software was a constraint satisfaction feature which can take in a list of constraints (e.g. the quantity of protein X has to be greater than 100) and modify a given network within a specified range of values to satisfy the given constraints, provided that one exists.

1.2 Motivation

It is well-established that using computers to assist in design and/or research has significant benefits both in terms of reduced time and costs. Synthetic biology is no different and in fact due to its engineering focus, particularly benefits from tools which assist researchers in getting desired results from engineered synthetic biological constructs. One simple example are software capable of simulating biological systems. In this case, the researcher may start with a computer model for a biological system already found in nature but may want to modify it to serve a desired purpose. They can then vary the properties of the computer model and simulate it to check if the objectives have been achieved. This saves time as no wet lab experiments would need to be conducted to observe the modified system.

However this process can be further improved. The researcher still needs to make an ”educated guess” about which properties to vary in the biological system in order to get the desired results. This project was motivated by this problem for the design of gene regulatory networks. It can automatically find a network to achieve given objectives and therefore may be a useful tool for researchers to have in their toolbox to save time spent on manually designing a network with desired features.

1.3 Background

This project involved design and implementation of a computational biology tool which combined knowledge from multiple fields. The biological basis of this project, the phenomenon it is concerned with, is a fundamental process called gene expression.

1.3.1 Gene expression

Every living organism has a DNA. It is made up of an ”alphabet” of four chemicals known as C, G, T, and A. DNA is also divided up into genes, which are simply sections of DNA containing ”instructions” (written in the four letter DNA alphabet) for building a specific chemical. Usually, this will be a protein such as the Lac repressor (LacI) [2]. Gene expression is the process wherein the instruction in the gene is used to build the protein it describes, i.e. the gene’s product is expressed. [2]

DNA is stored inside a cell’s nucleus; however ribosomes, the parts in a cell which ”translate” gene’s instructions to the product it describes, are outside the nucleus and the DNA cannot go out of the nucleus to reach the ribosomes. The solution to this problem is through the first step of gene expression: transcription. [2] A chemical called RNA polymerase binds to a region in the gene known as the promoter region to transcribe it, meaning (without going into much detail) it makes a copy of it called a messenger RNA (mRNA) which can go out of the nucleus.[2, 3] Once it does go out, a ribosome will read the mRNA’s instruction and translate it into whatever the mRNA’s corresponding gene ”codes for”, this process of translation is the second part of gene expression. [2]

The gene expression process can be regulated, meaning that it can be made faster or slower by affecting various steps of the process, such as transcription or translation. This project only focused on regulation of transcription since it tends to be the most common method of regulation of gene expression among organisms [4]. Regulation of transcription involves special proteins called transcription factors that bind to the promoter region in the gene and cause the RNA polymerase (described earlier) to either transcribe faster (called activation) or slower (called repression) and thus control the rate of production of mRNA and thus the quantity of mRNA in the organism. [5, p. 8] Consequently, this also affects the quantity of the chemical which the mRNA codes for. Note that a transcription factor which can regulate one gene doesn’t necessarily have a regulating effect on a different one. [5, p. 8]

It is also possible for a gene/mRNA to code for a transcription factor protein which regulates transcription of a gene within the same cell, or even regulate the same gene which codes for the transcription factor itself. These regulatory relationships make up a gene regulatory network (GRN), wherein transcription rate of each gene is dependent on the quantity of other genes’ products’ quantity in the network. [5, p. 8] For example, a diagrammatic

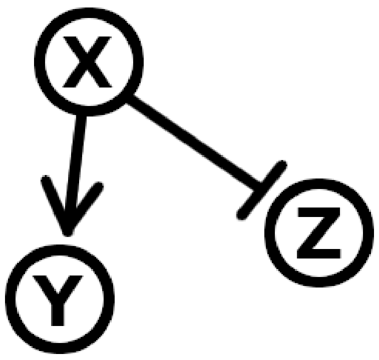


Figure 1: Diagram showing a simple GRN.

representation of a GRN is shown in figure 1 where gene X produces a transcription factor protein that activates gene Y's transcription and represses gene Z's transcription, therefore the transcription of both Y and Z is regulated by gene X.

1.3.2 Systems biology and synthetic biology

In recent years, an interdisciplinary field called systems biology has been gaining more importance. Many sources mention the difficulty of coming up with a definition for it [6, 7], however most definitions, including one given by the Institute of Systems Biology in Seattle, agree that it involves thinking of biological constructs as whole systems to understand them [8].

Gaining this understanding requires creating mathematical and computer models for such biological systems as well as tools and software to work with these. This project belongs in this field as a tool to help researchers and incorporates many theoretical concepts from it. These include the Hill equation, which offers a mathematical model for the regulation of transcription, Ordinary Differential Equation (ODE) models to describe the change in species quantities in the network over time, and the Systems Biology Markup Language (SBML) which is a markup language to describe biological models for easy transmission between researchers or different software [9]. SBML is particularly useful for online databases such as the BioModels Database [10] which host many biological models (including but not limited to GRNs) describing species, their initial quantities, values for various parameters and the equations used for modelling reactions happening in the model.

A field which has definitely benefited from systems biology is synthetic biology. *Nature* gives this definition: "Synthetic biology is the design and construction of new biological parts, devices, and systems, and the re-design of existing, natural biological systems for useful purposes.” [1]. Given the design-focused nature of this field, system biology’s mathematical/computer models are invaluable for simulating the design for novel biological constructs without having to implement them in organisms and doing lab experiments, thereby saving a lot of time.

The most relevant application of synthetic biology for this project is the design of synthetic biological circuits. These are novel GRNs not found in any biological organisms. Arguably the most famous example is the Repressilator [11]: three genes connected such that each gene represses the next, leading to oscillating amounts of protein quantities, as seen on figure 2.

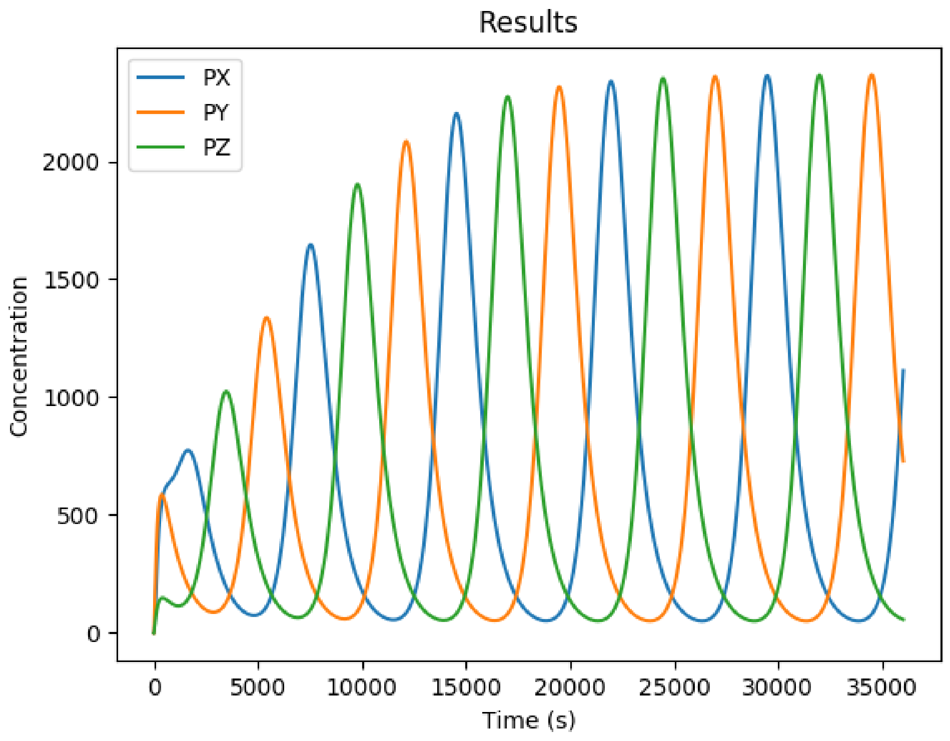


Figure 2: A simulation of the Repressilator network.

1.3.3 Similar software

Given the usefulness of computers in many fields, unsurprisingly, there have been many software similar to this project. Usually software which focus on GRNs offer the ability to create computer models and simulate these. One notable example is COPASI [12] which (among other features) allows building biological/chemical models (including but not limited to GRNs) and simulate them using a number of different simulation algorithms. It is a well-established tool and this project also aimed to offer the design and simulation features similar to COPASI's.

As far as computational biology/chemistry software go, COPASI is a general tool; it allows simulation of any biological/chemical models. There are however, many tools which specialise on GRNs or genes. For example Gene Designer [13] offers many advanced features and unlike COPASI, specifically focuses on detailed design of individual genes and their subcomponents (such as promoters, etc.). A similar tool, GenoCAD [14], also offers features to design the subcomponents of genes, but its main focus is on design of the entire network and regulatory relationships. It also offers simulation using the COPASI's API. A very similar tool is TinkerCell [15].

This project aimed to offer the best of both worlds: A biological model as designed in COPASI would be defined in terms of chemical reactions representing network phenomena. This is useful when a model has to be exported in SBML format (or when a model created in some other software has to be imported to COPASI), because SBML describes models in the same way as well, in terms of reactions. However this approach does not help the user when modelling a GRN as the user will have to manually convert all network phenomena into reactions. In such cases, software such as the aforementioned GenoCAD are more useful where the program automatically generates the necessary reactions to express network events, such as transcription, etc. This project also uses reactions to describe a GRN like COPASI (therefore making exporting to and importing from SBML very easy), but also allows the user to automatically generate reactions for common GRN events (therefore the user does not have to worry about this).

The key feature which sets this project apart from these similar software is the constraint satisfaction feature, which allows the user to specify a set of constraints on the quantity of species in a time range and automatically modify a given network to satisfy these constraints.

1.4 Requirements

When the project was first conceptualised, a number of general requirements for the project to achieve were decided on with the project supervisor. Note that these requirements reflect the big picture while the main tasks as mentioned in section 3.1 are a more detailed list of tasks necessary for the achievement of these requirements which are detailed below.

1. User should be able to design a GRN.
    (a) User can add gene objects to the current network and can specify regulatory relationships between the added genes.
    (b) User can specify properties of regulatory relationships to set the strength and type of regulation (i.e. whether activation or repression).
    (c) User can specify parameters associated with at least these types of reactions for every gene: transcription, translation of gene's mRNAs, degradation (decay of both mRNA and proteins)
    (d) Each gene is associated with an mRNA and the product of the mRNA's translation. User can to specify the names for each of these species separately.
2. User should be able to simulate a GRN.
    (a) The simulation does not need to use the most accurate simulation methods, however should offer a useful level of accuracy (i.e. at least deterministic simulation).
    (b) User can see a visualisation of the simulation results on a graph.
    (c) User can choose which species to show in the visualisation of results.
3. User should be able to perform constraint satisfaction of a GRN.
    (a) User can choose which parameters can be varied to satisfy given constraints.
    (b) User can constraint quantities of species to be less than or equal to ($\leq$) or greater than or equal to ($\geq$) a constant value (e.g. $X \leq 10$ constraints X's value to less than 10).
    (c) User can choose to apply the constraints to a given time range only (e.g. $X \leq 10$ only applies between the first 20 seconds of the simulation).
4. program should have a reasonably good user interface.
    (a) UI should be easy to use for a non-technical user. Therefore a command line interface may be intimidating for the target audience which might not be used to dealing with such interfaces.
    (b) UI should be intuitive in the sense that the user should be able to guess how to achieve necessary tasks/find necessary options using the UI.
    (c) There should be a visualisation of the GRN on the UI showing the genes and their regulatory relationships.
5. User should be able to import from and export to SBML

(a) A network designed in the program can be exported in a way that when imported to another software still retains the same meaning.
(b) program can import SBML files in such a way that the simulation will have the intended results.

1.5 Running the program

The program has a number of dependencies which have to be installed before it can be run. Firstly, Python 3.5 or above is required (due to the PyQt5 dependency). Secondly, these libraries (whose pip package names have been given) should be installed through Python's package manager, pip:

1. python-libsbml
2. matplotlib
3. scipy (or numpy)
4. graphviz
5. pyqt5

Finally, the last library on the list, graphviz, also requires a system wide software to be installed, which is also called Graphviz [16]. Once all the dependencies have been installed, one should navigate to the root of the project directory and run the file `gui.py` which will launch the main UI of the program. For example in a Bash terminal, this command could be used:

```
        python3 ui/gui.py
```

2 Design and Implementation

2.1 Technologies Used

The main development language used was Python (version 3.6.7). This was chosen due to Python being a widely used language, especially in scientific applications. This meant it had a large number of useful libraries, which especially include scientific libraries such as *NumPy*.

A number of scientific libraries were used. *NumPy*[17] was used for implementing deterministic simulation of GRNs, or more specifically to solve the ODEs which describe changes in the quantity of species in the network. *matplotlib*[18] provided a way for simulation results to be visualised on a Cartesian graph. *graphviz*[19] was used to offer visualisations of the GRNs created by the user where a graph would show species as nodes and regulatory relationships as edges. These libraries are very popular and thus have been thoroughly tested by many other users and have a large community of users which was very useful when searching for help materials online.

For parsing SBML files, the *libsbml*[20] library was used. This library was developed by the creators of the SBML standard and is supported with excellent documentation, making it easy to work with.

For designing the GUI, the Python binding of the *Qt*[21] library, *PyQt*[22], was used. This was chosen over the *tkinter*[23] library which, even though is one of the most widely used libraries for GUI development in Python[23], has an outdated look. *GTK+ 3*[24] was the library of choice in the early stages of the project, however shortly after development of the GUI started, this too was decided against due to it being a less known library than *Qt*, which would have caused problems with finding help materials. *Qt*/*PyQt* on the other hand is modern, very popular, easy to use and supported by great documentation, which contributed to this decision.

In addition to these, an integrated development environment (IDE) called *PyCharm*[25] was used which offered, among other features, syntax highlighting, code completion, and code step through capabilities all of which significantly sped up the development process. *Git*[26] and *GitHub*[27] were also used for version control and backing up the project in a remote repository, respectively.

2.2 Modelling

This part of the implementation involved deciding on a computer model to represent the biological process of gene expression and regulation. Gene expression is a very complex process with many details and irregularities, however a computer model which abstracts away most of the details would still be adequate for a large number of applications. A number of biological details were decided, with the help from the project supervisor, to be abstracted. For example in real life, transcription factors can bind promoters in various different ways [3]. Binding methods of transcription factors were abstracted from the model as this program is concerned with the simulation of the quantity of species, therefore this detail is irrelevant. Furthermore, in real life, various stages of gene expression is regulated, such as transcription, translation and post-translation stages [3, 4]. However, in the chosen computer model, only transcription can be regulated. Only transcription was chosen as it tends to be the most common method of regulation of gene expression among organisms [4].

After deciding on what to abstract and what to include, the chosen computer model was implemented in an object oriented language (Python) using a number of various objects to represent each separate item involved in gene expression. The main component, the GRN, has been implemented as a *Network* class with only two fields.

Each gene in a real GRN generally has two species associated with it: its mRNA, and the product of translating its mRNA (which is usually a protein). It was decided that this model would represent each gene in terms of the mRNA and product species with which it is associated. For example to represent a LacI gene in this model, two separate species have to be defined: an `mrna_laci` to represent the the mRNA and `protein_laci` to represent the product. These two species also have a quantity associated with them, which represents how many molecules of the species are present in the environment. Therefore the first of the two fields in the *Network* (called `species`) is a dictionary object where key is the name of a species and value is the current number of molecules (which is set to the species' initial quantity when the Network object is first initialised).

The second field is a list of *Reaction* objects (called `reactions`). Reactions in this context are defined in the same way as in chemistry: they convert a reactant to a product at a given rate `r`. Reactions are used to represent all network phenomena, such as transcription or degradation of molecules. There were a few reasons for choosing a model based on species and reactions. Most importantly, it provides great flexibility. In this model, the user can define any species they want (i.e. species other than mRNA and protein which are not directly related to gene expression) and use custom reactions to represent dependence of these species to, for instance, an arbitrary protein in the network. This wouldn't have been possible with a model representing the network as a collection of genes. Secondly, this model is preferred amongst most computational biology tools, as well as SBML, a popular file format used to encode a biological model for sharing. Therefore using this internal model makes it easier to convert to and from SBML and also provides a familiar model for the user who might have encountered a similar model in other software.

The *Reaction* object has three main fields: `left`, `right` and `rate_function`. There is additionally a `name` field to assign an identifier to the reaction which can be used when referring to this reaction elsewhere in the program (such as when doing constraint satisfaction). `left` and `right` are the reactant and product species respectively (currently only one species is supported on either sides) while `rate_function` specifies the equation used to compute the reaction rate. The `rate_function` field takes a value of type *Formula*. *Formula* is defined as an abstract class and there are several implementations of it describing the equations for various GRN related reactions, such as *TranscriptionFormula* or *TranslationFormula* to represent formulae for modelling transcription and translation respectively. Therefore, for example to make a *Reaction* object represent a translation event, one has to initialise an instance of *TranslationFormula* with desired properties (such as translation rate) and pass it to the Rate Function field of *Reaction* object.

It was decided to separate *Reaction* and *Formula* objects (as opposed to having multiple implementations of the Reaction object, e.g. TranslationReaction) as it made the code easier to reason with. This separation of concerns also meant the calculation of the reaction rate is completely abstracted away from the *Reaction* object: the abstract *Formula* class has a `compute()` function implemented by every subclass of *Formula*. A reaction class only has to call its rate function's `compute()` function to get the reaction rate without having to worry about how it is implemented by the *Formula* class.

2.2.1 TranscriptionFormula

TranscriptionFormula represents the formula used for computing the transcription rate of a gene (which may or may not be under regulation by a transcription factor). The mathematical model chosen for the formula is the Hill equation which has two forms: One for when the gene is being up regulated (equation 1[5, p. 13] in figure 3) and another for when the gene is being down regulated (equation 2[5, p. 13] in figure 3). *TranscriptionFormula* object automatically chooses the right version to use. The equations will be explained in the next paragraph. Note that these equations are only used when the gene is being regulated by at

$$\alpha \times (\frac{[T]^n}{[T]^n + K^n}) \tag{1}$$

$$\alpha \times (\frac{1}{1 + (\frac{[T]}{K})^n}) \tag{2}$$

Figure 3: Activation[5, p. 13] and Repression[5, p. 13] forms of the Hill equation.

most one gene.

The class signature and the parameters that *TranscriptionFormula* class' constructor takes are shown in figure 4. The code includes type hints of parameters [28] for clarity. The class has a `compute()` function which implements the Hill equations as shown in equations 1 and 2. The term α in both equations is the unregulated transcription rate (`rate` in figure 4), n is the constant known as Hill coefficient (`hill_coeff`), $[T]$ is the quantity of the regulating transcription factor and K is the quantity of transcription factors required to reach a considerable level of activation or repression (50%) of this gene's transcription rate α [5, p. 13] (Both of which are not seen in figure 4 because they are defined inside the *Regulation* class, instances of which are stored in the `regulators` variable). `transcribed_species` is the name of the mRNA species of the gene whose transcription is being described. `input_gate` field relates to regulation by multiple genes, which will be explained later.

```
class TranscriptionFormula(Formula):
    rate: float
    hill_coeff: float
    regulators: List[Regulation]
    transcribed_species: str
    input_gate: InputGate
```

Figure 4: TranscriptionFormula class's signature and parameters

The TranscriptionFormula class has a `regulators` field accepting objects of type *Regulation*. As the name suggests, this class represents regulation of a transcription reaction by a transcription factor (which is again represented by a species) and is shown in figure 5. `from_gene` and `to_gene` are such that `from_gene` contains the name of the species regulating `to_gene`'s transcription. The formula looks up the quantity of `from_gene` to get the $[T]$ term in both equations 1 and 2 while `k` is the term K in the equations.

```
    class Regulation:
        from_gene: str
        to_gene: str
        reg_type: RegType
        k: float
```

Figure 5: Regulation class's signature parameters

Separating the `K` value from *TranscriptionFormula* to each separate *Regulation* was necessary since if a gene is being regulated by multiple regulators, it may have a different `K` value for every different regulator.

In the case the gene is being regulated by multiple genes (currently up to two genes are supported), the equations 1 and 2 cannot be used as a mathematical model. Instead, combinatorial versions of the Hill equation as proposed by Mangan and Alon [29] were used. In this model, there are various ways in which the two genes can interact, dictated by their "input gate". This program supports AND and OR input gates (which the user must specify when creating a *TranscriptionFormula* object). The equations for AND gate is given in figure 6 and OR gate in figure 7. Note that H^+ and H^- are functions which refer to equations 1 and 2 respectively. In *TranscriptionFormula* class (figure 4) the field `input_gate` denotes the type of input gate the reaction uses. It is of type *InputGate*, which is an enum type accepting values AND, OR and NONE, the last being for reactions which are only regulated by a single gene, and thus do not need their input gate to be specified.

Finally, in the case that the gene is not being regulated by any gene at all (i.e. it is a constitutive gene), the unregulated transcription rate (the `rate` field) is taken as the transcription

Gene 1 Regulation	Gene 2 Regulation	Equation
Activation	Activation	$H^+([T1], n, k1) \times H^+([T2], n, k2)$
Activation	Repression	$H^+([T1], n, k1) \times H^-([T2], n, k2)$
Repression	Activation	$H^-([T1], n, k1) \times H^+([T2], n, k2)$
Repression	Repression	$H^-([T1], n, k1) \times H^-([T2], n, k2)$

Figure 6: Combinatorial Hill equation for an AND gate

let a = $(\frac{[T1]}{k1})^n$
let b = $(\frac{[T2]}{k2})^n$
let c = $1 + a + b$

Gene 1 Regulation	Gene 2 Regulation	Equation
Activation	Activation	$\frac{a+b}{c}$
Activation	Repression	$\frac{a+1}{c}$
Repression	Activation	$\frac{1+b}{c}$
Repression	Repression	$\frac{1}{c}$

Figure 7: Combinatorial Hill equation for an OR gate

rate without any use of the hill equation.

2.3 Simulation

2.3.1 Deterministic

Deterministic simulation, as the name suggests, is a system simulation which will always yield the same result using an algorithm which doesn't involve any randomness. It is meant to capture average values for the network, i.e. it gives values which can be ”expected” (from a statistical perspective) to be received when observing a real network.

Deterministic simulation was achieved using ordinary differential equations (ODEs) [30, pp. 43-62]. As explained in section 2.2, each reaction contains a rate function describing the chemical reaction's rate. An advantage of choosing to model network phenomena as chemical reactions was that they can be easily converted to ODEs [31, section ”Formulation of candidate model architectures”]. For example, *TranslationFormula*'s reaction rate (which describes translation of an mRNA to a Protein) is defined as $\beta \times [mRNA]$ where β is the translation rate and $[mRNA]$ is the quantity of the mRNA species. The ODE describing a translation reaction would then be: $\frac{d[Protein]}{dt} = \beta \times [mRNA]$, i.e. the quantity of protein species simply changes by the translation rate.

The first step of simulation is to build a system of ODEs, that is to convert all reactions in the network into ODEs describing the change in the quantity of each species over unit time in the way shown earlier. Next step is to solve this system of equations to get a function $f(t)$ where t is time and the output of the function is the network state (i.e. the quantity of each species) at time t [30, pp. 43-62]. To achieve this, the program made use of *NumPy*, a scientific library for Python which has an `odeint()` function to solve a system of ODEs. Once the system of ODEs prepared earlier is input to the function, the required function $f(t)$ is calculated which contains the results of the simulation.

2.3.2 Stochastic

When compared to deterministic simulation, stochastic simulation of a system involves the use of a random algorithm, therefore the simulation result is unpredictable and slightly different each time. However averaging n stochastic simulation approaches the deterministic simulation results as $n \to \infty$.

A stochastic simulation of the system is desirable: real biological systems experience random cellular noise which affects species' quantities in random amounts each time the simulation is run. The randomness of this simulation method therefore simulates this cellular noise and is thus a more accurate simulation.

There are several stochastic simulation algorithms available. With suggestion from the project supervisor, this project chose to make use of the Gillespie algorithm (also known as the Stochastic Simulation Algorithm) [32]. It was chosen because it is appropriate for GRN simulations and is widely used for this application in one form or another. Unlike deterministic simulation, this algorithm does not make use of ODEs (as, for example $\frac{d[X]}{dt}$, would suggest that reaction occurs in every unit time, while in Gillespie algorithm, reactions occur at random time intervals). Instead, every reaction is assigned a propensity, i.e. the probability that this reaction will occur t seconds from now, where t is also a randomly chosen amount inversely proportional to the sum of all propensities in the network. This program takes the propensity of each reaction to be its reaction rate [33]. The algorithm represents the current network state (the quantity of species) as a "state vector". Each time a reaction r occurs, the state vector is updated by adding r's "change vector". Change vector is essentially a vector which describes the change in quantity of each species in the network [33]. This program computes the change vector of each reaction by using its reaction rate (e.g. a *DegradationFormula*'s change vector could be $[-1, 0, 0]$, where -1 is the change in the decaying species' quantity while the two 0s represent other species which have not been affected by this reaction) [33].

2.3.3 Visualisation

The program also lets the user visualise the results of simulation on a graph with axis time (x) against quantity (y). To achieve this, *matplotlib* [18] (an external library) was used which offers many useful features, such as zooming in to the figures, out of the box.

2.4 Constraint Satisfaction

Constraint satisfaction is the key feature of this project. In short, it is a process by which a given network is modified to satisfy a given set of constraints. The first step was to decide on how to implement this design in code. Constraint satisfaction is done using a single function `find_network()`. The function takes a list of constraints through a list of *Constraint* objects.

2.4.1 Constraint object

```
class Constraint:
    species: str
    value_constraint: Callable[[float], float]
    time_period: Tuple[float, float]
```

Figure 8: Constraint class' signature and parameters

Figure 8 shows the class signature and constructor parameters that the *Constraint* class takes (excluding a `pretty_print` field used for user interface and is irrelevant here). `species` is simply the name of the species whose quantity will be constraint.

`value_constraint` is a Python function which takes a single float value and returns a float value. This represents the constraint. It is used by the constraint satisfaction algorithm in this way: when a simulation value is input into the `value_constraint` function, if the function returns a value less than or equal to 0, then this signifies that the input value satisfies the constraint specified by this `value_constraint` function. The lambda function shown in figure 9 specifies a constraint of $species \leq 200$ where species is the quantity of a species of the user's choosing (e.g. call it `mrna_laci` for this example's sake). For example if `mrna_laci` species' quantity is 250 (above its constraint and thus should be rejected), the function output will be 50, and the constraint satisfaction algorithm will reject the network which produced this value and try another one until a matching network whose quantity for `mrna_laci` will satisfy this constraint in the given `time_period`, which will be explained later.

```
        lambda species: species - 200
```

Figure 9: A lambda function which enforces that $species \leq 200$ where species is quantity of a species of the user's choosing

This way of specifying constraints was chosen over a simple `int` value specifying whether the constraint is of type $\leq$ or $\geq$ was chosen due to the flexibility it offers. Under this implementation, it would be possible to define arbitrarily complex constraints, such as one shown in figure 10 which defines a complex value constraint function in terms of other species `mrna_tetr` and `mrna_cl` and using Python functions such as `sin()`. Note that even though this implementation allows for such complex constraints to be implemented easily in the future, the current version of the program does not allow the user to define complex constraints involving Python functions or other species yet. This decision was made to focus on other features of the program with higher priority.

This way of using functions to specifying constraints is counter-intuitive. Therefore the user interface abstracts this away and lets the user specify a constraint by choosing the species to be constraint and which sign to use (e.g. $mrna_laci <= 200$), which is then parsed by the program to produce an appropriate value constraint function. The dialog used for entering constraints is shown in figure 44.

```
        lambda species: species - sin(mrna_tetr) + 2*mrna_cl
```

Figure 10: An arbitrarily complex value constraint function

Time period is a pair or 2-tuple which defines a time range in the format (start, end). It is used to specify within which time period the constraint should be satisfied. For example (0, 10) means that the constraint should hold between 0 and 10 seconds, both inclusive.

2.4.2 Mutable object

One can realise at this point that only defining a set of constraints is not enough because the constraint satisfaction algorithm is not limited in the network parameters which it can modify in order to satisfy the constraints. This is most probably undesirable: the user may want to limit, for example, the range of values a certain *TranslationReaction*'s rate value can take, rather than let the program try every number from 0 to infinity until a suitable network is found. This functionality is implemented using the idea of a ”Mutable”, i.e. a parameter which the user has allowed to vary between a given range of values in order to satisfy the constraints. The program supports three mutable classes shown in figure 11 (some class fields have been omitted from the figure as they are used internally and so are irrelevant for this explanation). Each class represents a different type of network parameter which can be mutated, as well as the range of values it can take.

VariableMutable describes a general variable with name `variable_name` which is allowed to have a value between `lower_bound` and `upper_bound` with intervals of `increments`. For example, a variable mutable with `lower_bound` = 5, `upper_bound` = 10 and `increments` = 1 means the constraint satisfaction algorithm will try the values 5, 6, 7, 8, 9, and 10 for this variable. This class is usually used for variables which are not a reaction formula’s parameter, for example to for mutating the initial quantity values of species.

ReactionMutable is a subclass of VariableMutable and has the same properties except that it denotes a variable inside a reaction whose name is given by `reaction_name`, i.e. it denotes a reaction formula’s parameter (e.g. a translation reaction’s reaction rate).

RegulationMutable is only used with transcription reactions. It denotes the possible regulation options a given transcription reaction is allowed to take during constraint satisfaction. `reaction_name` is the name of the reaction as before. `possible_regulators` are the set of all species this reaction is allowed to be regulated by, and `possible_reg_types` are all the possible ways this reaction can be regulated in: Activation, Repression or None (for constitutive genes). `k_variable` is a *VariableMutable* (as described earlier) to represent the range of values this regulation’s K value (as defined earlier in section 2.2.1) can take. `is_installed` denotes whether this regulation is present in the network or not, as the lack of regulation from this species is also a valid regulation option. Given this range of acceptable regulation options using the *RegulationMutable* object, the algorithm will then try all the option combinations to find which one satisfies all the constraints.

2.4.3 Constraint satisfaction algorithm

Now that the classes the constraint satisfaction function `find_network()` accept as input have been explained (namely the Constraint and Mutable objects), the final part is the algorithm used for finding a network which satisfies the given constraints. The function takes in a list of Constraint objects, a list of Mutable objects and the network which will be modified.

The main idea behind the algorithm is shown in figure 12. As discussed earlier, a *Mutable*

```
class VariableMutable:
    variable_name: str
    lower_bound: float
    upper_bound: float
    increments: float

class ReactionMutable(VariableMutable):
    reaction_name: str

class RegulationMutable:
    reaction_name: str
    possible_regulators: List[str]
    possible_reg_types: List[RegType]
    k_variable: VariableMutable
    is_installed: bool
```

Figure 11: The sets of mutable classes used by the program to represent values which can vary

object defines a range of values. As an example, if three mutables with value ranges 1-5, 6-10, and 16-20 (all with increments of 1) were input to function, this would create 5^3 unique combinations. Therefore, these create a pool of ”value combinations”. The algorithm goes through each of these value combinations, each time picking the next combination to try using a selection algorithm. As described earlier, this value combination represents values for network parameters which are allowed to vary, therefore the input network can be modified so that its network parameters match the ones described by the chosen combination. Next, the algorithm checks whether this modified network satisfies all the given constraints. If not, the process is repeated until a match is found or the pool of combination values have been exhausted.

If the selection algorithm chooses a combination of values randomly, the equivalent of performing an uninformed search, the computational cost involved will be too great: Even with only three mutables with a range of 0-10 and increments of 0.1, there would be a total of 1,000,000 unique combinations to try in the worst case.

To solve this problem, a best-first search algorithm was used to perform an informed search of the value combinations. The entire pool of combinations was modelled as a tree, an example of which is shown in figure 13. In this example, there are two mutables, X and Y, each of which have a lower bound of 0, upper bound of 2 and are incremented by 1 ((0,0)

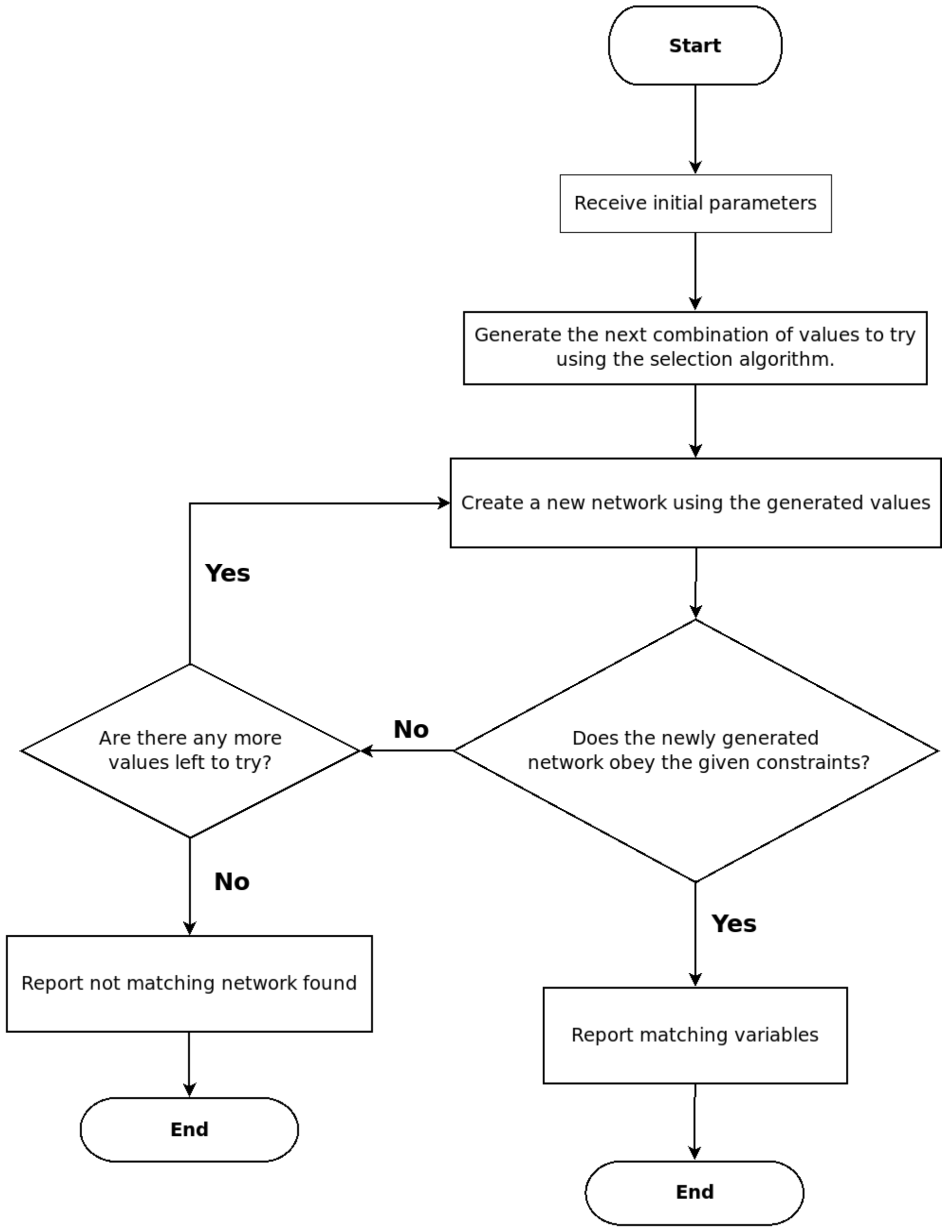

Start
Receive initial parameters
Generate the next combination of values to try using the selection algorithm.
Create a new network using the generated values
Does the newly generated network obey the given constraints?
No
Are there any more values left to try?
Yes
No
Yes
Report not matching network found
End
Report matching variables
End

Figure 12: Flowchart showing the basic idea behind the constraint satisfaction algorithm

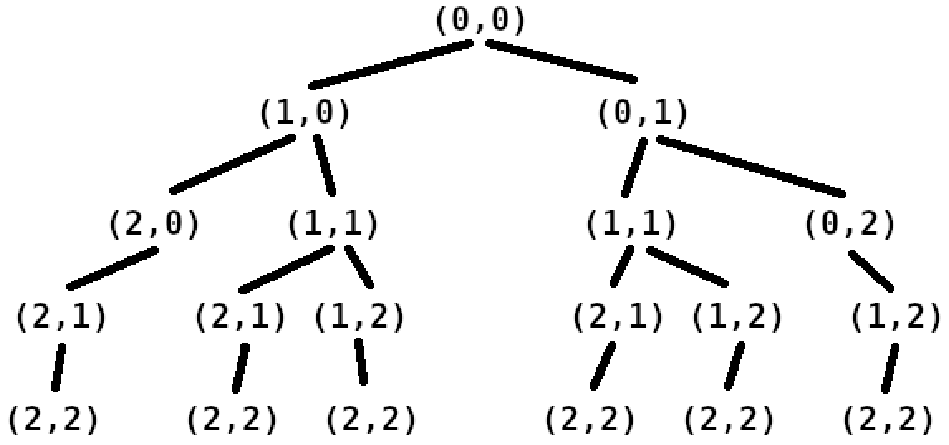


Figure 13: Example of a tree which is used by the best-first search algorithm to model all value combinations.

in the diagram represents X=0 and Y=0). At the root of the tree (first level), both are set to their lower bound values. In the second level of the tree, there is one node for each variable representing a combination of values where one of the variables have been increased by its increment value. For example the left node in the second level shows X's value increased to 1 and in the right node, Y's value increased to 1. This goes on until the upper bound of all variables have been reached.

The algorithm starts by checking the combination in the root node. If the combination doesn't satisfy all the constraints, one of the nodes in the next level has to be chosen. As is required in a best-first search, an evaluation function (or a heuristic function) is used to assign a value to each node in the next level denoting the node's closeness to satisfying the constraints (where 0 would mean the goal has been reached) [34, p. 5].

For example consider the tree of values in figure 14 along with some of the nodes' evaluation function outputs. When at the root level, the algorithm only computes the heuristics values for the children of the root node and not others. After comparing the two values, 10 and 20, the subtree with root (0,1) is chosen (as it has a lower heuristic value and is thus more promising) to be explored next. Then, the heuristic value is computed for (0,1)'s children only. Each time, the algorithm chooses the most promising node (as measured by the heuristic). Note that the algorithm can backtrack to an upper node: for example it can go back to (0,2) which has f=19 if when exploring (1,1) it discovers a child of (1,1) has a heuristic value greater than 19, making 19 the lowest heuristic value amongst discovered by

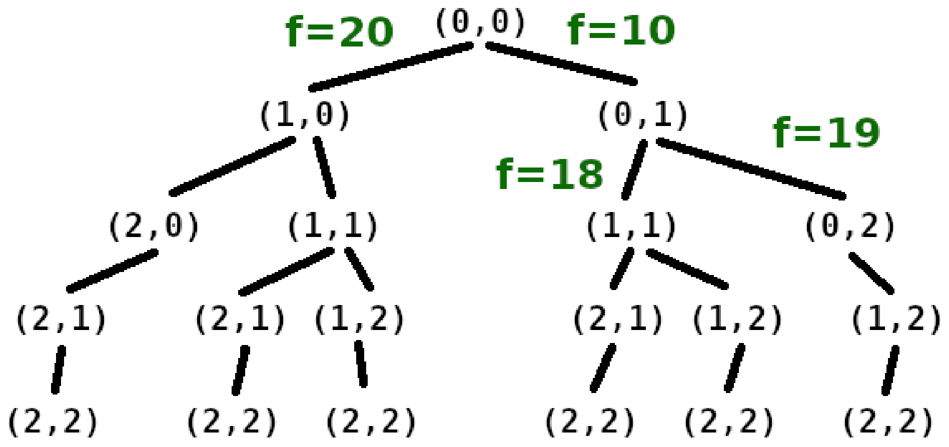


Figure 14: The same tree in figure 13 with some of the evaluation function values given.

unexplored nodes. This means that each time, the most promising nodes are explored first, so a match will be found (if one exists) much more quickly.

If no matching combination exists, best-first search will have to traverse the whole tree of value combinations. This may take a lot of time in the case that there are many mutables and the user may not want to wait for the whole tree to be traversed. Therefore, a parameter can be passed into the `find_network()` function called `give_up_after`. The function will ”give up” on looking for a value combination and report failure after trying for `give_up_after` many seconds. This amount can be chosen by the user depending on how much they are willing to wait for the algorithm to keep looking for a solution.

Evaluation function

The evaluation function used is defined by the algorithm shown in algorithm 1. As its input, it takes the result of simulating the current network and the set of constraints against which the simulation results will be evaluated. The simulation results are such that if the number of samples to take is s, and the simulation time is t seconds, the quantity of each species will be taken every t/s seconds and there will be s ”data points”.

The simulation method used is deterministic rather than stochastic. Deterministic simulation was chosen since it is considerably faster. Given that the evaluation function may have to be called many times in most cases, it was important for the simulation speed to be fast as otherwise constraint satisfaction process would be unreasonably slow.

As an example to explain the algorithm, consider the simulation results visualised in figure 15 and its evaluation against a single constraint: $pY \leq 100,000$ (as shown by a red line) in the time period $100s$ to $200s$ (as shown by a grey box). The for loop in the algorithm starts with the first (and in this example only) constraint c, namely $pY \leq 100,000$. First step is to find the data points within the `sim_results` which lie inside the time period in which this constraint is applied (i.e. inside the grey box), which are stored in `vals`, some of which are seen in the figure. Next, `not_sat` is set to a list of data points (in the time period) not satisfying the constraint. In this example, those points, as marked in the figure, are the ones greater than 100,000: 102000, 108000, 114000, 120000, 122000, 125000, 133000. Finally, a mean average of these are computed (117,714) and the constraint's `value_constraint()` function (`lambda v: v - 100000`) is applied to it to get 17,714. This number approximates how far away this network is from satisfying this specific constraint. This value is then computed for the other constraints as well (though this example only contains one) and summed to arrive at an evaluation value based on how close the network is to satisfying all the constraints at once (where lower numbers mean it is closer).

```
Function evaluate_network(sim_results, constraints)
    total ← 0
    for c ∈ constraints do
        vals ← results_between(sim_results, c.time_period)
        not_sat ← value v ∈ vals s.t. v does not satisfy c
        total ← total + c.value_constraint(mean(not_sat)
    end
    return total
```

Algorithm 1: Pseudocode for the evaluation function used.

Finding closest match

The `find_network()` function is for finding an exact match, i.e. a network which strictly satisfies all the given constraints. In some cases, there will not be any networks which satisfy the given constraints and the program will not report any networks. This may not be what the user wants. The network which comes closest to satisfying the constraints with the given mutables may also be useful. The `find_closest_network()` function uses simulated annealing algorithm to achieve this.

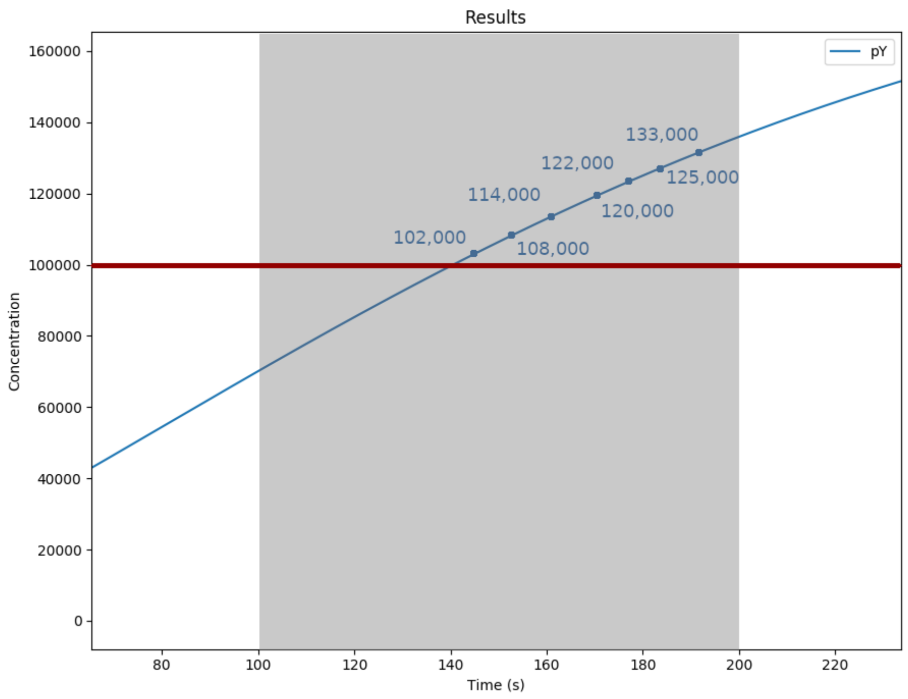


Figure 15: A graph showing the quantity of pY over time and a constraint ($pY \leq 100,000$) represented by a red line

Simulated annealing is a more sophisticated version of a hill climbing function, therefore its aim is to maximise/minimise a given function [34, pp. 63-65]. It avoids getting stuck in local maxima/minima (as opposed to a hill climbing algorithm) by randomly jumping to another value with a small probability. [34, pp. 63-65] In this case, the function it is trying to minimise is the evaluation function as described earlier which represents how close the current network is to satisfying all the networks. By finding the network which minimises the evaluation function, it finds the network which comes closest to satisfying all the constraints.

2.5 File Input/Output

This component focuses on the file input and output capabilities of the program. The main aim was supporting the SBML file format. SBML is a standard developed for the sharing of biological models, allowing systems biologists to easily distribute mathematical models. SBML is a very popular format supported by many computational biology tools, therefore it was very useful for this program to support importing from and exporting to SBML.

File Input

The SBML standard uses XML to describe models. Therefore the main steps in importing an SBML file were:

1. Parsing the XML in the file to create an abstract syntax tree (AST) or a similar intermediate representation which makes it easy to access the data in the file in Python.

2. Converting the intermediate representation into the internal model used by this project to represent GRNs.

To accomplish step 1, an external library created by the same group which created SBML was used. This Python library called *libsbml*[20] reads a given SBML file and creates an AST in terms of a number of classes defined by *libsbml*. For example a possible AST may look like in figure 16, where *ListOfSpecies*, *ListOfReactions*, etc. are real class names used by *libsbml* to represent the parsed tree's AST in Python.

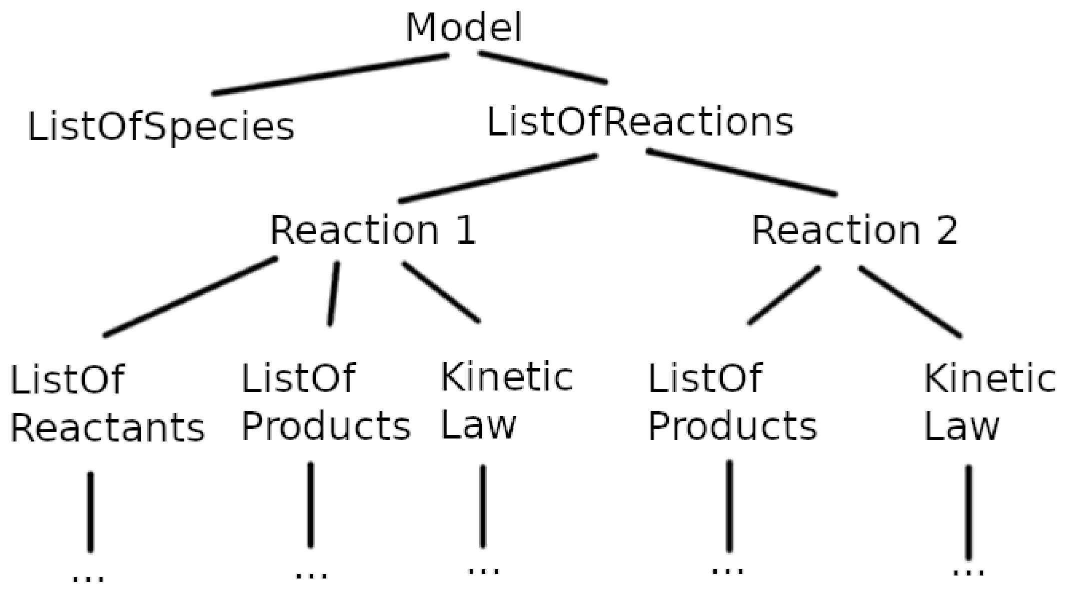


Figure 16: Example of an AST representing an SBML file.

While the properties of a model that an SBML file captures may vary, for a typical GRN model, these entities are usually included and are used by this program when importing an SBML file:

1. A list of all species in the model and their initial quantities.
2. A list of reactions which occur in the model, where each reaction has:
    - Reactant species.
    - Product species.
    - A formula describing the rate of reaction.
    - A list of reaction parameters which are local variables with a constant value that can be used in the definition of the reaction rate formula.
3. A list of parameters which are essentially global variables with a name (symbol) and a constant value which can be used in reaction rate formulae and other formulae.
4. A list of assignment rules which are similar to parameters with the difference that rather than defining the constant value of a variable, these define a formula in terms of constants, parameters and result of other assignment rules which is used to calculate the value of a variable. This value can again be used in formulae.

Because of the fact that the program's internal representation of a GRN closely follows the way models are represented by SBML (as seen in the above list), step 2 was relatively easy (excluding reactions). The conversion process involves creating a new *Network* class and populating it with data from the AST. Item 1 of the list directly corresponds to *Network* class's `species` field. The values described by both items 3 and 4 were added to a symbol table and passed to the *Network* class to be used by instances of the *CustomFormula* class (which will be discussed in detail later). However since assignment rules only describe a formula and not the actual value, this value had to be computed before being added to the symbol table.

The AST output by *libsbml* expresses formulae using a binary tree where each node is an operator or a number/variable represented with *libsbml*'s classes. *libsbml* does not automatically compute the formula's binary tree, therefore a function (`safe_evaluate_ast()`) was written which takes the binary tree and the symbol table prepared earlier (assignment rule may contain symbols such as global parameters and so the program needs to know their numeric value to compute the assignment rule, hence why the symbol table is needed) and computes the formula's value.

This function firstly calls a further function `evaluate_ast()` which computes the formula expressed by the binary tree by traversing the tree and manually recognising each operator and performing its intended function (e.g. if a "+" operator is found, it adds the left and right children of the node). The inconvenient part of this approach is that every operator that SBML supports must be manually handled. While `evaluate_ast()` function implements some basic operators, there are some that it can't recognise and it was decided that the project should focus on other features rather than extending the list of supported operands. However, as a quick way of adding support for more operands, this approach was taken: In the case of this function encountering an unrecognised operand, it will raise an exception. When this happens, `safe_evaluate_ast()` will not give up on trying to evaluate the binary tree and report failure, instead it will try calling a second function called `evaluate_ast_as_string()` with the same binary tree to try evaluating the same tree using a different method this time:

libsbml offers a function to translate the binary tree of a mathematical expression into a string representing it (e.g. "$(x + 5)/9$"). These mathematical expressions mostly follow Python's syntax; therefore if these strings were evaluated by Python's interpreter, the return value would be an evaluation of the expression. `evaluate_ast_as_string()` does exactly this by using Python's `eval()` function. To handle unknown variables in the passed string expression (e.g. x in $(x + 5)/9$), `eval()` function accepts a symbol table to look for values of unknown variables (e.g. $\{x : 10\}$). Therefore, the function is also passed the previously prepared symbol table containing values for network parameters which are likely to be in these mathematical expressions. Calling this second function when the first one fails makes the overall evaluation function (`safe_evaluate_ast()`) more likely to recognise and evaluate operands it may encounter. For example functions such as `sin()`, etc. are part of Python's syntax and therefore `eval()` will be able to evaluate these.

Next step is the conversion of reactions. Reactant and product species in SBML correspond directly to fields `left` and `right` (as discussed in section 2.2) respectively in the *Reaction* class. As discussed in section 2.2 this program's *Reaction* class represents reaction rate formulae using sub classes of the *Formula* class depending on the reaction's type. The problem with SBML files is that while the reaction type can be specified (through the reaction's SBO term), it is sometimes omitted and is not a reliable method of gathering this information. Furthermore, even if the reaction type is known, for example for a reaction `r` which is a translation reaction in the SBML, there is no way to deduce which of the terms in `r`'s reaction rate formula in the SBML is its translation rate. Therefore it wouldn't be possible to assign a *TranslationFormula* object to this reaction's formula in SBML when converting it. The solution was to create a sub class of *Formula* called *CustomFormula*. While other formulae like *TranslationFormula* already have a hard coded formula and only need values for this formula's parameters, *CustomFormula* object allows passing in the formula used to compute it as well, in the form of a string. As discussed earlier, a symbol table of global parameters is also passed to *CustomFormula* which provides the values of unknown variables in the given formula string. To convert reaction rate formulae, these steps were followed:

- Obtain the reaction's reaction rate formula's binary tree.
- Convert it to its string equivalent using the aforementioned *libsbml* function.
- Create a *CustomFormula* class and pass the string equivalent of reaction rate formula obtained in previous step to the class.
- Pass the entire symbol table as well as the reaction's local parameters to the class.

As discussed in section 2.2, each sub class of *Formula* have a `compute()` function. For *CustomFormula*, this function evaluates the passed string function. To do so, the aforementioned `safe_evaluate_ast()` function is used. This function needs both the string and binary tree form of the equation, therefore the string version of the equation is converted to binary tree using another *libsbml* function for doing so.

When performing constraint satisfaction with *CustomFormula* objects (and therefore with imported SBML files), the user is able to choose global parameters (which are represented with the *GlobalParameterMutable* class that is a subclass of `VariableMutable`) of the network (which are common to find in many SBML models) as mutables. Global parameters are usually used in rate functions, therefore the user is able to mutate the reaction rate of reactions imported from SBML by mutating the network's global parameters. However, this program currently does not support mutables of local parameters.

File Output

Output of the currently loaded GRN to an SBML file is relatively easier than input. As mentioned earlier, *libsbml* offers classes to represent a biological model. One relevant class is *Model*, which represents the entire model, i.e. the root of the AST. *libsbml* offers the functionality to convert a *Model* class to an SBML file. Therefore, the required process is to convert the currently loaded GRN to a *Model* class.

The process involves starting from the root of the AST, the *Model* class, and adding the necessary children at each level (e.g. species and reactions) by converting the program's GRN's members into their *libsbml* counterparts. The process was relatively easy except for the conversion of reactions. As mentioned before, SBML uses a binary tree to represent mathematical expressions, therefore this program's reaction rate formulae had to be converted to this format. *libsbml* offers a function to convert string formulae into binary trees. To be able to use this function, each *Formula* class had to implement a function to convert its formula into an equivalent string. For example a *TranslationFormula* could return $laci\_mrna \times 10$ where 10 is its translation rate and $laci\_mrna$ is the name of its corresponding mRNA.

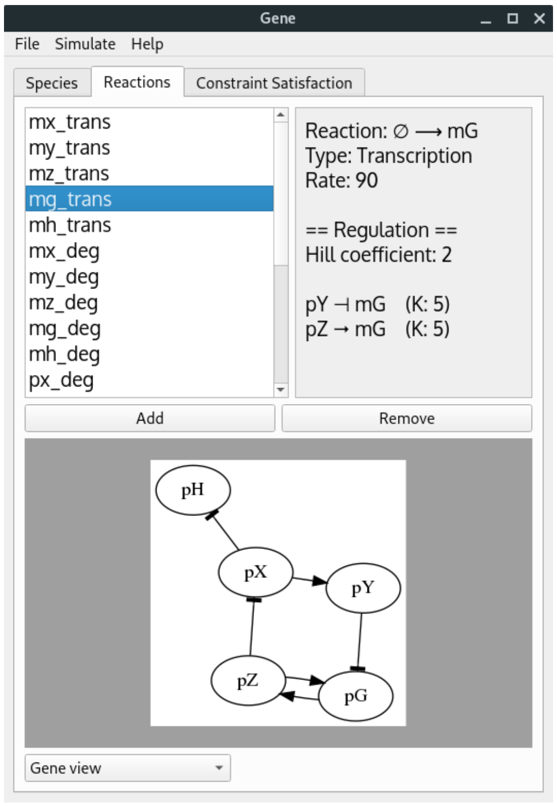
Once the *libsbml Model* class representing the GRN was built, a *libsbml* class was used to convert it into an SBML file.

2.6 User Interface

To create the user interface (UI) for the program, the Qt [21] library for Python (PyQt [22]) was chosen over other alternatives.

The most important consideration when designing the UI was that a large amount of information and options had to be shown to the user and laying out all of this in one screen would make the UI unnecessarily complicated and particularly ”intimidating” for first time users. Another consideration was designing a familiar UI; that is, it had to look and feel familiar to the user by following some simple design guidelines followed by most desktop applications.

A screenshot of the UI can be seen on figure 17. The three largest sections of the program have been separated into tabs so as not to overload the screen with information. The aforementioned figure shows the ”Reactions” tab. On the left is the name of all reactions shown using a ”list box”. Each reaction has some properties associated with it which is only revealed on the right when it is clicked. This is an intuitive design where the list of reaction names on the left offers the user an overview of the network's reactions. If the user then requires more details, clicking on any reaction brings up details of its properties on the right.



mx_trans
my_trans
mz_trans
mg_trans
mh_trans
mx_deg
my_deg
mz_deg
mg_deg
mh_deg
px_deg

Reaction: $\varnothing \longrightarrow$ mG
Type: Transcription
Rate: 90

== Regulation ==
Hill coefficient: 2

pY ⊣ mG (K: 5)
pZ → mG (K: 5)

Figure 17: A screenshot of the UI open to the “Reactions” tab.

Finally the bottom panel displays a visualisation of the network currently loaded in the program. Two different views are offered: gene view and reaction view (The combo box enabling switching between the two can be seen underneath the visualisation). The visualisation follows the usual network notation used in systems biology where an arrow with a sharp head (”$\rightarrow$”) represents activation and a flat head represents repression (”$\dashv$”). Figure 18 shows reaction view of the GRN which simply is a direct visualisation of the reactions. As discussed earlier in section 2.2, this program models regulation in such a way that gene X regulating gene Y means that the species pX (X’s protein) regulates the species mY’s (Y’s mRNA’s) transcription. Reaction view shows a graph with all species on it, and shows regulatory relationships in this way. For example the figure shows that the species pX activates the species mY and represses the species mH, which means gene X activates Y and represses gene H. The gene view equivalent of the same GRN is shown in figure 19. In this view, the mRNA and protein of a gene are not shown as separate nodes, but instead are shown as a single gene (which is automatically assigned the name of its protein). For example gene X is the node labelled ”pX”. This view is closer to the diagrams systems biologists tend to use.

It can again be seen that gene X (which is labelled by its product, ”pX”) regulates the genes of the same mRNA, namely pY and pH, and is regulated by pZ. The user does not have to explicitly state whether a species is an mRNA or protein. During visualisation, the program infers this information from whether a species is a part of a transcription reaction (therefore it’s an mRNA) or a translation reaction (therefore it’s a protein). To create these network visualisations, the program uses a third party library called *graphviz*[19] which is a general purpose tool to create visualisations of graphs.

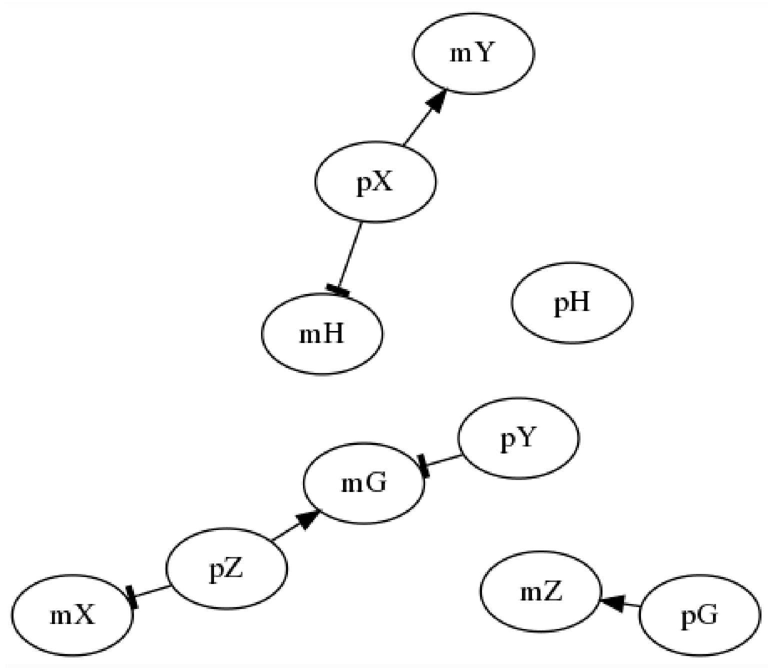


Figure 18: Reaction view visualisation of the GRN.

The UI needs to accept some data from the user. Validation of user input was an important consideration. The screenshot in figure 20 shows a dialog for adding a transcription reaction which accepts a number of properties. Transcription rate must be a floating point number; therefore only numerical characters and the point character (’.’) must be accepted. The text box has a filter for only allowing numerical characters, and therefore all entered data is guaranteed to be valid. Transcribed species field can take a string value representing one of the defined species in the network; therefore only the species in the network are valid values for this field. The field is a combo box whose items are species of the network, again guaranteeing that all input will be valid, and will also be more convenient for the user than writing the name in a text box. Further validation is done using the ”Is Regulated” check box in the form of disabling fields related to regulation and only allowing them when the check

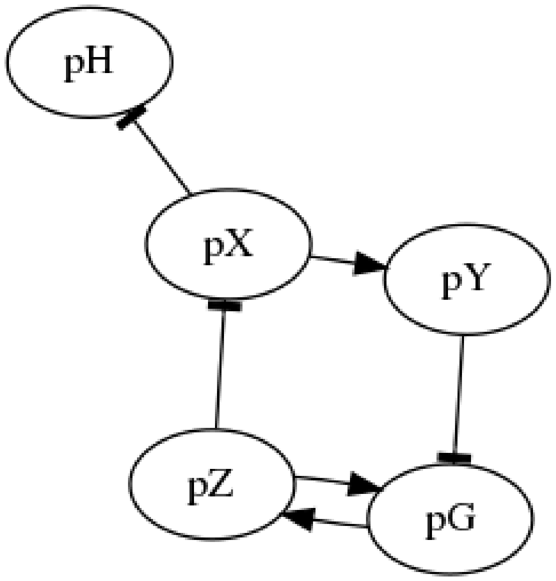
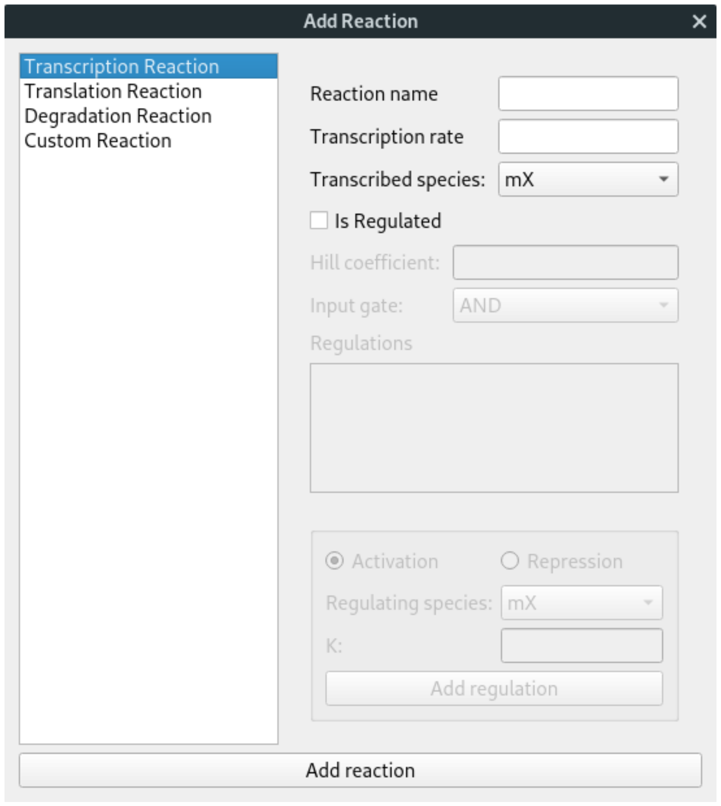


Figure 19: Gene view visualisation of the GRN.

box is checked.



Reaction name
Transcription rate
Transcribed species: mX
[ ] Is Regulated
Hill coefficient:
Input gate: AND
Regulations
[● Activation] ○ Repression
Regulating species: mX
K:

Figure 20: A screenshot of the ”Add Reaction” dialog.

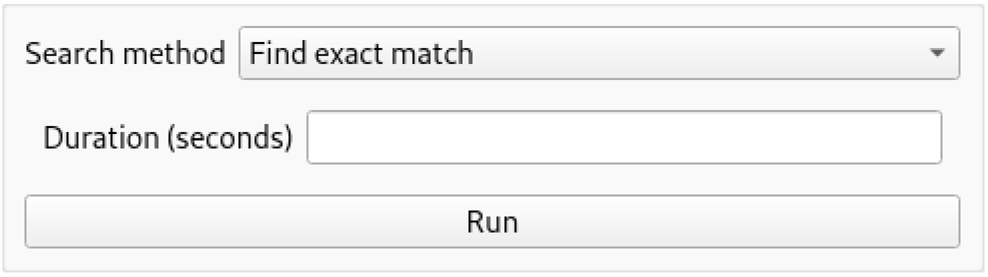
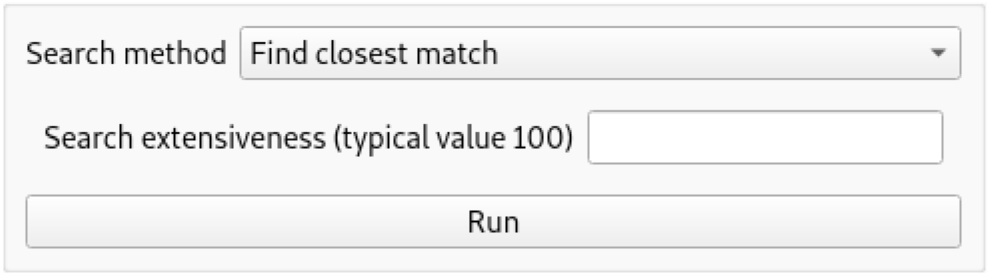


Figure 21: UI showing options for the search method ”Find exact match” which uses best-first search.

Exporting to and importing from SBML files are done through ”Open SBML file” and ”Save SBML file” options under the file menu in the top menu bar. Both use *PyQt*’s file dialog features which use the operating system’s native open and save file dialogs to let the user choose the file path of the SBML file to open or the path of the SBML file to save to, respectively.

Another important consideration was making some technical options as accessible as possible to the non-technical user. As an example, consider the two algorithms used for performing constraint satisfaction: best-first search and simulated annealing, whose options that are presented on the UI are shown in figures 21 and 22 respectively. Rather than present the user with the name of the algorithm used in each case (which the user does not need to know), the effect of using that algorithm is shown. For example simulated annealing algorithm, as explained before, aims to find the network which comes closest to satisfying all the constraints, therefore the appropriate non-technical name for this search method is ”Find closest match”. Furthermore, the simulated annealing algorithm accepts a parameter called ”schedule” which essentially specifies how ”extensive” the search for a maximum/minimum in the function should be by controlling how fast the ”temperature” of the algorithm will be lowered. Of course such terms as ”schedule” or ”temperature” are meaningless to a user with no knowledge of this algorithm, however the user should still be able to control these parameters as it is useful to be able to choose the extensiveness of the search. Therefore the idea of a schedule was abstracted by a field called ”search extensiveness” and the user is told that a typical value would for the field would be 100 (to give them a reference point). If they want to make the search more extensive, they can increase this number without having to know how the algorithm works.

The program uses a Model-View-Presenter (MVP) architecture to connect the GUI and the back-end. Using an MVP architecture meant the data (the Model) and the user inter-


Search method: Find closest match

Search extensiveness (typical value 100)

Figure 22: UI showing options for the search method ”Find closest match” which uses simulated annealing.

face (the View) could be kept separate: the View component is composed of a set of GUI classes, each class containing the code for one window of the GUI. The *GenePresenter* class is a singleton class which holds the three significant parts of the Model component of MVP: `network` (of type *Network*, holds the currently loaded GRN), `mutables` (of type list of Mutable classes), `constraints` (of type list of Constraint). *GenePresenter* being a singleton meant the Model could be easily accessed from all the GUI classes to directly display the data in Model to the user and save the data captured from the user to the Model. If this hadn't been the case, a *GenePresenter* instance would have to be created when the program was first launched and the same instance would have to be passed to every new window/screen which is opened so that they could access/store data using this instance. This approach would be more inconvenient than using a singleton class whose instance is automatically accessible from every class.

3 Project Management

3.1 Timetable and Main Tasks

After the project was suggested by the project supervisor, an initial set of features were decided with the supervisor which led to the requirements as seen in section 1.4. This eventually led to an initial system architecture containing a set of interdependent modules or ”main tasks” that had to be implemented to accomplish the requirements. At the beginning of the project, the tasks given in table 1 (along with the requirements it intends to achieve) were planned.

Tasks	Description	Achieved
1 (a) Model of parts (b) Model of relations (c) Whole model	Refers to creating classes to represent various objects of the network in Python.	1
2 (a) Generate equations (b) Solve equations (c) Visualise results	Refers to translating reactions into differential equations which represent change in species quantities. Solving these allow simulation of the network and subsequently visualise the results on a graph.	2
3 (a) Convert constraints (b) find the right circuit	The user specifies constraints in a format convenient to the user. This step translates those to mathematical equations the program can work with and uses those to modify a network to find a network which satisfies all the constraints.	3

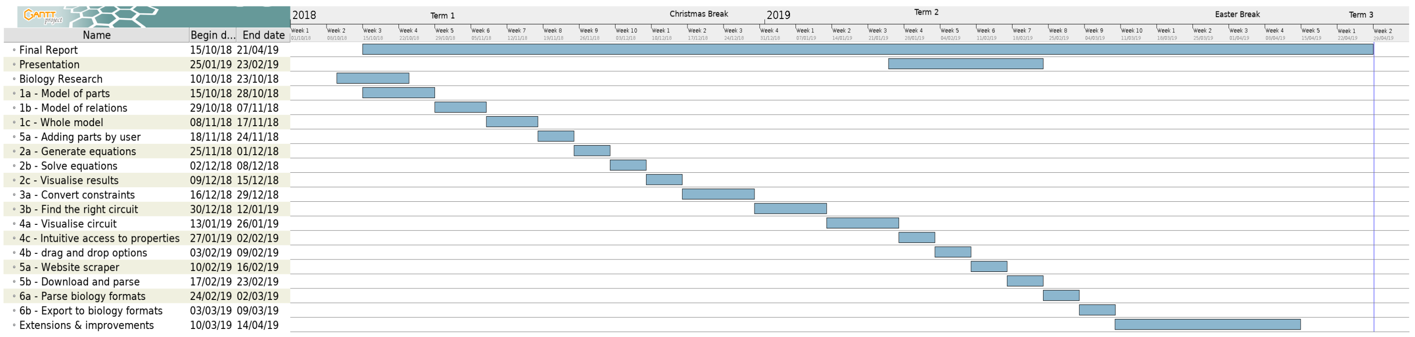
4 (a) Visualise circuit (b) Drag and drop options (c) Intuitive access to properties	This refers to visualising the genes and the regulatory relationships in a similar way to figure 1. User can use drag and drop options when creating networks. Intuitive access to properties refers to an intuitive user interface.	4
5 (a) Website scraper (b) Download and parse	Refers to downloading biological parts online for user to build networks. Website scraper scrapes relevant websites for parameters of such biological parts while parsing refers to turning these values into parts used in the network.	Not achieved
6 (a) Parse biology formats (b) Export to biology formats	Refers to importing from and exporting to SBML files.	5
Extensions & Improvements	Extension features and improvements to already implemented features.	Not applicable

Table 1: A table of main tasks, their description and the requirements they were aiming to achieve.

Task 5 in table 1 requires more explanation: At this stage in the project, the very start, the network model was conceptualised as a set of genes, rather than the final implementation as discussed in section 2.2. In this early concept, each gene would be modelled as a combination of its constituent parts: its promoter (to specify properties of its transcription), its coding region (to specify the mRNA the gene codes for) and so on. The user would then be able to download parts, such as promoters, etc. from online databases to build individual genes. This concept was later decided against, as is discussed in more detailed later.

Once the main tasks were laid out, the next step was to decide on the project timetable.

The initial timetable is shown in figure 23. Various strategies were used to determine the final version of the initial timetable. The tasks were dependent on each other in a hierarchical way (for example constraint satisfaction being based on simulation which in turn is based on modelling and so on), therefore tasks at the bottom of the hierarchy with no dependencies were placed at the very start of the project followed by tasks which depended on the previous tasks and so on. Due to the highly interdependent nature of tasks, parallelism was usually not possible. Tasks which were of less priority due to not being essential features were placed at the end of the timetable, as these had the highest probability of not being implemented due to time constraints. For example a website scraper to automatically import network parts online is a feature to improve user experience, but is not strictly necessary. When deciding on the time required for completing each task, the ones whose specific details were less clear and hard to predict (due to being inexperienced with this project at the start) were allowed more time. In general, the time required for most tasks were also purposefully overestimated to ensure as little problem with timing as possible. A final task called ”Extensions & Improvements” was also added. programs usually contain many bugs and require extensive testing to find and fix these. Moreover, there is always room for improvement. It was planned that bug fixes and improvements (mostly relating to user experience) would be done during this task.

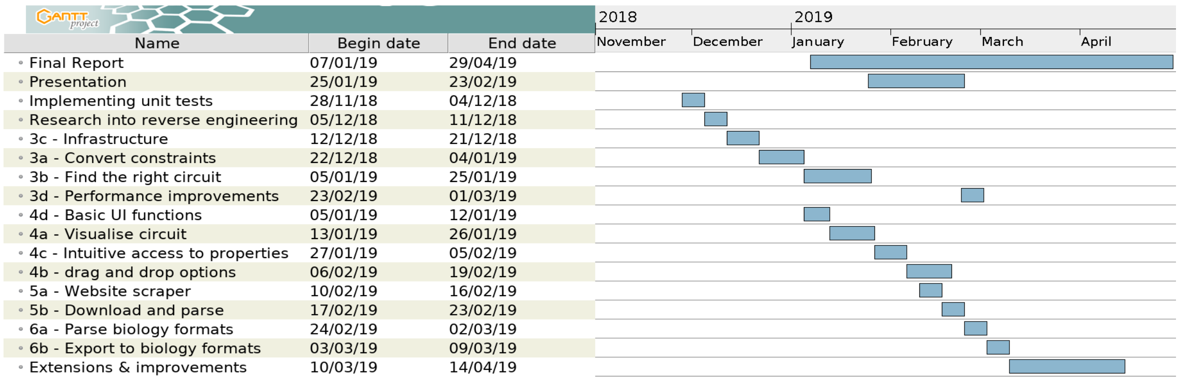


Name	Begin d...	End date
Final Report	15/10/18	21/04/19
Presentation	25/01/19	23/02/19
Biology Research	10/10/18	23/10/18
1a - Model of parts	15/10/18	28/10/18
1b - Model of relations	29/10/18	07/11/18
1c - Whole model	08/11/18	17/11/18
5a - Adding parts by user	18/11/18	24/11/18
2a - Generate equations	25/11/18	01/12/18
2b - Solve equations	02/12/18	08/12/18
2c - Visualise results	09/12/18	15/12/18
3a - Convert constraints	16/12/18	29/12/18
3b - Find the right circuit	30/12/18	12/01/19
4a - Visualise circuit	13/01/19	26/01/19
4c - Intuitive access to properties	27/01/19	02/02/19
4b - drag and drop options	03/02/19	09/02/19
5a - Website scraper	10/02/19	16/02/19
5b - Download and parse	17/02/19	23/02/19
6a - Parse biology formats	24/02/19	02/03/19
6b - Export to biology formats	03/03/19	09/03/19
Extensions & improvements	10/03/19	14/04/19

Figure 23: The original timetable proposed at the start of the project.

During the writing of the progress report, the timetable and the main tasks were updated. [35] One significant change was addition of a degree of parallelism where tasks related to GUI and constraint satisfaction were developed in parallel as they did not depend on each other. Furthermore, as the project progressed and more parts of it had been implemented, certain problems with the timetable became apparent. As a result, some new tasks were added to the ones given before (as seen in the updated timetable in figure 24):

- **3 (c) Infrastructure:** The infrastructure necessary for the constraint satisfaction module, such as internal representation classes for constraint satisfaction related objects.
- **3 (d) Performance Improvements:** As it became apparent that the process would likely be computationally costly at first, this task would aim to alleviate this problem.
- **4 (d) Basic UI functions:** The basic UI structure containing buttons, text boxes etc. to specify parameters and perform actions.
- **Research into reverse engineering:** At this stage in the project, the constraint satisfaction feature was called ”reverse engineering”. This task would involve spending some time to research methods and algorithms necessary for implementing this feature, as this was the novel part of the project and was relatively more difficult than other parts.



Name	Begin date	End date
Final Report	07/01/19	29/04/19
Presentation	25/01/19	23/02/19
Implementing unit tests	28/11/18	04/12/18
Research into reverse engineering	05/12/18	11/12/18
3c - Infrastructure	12/12/18	21/12/18
3a - Convert constraints	22/12/18	04/01/19
3b - Find the right circuit	05/01/19	25/01/19
3d - Performance improvements	23/02/19	01/03/19
4d - Basic UI functions	05/01/19	12/01/19
4a - Visualise circuit	13/01/19	26/01/19
4c - Intuitive access to properties	27/01/19	05/02/19
4b - drag and drop options	06/02/19	19/02/19
5a - Website scraper	10/02/19	16/02/19
5b - Download and parse	17/02/19	23/02/19
6a - Parse biology formats	24/02/19	02/03/19
6b - Export to biology formats	03/03/19	09/03/19
Extensions & improvements	10/03/19	14/04/19

Figure 24: The updated timetable proposed in the progress report.

Towards the end of the project, it was decided with consultation with the project supervisor that some tasks and features would not be implemented as some of the project priorities changed. Most notably, the feature of being able to download real network parts as found in real organisms was dropped as it was decided this was an extra feature and the project should focus on core features at this stage. The drag-and-drop GUI features were also dropped for the same reason, as it would require a very long time and redirect focus to a non-essential feature. It could be argued that neither features added any functionality which could not be achieved by the user in other ways: in other words, they only serve to improve usability. For example the user can still find out about the parameters of a network part and add them as a reaction object manually.

Finally, while both timetables mention ”Final Report” as a task, the work on the final report did not start until the Easter holiday due to a) the skills necessary for the writing of the final report not having been learnt until lectures and workshops later in term 2 and b) most details not having been finalised until the end of term 2.

In the specification document for this project, it was stated that deterministic simulation was chosen as a reasonably accurate method of simulation while an improvement on this would be to implement a stochastic simulation to provide a more accurate simulation. It was decided that stochastic simulation would be considered if deterministic simulation took less time than expected, leaving time in the timetable for the stochastic simulation's implementation.

In the end, deterministic simulation was successfully implemented ahead of schedule, leaving enough time for implementation of stochastic simulation. In the current version of the program, deterministic and stochastic simulation options are available alongside each other.

In conclusion, the timetable was approximately followed and all the main core tasks/features were implemented by the end of term 2 with small bug fixes left to fix over the Easter holiday. There were a couple of features (such as the drag-and-drop GUI, etc.) which had to be dropped. However this does not render project management a failure because the dropped features were usually ”optional” features which improved usability but did not contribute towards the core requirements of the program. Furthermore, redirecting resources from these optional features to core features meant that there was more time to make sure the core features were correctly implemented and perfected.

3.2 Software Development Methodology

The software development methodology used was a variant of incremental development. Development was done in cycles of varying lengths:

1. Implement new component/subcomponent.
2. Test whether it works as expected.
3. Improve until it does work as expected.
4. Refactor the code to make it more readable before going back to step 1.

This methodology worked very well with this particular project. The various components of the system and the functionality that each of them was expected to add to the system were well-defined. As a result, each component/subcomponent could be associated with a specific development cycle and thus lead to an organised development process.

The second point of the methodology refers to manual unit testing of the newly implemented component. This was not done using any unit testing frameworks but simply by writing a ”main” method inside the class being tested. The testing procedure in most cases involved creating a small test network (and/or other test data if needed) to ascertain the component's implementation's correctness by manually examining its output and verifying that it is the expected output and/or the function works as expected. This was a good strategy with components producing small amounts of output (such as a small list of numbers) as it could be compared with the expected output which could be computed by hand. However for some components producing much larger amounts of data (such as deterministic simulation of a network) or one which produces randomly distributed data (such as stochastic simulation), this was not possible. With such components, the strategy was to instead examine the patterns found in the data rather than individual data points. For example the visualisation of deterministic simulation results revealed a graph which was visually compared with the simulation of the same network in another simulation program (such as COPASI) which is known to work correctly to test the function's correctness.

The code written in step one was usually only concerned with implementing the necessary feature and thus writing clean, readable code which follows best programming practices was not a priority. The last step of the methodology was therefore a crucial one in ensuring the written code would remain readable and easy to work with so that the subsequent features could be built on the current codebase easily. Refactoring techniques used included adding more abstractions (such as breaking up a very large function into smaller functions), making variable/function names more descriptive, and introducing separation of concerns by separating large sections of code into separate classes. Refactoring became increasingly more important as the codebase grew in size; in fact towards the end of the project, step 3 of the methodology involved spending a significant amount of time refactoring the code to make fixing the problems with the implementation of the new feature easier or even possible.

3.3 Backup Strategy

The project made use of a version control software (*Git*) which was regularly pushed to a private remote repository hosted by *GitHub*, therefore the project files were backed up both at a local and remote location.

4 Evaluation

The project can be evaluated in various ways: Have most of the core functions been implemented? Have they been implemented correctly? Some originally planned features may have been changed or dropped completely; does the current form of the project still satisfy the objectives of the project? What about the usability; is it user friendly and easy to use?

Since this project is intended to be used by an end user, it is appropriate to evaluate its correctness in two categories: functional (does it correctly satisfy all the stated requirements as stated in section 1.4) and usability (is the UI intuitive and easy to use?). This section details evaluation of large, core functionality which combine many smaller units rather than individual functions, which was the aim of the manual unit tests as explained in section 3.2.

4.1 Functional evaluation

Requirement 1: User should be able to design a GRN. The network whose diagram is shown in figure 25 was implemented in this program. Referring back to the sub-requirements of this requirement, this network has genes, specifies the required types of reactions and values for parameters, and both mRNA and proteins do have different names in this design. Therefore somehow being able to prove this network's implementation using this program is correct would also prove that design features have been implemented correctly. The fact that the evaluation of deterministic simulation (as explained later) provides a level of confidence that this function is working as expected.

Requirement 2: User should be able to simulate a GRN. The evaluation method used for this requirement was to pick two networks, whose network diagrams are shown in figures 25 and 26. The former was designed using this program, while the latter was imported from an SBML file for the Repressilator [36].

This allowed testing simulation of both networks designed in this program and those which have been imported from an SBML file. Subsequently, both networks were replicated in COPASI (the former through exporting the file from this program and later importing it to COPASI, while the latter was already in SBML format and was also imported to SBML).

The visualisation for the simulation results belonging to the first network are shown in figures 27 and 28 comparing this program's simulation and COPASI's respectively, while the results for the second network are shown in figures 29 and 30. Both cases show a clear match between this program's deterministic simulation and the deterministic simulation done by

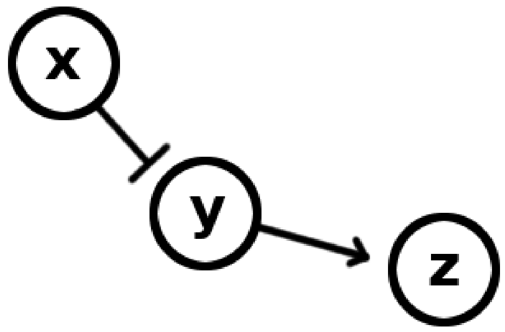


Figure 25: The network diagram for the first network.

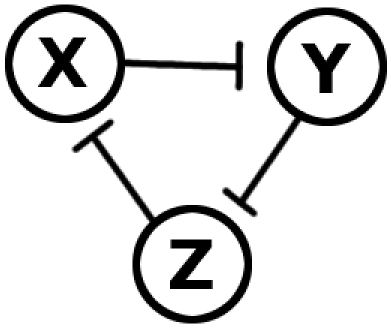


Figure 26: The network diagram for the second network.

COPASI. Therefore assuming that COPASI can simulate networks correctly, this program's deterministic simulation is also likely to be correct and this requirement is satisfied.

Stochastic simulation

Referring back to requirement 2, a deterministic simulation is enough to achieve this requirement. However, as described earlier, stochastic simulation was also implemented and should also be evaluated.

The first test involved the network seen in figure 25 again. The stochastic simulation of the network was performed both in this program and COPASI.

The results are shown in figures 31 and 32 for this program and COPASI respectively. There is a clear mismatch between the results. Upon analysing the results of this test, the Gillespie algorithm in this program was checked for correct implementation without any obvious errors discovered.

While trying to diagnose the problem, further tests were done with various different networks. Most notably, see the results of stochastic simulation of one of the networks in this program and COPASI in figures 33 and 34. There is a much closer match between the

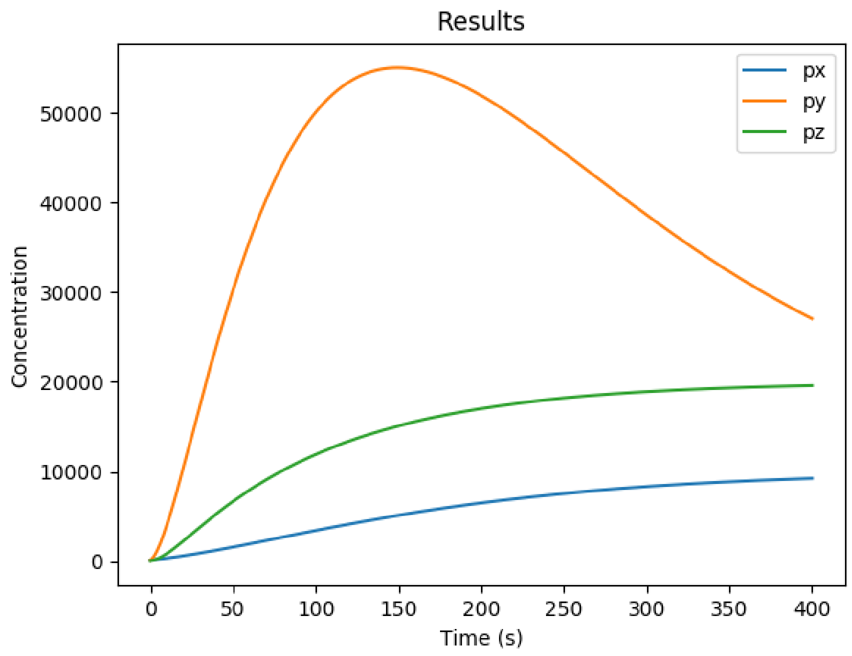


Figure 27: Visualisation of the simulation of the first network in this program.

simulation results this time. Further investigation revealed the problem is likely related to the number of reactions in each network, where the first test network had 12 reactions while the second one (which matched the COPASI simulation much more closely) had 6 reactions.

Requirement 3: User should be able to perform constraint satisfaction of a GRN. Evaluation of this feature involved two tests using a single network whose simulation results (before any constraint satisfaction) are shown in figure 35.

The first test involved only one constraint ($pY <= 150,000$ in time range 150-400 seconds) and one mutable (pY translation rate: 0 to 20 with increments of 1). To prove that constraint satisfaction works, the process has to be run to receive the parameters of a network which satisfy all the constraints. Subsequently, it would be fairly simple to simulate a network with the received parameters to check that the new network indeed does satisfy all the constraints. The process successfully finds a matching network which is shown in figure 36.

The second test involved three constraints, so that it could be tested whether more than one constraints are handled correctly as well. The test involved these constraints:

1. $pY <= 150,000$ in 150-400 seconds (as before)

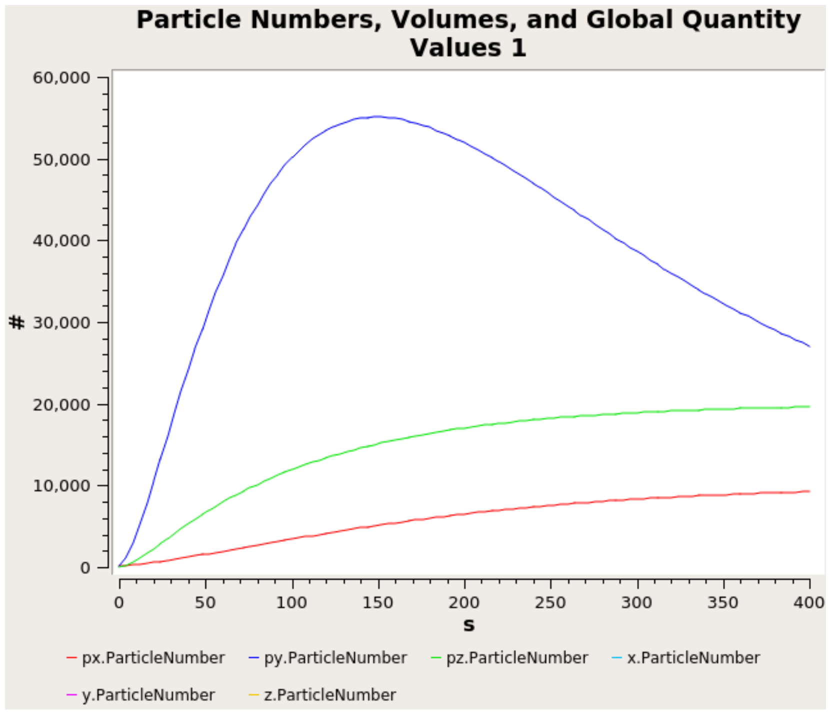


Figure 28: Visualisation of the simulation of the first network in COPASI.

2. $pY >= 50,000$ in 150-400 seconds
3. $pH >= 10,000$ in 250-300 seconds

and these mutables:

1. pY translation rate: 0 - 20 with increments of 1 (as before)
2. pH degradation rate: 0 - 0.1 with increments of 0.001

The program was able to find a matching network with parameters pH degradation rate = 0.001 and pY translation rate = 2.0, where the simulation of this modified network was as shown in figure 37.

The tests show that the program was able to find a valid solution. These results provide some confidence that the feature works as expected. However these tests do not check whether the program reports any false negatives (i.e. there is a solution, but the program reports failure) as this would require a person to manually check that given these constraints and mutables there actually is no solution. This would require a significant amount of time, which was not anticipated and not planned for, therefore could not be done within the time constraints of the project.

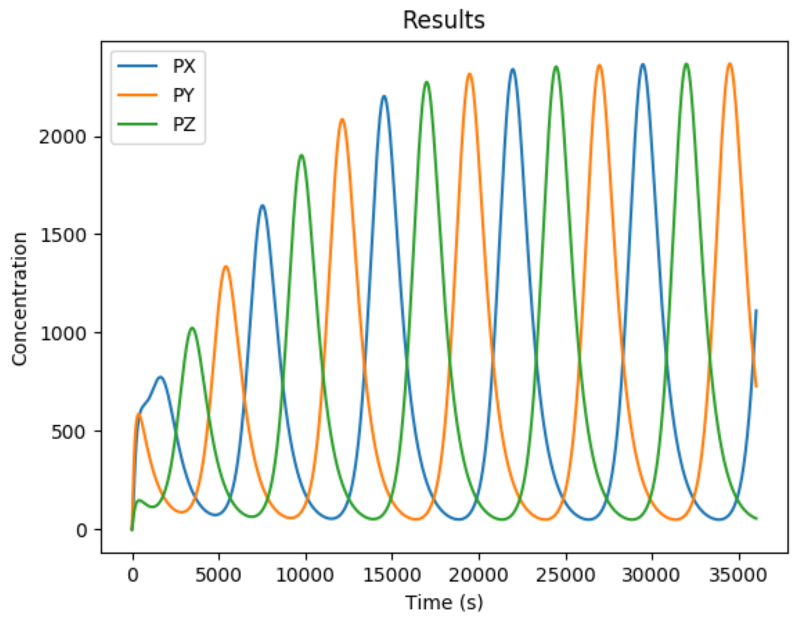


Figure 29: Visualisation of the simulation of the second network (Repressilator) in this program.

The tests do still give some confidence that there are no false positives (because all the reported results were manually checked and found to actually satisfy all the constraints and so should not contain any false positives).

Requirement 4: program should have a reasonably good user interface. See section 4.2.

Requirement 5: User should be able to import from and export to SBML. The export function was tested by exporting the network whose diagram is shown in figure 25 and which has been designed in this program to SBML. Subsequently, the file was imported to COPASI. COPASI performs a form of SBML validation, therefore this process would check the validity of the SBML. Furthermore, simulating the imported SBML on COPASI (and receiving matching results as the simulation performed in this program) would provide some confidence of the correctness of the exported SBML correctly describing the network as designed in this program. The exported network contained most of the supported design features this program offers: species, three types of reactions (transcription, translation, degradation), regulation between genes, etc. and therefore the correct exporting of these features were

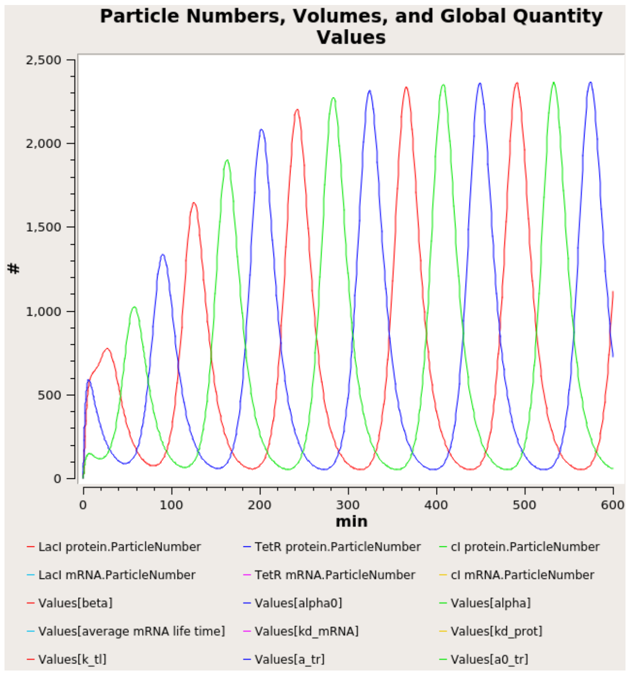


Figure 30: Visualisation of the simulation of the second network (Repressilator) in COPASI.

tested by this test.

The test revealed a number of errors in the SBML export function. To begin with, COPASI raised an error warning that certain parameters were missing from the SBML file which meant the model could not be understood. These were then addressed by adding the required parameters during the SBML export process in this program and the test was conducted again, without any errors from COPASI. However, simulating the imported SBML revealed another error. Comparing figure 27 and figure 38 shows that there is a mismatch in the results, even though the shape of the graphs match. It was understood later that this was because the unit for species quantities was not specified in the exported SBML file and was assumed by COPASI to be mol/l, while this program uses ”number of molecules” as a unit for species quantities. This was causing the simulation results to have incorrect units. This problem was also addressed. The final test conducted (after addressing all the aforementioned problems) resulted in the simulation results visualised in figure 28, which, as explained earlier, are the expected results.

As described earlier, an SBML file for Repressilator was imported for testing whose results matched that of COPASI’s. Assuming COPASI’s SBML import function is correct, this provides some confidence that the import function has been implemented correctly as well.

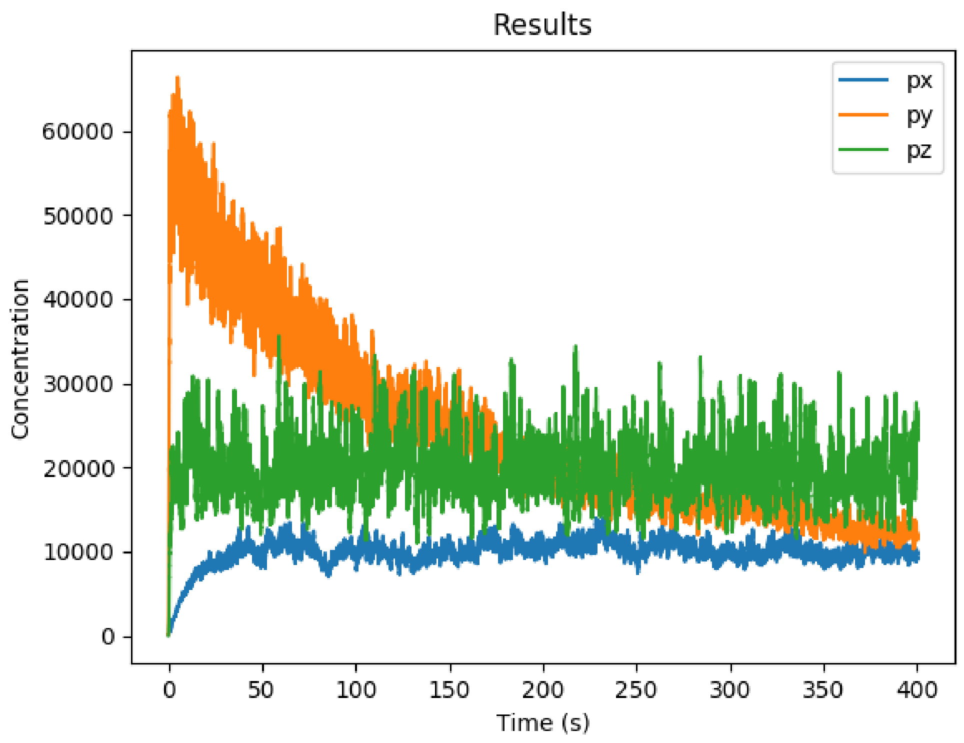


Figure 31: Visualisation of the stochastic simulation of the first network in this program.

4.2 User Acceptance Testing

A user acceptance test (UAT) was conducted with a third year university student studying Medicine who had knowledge of gene expression and gene regulation but not systems biology or computer science. He was explained the basic features of the program, given a brief explanation of the user interface and was allowed to explore the program. He could ask for assistance and clarifications during the test.

His first remark was that he felt lost when he first opened the program's user interface and didn't know how to go about doing what he wanted. He later explained that this was due to the fact that there was not enough help information available guiding the user. Upon these remarks, he was explained where to find the basic features of the program on the user interface. After getting a taste of the program, he reported that the various features of the program were in fact relatively easy to understand after a few times of trial and error, but also suggested that including text in the program to explain the purpose of some fields or help information to explain the program would have made the program easier to understand.

Another remark he had was the absence of units of chemicals, which he explained would

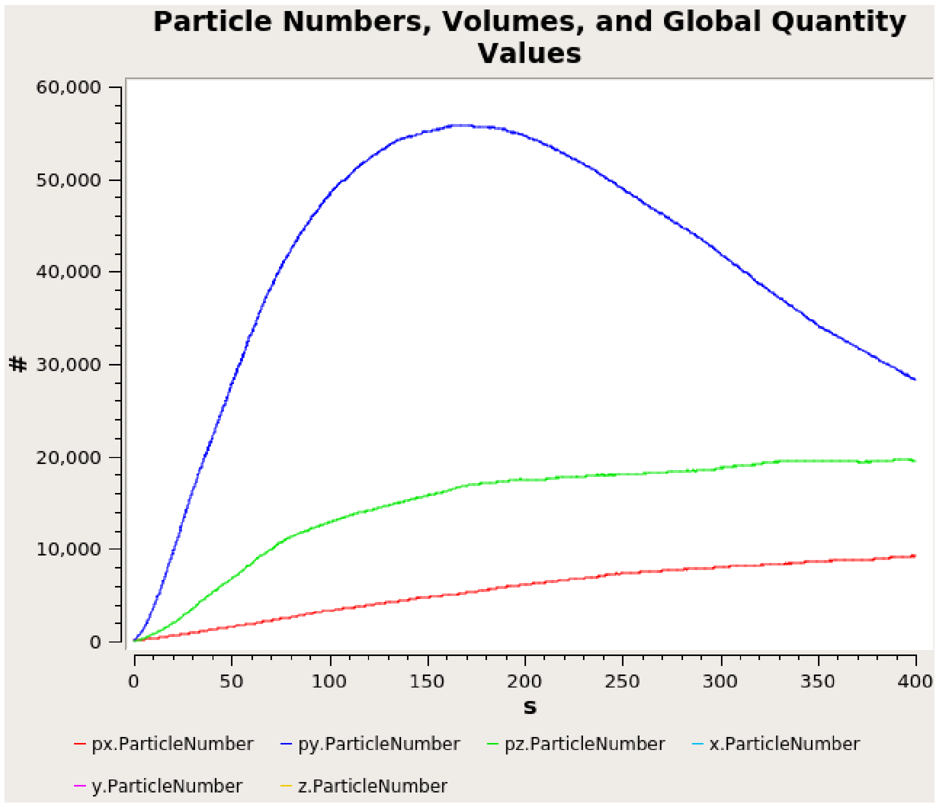


Figure 32: Visualisation of the stochastic simulation of the first network in COPASI.

be necessary to convert available data into the right units.

He was generally satisfied with the simulation feature but given he had no systems biology background, found the names deterministic and stochastic simulation confusing. He also expressed his opinion that the constraint satisfaction feature of the program was a very useful feature.

The received feedback was incorporated into the program in these ways:

1. More help information was added into the UI, as well as a help menu to offer guidance the user.
2. Units for chemicals and other values were specified in the help menu for the program, as was suggested by the tester.
3. Some technical names were changed/amended to be more accessible to people with no computer science/systems biology background.

In conclusion, given that the tester was generally satisfied with the user interface except

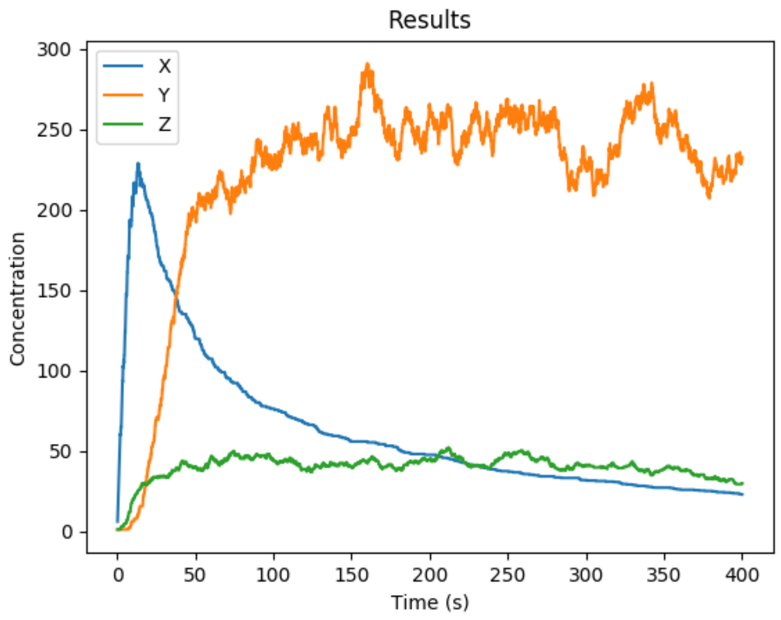


Figure 33: Visualisation of the stochastic simulation of the second network in this program.

the points he raised, addressing those problems (which have been done) will give some confidence that requirement 4 in section 1.4 has been achieved.

4.3 Example Usage

It is appropriate to describe an example usage of the program to demonstrate its various features. For example, say a researcher is investigating the network whose network diagram is shown in figure 39. The researcher doesn't have an SBML file of the model, so has to create the model from scratch using this program's user interface and design features.

The network consists of three genes: x, y, and z. The user first has to start by defining the species which will be present in the model. As explained before, each gene corresponds to two species, one for the gene's mRNA and another for its gene expression product. Therefore, the species shown in figure 40 and their corresponding initial quantities were defined where, for example, `x` and `px` refer to x's mRNA and protein product respectively.

Next step is defining the reactions in the network. In this case, each gene is associated with four reactions (transcription, translation, degradation for both mRNA and product). For example for gene y these reactions are defined:

1. $\emptyset \rightarrow y$: Transcription of the gene.

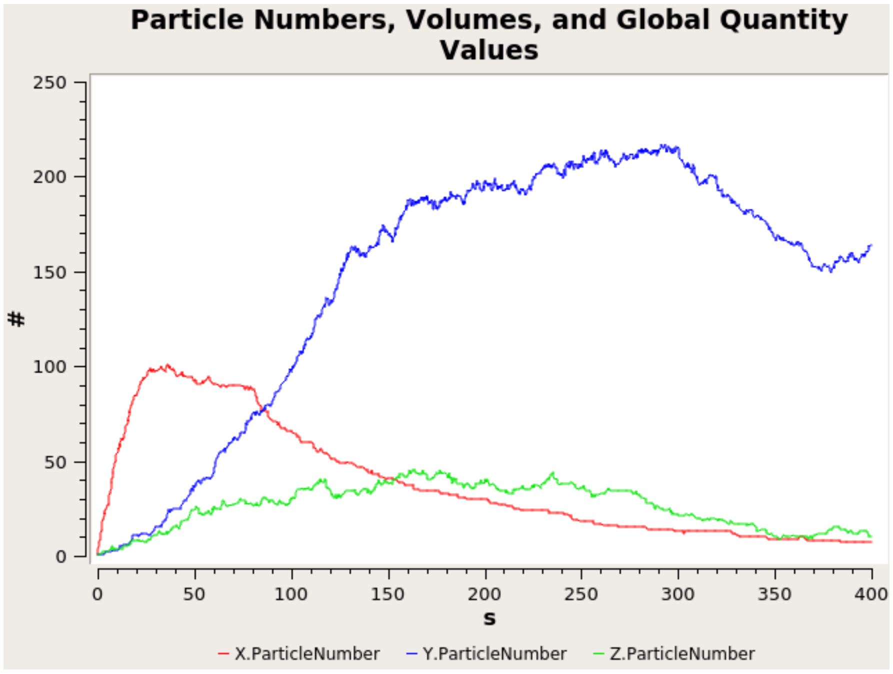


Figure 34: Visualisation of the stochastic simulation of the second network in COPASI.

2. $y \rightarrow py$: Translation of y's mRNA to its product, py.
3. $y \rightarrow \emptyset$: Degradation of y's mRNA.
4. $py \rightarrow \emptyset$: Degradation of y's product, py.

Figure 41 shows adding transcription reaction for gene y (reaction 1) along with the parameters it takes. This gene's mRNA species was defined earlier to be y. The "transcribed species" field ties the species y to this transcription reaction, i.e. describes y's transcription.

y is regulated by gene x (or more precisely the transcription factor px which is the product of gene x). To express this relationship in this program, the "Is Regulated" check box is checked to enable the regulation related fields. Subsequently, the "Regulations" field of y's transcription reaction has to be set. Figure 41 shows the "Regulations" field containing the item "$px \dashv y$" to represent gene x's product (px) regulating (repressing) transcription of y, as well as the "Add regulation" options at the bottom used to add it.

The user may then want to do a deterministic simulation of the network using the "Simulate" menu on the menu bar, which brings up the dialog shown in figure 42, letting the user choose which species' results to show and for how long to simulate the network. This produces the results whose visualisation is shown in figure 43.

Upon seeing the simulation results, the user may want to modify the network so that pY is always $\leq$ 50,000 in the first 400 seconds of the simulation, but also wants pY to be $\geq$

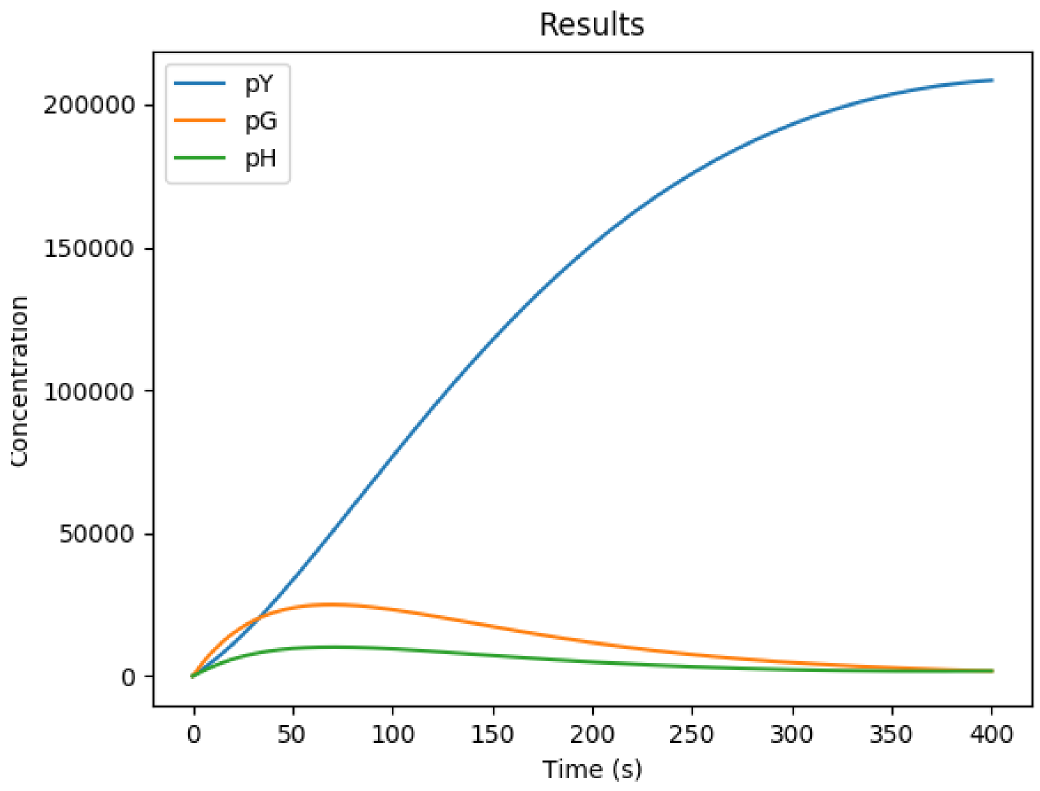


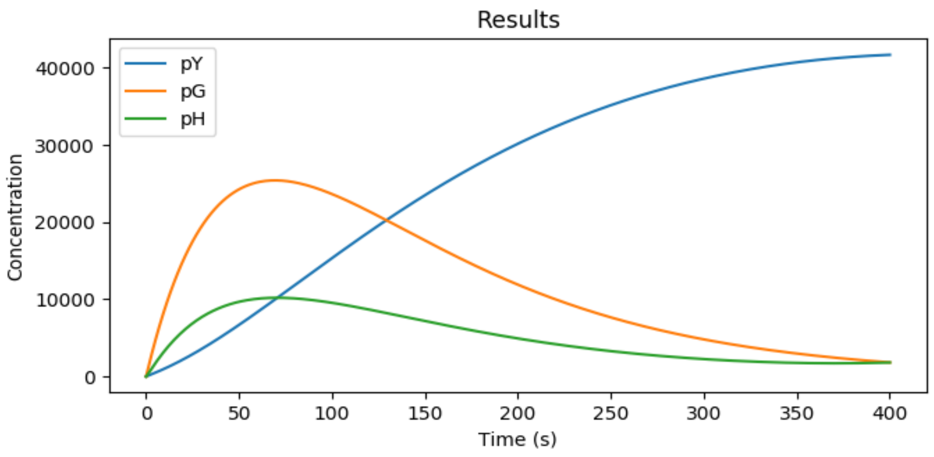
Figure 35: Visualisation of the simulation results of the network before any constraint satisfaction was done.

10,000 in the time range 100 - 400 seconds. The constraint satisfaction feature can be used for this. The user would add the first constraint using the diagram shown in figure 44 and after adding the second one as well arrive at the constraints screen shown in figure 45.

Next step is to define mutables: the network parameters which are allowed to vary. In this case, the user only wants to vary the translation rate of y ($y \rightarrow py$) which is added using the dialog shown in figure 46.

Once all the properties have been defined, the user chooses 20 seconds as the duration of the process and use the run button, which is visible on figure 45, to start the process. Once the program finds a suitable network, its parameters are displayed on the screen on a window such as the one seen on figure 47 which shows that a translation rate of 0.5 will satisfy the constraints.

Once the user clicks OK, they are presented with another screen (shown in figure 48) showing the simulation results of the modified network and its network visualisation.



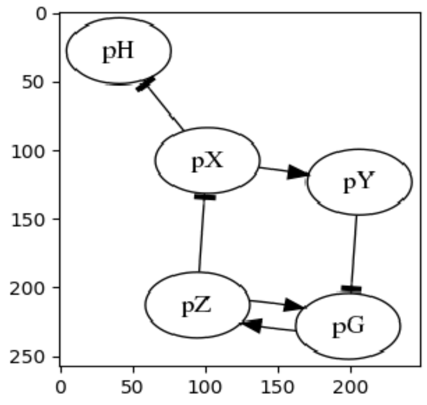


Figure 36: The result of running constraint satisfaction process with one constraint.

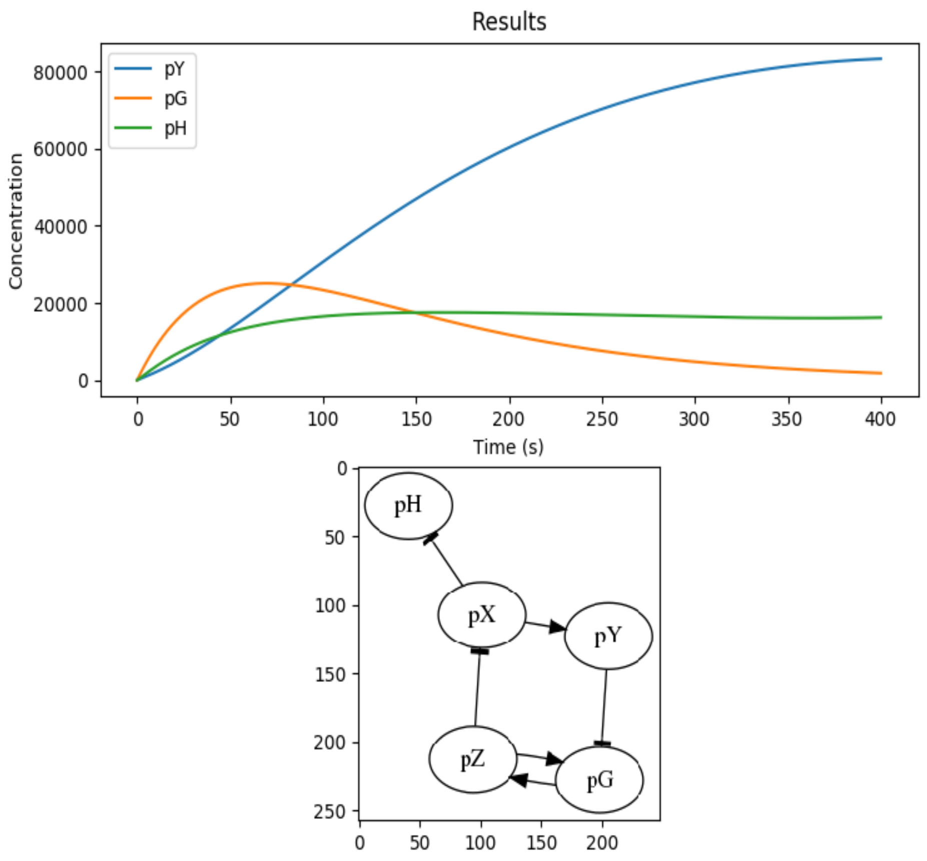


Figure 37: The result of running constraint satisfaction process with three constraints.

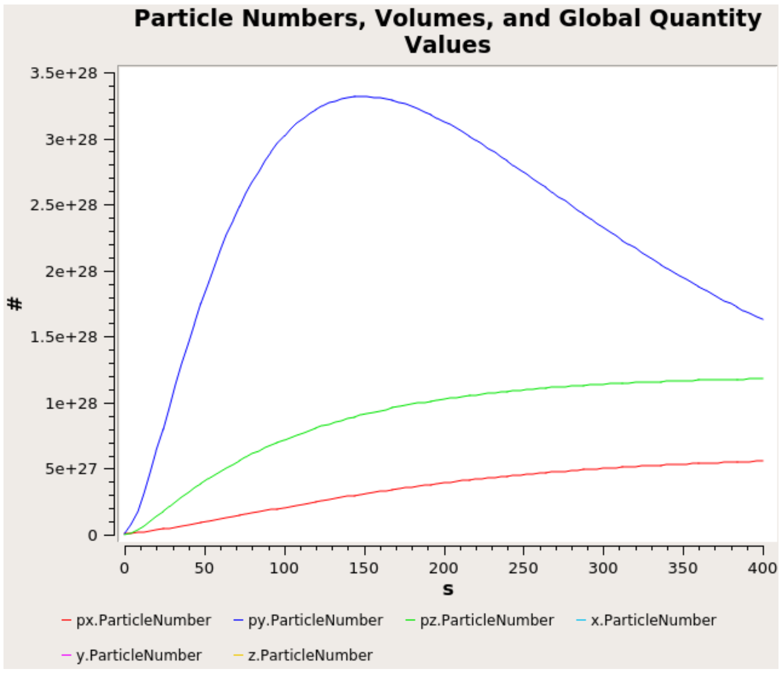


Figure 38: Visualisation of the simulation of the first network in COPASI which used incorrect units for the y-axis quantity.

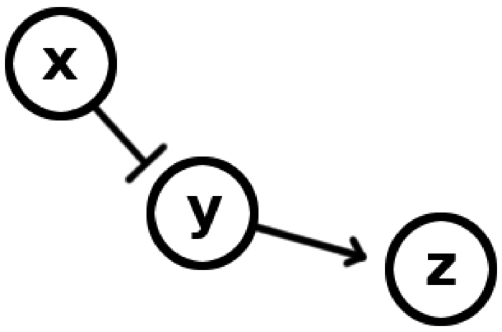
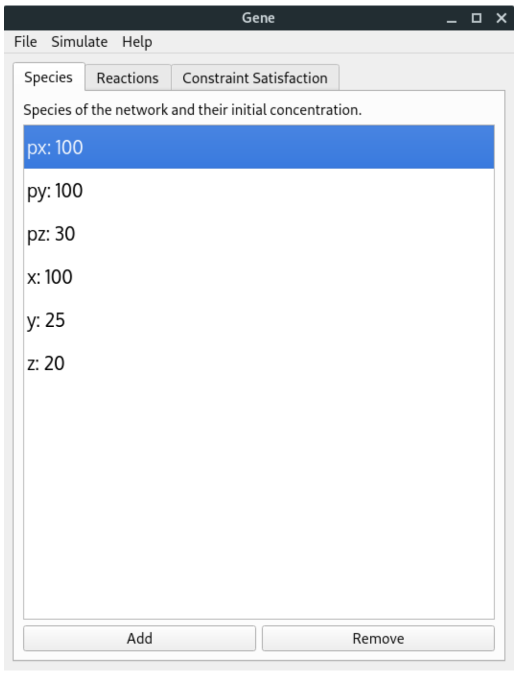


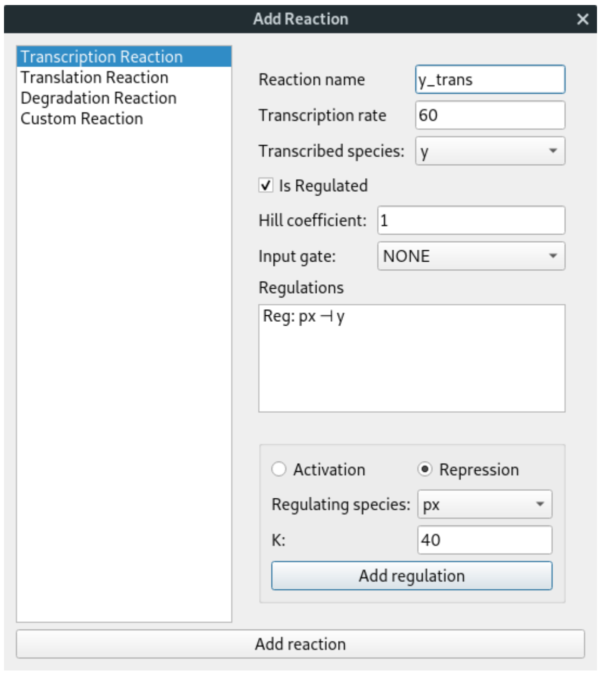
Figure 39: The sample network that is being investigated.



Species of the network and their initial concentration.

px: 100
py: 100
pz: 30
x: 100
y: 25
z: 20

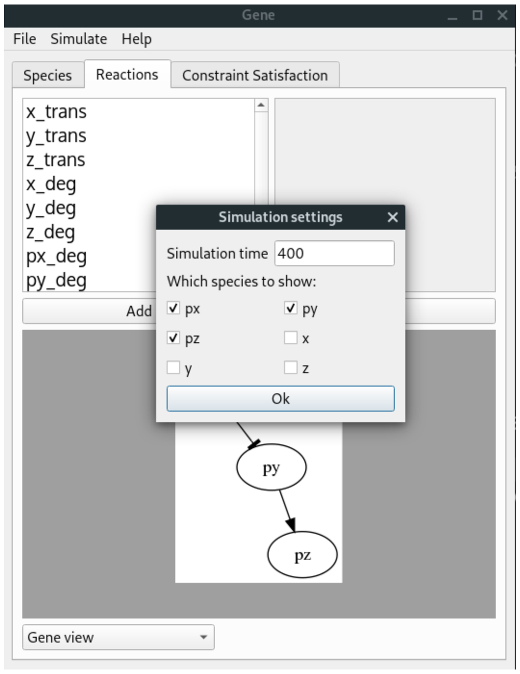
Figure 40: Species screen showing the added species and their corresponding initial quantities.



Reaction name: y_trans
Transcription rate: 60
Transcribed species: y
[✓] Is Regulated
Hill coefficient: 1
Input gate: NONE
Regulations
Reg: px ⊣ y

Activation [● Repression]
Regulating species: px
K: 40

Figure 41: Add reaction dialog showing transcription reaction properties.



Simulation time 400

Which species to show:

✓ px ✓ py
✓ pz ☐ x
☐ y ☐ z

Figure 42: Dialog showing simulation settings.

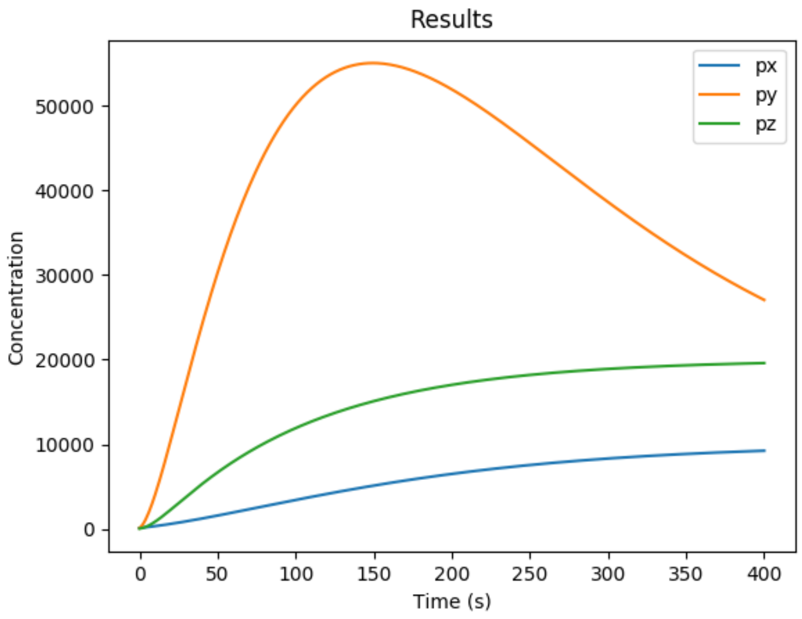
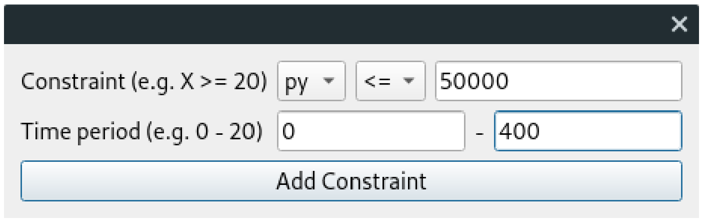
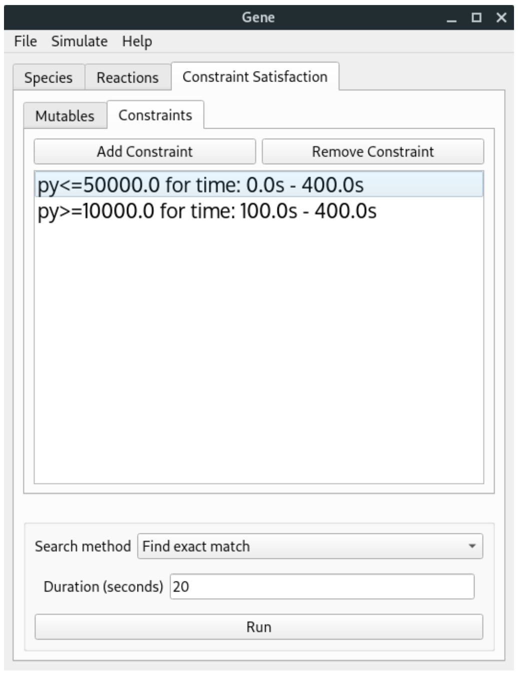


Figure 43: Visualisation of simulation results for the sample network.


Constraint (e.g. X >= 20) py <= 50000

Time period (e.g. 0 - 20) 0 - 400

Figure 44: Dialog showing the addition of the constraint $pY \leq 50,000$.


Gene
File Simulate Help
Species Reactions Constraint Satisfaction
Mutables Constraints
Add Constraint Remove Constraint
Run

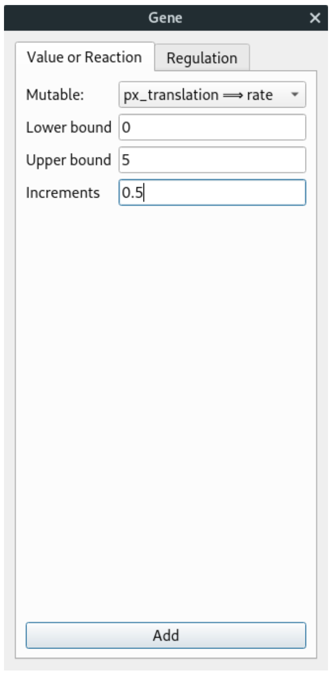
py<=50000.0 for time: 0.0s - 400.0s

py>=10000.0 for time: 100.0s - 400.0s

Search method Find exact match

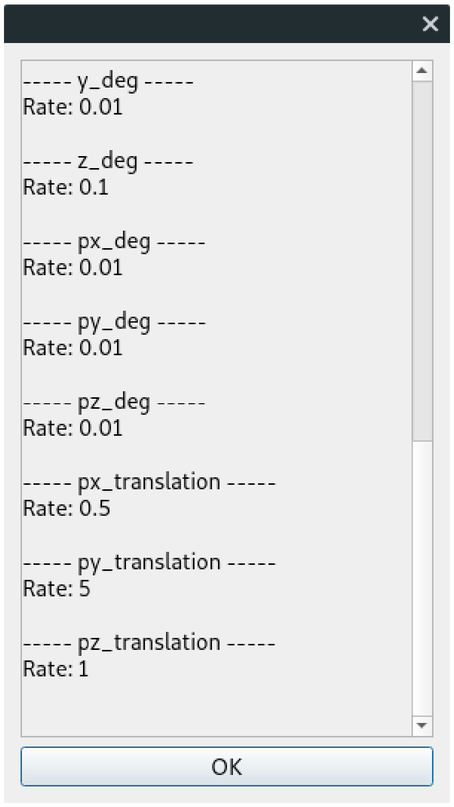
Duration (seconds) 20

Figure 45: Constraints screen showing all the constraints added by the user.



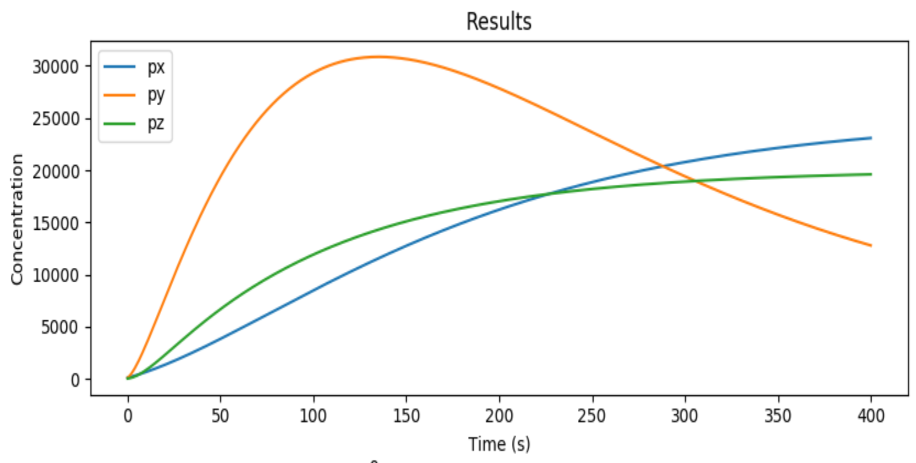
Mutable: px_translation ⟹ rate
Lower bound 0
Upper bound 5
Increments 0.5

Figure 46: Add mutable dialog showing the user defining translation rate of py as a mutable.



```
----- y_deg -----
Rate: 0.01

----- z_deg -----
Rate: 0.1

----- px_deg -----
Rate: 0.01

----- py_deg -----
Rate: 0.01

----- pz_deg -----
Rate: 0.01

----- px_translation -----
Rate: 0.5

----- py_translation -----
Rate: 5

----- pz_translation -----
Rate: 1
```

Figure 47: Screen showing the new, modified values of the network which satisfies all the constraints.



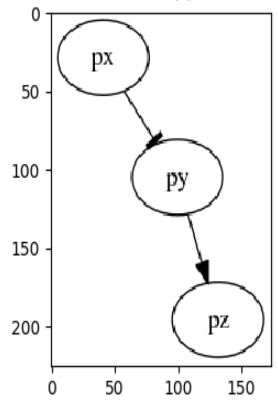


Figure 48: Screen showing the simulation and network visualisation of the modified network.

5 Conclusions

The main aim of this project was to create a computational biology tool to help researchers working on gene regulatory networks by providing facilities to automatically create networks capable of producing necessary products at desired amounts with an arbitrary set of constraints. The product of this project is a piece of software capable of this and acts as a good foundation for further research/development into computational tools to speed up design of synthetic networks and help systems biologists.

More specifically, referring back to section 3.1, some of the features which were planned for this project have been dropped: For example the software is not capable of downloading gene/network parts found in real organisms or able to let the user add these parts themselves for later use for that matter. However as discussed before in section 3, these were not essential features and their absence does not render the final software useless as it has still been able to implement all the core functionality such as simulation modelling, and most importantly constraint satisfaction. Therefore, it is useful to focus on the success of constraint satisfaction as it was the feature which motivated this project in the first place. In addition to core functionality, the code for the program has been written in such a way that it allows easy extension and thus it would be relatively easy to extend the capabilities of the program in the future. As is discussed later, there is a lot of room for improvement and exciting possibilities for future work, therefore the code being extensive is a definite advantage.

The main challenge and the difficult part of this project was the initial stage when a lot of systems biology knowledge had to be learned so that an adequate way of modelling and simulating GRNs could be devised. This proved to be an especially difficult task as the available systems biology resources were mostly written for other researchers and were not accessible for someone with no prior knowledge of this field and very little general biology knowledge.

Another challenge was working on a project at this scale for the first time, which required a great level of organisation and always necessitated writing readable and maintainable code because new code was being added almost daily and being able to understand code written months ago was crucial in integrating new features into the current codebase.

This project was a great opportunity to experience working on such a large project. This has inevitably led to some lessons learnt. This project did not take advantage of test driven development (TDD), which is the practice of writing automated unit tests (meaning the output of the unit is automatically checked against expected output) for the planned features/-functions even before the code itself has been written. This project instead used manual unit tests (so the output was manually compared by a real human to expected output) written after the implementation of the unit. This proved to be inefficient, as manual checking of output required a lot of time.

Another lesson learnt was that there is no perfect design which stays the same from the start to the end. During this project, the design for various features had to be changed due to unforeseen problems in the original design, or change of priorities and so on. The way to stay on top of all these changes was to write code in such ways that it could be changed without having to rewrite an entire portion of the code, in other words, it meant having to write adaptable code.

5.1 Future Work

Modelling The internal representation used for this project aimed to abstract many complicated aspects of real biological networks, however more detailed and accurate representations are possible. For example it is currently not possible to regulate any part of gene expression except transcription. It may be useful to provide a model of gene expression which gives finer control over the process in the future. Furthermore, even though transcription regulation is currently supported, a species can be regulated by at most two other species. In the future, a mathematical model allowing for more regulations could be implemented to allow modelling of real GRNs with such properties.

Simulation Stochastic simulation of networks is currently done using the Gillespie algorithm as discussed in section 2.3.2. Due to the fact that elapsed simulation time increasingly gets smaller as the number of reactions go up, stochastic simulation of networks above a certain number of reactions will start taking an unreasonable amount of time. There exists some optimisations for the Gillespie algorithm (such as Tau leaping) which can be implemented in the future to reduce the simulation time. Other possible solutions to this problem are discussed later in the ”Performance” paragraph.

User Interface The UI was found to be satisfactory by the user acceptance tester, however there is still room for improvement to make the UI more convenient. Firstly, a feature in many user interfaces is to remember the user’s chosen settings from when a feature was last used. For example the user currently has to pick which species’ simulation results to show from a dialog with check boxes before performing a simulation. It would be useful if when the user simulates the network again the species checked from last time were automatically checked again, as the user is likely to not change this setting from the last time.

The program has a large number of fields the user has to fill in. This can prove to be an inconvenience for the user. One potential solution could be to fill in some fields with default values, especially if a field is one whose value is usually not changed from a certain default value.

Lastly, the program only allows adding and removing of items (e.g. species, reactions, etc.) therefore if an item parameter needs to be changed, the only way is to remove it and add it again. Adding an edit option would solve this inconvenience.

Input/Output SBML offers options to add many optional metadata into network (for example a summary of the network, type of a reaction, etc.). The program currently does not fill in most of these metadata, however it could be useful if these were provided when exporting a network. Similarly, such metadata are not extracted when an SBML file is imported.

As was described earlier in section 2.5, a problem with SBML importing is that when mathematical expressions in the SBML file are being parsed, not all operands are recognised. Most notably, the SBML standard allows defining custom operands in the form of custom functions. These are not currently supported. A future version should be able to support more operands.

Performance Currently the program does not take advantage of parallelism by distributing workload to multiple processor cores. In the future, doing this might speed up simulation or the constraint satisfaction processes. It may also be beneficial to implement simulation and constraint satisfaction modules in C/C++ which are, unlike Python, not interpreted, not automatically garbage collected, and do not perform type checking during runtime and therefore will not suffer from the overhead associated with these.

Miscellaneous Currently, the program offers a default, unchangeable set of units used for all values entered, forcing the user to make sure all their values comply with these units. In the future, it may be useful to let the user choose which units to use for each quantity. Using a single unit also causes some problems when an SBML model is imported with different units to this program's default units, which should be solved in the future.

Furthermore, a useful feature for this program to have is the ability to automatically download and import a model from the BioModels database.

Implementation specific improvements As discussed before, the program uses an MVP architecture. However the main advantage of this architecture is a complete separation of the UI (the view) and processing (the presenter) meaning that ideally the view should call an appropriate function in the presenter for all processing. Currently however, this is only partially the case and taking advantage of this separation of concerns to a greater extent in the future would make the code more readable and maintainable.

6 Acknowledgements

I would like to thank my supervisor Dr Sara Kalvala who suggested this project topic and helped me learn the biology and systems biology knowledge I needed for this project (especially since I had no knowledge in any of these areas previously) and was always more than happy to provide very helpful feedback. I would also like to acknowledge that I have used Prof Nathan Griffiths' ”Informed Search and Local Search” lecture slides for CS255 from 2017 [34] to help me implement parts of the project requiring artificial intelligence techniques, such as best-first search and simulated annealing, latter of whose implementation in this programme is based on the algorithm given in the lecture slides [34, p. 65]. I would also like to acknowledge the help of the book by Vikram Singh and Pawan K. Dhar [30] at the start of the project for letting me get a grasp of ODE models of GRNs and the Repressilator simulation by Andrey Akinshin [37] which helped me understand how ODE models can be solved and simulated using computers. Finally, I would like to acknowledge the huge help that Uri Alon's book [5] has had on my learning of systems biology concepts at the very beginning of this project. The book was suitable for a beginner and was very effective in teaching systems biology to people coming from other disciplines. As a result, I was able to start this project with a strong background and understanding.

References

[1] "Synthetic biology." nature.com. https://www.nature.com/subjects/synthetic-biology (accessed Apr. 22, 2019).

[2] "Intro to gene expression (central dogma)." khanacademy.org. https://www.khanacademy.org/science/high-school-biology/hs-molecular-genetics/hs-rna-and-protein-synthesis/a/intro-to-gene-expression-central-dogma (accessed Apr. 28, 2019).

[3] "Transcription factors." khanacademy.org. https://www.khanacademy.org/science/biology/gene-regulation/gene-regulation-in-eukaryotes/a/eukaryotic-transcription-factors (accessed Apr. 28, 2019).

[4] "Regulation after transcription." khanacademy.org. https://www.khanacademy.org/science/biology/gene-regulation/gene-regulation-in-eukaryotes/a/regulation-after-transcription (accessed Apr. 29, 2019).

[5] U. Alon, *An introduction to systems biology: design principles of biological circuits*. Chapman and Hall/CRC, 2006.

[6] "Systems Biology as Defined by NIH." irp.nih.gov. https://irp.nih.gov/catalyst/v19i6/systems-biology-as-defined-by-nih (accessed Apr. 22, 2019).

[7] M. S. B. Team, "What is systems biology?." blog.nature.com. http://blogs.nature.com/sevenstones/2007/07/what_is_systems_biology_3.html (accessed Apr. 22, 2019).

[8] "What is Systems Biology." systemsbiology.org. https://systemsbiology.org/about/what-is-systems-biology/ (accessed Apr. 22, 2019).

[9] "About." sbml.org. http://sbml.org/About (accessed Apr. 29, 2019).

[10] "Biomodels database." https://www.ebi.ac.uk/biomodels-main/ accessed 09.10.2018.

[11] M. B. Elowitz and S. Leibler, “A synthetic oscillatory network of transcriptional regulators,” *Nature*, vol. 403, no. 6767, p. 335, 2000.

[12] “COPASI (4.24), Biocomplexity Institute of Virginia Tech , the University of Heidelberg and the University of Connecticut, UConn Health.” Accessed: Apr. 11, 2019. [Online]. Available: `http://copasi.org/`.

[13] “Gene Designer (2.0), Atum.” Accessed: Apr. 11, 2019. [Online]. Available: `https://www.atum.bio/resources/genedesigner`.

[14] “GenoCAD, GenoFAB.” Accessed: Apr. 11, 2019. [Online]. Available: `https://genocad.com/`.

[15] “TinkerCell (1.2.926).” Accessed: Apr. 11, 2019. [Online]. Available: `http://www.tinkercell.com/`.

[16] “Graphviz.” Accessed: Apr. 11, 2019. [Online]. Available: `https://graphviz.gitlab.io/download/`.

[17] “NumPy (1.15.2).” Accessed: Apr. 29, 2019. [Online]. Available: `http://www.numpy.org/`.

[18] “Matplotlib (3.0.3).” Accessed: Apr. 29, 2019. [Online]. Available: `https://matplotlib.org/`.

[19] “Graphviz (0.10.1), Sebastian Bank.” Accessed: Apr. 22, 2019. [Online]. Available: `https://graphviz.readthedocs.io/en/stable/index.html`.

[20] S. M. Keating, F. Bergmann, B. G. Olivier, L. P. Smith, and M. Hucka, “libsbml-5.17.0,” May 2018.

[21] “Qt, The Qt Company.” Accessed: Apr. 11, 2019. [Online]. Available: `https://www.qt.io/download-qt-installer?hsCtaTracking=9f6a2170-a938-42df-a8e2-a9f0b1d6cdce\%7C6cb0de4f-9bb5-4778-ab02-bfb62735f3e5`.

[22] “PyQt (5.11.3), Riverbank Computing.” Accessed: Apr. 11, 2019. [Online]. Available: `https://www.riverbankcomputing.com/software/pyqt/download5`.

[23] "TkInter (1.15.2)." Accessed: Apr. 29, 2019. [Online]. Available: `https://wiki.python.org/moin/TkInter`.

[24] "Gtk+ 3." Accessed: Apr. 29, 2019. [Online]. Available: `https://www.gtk.org/`.

[25] "PyCharm Community Edition (2019.1.1), JetBrains." Accessed: Apr. 16, 2019. [Online]. Available: `https://www.jetbrains.com/pycharm/`.

[26] "Git (2.19.1), Junio Hamano et al." Accessed: Apr. 16, 2019. [Online]. Available: `https://git-scm.com/downloads`.

[27] "GitHub, GitHub, Inc." Accessed: Apr. 16, 2019. [Online]. Available: `https://github.com/`.

[28] R. Gonzalez, P. House, I. Levkivskyi, L. Roach, and G. van Rossum, "PEP 526 – Syntax for Variable Annotations." Accessed: Apr. 28, 2019. [Online]. Available: `https://www.python.org/dev/peps/pep-0526/`.

[29] S. Mangan and U. Alon, "Structure and function of the feed-forward loop network motif," *Proceedings of the National Academy of Sciences*, vol. 100, no. 21, pp. 11980–11985, 2003.

[30] V. Singh, P. K. Dhar, and editors, *"Modelling Methodologies for Systems Biology" in Systems and Synthetic Biology*. Springer, Dordrecht, 2015.

[31] J. E. Bowyer, E. LC. de los Santos, K. M. Styles, A. Fullwood, C. Corre, and D. G. Bates, "Modeling the architecture of the regulatory system controlling methylenomycin production in streptomyces coelicolor," *Journal of Biological Engineering*, vol. 11, p. 30, Oct 2017.

[32] D. T. Gillespie, "Exact stochastic simulation of coupled chemical reactions," *The Journal of Physical Chemistry*, vol. 81, no. 25, pp. 2340–2361, 1977.

[33] N. Monk, "University of Nottingham. Introduction to Biological Modelling. [PDF]." Available: `https://www.maths.nottingham.ac.uk/minisites/mathsforlife/cpib2012/lect3p2_stochastic_handout.pdf` accessed 29.04.2019.

[34] “Prof Nathan Griffiths. University of Warwick. (2017). CS255 Artificial Intelligence: Informed Search and Local Search [Lecture Slides].”

[35] I. Kutlar, “CS310 - Progress Report.” Nov. 27, 2018.

[36] “Repressilator model at Biomodels Database.” www.ebi.ac.uk/biomodels-main/. `http://www.ebi.ac.uk/biomodels-main/BIOMD0000000012` (accessed Apr. 29, 2019).

[37] “Elowitz.R.” Andrey Akinshin, Accessed: Apr. 29, 2019. [Online]. Available: `https://gist.github.com/AndreyAkinshin/37f3e68a1576f9ea1e5c01f2fd64fe5e`.