Comparison stochastic vs deterministic modeling of *S. pneumoniae*, using the Gillespie Algorithm

Tycho Marinus

25-06-2015

Contents

1 Introduction

Computers have been used for modeling and simulation from their early days. The connection between biology and informatics started mainly with the use of computer databases, storing protein sequences, allowing for quick searches on similarities. However, the connection between the two has grown over the years. The use of informatics for biology has expanded from just the use of databases, towards large scale simulations. From large scale animal behaviour models to models and simulations of protein interaction inside bacteria. This thesis is deals with system biology, a field that uses computational and mathematical modelling to understand biological systems. With these models and simulations, it becomes possible to make predictions about certain biological systems. Most *in vivo* and *in vitro* experiments are both expensive and time consuming. Thus being able to first experiment inside a simulation can reduce the amount of expensive experiments.

Streptococcus pneumoniae or formerly known as *Diplococcus pneumoniae* is a wide spread bacteria. The name Pneumonia is associated with lung infection, but *S. pneumoniae*can also cause a variety of different infections, spanning from light ear- (otitis) and nose- (rhinitis) to brain- and spinal cord infections (meningitis)[1]. Even though it is possible to prevent and treat *S. pneumoniae*efficiently, it still is the cause of death for many children and elderly. Especially in Asia and Africa, where it is still the main cause of child death[10]. An estimate from The World Health Organization (WHO) indicated that around 1.6 million people die of *S. pneumoniae*each year, of which around 50% were under the age of 5 [1].

*S. pneumoniae*was first discovered in 1881 and has been studied intensively [12]. Currently there is a vaccine and in case of infection there is an antibiotics treatment. The antibiotics used to treat *S. pneumoniae*have a high chance of success. However, the high mutation rate of *S. pneumoniae*, certain types of vaccines and antibiotics are becoming less efficient[3]. This high mutation rate in *S. pneumoniae*is caused by a property called competence. Competence highly increases the mutation rate of a cell under certain conditions. This is done by taking in external bits of DNA. For *S. pneumoniae*, competence is activated by stress, for example caused by antibiotics. Therefore each time a treatment is started with antibiotics the chance of *S. pneumoniae*mutating against that specific antibiotic is increased. This can lead to a depletion of antibiotics to use. One possible remedy for this would be to stop the competence activation, lowering the mutation rate of *S. pneumoniae*. For this reason it is important to understand the exact workings of competence in *S. pneumoniae*.

With experiments and models it was discovered that *S. pneumoniae*is not competent all the time. When the bacteria becomes competent, it will reach a peak in roughly 8 to 9 minutes, followed by a decline in 4.5 minutes. After an invocation, the *S. pneumoniae*competence is unable to be activated for around 80 to 90 minutes [6]. The deactivation of competence is called competence shutdown. Models about competence in *S. pneumoniae*are already available[4]. However, new research has identified new factors that have influence on the competence, such as the protein Dpra [6].

With these new discoveries an updated model might give a more accurate simulation of competence. The updated model was created by Stefany Moreno. Like the previously mentioned model [4], the updated model also uses a system

ordinary differential equations to calculate the model. These ODEs are the most used method for solving models of massaction kinetics. Massaction kinetics are models that contain a direct correlation between a speed or rate of a chemical reaction and the number of proteins. Simply put, a higher reaction rate gives a larger number of proteins for that reaction. In this thesis we want to look at an alternative method of solving these massaction kinetics models. An event-driven, stochastic solution has been made that we compare to the deterministic solution.

The motivation for this alternative method is that both the old model from Karlsson [4] and the new updated model by Moreno have been solved deterministically with differential equations. This might give problems in how accurate these results are in comparison to the real world at low molecular counts. Therefore this thesis is about comparing Morenoś deterministic solution to an alternative event driven, stochastic, solution. Furthermore the aim of this work is to see what the cost of creating a stochastic solution might be and finally see if it might be possible to automate this conversion process from deterministic to stochastic.

2 Background

2.1 Model of competence in *S. pneumoniae*

Some cells have the ability to incorporate external DNA. This property is called competence. Competence can be artificially induced while other cells can become competent without artificial interaction. The latter group of cells are called naturalcompetent. *S. pneumoniae*is a bacteria falls within the latter group. When a cell dies or undergoes lysis its DNA will start to disintegrate and float freely in the environment. Cells with competence are able to take parts of these "free" strands and take them in. There are multiple uses for this ability. A cell is able to use the DNA as food, use it to repair its own DNA or use it to improve his own genetic diversity. A cell is able to improve its diversity by combining the free bits of DNA and integrating it into its own genome. This process will speed up mutations and thus the evolution of the cell.

2.2 Biological model *S. pneumoniae*competence

Before a mathematical model can be created, it is important to understand the biological model. The created biological model for competence in *S. pneumoniae*contains 23 different proteins and peptides. The main activator of the competence process is CSP (Competence Stimulating Peptide). CSP can be sensed from an external source or created and excreted by the cell itself. CSP sensing will also increase the CSP creation and excretion of the cell itself, triggering neighbouring cells to also become competent. In order to gain better understanding of this cycle, we have to look at the proteins involved in this process.

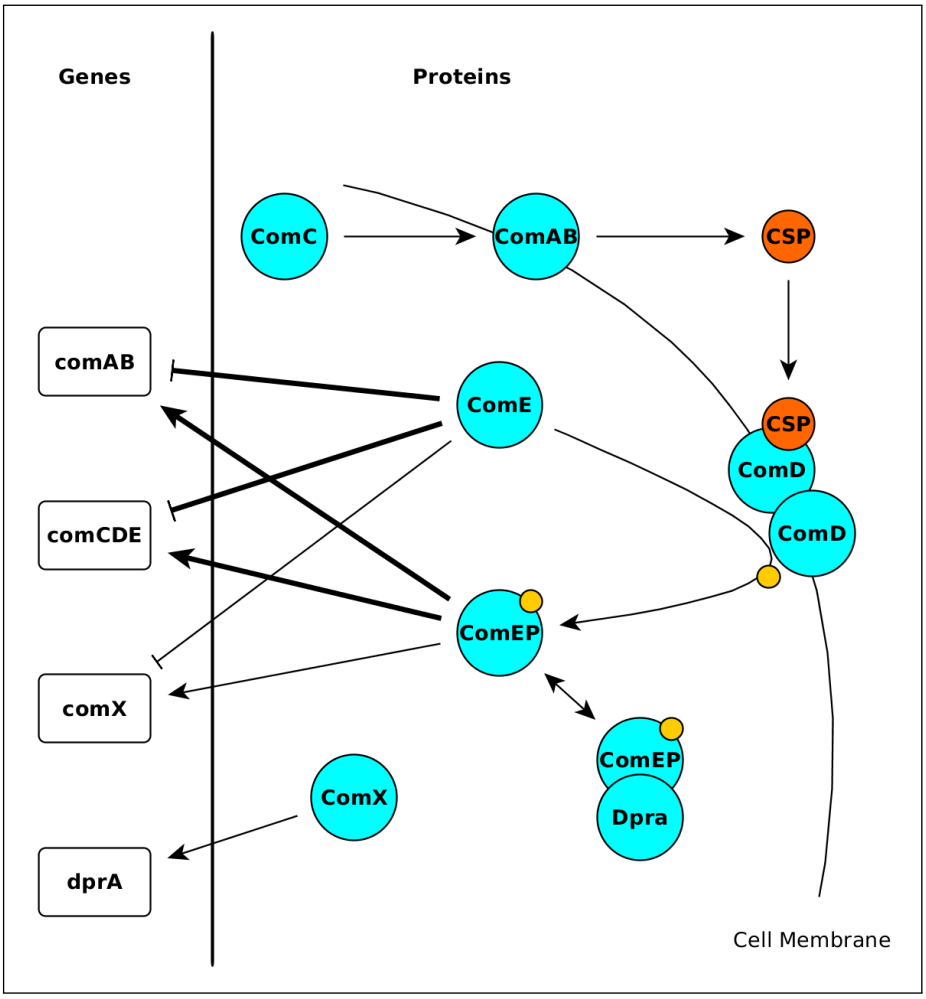


Figure 1: Diagram showing relationships between proteins and genes. Thick lines indicate a higher binding probability.

The receptor of CSP is ComD which is a protein dimer that sits inside the membrane of the cell. This allows external CSP to bind to ComD triggering phosphorylation of the internal protein ComE. Phosphorylation is the process of adding a phosphate group to a molecule, in this case a protein. Phosphorylated ComE (ComE˜P) will promote the production of ComAB, ComC,ComD,ComE and ComX proteins, trough transcription of the comAB comCDE and comX genes.[1] This gene group is part of the genes called early com (competence) genes. If ComE does not get phosphorylated, it can bind to the early com genes, blocking the transcription. So depending whether or not ComE is phosphorylated or not, it will either promote or suppress production of ComAB, ComC,D,E and ComX. It does have a stronger connection for phosphorylation, however. Thus if CSP is activating phosphorylation of ComE by ComD, then that is stronger than the suppression mechanism. ComAB and ComC,D,E are

[1]Note that captitol Com is used to indicate proteins, while lower case com indicates genes.

proteins that activate the start of the competence mechanism. ComAB transports CSP, produced in the cell, outside the cell membrane, promoting competence in neighbouring cells. comCDE, transcription activated by ComE˜P, will produce ComC, ComD and ComE. ComC is an early state of CSP. While ComD and ComE have been explained. This leads us to conclude that if there is a external source of CSP a cell will also start producing CSP, going full circle. However there are also side proteins and mechanisms that get activated by the CSP circle. The late com protein ComX is one of these proteins, the comX gene is activated by ComE˜P. ComE˜P will also activate transcription of the comW gene, producing ComW [8]. ComW prevents proteolysis of the ComX, stabilizing and thus prolonging the presence of ComX. The protein ComX is the main protagonist in controlling transcription of the late competence genes. Activating the transcription of genes for uptake and processing of external DNA.

2.3 Mathematical Biological model

From this biological model Moreno created the mathematical model for the deterministic solution. This model was crafted by converting known reactions into formulas. As an example we look at the amount of ComC in the model. We know that the amount of ComC is based on either the suppression by ComE, or the transcription activation ComE˜P. We also know that ComC is used in the production of CSP, and that there is a certain amount of degradation of the protein

$$\frac{dComC}{dt} = \underbrace{\frac{\sigma C}{\delta_R^C} \cdot (\beta_C^0 \cdot (1 - Y_{EP}^C) + \beta_C \cdot Y_{EP}^C)}_{\text{Production ComC}} - \underbrace{\frac{e \cdot ComAB \cdot ComC}{ComC + k_e}}_{ComC\ used\ for\ CSP} - \underbrace{\delta_C \cdot ComC}_{degradation\ ComC}$$

This shows all the reactions that have influence on the the protein ComC. Each term is a reaction in the biological model, either increasing or decreasing the amount of ComC in the simulation. For example if we look at $-\delta_C ComC$, then this indicates the degradation of ComC. Delta C (δ_C) indicates the speed of the degradation over time. To calculate how many ComC proteins would degradate, we multiply the amount of ComC by its degradation rate δ_C. $\frac{\sigma C}{\delta_R^C} \cdot (\beta_C^0 \cdot (1 - Y_{EP}^C) + \beta_C \cdot Y_{EP}^C)$ shows the production of ComC based on the amount of ComE˜P (Y_{EP}^C). And lastly we see $-\frac{e \cdot ComAB \cdot ComC}{ComC + k_e}$ indicating the amount of ComC that is used for the production of CSP.

In a similar way the other process parts of the competence system have been converted into mathematical equations (by Moreno). In appendix B on page 31 a full list of all the equations is given.

3 Models and Simulations

As indicated earlier, simulation of biological processes can save time and cost of in vivo and in vitro experiments. Most processes can be modelled and simulated if there is enough information about the system. One problem with this is that if one important or even a few non important variables are missing in the model, then the results of the simulation can stray away from the real world. However, this does not per se mean that the results are useless.

For example, Mirouze [6] discovered that the protein Dpra has a large influence in the competence shutdown of the *S. pneumoniae*. The 2007 model from Karlsson 2007 [4], does not contain this protein. The addition of Dpra in combination with other newly discovered proteins is the main reason for creating a new model.

Choosing simulation method

Once a (biological) model is created there are a different methods to produce results and predictions based on the model. These methods can be separated in two categories, stochastic and deterministic. Both are used for algorithms based on a time component.

3.1 Deterministic methods

Deterministic methods are all methods that will produce the same result if the same input is used. An easy example of a deterministic function would be

$$\frac{dComD}{dt} = \gamma_{ComD} ComD - \delta_{ComD} ComD$$

In biology systems the most common method used is the use of ordinary differential equations (ODE), or a system of ODEs [11]. This is mostly because creating and solving ODEs is both relatively easy and fast. This is also helped by the fact that there are a lot of different tools available that help with solving and creating the ODEs. ODEs are basically formulas that, once solved, will result in the data for the model. ODEs are not the only deterministic method that exists. The fact that deterministic methods always produce the same result has some benefits. An example is that if one tries out new parameter values only a single execution is required to get to a result. The downside is however that real world experiments will often not result in fully identical results. Mostly because of fundamental randomness at quantum scales. Very slight differences in temperatures or quantities might alter the result. Often these differences are minimal and not a problem, but the variations that a stochastic method provides can give better insight in the system.

There are mixed methods such as the stochastic differential equation (SDE) that adds a stochastic component into a ODE. Creating a mixture of the two. But theoretically these are just stochastic methods, often just variations of deterministic methods to provide a stochastic component.

3.2 Event driven methods

While deterministic methods will always produce the same output, event driven methods will not. Event driven methods are therefore called stochastic. They contain randomness, resulting in different outcomes each run. For a function (or algorithm) to be stochastic there needs to be a random factor in the equation, for a basic example

$$\frac{dComD}{dt} = \gamma_{ComD}ComD - \delta_{ComD}ComD + \sigma$$

Where $\sigma \in U([0,1])$, so σ is some value between 0 and 1. Each time this function is calculated, the added sigma will add randomness to the result. Therefore the output of the function will may vary each time.

Creating stochastic algorithms is often more complex than their deterministic counterparts. Stochastic algorithms have to be written for each model. We could not find general stochastic tools that makes it easy to create the algorithm through an easy to use interface. Because of this additional complexity stochastic methods are less used in system biology. There are, however, a lot of code examples available. One example of an event driven method is the Gillespie algorithm. Simply put, it is an algorithm that uses weighted chance to decide what event happens based on the previous state of the model. This weighted chance is based on the number of certain molecules in the model, in combination with a probability of that event happening. Once an event is selected the model is updated and a new event is randomly chosen. Each update of the model will add or subtract a certain (integer) number from some substances of the model. Resulting in a new state, with slightly adjusted chances for the events. From this new state the entire process can start again to calculate the next state. These steps are repeated until an equilibrium is found or until a simulation time has been exceeded.

3.3 Event driven vs Stochastic

Even though deterministic methods are considered to be easier to use and implement, they do have some problems. When thinking about modelling as replicating a real world system, it becomes logical to include some degree of randomness in the model. The reason behind this is that it is very hard to completely cover every aspect of a system. There are just too many parameters to take into account, which makes it highly likely that some parameter will be missed. This gives all the more reason for the use of a stochastic method. Since the added random factor might be able to cover the missing parameters, or changes and inaccuracies in parameters. However, there is a cost to pay for using stochastic methods. Both the time to create and to run such a model takes a lot longer. If we look at a stochastic function based on time:

$$f(x,t) = f(x,t-1) + \sigma$$

Where $f(x,0) = 0$ and t is the time ($t \in \mathbb{R}$), we see that to calculate any $f(x,t)$, we will need to know the previous state of the function. Additionally for every iteration we increase the time by a unit, a new random value for σ has to be created. Generating a few random numbers will not cause a problem. But since this function will not result in a contiguous graph, it might prove necessary to use a very small time step, to create a smooth curvature. This will result in a large number of random numbers that need to be generated. While the most common method for deterministic methods, the ODE, can be solved by different techniques, such as the Runge-Kutta method, that takes significantly less time. Therefore it is a choice between possible inaccuracies and calculation time.

3.4 Ordinary differential equation vs Gillespie algorithm

If we take a closer look into the deterministic ODE and stochastic Gillespie algorithm we can find some additional important differences.

When using ODEs in modelling, a series of ODEs representing the model need to be created. These combinations of ODEs are called system of ODEs. As explained ODEs will always produce the same result, which is not realistic, in comparison how the real system works. Besides this there is another point that makes ODEs less realistic. Namely, they are calculated by solving the differential equation, resulting in a pure smooth series. The values from this can be any float positive or negative. This gives two problems. When modelling for example a specific process in a cell, the quantities should be counted as integer (natural numbers) because logically there cannot be a half or a negative number of proteins (or any other real world substance). Most ODE solvers are able to remove the possibility of negativity. However they still work with float numbers. This can have strange effect on the models results. For example if at some point the quantity of some substance is around 0.2 then an equation influenced by this value will still just calculate a result, assuming that this is correct. This can, however, become a problem. For instance, there is quite a large difference between 0.2 and the rounded value (0), if you keep calculating with it. This is less of a concern when the values stay higher. Logically the larger the amounts the less of a concern this should be. It is still something that should be looked into before the results can be taken seriously.

The Gillespie algorithms does not have this problem. It starts with a state in which every substance has an integer value. Each step a reaction is chosen. This reaction then adds or subtracts a (integer) value from some substances. This way the state of the model will always hold integer values. This is more realistic, but logically more computational heavy.

For these reasons it might be important to analyse models both deterministic and stochastic. When none of these problems occur, the easy route of the simple deterministic ODE can be chosen.

4 Implementation of the models

4.1 Deterministic model

The first model that we tried to reproduce was of the 2007 model by Karlsson[4]. However, this happened to be problematic since the paper did not contain all the necessary information for recreation of the model mainly because some variables values that were missing in combination with vagueness about the units of the values.

As stated before, Stefany Moreno was able to create a similar biological model. From which she created both versions of the deterministic models (without Dpra and with Dpra). This was done by creating a system of non-linear ordinary differential equations, with an extended version which included Dpra.

All the models are systems of ODE, which can be solved by ODE solvers. The code is written in python and uses the LSODA [5] solver. LSODA is a "Ordinary Differential Equation Solver for Stiff or Non-Stiff System" which is able to switch dynamically between a stiff and nonstiff method depending on the data. LSODA is part of the ODEPACK contained in scipy, a "Python-based

ecosystem of open-source software for mathematics, science, and engineering" [9].

4.2 Event driven

We chose to use the Gillespie algorithm for the event driven method. The reason for this is that it is a relatively simple algorithm that has a clear structure. This algorithm also has a very exact simulation, in which each step will always hold a valid "natural" state.

4.3 General Gillespie algorithm

The Gillespie algorithm has been shortly explained in the previous section. Here we will take a closer look at the algorithm. As explained above the basic steps for the algorithm are [2]:

```
Initialize: Set number of molecules in model state 0, set reaction constants
while t ≠ endTime do
    Generate random number, to choose next reaction. This will also set
    time interval step.
    Update the model with the selected reaction, increase the time t by the
    interval value.
end while
return Finalstate
```

The algorithm starts with initializing all the molecules at the correct value for state 0. The reaction constants together with the number of molecules for a certain reaction define the reaction hazard. As a final step of the initialization a random generator is chosen, which is used to throw the weighted dice for selecting a reaction.

Calculating the reaction hazard is quite straightforward. First all the reactions are calculated. Reactions are often in the form of a molecule count multiplied by a rate constant. However others are more simplistic, for example consisting of only a single rate constant. More complex reactions also exist, with multiple molecules and rate constants having influence on the final rate. For every modeled reaction the rate is calculated. These rates represent the chance that a specific reaction takes place. Reactions that interact with a molecule that is present in large quantities will have a higher chance of being chosen than reactions with a similar constant value, but with a smaller quantity. All these reactions are stored separately and also added together to get a rate sum, the combined value of all the reactions. To choose the reaction, a random number between zero and one is created which is then multiplied with the rate sum. To get the chosen reaction, the separate reaction rates are added together and compared. If at one point the combined value exceeds the random value, then the reaction of that rate will be executed.

Executing a reaction is done by looking at the reaction and adding or subtracting a certain number from a molecule quantity, dependent on what is consumed and produced by that reaction. After quantities of the molecules are adjusted the whole process will start again.

4.4 Converting ODE equations to stochastic equations

In order to implement the Gillespie algorithm it is required to first convert the ODE equations into a usable format for the stochastic solution. This is a straightforward process. If we expand our example from earlier

$$\frac{dComD}{dt} = \gamma_{ComD} ComD - \delta_{ComD} ComD$$

This simple example shows that the amount of ComD is dependent on its synthesis ($\gamma_{ComD} ComD$) and the degradation ($\delta_{ComD} ComD$). Here gamma and delta represent synthesis rate and degradation rate, respectively. ComD indicates the current amount of this protein in the model. Each element of this equations is equal to a specific reaction that can take place. However, for the Gillespie algorithm we need to separate the equation into single reactions, given by

$$\begin{aligned} Synthesis &= \gamma_{ComD} ComD \\ Degradation &= -\delta_{ComD} ComD \end{aligned}$$

Converting this simple example will give two reactions

$$\begin{aligned} \phi &\xrightarrow{\gamma_{ComD}} ComD \\ ComD &\xrightarrow{\delta_D} \phi \end{aligned}$$

By separating the reactions into single reactions it becomes possible to calculate the reaction hazard.

When we look at a more complicated example we can also see specific interaction between two differential equations when we convert them. We take the example of the production of CSP. As explained before CSP is created by ComAB by consumption of ComC. The differential equations showing this are:

$$\frac{dComC}{dt} = \frac{\sigma C}{\delta_R^C} \cdot (\beta_C^0 \cdot (1 - Y_{EP}^C) + \beta_C \cdot Y_{EP}^C) - \frac{e \cdot ComAB \cdot ComC}{ComC + k_e} - \delta_C \cdot ComC$$

$$\frac{CSP}{dt} = \frac{e \cdot ComAB \cdot ComC}{ComC + k_e} - \delta_{CSP} \cdot CSP$$

For ComC we can quickly see similarities with our previous example for synthesis and degradation. Degradation of CSP is also straightforward. Giving us these reactions:

$$\begin{aligned} \phi &\xrightarrow{\lambda_{ComC}} ComC \\ ComC &\xrightarrow{\delta_C} \phi \\ CSP &\xrightarrow{\delta_{CSP}} \phi \end{aligned}$$

For CSP synthesis we notice that the term $\frac{e \cdot ComAB \cdot ComC}{ComC + k_e}$ appears in both equations as an addition and a subtraction. This shows that ComC is used to produce CSP. In essence this is just one reaction and thus we want to convert it into one reaction. This will give us the reaction:

$$ComAB + ComC \xrightarrow{\overline{ComC+k_e}^{e}} ComAB + CSP$$

This shows us that one unit of ComC is consumed for the synthesis of CSP. Since ComAB appears on both sides of the reaction, it indicates that ComAB is only used as a enzyme and is not consumed.

Once all the ODE equations are translated with this method, then they can be used in a Gillespie algorithm. For a full list of all the differential equations see page 31 and for the converted reactions see page 27.

4.5 Implementation Gillespie algorithm

In a first attempt to create the event driven solver, Matlab was chosen to implement the Gillespie algorithm. The implementation worked correctly, however, there where some problems with the calculation speed. The average time for the Matlab implementation was around 7.4 seconds with an simulation time of 100.000 seconds. When implemented in C++ the running time was cut down to only 2.6 seconds. Since this first initial implementation did not contain the proteins related to Dpra, the difference would only become larger with more proteins and thus more variables. Therefore the entire code was converted into C++ and the later extensions where also implemented in C++.
The final implementation scales linear with the simulation time. However this is only the case when the protein numbers stay equal, there are more variables that have an influence on the execution time. The main influence for the execution time is the total chance of a reaction taking place. When there is a low number of protein molecules in the model the time it takes for a reaction to take place increases. If this is the case the time increases more quickly and therefore the amount of total reactions over the entire simulation decreases. This decrease in amount of reactions leads to a direct increase in performance, lowering the execution time significantly[2].

4.5.1 C++ implementation

There are a few required input parameters for the C++ program, namely: rates file, result file name, step size, end time and number of cycles.

The full code can be found in appendix D page 33. Here the important parts of the algorithm are explained. The C++ code is separated into two parts. First the file, *Gillespie*.cc, handles initialization of the program and the general algorithm as described earlier. The second file, *reactions.cc* handles the exact state of the simulation. It also handles the calculating of hazards and adjusting the model to update to a new state. Since the reaction rates are not hard coded into the program the first steps are to parse the rates.txt file. This is done in a straight naive parsing method. Then the program is ready for its core algorithm.

[2] All timing tests done on an Intel Core i5-4670K CPU at 4Ghz, timed on a single thread

Listing 1: Gillespie

```
void Gillespie::run(double timeEnd, int cycles)
{
    double currentTime, measureTime;
    int idx;
    for(idx = 0; idx < cycles; ++idx)
    {
        currentTime = 0.0;
        measureTime = TIMESTEP;
        reactions->resetCurStat();
        while(currentTime < timeEnd)
        {
            reactions->calcHazard();
            currentTime += this->getNextTime();
            reactions->chooseReaction(currentTime);
            if(currentTime > measureTime)
            {
                reactions->measureState(measureTime);
                measureTime += TIMESTEP;
            }
        }
    }
    reactions->calcAverage();
    reactions->printStatus();
    reactions->writeResult();
}
```

The inner loop here clearly shows the translation from pseudocode (page 10) to C++. First the reactions hazards are calculated. This is simply done by the formulas given in appendix A, see page 27. Followed by the calculation of the sum of these hazards.

When the reaction hazard sum is known it becomes possible to calculate the time until a new reaction will occur.

Listing 2: Find the new time step.

```
/* Generate next time interval */
double Gillespie::getNextTime()
{
    double randVal;
    do
    {
        randVal = ((double) rand() / (double)(RAND_MAX));
    } while ( randVal == 0);
    return ((1.0/reactions->getSum())*(log(1.0/randVal)));
}
```

First a random value $0 < rand <= 1$ is chosen. Then with this a random time is calculated, which is used as indicator for how long it takes before a new reaction happens. This time is then added to the current time, indicating the time between the next and the previous reaction.

The following step is to find the next reaction that takes place.

Listing 3: Select reaction.

```
void Reactions::chooseReaction(double currentTime)
{
  double cumSumArr[NUMREACTIONS];
  double randVal;
  int idx, reacNum;
  int diffComX;

    randVal = ((double) rand() / (double)(RAND_MAX));
    randVal = randVal * rateSum;

  cumSumArr[0] = rateValues[0];
  if(cumSumArr[0] <= randVal)
  {
    for(idx = 1; idx < NUMREACTIONS; ++idx)
    {
      cumSumArr[idx] = rateValues[idx] + cumSumArr[idx-1];
      if(cumSumArr[idx] > randVal)
      {
        reacNum = idx;
        break;
      }
    }
  }
  else
  {
    reacNum = 0;
  }
  diffComX = curStat[9];
  for(idx = 0; idx < NUMPROTEINS; ++idx)
  {
    curStat[idx] += reactionsArray[reacNum][idx];
  }
  if( flagComX == false && diffComX < 150 && curStat[9] >= 150)
  {
    cout << currentTime << endl;
    flagComX = true;
  }
}
```

Also here a random value is taken to, this random value is $0 <= rand <= \sum reactionhazards$. The cumulative sum of the reaction hazards are taken until the random time is smaller than the sum. This is then followed by adjusting the state with that given reaction. The reactions are all stored in a large 2D array, indicating which proteins get adjusted. See appendix on page 27.

These are the essential parts for the Gillespie algorithm. However, a few things have been added in the program. There is a timer that will store current states of the model, otherwise only the result would be visible. An extra loop around the Gillespie algorithm makes it possible to take multiple runs with the same starting condition, followed by taking the average of all the runs.

The final result is written to the output file in a comma separated file.

5 Results

The main goal for the event driven implementation, was to find possible differences against the deterministic method. This was first done by testing the same experiments done by Moreno. After which new test runs where done to find possible differences.

5.1 Competence activation comparison

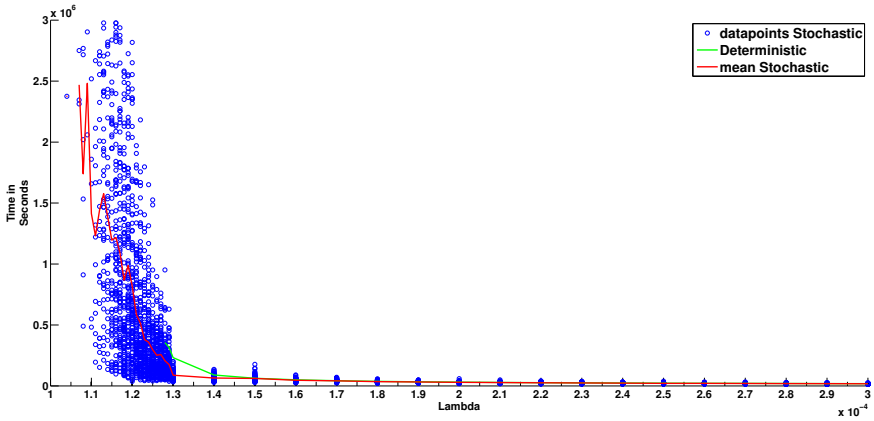
The first comparison is based on tests on competence activation. These tests are carried out without the presence of the Dpra protein, since it suppresses competence activation by binding to the ComEP protein. We started with a comparison between the bistable switch point by adjusting the phosphorylation rate of ComE by ComDPdim (λ)[3]. Both simulations are considered to have active competence once the amount of the ComX protein becomes higher than 150. ComX is chosen for this because it is the activator of the genes for competence activation. The Gillespie implementation appears to have an earlier competence activation. This is especially apparent when a lower value for lambda is used. The range of the data starts at 1.0e-04 lambda to 3.0e-04 lambda with an step size of 1.0e-05. Between 1.0e-04 and 1.3e-04 extra samples were taken with step size 1.0e-06.

When lambda is lowered to 0.000110034 the stochastic simulation still has a small change of activating competence. This however only happens when the simulation time is extremely high, only activating after roughly 1.728e+6 seconds (20 days) see figure 3. Table 1 shows the activation percentages of the stochastic simulation after one hundred runs. For all lambda values above 0.000121 and with CSP induction of 10.000 the stochastic simulation gives a 100% activation.

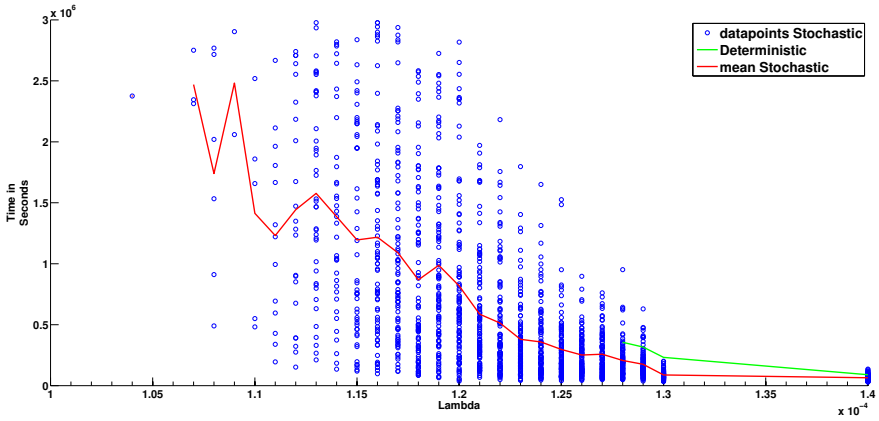
Lambda	No CSP	1000 CSP induction
0.000100	0	0
0.000101	0	0
0.000102	0	0
0.000103	0	1
0.000104	1	0
0.000105	0	0
0.000106	0	2
0.000107	3	3
0.000108	6	6
0.000109	2	8
0.000110	5	8
0.000111	13	23
0.000112	22	27
0.000113	39	27
0.000114	35	47
0.000115	47	67
0.000116	69	72
0.000117	81	80
0.000118	87	91
0.000119	96	92
0.000120	100	99
0.000121	99	100

Table 1: Table showing percentage of activation.

[3]For a full list of variable values used, see appendix C on page 33



(a) Lambda step size 1.0e-05



(b) Lambda step size 5.0e-06

Figure 2: Plots showing both stochastic and deterministic activation times. There are a hundred data points for each lambda value, average is taken from these points.

When slowly increasing lambda, starting at 1.0e-04, the deterministic simulation also activates competence. However the time gap between these two can be as high as 150.000 seconds, 41 hours. While increasing lambda this gap becomes smaller, until both simulations activate competence roughly at the same time. For high values of lambda, larger than 4.0e-04, the activation is very similar, with the stochastic simulation showing activation slightly earlier. This difference keeps getting smaller the higher lambda is chosen. Some stochastic runs have a slower activation than the deterministic simulation, but the average time is always lower, figure 2.

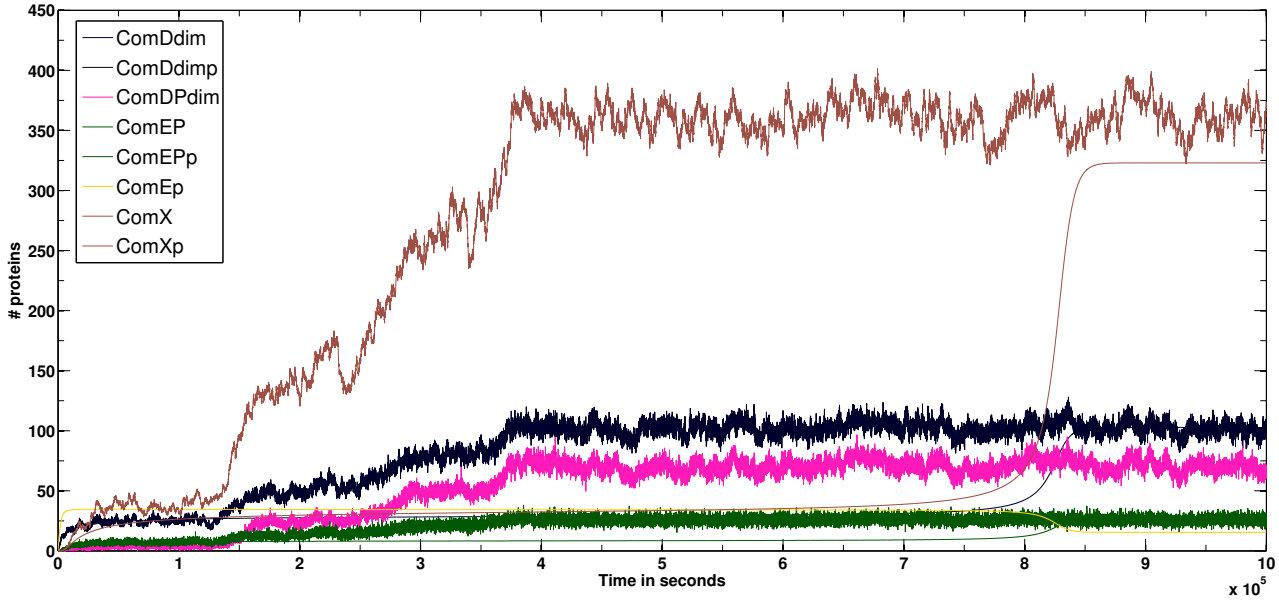
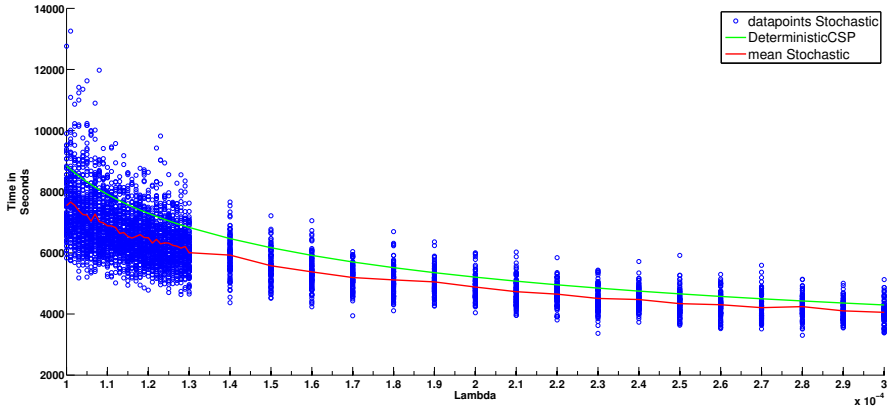


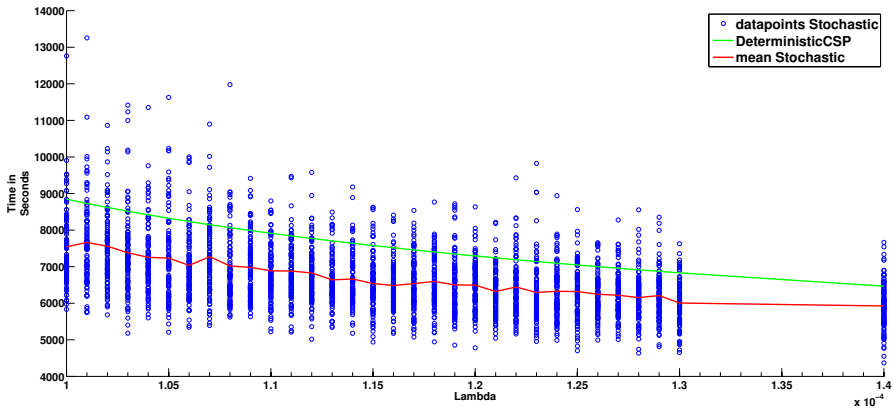
Figure 3: Jagged lines are from the stochastic simulation.
Smooth lines are from deterministic simulation.
$\lambda = 0.00012793387$

5.2 Competence activation comparison with CSP induction

CSP induction was simulated by adjusting the starting amount of CSP in both simulations. Two starting values for CSP induction have been tested, 1.000 and 10.000.



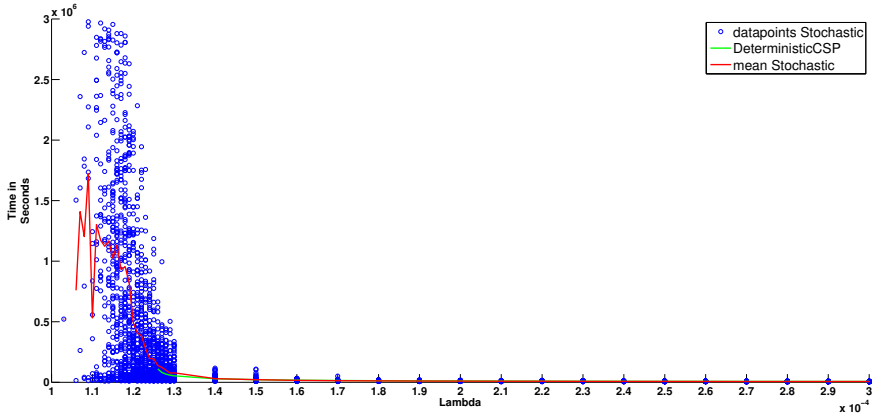
(a) Lambda step size 1.0e-05



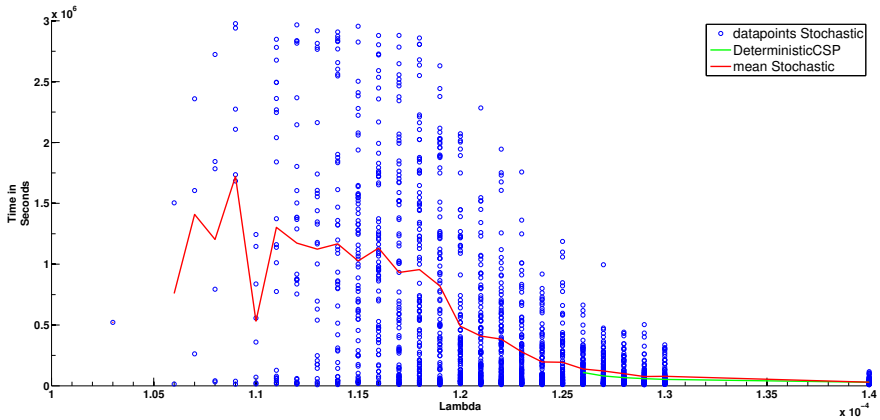
(b) Lambda step size 5.0e-06

Figure 4: Plots showing both stochastic and deterministic activation times with 10.000 CSP induction. There are a hundred data points for each lambda value, average is taken from these points.

The differences between the stochastic and deterministic results are smaller than when there is no CSP induction, figure 4a and 4b. However with CSP induction it follows the same trend as without. When the lambda value is large the difference is smaller, and when lowering the lambda value the difference increases. With very small lambda values the difference is not as great compared to no CSP induction. However this can be explained by the minimal lambda value of 1.0e-04 which was tested.



(a) Lambda step size 1.0e-05



(b) Lambda step size 5.0e-06

Figure 5: Plots showing both stochastic and deterministic activation times with 1.000 CSP induction. There are a hundred data points for each lambda value, average is taken from these points.

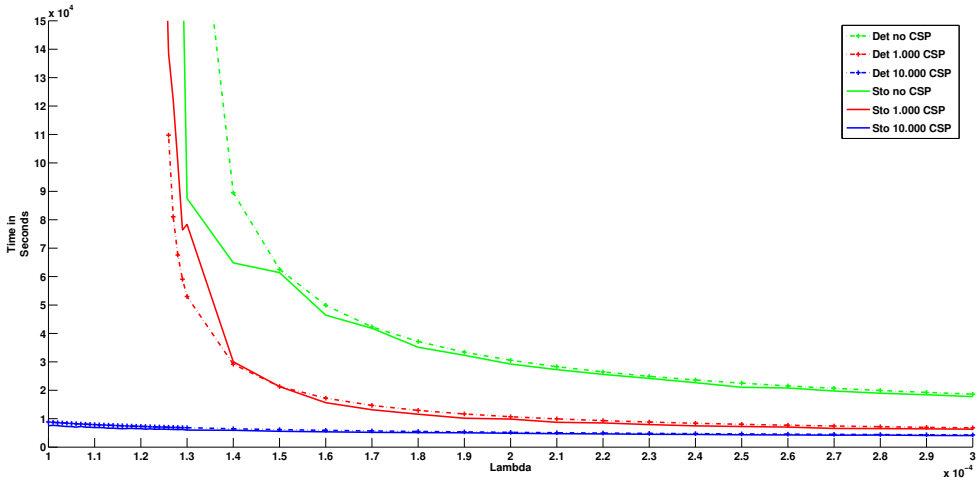


Figure 6: For clarity, plot showing both stochastic and deterministic mean activation times comparison for the different CSP inductions. Step size 1.0e-05.

With only 1.000 CSP induction there is a smaller difference. When adding smaller amounts of CSP the competence activation of the deterministic simulation is even lower, at some points, than the stochastic simulation. Besides this the difference overall for activation is here the smallest, as seen in figure 5.

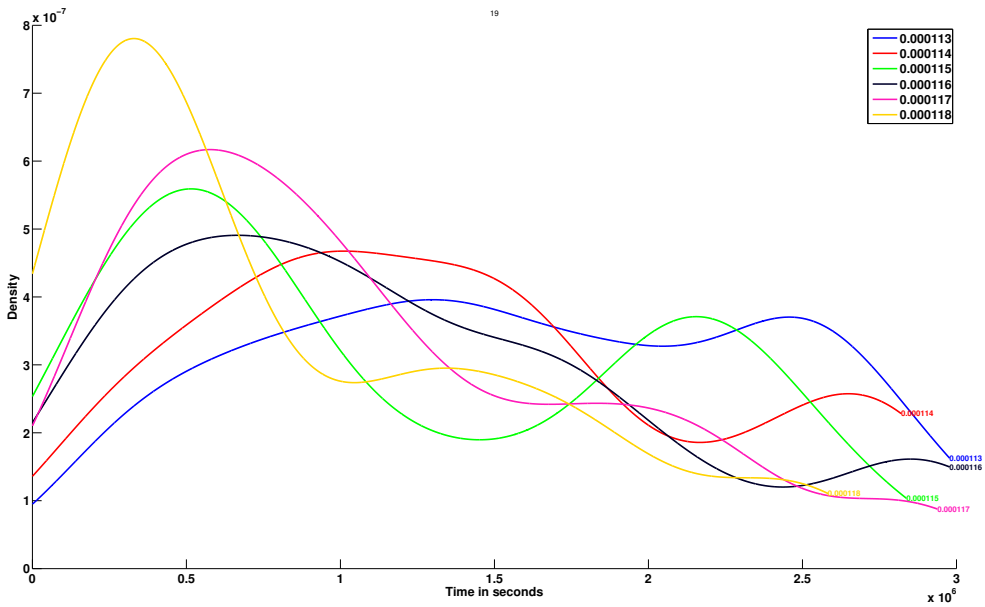


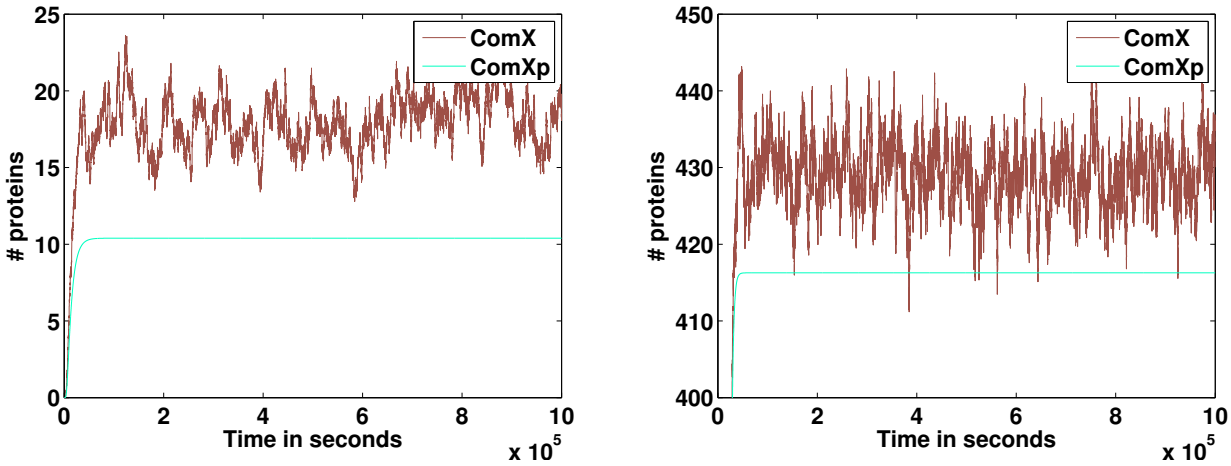
Figure 7: Kernel density plot.

Using kernel density estimates on the stochastic data we found some indica-

tions of bimodality. When plotting the density estimates at lambda values from 1.13e-04 to 1.18e-04 a growing bimodality forms from 1.13e04 reaching a clear double peak at 1.15e-04, which then declines and smooths out again.

5.3 Individual proteins comparison

All but one protein show no noticeable difference after stabilization. While competence activation is starting, a difference can be seen, however this is expected because of the differences in activation times. Once a stable state is reached all the proteins have a similar value. The exception is ComX, as it has a slightly higher value in the stochastic solution, compared to the deterministic solution.



(a) No competence activation

(b) With competence activation

Figure 8: ComX stochastic vs deterministic, "ComXp" is from the deterministic simulation

When competence does not activate the mean difference after stabilization for ComX is around 7.5, with the stochastic standard deviation of 1.7113 and a standard error of mean 0.0104, see figure 8a.

The difference is slightly higher when stabilizing with active competence. The difference is then around 12.8, with the stochastic SD of 4.3207 and SEM 0.0262, see figure 8b.

6 Discussion

First, we found three properties of the stochastic simulation that are noteworthy from the results.

- The stochastic simulation activates earlier compared to the deterministic simulation.
- The bimodality of the activation times in around 1.15e-04 lambda values
- A difference in the value of ComX was found between the two simulations.

The difference in activation time between the two simulations could simply be caused by the stochastic property of the Gillespie Algorithm. When simulating deterministically, the simulated cell behaves exactly the same way each run, acting as if each cell would always behave exactly the same. With the stochastic simulation each run represents a single cell, but now with the possibility of anomalies. This entails that the stochastic simulated cell has the possibility to randomly create a bit more CSP, which in turn increases the chance of earlier competence activation. Over time the chance of this added CSP creation grows over time. This could lead to an increased possibility of early activation, which is the case for most simulations run, see figure 7. This is a possible explanation for the earlier activation times for the stochastic model.

The bimodality found in the activation times around roughly 1.15e-04, figure 7, is likely because of the bistable switch also found in Karlsson 2007 model [4]. This was also found in the current deterministic model by Moreno [7].

Lastly we take a look at the difference in ComX value. With the current information we cannot explain this difference. The difference is only found for ComX, both models have been investigated for possible errors and differences in reaction values, no differences where found. The possibility that it is caused by the differences between the ODE and Gillespie Algorithm within small values, can be crossed out because this difference also occurs when ComX has a relatively high value. And protein ComEPdim is consistent while having lower values. Therefore, the differences in ComX values cannot be caused by the possible inaccuracies of working with low quantities in a ODE.

7 Future work

The main focus of this thesis was the comparison of deterministic versus stochastic simulations. This has been done for this specific biological model. However the code created is quite static and aimed at this specific model. The program could highly benefit from being refactored into a more dynamic program.

For example, adding a new protein and corresponding reactions requires multiple files to be edited now. The reason for this is that even though the proteins and their rates are given in an input file, the parsing of this file is done in a very static method, with a fixed number of proteins and a fixed order of the proteins. Adding a reaction takes even more work. For this reason, the large reactions array (the array indicating what the effect of each reaction is on the model) has to be adjusted. If it is just a new reaction only the height would increase, but for each added protein (or modelled substance) each current reaction has to increase in length. The first step here would be to change the input file in such a way that it contains not only the reaction rates. It could contain a list of, proteins, reaction rates, reaction equations and reaction effects. Especially the last two require quite some work. It might be best to start working with XML style of input files. This is because the reaction equations are quite flexible. The simple ones might be easily parsed but there are more complex equations that reference multiple different proteins and reaction rates. Linking all of this together might be a difficult task.

However, it would not be impossible and might even be worth the time. At the time of writing this thesis the current model has already become slightly outdated. Since *S. pneumoniae*is such a widely researched bacteria, new ideas

about the workings of competence appear quite often. A more flexible program should allow quick adjustments to the program to continue using the same basis while including the updates.

It should even be possible to look a step further. To create a more flexible Gillespie Algorithm program that is able to not only work with this *S. pneumoniae*model but is able to be used as an alternative for all ODE simulations. This could be done, since the conversion from ODE to stochastic equations has a fixed set of rules. The main difficulty would be to create an easy method for the program to parse the ODE equations into a standard format. This without enforcing difficult time consuming tasks for the user, such as the reaction conversion that is needed.

There are a lot of examples and usable code for different languages of the Gillespie Algorithm. However implementing if from a biological or ODE model still takes quite some time and also some coding experience. Especially when implemented in a low level programming language for speed, which would be recommended if the simulation would be used extensively. This might be the reason why it is not used very often in comparison to ODE simulations. As explained earlier this lack of ready to use tools, could probably be solved. Creating an easy to use Gillespie program that only requires extra processing hours, and not human resources, might increase the usage of stochastic simulating.

Besides the scalability and reusability the program could also benefit from other optimizations, mainly by the use of parallel execution. For a single run parallelization would slightly increase the execution time, this is because the only part that can be executed in parallel is the calculation of the reaction hazards. The benefit from this is limited while the overhead for combining the results for each time step would cost more. However since it is often recommended to take an average of multiple runs making those runs run in parallel would improve the total run time significantly.

Lastly if expended to a full program, an easy to use Gillespie program would benefit from a more expanded data output. When using averages it would be good to have an easy way to find separate spikes for each value. This cannot solely be done with keeping a maximum and minimum value for each run. Besides storing the average of the multiple runs also storing the separate runs would allow the user to see specific data from runs where a certain peak appeared. This would increase the user's awareness of possible anomalies that could occur.

Considering this specific biological model. While working on the stochastic simulation Moreno has expended the deterministic model such that that it includes cell growth. As said in the discussion a possible problem with the simulation's and the in-vitro experiments activation time might have been caused because of the lack of simulated cell growth. Besides this the difference for the ComX protein could not be explained within this thesis. An in-vitro experiment could have the possibility to confirm which simulation is a closer fit. Followed by an investigation into why this difference occurs. Especially since this difference also occurs when the quantity of the ComX protein is high, while it is assumed that when the quantities are high the difference between stochastic and deterministic simulation should be close to equal.

8 Conclusion

This thesis explored the differences between stochastic and deterministic modelling and simulating of biological processes. Doing this would also allow us to investigate the process that is needed for a conversion from deterministic to stochastic simulation. The ultimate aim would be that of analysing the possibility of automating the conversion process.

This was explored by converting a deterministic simulation of the competence shutdown of *S. pneumoniae*into a stochastic simulation. I cooperated with Stefany Moreno who created the deterministic simulation, using a system of ODEs. The Gillespie Algorithm was chosen for the stochastic simulation. This algorithm was chosen because it is well documented and easy to implement.

Working on the conversion, from deterministic to stochastic simulation, gave insight into this specific model. The steps needed for converting a system of ODEs into the reactions for the Gillespie algorithm, are steps that are possible to be done by an automated computer program. This indicates that creating a program that is able to convert system ODEs into a stochastic Gillespie simulation is, in theory, possible. However, it is important to note that this conclusion is only based on this one specific of system ODEs. Especially since the ODEs in this model are all relatively simple equations. Expanding into more general models and system of ODEs could quite possibly introduce new unforeseen problems.

The final comparison between the two models has shown some differences in the results, namely, the stochastic Gillespie algorithm has a lower competence activation time compared to the deterministic system of ODEs and the value of ComX is slightly higher in the results from the stochastic simulation. With the current information, from the two simulations, I am unable to point to the simulation that is the most similar to the natural behaviour.

For these reasons, to be able to give a comprehensive answers to our two original questions, several additional elements have to be explored. To create a general purpose ODE to Gillespie converter, a more extended analysis of different system ODEs are needed. Exploring different exceptions in the converting process is necessary to be able to make it easy to use for a large public. Lastly, to understand the reasons for the differences between the created stochastic solution and the deterministic solution, a more in-depth analysis of both simulations is needed with possibly an extension of the stochastic simulation such that it also includes cell growth.

References

[1] Weekly Epidemiological. Pneumococcal conjugate vaccine for childhood immunization–WHO position paper. *Releve epidemiologique hebdomadaire / Section d'hygiene du Secretariat de la Societe des Nations = Weekly epidemiological record / Health Section of the Secretariat of the League of Nations*, 82(12):93–104, 2007.

[2] Daniel T. Gillespie. A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. *Journal of Computational Physics*, 22(4):403–434, 1976.

[3] Ronald N. Jones, Michael R. Jacobs, and Helio S. Sader. Evolving trends in Streptococcus pneumoniae resistance: Implications for therapy of community-acquired bacterial pneumonia. *International Journal of Antimicrobial Agents*, 36(3):197–204, 2010.

[4] Diana Karlsson, Stefan Karlsson, Erik Gustafsson, Birgitta Henriques Normark, and Patric Nilsson. Modeling the regulation of the competence-evoking quorum sensing network in Streptococcus pneumoniae. *Bio Systems*, 90(1):211–23, 2007.

[5] L. R. Petzold. http://www.oecd-nea.org/tools/abstract/detail/uscd1227, 2015.

[6] Nicolas Mirouze, Mathieu a Bergé, Anne-Lise Soulet, Isabelle Mortier-Barrière, Yves Quentin, Gwennaele Fichant, Chantal Granadel, Marie-Françoise Noirot-Gros, Philippe Noirot, Patrice Polard, Bernard Martin, and Jean-Pierre Claverys. Direct involvement of DprA, the transformation-dedicated RecA loader, in the shut-off of pneumococcal competence. *Proceedings of the National Academy of Sciences of the United States of America*, 110(11):E1035–44, March 2013.

[7] Stefany Moreno-Gamez, Supervisors:, Jan-willem Veening, Franz J Weissing, and Morten Kjos. *CSP production and its pivotal role in pneumococcal competence development*. Master thesis, University of Groningen, 2013.

[8] Andrew Piotrowski, Ping Luo, and Donald a. Morrison. Competence for genetic transformation in Streptococcus pneumoniae: Termination of activity of the alternative sigma factor ComX is independent of proteolysis of ComX and ComW. *Journal of Bacteriology*, 191(10):3359–3366, 2009.

[9] Scipy. http://www.scipy.org/, 2015.

[10] J Anthony G Scott, W Abdullah Brooks, J S Malik Peiris, Douglas Holtzman, and E Kim Mulholland. Review series Pneumonia research to reduce childhood mortality in the developing world. *J Clin Invest*, 118(4):1291–300, 2008.

[11] Jamie Twycross, Leah R Band, Malcolm J Bennett, John R King, and Natalio Krasnogor. Stochastic and deterministic multiscale models for systems biology: an auxin-transport case study. *BMC systems biology*, 4:34, January 2010.

[12] D a Watson, D M Musher, J W Jacobson, and J Verhoef. A brief history of the pneumococcus in biomedical research: a panoply of scientific discovery. *Clinical infectious diseases : an official publication of the Infectious Diseases Society of America*, 17(5):913–924, 1993.

9 Appendix A: Stochastic Equations

9.1 Degradation and Synthesis

$$
\begin{aligned}
E_H &= \frac{ComE}{K_H^E} \\
EP_C &= \frac{ComEP^{dim}}{K_C^{EP}} \\
Y_C^{EP} &= \frac{EP_C}{EP_C+(1+E_H)^2}
\end{aligned}
$$

$$
\begin{aligned}
EP_X &= \frac{ComEP^{dim}}{K_X^{EP}} \\
E_L &= \frac{ComE}{K_L^E} \\
Y_X^{EP} &= \frac{EP_X}{EP_X+((1+E_L)*(1+E_H))}
\end{aligned}
$$

$$
\begin{aligned}
\lambda_{ComD} &= \frac{\sigma_D}{\delta_D^R}(\beta_D^0(1-Y_C^{EP})+\beta_D Y_C^{EP}) \\
\lambda_{ComE} &= \frac{\sigma_E}{\delta_E^R}(\beta_E^0(1-Y_C^{EP})+\beta_E Y_C^{EP}) \\
\lambda_{ComC} &= \frac{\sigma_C}{\delta_C^R}(\beta_C^0(1-Y_C^{EP})+\beta_C Y_C^{EP}) \\
\lambda_{ComAB} &= \frac{\sigma_{AB}}{\delta_{AB}^R}(\beta_{AB}^0(1-Y_C^{EP})+\beta_{AB} Y_C^{EP}) \\
\lambda_{ComX} &= \frac{\sigma_X}{\delta_X^R}(\beta_X^0(1-Y_X^{EP})+\beta_X Y_X^{EP})
\end{aligned}
$$

ComAB

$$ ComAB \xrightarrow{\delta_{AB}} \phi \tag{1} $$

$$ \phi \xrightarrow{\lambda_{ComAB}} ComAB \tag{2} $$

ComC

$$ ComC \xrightarrow{\delta_C} \phi \tag{3} $$

$$ \phi \xrightarrow{\lambda_{ComC}} ComC \tag{4} $$

CSP

$$ CSP \xrightarrow{\delta_{CSP}} \phi \tag{5} $$

ComD

$$ ComD \xrightarrow{\delta_D} \phi \tag{6} $$

$$ \phi \xrightarrow{\lambda_{ComD}} ComD \tag{7} $$

$ComD^{dim}$

$$ ComD^{dim} \xrightarrow{\delta_{D_{dim}}} \phi \tag{8} $$

ComE

$$ ComE \xrightarrow{\delta_E} \phi \tag{9} $$

$$ \phi \xrightarrow{\lambda_{ComE}} ComE \tag{10} $$

ComEP

$$ComEP \xrightarrow{\delta_{EP}} \phi \tag{11}$$

$ComEP_{dim}$

$$ComEP_{dim} \xrightarrow{\delta_{EP_{dim}}} \phi \tag{12}$$

$ComDP^{dim}$

$$ComDP^{dim} \xrightarrow{\delta_{DP_{dim}}} \phi \tag{13}$$

ComX

$$ComX \xrightarrow{\delta_X} \phi \tag{14}$$

$$\phi \xrightarrow{\lambda_{ComX}} ComX \tag{15}$$

9.2 Export

$$ComAB + ComC \xrightarrow{\frac{e}{ComC+k_e}} ComAB + CSP \tag{16}$$

$$2ComD \xrightarrow{d_D} ComD^{dim} \tag{17}$$

$$ComD^{dim} \xrightarrow{d_D^-} 2ComD \tag{18}$$

$$ComD^{dim} \xrightarrow{k_0} ComDP^{dim} \tag{19}$$

$$ComDP^{dim} \xrightarrow{k^-} ComD^{dim} \tag{20}$$

$$ComDP^{dim} + ComE \xrightarrow{\lambda} ComD^{dim} + ComEP \tag{21}$$

$$2ComEP \xrightarrow{d_{EP}} ComEP^{dim} \tag{22}$$

$$ComEP^{dim} \xrightarrow{d_{EP}^-} 2ComEP \tag{23}$$

$$ComD^{dim} \xrightarrow{CSP*k} ComDP^{dim} \tag{24}$$

9.3 Added equations for Dpra

$$XdprA = \frac{ComX}{K^X}$$

$$\lambda_{DprA} = \frac{\sigma_{DprA}*\beta_{DprA}}{\delta^R_{DprA}} * (\frac{X_{DprA}}{1+X_{DprA}})$$

$$DprA \xrightarrow{\delta_{DprA}} \phi \tag{25}$$

$$\phi \xrightarrow{\lambda_{DprA}} DprA \tag{26}$$

$$2DprA \xrightarrow{d_{DprA}} DprA^{dim} \tag{27}$$

$$DprA^{dim} \xrightarrow{\delta_{DprA^{dim}}} \phi \tag{28}$$

$$DprA^{dim} \xrightarrow{d^{-}_{DprA}} 2DprA \tag{29}$$

$$DprA^{dim} + ComEP^{dim} \xrightarrow{r} DEP \tag{30}$$

$$DEP \xrightarrow{r^{-}} DprA^{dim} + ComEP^{dim} \tag{31}$$

$$DEP \xrightarrow{\delta_{DEP}} \phi \tag{32}$$

10 Appendix B: ODE equations

10.1 Degradation and Synthesis

$$
\begin{aligned}
E_H &= \frac{ComE}{K_H^E} \\
EP_C &= \frac{ComEP^{dim}}{K_C^{EP}} \\
Y_C^{EP} &= \frac{EP_C}{EP_C+(1+E_H)^2}
\end{aligned}
$$

$$
\begin{aligned}
EP_X &= \frac{ComEP^{dim}}{K_X^{EP}} \\
E_L &= \frac{ComE}{K_L^E} \\
Y_X^{EP} &= \frac{EP_X}{EP_X+((1+E_L)*(1+E_H))}
\end{aligned}
$$

Synthesis of ComAB

$$
\frac{dComAB}{dt} = \frac{\sigma_{AB}}{\delta_{AB}^R}(\beta_{AB}^0(1-Y_C^{EP})+\beta_{AB}Y_C^{EP}) - \delta_{AB}ComAB
$$

Synthesis of ComC and export of CSP

$$
\begin{aligned}
\frac{dComC}{dt} &= \frac{\sigma_C}{\delta_C^R}(\beta_C^0(1-Y_C^{EP})+\beta_C Y_C^{EP}) - \delta_C ComC - \frac{eComABComC}{ComC+k_e} \\
\frac{dCSP}{dt} &= \frac{eComABComC}{ComC+k_e} - \delta_{CSP}CSP
\end{aligned}
$$

Synthesis of ComD

$$
\begin{aligned}
\frac{dComD}{dt} &= \frac{\sigma_D}{\delta_D^R}(\beta_D^0(1-Y_C^{EP})+\beta_D Y_C^{EP}) - 2d_D ComD^2 + 2d_D^- ComD^{dim} - \delta_D ComD \\
\frac{dComD^{dim}}{dt} &= d_D ComD^2 - k_0 ComD^d im + k^- ComDP^{dim} - kCSPComD^{dim} + \lambda ComDP^{dim}ComE \\
&\quad -\delta_{Ddim}ComD^{dim} \\
\frac{dComDP^{dim}}{dt} &= k_0 ComD^d im + kCSPComD^{dim} - k^- ComDP^{dim} - \lambda ComDP^{dim}ComE \\
&\quad -\delta_{DPdim}ComDP^{dim}
\end{aligned}
$$

Synthesis of ComE and ComEP

$$
\begin{aligned}
\frac{dComE}{dt} &= \frac{\sigma_E}{\delta_E^R}(\beta_E^0(1-Y_C^{EP})+\beta_E Y_C^{EP}) - \lambda ComDP^{dim} - \delta_E ComE \\
\frac{dComEP}{dt} &= \lambda ComDP^{dim} - 2d_{EP}ComEP^2 + 2d_{EP}^- ComEP^{dim} - \delta_{EP}ComEP \\
\frac{dComEP^{dim}}{dt} &= d_{EP}ComEP^2 - d_{EP}^- ComEP^{dim} - \delta_{EPdim}ComEP^{dim} - \\
&\quad rDprA^{dim}ComEP^{dim} + r^- DEP
\end{aligned}
$$

Synthesis of ComX and DprA

$$
\begin{aligned}
\frac{dComX}{dt} &= \frac{\sigma_X}{\delta_X^R}(\beta_X^0(1-Y_X^{EP})+\beta_X Y_X^{EP})-\delta_X ComX \\
\frac{dDprA}{dt} &= \frac{\sigma_{DprA}*\beta_{DprA}}{\delta_{DprA}^R}*(\frac{X_{DprA}}{1+X_{DprA}})-\delta^{DprA}DprA-2d_{DprA}DprA^2+2d_{DprA}^{-}DprA^{dim} \\
\frac{dDpra^{dim}}{dt} &= d_{DprA}DprA^2-d_{DprA}^{-}DprA^{dim}-rDprA^{dim}ComEP^{dim} \\
&\quad +r^{-}DEP-\delta_{DprA^{dim}}DprA^{dim} \\
\frac{dDEP}{dt} &= rDprA^{dim}ComEP^{dim}-r^{-}DEP-\delta DEPDEP
\end{aligned}
$$

11 Appendix C: Variable values

Listing 4: Parameter values.

```
55
1=0.0004 deltaAB  (d_ab) Degration Rates
2=0.0004 deltaC
3=0.0004 deltaCSP
4=0.0004 deltaD
5=0.0004 deltaDdim
6=0.0004 deltaE
7=0.0004 deltaEP
8=0.0004 deltaEPdim
9=0.0004 deltaDPdim
10=0.0004 deltaX
11=50 Ke       (ke)
12=0.006 e       (re)
13=0.024 dD      (dim)
14=0.1 dMinD     (dimn)
15=0.0 k0        (alp)
16=0.05 kMin     (kapd) 0.000110034  00012793387 00012773387
17=0.000126 lambda  (lamb)
18=0.024 dEP     (dep)
19=0.1 dMinEP      (depn)
20=0.0001 k      (kap)
21=10 Keh        (khe)
22=10 KEPC       (kcep)
23=0.04 sigAB      (sig_ab) translation rate
24=0.01 deltaRAB  (sigr_ab) RNA degradation rate
25=0.04 sigD
26=0.01 deltaRD
27=0.04 sigC
28=0.01 deltaRC
29=0.04 sigX
30=0.01 deltaRX
31=0.04 sigE
32=0.01 deltaRE
33=0.004 beta0AB  (b0_ab) Basal transcription rate
34=0.004 beta0D
35=0.004 beta0C
36=0.0 beta0X
37=0.004 beta0E
38=0.05 betaAB    (b_ab) ComEP induced Transcription rate
39=0.05 betaD
40=0.05 betaC
41=0.05 betaX
42=0.05 betaE
43=0.0004 deltaDprA (d_dpra)
44=0.024 dDprA    (ddpra)
45=0.1 dMinDprA   (dndpra)
46=0.0004 deltaDprAdim (d_dprad)
47=0.01 r        (r)
48=0.00015 rMin     (rn)
49=0.0004 deltaDEP  (d_dep)
50=10 K^X        (kx)
51=20 Kepx       (kxep)
52=20 Kel        (kle)
53=0.15 sigDpra   (sig_dpra)
54=0.01 deltaRDprA  (sigr_dpra)
55=0.0 betaDprA   (b_dpra)
```

12 Appendix D: Source Code

Listing 5: gillespie.cc

```
#include "reactions.h"
#include "gillespie.h"
#include "string"
#include <cstdlib>
#include <cmath>

using namespace std;

/* Generate next time interval */
double Gillespie::getNextTime()
{
    double randVal;
    do
    {
        randVal = ((double) rand() / (double)(RAND_MAX));
    } while ( randVal == 0);
    return ((1.0/reactions->getSum())*(log(1.0/randVal)));
}

/* Main loop iterating over steps. */
void Gillespie::run(double timeEnd, int cycles)
{
    double currentTime, measureTime;
    int idx;
    for(idx = 0; idx < cycles; ++idx)
    {
        currentTime = 0.0;
        measureTime = TIMESTEP;
        reactions->resetCurStat();
        while(currentTime < timeEnd)
        {
            reactions->calcHazard();
            currentTime += this->getNextTime();
            reactions->chooseReaction(currentTime);
            if(currentTime > measureTime)
            {
                reactions->measureState(measureTime);
                measureTime += TIMESTEP;
            }
        }
    }
    reactions->calcAverage();
    reactions->printStatus();
    reactions->writeResult();
}

int main(int argc, char *argv[])
{
    double timeEnd, cycles;
    double stepSize = 1;
    if(argc < 6)
    {
        cout << "not enough arguments give rate constants filename please. (filename outname timeEnd stepSize cycles)" << endl;
        return -1;
    }
    string fileName(argv[1]);
    string outName(argv[2]);
    timeEnd = (double)atoi(argv[3]);
    stepSize = (double)atoi(argv[4]);
    cycles = (double)atoi(argv[5]);
    TIMESTEP = stepSize;
    Reactions* react = new Reactions(fileName, outName, stepSize ,timeEnd, cycles);
    Gillespie alg(react);
    alg.run(timeEnd, cycles);
    return 0;
}
```

Listing 6: gillespie.h

```
#ifndef GILLESPIE_H
#define GILLESPIE_H
#include <cstdlib>
#include "reactions.h"
#include <time.h>

using namespace std;

double TIMESTEP = 1;

class Gillespie
{
public:
  Gillespie(Reactions* reactions) : reactions(reactions)
  {
    srand((unsigned)time(NULL));
  }
  ~Gillespie()
```

```
  {
    delete reactions;
  }

  void run(double endTime, int cycles);

private:
  Reactions* reactions;
  double getNextTime();
};
#endif
```

Listing 7: reactions.cc

```
#include <iostream>
#include <fstream>
#include <string>
#include <stdlib.h>
#include <boost/lexical_cast.hpp>
#include <limits>

#include "reactions.h"

using namespace std;
/* Defining the reactions */
int reactionsArray[NUMREACTIONS][NUMPROTEINS] ={{-1, 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0},
                                {1,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0},
                                {0, -1, 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0},
                                {0,  1,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0},
                                {0,  0, -1, 0,  0,  0,  0,  0,  0,  0,  0,  0,  0},
                                {0,  0,  0, -1, 0,  0,  0,  0,  0,  0,  0,  0,  0},
                                {0,  0,  0,  1,  0,  0,  0,  0,  0,  0,  0,  0,  0},
                                {0,  0,  0,  0, -1, 0,  0,  0,  0,  0,  0,  0,  0},
                                {0,  0,  0,  0,  0,  -1, 0,  0,  0,  0,  0,  0,  0},
                                {0,  0,  0,  0,  0,  1,  0,  0,  0,  0,  0,  0,  0},
                                {0,  0,  0,  0,  0,  0, -1, 0,  0,  0,  0,  0,  0},
                                {0,  0,  0,  0,  0,  0,  0, -1, 0,  0,  0,  0,  0},
                                {0,  0,  0,  0,  0,  0,  0,  0, -1, 0,  0,  0,  0},
                                {0,  0,  0,  0,  0,  0,  0,  0,  0, -1, 0,  0,  0},
                                {0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  1, 0,  0},
                                {0, -1,  1,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0},
                                {0,  0,  0, -2,  1,  0,  0,  0,  0,  0,  0,  0,  0},
                                {0,  0,  0,  2, -1,  0,  0,  0,  0,  0,  0,  0,  0},
                                {0,  0,  0,  0, -1,  0,  0,  0,  1,  0,  0,  0,  0},
                                {0,  0,  0,  0,  1,  0,  0,  0, -1,  0,  0,  0,  0},
                                {0,  0,  0,  0,  1, -1,  1,  0, -1,  0,  0,  0,  0},
                                {0,  0,  0,  0,  0,  0, -2,  1,  0,  0,  0,  0,  0},
                                {0,  0,  0,  0,  0,  0,  2, -1,  0,  0,  0,  0,  0},
                                {0,  0,  0,  0, -1,  0,  0,  0,  1,  0,  0,  0,  0},

                                {0,  0,  0,  0,  0,  0,  0,  0,  0,  0,-1,  0,  0},
                                {0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  1,  0,  0},
                                {0,  0,  0,  0,  0,  0,  0,  0,  0,  0,-2,  1,  0},
                                {0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,-1,  0},
                                {0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  2,-1,  0},
                                {0,  0,  0,  0,  0,  0,  0,  0, -1,  0,  0,-1,  1},
                                {0,  0,  0,  0,  0,  0,  0,  1,  0,  0,  0,  1,-1},
                                {0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,-1}};

/* Initialization of arrays and counter */
void Reactions::prepareMeasureArr(double stepSize, int timeSteps)
{
  int idx;
  numSteps = (timeSteps / stepSize) + 1;
  resultMatrix = new double*[numSteps];
  resultMinMaxMatrix = new double*[numSteps*2];
  for(idx = 0; idx < numSteps; ++idx)
  {
    resultMatrix[idx] = new double[NUMPROTEINS+1]();
    resultMinMaxMatrix[idx*2] = new double[NUMPROTEINS]();
    resultMinMaxMatrix[(idx*2)+1] = new double[NUMPROTEINS]();
  }
}

/* Resetting the run, also allows protein induction at start of a run */
void Reactions::resetCurStat()
{
  int idx;
    for(idx = 0; idx < NUMPROTEINS; ++idx)
    {
      curStat[idx] = 0;
    }

    //CSP induction!
  resultMatrix[0][CSP] += 1000;
    curStat[CSP] = 1000;
  stepsTaken = 1;
  flagComX = false;
}

/* Reading in the reaction rates from file */
void Reactions::loadReactionRates(string rateFile)
{
  ifstream iFile(rateFile.c_str());
    string token;
```

```
    double value;
    int num, idx, idy;
    getline(iFile, token);
  int amount = atoi(token.c_str());
  reactArr = new double[amount];
  rateValues = new double[NUMREACTIONS];
  curStat = new int[NUMPROTEINS];
    /* While there is still a line. Or untill all rates have been read. */
    while(getline(iFile, token)) {
      num = atoi(token.substr(0, token.find("=")).c_str());
      value = strtod(token.substr(token.find("=")+1, token.find(" ")).c_str(), NULL);
        reactArr[num-1] = value;
    }


    for(idx = 0; idx < NUMREACTIONS; ++idx)
    {
      rateValues[idx] = 0;
    }
    resultMatrix[0][idx] = 0;
    for(idy = 0; idy < numSteps; ++idy)
    {
      for(idx = 0; idx < NUMPROTEINS; ++idx)
      {
        resultMatrix[idy][idx] = 0;
        resultMinMaxMatrix[(idy*2)+1][idx] = std::numeric_limits<double>::min();
        resultMinMaxMatrix[(idy*2)][idx] = std::numeric_limits<double>::max();
      }
    }

    iFile.close();
}

/* Calculate the reaction hazard for all the reactions */
void Reactions::calcHazard()
{
  int idx;
  double Eh = curStat[ComE]/reactArr[Keh];
  double EPc = curStat[ComEPdim]/reactArr[KECP];
  double Ycep = EPc/(EPc + ((1+Eh) * (1+Eh)));

  double lambdaComAB = (reactArr[SigAB]/reactArr[deltaRAB]) * (reactArr[beta0AB] * (1 - Ycep) + reactArr[betaAB] * Ycep);
  double lambdaComD = (reactArr[SigD]/reactArr[deltaRD]) * (reactArr[beta0D] * (1 - Ycep) + reactArr[betaD] * Ycep);
  double lambdaComC = (reactArr[SigC]/reactArr[deltaRC]) * (reactArr[beta0C] * (1 - Ycep) + reactArr[betaC] * Ycep);
  double lambdaComE = (reactArr[SigE]/reactArr[deltaRE]) * (reactArr[beta0E] * (1 - Ycep) + reactArr[betaE] * Ycep);

  double EPx = curStat[ComEPdim]/(double)reactArr[Kepx];
  double El = curStat[ComE]/(double)reactArr[Kel];

  double Yxep = EPx/(EPx + ((1+El)*(1+El)));
  double lambdaComX = (reactArr[SigX]/reactArr[deltaRX]) * (reactArr[beta0X] * (1 - Yxep) + reactArr[betaX] * Yxep);

  double XdprA = curStat[ComX]/KX;
  double lambdaDprA = ((reactArr[SigDprA]*reactArr[betaDprA])/reactArr[deltaDprA])*(XdprA/(1+XdprA));


  rateValues[0] = curStat[ComAB] * reactArr[deltaAB];
  rateValues[1] = lambdaComAB;
  rateValues[2] = curStat[ComC] * reactArr[deltaC];
  rateValues[3] = lambdaComC;
  rateValues[4] = curStat[CSP] * reactArr[deltaCSP];
  rateValues[5] = curStat[ComD] * reactArr[deltaD];
  rateValues[6] = lambdaComD;
  rateValues[7] = curStat[ComDdim] * reactArr[deltaDdim];
  rateValues[8] = curStat[ComE] * reactArr[deltaE];
  rateValues[9] = lambdaComE;
  rateValues[10] = curStat[ComEP] * reactArr[deltaEP];
  rateValues[11] = curStat[ComEPdim] * reactArr[deltaEPdim];
  rateValues[12] = curStat[ComDPdim] * reactArr[deltaDPdim];
  rateValues[13] = curStat[ComX] * reactArr[deltaX];
  rateValues[14] = lambdaComX;
  rateValues[15] = curStat[ComAB] * curStat[ComC] * (reactArr[e]/(curStat[ComC]+reactArr[Ke]));
  rateValues[16] = curStat[ComD] * (curStat[ComD] -1) * reactArr[dD];
  rateValues[17] = curStat[ComDdim] * reactArr[dMinD];
  rateValues[18] = curStat[ComDdim] * reactArr[k0];
  rateValues[19] = curStat[ComDPdim] * reactArr[kMin];
  rateValues[20] = curStat[ComDPdim] * curStat[ComE] * reactArr[lambda];
  rateValues[21] = curStat[ComEP] * (curStat[ComEP]-1) * reactArr[dEP];
  rateValues[22] = curStat[ComEPdim] * reactArr[dMinEP];
  rateValues[23] = curStat[CSP] * curStat[ComDdim] * reactArr[k];


  //DPRA reactions
  rateValues[24] = curStat[DprA]*reactArr[deltaDprA];
  rateValues[25] = lambdaDprA;
  rateValues[26] = curStat[DprA] * (curStat[DprA] -1) * reactArr[dDprA];
  rateValues[27] = curStat[DprAdim] * reactArr[deltaDprAdim];
  rateValues[28] = curStat[DprAdim] * reactArr[dMinDprA];
  rateValues[29] = curStat[DprAdim] * curStat[ComEPdim] * reactArr[r];
  rateValues[30] = curStat[DEP] * reactArr[rMin];
  rateValues[31] = curStat[DEP] * reactArr[deltaDEP];
```

```
  // rateValues[24] = curStat[ComDPdim] * reactArr[kMin];

  rateSum = 0.0;
  for(idx = 0; idx < NUMREACTIONS; ++idx)
  {
    rateSum += rateValues[idx];
    if(rateValues[idx] < 0)
    {
      cout << "Error, negative value found!" << endl;
      cout <<  idx << " " << rateValues[idx] << " " << curStat[ComDPdim] <<endl;
      exit(-1);
    }
  }
}

/* Choosing the reaction, based on time given */
void Reactions::chooseReaction(double currentTime)
{
  double cumSumArr[NUMREACTIONS];
  double randVal;
  int idx, reacNum;
  int diffComX;

    randVal = ((double) rand() / (double)(RAND_MAX));
    randVal = randVal * rateSum;

  cumSumArr[0] = rateValues[0];
  if(cumSumArr[0] <= randVal)
  {
    for(idx = 1; idx < NUMREACTIONS; ++idx)
    {
      cumSumArr[idx] = rateValues[idx] + cumSumArr[idx-1];
      if(cumSumArr[idx] > randVal)
      {
        reacNum = idx;
        break;
      }
    }
  }
  else
  {
    reacNum = 0;
  }
  diffComX = curStat[9];
  for(idx = 0; idx < NUMPROTEINS; ++idx)
  {
    curStat[idx] += reactionsArray[reacNum][idx];
  }
  if( flagComX == false && diffComX < 150 && curStat[9] >= 150)
  {
    cout << currentTime << endl;
    flagComX = true;
  }
}

/* Store the current state in the result arrays */
void Reactions::measureState(double measureTime)
{
  int idx;
  if(stepsTaken > numSteps)
  {
    cerr << "Error does not compute" << endl;
    exit(-1);
  }
  for(idx = 0; idx < NUMPROTEINS; ++idx)
  {
    resultMatrix[stepsTaken][idx] += curStat[idx];
    if(curStat[idx] > resultMinMaxMatrix[(stepsTaken*2)+1][idx])
    {
      resultMinMaxMatrix[(stepsTaken*2)+1][idx] = curStat[idx];
    }
    if(curStat[idx] < resultMinMaxMatrix[(stepsTaken*2)][idx])
    {
      resultMinMaxMatrix[(stepsTaken*2)][idx] = curStat[idx];
    }

  }
  resultMatrix[stepsTaken][idx] = measureTime;
  stepsTaken++;
}

double Reactions::getSum()
{
  return rateSum;
}

/* Calculate average over multiple runs */
void Reactions::calcAverage()
{
  int idx, idy;
  // cout << " #: " << resultMatrix[0][CSP] << endl;
  for(idx = 0; idx < numSteps; ++idx)
  {
    for(idy = 0; idy < NUMPROTEINS; ++idy)
    {
      resultMatrix[idx][idy] = resultMatrix[idx][idy] / cycles;
    }
  }
```

```
}

void Reactions::printStatus()
{
  int idx;
  for(idx = 0; idx < NUMPROTEINS; ++idx)
  {
    // cout << PROTNAMES[idx*3] << " #: " << resultMatrix[stepsTaken-1][idx] << endl; //avg
    // cout << PROTNAMES[(idx*3)+1] << " #: " << resultMinMaxMatrix[stepsTaken*2+1][idx] << endl; //min
    // cout << PROTNAMES[(idx*3)+2] << " #: " << resultMinMaxMatrix[((stepsTaken-1)*2)+1][idx] << endl; //max
  }
}

void Reactions::writeResult()
{
  int idx, idy;
  ofstream outputFile(outputFileStr.c_str());

  outputFile << "Time, ";
  for(idx = 0; idx < NUMPROTEINS-1; ++idx)
  {
    outputFile << PROTNAMES[idx*3] << ", ";
    outputFile << PROTNAMES[(idx*3)+1] << ", ";
    outputFile << PROTNAMES[(idx*3)+2] << ", ";
  }
  outputFile << PROTNAMES[idx*3] << ", ";
  outputFile << PROTNAMES[(idx*3)+1] << ", ";
  outputFile << PROTNAMES[(idx*3)+2] << endl;

  for(idx = 0; idx < numSteps; ++idx)
  {
    outputFile << resultMatrix[idx][NUMPROTEINS] << ", ";
    for(idy = 0; idy < NUMPROTEINS-1; ++idy)
    {
      outputFile << resultMatrix[idx][idy] << ", ";
      outputFile << resultMinMaxMatrix[(idx*2)][idy] << ", ";
      outputFile << resultMinMaxMatrix[(idx*2)+1][idy] << ", ";
    }
    outputFile << resultMatrix[idx][idy] << ", ";
    outputFile << resultMinMaxMatrix[(idx*2)][idy] << ", ";
    outputFile << resultMinMaxMatrix[(idx*2)+1][idy] << endl;
  }

}
```

Listing 8: reactions.h

```
#ifndef REACTIONS_H
#define REACTIONS_H

#include <iostream>
#include <string>
#include <sys/stat.h>


/* Enum with all the different proteins */

const int NUMPROTEINS = 13;
const int NUMREACTIONS = 32;

const std::string PROTNAMES[] = {
      "ComAB",
      "minComAB",
      "maxComAB",
      "ComC",
      "minComC",
      "maxComC",
      "CSP",
      "minCSP",
      "maxCSP",
      "ComD",
      "minComD",
      "maxComD",
      "ComDdim",
      "minComDdim",
      "maxComDdim",
      "ComE",
      "minComE",
      "maxComE",
      "ComEP",
      "minComEP",
      "maxComEP",
      "ComEPdim",
      "minComEPdim",
      "maxComEPdim",
      "ComDPdim",
      "minComDPdim",
      "maxComDPdim",
      "ComX",
      "minComX",
      "maxComX",
      "DprA",
      "minDprA",
```

```
        "maxDprA",
        "DprAdim",
        "minDprAdim",
        "maxDprAdim",
        "DEP",
        "minDEP",
        "maxDEP"
        };

const int ComAB = 0;
const int ComC = 1;
const int CSP = 2;
const int ComD = 3;
const int ComDdim = 4;
const int ComE = 5;
const int ComEP = 6;
const int ComEPdim = 7;
const int ComDPdim = 8;
const int ComX = 9;
const int DprA = 10;
const int DprAdim = 11;
const int DEP = 12;

const int deltaAB = 0;
const int deltaC = 1;
const int deltaCSP = 2;
const int deltaD = 3;
const int deltaDdim = 4;
const int deltaE = 5;
const int deltaEP = 6;
const int deltaEPdim = 7;
const int deltaDPdim = 8;
const int deltaX = 9;
const int Ke = 10;
const int e = 11;
const int dD = 12;
const int dMinD = 13;
const int k0 = 14;
const int kMin = 15;
const int lambda = 16;
const int dEP = 17;
const int dMinEP = 18;
const int k = 19;
const int Keh = 20;
const int KECP = 21;
const int SigAB = 22;
const int deltaRAB = 23;
const int SigD = 24;
const int deltaRD = 25;
const int SigC = 26;
const int deltaRC = 27;
const int SigX = 28;
const int deltaRX = 29;
const int SigE = 30;
const int deltaRE = 31;
const int beta0AB = 32;
const int beta0D = 33;
const int beta0C = 34;
const int beta0X = 35;
const int beta0E = 36;
const int betaAB = 37;
const int betaD = 38;
const int betaC = 39;
const int betaX = 40;
const int betaE = 41;
const int deltaDprA = 42;
const int dDprA = 43;
const int dMinDprA = 44;
const int deltaDprAdim = 45;
const int r = 46;
const int rMin = 47;
const int deltaDEP = 48;
const int KX = 49;
const int Kepx = 50;
const int Kel = 51;
const int SigDprA = 52;
const int deltaRDprA = 53;
const int betaDprA = 54;

class Reactions
{
public:
  Reactions(std::string rateFile, std::string outName, double stepSize, int
        timeSteps, int runs)
  {
    prepareMeasureArr(stepSize, timeSteps);
    loadReactionRates(rateFile);
    fileName = rateFile;
    cycles = runs;
    stepsTaken = 1;
    outputFileStr = outName;
  }
  ~Reactions()
  {
    int idx;
    delete[] reactArr;
    delete[] curStat;
    delete[] rateValues;
    for(idx = 0; idx < numSteps; ++idx)
```

```
    {
      delete[] resultMatrix[idx];
        delete[] resultMinMaxMatrix[idx*2];
        delete[] resultMinMaxMatrix[(idx*2)+1];
    }
    delete[] resultMatrix;
    delete[] resultMinMaxMatrix;
  }

  void calcHazard();
  double getSum();
  void chooseReaction(double currentTime);
  void printStatus();
  void measureState(double measureTime);
  void writeResult();
  void calcAverage();
  void resetCurStat();

private:

  void loadReactionRates(std::string rateFile);
  void prepareMeasureArr(double stepSize, int timeSteps);

    int numProteins;
    int* curStat;
    double* rateValues;
    double* reactArr;
    double** resultMatrix;
    double** resultMinMaxMatrix;
    int numSteps;
    int stepsTaken;
    int cycles;
    double rateSum;
    bool flagComX;
    std::string fileName;
    std::string outputFileStr;

};
#endif
```