2011

Accelerating Finite State Projection through General Purpose Graphics Processing

Thomas Trimeloni
Virginia Commonwealth University

Accelerating Finite State Projection through General Purpose Graphics Processing

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Engineering at Virginia Commonwealth University.

by

Thomas V. Trimeloni,
B.S.E. – Miami University (2009)

Director: Dr. James M. McCollum,
Assistant Professor, Department of Electrical and Computer Engineering

Virginia Commonwealth University
Richmond, Virginia
April, 2011

Acknowledgement

The author wishes to thank several people. I would like to thank my parents for their support of my academic endeavors. I would like to thank all of my friends for their help in keeping me sane throughout the duration of this project. I would also like to thank Dr. Simpson for his consultation on this work. I would finally like to thank Dr. McCollum for his continued advice and direction throughout the project, and for being a great mentor.

Table of Contents

List of Figures

Abstract

ACCELERATING FINITE STATE PROJECTION THROUGH GENERAL PURPOSE GRAPHICS PROCESSING

By Thomas V. Trimeloni, B.S.E.

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Engineering at Virginia Commonwealth University.

Virginia Commonwealth University, 2011

Major Director: Dr. James M. McCollum,
Assistant Professor, Department of Electrical and Computer Engineering

The finite state projection algorithm provides modelers a new way of directly solving the chemical master equation. The algorithm utilizes the matrix exponential function, and so the algorithm's performance suffers when it is applied to large problems. Other work has been done to reduce the size of the exponentiation through mathematical simplifications, but efficiently exponentiating a large matrix has not been explored. This work explores implementing the finite state projection algorithm on several different high-performance computing platforms as a means of efficiently calculating the matrix exponential function for large systems.

This work finds that general purpose graphics processing can accelerate the finite state projection algorithm by several orders of magnitude. Specific biological models and modeling techniques are discussed as a demonstration of the algorithm implemented on a general purpose graphics processor. The results of this work show that general purpose graphics processing will be a key factor in modeling more complex biological systems.

Chapter 1 - Introduction

Computational modeling has proved to be an invaluable tool for engineers and scientists designing complex systems. In electrical and computer engineering, tools like Simulation Program with Integrated-Circuit Emphasis (SPICE) [1] and Matrix Laboratory (MATLAB) [2] allow engineers to build and debug designs in a virtual environment before ever producing a physical prototype of the system. This saves countless hours, as changes to the design can be made and tested quickly. Modeling allows engineers to sweep and optimize design parameters, ensuring that the best and safest design is produced. Without computational modeling some of engineering's most cutting edge designs, like the muti-core microprocessor or advanced fighter aircraft, would have been next to impossible to build.

A similar approach is now being employed for the analysis of biological data and the design of synthetic biological and pharmaceutical systems. Gene-regulatory networks are used by cells to control their basic functions needed to support life. Recent research has demonstrated that accurate modeling of gene-regulatory networks can reveal new insights into the inner workings of these systems. [3, 4, 5, 6, 7] By gaining a better understanding of these systems through modeling, invaluable knowledge will be gained as to how life works within a cell. Furthermore, by gaining an understanding of the gene-pathways in infectious diseases like HIV or the mitosis cycles in cancer cells, possible cures or treatments could be developed quickly and cheaply.

There are currently two different approaches to modeling a biological system. The first method is deterministic modeling, which models the system as a series of coupled differential equations. The second method is stochastic modeling, which uses computational methods to capture the stochastic fluctuations that occur in populations within the system. The advantage of

using stochastic modeling is that it can yield more information about the system by showing how these stochastic fluctuations can influence the behavior of the system. Stochastic Simulation Algorithms (SSAs) have traditionally been used to perform stochastic modeling, but they tend to be very computationally inefficient. It can also be difficult to assess the error in the resulting simulation.

A new algorithm for stochastic modeling, called Finite-State Projection (FSP) has recently been developed by Brian Munsky and Mustafa Khammash, [8] which attempts to address several of these issues associated with SSAs. Rather than producing a stochastic simulation, FSP numerically solves the underlying equations that govern the system. The FSP algorithm takes a user specified error as a parameter, and ensures the solution produced is accurate to within the error parameter. Since the algorithm solves the governing equations directly, reactions that occur relatively infrequently will still be weighted in the solution. This provides the modeler a more accurate means of modeling a system.

This work will address the computational inefficiencies of FSP, as a simple implementation can quickly become computationally costly. Systems being simulated with FSP can quickly grow to the point where a solution would take weeks or months to produce.

This work attempts to increase the performance of the FSP algorithm by exploring several different implementations of it. As FSP is largely a linear-algebraic problem, this work explores the performance of FSP in a parallel, high-performance computing environment. Through the use of multi-core processing and general-purpose graphics processing, a more computationally efficient implementation of FSP is proposed.

The second chapter of this document is intended to provide a background on stochastic modeling and FSP. The current strengths and limitations of these methods will be discussed. A

brief overview of current work in this area will be presented. The third chapter of this document will explain FSP in depth, and its implementation in MATLAB and C in regard to a basic gene expression network. A brief overview of the libraries and techniques used will be provided. This chapter will also provide examples and discussions of how even relatively simple models can become computationally intractable. The fourth chapter will overview the different techniques explored to accelerate the FSP algorithm. An overview and background will be provided on general-purpose graphics processing in specific regard to FSP. The success of these techniques will also be presented. The fifth chapter will demonstrate how gene-expression networks can be efficiently modeled through the use of FSP implemented in a general-purpose graphics processing environment. Models demonstrating negative-autoregulation, bi-stable oscillation, and the repressilator gene-regulatory network will be presented.

By more efficiently simulating gene-regulatory networks, scientists will be able to learn more about the functions cells use to maintain life. Furthermore, scientists will be able to develop new drugs and gene-therapy techniques more quickly and cheaply. This work attempts to provide scientists a more efficient and accurate tool for exploring the inner workings of gene-regulatory networks.

Chapter 2 - Modeling Gene Expression

This chapter will familiarize the reader with deterministic and stochastic modeling followed by a description of the finite-state projection (FSP) algorithm. A review of recent publications related to FSP is provided.

2.1 Gene Expression and Regulatory Networks

The goal of this section is to briefly introduce gene expression and gene regulatory networks to the reader. For a more detailed background on molecular genetics, consult [9].

Deoxyribonucleic Acid (DNA) contains coded information that describes the traits of an organism. These traits include everything from how a cell looks to how it performs specific functions. The process through which genetic information is retrieved from DNA and is utilized by the cell is called gene expression.

DNA is a double-helix shape composed of bases, specifically adenine (A), cytosine (C), guanine (G), and thymine (T). The bases form pairs, where A fuses to T and C fuses to G. A group of three bases is called a codon. Each codon codes for a specific amino acid. A group of codons that encode an entire protein is called a gene.

The DNA does not produce the proteins directly, but through a two-step process called transcription and translation. In transcription, DNA forms a complementary copy of itself by producing a single-stranded molecule called messenger ribonucleic acid (mRNA). RNA polymerase effectively "unwinds" the DNA molecule, so that RNA nucleotides can pair with the DNA's bases. The RNA polymerase then forms the rest of the mRNA molecule, which breaks the hydrogen bonds holding the new molecule of mRNA to the DNA. mRNA molecules may contain the genetic code for one protein, or more often contain the code for several. Once a strand of mRNA is transcribed, it produces protein through a process called translation. In

translation, a ribosome will bind to a section of mRNA responsible for a specific gene and produce an amino-acid chain called a polypeptide. The polypeptide will then "fold" into a specific protein. The proteins that are produced by a strand of DNA are biologically functional, meaning that they form the mechanism through which regulatory signals are passed throughout the cell.

Proteins are not all produced at equal rates. The production of some proteins can be induced by the presence of other molecules, causing their rate of production to increase. Similarly, protein production can also be repressed by the presence of other molecules. A protein molecule can bind to the promoter site of a specific gene, either increasing or decreasing the likelihood that RNA polymerase will bind and begin the process of transcription. Through these processes, protein production can be controlled and regulated through complex feedback systems. Proteins can regulate production of other proteins and sometimes even themselves.

An understanding of how gene expression and regulatory systems are implemented and can be constructed within a cell is of great interest to the pharmaceutical, biological engineering, and synthetic biology communities. As such, there is great interest in developing tools that can predict the outcome of biological experiments or assist in the interpretation of the data through modeling [3, 10]. The focus of this work is to accelerate a new method of modeling, called Finite State Projection, which accurately represents the noisy environment of the cell.

2.2 A Model of Transcription and Translation with Deterministic Methods

One of the most basic models is the simple transcription-translation model, which encompasses the production and decay of mRNA and proteins. It assumes that a gene produces mRNA according to rate constant k_m. Once the mRNA enters the system, it will produce proteins according to rate constant k_p, and eventually decay out of the system according to the

rate constant γ_m. Once a protein is produced, it decays out of the system according to the rate constant γ_p. A schematic of the model is shown in Figure 1.

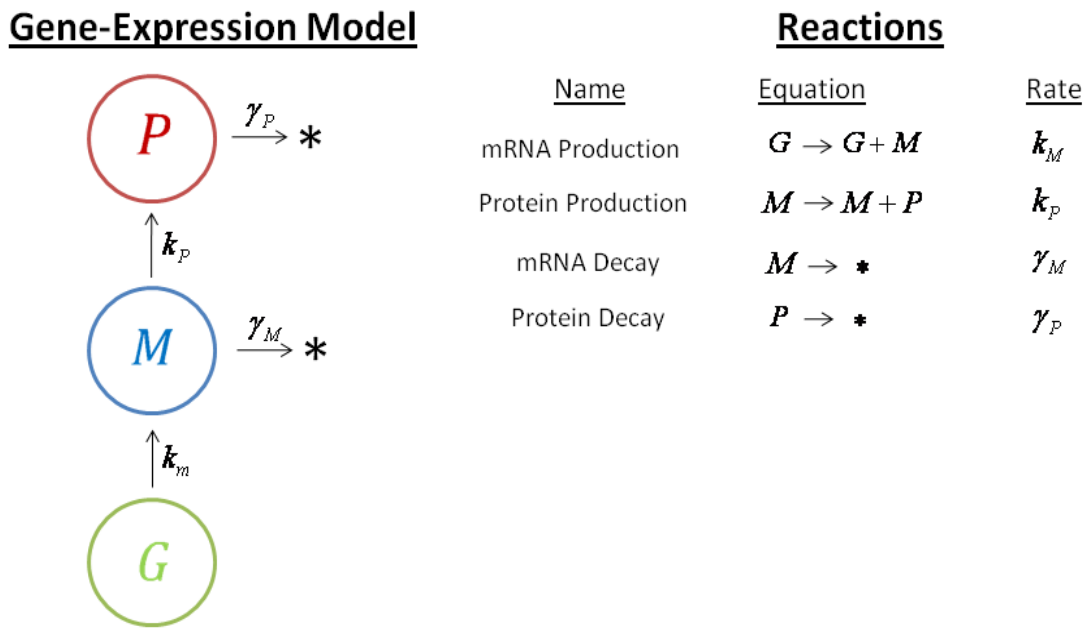


Figure 1 – Gene expression model.

The underlying framework through which the gene-expression model, or any other model of a chemically reacting system, works is through coupled ordinary differential equations (ODEs). For example, the coupled ordinary differential equations for the gene-expression model are shown below.

$$\frac{dM}{dt} = Gk_m - M\gamma_M \quad \textbf{(1)}$$

$$\frac{dP}{dt} = Mk_P - P\gamma_P \quad \textbf{(2)}$$

Equation 1 relates the change in the population of mRNA over time, and equation 2 relates the change in population of protein over time. Assuming each species starts with a

population of 0, and that k_m and k_p are equal to 3 and γ_M and γ_P are equal to 1, solving the ODEs yields a solution as shown in Figure 2.

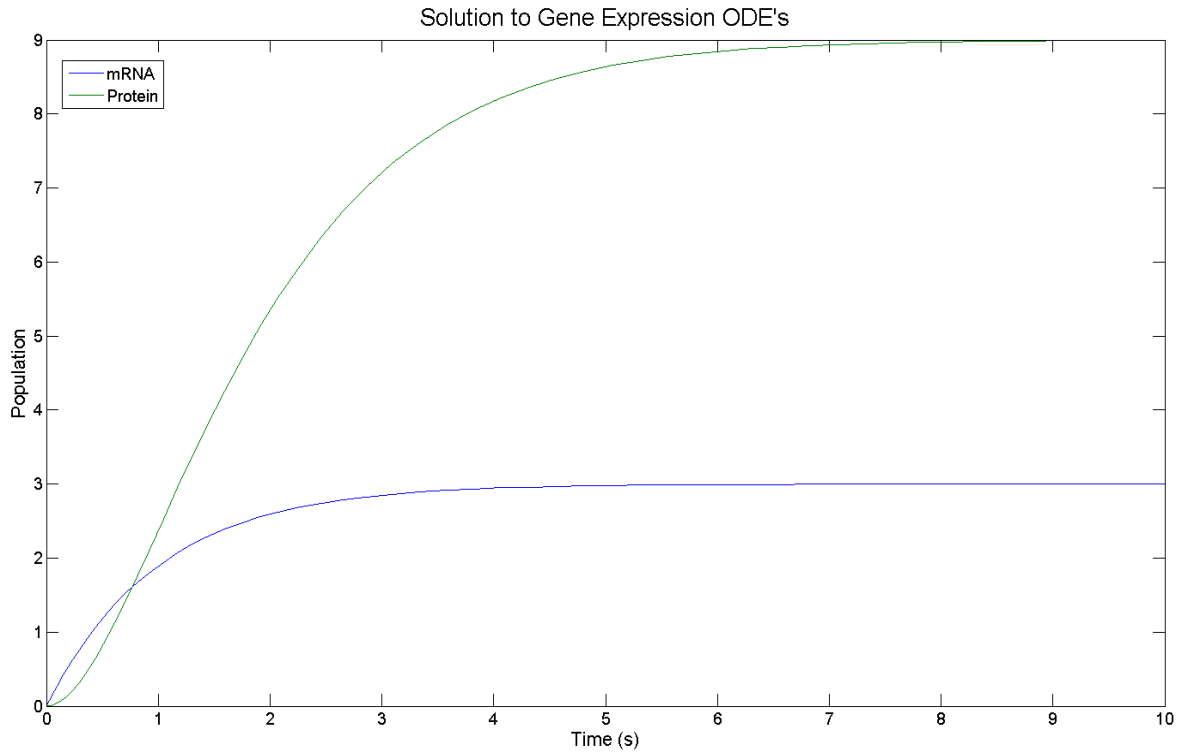


Figure 2 – Solution of ODEs describing gene expression.

As can be seen from Figure 2, we are able to get some information from the ODE solution. For example, we know that a mean population of 9 protein and 3 mRNA molecules should be expected in the system. It is important to note that the solution does not give any information about the noise of the system.

2.3 A Model of Transcription and Translation with Stochastic Methods

A more accurate formulation of the system can be made through the Chemical Master Equation (CME). [11] This formulation is as follows.

$$\dot{p}(x;t) = -p(x;t)\sum_{\mu=1}^{R} a_\mu(x) + \sum_{\mu=1}^{R} p(x - v_\mu; t)a_\mu(x - v_\mu) \quad \textbf{(3)}$$

The formulation calculates the change in probability of being in state *x* at time *t*. The first summation on the right side of the equation represents the probability that the system will remain in state *x*. In the summation, *R* represents the total number of reactions that can occur. The term $a_\mu(x)$ represents the propensity of each specific reaction μ occurring in state *x*. A propensity is simply the population of the reacting species times the specific rate constant. The second summation on the right side of the equation represents the probability that the system will change states. The term $p(x - v_\mu; t)$ represents the probability of transitioning from state *x* to the state represented by v_μ at time *t*. This probability is multiplied by $a_\mu(x - v_\mu)$, which represents the propensity of the reaction μ occurring in the state indexed in v_μ. The CME allows the population trends to be computed by Stochastic Simulation Algorithms (SSAs) as a "random walk", by approximating the CME through Monte-Carlo methods. Figure 3 shows the result of the stochastic simulation of the same gene expression system that was previously modeled deterministically. It should be noted that Figure 3 shows the result of only one run of an SSA. Hundreds or thousands of these simulations are typically produced in order to properly characterize the system statistically.

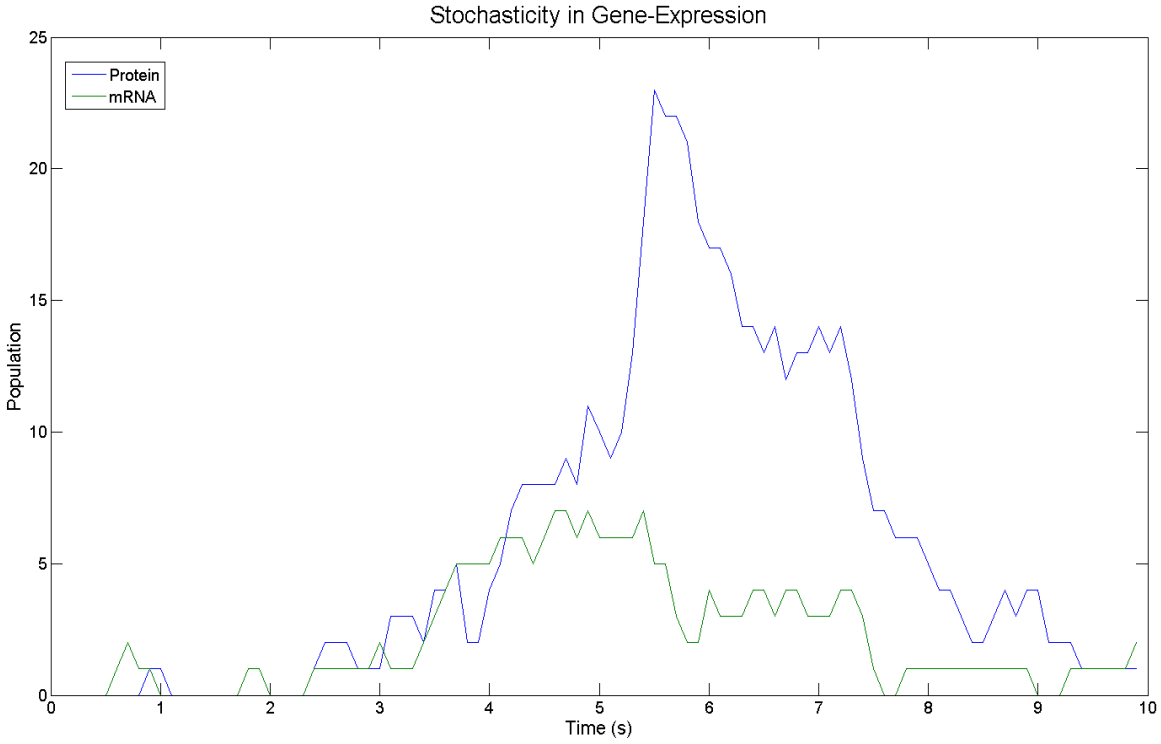


Figure 3 – Results of modeling gene expression with a SSA.

From observation, Figure 2 and Figure 3 are very different in terms of capturing the noise of the system. In a gene-regulatory network, there is more going on than can be described by simply solving the ODEs that describe the system. While the ODEs do describe the mean population levels of the system, they say nothing about the probability distributions of the populations. A large amount of research exists showing that the probability distributions of the system can yield valuable information about how the system works. [3, 4, 5, 7] This demonstrates the need for FSP, and similar algorithms.

2.4 The Finite State Projection Algorithm

Recent work by Brian Munsky and Mustafa Khammash has produced a new algorithm for simulating stochastic systems, called the Finite-State Projection (FSP) algorithm. Rather than solve the ODEs describing a system for a deterministic solution or use Monte-Carlo methods to simulate the system like an SSA, FSP solves or approximates the solution to the CME. By doing

so, FSP is able to calculate the probability distribution of states within the system at a given point in time. [8] In systems where the number of states is finite, the FSP algorithm will arrive at an exact numerical solution. Otherwise, the algorithm will approximate the solution to within a user specified error. The FSP algorithm is outlined below in equations 4, 5, and 6.

$$A_k = submatrix(X_0) \tag{4}$$

$$V_k = e^{A_k * t_f} * V_0 \tag{5}$$

If$(1 - sum(V_k) > \varepsilon)$ **then** increase states in X_0 and **goto** equation 4 (**6**)

In equation 1, X_0 represents the whole list of possible states the system can exist in at time 0. If the system is state limited, the size of X_0 will be finite. Otherwise, the list X_0 will be countably infinite in size. The algorithm begins by choosing a small number of states from X_0 and constructs A_k, a sub matrix representative of *k* states from X_0. The sub matrix A_k is constructed via the CME, where the rows and columns are indexed via the *k* selected states. Given a specific row *i* in A_k, each column element *j* represents the propensity that the system will transition from state *i* to state *j*. As such, the summation of each column will approximate the CME for each state, or equal the CME in systems that are state limited. Formally, A_k is constructed as follows:

$$A_{ij} = \begin{cases} -\sum_{\mu=0}^{R} a_\mu(x_i) & for\ i = j \\ a_\mu(x_i) & for\ all\ j\ that\ begin\ with\ i \\ 0 & otherwise \end{cases} \tag{7}$$

Equation 2 of the algorithm requires that the initial conditions of the system, V_0, be formulated by the user. V_0 is a probability vector of size k that gives the initial probabilities for each indexed state, and as it is a probability vector should sum to exactly 1. This vector is then multiplied by the result of the matrix exponential $e^{A_k * t_f}$, which yields the projected vector V_k. The vector V_k is then the evolution of the vector V_0 after t_f seconds have elapsed. The t_f, or final time parameter, is a user entered parameter that determines the time scale of the projection. As V_k is only representative of a truncated state-space, there will be some error in its calculation. Calculating the exact error in V_k, equation 3 of the algorithm, is simply computed by taking $1 - sum(V_k)$. By calculating V_k directly from the CME, the probabilities for all indexed states will be exact. As such, the total error in V_k is simply the sum of the probabilities of all of the missing states, or more simply 1 minus the sum of all the probabilities for computed states. If this error is greater than a tolerance ε provided by the user, more states are added and the algorithm repeats. Since a probability will never be negative, the addition of more states will always decrease the error. This makes a solution viable for almost any system through a concept called N-step reachability. [8] This also makes the assumption that the computer making these computations has enough memory and computational power to accomplish this task in a reasonable amount of time. By directly solving the CME for a probability distribution, FSP provides modelers the numerical accuracy of traditional ODE solutions, while providing statistically relevant information such as population variance that was previously only attainable through Monte-Carlo methods like SSAs. The solution for the system described previously is now presented as solved by FSP in Figure 4. Dark blue areas of the figure represent low probabilities and transitions to red represent areas of high probability.

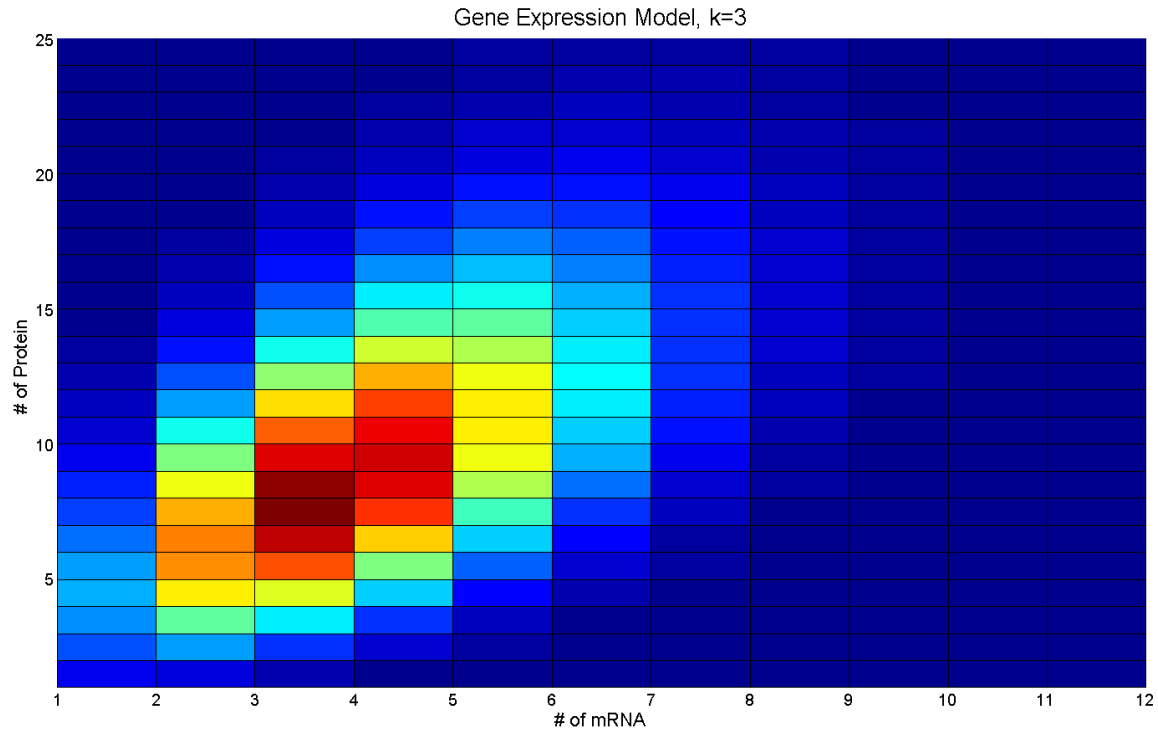


Figure 4 – Result of modeling the gene expression system with FSP

There are still drawbacks to the usage of FSP. The matrix-exponential function is very computationally intensive for large matrices and a large value of t_f, or final time parameter, poses problems as well, as demonstrated in Chapter 3 and Chapter 4 of this document.

2.5 Previous Work Involving Finite State Projection

A Krylov-space based algorithm was developed by a group lead by Roger Sidje that tries to circumvent the problem of computing large matrix-exponentials by reducing the problem to a Krylov-subspace using the Arnoldi process. While their method was able to show up to an order of magnitude speedup over the original algorithm, they did so at the cost of accuracy for certain problems. Their algorithm repeatedly steps forward through time until the final time parameter has been met, and as such too large of a step will reduce the accuracy of the projection. A large final time parameter will require more steps through time, and thus hinder the performance of the algorithm. [12]

The original authors of the FSP algorithm, Brian Munsky and Mustafa Khammash, have also expanded on their original work with FSP to create a version capable of producing an error-bounded output at periodic time intervals. Their implementation, called multiple time interval FSP, worked very efficiently for some models as it exploits the linearity of the projection. This reduces the size of the matrix to be exponentiated, but does not address how a large matrix can be efficiently exponentiated. [13]

Recent work by a group at the University of California, Santa Barbara has produced a new formulation of FSP called diffusive FSP. This version is designed to solve inhomogeneous systems, as the original algorithm assumes a homogeneous system. A homogeneous system assumes that all of the molecules within the system are uniformly spaced and that the system is well-mixed. If that is not the case, then the system is considered inhomogeneous. [14]

Chapter 3 – Performance Analysis of Finite State Projection

This chapter provides an analysis of the performance of the FSP algorithm to date. Performance will be quantified in terms of the problem size, and be presented in terms of the simple transcription-translation gene expression system. Different implementations of the algorithm are explored, and a background of each is provided.

3.1 MATLAB Implementation of Gene Expression Using FSP

A gene expression simulation can be coded in Mathworks MATLAB as follows. First, the concept of states in a gene regulatory network will be introduced. A state is any configuration that the system can take. For example, there would be a state in the gene expression model discussed in the previous chapter where the system contained 5 protein and 2 mRNA molecules. A truncated infinitesimal generator matrix must be generated that represents the state-space of interest. An infinitesimal generator matrix is a matrix that describes how the system can transition between states based on the CME discussed in the previous section. The rows and columns of the matrix are indexed by states, such that an element in the matrix would describe the propensity of transitioning from the row-indexed state to the column-indexed state. Since a gene-expression system can reach a countably infinite number of states, the state-space is truncated by removing states that have a very low likelihood of being reached. For example, a system that would have a mean expected protein population of one-hundred would be very unlikely to reach a population level of one-million. Knowing this, states that have more protein are not added unless the specified error criteria is not met.

Once a truncated state space has been established, generation of the infinitesimal generator matrix can begin. For any given system, this matrix will be square and have all negative values along its diagonal. Furthermore, because the state space is truncated the rows

and columns will not necessarily sum to be zero. MATLAB code that generates the infinitesimal generator matrix for a simple gene-expression system is presented below in Figure 5.

```
BigA = zeros((maxM+1)^2,(maxM+1)^2);
for rows=1:(maxM+1)^2
    BigA(rows,rows) = -km - floor((rows-1)/(maxM+1))*gammam - floor((rows-1)/(maxM+1))
*kp - mod(rows-1,maxM+1)*gammap;
    if rows-maxM-1 > 0
        BigA(rows-maxM-1,rows) = km;
    end
    if rows+maxM+1 <= (maxM+1)^2
        BigA(rows+maxM+1,rows) = floor((rows+maxM)/(maxM+1))*gammam;
    end
    if rows > 1
        BigA(rows,rows-1) = mod(rows-1,maxM+1)*gammap;
        if rows <= (maxM+1)^2
            if mod(rows-1,maxM+1) == 0
                BigA(rows-1,rows) = 0;
            else
                BigA(rows-1,rows) = floor((rows-2)/(maxM+1))*kp;
            end
        end
    end
end
tic
Prob = [1; zeros((maxM+1)^2-1,1)];
BigA = BigA.';
tempVec = expm(BigA*tf)*Prob;
```

Figure 5 – MATLAB code example for building the infinitesimal generator matrix

In the above code, the variable `BigA` is the infinitesimal generator matrix. The way this matrix is formulated for this, or any other system being modeled by FSP, is as follows.

$$A_{ij} = \begin{cases} -\sum_{\mu=0}^{R} a_\mu(x_i) & for\ i = j \\ a_\mu(x_i) & for\ all\ j\ that\ begin\ with\ i \\ 0 & otherwise \end{cases} \quad (8)$$

The variable `maxM` represents the maximum number of protein currently expected in the state space, and so states beyond that are not added. As in the previous description, the variables `km` and `kp` are the propensity of creating mRNA and protein respectively. Similarly, `gammam` and `gammap` are the propensity of mRNA and protein decay.

Once the infinitesimal generator matrix has been constructed, the next step is to project the state space out to the final time of interest. This is accomplished by multiplying the matrix by the scalar final time of interest, and performing a matrix exponential on the result. The resulting exponentiated matrix is multiplied by the vector representing the initial probability of the state-space. The vector that results from these steps is the projected state-space probability map of the system at the specified final time. [8] In MATLAB, this is done through the use of the built in matrix exponential function. The following line of code demonstrates how this is performed.

```
tempVec = expm(BigA*tf)*Prob;
```

`tempVec` is the new projected state-space probability vector, `tf` is the final-time of interest, and `Prob` is the initial condition vector.

Since FSP can only approximate the solution to a system with a large or infinite number of states, an error assessment is performed to see if the projection is within the given error criteria. A complete probability vector will always sum to one. However, since FSP operates on a truncated state-space the probability vector it produces will always be less than or equal to one. Knowing this, the error is simply the sum of the vector produced by FSP subtracted from one. This is shown in the following line of MATLAB code.

```
tempError = 1 - sum(tempVec)
```

Where tempVec is the projected state-space probability vector, and tempError is the current error for the given truncated state-space. If this error is not within the given error bounds, the state-space is expanded and the process is repeated until enough states are added to meet the given error criteria. Additional states must have a positive probability, so adding more

states to the truncated state-space will always increase the accuracy of the projection through N-step reachability. [8]

3.2 BLAS & C

The first step taken in accelerating FSP was the implementation of an efficient serial algorithm to compare against MATLAB. For our efficient serial implementation, we chose to write FSP in a C program that implements the Basic Linear Algebra Subprograms (BLAS) library.

MATLAB was originally developed in the 1970's by Cleve Moler, so that his students would be able to implement linear-algebra routines quickly without having to take the time to learn the FORTRAN programming language. [2] Today, MATLAB is still the primary tool used by engineers for numerical analysis due to the fact that it is relatively simple to implement an algorithm with MATLAB's plethora of built-in functions. However, MATLAB is usually hindered in terms of performance by the fact that its programs do not directly run on the computer. Programs run inside of MATLAB, meaning that the user needs to have MATLAB installed on their machine in order to run a MATLAB program. While this does help keep MATLAB programs simple to write, computational overhead is also introduced.

In order to get around the problem of computational overhead, an implementation of FSP was written in C that uses the BLAS library. C is a lower-level programming language than MATLAB, and as such requires less computational overhead. In order to run a program in C, all that is required is that the executable file be compiled for the user's specific system architecture. C was developed in the 1970's at Bell Laboratories, and since then has developed a reputation as being a very computationally efficient language. C was then the natural choice for developing a quick, portable version of FSP.

The BLAS libraries add further performance to serial implementation. The BLAS libraries were originally developed in 1979 at the University of Tennessee, and form the backbone of LAPACK, ATLAS, and many other libraries used in high-performance computing to this day due to their efficiency and portability. [15] The BLAS libraries provide functions to perform many different linear-algebraic operations, and so they were included in this work in order to efficiently implement the matrix exponential function.

What makes the comparison of these two different implementations interesting is that MATLAB uses ATLAS, which is a further optimized version of BLAS, to perform many of its linear algebra operations. What this means is that even though implementing FSP in C with BLAS should reduce the overhead from MATLAB, MATLAB still uses a more efficient set of linear algebra libraries. The results of this comparison are discussed at the end of this chapter.

3.3 Implementing FSP in C

In order to implement FSP in C, a way of computing the exponential of a matrix was needed. The exponential of a matrix X is defined by the following equation.

$$e^X = \sum_{k=0}^{\infty} \frac{1}{k!} X^k \tag{9}$$

The immediately obvious problem is that the direct computation of the exponential of a matrix involves an infinite summation. While this summation could be truncated in order to allow this computation to be done on a computer, accuracy would suffer greatly. This problem has been studied in great detail by Cleve Moler and Charles Van Loan in their two papers "Nineteen Dubious Ways to Compute the Exponential of a Matrix" and "Nineteen Dubious Ways to Compute the Exponential of a Matrix, Twenty-Five Years Later". [16, 17] These two

works detail nineteen different ways to exponentiate a matrix, characterizing each method in terms of the shape of the matrix and computational efficiency. Since the shape of the matrices used in FSP will be largely model dependant, methods that worked best for the general case were ideal. Furthermore, since the goal of this work is to increase the performance of FSP without compromising accuracy, methods that were computationally efficient were also ideal. Given these two points, a literature search yielded a hybrid method of Pade Approximation combined with Scaling and Squaring. [17]

Pade Approximation of order p and q is defined by the following equation.

$$e^X \approx R_{pq} = \left[D_{pq}(X)\right]^{-1} N_{pq}(X) \tag{10}$$

Where

$$N_{pq}(X) = \sum_{j=0}^{p} \frac{(p+q+j)!\,p!}{(p+q)!\,j!\,(p-j)!} X^j \tag{11}$$

And

$$D_{pq}(X) = \sum_{j=0}^{q} \frac{(p+q+j)!\,q!}{(p+q)!\,j!\,(q-j)!} (-X)^j \tag{12}$$

As can be seen in equations 10, 11, and 12, all of the linear algebraic operations required for the computation of a matrix exponential are scalar-matrix multiplication, matrix-matrix multiplication, matrix-matrix addition, and matrix inversion. All of these operations can be easily implemented with BLAS and can also be easily performed in a parallel environment.

Moler and Loan even go so far as to test different values of p and q for accuracy depending on the shape of X. They establish that in the general case, p=q=10 yields the best accuracy for this method. One limitation to this method, as described by Moler and Loan, is if the norm of matrix X is large. This can cause significant roundoff error, especially as the norm of X increases above 1. This can be addressed with the additional Scaling and Squaring method, which is defined as follows.

$$e^X = (e^{X/m})^m \quad \textbf{(13)}$$

The main idea behind equation 13 is that the norm of X can be lowered by multiplying it by the scalar term $1/m$. In order to achieve the best performance and accuracy, *m* should be chosen to be the largest power of 2 such that $\frac{\|X\|}{m} \leq 1$. By choosing *m* such that this condition is satisfied the Pade approximant will have the greatest accuracy and the number of matrix-matrix multiplies required for the squaring step will be reduced. The source code for these operations as performed in each FSP algorithm can be seen in the appendix of this document. This is also the way MATLAB performs the matrix exponential operation, which can be seen in the source code of MATLAB's expm function.

3.4 Multi-Core Computing & ATLAS

For the next implementation, our C implementation was modified to use the Automatically Tuned Linear Algebra Software (ATLAS) library. This library is a modified version of the BLAS libraries, but is specifically tuned to the host computing architecture on install. In this case, the host processor was an AMD Phenom II quad-core processor. What is meant by saying the library is "tuned" is that each linear-algebraic operation is optimized to take advantage of specific designs in the hardware that the library is running on. For example, certain

instructions in a given instruction set may take more or less time on a given CPU architecture, or may not even be available. As such, different combinations of instructions can reduce the processing time on different architectures while still producing a mathematically equivalent result. [18] This addition levels the playing-field between our algorithm and MATLAB in terms of efficient linear algebra computations.

Another advantage that was afforded to both the MATLAB implementation and our C implementation in this test is the ability use all four of the host computer's CPU cores. Multi-core computing has recently become a mainstay in desktop computing. Almost all desktop computers being sold presently come with a multi-core processor. These processors provide several distinct advantages over their single-core counterparts. The most obvious being the ability to run two different programs, or two different parts of the same program, at the same time. This provides an advantage to this work, as almost all of the computation required for an iteration of the FSP algorithm can be broken up such that each core on the CPU can perform a part of the operation. For example, think about a scalar term is being multiplied by a vector. If this was being done in serial, the scalar would be multiplied by the first element in the vector, and once that had completed the scalar would be multiplied the second vector element, and so on. A quad-core machine would be able to accomplish this operation much faster, as each multiplication does not depend on the result of any of the others. Rather than wait for the scalar to be multiplied by the first element and then commence with the second element, the second element could be multiplied by the scalar on a second core.

Another advantage provided by multi-core computing is the proximity of the cores to each other. In other distributed computing environments if two tasks running in parallel needed information about one another every so often, that information would have to be put on a hard

drive or sent over a network. These methods can introduce large amounts of latency in the operation, and as such can bottleneck performance. In a multi-core machine, each core physically resides within the same chip. By doing so latency can be decreased, as AMD designed the chip such that information that needs to be shared can be placed in faster memories like the processor's L3 cache. [19] The performance results of this implementation will be discussed in the next section of this chapter.

3.5 Performance Evaluation of FSP in MATLAB

For systems that require small numbers of states to achieve an accurate projection, the MATLAB implementation of FSP works remarkably well. However, as more states are added the computation time increases as the square of the number of states in the projection. This is shown below in Figure 6.

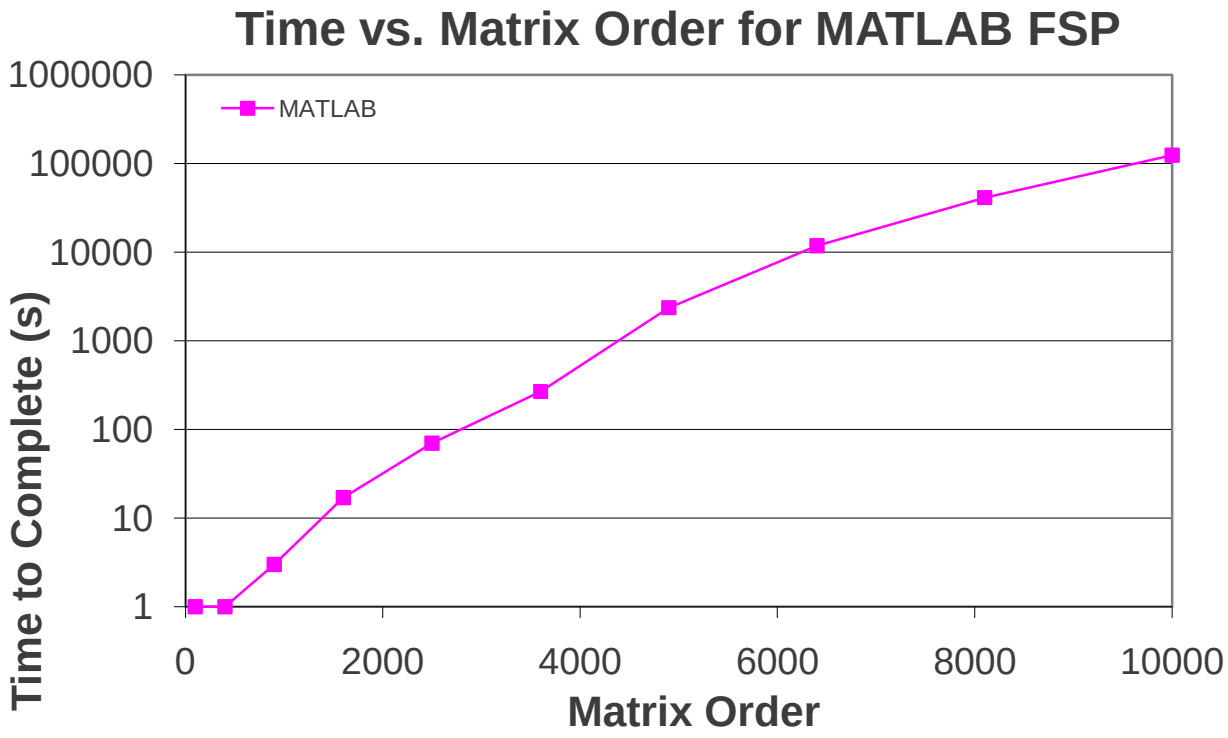


Figure 6 – Wall clock computation time of serial MATLAB FSP

As can be seen from Figure 6, a system requiring 10,000 states will be problematic for MATLAB to solve. Just one iteration of the FSP algorithm involving 10,000 states took 34 hours on a modern computer using one thread. Considering that the FSP algorithm repeatedly iterates, gradually expanding the state-space that it is operating in, it is reasonable to see how simulations of this magnitude could take weeks or even months to complete.

3.6 Performance Results of BLAS & C

Once a serial version of FSP was implemented in C and BLAS, its performance was compared to the MATLAB implementation for various problem sizes. The C & BLAS implementation was designed to run on one of the CPUs cores, and the source code can be viewed in the appendix of this document. The source code was compiled using all of the compiler's optimization flags. For each test, a simple gene expression model was used to generate the states. In order to ensure the accuracy and fairness of these tests, the number of states was passed into the program as a parameter and the approximate infinitesimal generator matrix was constructed to be of that specific size. For the purposes of establishing a baseline test, MATLAB was restricted to running in one thread so that it would also only be able to take advantage of the processing power of one CPU core. These results are shown below in Figure 7 and Figure 8.

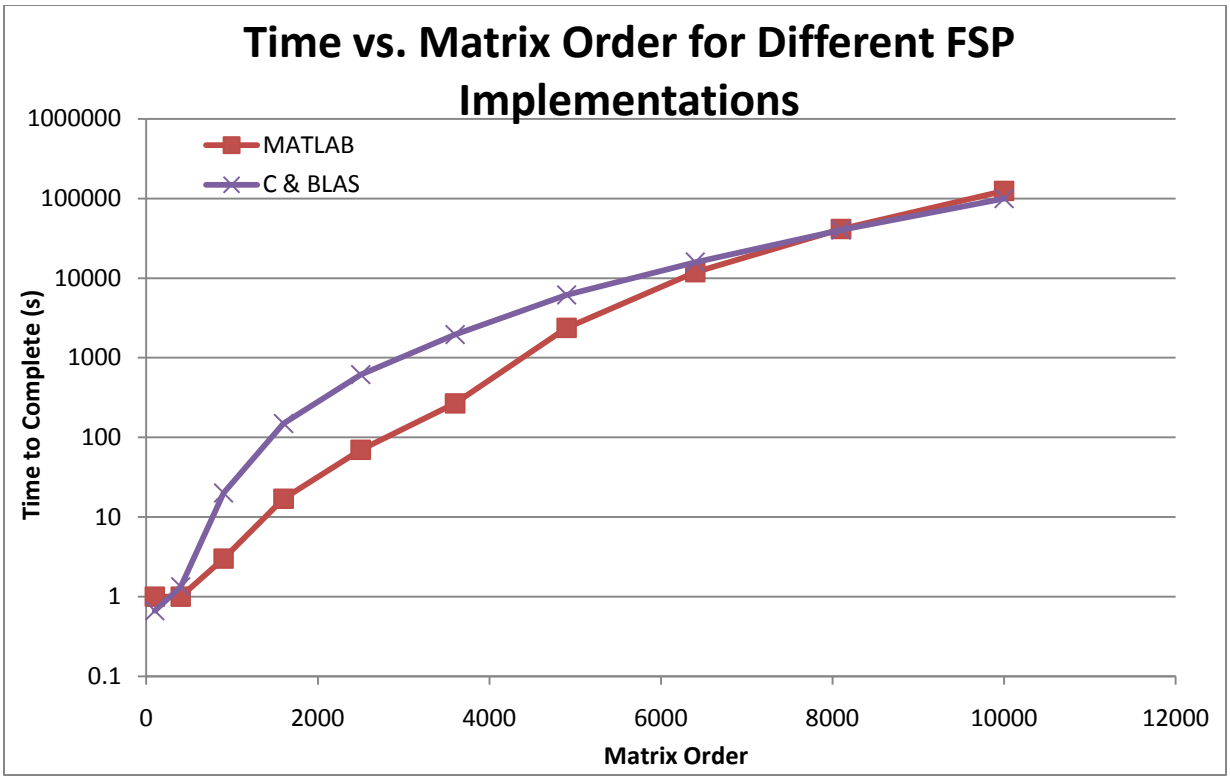


Figure 7 – Wall clock computation time of BLAS FSP and serial MATLAB FSP

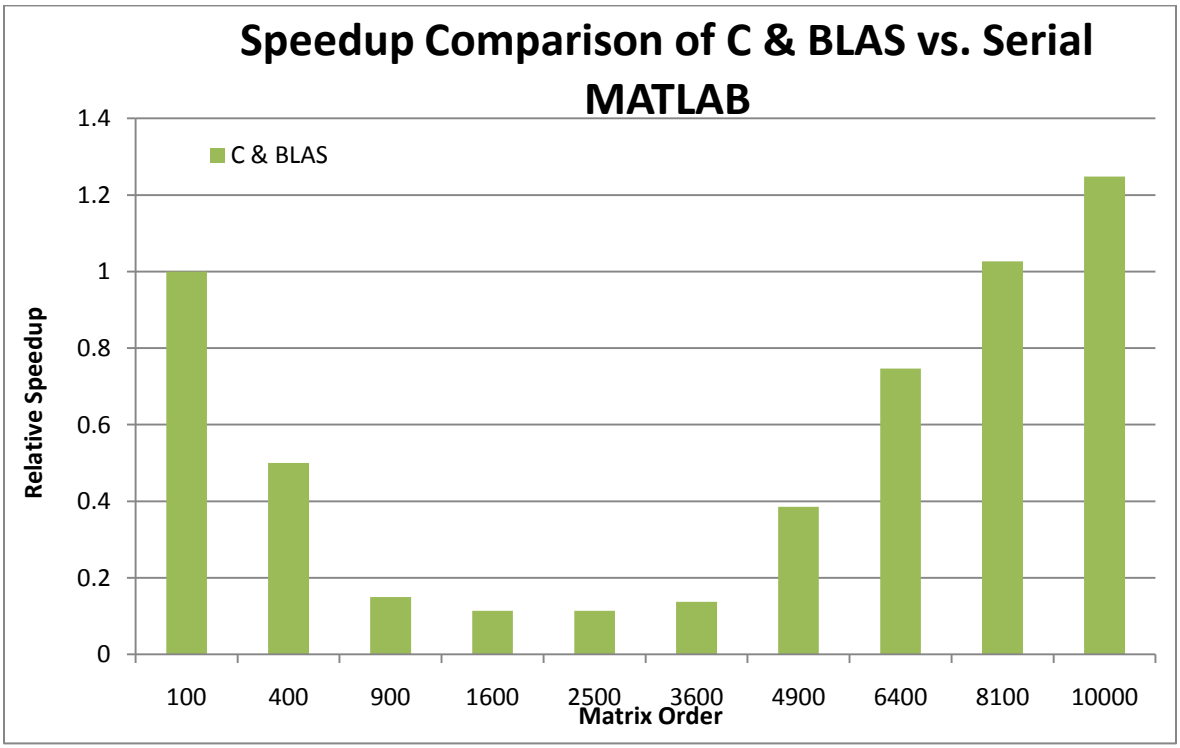


Figure 8 – Speedup comparison of BLAS FSP and serial MATLAB FSP

As can be seen from the above figures, the MATLAB implementation of FSP can solve smaller problems quicker, but as the problem size grows the overhead from MATLAB increases and our C & BLAS implementation becomes slightly faster. MATLAB is also optimized to deal with smaller, sparse matrices, and so it makes sense that MATLAB would be faster for these smaller tests. The fact that our C & BLAS implementation was able to perform faster than MATLAB for the larger tests indicates that our version is coded efficiently. As shown in Figure 7, a matrix of order 2500 takes the MATLAB FSP algorithm about 1 minute to iterate through, while a matrix of order 10,000 takes MATLAB about 34 hours. Because of this, the focus needs to be on improving performance for larger problems. It is important to realize that most biological system models will require a very large number of states, further emphasizing the fact that performance for large matrices should be the goal. The results also confirm the hypothesis that MATLAB's overhead introduces computational inefficiency, as our algorithm was still able to perform better than MATLAB for matrices of order 8,100 and greater while using a less-optimized version of BLAS.

3.7 Multi-Core and ATLAS Results

The results of the comparison between the MATLAB implementation of FSP and our C implementation using ATAS are as follows. The test was set up in the same way as the serial test. The same simple gene expression model was used to generate the states, and the state-size was given as a parameter to the program. The results are shown in Figure 9 and Figure 10.

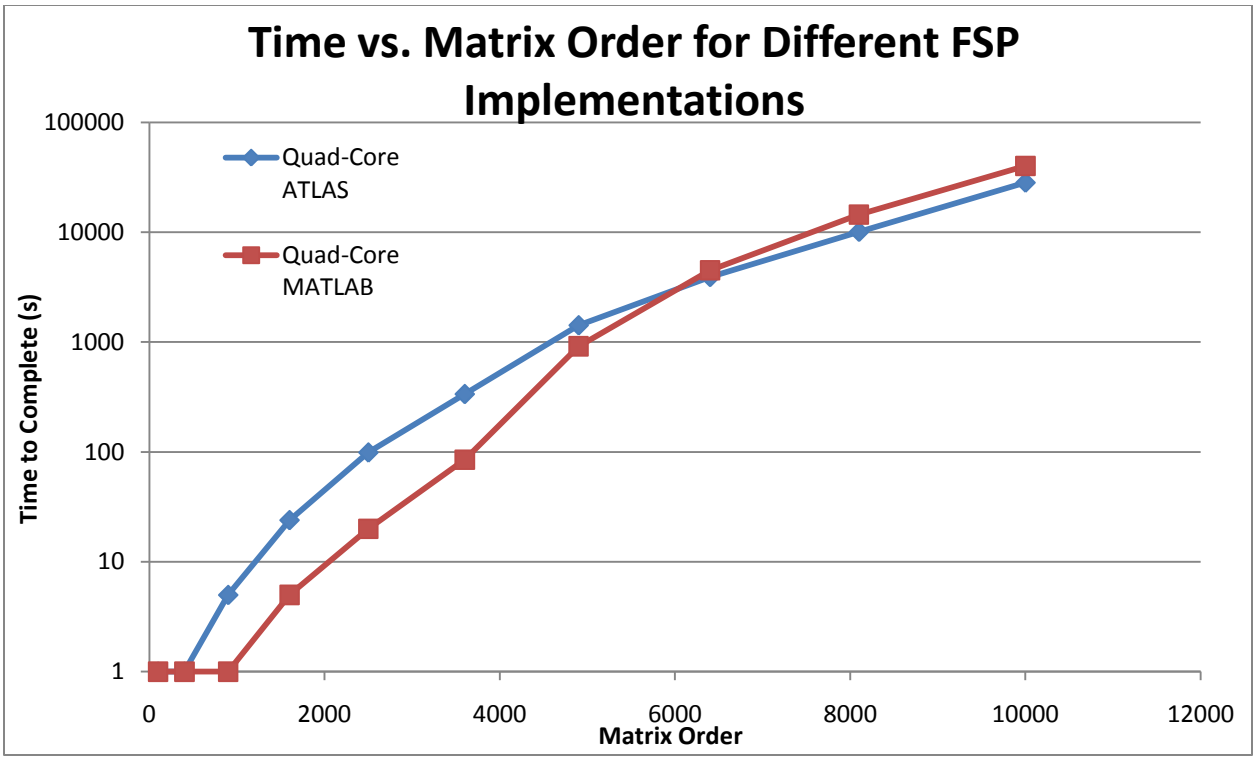


Figure 9 – Wall clock computation time for quad-core versions of FSP

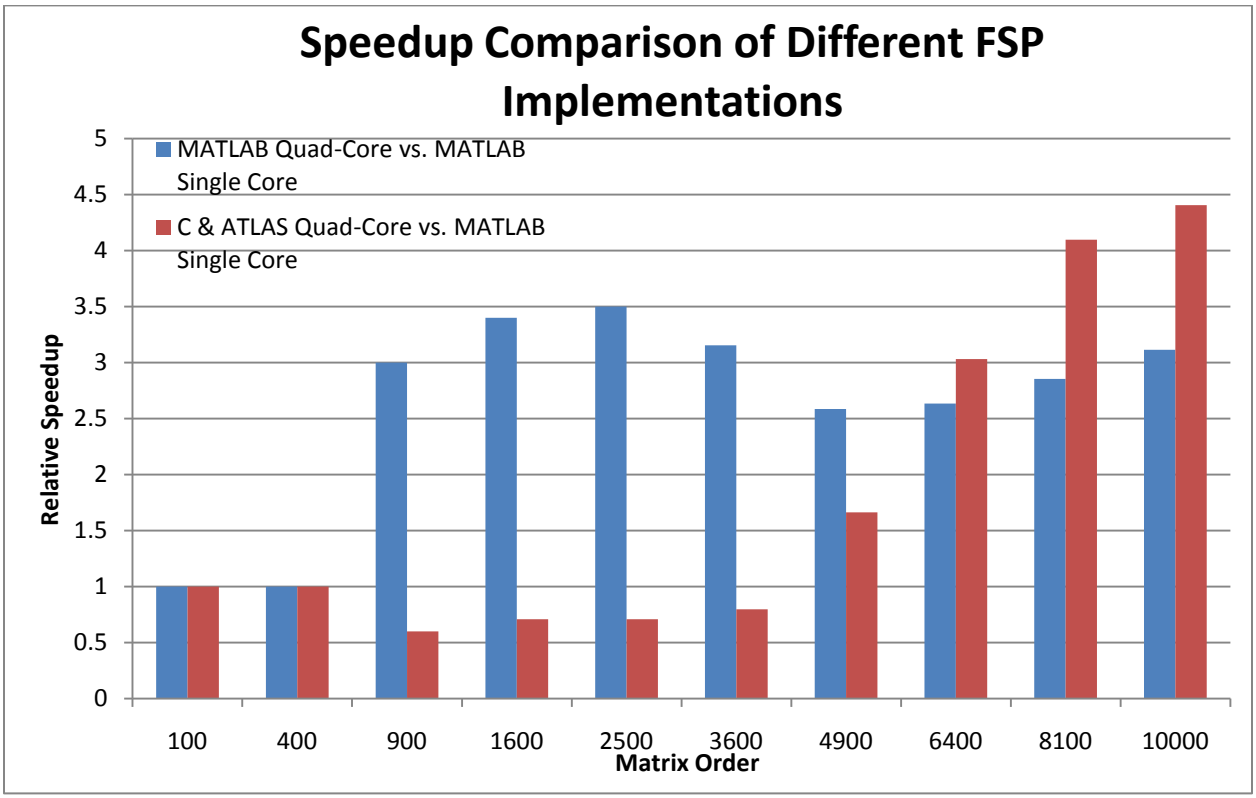


Figure 10 – Speedup comparison of the quad-core FSP implementations to serial MATLAB

As shown in the previous two figures, our C implementation using ATLAS was the fastest for large problem sizes, achieving a 4.4x speedup against single-core MATLAB and a 1.4x speedup against quad-core MATLAB for a matrix of order 10,000. While MATLAB is still consistently faster for smaller problem sizes, it is important to note the scale of the computation time relative to the speedup. For example, when the matrix was of order 2500, MATLAB using all four cores finished about 1 minute faster than our C implementation. However, when the matrix was of order 10,000, the C implementation finished more than three hours before MATLAB using four cores. What this means in terms of performance is that even though MATLAB was able to perform the smaller tasks faster, the test suite as a whole took about 4 and a half hours longer using MATLAB on all four cores than our C implementation using ATLAS. These results confirm our reasoning from section 3.6 that overall FSP performance will be improved the most by efficiently solving larger problems.

3.8 The Need for Faster FSP

The results from the previous section clearly demonstrate the need for an FSP algorithm that can efficiently handle these large systems. In the gene-expression model, the last matrix exponentiation alone took approximately 80 seconds to complete. Even when MATLAB was allowed to use all 4 cores of a modern AMD Phenom II processor, an exponential rise in computation time is still observed for systems needing a relatively large state-space. This is shown in Figure 11, and relative speedup is shown in Figure 12.

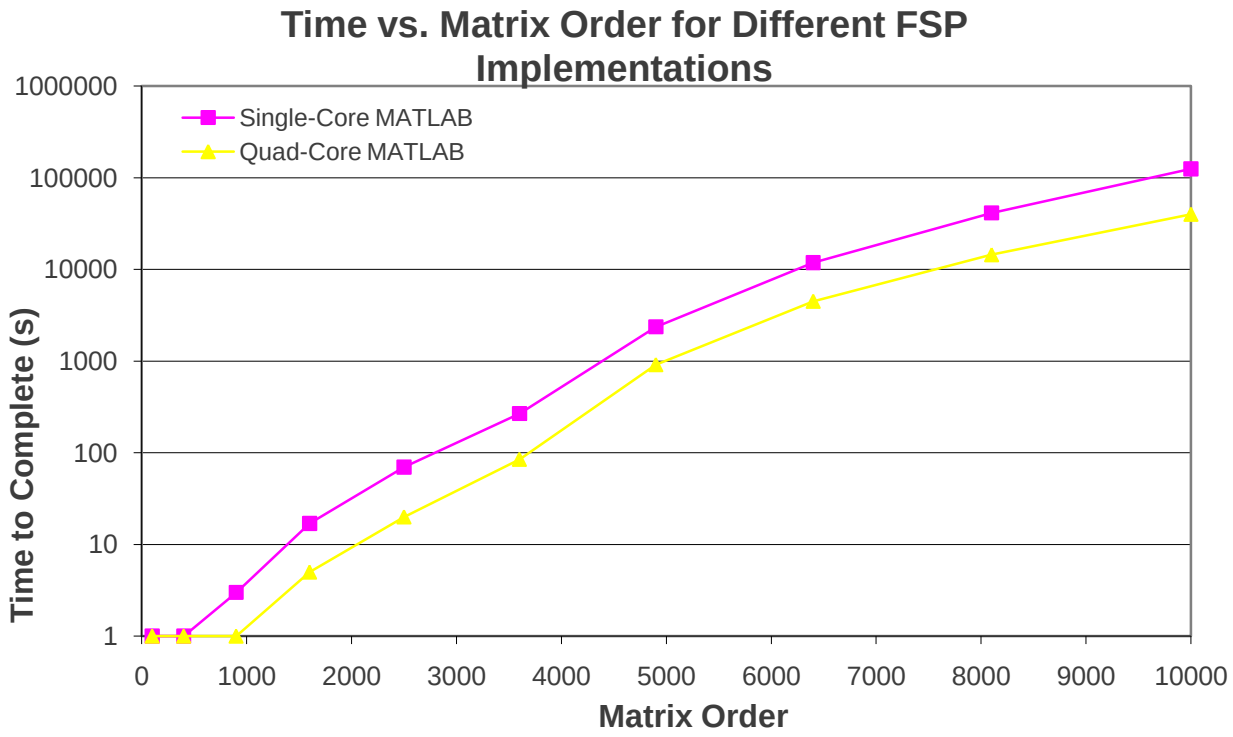


Figure 11 – Wall clock computation time of serial and quad-core MATLAB FSP

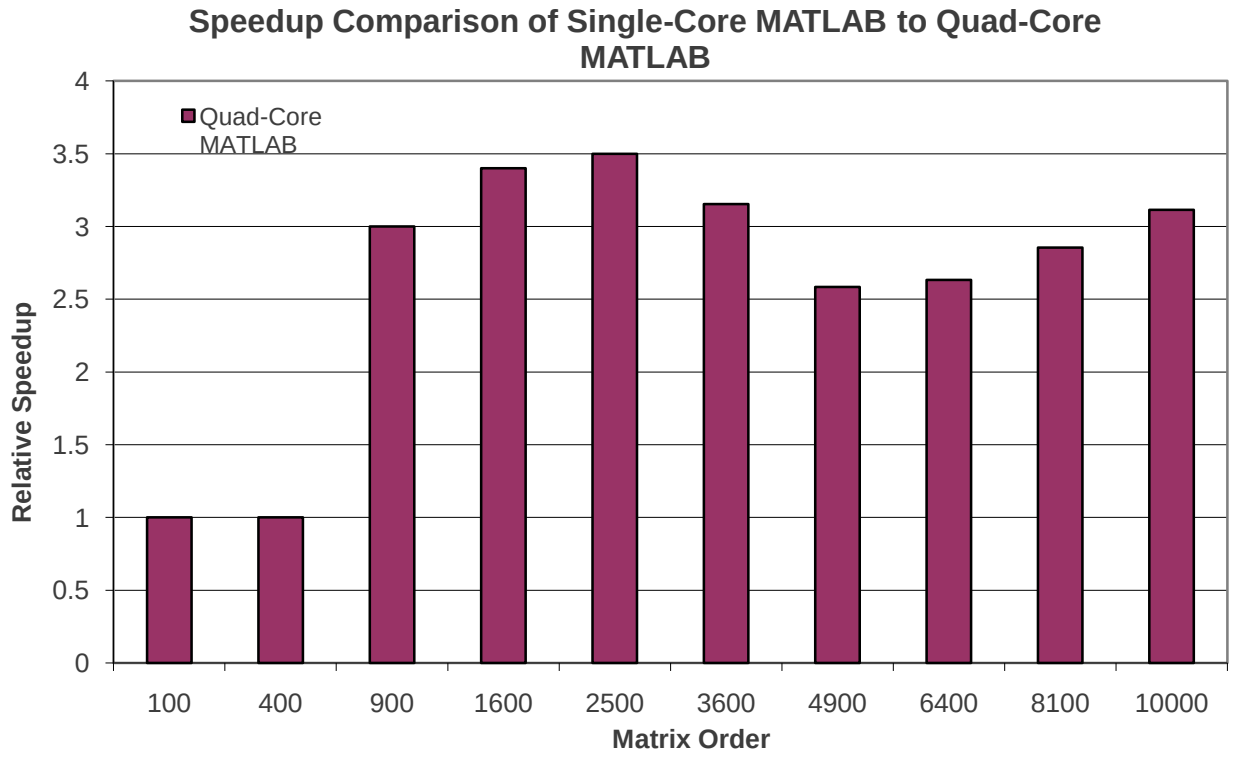


Figure 12 – Speedup comparison of serial and quad-core MATLAB FSP

Even though moving from one core to four cores did improve performance, a large gene expression system would still take longer than one week to simulate. Furthermore, the computation time data still shows an exponential increase in processing time. This suggests that running MATLAB in a multi-core environment would only allow simulations to be done quickly on slightly larger systems. We also observe a slight drop in the speedup MATLAB achieved when running on four cores as opposed to one, as the largest speedup observed occurred with a matrix of order 2500. This also suggests that simply installing MATLAB on a machine with more cores is not a viable solution for simulating larger systems requiring a larger state-space.

Chapter 4 – Accelerating Finite State Projection with General Purpose Graphics Processing

This chapter discusses the use of General Purpose Graphics Processing Units (GPGPUs) in regard to FSP. Background information on GPGPUs and their application to the acceleration of matrix operations is also presented. State-saving measures are examined in regard to the gene-expression model that can be applied to even larger systems to be modeled. The performance of the algorithm in regard to the final time parameter is also discussed.

4.1 General-Purpose Graphics Processing

A new method for high-performance computing is the General-Purpose Graphics Processing Unit (GPGPU). These devices, which had previously only been used for image processing and video games, allow the FSP algorithm to be accelerated directly rather than through mathematical simplification. Their ability to handle large linear-algebra problems efficiently has been demonstrated and proven. [19, 20]

Since a majority of the operations required for the calculations of FSP are independent elementary row and column operations, this problem is well suited to parallelization. Traditionally, this would mean implementing finite state projection in either a shared-memory or distributed-memory parallel supercomputing environment. Recent work by Vasily Volkov and James Demmel shows that general-purpose graphics processors can handle large, linear algebra problems very efficiently. [19] The authors compare the speed of performing a LU-factorization on an Intel quad-core processor, a hybrid processor-graphics card method, and a pure graphics card method. The result of their work is that the pure graphics card method shows more than a 3x speedup over the processor in performing the LU-factorization.

Similarly to how multi-core processors achieve speedup through parallelism and physical proximity, GPGPUs allow for large performance gains by placing many cores on a single chip. Furthermore, the memory architecture of GPGPUs allows for very high throughput. The memory is designed to be clocked very quickly and has a very wide bus width. The GPGPU used for this test is a NVIDIA Tesla C1060. It has 240 thread processors that have access to 4 gigabytes of 512-bit wide RAM. What these numbers mean in terms of performance is that the NVIDIA Tesla is optimized for large linear-algebraic problems. It was designed to move and operate on large amounts of data quickly, which is exactly what is required for an implementation of FSP that is able to handle large problem sizes.

Recent work has demonstrated the performance gains of using GPGPUs in high-performance computing. The TOP500 [21] list is a ranking system devised by faculty from the University of Manheim, the University of Tennessee, and Lawrence Berkeley National Laboratory to rank the performance of supercomputers from around the world. Their test suite is mainly comprised of problems involving a package called LINPACK, which uses BLAS to solve numerical calculations. According to the TOP500 list, the current fastest computer is the Chinese made Tianhe-1A. This computer is designed to solve problems using over 14,000 multi-core CPUs, as well as using over 7,000 NVIDIA Tesla GPGPUs. What makes the Tianhe-1A unique is that the second fastest computer on the TOP500 list, Oak Ridge National Laboratory's Jaguar, has roughly 4,000 more CPUs than the Tianhe-1A but is still almost twice as slow due to its lack of inclusion of GPGPUs. [21]

Another reason the GPGPU is attractive is the advent of the CUBLAS library. This library provides the same function calls as the BLAS library, but has been optimized by NVIDIA

to run on their graphics processors. This makes programs that support BLAS simple to convert, allowing users very easy access to the processing power afforded by these novel devices.

4.2 General-Purpose Graphics Processing Results

The results of the GPGPU implementation of FSP are shown below. It is important to note that one section of code, the section that finds the inverse-matrix for the Pade approximation step in the FSP algorithm, was changed so that the rows of the infinitesimal generator matrix are zero-padded. This takes advantage of the memory architecture of the GPGPU, and was not done in the previous implementations as this addition would have slowed them down. The rest of the program that was running on the graphics card is a straight conversion of the original C implementation of FSP using BLAS. The source code can be viewed in the appendix of this document. Again, the test was performed using a simple gene-expression model to generate the states, with the number of states being a parameter passed into the program. The results are as follows in Figure 13 and Figure 14.

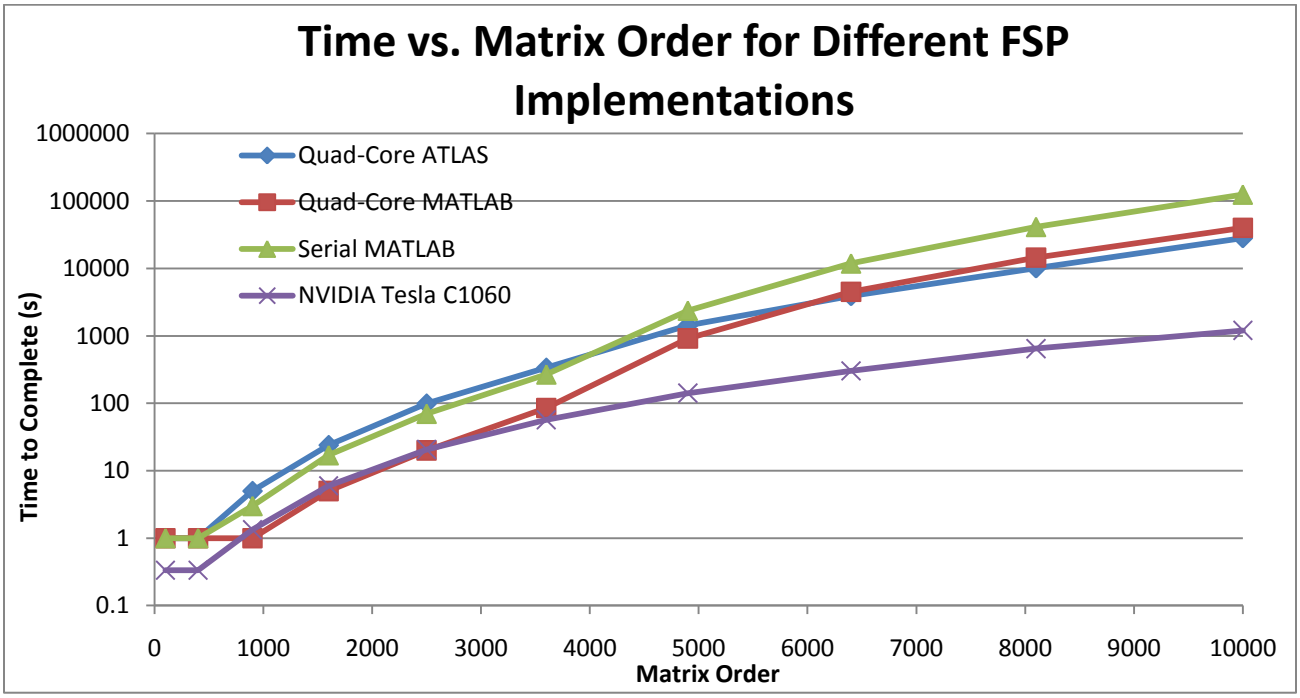


Figure 13 – Wall clock computation time demonstrating GPGPU FSP

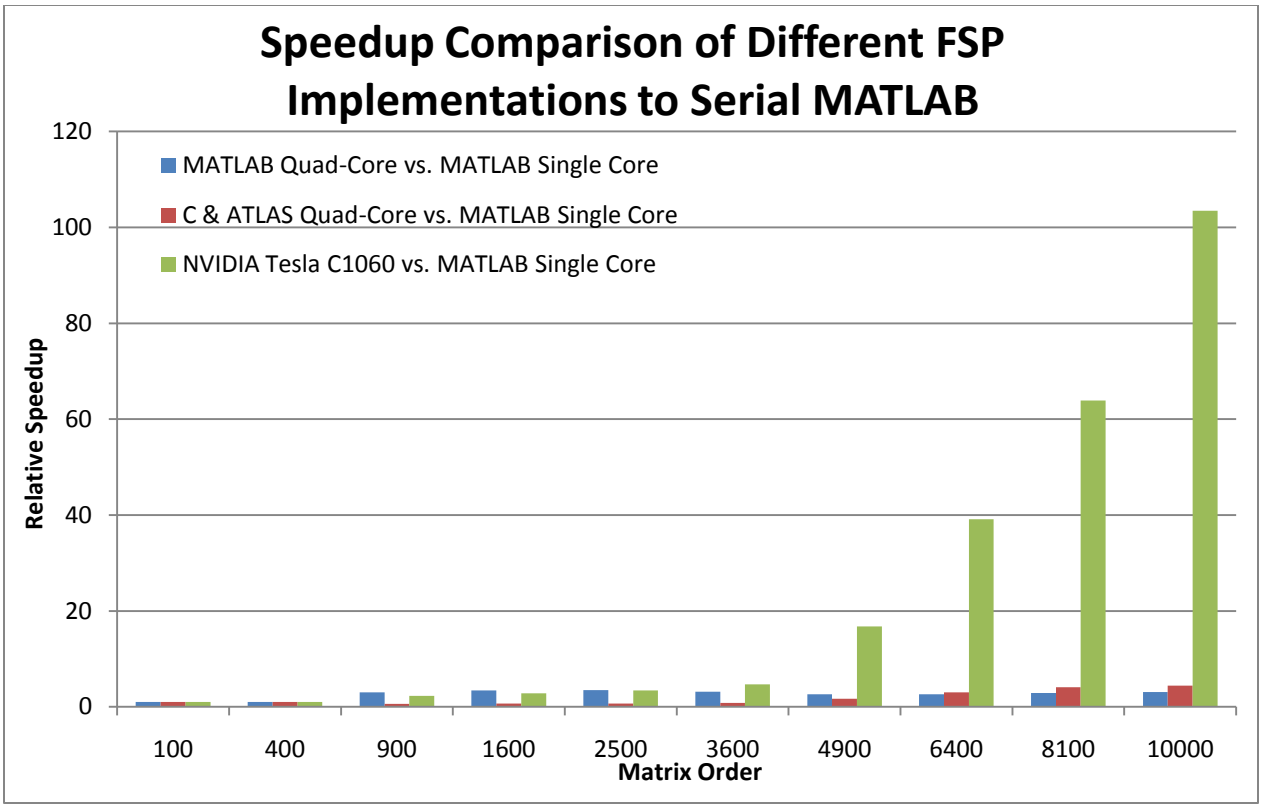


Figure 14 – Speedup of GPGPU FSP

As can be seen from the previous figures, the NVIDIA Tesla implementation of FSP performed well. Not only did it perform roughly the same as the MATLAB quad-core implementation for small problem sizes, but achieved a speedup of over 100x against the MATLAB single-core implementation for a matrix of order 10,000. It also completed the test suite in 40 minutes. The second fastest implementation, our C implementation using ATLAS, completed the test suite in a little over 12 hours, almost 20 times slower. What this means in terms of FSP performance is that large systems that would have previously take weeks to simulate can now be simulated in a matter of hours. The actual application of the GPGPU implementation of FSP to several more complex modeling problems is discussed in greater detail in Chapter 5.

4.3 Gene Expression Model with GPGPU FSP

In this section, the implementation of the transcription-translation model discussed previously will be implemented in our GPGPU FSP algorithm in order to demonstrate a problem of significant size. For review, a gene exists that transcribes mRNA at a rate k_m. This rate constant describes the exponentially distributed rate at which the gene transcribes mRNA. Once mRNA exists in the system, it can translate protein at a rate k_p. Again, this rate constant simply describes the exponential distribution relating how often a ribosome will translate a protein from the mRNA. Once in the system, both mRNA and protein will decay at rates γ_m and γ_p respectively. A schematic of this network can be seen below in Figure 15.

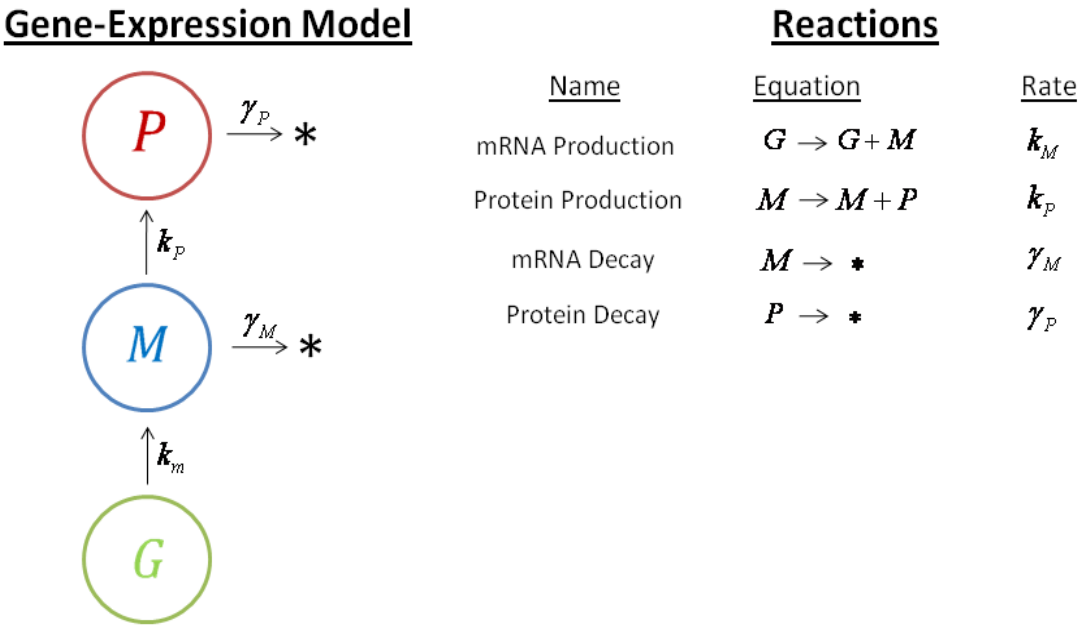


Figure 15 – Gene expression model

Following are the results of the full gene expression model. The model was implemented to run via our GPGPU FSP implementation, which itself ran on an NVIDIA Tesla C1060. The source code used to generate the approximate infinitesimal generator matrix is shown below in

Figure 16. The full source code used for this model can be viewed in its entirety in the appendix of this document.

```
__host__ double *addStates(double km, double kp, double g1, double g2, int size)
{
  int i,j,num,num1,num2;

  double *A = (double *)calloc(size*size,sizeof(double));

  for(i=0;i<size;i++){
    for(j=0;j<size;j++){
      if(i == j){
       *(A+(j*size+i)) = -km - (double)*(m1s+i)*kp - (double)*(p1s+i)*g1 -
(double)*(m1s+i)*g2;
      }else{
       num = abs(*(p1s+i) - *(p1s+j)) + abs(*(m1s+i) - *(m1s+j));
       if(num != 1){
         *(A+(j*size+i)) = 0.0;
       }else if(num == 1){
         num1 = *(p1s+i) - *(p1s+j);
         num2 = *(m1s+i) - *(m1s+j);
         if(num1 != 0){
           if(num1 > 0){
             *(A+(j*size+i)) = (double)*(p1s+i)*g1;
           }else{
             *(A+(j*size+i)) = (double)*(m1s+i)*kp;
           }
         }else if(num2 != 0){
           if(num2 > 0){
             *(A+(j*size+i)) = (double)*(m1s+i)*g2;
           }else{
             *(A+(j*size+i)) = km;
           }
         }
       }
      }
    }
  }
  return A;
}
__host__ void growStates(int &mRNA, int &protein, int &size)
{
  int i;
  mRNA++;
  protein++;
  for(i=0;i<protein;i++){
    size++;

    m1s = (int*)realloc(m1s,size*sizeof(int));
    *(m1s+(size-1)) = (double)mRNA;

    p1s = (int*)realloc(p1s,size*sizeof(int));
    *(p1s+(size-1)) = (double)i;
  }

  for(i=0;i<=mRNA;i++){
    size++;
    m1s = (int*)realloc(m1s,size*sizeof(int));
    *(m1s+(size-1)) = (double)i;

    p1s = (int*)realloc(p1s,size*sizeof(int));
    *(p1s+(size-1)) = (double)protein;
  }
}
```

Figure 16 – C code example for building an infinitesimal generator matrix

The results from the above model are as follows. Shown below in Figure 17 is the probability distribution of both protein and mRNA. The test was done with k_m= 5 and k_p= 5, and with both decay rates, γ_m and γ_p, left at 1.0

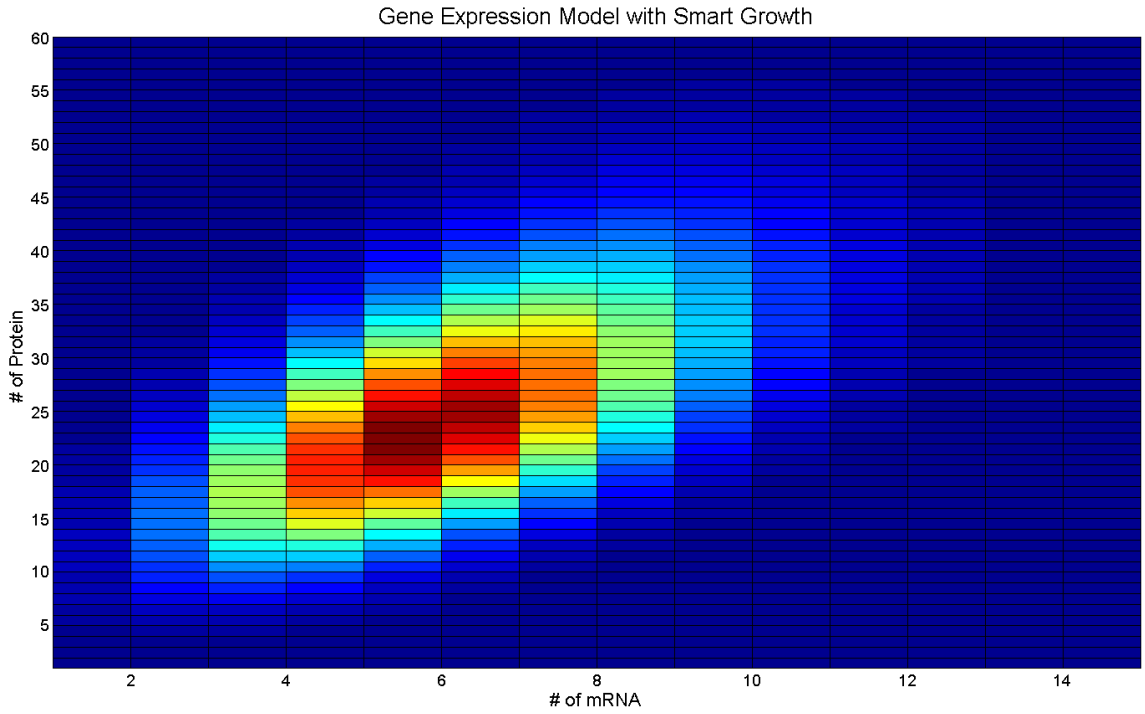


Figure 17 – Results of a large gene expression system modeled with GPGPU FSP

Figure 17 shows the probability distribution of both mRNA and protein in the system 3 hours after the system was initialized with none of either molecule. As expected, we observe a mean mRNA population of 5, and a mean protein population of 25. It is worth noting, however, that the algorithm required 10201 states to achieve an accuracy of 10^{-6}. This is nearly the maximum number of states that can be handled by a single NVIDIA Tesla C1060, which is not promising at all if larger systems are to be simulated. Systems that incorporate multiple populations can quickly require a large number of states, as the dimensionality of the state-space is proportional to the number of populations present in the system. For example, if a system only contained one species then the number of states would equal the population of the species. However, if another species is added to the system and has a population that grows proportionally to the first species, then the number of states would equal the square of the

population of either species. Due to this fact, it is worth examining state-saving measures that can be implemented in a model.

4.4 State-Saving Measures in Simulating Gene Expression

From a modeler's perspective, the intermediary step of including mRNA in the model may or may not be included. Very similar results should still be ascertained by simplifying the model such that the gene directly produces protein at rate $k_m * k_p$, and having the protein decay at rate $\gamma_m * \gamma_p$. Both implementations of this model are presented later in this section. A schematic of the simplified gene expression model can be seen below in Figure 18.

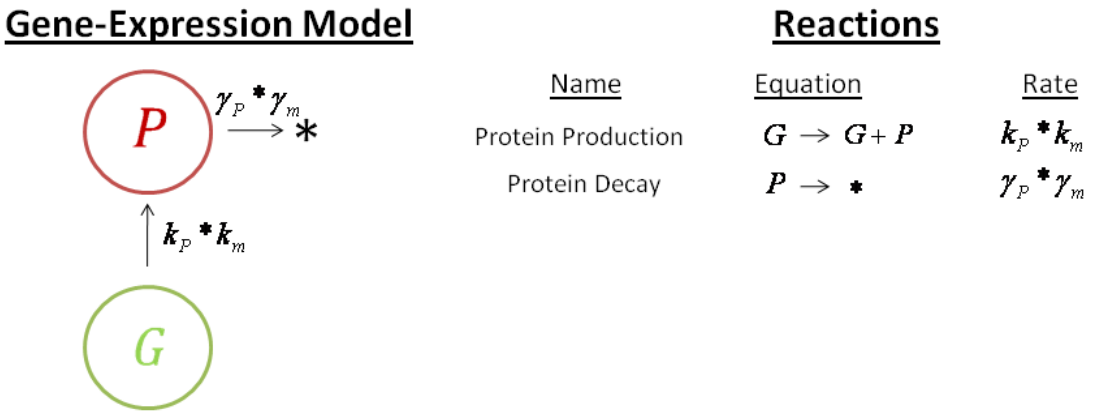


Figure 18 – Simplified gene expression model

As observed in the previous section, the state-space required to simulate systems with multiple species can become large very quickly. However, the number of states required to simulate the system can be reduced by making efficient use of the available state-space. For example, from examining Figure 17 it is clear that if mRNA has a mean population of 5 then it does not make a lot of sense to incorporate states that have mRNA populations that are much greater than 5. Another observation that can be made from Figure 17 is that the probability distribution of the system is fairly smooth and continuous. From these observations, it becomes possible to add states in a more educated manner than simply growing the state-space as a

square. Rather, only states adjacent to states that have a computed high probability are added to the state-space. By designing the algorithm to hunt for the states with a higher probability, the target accuracy can be reached more quickly and efficiently. Figure 19 and Figure 20 demonstrate this.

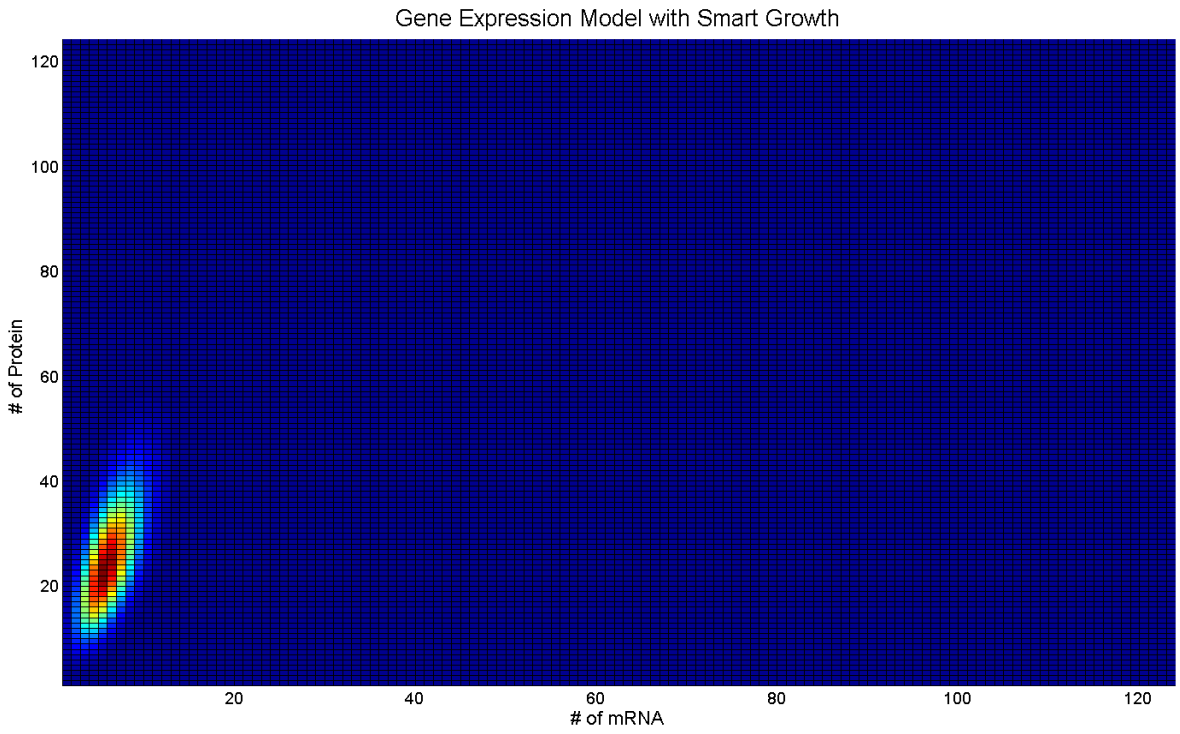


Figure 19 – Solution to gene expression system incorporating smart state growth

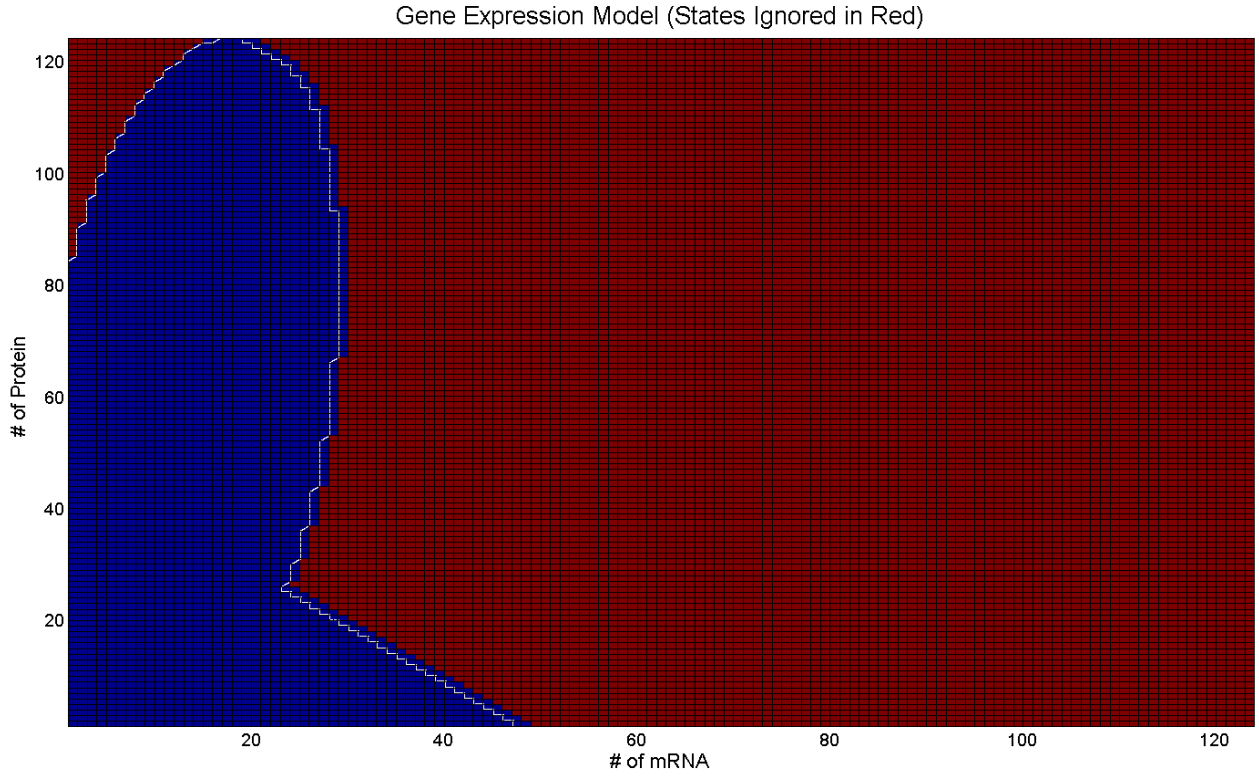


Figure 20 – States used for solution (blue) of the gene expression system with smart growth

Figure 19 shows the results of the same model being simulated as in Figure 17, but with the addition of the smart state-space growth technique proposed previously. As can be seen from observation Figure 17 and Figure 19 are nearly identical, indicating that no information was lost as a result of trimming the state-space. The mean and variance of the protein's distribution is 25 and 88, respectively for Figure 17. Similarly, the protein from Figure 19 has a mean and variance of 25 and 88 respectively. A major difference that is not readily apparent by comparing Figure 17 and Figure 19, is that the latter required only 3343 states to generate and still achieve a target error of 10^{-6}. Figure 20 makes this more apparent, as only the blue area was used in the computation. It is also worth noting that since the algorithm that added states intelligently required fewer states to solve the system, it took approximately 1 hour to get a result. The naïve algorithm, however, took approximately 6 hours to solve the system.

Another technique that can be used to reduce the state-space of a model is to simplify the model. As discussed briefly before, if the modeler is only after the statistical mean of protein, then mRNA can be omitted from the model. By modifying the model such that a gene directly produces protein at a rate of $k_m * k_p$, and the protein decays at a rate of $\gamma_m * \gamma_p$, a similar distribution of protein can be obtained.

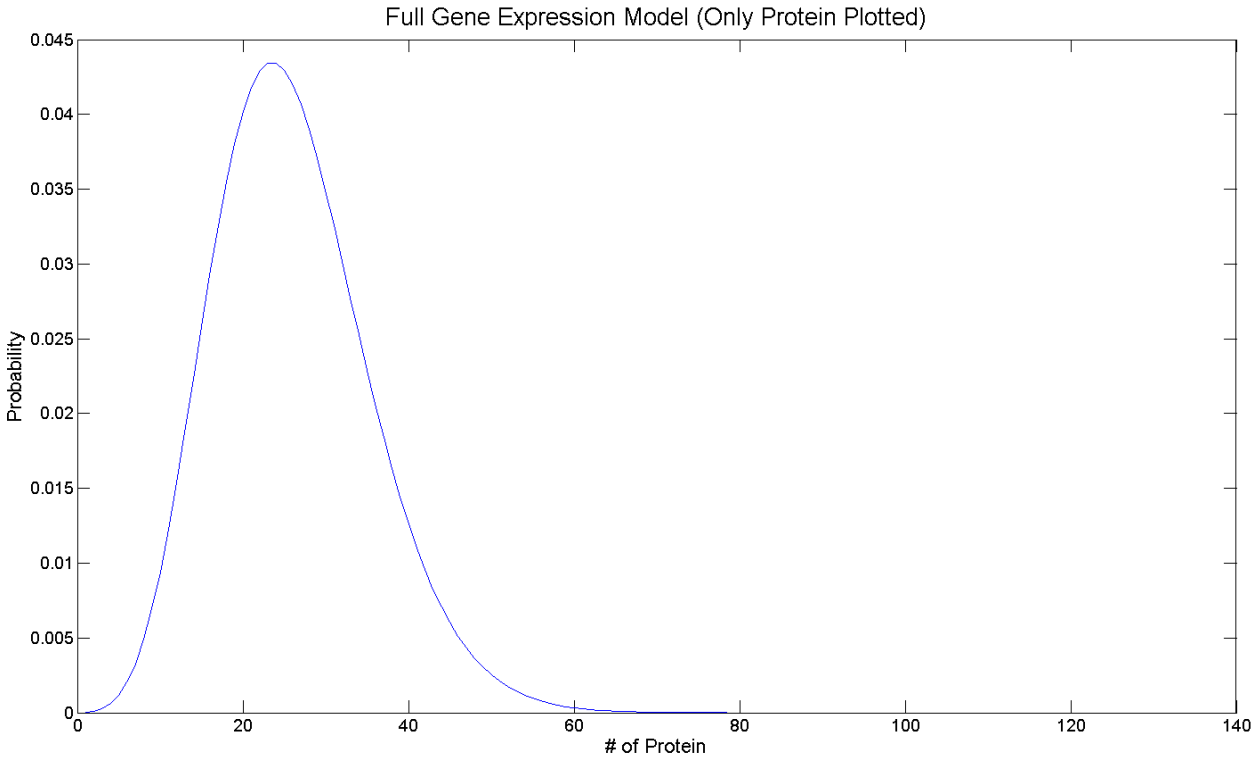


Figure 21 – Probability distribution of protein from full gene expression model

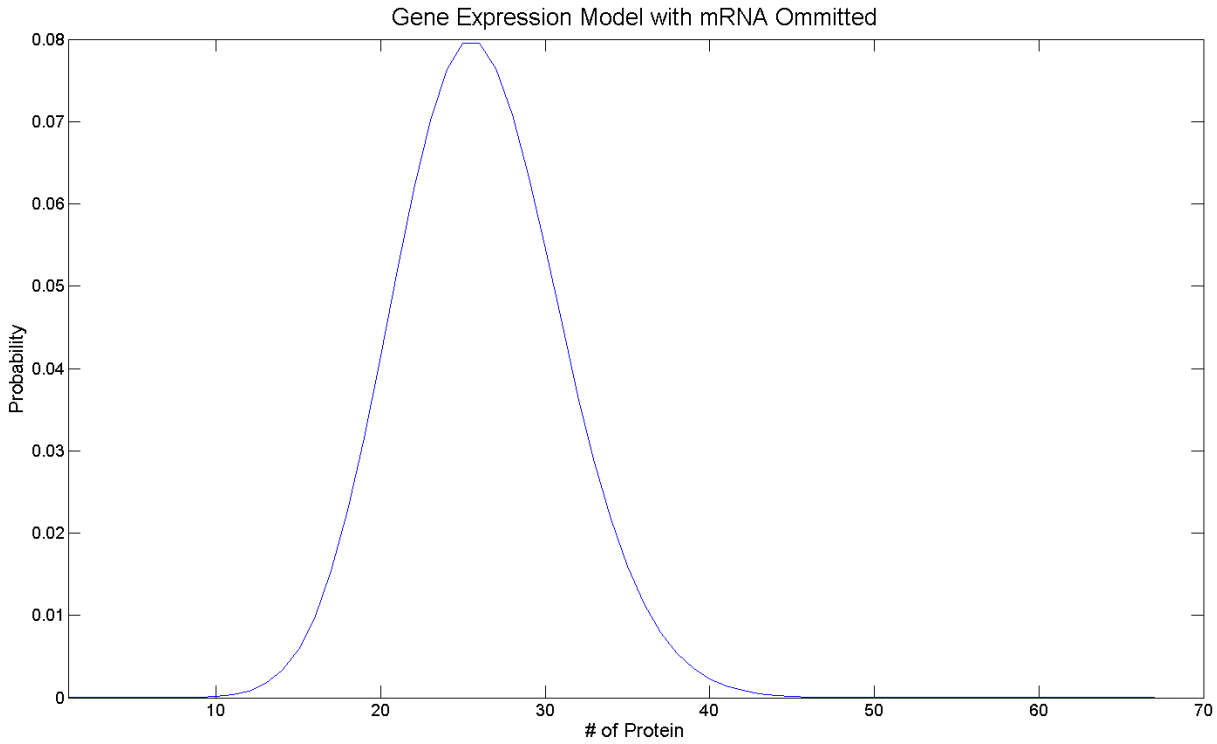


Figure 22 – Probability distribution of protein from simplified gene expression model

Figure 21 and Figure 22 show the protein distribution for the system described previously. Figure 21 shows the distribution of only protein from the model that also incorporated mRNA. This was done by simply summing the probabilities for the entire range of mRNA population levels, which creates a point representing the probability of a single protein population level. Figure 22 shows the distribution of the protein population from the simplified model. From observation, both figures look nearly identical and both produce a mean of 25.

It is important to note that by making more efficient use of the available state-space models will not only simulate faster, but can be made larger and more complex. Many systems that exist in the natural world often do involve multiple species or complex feedback mechanisms. Some examples of such systems which incorporate these simplifications follow in the following chapter.

4.5 Sensitivity to the Final Time Parameter

Another aspect to the performance of the FSP algorithm is the final time (t_f) parameter. A large t_f value will increase the norm of the infinitesimal generator matrix, and require that the algorithm make more loops through the scaling and squaring step of exponentiating the matrix. For this test, the infinitesimal generator matrix was held at a constant order of 10,000 and only the t_f parameter was swept. The results are shown below in Figure 23.

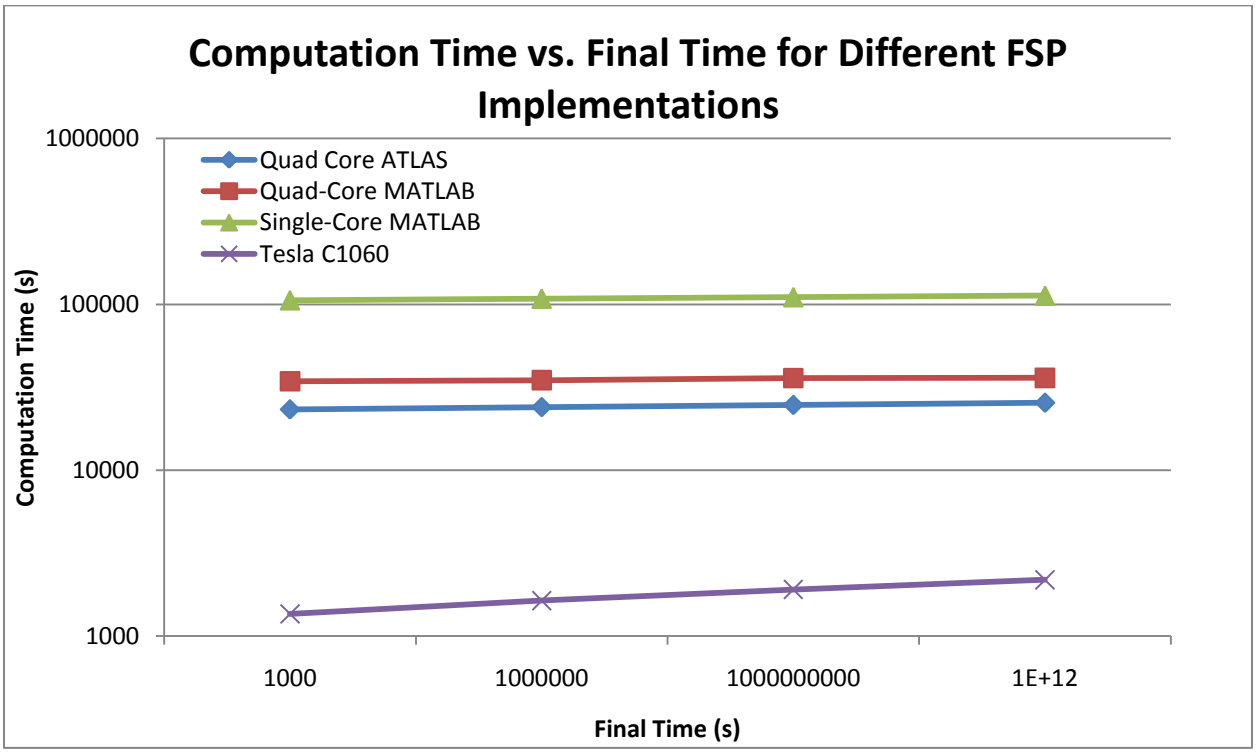


Figure 23 – Wall clock computation time demonstrating the final time parameter

The results show that while the Tesla GPGPU is still the fastest, the other implementations are able to handle the increased t_f parameter slightly better as a reduction in speedup is observed. In the case where t_f was equal to 1000, a speedup of almost 100x was observed. When t_f was increased all the way to 10^{12} seconds, a speedup of only 60x was

observed. This indicates a potential weakness in our implementation that could be improved upon in future work.

Chapter 5 – Modeling Applications

This chapter provides a brief overview of some more complicated biological systems and provide information as to how they are relevant. The results of modeling these systems with GPGPU FSP are then discussed.

5.1 Modeling Biological Systems

This section will explain applications of some more complex gene expression systems, and demonstrate how they would be modeled. A biological system is any group of molecules that can interact within the system. These systems can be very simple, as in the case of simple gene expression discussed previously. They can also scale into much larger, more complicated systems, like the nervous system in the human body. For the purposes of this work the focus will be more on gene expression systems, as they are more common in the field of systems biology.

Systems involving gene expression often involve complex regulatory mechanisms that allow a cell to control which proteins are being expressed within the cell. These systems, called gene regulatory networks, are essential for the survival of cells. Without the ability to regulate protein production, cells would not be able to adapt to their environment, locate sources of food, or even move. [22]

Gene regulatory networks were first discovered by Jacques Monod, when he discovered that protein responsible for the metabolism of lactose in E. Coli are expressed only in the presence of lactose and the absence of glucose. For the discovery of this system, called the lac operon, Jacques Monod was awarded the Nobel Prize in 1965. The lac operon's operation can be thought of like a thermostat. If there is no glucose to metabolize, but lactose is present, then E. Coli switches on production of proteins to metabolize this secondary food source; similar to a thermostat switching on the heat if a house is too cold. These systems can be very complicated,

however, by their very nature. For example, the human immunodeficiency virus (HIV) contains only 9 different genes that express protein. This is miniscule compared to the approximately 23,000 genes within the human genome. [23] However, the mechanisms through which the virus regulates its gene expression is still not fully understood.

Modeling gene regulatory networks is a powerful tool in the arsenal of scientists that attempt to understand them. Models tell scientists many things about a network that would be difficult to ascertain through pure experimentation. For example, Adam Arkin and his colleagues at the University of California, Berkeley have used biophysical modeling for many different systems. They used a stochastic model to demonstrate that stochastic fluctuations in protein expression can have an effect on which gene is expressed in phage λ infected E. Coli cells. [24] For another example, in 1998 researchers developed a mathematical model describing the process of cell-division in frog-eggs. [4] At the time, the details of this gene regulatory network had not been discovered. However, 5 years later the predictions made by this model were confirmed in the laboratory. [25]

5.2 Modeling Negative Autoregulation

Many systems that exist in nature, such as gene networks in E. Coli [3], make use of negative autoregulation to control their expression of protein. The way negative autoregulation works is that an increased presence of a given protein will actually slow down or shut off production of that protein. These mechanisms help a cell keep its protein levels in check, as excess protein will cause the cell to slow down its production until the proteins population has returned to normal. A characteristic of negative feedback that is often of interest to scientists is what is known as kinetics. The term kinetics describes how quickly the autoregulation reacts to the varying levels of protein within a cell. For example, a negative autoregulation motif that

switches off production of protein for a long time before producing more protein is said to have slow kinetics. Similarly, if the system reacts very quickly to the varying level of protein within the system, it is said to have fast kinetics. This concept can be thought of like the period of a pulse-width modulated signal.

To demonstrate this, a simple gene regulatory network was implemented that incorporates negative autoregulation. A schematic of the gene regulatory network is shown below in Figure 24.

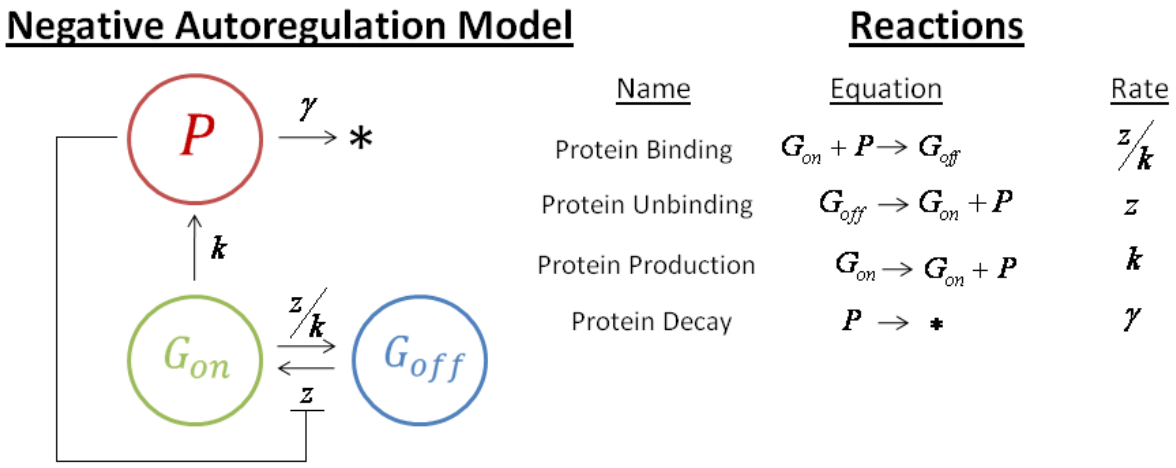


Figure 24 – Negative autoregulation model

In the system shown above, the gene which expresses protein has two states. In its "on" state, the gene is free to express protein according to the rate constant k. Once protein has entered the system, there exists an affinity for it to bind to the gene and switch it to its "off" state. While in this state, the gene is unable to produce more protein until the bound protein unbinds. The kinetics of the system, which is the affinity of protein to bind and unbind with the gene, is characterized by the rate constant z. As z increases, the protein are able to bind and unbind much quicker, creating a system with fast kinetics. Similarly, as z decreases the protein bind and unbind to and from the gene at a slower rate, creating a system with slow kinetics. As in the

previous system, once a protein is created it can also decay out of the system according to rate constant γ.

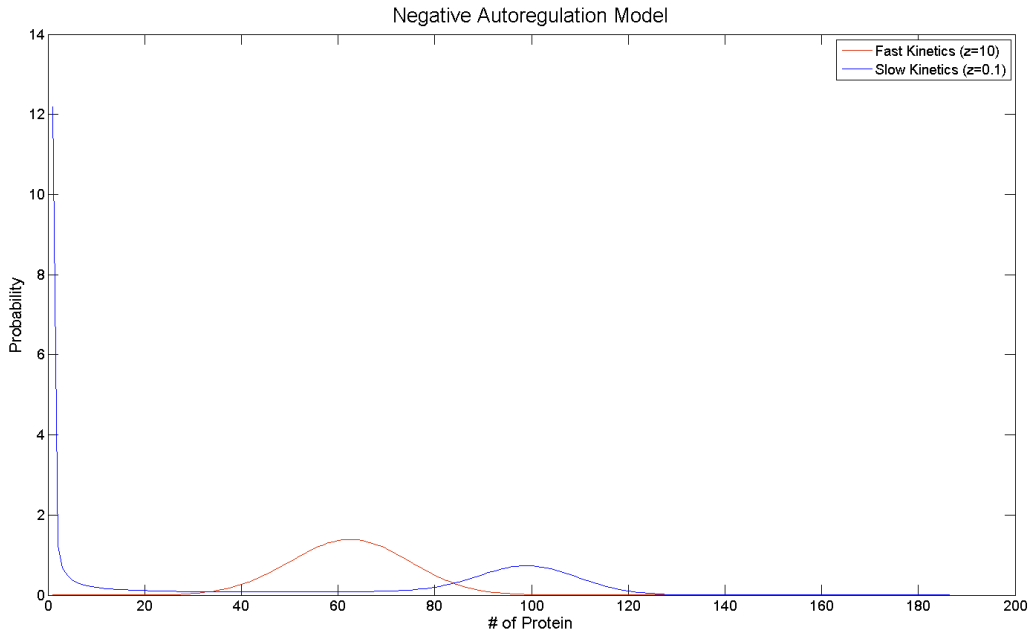


Figure 25 – Demonstration of kinetics in negative autoregulation

In Figure 25, the results of two different simulations are shown. Both simulations had rate constant k=100 and γ=1. The difference between the two, however, is the kinetics. The red line in Figure 25 shows the system with z = 10, creating a system with fast kinetics. This system had a mean protein level of 61.4, a variance in protein of 148.0, and required 297 states to achieve an accuracy of 10^{-6}. Since k was equal to 100, it would be natural to expect to see more of the probability trending near a protein population of 100. However, since the kinetics of the system were fast, the likelihood is that a protein would bind to the gene and switch it off before the protein population could build to 100. This system exemplifies how negative autoregulation can keep the protein population of a cell in check.

The blue line in Figure 25 shows the probability distribution of protein in the system with slow kinetics, characterized by setting z = 0.1. This system had a mean protein level of 52.1 a

variance in protein of 2127.5, and required 355 states to achieve an accuracy of 10^{-6}. This distribution clearly shows the two states that the gene can exist in, with a likelihood of there being close to 100 protein in the system when the gene is on and close to 0 protein in the system when the gene is off. This is because the slow kinetics of the system make it very difficult for the gene to change states, allowing the protein population plenty of time to adapt to the gene's respective state. Since the systems involved required a very small state-space, the results were obtained from MATLAB. The source code used is shown below.

```
while tempError > error
    BigA = zeros(length(p1s),length(p1s));
    for i=1:length(p1s)
        for j=1:length(p1s)
            if(i == j)
                BigA(j,i) = -g1s(i)*k1 - p1s(i)*g1 - p1s(i)*g1s(i)*k1r - (abs(g1s(i)-1)) ↙
*k1f;
            else
                num = abs(p1s(i)-p1s(j))+abs(g1s(i)-g1s(j));
                if(num ~= 1)
                    BigA(j,i) = 0;
                elseif(num == 1)
                    num1 = p1s(i)-p1s(j);
                    num4 = g1s(i)-g1s(j);
                    if(num1 ~= 0)
                        if(num1 > 0)
                            BigA(j,i) = p1s(i)*g1;
                        else
                            BigA(j,i) = g1s(i)*k1;
                        end
                    elseif(num4 ~= 0)
                        BigA(j,i) = p1s(i)*g1s(i)*k1r + (abs(g1s(i)-1))*k1f;
                    else
                        disp('This should never appear')
                    end
                end
            end
        end
    end
    Prob = zeros(length(p1s),1);
    Prob(1) = 1;
    %    BigA = BigA.';
    tempVec = expm(BigA*tf)*Prob;
    tempError = 1 - sum(tempVec);
```

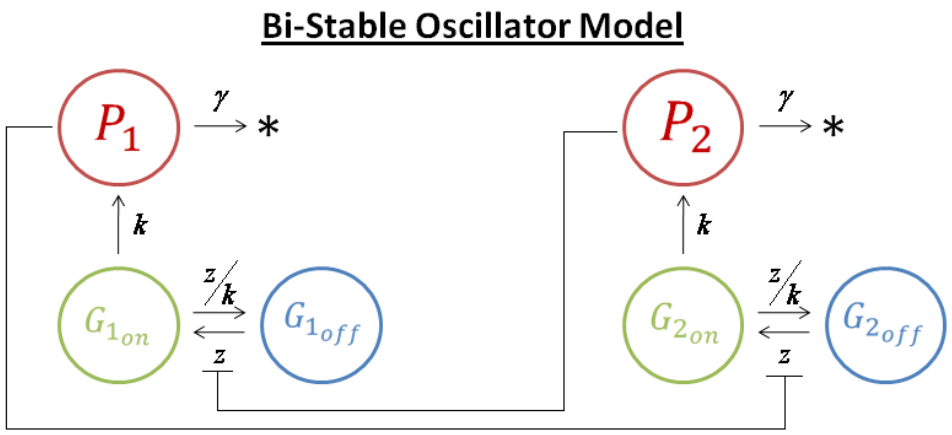
Figure 26 – MATLAB code for negative autoregulation model

As is easily observed from Figure 25, the kinetics of a system can drastically change the resulting probability distribution of protein within the system. It is important to recall that the only difference between the two systems shown in Figure 25 was the kinetics. This change, however, resulted in drastically different mean and variance levels, as well as the number of states required to achieve the same amount of accuracy.

5.3 The Bi-stable Oscillator

The bi-stable oscillator, or "toggle-switch", is a gene regulatory network in which two different protein are expressed, but regulate the production of one another. This action creates a system in which the two different protein populations will oscillate out of phase with each other. These systems can create "switches" which help cells make decisions regarding different functions. A collaborative group between the Stanford University School of Medicine and Rutgers University created a mathematical model, which they verified in the lab, describing a bi-stable oscillatory network in frog eggs that helps act as a trigger for cell mitosis. [26] Modeling and understanding these systems is crucial in helping to explain the decision making process of a cell.

A bi-stable oscillatory system is now presented, again demonstrating both slow and fast kinetics. The schematic of the system is shown below in Figure 27.



Reactions

Name	Equation	Rate	Name	Equation	Rate
Protein Binding	$G_{1_{on}} + P_2 \rightarrow G_{1_{off}}$	z/k	Protein Binding	$G_{2_{on}} + P_1 \rightarrow G_{2_{off}}$	z/k
Protein Unbinding	$G_{1_{off}} \rightarrow G_{1_{on}} + P_2$	z	Protein Unbinding	$G_{2_{off}} \rightarrow G_{2_{on}} + P_1$	z
Protein Production	$G_{1_{on}} \rightarrow G_{1_{on}} + P_1$	k	Protein Production	$G_{2_{on}} \rightarrow G_{2_{on}} + P_2$	k
Protein Decay	$P_1 \rightarrow *$	γ	Protein Decay	$P_2 \rightarrow *$	γ

Figure 27 – Bi-stable oscillator model

In the above system, there are two separate genes. Each gene directly produces protein at the same rate, characterized by rate constant *k* which is equal to 10. As before, both proteins can decay out of the system according to the same rate constant γ which is set equal to 1. In similar fashion to the negative autoregulation model, this model also incorporates an element of negative feedback as described above. Rather than the proteins regulating their own production, they instead regulate production of the other gene's protein. This negative feedback can also be characterized with the kinetics parameter *z*. It is worth noting that the bi-stable oscillator has been simulated with FSP by Munsky and Khammash. However, as their implementation of FSP was only in MATLAB, they were not able to model the kinetics of the system by incorporating a two-state gene. The kinetics were instead approximated by changing the rate constant at which

each gene expressed protein according to Hill Equations, which was done as a state-saving measure. Rather than having a gene with an explicit "on" or "off" state, Hill Equations vary the rate constant at which protein is produced. The rate constant can vary as the inverse of the population of the repressing protein raised to a specific power. Unlike the state-saving measures proposed previously, approximating the kinetics through Hill Equations is not as accurate as modeling the gene as a two-state entity. [27]

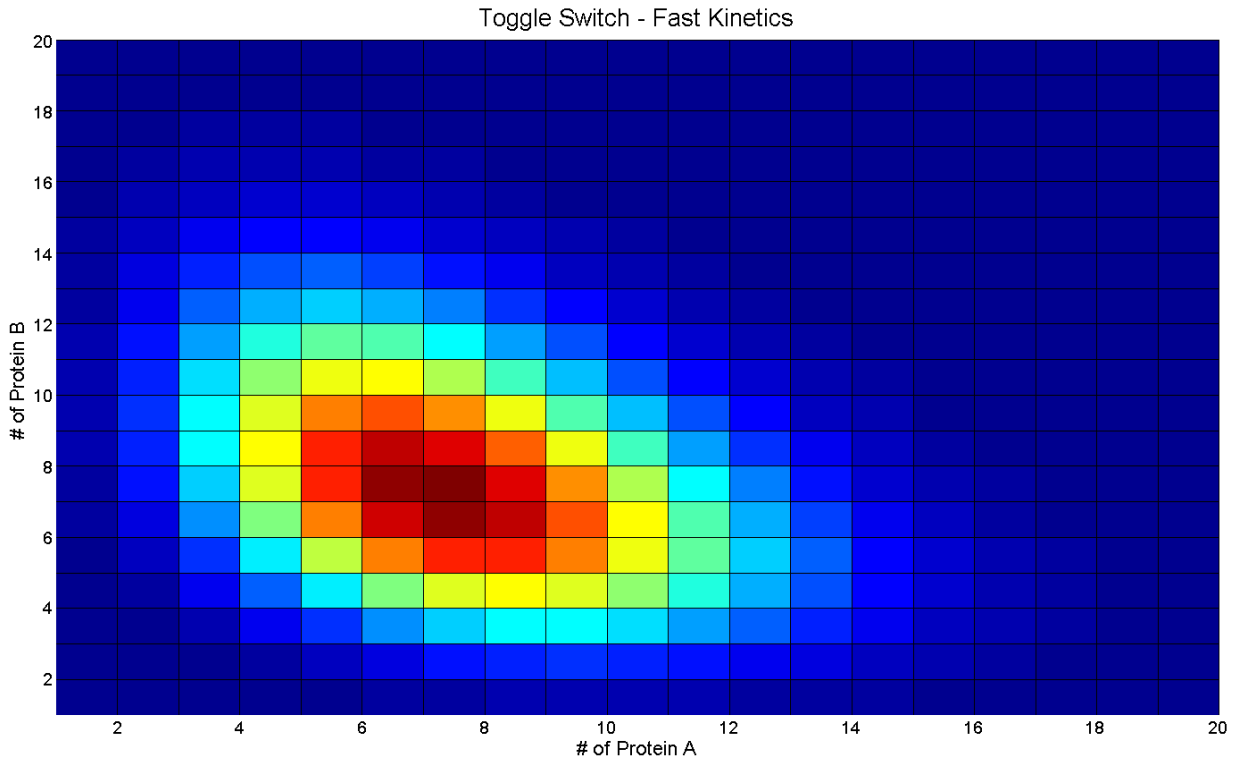


Figure 28 – Bi-stable oscillator demonstrating fast kinetics

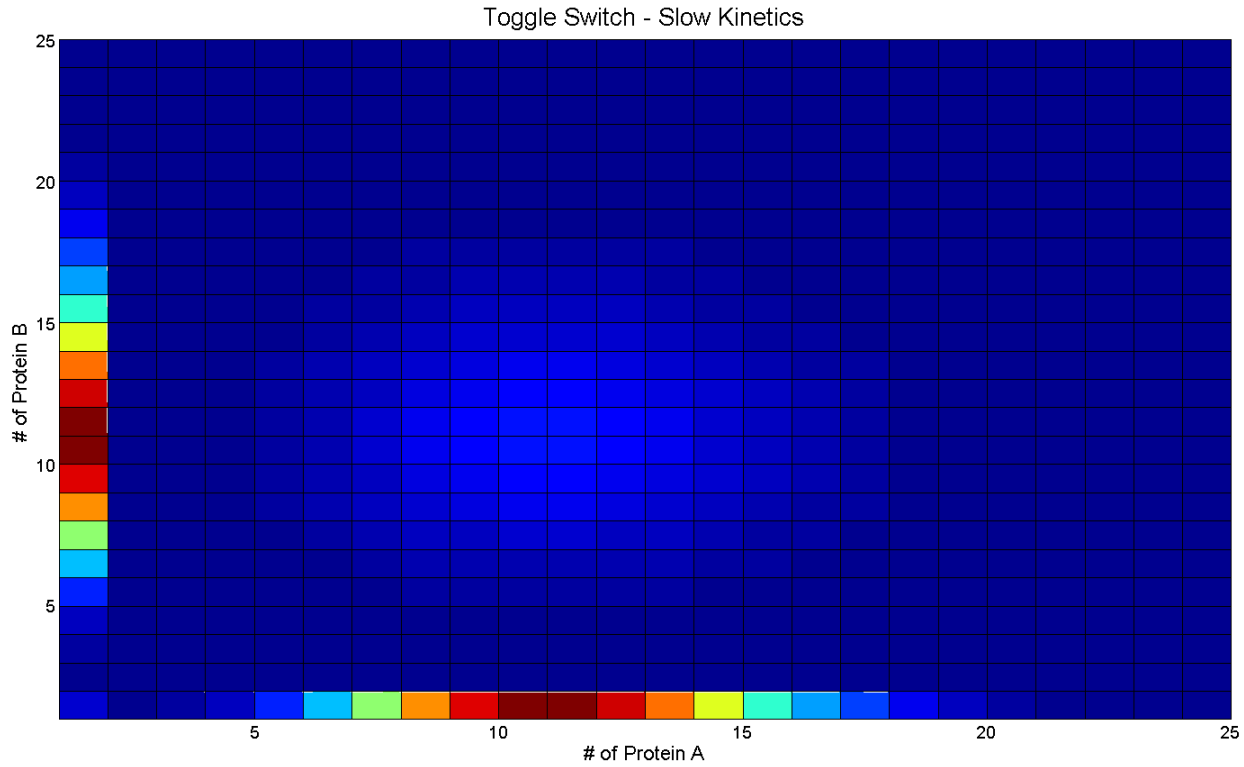


Figure 29 – Bi-stable oscillator demonstrating slow kinetics

The results of the simulations discussed above are shown in Figure 28 and Figure 29. Figure 28 shows the system with $z = 10$, characterizing fast kinetics. Both protein populations had a mean of 6.32 and a variance of 8.8. This system required 9985 states to achieve a target error of 10^{-9}, and required approximately 2 hours to simulate with our GPU implementation of FSP. As would be expected from examining the previous system incorporating fast kinetics, the protein populations are down-regulated below where we would expect them to be. What makes this system interesting, however, is that rather than each protein being a regulator of its own production, the protein population levels are kept in check by the opposing protein.

Figure 29 shows the same system, but this time with $z = 0.01$ which characterizes slow kinetics. Both protein populations had a mean of 6.65 and a variance of 28.6. The system required 7887 states to achieve a target error of 10^{-9}, and required approximately 1.5 hours to

simulate with our GPU implementation of FSP. This system more closely resembles the system in frog eggs described by [26], in that the system would more clearly oscillate between protein. We see that it is very unlikely to get a high concentration of one protein in the presence of another, yet still happens occasionally as the system oscillates between the two genes cycling on and off. The source code used to generate these results can be viewed in the appendix of this document.

5.3 Approximated Repressilator Network

An even more complex gene regulatory network is called the repressilator. This system incorporates three different genes that oscillate in a cyclic fashion, such that only one of the three respective proteins will be dominant in the system at a time. [10] Michael Elowitz and Stanislas Leibler at Princeton University successfully designed a repressilator system in E. Coli. What made their system interesting was that the oscillatory period was longer than the cell's division time. This means that the oscillator had to be passed to newer generations of cells, demonstrating how cell networks can communicate across generations. While repressilator systems have yet to be fully understood, studying these systems could lead to insights as to how networks of cells can work together and communicate in nature.

To demonstrate this, a repressilator network was implemented. A schematic of the system is shown below in Figure 30.

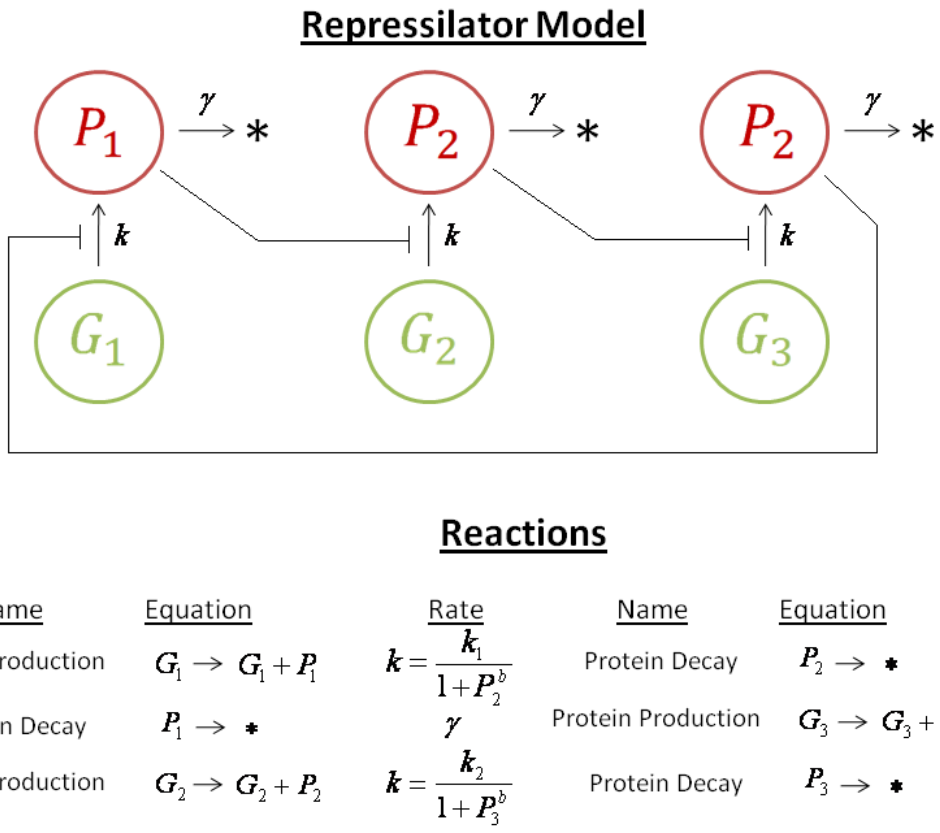


Figure 30 – Repressilator model

As this system truly pushes the limits of what can be done with a single graphics card, the kinetics of the system were approximated through Hill Equations. The speed of the system's kinetics is characterized by the parameter *b*, which is inversely proportional to the speed of the kinetics. Each gene produces protein at a rate characterized by the parameter *k*, which was set to 15 for each gene, but the rate will vary according to the Hill Equation. Similarly to the previous models, protein in the system all decay according to the rate constant γ, which is equal to 1. The results of these simulations are shown in Figure 31 and Figure 32.

Repressilator – b=1, k=15

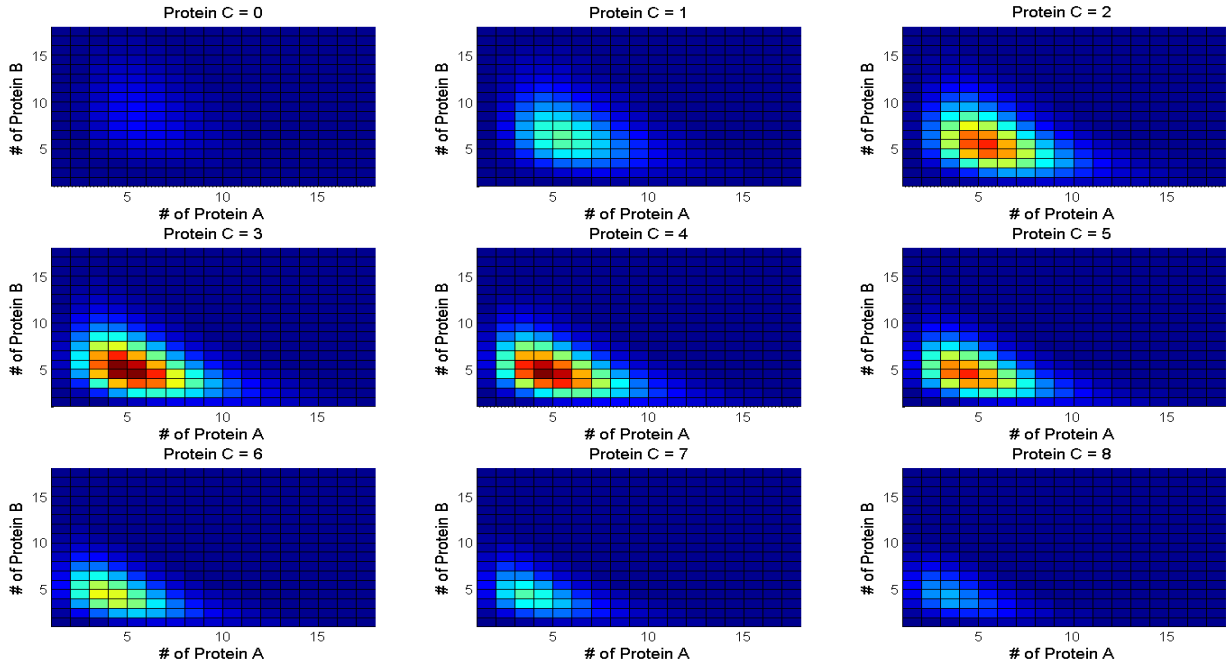


Figure 31 – Repressilator model demonstrating fast kinetics

Repressilator – b=2, k=15

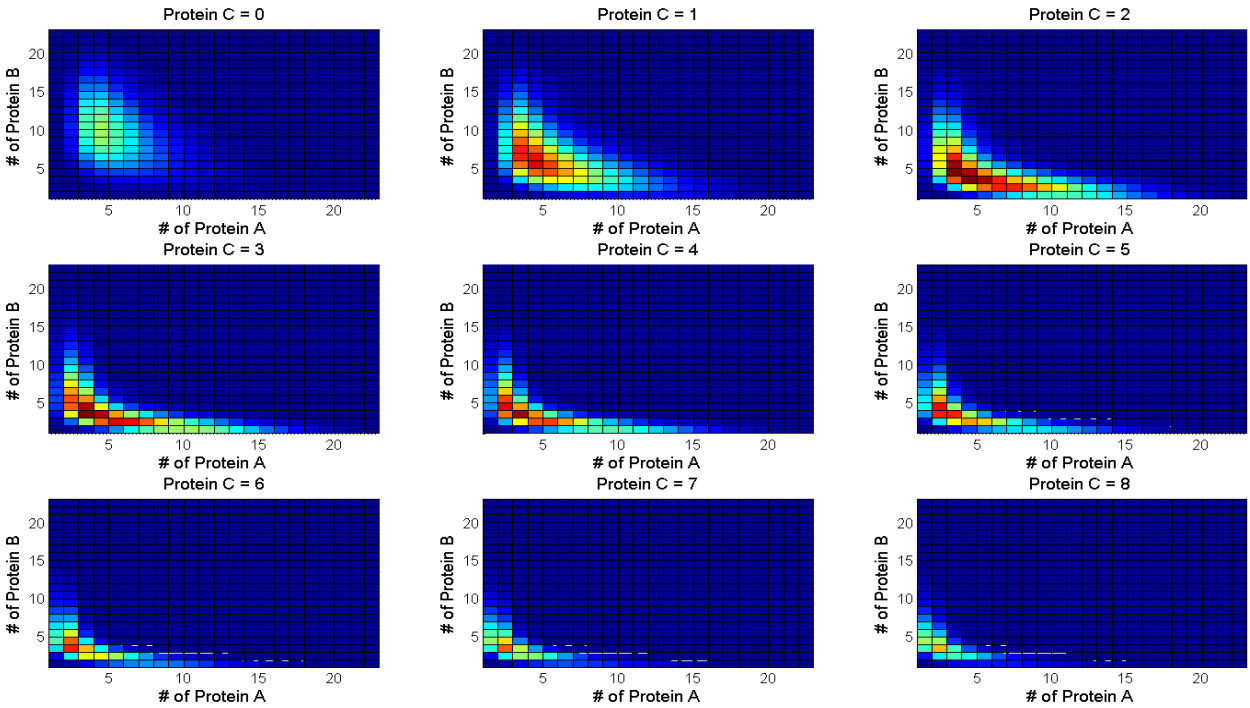


Figure 32 – Repressilator model demonstrating slow kinetics

Figure 31 shows the system with *b*=1, characterizing fast kinetics. Each of the three protein populations had a mean of 4 and a population variance of 6.6. The simulation required 9876 states to achieve a target error of 10^{-3}, and took approximately 1 hour and 40 minutes to simulate with our GPU implementation of FSP. As would be expected, we see the highest probabilities in the distribution clouded around the area where each protein population is around 4. The kinetics of the system are too quick to allow any one population to build to a steady state value with a high degree of likelihood.

Figure 32 shows the system with *b*=2, characterizing slower kinetics. As this system would have a slower period of oscillation, we observe a higher likelihood of larger protein populations. Each protein population had a mean of 3.9 and a population variance of 12.4. The slight decrease in the mean is likely due to the fact that a target error of only 10^{-2} was achieved with 9876 states. A greater target error should be achieved in order to ascertain more accurate statistical measurements of the population. The source code used for both of these models can be viewed in the appendix of this document.

Despite the inability of our GPU implementation of FSP to achieve a higher target error for the repressilator system, it demonstrates the ability of the FSP algorithm to model larger systems than ever before. It is important to note that this algorithm is still in its infancy, and further improvements could still be made to allow even larger systems to be simulated. This will be discussed in more detail in chapter 6 of this document.

Chapter 6

6.1 Conclusions

Gene regulatory networks are the basis through which cells perform many of the functions necessary for them to survive. These tasks can range from finding food and moving after it, to the decision as to whether or not the cell should divide. Stochastic analysis of these systems can lead to a deeper understanding of their inner workings, providing insights that could not have been gained through pure deterministic modeling. As such, algorithms that are able to accurately simulate and capture the stochastic components of these systems are crucial. Accurate simulation of these systems can validate observations made in the lab, predict systematic behaviors, and even aid in the development of new gene therapy techniques or drugs.

Finite-state projection (FSP) provides scientists a mechanism for better characterizing these slowly reacting systems. The FSP algorithm also provides a guarantee for error in the simulation, as it directly solves or approximates the chemical master equation. The future of modeling biological systems with FSP clearly lies in General Purpose Graphics Processing Units (GPGPUs). The results achieved by applying GPGPUs to FSP easily distinguish themselves from the other platforms tested, as the GPGPU implementation achieved a max speedup of over 100x that of the original implementation. This suggests that with more work, systems of increasing complexity can be modeled accurately and efficiently.

Previous work with the FSP algorithm has demonstrated its potential, but has also shown that the algorithm is limited to solving only smaller problems. Due to the computational cost of the matrix exponential function, FSP's original implementation in Mathwork's MATLAB proved an inefficient means for solving larger systems. Systems such as the bi-stable oscillator or repressilator would have taken weeks to simulate in MATLAB, even using modern multi-core

processing. By implementing the algorithm to run on a general-purpose graphics processing unit (GPGPU), these simulation times have been reduced from several weeks to a few hours. GPGPUs have shown extraordinary promise in the realm of supercomputing, and their relative size and cost brings them within reach of average researchers in industry or academia. This work demonstrates that by implementing FSP on a GPGPU, a relative speedup of over 100x can be observed over FSP running in MATLAB when solving a large problem. Furthermore, a speedup of over 75x was observed across the entire test suite. This demonstrates that GPGPUs are a clear solution for the efficient implementation of FSP.

6.2 Future Work

Further enhancement to the work done here is possible along several avenues. The algorithm was limited by hardware to a state-space size of 10,000. Although this was due to memory constraints of the system, ways of adding more states could be explored. Hard disks, which can hold massive amounts of data relative to the onboard random-access memory of a computer, could be explored in terms of simulating systems with more than 10,000 states. Methods could also be explored in regard to handling a large t_f parameter. Additionally, the algorithm currently is designed to only run on one graphics card. New technologies like NVIDIA's Scalable Link Interface or ATI's CrossFire allow multiple GPUs to work in tandem on a single problem. New libraries, like CULA, are being developed that allow relatively simplistic deployment of algorithms that take advantage of these new technologies. Additions like these hold potential for massive systems to be simulated efficiently with FSP.

Bibliography

[1] D. Pederson, R. Newton, A. Sangiovanni-Vincentelli, C. Wayne, J. M. Rabaey T. Quarles. (2001, June) The Spice Page. [Online]. http://bwrc.eecs.berkeley.edu/classes/icbook/spice/

[2] MathWorks. (2011, March) Matlab Homepage. [Online]. http://www.mathworks.com/products/matlab/

[3] M.S. Allen, J.M. McCollum, R.D. Dar, J.R. Wilgus, G.S. Saylor, N.F. Samatova, C.D. Cox, M.L. Simpson D.W. Austin, "Gene Network Shaping of Inherent Noise Spectra," *Nature*, vol. 439, no. 7076, 2006.

[4] C.J. Tyson, B. Novak, J.J Tyson G. Marlovits, "Modeling M-phase control in Xenopus oocyte extracts: the surveillance mechanism for unreplicated DNA ," *Biophysical Chemistry*, vol. 72, no. 1-2, 1998.

[5] A. Arkin H.H. McAdams, "It's a noisy business! Genetic regulation at the nanomolar scale ," *Trends in Genetics*, vol. 15, no. 2, 1999.

[6] P.N. Devreotes P.A. Iglesias, "Navigating through models of chemotaxis," *Current Opinion in Cell Biology*, vol. 20, no. 1, 2008.

[7] J. Peccoud P.J.E. Goss, "Quantitative modeling of stochastic systems in molecular biology by using stochastic Petri nets," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 92, no. 12, 1998.

[8] M. Khammash B. Munsky, "The finite state projection algorithm for the solution of the chemical master equation ," *Journal of Chemical Physics*, vol. 124, no. 4, 2006.

[9] D.W. Smith, M.T. Madigan T.D. Brock, *Biology of Microorganisms, 4th Edition*. Englewood Cliffs: Prentice-Hall Inc., 1984.

[10] S. Leibler M.B. Elowitz, "A synthetic oscillatory network of transcriptional regulators," *Nature*, vol. 403, no. 6767, 2000.

[11] D.T. Gillespie, "Exact stochastic simulation of coupled chemical reactions," *Journal of Physical Chemistry*, vol. 81, no. 25, 1977.

[12] M. Hegland, F. Macnamara, R.B. Sidje K. Burrage, "A Krylov-based Finite State Projection algorithm for solving the chemical master equation arising in the discrete modelling of biological systems," , Raleigh, 2006.

[13] M. Khammash B. Munsky, "A multiple time interval finite state projection algorithm for the solution to the chemical master equation," *Journal of Computational Physics*, vol. 226, no. 1, 2007.

[14] M.J. Lawson, L. Petzold, M. Khammash B. Drawert, "The diffusive finite state projection algorithm for efficient simulation of the stochastic reaction-diffusion master equation ," *Journal of Chemical Physics*, vol. 132, no. 7, 2010.

[15] J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, R.C. Whaley L.S. Blackford, "An Updated Set of Basic Linear Algebra Subprograms (BLAS)," *ACM Transactions on Mathematical Software*, vol. 28, no. 2, 2002.

[16] C.V. Loan C. Moler, "Nineteen Dubious Ways to Compute the Exponential of a Matrix," *SIAM Review*, vol. 20, no. 4, 1978.

[17] C.V. Loan C. Moler, "Nineteen Dubious Ways to Compute the Exponential of a Matrix, Twenty-Five Years Later," *SIAM Review*, vol. 45, no. 1, 2003.

[18] A. Petitet, J.J. Dongarra R.C. Whaley, "Automated Empirical Optimization of Software and the ATLAS Project," *Parallel Computing*, vol. 27, no. 1-2, 2001.

[19] J. Demmel V. Volkov, "LU, QR and Cholesky Factorizations using Vector Capabilities of GPUs," Berkeley, 2008.

[20] D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A.E. Lefohn, T.J. Purcell J.D. Owens, "A Survey of General-Purpose Computation on Graphics Hardware," *Computer Graphics Forum*, vol. 26, no. 1, 2007.

[21] TOP500.org. (2011, March) TOP500 Supercomputing Sites. [Online]. http://www.top500.org/

[22] C.H. Huang, P.A. Iglesias, P.N. Devreotes Y. Xiong, "Cells navigate with a local-excitation, global-inhibition-biased excitable network," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 107, no. 40, 2010.

[23] Department of Energey's Biological and Environmental Research Information System (BERIS). (2011, March) Genome Programs of the U.S. Department of Energy. [Online]. http://www.ornl.gov/sci/techresources/Human_Genome/home.shtml

[24] J. Ross, H.H. McAdams A. Arkin, "Stochastic Kinetic Analysis of Developmental Pathway Bifurcation in Phage

[25] J. Moore, K. Chen, A.D. Lassaletta, C.S. Yi, J.J. Tyson, J.C. Sible W. Sha, "Hysteresis drives cell-cycle transitions in Xenopus laevis egg extracts ," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 100, no. 3, 2003.

[26] E.D. Sontag, J.E. Ferrel J.R. Pomerening, "Building a cell cycle oscillator: hysteresis and bistability in the activation of Cdc2," *Nature Cell Biology*, vol. 5, no. 4, 2003.

[27] M. Maurin, F. Rougier, X. Barbaut, L. Bourguignon, M. Ducher, P. Maire S. Goutelle, "The Hill equation: a review of its capabilities in pharmacological modelling," *Fundamental & Clinical Pharmacology*, vol. 22, no. 6, 2008.

Appendix A: MATLAB Source Code

testmodel2fast.m

```
%function [finaltime,tempVec,tempError] = testmodel2fast(km,kp,maxM,tf)
kp       = 3;
gammap   = 1;
km       = 3;
gammam   = 1;
tf       = 1000;
error    = 10^-6;
tempError = 1;
leap     = 0;
maxM = 2;

%build states
while tempError > error
    %clear BigA;
    BigA = zeros((maxM+1)^2,(maxM+1)^2);
    for rows=1:(maxM+1)^2
        BigA(rows,rows) = -km - floor((rows-1)/(maxM+1))*gammam -
floor((rows-1)/(maxM+1))*kp - mod(rows-1,maxM+1)*gammap;
        if rows-maxM-1 > 0
            BigA(rows-maxM-1,rows) = km;
        end
        if rows+maxM+1 <= (maxM+1)^2
            BigA(rows+maxM+1,rows) = floor((rows+maxM)/(maxM+1))*gammam;
        end
        if rows > 1
            BigA(rows,rows-1) = mod(rows-1,maxM+1)*gammap;
            if rows <= (maxM+1)^2
                if mod(rows-1,maxM+1) == 0
                    BigA(rows-1,rows) = 0;
                else
                    BigA(rows-1,rows) = floor((rows-2)/(maxM+1))*kp;
                end
            end
        end
    end
    tic
    Prob = [1; zeros((maxM+1)^2-1,1)];
    BigA = BigA.';
    tempVec = expm(BigA*tf)*Prob;
    tempError = 1 - sum(tempVec)
    if tempError <= error
        break;
    end
    maxM = maxM+1;
end
disp('preparing results...')
for i=1:length(tempVec)
    outMat(ceil(i/(sqrt(length(tempVec)))),mod(i-1,sqrt(length(tempVec)))+1)
= tempVec(i);
end
for i=1:length(outMat)
    pvec(i) = sum(outMat(:,i));
    mvec(i) = sum(outMat(i,:));
```

```
end
stem(pvec);
title('protein')
figure
stem(mvec)
title('mRNA')
```

geneExpModel.m

```
k1       = 25;
g1       = 1;

tf       = 10000;
error    = 10^-6;
tempError = 1;
leap     = 0;

%build states
maxM = 1;
while tempError > error
    for i=1:(maxM+1)
        p1s(i) = i-1;
    end
    BigA = zeros((maxM+1),(maxM+1));
    for i=1:((maxM+1))
        for j=1:((maxM+1))
            if(i == j)
                BigA(j,i) = -k1 - p1s(i)*g1;
            else
                num = p1s(i)-p1s(j);
                if(abs(num) ~= 1)
                    BigA(j,i) = 0;
                elseif(num > 0)
                    BigA(j,i) = p1s(i)*g1;
                else
                    BigA(j,i) = k1;
                end
            end
        end
    end
    Prob = zeros((maxM+1),1);
    Prob(1) = 1;
%    BigA = BigA.';
    tempVec = expm(BigA*tf)*Prob;
    tempError = 1 - sum(tempVec)
    if tempError <= error
        break;
    end
    maxM = maxM+floor(leap*tempError)+1
    if 2*(maxM+1) > 11000
        break;
    end
%     if maxM == 4
%         break;
%     end
end
```

testmodelErrorSweep.m

```
k        = 3;
gamma    = 1;
tf       = 10000;

states = 5:1:100;

for j=1:length(states)
    i=0;
    Prob = [1 0]';
    while (i+2) <= states(j)
        if i==0
            BigA = [-k gamma; k -k-gamma];
        else
            BigA = [[BigA; zeros(1,i) k] [zeros(i,1); (i+1)*gamma; -k-
(i+1)*gamma]];
            Prob = [Prob; 0];

        end
        i=i+1;
    end
    tempVec(1:length(BigA),j) = expm(BigA*tf)*Prob;
    tempError(j) = 1 - sum(tempVec(:,j))
    disp(j)
end
```

modelSweep.m

```
clear all
curError = 0;
k1 = 40;
k2 = 40;
while curError < 10^-2
    [prob, curError, BigA] = Toggle(k1,k2);
    tempStr = strcat('ToggleSwitch/toggle',int2str(k1),'.mat');
    save(tempStr,'prob','curError','BigA','k1','k2');
    clear tempStr prob BigA
    k1 = k1 + 1;
    k2 = k2 + 1;
    if k1 > 500
        break;
    end
end
clear all
curError = 0;
k1 = 1;
k2 = 1;
while curError < 10^-2
    [prob, curError, BigA] = ToggleMike(k1,k2);
    tempStr = strcat('ToggleSwitch/togglemike',int2str(k1),'.mat');
    save(tempStr,'prob','curError','BigA','k1','k2');
    clear tempStr prob BigA
    k1 = k1 + 1;
    k2 = k2 + 1;
    if k1 > 500
        break;
    end
```

```
end
clear all
curError = 0;
k1 = 1;
k2 = 1;
k3 = 1;
while curError < 10^-2
    [prob, curError, BigA] = RepressSelective(k1,k2,k3);
    tempStr = strcat('Repressilator/repress',int2str(k1),'.mat');
    save(tempStr,'prob','curError','BigA','k1','k2','k3');
    clear tempStr prob BigA
    k1 = k1 + 1;
    k2 = k2 + 1;
    k3 = k3 + 1;
end
clear all
curError = 0;
k1 = 1;
k2 = 1;
k3 = 1;
while curError < 10^-2
    [prob, curError, BigA] = RepressMikeSelective(k1,k2,k3);
    tempStr = strcat('Repressilator/repressmike',int2str(k1),'.mat');
    save(tempStr,'prob','curError','BigA','k1','k2','k3');
    clear tempStr prob BigA
    k1 = k1 + 1;
    k2 = k2 + 1;
    k3 = k3 + 1;
end
```

NegFeed.m

```
function [tempVec, tempError, BigA] = NegFeed(k1)

%k1       = 200;
g1       = 1;

b1       = 1;

tf       = 10000;
error    = 10^-6;
tempError = 1;
leap     = 0;

%build states
maxM = 1;
while tempError > error
    for i=1:(maxM+1)
        p1s(i) = mod((i-1),maxM+1);
    end
    BigA = zeros((maxM+1),(maxM+1));
    for i=1:(maxM+1)
        for j=1:(maxM+1)
            if(i == j)
                BigA(j,i) = -k1/(1+p1s(i)^b1) - p1s(i)*g1;
            else
                num = abs(p1s(i)-p1s(j));
```

```
                if(num ~= 1)
                    BigA(j,i) = 0;
                else
                    num = p1s(i)-p1s(j);
                    if(num > 0)
                        BigA(j,i) = p1s(i)*g1;
                    else
                        BigA(j,i) = k1/(1+p1s(i)^b1);
                    end
                end
            end
        end
    end

    Prob = zeros((maxM+1),1);
    Prob(1) = 1;
    %    BigA = BigA.';
    tempVec = expm(BigA*tf)*Prob;
    tempError = 1 - sum(tempVec);
    if tempError <= error
        break;
    end
    maxM = maxM+floor(leap*tempError)+1;
    if (maxM+1) > 11000
        break;
    end
    %     if maxM == 4
    %         break;
    %     end
end
```

NegFeedMike.m

```
function [tempVec, tempError, BigA] = NegFeedMike(k1,z)

%k1       = 200;
g1       = 1;

k1f      = z;
k1r      = z/k1;

tf       = 1000;
error    = 10^-6;
tempError = 1;
leap     = 0;

%build states
maxM = 1;
while tempError > error
    for i=1:(maxM+1)
        tempp1s(i) = mod((i-1),maxM+1);
    end
    for i=1:2
        p1s( ((maxM+1))*(i-1)+1:((maxM+1))*i ) = tempp1s;
        if(mod(i,2) == 1)
            tempg1s = zeros((maxM+1),1);
        else
```

```
            tempg1s = ones((maxM+1),1);
        end
        g1s( ((maxM+1))*(i-1)+1:((maxM+1))*i ) = tempg1s;
    end
    clear tempp1 tempg1s;
    BigA = zeros(2*(maxM+1),2*(maxM+1));
    for i=1:(2*(maxM+1))
        for j=1:(2*(maxM+1))
            if(i == j)
                BigA(j,i) = -g1s(i)*k1 - p1s(i)*g1 - p1s(i)*g1s(i)*k1r -
(abs(g1s(i)-1))*k1f;
            else
                num = abs(p1s(i)-p1s(j))+abs(g1s(i)-g1s(j));
                if(num ~= 1)
                    BigA(j,i) = 0;
                elseif(num == 1)
                    num1 = p1s(i)-p1s(j);
                    num4 = g1s(i)-g1s(j);
                    if(num1 ~= 0)
                        if(num1 > 0)
                            BigA(j,i) = p1s(i)*g1;
                        else
                            BigA(j,i) = g1s(i)*k1;
                        end
                    elseif(num4 ~= 0)
                        BigA(j,i) = p1s(i)*g1s(i)*k1r + (abs(g1s(i)-1))*k1f;
                    else
                        disp('This should never appear')
                    end
                end
            end
        end
    end
    Prob = zeros(2*(maxM+1),1);
    Prob(1) = 1;
%    BigA = BigA.';
    tempVec = expm(BigA*tf)*Prob;
    tempError = 1 - sum(tempVec);
    if tempError <= error
        break;
    end
    maxM = maxM+floor(leap*tempError)+1;
    if 2*(maxM+1) > 11000
        break;
    end
%     if maxM == 4
%         break;
%     end
end
```

NegFeedSweep.m

```
%function [tempVec, tempError, BigA] = NegFeedSweep(k1)

k1       = 100;
g1       = 1;
z        = 0.1;
```

```
k1f      = z;
k1r      = z/k1;

tf       = 10000;
error    = 10^-9;
tempError = 1;
leap     = 0;

%build states
maxM = 1;
for i=1:(maxM+1)
    tempp1s(i) = mod((i-1),maxM+1);
end
for i=1:2
    p1s( ((maxM+1))*(i-1)+1:((maxM+1))*i ) = tempp1s;
    if(mod(i,2) == 1)
        tempg1s = zeros((maxM+1),1);
    else
        tempg1s = ones((maxM+1),1);
    end
    g1s( ((maxM+1))*(i-1)+1:((maxM+1))*i ) = tempg1s;
end
edgeStates = [];
for i=1:length(p1s)
    if p1s(i) == max(p1s)
        edgeStates(length(edgeStates)+1) = i;
    end
end
clear tempp1 tempg1s;
counter = 1;
while tempError > error
    BigA = zeros(length(p1s),length(p1s));
    for i=1:length(p1s)
        for j=1:length(p1s)
            if(i == j)
                BigA(j,i) = -g1s(i)*k1 - p1s(i)*g1 - p1s(i)*g1s(i)*k1r -
(abs(g1s(i)-1))*k1f;
            else
                num = abs(p1s(i)-p1s(j))+abs(g1s(i)-g1s(j));
                if(num ~= 1)
                    BigA(j,i) = 0;
                elseif(num == 1)
                    num1 = p1s(i)-p1s(j);
                    num4 = g1s(i)-g1s(j);
                    if(num1 ~= 0)
                        if(num1 > 0)
                            BigA(j,i) = p1s(i)*g1;
                        else
                            BigA(j,i) = g1s(i)*k1;
                        end
                    elseif(num4 ~= 0)
                        BigA(j,i) = p1s(i)*g1s(i)*k1r + (abs(g1s(i)-1))*k1f;
                    else
                        disp('This should never appear')
                    end
                end
```

```
            end
        end
    end
    Prob = zeros(length(p1s),1);
    Prob(1) = 1;
    %    BigA = BigA.';
    tempVec = expm(BigA*tf)*Prob;
    tempError = 1 - sum(tempVec);
    pfin(1:max(p1s)+1,counter) = zeros(max(p1s)+1,1);
    for i=1:max(p1s)+1
        for j=1:length(p1s)
            if((i-1) == p1s(j))
                pfin(i,counter) = pfin(i,counter) + tempVec(j);
            end
        end
    end
    states(counter) = length(tempVec);
    counter = counter + 1;
    disp(length(tempVec))
    disp(tempError)
    if tempError <= error
        break;
    else
        numAdded = 0;
        edgeSize = length(edgeStates);
        while (numAdded < length(tempVec)) && (numAdded < 10) && (edgeSize >
0)
            maxNum = -1;
            maxIndex = -1;
            maxEdgeIndex = -1;
            for i=1:edgeSize
                if tempVec(edgeStates(i)) > maxNum
                    maxIndex = edgeStates(i);
                    maxNum = tempVec(maxIndex);
                    maxEdgeIndex = i;
                end
            end
            numFound1 = 0;
            numFound2 = 0;
            numFound3 = 0;
            for i=1:length(p1s)
                if (p1s(i) == p1s(maxIndex)+1)
                    numFound1 = 1;
                end
                if (p1s(i) == p1s(maxIndex)-1) || (p1s(maxIndex)-1 < 0)
                    numFound2 = 1;
                end
                if (p1s(i) == p1s(maxIndex)) && (g1s(i) == abs(g1s(maxIndex)-
1))
                    numFound3 = 1;
                end
            end
            if numFound1 == 0
                p1s(length(p1s)+1) = p1s(maxIndex)+1;
                g1s(length(g1s)+1) = g1s(maxIndex);
                numAdded = numAdded + 1;
                edgeStates(length(edgeStates)+1) = length(p1s);
```

```
            end
            if numFound2 == 0
                p1s(length(p1s)+1) = p1s(maxIndex)-1;
                g1s(length(g1s)+1) = g1s(maxIndex);
                numAdded = numAdded + 1;
                edgeStates(length(edgeStates)+1) = length(p1s);
            end
            if numFound3 == 0
                p1s(length(p1s)+1) = p1s(maxIndex);
                g1s(length(g1s)+1) = abs(g1s(maxIndex)-1);
                numAdded = numAdded + 1;
                edgeStates(length(edgeStates)+1) = length(p1s);
            end
            edgeStates(maxEdgeIndex) = [];
            edgeSize = edgeSize - 1;
        end
    end
%     end
%     maxM = maxM+floor(leap*tempError)+1;
%     if 2*(maxM+1) > 11000
%         break;
%     end
%     if maxM == 4
%         break;
%     end
end
```

plotData.m

```
tempVec = importdata('results.txt');
for i=1:length(tempVec)
    outMat(ceil(i/(sqrt(length(tempVec)))),mod(i-1,sqrt(length(tempVec)))+1)
= tempVec(i);
end
for i=1:length(outMat)
    pvec(i) = sum(outMat(:,i));
    mvec(i) = sum(outMat(i,:));
end
stem(pvec);
title('protein')
figure
stem(mvec)
title('mRNA')
```

psys.m

```
function dydt = psys(t,y)
dydt = [3 - y(1); 3*y(1) - y(2)];
```

Repress.m

```
function [pfin, tempError, BigA] = Repress(k1,k2,k3)

%k1       = 8;
g1       = 1;
%k2       = 8;
g2       = 1;
%k3       = 8;
```

```
g3       = 1;

b1       = 1;
b2       = 1;
b3       = 1;

tf       = 10;
error    = 0.1;
tempError = 1;
leap     = 0;

%build states
maxM = 1;
while tempError > error
    for i=1:(maxM+1)^3
        p1s(i) = mod((i-1),maxM+1);
        p2s(i) = p1s(floor((i-1)/(maxM+1))+1);
        p3s(i) = p2s(floor((i-1)/(maxM+1))+1);
    end
    BigA = zeros((maxM+1)^3,(maxM+1)^3);
    for i=1:(maxM+1)^3
        for j=1:(maxM+1)^3
            if(i == j)
                BigA(j,i) = -k1/(1+p3s(i)^b1) - k2/(1+p1s(i)^b2) -
k3/(1+p2s(i)^b3) - p1s(i)*g1 - p2s(i)*g2 - p3s(i)*g3;
            else
                num = abs(p1s(i)-p1s(j))+abs(p2s(i)-p2s(j))+abs(p3s(i)-
p3s(j));
                if(num ~= 1)
                    BigA(j,i) = 0;
                elseif(num == 1)
                    num1 = p1s(i)-p1s(j);
                    num2 = p2s(i)-p2s(j);
                    num3 = p3s(i)-p3s(j);
                    if(num1 ~= 0)
                        if(num1 > 0)
                            BigA(j,i) = p1s(i)*g1;
                        else
                            BigA(j,i) = k1/(1+p3s(i)^b1);
                        end
                    elseif(num2 ~= 0)
                        if(num2 > 0)
                            BigA(j,i) = p2s(i)*g2;
                        else
                            BigA(j,i) = k2/(1+p1s(i)^b2);
                        end
                    else
                        if(num3 > 0)
                            BigA(j,i) = p3s(i)*g3;
                        else
                            BigA(j,i) = k3/(1+p2s(i)^b3);
                        end
                    end
                end
            end
        end
    end
```

```
    Prob = [1; zeros((maxM+1)^3-1,1)];
%    BigA = BigA.';
    tempVec = expm(BigA*tf)*Prob;
    tempError = 1 - sum(tempVec)
    disp(length(tempVec))
    if tempError <= error
        break;
    end
    maxM = maxM+floor(leap*tempError)+1;
    if (maxM+1)^3 > 11000
        break;
    end
%     if maxM == 4
%         break;
%     end
end
p1fin = zeros(maxM+1,1);
p2fin = zeros(maxM+1,1);
p3fin = zeros(maxM+1,1);
pfin = zeros(maxM+1,maxM+1,maxM+1);
for i=1:(maxM+1)
    for j=1:(maxM+1)^3
        if(p1s(j) == (i-1))
            p1fin(i) = p1fin(i) + tempVec(j);
        end
        if(p2s(j) == (i-1))
            p2fin(i) = p2fin(i) + tempVec(j);
        end
        if(p3s(j) == (i-1))
            p3fin(i) = p3fin(i) + tempVec(j);
        end
    end
end
for i=1:(maxM+1)
    for j=1:(maxM+1)
        for k=1:(maxM+1)
            pfin(i,j,k) = tempVec((i-1)+1  + (j-1)*(maxM+1) + (k-
1)*(maxM+1)^2);
        end
    end
end
```

RepressMike.m

```
function [pfin, tempError, BigA] = RepressMike(k1,k2,k3)

%k1       = 2;
g1       = 1;
%k2       = 2;
g2       = 1;
%k3       = 2;
g3       = 1;

k1f      = 1;
k1r      = 1;
k2f      = 1;
k2r      = 1;
```

```
k3f      = 1;
k3r      = 1;

tf       = 10000;
error    = 10^-6;
tempError = 1;
leap     = 0;

%build states
maxM = 1;
while tempError > error
    for i=1:(maxM+1)^3
        tempp1s(i) = mod((i-1),maxM+1);
        tempp2s(i) = tempp1s(floor((i-1)/(maxM+1))+1);
        tempp3s(i) = tempp2s(floor((i-1)/(maxM+1))+1);
    end
    for i=1:8
        p1s( ((maxM+1)^3)*(i-1)+1:((maxM+1)^3)*i ) = tempp1s;
        p2s( ((maxM+1)^3)*(i-1)+1:((maxM+1)^3)*i ) = tempp2s;
        p3s( ((maxM+1)^3)*(i-1)+1:((maxM+1)^3)*i ) = tempp3s;
        if(mod(i,2) == 1)
            tempg1s = zeros((maxM+1)^3,1);
        else
            tempg1s = ones((maxM+1)^3,1);
        end
        if(mod((i-1),4) < 2)
            tempg2s = zeros((maxM+1)^3,1);
        else
            tempg2s = ones((maxM+1)^3,1);
        end
        if(mod((i-1),8) < 4)
            tempg3s = zeros((maxM+1)^3,1);
        else
            tempg3s = ones((maxM+1)^3,1);
        end
        g1s( ((maxM+1)^3)*(i-1)+1:((maxM+1)^3)*i ) = tempg1s;
        g2s( ((maxM+1)^3)*(i-1)+1:((maxM+1)^3)*i ) = tempg2s;
        g3s( ((maxM+1)^3)*(i-1)+1:((maxM+1)^3)*i ) = tempg3s;
    end
    clear tempp1s tempp2s tempp3s tempg1s tempg2s tempg3s;
    BigA = zeros(8*(maxM+1)^3,8*(maxM+1)^3);
    for i=1:(8*(maxM+1)^3)
        for j=1:(8*(maxM+1)^3)
            if(i == j)
                BigA(j,i) = -g1s(i)*k1 - g2s(i)*k2 - g3s(i)*k3 - p1s(i)*g1 -
p2s(i)*g2 - p3s(i)*g3 - p3s(i)*g1s(i)*k1r - p1s(i)*g2s(i)*k2r -
p2s(i)*g3s(i)*k3r - (abs(g1s(i)-1))*k1f - (abs(g2s(i)-1))*k2f - (abs(g3s(i)-
1))*k3f;
            else
                num = abs(p1s(i)-p1s(j))+abs(p2s(i)-p2s(j))+abs(p3s(i)-
p3s(j))+abs(g1s(i)-g1s(j))+abs(g2s(i)-g2s(j))+abs(g3s(i)-g3s(j));
                if(num ~= 1)
                    BigA(j,i) = 0;
                elseif(num == 1)
                    num1 = p1s(i)-p1s(j);
                    num2 = p2s(i)-p2s(j);
                    num3 = p3s(i)-p3s(j);
```

```
                    num4 = g1s(i)-g1s(j);
                    num5 = g2s(i)-g2s(j);
                    num6 = g3s(i)-g3s(j);
                    if(num1 ~= 0)
                        if(num1 > 0)
                            BigA(j,i) = p1s(i)*g1;
                        else
                            BigA(j,i) = g1s(i)*k1;
                        end
                    elseif(num2 ~= 0)
                        if(num2 > 0)
                            BigA(j,i) = p2s(i)*g2;
                        else
                            BigA(j,i) = g2s(i)*k2;
                        end
                    elseif(num3 ~= 0)
                        if(num3 > 0)
                            BigA(j,i) = p3s(i)*g3;
                        else
                            BigA(j,i) = g3s(i)*k3;
                        end
                    elseif(num4 ~= 0)
                        BigA(j,i) = p3s(i)*g1s(i)*k1r + (abs(g1s(i)-1))*k1f;
                    elseif(num5 ~= 0)
                        BigA(j,i) = p1s(i)*g2s(i)*k2r + (abs(g2s(i)-1))*k2f;
                    elseif(num6 ~= 0)
                        BigA(j,i) = p2s(i)*g3s(i)*k3r + (abs(g3s(i)-1))*k3f;
                    else
                        disp('This should never appear')
                    end
                end
            end
        end
    end
    Prob = zeros(8*(maxM+1)^3,1);
    Prob(1) = 1;
%    BigA = BigA.';
    tempVec = expm(BigA*tf)*Prob;
    tempError = 1 - sum(tempVec);
    if tempError <= error
        break;
    end
    maxM = maxM+floor(leap*tempError)+1;
    if 8*(maxM+1)^3 > 11000
        break;
    end
%     if maxM == 4
%         break;
%     end
end
p1fin = zeros(maxM+1,1);
p2fin = zeros(maxM+1,1);
p3fin = zeros(maxM+1,1);
pfin = zeros(maxM+1,maxM+1,maxM+1);
for i=1:(maxM+1)
    for j=1:(8*(maxM+1)^3)
        if(p1s(j) == (i-1))
```

```
            p1fin(i) = p1fin(i) + tempVec(j);
        end
        if(p2s(j) == (i-1))
            p2fin(i) = p2fin(i) + tempVec(j);
        end
        if(p3s(j) == (i-1))
            p3fin(i) = p3fin(i) + tempVec(j);
        end
    end
end
for i=1:(maxM+1)
    for j=1:(maxM+1)
        for k=1:(maxM+1)
            pfin(i,j,k) = tempVec((i-1)+1  + (j-1)*(maxM+1) + (k-
1)*(maxM+1)^2);
        end
    end
end
```

RepressMikeSelective.m

```
%function [pfin, tempError, BigA] = RepressMikeSelective(k1,k2,k3)

k1       = 2;
g1       = 1;
k2       = 2;
g2       = 1;
k3       = 2;
g3       = 1;

k1f      = 1;
k1r      = .1;
k2f      = 1;
k2r      = .1;
k3f      = 1;
k3r      = .1;

tf       = 100;
error    = 0.1;
tempError = 1;
leap     = 0;

%build states
maxM = 1;
while tempError > error
    for i=1:(maxM+1)^3
        tempp1s(i) = mod((i-1),maxM+1);
        tempp2s(i) = tempp1s(floor((i-1)/(maxM+1))+1);
        tempp3s(i) = tempp2s(floor((i-1)/(maxM+1))+1);
    end
    for i=1:8
        p1s( ((maxM+1)^3)*(i-1)+1:((maxM+1)^3)*i ) = tempp1s;
        p2s( ((maxM+1)^3)*(i-1)+1:((maxM+1)^3)*i ) = tempp2s;
        p3s( ((maxM+1)^3)*(i-1)+1:((maxM+1)^3)*i ) = tempp3s;
        if(mod(i,2) == 1)
            tempg1s = zeros((maxM+1)^3,1);
        else
```

```
            tempg1s = ones((maxM+1)^3,1);
        end
        if(mod((i-1),4) < 2)
            tempg2s = zeros((maxM+1)^3,1);
        else
            tempg2s = ones((maxM+1)^3,1);
        end
        if(mod((i-1),8) < 4)
            tempg3s = zeros((maxM+1)^3,1);
        else
            tempg3s = ones((maxM+1)^3,1);
        end
        g1s( ((maxM+1)^3)*(i-1)+1:((maxM+1)^3)*i ) = tempg1s;
        g2s( ((maxM+1)^3)*(i-1)+1:((maxM+1)^3)*i ) = tempg2s;
        g3s( ((maxM+1)^3)*(i-1)+1:((maxM+1)^3)*i ) = tempg3s;
    end
    i = 2;
    while(i <= length(p1s))
        if (p1s(i) + p2s(i) + p3s(i)) > (maxM+1)
            p1s(i) = [];
            p2s(i) = [];
            p3s(i) = [];
            g1s(i) = [];
            g2s(i) = [];
            g3s(i) = [];
        else
            i=i+1;
        end
    end
    if length(p1s) > 11000
        break
    end
    clear tempp1s tempp2s tempp3s tempg1s tempg2s tempg3s;
    BigA = zeros(length(p1s),length(p1s));
    for i=1:length(BigA)
        for j=1:length(BigA)
            if(i == j)
                BigA(j,i) = -g1s(i)*k1 - g2s(i)*k2 - g3s(i)*k3 - p1s(i)*g1 -
p2s(i)*g2 - p3s(i)*g3 - p3s(i)*g1s(i)*k1r - p1s(i)*g2s(i)*k2r -
p2s(i)*g3s(i)*k3r - (abs(g1s(i)-1))*k1f - (abs(g2s(i)-1))*k2f - (abs(g3s(i)-
1))*k3f;
            else
                num = abs(p1s(i)-p1s(j))+abs(p2s(i)-p2s(j))+abs(p3s(i)-
p3s(j))+abs(g1s(i)-g1s(j))+abs(g2s(i)-g2s(j))+abs(g3s(i)-g3s(j));
                if(num ~= 1)
                    BigA(j,i) = 0;
                elseif(num == 1)
                    num1 = p1s(i)-p1s(j);
                    num2 = p2s(i)-p2s(j);
                    num3 = p3s(i)-p3s(j);
                    num4 = g1s(i)-g1s(j);
                    num5 = g2s(i)-g2s(j);
                    num6 = g3s(i)-g3s(j);
                    if(num1 ~= 0)
                        if(num1 > 0)
                            BigA(j,i) = p1s(i)*g1;
                        else
```

```
                        BigA(j,i) = g1s(i)*k1;
                    end
                elseif(num2 ~= 0)
                    if(num2 > 0)
                        BigA(j,i) = p2s(i)*g2;
                    else
                        BigA(j,i) = g2s(i)*k2;
                    end
                elseif(num3 ~= 0)
                    if(num3 > 0)
                        BigA(j,i) = p3s(i)*g3;
                    else
                        BigA(j,i) = g3s(i)*k3;
                    end
                elseif(num4 ~= 0)
                    BigA(j,i) = p3s(i)*g1s(i)*k1r + (abs(g1s(i)-1))*k1f;
                elseif(num5 ~= 0)
                    BigA(j,i) = p1s(i)*g2s(i)*k2r + (abs(g2s(i)-1))*k2f;
                elseif(num6 ~= 0)
                    BigA(j,i) = p2s(i)*g3s(i)*k3r + (abs(g3s(i)-1))*k3f;
                else
                    disp('This should never appear')
                end
            end
        end
    end
    Prob = zeros(length(BigA),1);
    Prob(1) = 1;
%    BigA = BigA.';
    tempVec = expm(BigA*tf)*Prob;
    tempError = 1 - sum(tempVec);
    disp(length(tempVec))
    disp(tempError)
    if tempError <= error
        break;
    end
    maxM = maxM+floor(leap*tempError)+1;
%     if maxM == 4
%         break;
%     end
end
pfin = zeros(maxM+1,maxM+1,maxM+1);
for i=1:length(pfin)
    for j=1:length(pfin)
        for k=1:length(pfin)
            for l=1:length(p1s)
                if ((p1s(l)+1) == i) && ((p2s(l)+1) == j) && ((p3s(l)+1) ==
k)
                    pfin(i,j,k) = tempVec(l);
                end
            end
        end
    end
end
```

RepressMikeSelectiveSmart.m

```
%function [pfin, tempError, BigA] = RepressMikeSelectiveSmart(k1,k2,k3)

k1       = 1;
g1       = 1;
k2       = 1;
g2       = 1;
k3       = 1;
g3       = 1;

k1r      = 1;
k1f      = 1;
k2r      = 1;
k2f      = 1;
k3r      = 1;
k3f      = 1;

tf       = 100;
error    = 0.1;
tempError = 1;
leap     = 0;

%build states
maxM = 1;
for i=1:(maxM+1)^3
    tempp1s(i) = mod((i-1),maxM+1);
    tempp2s(i) = tempp1s(floor((i-1)/(maxM+1))+1);
    tempp3s(i) = tempp2s(floor((i-1)/(maxM+1))+1);
end
for i=1:8
    p1s( ((maxM+1)^3)*(i-1)+1:((maxM+1)^3)*i ) = tempp1s;
    p2s( ((maxM+1)^3)*(i-1)+1:((maxM+1)^3)*i ) = tempp2s;
    p3s( ((maxM+1)^3)*(i-1)+1:((maxM+1)^3)*i ) = tempp3s;
    if(mod(i,2) == 1)
        tempg1s = zeros((maxM+1)^3,1);
    else
        tempg1s = ones((maxM+1)^3,1);
    end
    if(mod((i-1),4) < 2)
        tempg2s = zeros((maxM+1)^3,1);
    else
        tempg2s = ones((maxM+1)^3,1);
    end
    if(mod((i-1),8) < 4)
        tempg3s = zeros((maxM+1)^3,1);
    else
        tempg3s = ones((maxM+1)^3,1);
    end
    g1s( ((maxM+1)^3)*(i-1)+1:((maxM+1)^3)*i ) = tempg1s;
    g2s( ((maxM+1)^3)*(i-1)+1:((maxM+1)^3)*i ) = tempg2s;
    g3s( ((maxM+1)^3)*(i-1)+1:((maxM+1)^3)*i ) = tempg3s;
end
edgeStates = [];
for i=1:length(p1s)
    if (p1s(i) == 1) || (p2s(i) == 1) || (p3s(i) == 1)
        edgeStates(length(edgeStates)+1) = i;
```

```
    end
end
while tempError > error
    if length(p1s) > 11000
        break
    end
    clear tempp1s tempp2s tempp3s tempg1s tempg2s tempg3s;
    BigA = zeros(length(p1s),length(p1s));
    for i=1:length(BigA)
        for j=1:length(BigA)
            if(i == j)
                BigA(j,i) = -g1s(i)*k1 - g2s(i)*k2 - g3s(i)*k3 - p1s(i)*g1 -
p2s(i)*g2 - p3s(i)*g3 - p3s(i)*g1s(i)*k1r - p1s(i)*g2s(i)*k2r -
p2s(i)*g3s(i)*k3r - (abs(g1s(i)-1))*k1f - (abs(g2s(i)-1))*k2f - (abs(g3s(i)-
1))*k3f;
            else
                num = abs(p1s(i)-p1s(j))+abs(p2s(i)-p2s(j))+abs(p3s(i)-
p3s(j))+abs(g1s(i)-g1s(j))+abs(g2s(i)-g2s(j))+abs(g3s(i)-g3s(j));
                if(num ~= 1)
                    BigA(j,i) = 0;
                elseif(num == 1)
                    num1 = p1s(i)-p1s(j);
                    num2 = p2s(i)-p2s(j);
                    num3 = p3s(i)-p3s(j);
                    num4 = g1s(i)-g1s(j);
                    num5 = g2s(i)-g2s(j);
                    num6 = g3s(i)-g3s(j);
                    if(num1 ~= 0)
                        if(num1 > 0)
                            BigA(j,i) = p1s(i)*g1;
                        else
                            BigA(j,i) = g1s(i)*k1;
                        end
                    elseif(num2 ~= 0)
                        if(num2 > 0)
                            BigA(j,i) = p2s(i)*g2;
                        else
                            BigA(j,i) = g2s(i)*k2;
                        end
                    elseif(num3 ~= 0)
                        if(num3 > 0)
                            BigA(j,i) = p3s(i)*g3;
                        else
                            BigA(j,i) = g3s(i)*k3;
                        end
                    elseif(num4 ~= 0)
                        BigA(j,i) = p3s(i)*g1s(i)*k1r + (abs(g1s(i)-1))*k1f;
                    elseif(num5 ~= 0)
                        BigA(j,i) = p1s(i)*g2s(i)*k2r + (abs(g2s(i)-1))*k2f;
                    elseif(num6 ~= 0)
                        BigA(j,i) = p2s(i)*g3s(i)*k3r + (abs(g3s(i)-1))*k3f;
                    else
                        disp('This should never appear')
                    end
                end
            end
        end
```

```
    end
    Prob = zeros(length(BigA),1);
    Prob(1) = 1;
%    BigA = BigA.';
    tempVec = expm(BigA*tf)*Prob;
    tempError = 1 - sum(tempVec);
    disp(length(tempVec))
    disp(tempError)
    if tempError <= error
        break;
    else
        numAdded = 0;
        edgeSize = length(edgeStates);
        while (numAdded < length(tempVec)) && (numAdded < 80) && (edgeSize >
0)
            maxNum = -1;
            maxIndex = -1;
            maxEdgeIndex = -1;
            for i=1:edgeSize
                if tempVec(edgeStates(i)) > maxNum
                    maxIndex = edgeStates(i);
                    maxNum = tempVec(maxIndex);
                    maxEdgeIndex = i;
                end
            end
            numFound1 = 0;
            numFound2 = 0;
            numFound3 = 0;
            numFound4 = 0;
            numFound5 = 0;
            numFound6 = 0;
            numFound7 = 0;
            numFound8 = 0;
            numFound9 = 0;
            for i=1:length(p1s)
                if (p1s(i) == p1s(maxIndex)+1) && (p2s(i) == p2s(maxIndex))
&& (p3s(i) == p3s(maxIndex)) && (g1s(i) == g1s(maxIndex)) && (g2s(i) ==
g2s(maxIndex)) && (g3s(i) == g3s(maxIndex))
                    numFound1 = 1;
                end
                if ((p1s(i) == p1s(maxIndex)-1) && (p2s(i) == p2s(maxIndex))
&& (p3s(i) == p3s(maxIndex)) && (g1s(i) == g1s(maxIndex)) && (g2s(i) ==
g2s(maxIndex)) && (g3s(i) == g3s(maxIndex))) || (p1s(maxIndex)-1 < 0)
                    numFound2 = 1;
                end
                if (p1s(i) == p1s(maxIndex)) && (p2s(i) == p2s(maxIndex)+1)
&& (p3s(i) == p3s(maxIndex)) && (g1s(i) == g1s(maxIndex)) && (g2s(i) ==
g2s(maxIndex)) && (g3s(i) == g3s(maxIndex))
                    numFound3 = 1;
                end
                if ((p1s(i) == p1s(maxIndex)) && (p2s(i) == p2s(maxIndex)-1)
&& (p3s(i) == p3s(maxIndex)) && (g1s(i) == g1s(maxIndex)) && (g2s(i) ==
g2s(maxIndex)) && (g3s(i) == g3s(maxIndex))) || (p2s(maxIndex)-1 < 0)
                    numFound4 = 1;
                end
```

```
                if (p1s(i) == p1s(maxIndex)) && (p2s(i) == p2s(maxIndex)) &&
(p3s(i) == p3s(maxIndex)+1) && (g1s(i) == g1s(maxIndex)) && (g2s(i) ==
g2s(maxIndex)) && (g3s(i) == g3s(maxIndex))
                    numFound5 = 1;
                end
                if ((p1s(i) == p1s(maxIndex)) && (p2s(i) == p2s(maxIndex)) &&
(p3s(i) == p3s(maxIndex)-1) && (g1s(i) == g1s(maxIndex)) && (g2s(i) ==
g2s(maxIndex)) && (g3s(i) == g3s(maxIndex))) || (p3s(maxIndex)-1 < 0)
                    numFound6 = 1;
                end
                if (p1s(i) == p1s(maxIndex)) && (p2s(i) == p2s(maxIndex)) &&
(p3s(i) == p3s(maxIndex)) && (g1s(i) == abs(g1s(maxIndex)-1)) && (g2s(i) ==
g2s(maxIndex)) && (g3s(i) == g3s(maxIndex))
                    numFound7 = 1;
                end
                if (p1s(i) == p1s(maxIndex)) && (p2s(i) == p2s(maxIndex)) &&
(p3s(i) == p3s(maxIndex)) && (g1s(i) == g1s(maxIndex)) && (g2s(i) ==
abs(g2s(maxIndex)-1)) && (g3s(i) == g3s(maxIndex))
                    numFound8 = 1;
                end
                if (p1s(i) == p1s(maxIndex)) && (p2s(i) == p2s(maxIndex)) &&
(p3s(i) == p3s(maxIndex)) && (g1s(i) == g1s(maxIndex)) && (g2s(i) ==
g2s(maxIndex)) && (g3s(i) == abs(g3s(maxIndex)-1))
                    numFound9 = 1;
                end
            end
            if (numFound1 == 0) && (g1s(maxIndex) == 1)
                p1s(length(p1s)+1) = p1s(maxIndex)+1;
                p2s(length(p2s)+1) = p2s(maxIndex);
                p3s(length(p3s)+1) = p3s(maxIndex);
                g1s(length(g1s)+1) = g1s(maxIndex);
                g2s(length(g2s)+1) = g2s(maxIndex);
                g3s(length(g3s)+1) = g3s(maxIndex);

                numAdded = numAdded + 1;
                edgeStates(length(edgeStates)+1) = length(p1s);
            elseif numFound2 == 0
                p1s(length(p1s)+1) = p1s(maxIndex)-1;
                p2s(length(p2s)+1) = p2s(maxIndex);
                p3s(length(p3s)+1) = p3s(maxIndex);
                g1s(length(g1s)+1) = g1s(maxIndex);
                g2s(length(g2s)+1) = g2s(maxIndex);
                g3s(length(g3s)+1) = g3s(maxIndex);

                numAdded = numAdded + 1;
                edgeStates(length(edgeStates)+1) = length(p1s);
            elseif (numFound3 == 0) && (g2s(maxIndex) == 1)
                p1s(length(p1s)+1) = p1s(maxIndex);
                p2s(length(p2s)+1) = p2s(maxIndex)+1;
                p3s(length(p3s)+1) = p3s(maxIndex);
                g1s(length(g1s)+1) = g1s(maxIndex);
                g2s(length(g2s)+1) = g2s(maxIndex);
                g3s(length(g3s)+1) = g3s(maxIndex);

                numAdded = numAdded + 1;
                edgeStates(length(edgeStates)+1) = length(p1s);
            elseif numFound4 == 0
```

```
    p1s(length(p1s)+1) = p1s(maxIndex);
    p2s(length(p2s)+1) = p2s(maxIndex)-1;
    p3s(length(p3s)+1) = p3s(maxIndex);
    g1s(length(g1s)+1) = g1s(maxIndex);
    g2s(length(g2s)+1) = g2s(maxIndex);
    g3s(length(g3s)+1) = g3s(maxIndex);

    numAdded = numAdded + 1;
    edgeStates(length(edgeStates)+1) = length(p1s);
elseif (numFound5 == 0) && (g3s(maxIndex) == 1)
    p1s(length(p1s)+1) = p1s(maxIndex);
    p2s(length(p2s)+1) = p2s(maxIndex);
    p3s(length(p3s)+1) = p3s(maxIndex)+1;
    g1s(length(g1s)+1) = g1s(maxIndex);
    g2s(length(g2s)+1) = g2s(maxIndex);
    g3s(length(g3s)+1) = g3s(maxIndex);

    numAdded = numAdded + 1;
    edgeStates(length(edgeStates)+1) = length(p1s);
elseif numFound6 == 0
    p1s(length(p1s)+1) = p1s(maxIndex);
    p2s(length(p2s)+1) = p2s(maxIndex);
    p3s(length(p3s)+1) = p3s(maxIndex)-1;
    g1s(length(g1s)+1) = g1s(maxIndex);
    g2s(length(g2s)+1) = g2s(maxIndex);
    g3s(length(g3s)+1) = g3s(maxIndex);

    numAdded = numAdded + 1;
    edgeStates(length(edgeStates)+1) = length(p1s);
elseif numFound7 == 0
    p1s(length(p1s)+1) = p1s(maxIndex);
    p2s(length(p2s)+1) = p2s(maxIndex);
    p3s(length(p3s)+1) = p3s(maxIndex);
    g1s(length(g1s)+1) = abs(g1s(maxIndex)-1);
    g2s(length(g2s)+1) = g2s(maxIndex);
    g3s(length(g3s)+1) = g3s(maxIndex);

    numAdded = numAdded + 1;
    edgeStates(length(edgeStates)+1) = length(p1s);
elseif numFound8 == 0
    p1s(length(p1s)+1) = p1s(maxIndex);
    p2s(length(p2s)+1) = p2s(maxIndex);
    p3s(length(p3s)+1) = p3s(maxIndex);
    g1s(length(g1s)+1) = g1s(maxIndex);
    g2s(length(g2s)+1) = abs(g2s(maxIndex)-1);
    g3s(length(g3s)+1) = g3s(maxIndex);

    numAdded = numAdded + 1;
    edgeStates(length(edgeStates)+1) = length(p1s);
elseif numFound9 == 0
    p1s(length(p1s)+1) = p1s(maxIndex);
    p2s(length(p2s)+1) = p2s(maxIndex);
    p3s(length(p3s)+1) = p3s(maxIndex);
    g1s(length(g1s)+1) = g1s(maxIndex);
    g2s(length(g2s)+1) = g2s(maxIndex);
    g3s(length(g3s)+1) = abs(g3s(maxIndex)-1);
```

```
                numAdded = numAdded + 1;
                edgeStates(length(edgeStates)+1) = length(p1s);
            else
                edgeStates(maxEdgeIndex) = [];
                edgeSize = edgeSize - 1;
            end
        end
    end
    if (length(p1s) == 112)
        break;
    end
%     if maxM == 4
%         break;
%     end
end
pSize = max([max(p1s),max(p2s),max(p3s)]);
pfin = zeros(pSize+1,pSize+1,pSize+1);
for i=1:length(pfin)
    for j=1:length(pfin)
        for k=1:length(pfin)
            for l=1:length(p1s)
                if ((p1s(l)+1) == i) && ((p2s(l)+1) == j) && ((p3s(l)+1) ==
k)
                    pfin(i,j,k) = tempVec(l);
                end
            end
        end
    end
end
```

RepressSelective.m

```
%function [pfin, tempError, BigA] = RepressSelective(k1,k2,k3,error)

k1       = 3;
g1       = 1;
k2       = 3;
g2       = 1;
k3       = 3;
g3       = 1;

b1       = 1;
b2       = 1;
b3       = 1;

tf       = 100;
error    = 10;
tempError = 1;
leap     = 0;

%build states
maxM = 1;
while tempError > error
    for i=1:(maxM+1)^3
        p1s(i) = mod((i-1),maxM+1);
        p2s(i) = p1s(floor((i-1)/(maxM+1))+1);
        p3s(i) = p2s(floor((i-1)/(maxM+1))+1);
```

```
    end
    i = 2;
    while(i <= length(p1s))
        if (p1s(i) + p2s(i) + p3s(i)) > (maxM+1)
            p1s(i) = [];
            p2s(i) = [];
            p3s(i) = [];
        else
            i=i+1;
        end
    end
    if length(p1s) > 11000
        break
    end
    BigA = zeros(length(p1s),length(p1s));
    for i=1:length(BigA)
        for j=1:length(BigA)
            if(i == j)
                BigA(j,i) = -k1/(1+p3s(i)^b1) - k2/(1+p1s(i)^b2) -
k3/(1+p2s(i)^b3) - p1s(i)*g1 - p2s(i)*g2 - p3s(i)*g3;
            else
                num = abs(p1s(i)-p1s(j))+abs(p2s(i)-p2s(j))+abs(p3s(i)-
p3s(j));
                if(num ~= 1)
                    BigA(j,i) = 0;
                elseif(num == 1)
                    num1 = p1s(i)-p1s(j);
                    num2 = p2s(i)-p2s(j);
                    num3 = p3s(i)-p3s(j);
                    if(num1 ~= 0)
                        if(num1 > 0)
                            BigA(j,i) = p1s(i)*g1;
                        else
                            BigA(j,i) = k1/(1+p3s(i)^b1);
                        end
                    elseif(num2 ~= 0)
                        if(num2 > 0)
                            BigA(j,i) = p2s(i)*g2;
                        else
                            BigA(j,i) = k2/(1+p1s(i)^b2);
                        end
                    else
                        if(num3 > 0)
                            BigA(j,i) = p3s(i)*g3;
                        else
                            BigA(j,i) = k3/(1+p2s(i)^b3);
                        end
                    end
                end
            end
        end
    end
    Prob = zeros(length(BigA),1);
    Prob(1) = 1;
%    BigA = BigA.';
    tempVec = expm(BigA*tf)*Prob;
    tempError = 1 - sum(tempVec);
```

```
    disp(length(tempVec))
    disp(tempError)
    if tempError <= error
        break;
    end
    maxM = maxM+floor(leap*tempError)+1;
%     if maxM == 4
%         break;
%     end
end
pfin = zeros(maxM+1,maxM+1,maxM+1);
for i=1:length(pfin)
    for j=1:length(pfin)
        for k=1:length(pfin)
            for l=1:length(p1s)
                if ((p1s(l)+1) == i) && ((p2s(l)+1) == j) && ((p3s(l)+1) ==
k)
                    pfin(i,j,k) = tempVec(l);
                end
            end
        end
    end
end
```

RepressSelectiveSmart.m

```
%function [pfin, tempError, BigA] = RepressSelectiveSmart(k1,k2,k3,error)

k1       = 10;
g1       = 1;
k2       = 10;
g2       = 1;
k3       = 10;
g3       = 1;

b1       = 1;
b2       = 1;
b3       = 1;

tf       = 100;
error    = 0.01;
tempError = 1;
leap     = 0;

%build states
maxM = 1;
for i=1:(maxM+1)^3
    p1s(i) = mod((i-1),maxM+1);
    p2s(i) = p1s(floor((i-1)/(maxM+1))+1);
    p3s(i) = p2s(floor((i-1)/(maxM+1))+1);
end
edgeStates = 2:8;
while tempError > error
    if length(p1s) > 11000
        break
    end
    BigA = zeros(length(p1s),length(p1s));
```

```
    for i=1:length(BigA)
        for j=1:length(BigA)
            if(i == j)
                BigA(j,i) = -k1/(1+p3s(i)^b1) - k2/(1+p1s(i)^b2) -
k3/(1+p2s(i)^b3) - p1s(i)*g1 - p2s(i)*g2 - p3s(i)*g3;
            else
                num = abs(p1s(i)-p1s(j))+abs(p2s(i)-p2s(j))+abs(p3s(i)-
p3s(j));
                if(num ~= 1)
                    BigA(j,i) = 0;
                elseif(num == 1)
                    num1 = p1s(i)-p1s(j);
                    num2 = p2s(i)-p2s(j);
                    num3 = p3s(i)-p3s(j);
                    if(num1 ~= 0)
                        if(num1 > 0)
                            BigA(j,i) = p1s(i)*g1;
                        else
                            BigA(j,i) = k1/(1+p3s(i)^b1);
                        end
                    elseif(num2 ~= 0)
                        if(num2 > 0)
                            BigA(j,i) = p2s(i)*g2;
                        else
                            BigA(j,i) = k2/(1+p1s(i)^b2);
                        end
                    else
                        if(num3 > 0)
                            BigA(j,i) = p3s(i)*g3;
                        else
                            BigA(j,i) = k3/(1+p2s(i)^b3);
                        end
                    end
                end
            end
        end
    end
    Prob = zeros(length(BigA),1);
    Prob(1) = 1;
%    BigA = BigA.';
    tempVec = expm(BigA*tf)*Prob;
    tempError = 1 - sum(tempVec);
    disp(length(tempVec))
    disp(tempError)
    if tempError <= error
        break;
    else
        numAdded = 0;
        edgeSize = length(edgeStates);
        numToAdd = ((log10(tempError)-log10(error))^2)*1000+10;
        while (numAdded < length(tempVec)) && (numAdded < numToAdd) &&
(edgeSize > 0)
            maxNum = -1;
            maxIndex = -1;
            maxEdgeIndex = -1;
            for i=1:edgeSize
                if tempVec(edgeStates(i)) > maxNum
```

```
                maxIndex = edgeStates(i);
                maxNum = tempVec(maxIndex);
                maxEdgeIndex = i;
            end
        end
        numFound1 = 0;
        numFound2 = 0;
        numFound3 = 0;
        numFound4 = 0;
        numFound5 = 0;
        numFound6 = 0;
        for i=1:length(p1s)
            if (p1s(i) == p1s(maxIndex)+1) && (p2s(i) == p2s(maxIndex))
&& (p3s(i) == p3s(maxIndex))
                numFound1 = 1;
            end
            if ((p1s(i) == p1s(maxIndex)-1) && (p2s(i) == p2s(maxIndex))
&& (p3s(i) == p3s(maxIndex))) || (p1s(maxIndex)-1 < 0)
                numFound2 = 1;
            end
            if (p1s(i) == p1s(maxIndex)) && (p2s(i) == p2s(maxIndex)+1)
&& (p3s(i) == p3s(maxIndex))
                numFound3 = 1;
            end
            if ((p1s(i) == p1s(maxIndex)) && (p2s(i) == p2s(maxIndex)-1)
&& (p3s(i) == p3s(maxIndex))) || (p2s(maxIndex)-1 < 0)
                numFound4 = 1;
            end
            if (p1s(i) == p1s(maxIndex)) && (p2s(i) == p2s(maxIndex)) &&
(p3s(i) == p3s(maxIndex)+1)
                numFound5 = 1;
            end
            if ((p1s(i) == p1s(maxIndex)) && (p2s(i) == p2s(maxIndex)) &&
(p3s(i) == p3s(maxIndex)-1)) || (p3s(maxIndex)-1 < 0)
                numFound6 = 1;
            end
        end
        if numFound1 == 0
            p1s(length(p1s)+1) = p1s(maxIndex)+1;
            p2s(length(p2s)+1) = p2s(maxIndex);
            p3s(length(p3s)+1) = p3s(maxIndex);
            numAdded = numAdded + 1;
            edgeStates(length(edgeStates)+1) = length(p1s);
        elseif numFound2 == 0
            p1s(length(p1s)+1) = p1s(maxIndex)-1;
            p2s(length(p2s)+1) = p2s(maxIndex);
            p3s(length(p3s)+1) = p3s(maxIndex);
            numAdded = numAdded + 1;
            edgeStates(length(edgeStates)+1) = length(p1s);
        elseif numFound3 == 0
            p1s(length(p1s)+1) = p1s(maxIndex);
            p2s(length(p2s)+1) = p2s(maxIndex)+1;
            p3s(length(p3s)+1) = p3s(maxIndex);
            numAdded = numAdded + 1;
            edgeStates(length(edgeStates)+1) = length(p1s);
        elseif numFound4 == 0
            p1s(length(p1s)+1) = p1s(maxIndex);
```

```
                p2s(length(p2s)+1) = p2s(maxIndex)-1;
                p3s(length(p3s)+1) = p3s(maxIndex);
                numAdded = numAdded + 1;
                edgeStates(length(edgeStates)+1) = length(p1s);
            elseif numFound5 == 0
                p1s(length(p1s)+1) = p1s(maxIndex);
                p2s(length(p2s)+1) = p2s(maxIndex);
                p3s(length(p3s)+1) = p3s(maxIndex)+1;
                numAdded = numAdded + 1;
                edgeStates(length(edgeStates)+1) = length(p1s);
            elseif numFound6 == 0
                p1s(length(p1s)+1) = p1s(maxIndex);
                p2s(length(p2s)+1) = p2s(maxIndex);
                p3s(length(p3s)+1) = p3s(maxIndex)-1;
                numAdded = numAdded + 1;
                edgeStates(length(edgeStates)+1) = length(p1s);
            else
                edgeStates(maxEdgeIndex) = [];
                edgeSize = edgeSize - 1;
            end
        end
    end
%     if maxM == 4
%         break;
%     end
end
pSize = max([max(p1s),max(p2s),max(p3s)]);
pfin = zeros(pSize,pSize,pSize);
for i=1:length(pfin)
    for j=1:length(pfin)
        for k=1:length(pfin)
            for l=1:length(p1s)
                if ((p1s(l)+1) == i) && ((p2s(l)+1) == j) && ((p3s(l)+1) ==
k)
                    pfin(i,j,k) = tempVec(l);
                end
            end
        end
    end
end
```

testmodel2SweepBiz.m

```
% clear all
%
% disp('0')
% [finaltime,probVec,error] = testmodel2fast(5,5,9,10);
% save('mrun0.run.mat','finaltime','probVec','error');
% clear all
%
% disp('1')
% [finaltime,probVec,error] = testmodel2fast(10,10,19,10);
% save('mrun1.run.mat','finaltime','probVec','error');
% clear all
%
% disp('2')
% [finaltime,probVec,error] = testmodel2fast(15,15,29,10);
```

```
% save('mrun2.run.mat','finaltime','probVec','error');
% clear all
%
% disp('3')
% [finaltime,probVec,error] = testmodel2fast(20,20,39,10);
% save('mrun3.run.mat','finaltime','probVec','error');
% clear all
%
% disp('4')
% [finaltime,probVec,error] = testmodel2fast(25,25,49,10);
% save('mrun4.run.mat','finaltime','probVec','error');
% clear all
%
% disp('5')
% [finaltime,probVec,error] = testmodel2fast(30,30,59,10);
% save('mrun5.run.mat','finaltime','probVec','error');
% clear all
%
% disp('6')
% [finaltime,probVec,error] = testmodel2fast(35,35,69,10);
% save('mrun6.run.mat','finaltime','probVec','error');
% clear all
%
% disp('7')
% [finaltime,probVec,error] = testmodel2fast(40,40,79,10);
% save('mrun7.run.mat','finaltime','probVec','error');
% clear all
%
% disp('8')
% [finaltime,probVec,error] = testmodel2fast(45,45,89,10);
% save('mrun8.run.mat','finaltime','probVec','error');
% clear all
%
% disp('9')
% [finaltime,probVec,error] = testmodel2fast(50,50,99,10);
% save('mrun9.run.mat','finaltime','probVec','error');
% clear all
%
% disp('0')
% [finaltime,probVec,error] = testmodel2fast(20,20,99,1000);
% save('mitrun0.run.mat','finaltime','probVec','error');
% clear all
%
% disp('1')
% [finaltime,probVec,error] = testmodel2fast(20,20,99,1000000);
% save('mitrun1.run.mat','finaltime','probVec','error');
% clear all
%
% disp('2')
% [finaltime,probVec,error] = testmodel2fast(20,20,99,1000000000);
% save('mitrun2.run.mat','finaltime','probVec','error');
clear all

disp('3')
[finaltime,probVec,error] = testmodel2fast(20,20,99,1000000000000);
save('mitrun3.run.mat','finaltime','probVec','error');
% clear all
```

Toggle.m

```
function [pfin, tempError, BigA] = Toggle(k1,k2)

%k1       = 10;
g1       = 1;
%k2       = 10;
g2       = 1;

b1       = 1;
b2       = 1;

tf       = 10000;
error    = 10^-6;
tempError = 1;
leap     = 10;

%build states;
maxM = 1;
while tempError > error
    for i=1:(maxM+1)^2
        p1s(i) = mod((i-1),maxM+1);
        p2s(i) = p1s(floor((i-1)/(maxM+1))+1);
    end
    BigA = zeros((maxM+1)^2,(maxM+1)^2);
    for i=1:(maxM+1)^2
        for j=1:(maxM+1)^2
            if(i == j)
                BigA(j,i) = -k1/(1+p2s(i)^b1) - k2/(1+p1s(i)^b2) - p1s(i)*g1
- p2s(i)*g2;
            else
                num = abs(p1s(i)-p1s(j))+abs(p2s(i)-p2s(j));
                if(num ~= 1)
                    BigA(j,i) = 0;
                elseif(num == 1)
                    num1 = p1s(i)-p1s(j);
                    num2 = p2s(i)-p2s(j);
                    if(num1 ~= 0)
                        if(num1 > 0)
                            BigA(j,i) = p1s(i)*g1;
                        else
                            BigA(j,i) = k1/(1+p2s(i)^b1);
                        end
                    elseif(num2 ~= 0)
                        if(num2 > 0)
                            BigA(j,i) = p2s(i)*g2;
                        else
                            BigA(j,i) = k2/(1+p1s(i)^b2);
                        end
                    else
                        disp('Should never see this');
                    end
                end
            end
        end
    end
    Prob = zeros((maxM+1)^2,1);
```

```
    Prob(1) = 1;
%    BigA = BigA.';
    tempVec = expm(BigA*tf)*Prob;
    tempError = 1 - sum(tempVec);
    if tempError <= error
        break;
    end
    maxM = maxM+floor(leap*tempError)+1;
    if (maxM+1)^2 > 11000
        break;
    end
%     if maxM == 4
%         break;
%     end
end
p1fin = zeros(maxM+1,1);
p2fin = zeros(maxM+1,1);
pfin = zeros(maxM+1,maxM+1);
for i=1:(maxM+1)
    for j=1:(maxM+1)^2
        if(p1s(j) == (i-1))
            p1fin(i) = p1fin(i) + tempVec(j);
        end
        if(p2s(j) == (i-1))
            p2fin(i) = p2fin(i) + tempVec(j);
        end
    end
end
for i=1:(maxM+1)
    for j=1:(maxM+1)
        pfin(i,j) = tempVec((i-1)+1  + (j-1)*(maxM+1));
    end
end
```

ToggleMike.m

```
%function [pfin, tempError, BigA] = ToggleMike(k1,k2)

k1       = 3;
g1       = 1;
k2       = 3;
g2       = 1;

k1f      = 1;
k1r      = 0.5;
k2f      = 1;
k2r      = 0.5;

tf       = 100;
error    = 0.1;
tempError = 1;
leap     = 0;

%build states
maxM = 1;
while tempError > error
    for i=1:(maxM+1)^2
```

```
        tempp1s(i) = mod((i-1),maxM+1);
        tempp2s(i) = tempp1s(floor((i-1)/(maxM+1))+1);
    end
    for i=1:4
        p1s( ((maxM+1)^2)*(i-1)+1:((maxM+1)^2)*i ) = tempp1s;
        p2s( ((maxM+1)^2)*(i-1)+1:((maxM+1)^2)*i ) = tempp2s;
        if(mod(i,2) == 1)
            tempg1s = zeros((maxM+1)^2,1);
        else
            tempg1s = ones((maxM+1)^2,1);
        end
        if(mod((i-1),4) < 2)
            tempg2s = zeros((maxM+1)^2,1);
        else
            tempg2s = ones((maxM+1)^2,1);
        end
        g1s( ((maxM+1)^2)*(i-1)+1:((maxM+1)^2)*i ) = tempg1s;
        g2s( ((maxM+1)^2)*(i-1)+1:((maxM+1)^2)*i ) = tempg2s;
    end
    clear tempp1s tempp2s tempg1s tempg2s;
    BigA = zeros(4*(maxM+1)^2,4*(maxM+1)^2);
    for i=1:(4*(maxM+1)^2)
        for j=1:(4*(maxM+1)^2)
            if(i == j)
                BigA(j,i) = -g1s(i)*k1 - g2s(i)*k2 - p1s(i)*g1 - p2s(i)*g2 -
p2s(i)*g1s(i)*k1r - p1s(i)*g2s(i)*k2r - (abs(g1s(i)-1))*k1f - (abs(g2s(i)-
1))*k2f;
            else
                num = abs(p1s(i)-p1s(j))+abs(p2s(i)-p2s(j))+abs(g1s(i)-
g1s(j))+abs(g2s(i)-g2s(j));
                if(num ~= 1)
                    BigA(j,i) = 0;
                elseif(num == 1)
                    num1 = p1s(i)-p1s(j);
                    num2 = p2s(i)-p2s(j);
                    num4 = g1s(i)-g1s(j);
                    num5 = g2s(i)-g2s(j);
                    if(num1 ~= 0)
                        if(num1 > 0)
                            BigA(j,i) = p1s(i)*g1;
                        else
                            BigA(j,i) = g1s(i)*k1;
                        end
                    elseif(num2 ~= 0)
                        if(num2 > 0)
                            BigA(j,i) = p2s(i)*g2;
                        else
                            BigA(j,i) = g2s(i)*k2;
                        end
                    elseif(num4 ~= 0)
                        BigA(j,i) = p2s(i)*g1s(i)*k1r + (abs(g1s(i)-1))*k1f;
                    elseif(num5 ~= 0)
                        BigA(j,i) = p1s(i)*g2s(i)*k2r + (abs(g2s(i)-1))*k2f;
                    else
                        disp('This should never appear')
                    end
                end
```

```
            end
        end
    end
    Prob = zeros(4*(maxM+1)^2,1);
    Prob(1) = 1;
%    BigA = BigA.';
    tempVec = expm(BigA*tf)*Prob;
    tempError = 1 - sum(tempVec);
    disp(length(tempVec))
    disp(tempError)
    if tempError <= error
        break;
    end
    maxM = maxM+floor(leap*tempError)+1;
    if ( 4*(maxM+1)^2 > 11000)
        break;
    end
%     if maxM == 4
%         break;
%     end
end
p1fin = zeros(maxM+1,1);
p2fin = zeros(maxM+1,1);
pfin = zeros(maxM+1,maxM+1);
for i=1:(maxM+1)
    for j=1:(maxM+1)^2
        if(p1s(j) == (i-1))
            p1fin(i) = p1fin(i) + tempVec(j);
        end
        if(p2s(j) == (i-1))
            p2fin(i) = p2fin(i) + tempVec(j);
        end
    end
end
for i=1:(maxM+1)
    for j=1:(maxM+1)
        pfin(i,j) = tempVec((i-1)+1  + (j-1)*(maxM+1));
    end
end
```

ToggleMikeSweep.m

```
%function [pfin, tempError, BigA] = ToggleMike(k1,k2) THIS IS NOT DONE!!!

k1       = 2;
g1       = 1;
k2       = 2;
g2       = 1;
z        = 1;

k1f      = z;
k1r      = z/k1;
k2f      = z;
k2r      = z/k2;

tf       = 10;
error    = 0.1;
```

```
tempError = 1;
leap     = 0;

%build states
maxM = 1;
for i=1:(maxM+1)^2
    tempp1s(i) = mod((i-1),maxM+1);
    tempp2s(i) = tempp1s(floor((i-1)/(maxM+1))+1);
end
for i=1:4
    p1s( ((maxM+1)^2)*(i-1)+1:((maxM+1)^2)*i ) = tempp1s;
    p2s( ((maxM+1)^2)*(i-1)+1:((maxM+1)^2)*i ) = tempp2s;
    if(mod(i,2) == 1)
        tempg1s = zeros((maxM+1)^2,1);
    else
        tempg1s = ones((maxM+1)^2,1);
    end
    if(mod((i-1),4) < 2)
        tempg2s = zeros((maxM+1)^2,1);
    else
        tempg2s = ones((maxM+1)^2,1);
    end
    g1s( ((maxM+1)^2)*(i-1)+1:((maxM+1)^2)*i ) = tempg1s;
    g2s( ((maxM+1)^2)*(i-1)+1:((maxM+1)^2)*i ) = tempg2s;
end
edgeStates = [];
for i=1:length(p1s)
    if (p1s(i) == 1) || (p2s(i) == 1)
        edgeStates(length(edgeStates)+1) = i;
    end
end
clear tempp1s tempp2s tempg1s tempg2s;
counter = 1;
while tempError > error
    BigA = zeros(length(p1s),length(p1s));
    for i=1:length(p1s)
        for j=1:length(p1s)
            if(i == j)
                BigA(j,i) = -g1s(i)*k1 - g2s(i)*k2 - p1s(i)*g1 - p2s(i)*g2 -
p2s(i)*g1s(i)*k1r - p1s(i)*g2s(i)*k2r - (abs(g1s(i)-1))*k1f - (abs(g2s(i)-
1))*k2f;
            else
                num = abs(p1s(i)-p1s(j))+abs(p2s(i)-p2s(j))+abs(g1s(i)-
g1s(j))+abs(g2s(i)-g2s(j));
                if(num ~= 1)
                    BigA(j,i) = 0;
                elseif(num == 1)
                    num1 = p1s(i)-p1s(j);
                    num2 = p2s(i)-p2s(j);
                    num4 = g1s(i)-g1s(j);
                    num5 = g2s(i)-g2s(j);
                    if(num1 ~= 0)
                        if(num1 > 0)
                            BigA(j,i) = p1s(i)*g1;
                        else
                            BigA(j,i) = g1s(i)*k1;
                        end
```

```
                    elseif(num2 ~= 0)
                        if(num2 > 0)
                            BigA(j,i) = p2s(i)*g2;
                        else
                            BigA(j,i) = g2s(i)*k2;
                        end
                    elseif(num4 ~= 0)
                        BigA(j,i) = p2s(i)*g1s(i)*k1r + (abs(g1s(i)-1))*k1f;
                    elseif(num5 ~= 0)
                        BigA(j,i) = p1s(i)*g2s(i)*k2r + (abs(g2s(i)-1))*k2f;
                    else
                        disp('This should never appear')
                    end
                end
            end
        end
    end
    Prob = zeros(length(p1s),1);
    Prob(1) = 1;
%     BigA = BigA.';
    tempVec = expm(BigA*tf)*Prob;
    tempError = 1 - sum(tempVec);
    disp(length(p1s))
    disp(tempError)
    pSize = max([max(p1s)+1,max(p2s)+1]);
    pfin(1:pSize,1:pSize,counter) = zeros(pSize,pSize);
    for i=1:pSize
        for j=1:pSize
            for l=1:length(p1s)
                if ((p1s(l)+1) == i) && ((p2s(l)+1) == j)
                    pfin(i,j,counter) = pfin(i,j,counter) + tempVec(l);
                end
            end
        end
    end
    states(counter) = length(tempVec);
    counter = counter+1;
    if tempError <= error
        break;
    else
        numAdded = 0;
        edgeSize = length(edgeStates);
        while (numAdded < length(tempVec)) && (numAdded < 10) && (edgeSize >
0)
            maxNum = -1;
            maxIndex = -1;
            maxEdgeIndex = -1;
            for i=1:edgeSize
                if tempVec(edgeStates(i)) > maxNum
                    maxIndex = edgeStates(i);
                    maxNum = tempVec(maxIndex);
                    maxEdgeIndex = i;
                end
            end
            numFound1 = 0;
            numFound2 = 0;
            numFound3 = 0;
```

```
            numFound4 = 0;
            numFound5 = 0;
            numFound6 = 0;
            for i=1:length(p1s)
                if (p1s(i) == p1s(maxIndex)+1) && (p2s(i) == p2s(maxIndex))
&& (g1s(i) == g1s(maxIndex)) && (g2s(i) == g2s(maxIndex))
                    numFound1 = 1;
                end
                if ( (p1s(i) == p1s(maxIndex)-1) && (p2s(i) == p2s(maxIndex))
&& (g1s(i) == g1s(maxIndex)) && (g2s(i) == g2s(maxIndex)) ) ||
(p1s(maxIndex)-1 < 0)
                    numFound2 = 1;
                end
                if (p1s(i) == p1s(maxIndex)) && (p2s(i) == p2s(maxIndex)+1)
&& (g1s(i) == g1s(maxIndex)) && (g2s(i) == g2s(maxIndex))
                    numFound3 = 1;
                end
                if ( (p1s(i) == p1s(maxIndex)) && (p2s(i) == p2s(maxIndex)-1)
&& (g1s(i) == g1s(maxIndex)) && (g2s(i) == g2s(maxIndex)) ) ||
(p2s(maxIndex)-1 < 0)
                    numFound4 = 1;
                end
                if (p1s(i) == p1s(maxIndex)) && (p2s(i) == p2s(maxIndex)) &&
(g1s(i) == abs(g1s(maxIndex)-1)) && (g2s(i) == g2s(maxIndex))
                    numFound5 = 1;
                end
                if (p1s(i) == p1s(maxIndex)) && (p2s(i) == p2s(maxIndex)) &&
(g1s(i) == g1s(maxIndex)) && (g2s(i) == abs(g2s(maxIndex)-1))
                    numFound6 = 1;
                end
            end
            if (numFound1 == 0) && (g1s(maxIndex) == 1)
                p1s(length(p1s)+1) = p1s(maxIndex)+1;
                p2s(length(p2s)+1) = p2s(maxIndex);
                g1s(length(g1s)+1) = g1s(maxIndex);
                g2s(length(g2s)+1) = g2s(maxIndex);

                numAdded = numAdded + 1;
                edgeStates(length(edgeStates)+1) = length(p1s);
            end
            if numFound2 == 0
                p1s(length(p1s)+1) = p1s(maxIndex)-1;
                p2s(length(p2s)+1) = p2s(maxIndex);
                g1s(length(g1s)+1) = g1s(maxIndex);
                g2s(length(g2s)+1) = g2s(maxIndex);

                numAdded = numAdded + 1;
                edgeStates(length(edgeStates)+1) = length(p1s);
            end
            if (numFound3 == 0) && (g2s(maxIndex) == 1)
                p1s(length(p1s)+1) = p1s(maxIndex);
                p2s(length(p2s)+1) = p2s(maxIndex)+1;
                g1s(length(g1s)+1) = g1s(maxIndex);
                g2s(length(g2s)+1) = g2s(maxIndex);

                numAdded = numAdded + 1;
                edgeStates(length(edgeStates)+1) = length(p1s);
```

```
            end
            if (numFound4 == 0)
                p1s(length(p1s)+1) = p1s(maxIndex);
                p2s(length(p2s)+1) = p2s(maxIndex)-1;
                g1s(length(g1s)+1) = g1s(maxIndex);
                g2s(length(g2s)+1) = g2s(maxIndex);

                numAdded = numAdded + 1;
                edgeStates(length(edgeStates)+1) = length(p1s);
            end
            if (numFound5 == 0)
                p1s(length(p1s)+1) = p1s(maxIndex);
                p2s(length(p2s)+1) = p2s(maxIndex);
                g1s(length(g1s)+1) = abs(g1s(maxIndex)-1);
                g2s(length(g2s)+1) = g2s(maxIndex);

                numAdded = numAdded + 1;
                edgeStates(length(edgeStates)+1) = length(p1s);
            end
            if (numFound4 == 0)
                p1s(length(p1s)+1) = p1s(maxIndex);
                p2s(length(p2s)+1) = p2s(maxIndex);
                g1s(length(g1s)+1) = g1s(maxIndex);
                g2s(length(g2s)+1) = abs(g2s(maxIndex)-1);

                numAdded = numAdded + 1;
                edgeStates(length(edgeStates)+1) = length(p1s);
            end
            edgeStates(maxEdgeIndex) = [];
            edgeSize = edgeSize - 1;
        end
    end
%     if maxM == 4
%         break;
%     end
end
```

process.m

```
pSize = max([max(data(:,1))+1,max(data(:,2))+1]);
pfin(1:pSize,1:pSize) = zeros(pSize,pSize);
for i=1:pSize
    for j=1:pSize
        for l=1:length(data(:,5))
            if ((data(l,1)+1) == i) && ((data(l,2)+1) == j)
                pfin(i,j) = pfin(i,j) + data(l,5);
            end
        end
    end
end
```

processGenemRNA.m

```
size = sqrt(length(data));
maxNum = max(max(data(:,1)),max(data(:,2)));
pfin = zeros(maxNum+1,maxNum+1);
for i=1:length(data)
    pfin(data(i,1)+1,data(i,2)+1) = data(i,3);
```

```
end
```

processRepress.m

```
pSize = max([max(data(:,1))+1,max(data(:,2))+1,max(data(:,3))+1]);
pfin(1:pSize,1:pSize,1:pSize) = zeros(pSize,pSize,pSize);
for i=1:pSize
    for j=1:pSize
        for k=1:pSize
            for l=1:length(data(:,4))
                if ((data(l,1)+1) == i) && ((data(l,2)+1) == j) &&
((data(l,3)+1) == k)
                    pfin(i,j,k) = pfin(i,j,k) + data(l,4);
                end
            end
        end
    end
end
```

Appendix B: BLAS & ATLAS Source Code

FSPtest.c

```
#include <stdio.h>
#include <stdlib.h>
#include "matrices.h"

double *addStates(int size, int kp, int gammap, int km, int gammam)
{
  int i;
  int m,p;
  int side = size*size;
  double *A = calloc(side*side,sizeof(double));
  for(i=0;i<side;i++){
    m = i/size;
    p = i%size;
    *(A+i*side+i) = (double)(-km - m*gammam - m*kp - p*gammap);
    if(i != (side-1)){
      *(A+(i+1)*side+i) = (double)(m*kp);
      *(A+i*side+i+1) = (double)(((p+1)%size)*gammap);
    }
    if((m > 0) && (m < size)){
      *(A+(i-size)*side+i) = (double)(gammam*m);
      *(A+i*side+(i-size)) = (double)km;
    }
  }
  return A;
}

int main(int argc, char **argv)
{
  int kp = 4;
  int gammap = 1;
  int km = 4;
  int gammam = 1;
  int i,j;
  int gamma = 1;
  int tf = 60*60*7;
  int currentSize = 2;
  int side;
  int p = 10;
  int q = 10;
  int leap = 0;
  double targetError = 1e-6;
  double tempError = 1.0;

  double *reacMat;
  double *probVec;
  FILE *fp;

  while(1){
    printf("%d\n",currentSize);
    side = currentSize*currentSize;
    reacMat = addStates(currentSize,kp,gammap,km,gammam);
    fp = fopen("../../MATLAB/reacMat.txt","w+");
    for(i=0;i<side;i++){
```

```
      for(j=0;j<side;j++){
      fprintf(fp,"%4.10f ",*(reacMat+i*side+j));
      }
      fprintf(fp,"\n");
    }
    fclose(fp);
    probVec = calloc(side,sizeof(double));
    *probVec = 1.0;

    matScale(side,reacMat,tf);
    Pade(side,reacMat,p,q);
    vecMult(side,reacMat,probVec);
    reacMat = realloc(reacMat,sizeof(double)*side);
    tempError = 1.0 - vecNorm(side,reacMat);
    printf("error - %4.10f -> %4.10f\n",tempError,targetError);
    if(tempError < targetError){
      break;
    }else{
      currentSize += tempError*leap + 1;
      free(reacMat);
      free(probVec);
    }
  }
  printf("\n\n");

  fp = fopen("../../MATLAB/results.txt","w+");

  for(i=0;i<side;i++){
    printf("%4.10f\n",*(reacMat+i));
    fprintf(fp,"%4.10f\n",*(reacMat+i));
  }
  fclose(fp);

  printf("\n%d\n",currentSize);
  printf("\n%f\n",tempError);

  free(probVec);
  free(reacMat);
}
```

MandPModel.c

```
#include <stdio.h>
#include <stdlib.h>
#include "matrices.h"

double *addStates(int size, int kp, int gammap, int km, int gammam)
{
  int i;
  int m,p;
  int side = size*size;
  double *A = calloc(side*side,sizeof(double));
  for(i=0;i<side;i++){
    m = i/size;
    p = i%size;
    *(A+i*side+i) = (double)(-km - m*gammam - m*kp - p*gammap);
    if(i != (side-1)){
```

```
      *(A+(i+1)*side+i) = (double)(m*kp);
      *(A+i*side+i+1) = (double)(((p+1)%size)*gammap);
    }
    if((m > 0) && (m < size)){
      *(A+(i-size)*side+i) = (double)(gammam*m);
      *(A+i*side+(i-size)) = (double)km;
    }
  }
  return A;
}

int main(int argc, char **argv)
{
  int kp = 2;
  int gammap = 1;
  int km = 2;
  int gammam = 1;
  int i,j;
  int gamma = 1;
  int tf = 10;
  int currentSize = 2;
  int side;
  int p = 10;
  int q = 10;
  double targetError = 10e-7;
  double tempError = 1.0;

  double *reacMat;
  double *probVec;
  //FILE *fp;

  while(1){
    printf("%d\n",currentSize);
    side = currentSize*currentSize;
    reacMat = addStates(currentSize,kp,gammap,km,gammam);
    /*fp = fopen("../MATLAB/reacMat.txt","w+");
    for(i=0;i<side;i++){
      for(j=0;j<side;j++){
      fprintf(fp,"%4.10f ",*(reacMat+i*side+j));
      }
      fprintf(fp,"\n");
    }
    fclose(fp);*/
    probVec = calloc(side,sizeof(double));
    *probVec = 1.0;

    matScale(side,reacMat,tf);
    Pade(side,reacMat,p,q);
    vecMult(side,reacMat,probVec);
    reacMat = realloc(reacMat,sizeof(double)*side);
    tempError = 1.0 - vecNorm(side,reacMat);
    printf("error - %4.10f -> %4.10f\n",tempError,targetError);
    free(reacMat);
    free(probVec);
    if(tempError < targetError){
      break;
    }else{
```

```
      currentSize++;
    }
  }

  /*fp = fopen("../MATLAB/results.txt","w+");

  for(i=0;i<side;i++){
    printf("%4.10f\n",*(reacMat+i));
    fprintf(fp,"%4.10f\n",*(reacMat+i));
  }
  fclose(fp);*/

  printf("\n%d\n",currentSize);
  printf("\n%f\n",tempError);
}
```

matrices.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "matrices.h"

double factorial(int n)
{
  double result = 1;
  while(n > 1){
    result *= (double)n;
    n--;
  }
  return result;
}

double Norm(int lda, double *A)
{
  int i,j;
  double temp,result;
  for(i=0;i<lda;i++){
    temp = 0.0;
    for(j=0;j<lda;j++){
      temp += abs(*(A+(lda*j+i)));
    }
    if(i == 0){
      result = temp;
    }else{
      if(temp > result){
      result = temp;
      }
    }
  }
  return result;
}

double vecNorm(int lda, double *A)
{
  int i;
  double result = 0.0;
```

```
  for(i=0;i<lda;i++){
    result += *(A+i);
  }
  return result;
}

void rowSwitch(int lda, double *A, int row1, int row2)
{
  int i;
  double temp;
  for(i=0;i<lda;i++){
    temp = *(A+(lda*row1+i));
    *(A+(lda*row1+i)) = *(A+(lda*row2+i));
    *(A+(lda*row2+i)) = temp;
  }
}

void printMat(int lda, double *A)
{
  int i,j;
  for(i=0;i<lda;i++){
    printf("[");
    for(j=0;j<lda;j++){
      printf("%4.10f ",*(A+(lda*i+j)));
    }
    printf("]\n");
  }
  printf("\n");
}

/* A*B = A */
void matMult(int lda, double *A, double *B)
{
  int i,j,k;
  double *C = calloc(lda*lda, sizeof(double));
  for(i=0;i<lda;i++){
    for(j=0;j<lda;j++){
      for(k=0;k<lda;k++){
      *(C+(lda*i+j)) += *(A+(lda*i+k)) * *(B+(lda*k+j));
      }
    }
  }
  memcpy(A,C,sizeof(double)*lda*lda);
  free(C);
}

/* A+B=A  */
void matAdd(int lda, double *A, double *B)
{
  int i,j;
  for(i=0;i<lda;i++){
    for(j=0;j<lda;j++){
      *(A+(lda*i+j)) += *(B+(lda*i+j));
    }
  }
}
```

```
/* scalar*A = A */
void matScale(int lda, double *A, double scalar)
{
  int i,j;
  for(i=0;i<lda;i++){
    for(j=0;j<lda;j++){
      *(A+(lda*i+j)) *= scalar;
    }
  }
}

void vecMult(int lda, double *A, double *x)
{
  int i,j;
  double *answer = calloc(lda,sizeof(double));

  for(i=0;i<lda;i++){
    for(j=0;j<lda;j++){
      *(answer+i) += *(A+(lda*i+j)) * *(x+j);
    }
  }
  memcpy(A,answer,sizeof(double)*lda);
  free(answer);
}

void matInvert(int lda, double *A)
{
  int i,j,k;
  double multterm,topelement;
  double *B = calloc(lda*lda,sizeof(double));

  for(i=0;i<(lda*lda);i++){
    *(B+i) = *(A+i);
    if(i%(lda+1) == 0){
      *(A+i) = 1.0;
    }else{
      *(A+i) = 0.0;
    }
  }

  for(i=0;i<lda;i++){
    topelement = *(B+(i*lda+i));
    if(topelement == 0){
      if( (i+1) < lda){
      rowSwitch(lda,A,i,i+1);
      rowSwitch(lda,B,i,i+1);
      }else{
      printf("Matrix may be singular...\n");
      break;
      }
    }else{
      for(j=0;j<lda;j++){
      *(B+(i*lda+j)) = *(B+(i*lda+j))/topelement;
      *(A+(i*lda+j)) = *(A+(i*lda+j))/topelement;
      }
    }
```

```
    for(j=0;j<lda;j++){
      if(j == i){
      j++;
      }
      if (j != lda){
      multterm = *(B+(j*lda+i));
      multterm *= -1.0;
      for(k=0;k<lda;k++){
        *(B+(j*lda+k)) += multterm * *(B+(i*lda+k));
        *(A+(j*lda+k)) += multterm * *(A+(i*lda+k));
      }
      }
    }
  }
  free(B);
}

double *NBlock(int lda, double *A, int p, int q)
{
  int j,i;
  double scalar;
  double *N = calloc(lda*lda, sizeof(double));
  double *Atemp = calloc(lda*lda, sizeof(double));
  double *temp = calloc(lda*lda, sizeof(double));

  for(j=0;j<lda;j++){
    *(N+(j*(lda+1))) = 1.0;
    *(Atemp+(j*(lda+1))) = 1.0;
  }
  for(j=1;j<p;j++){
    scalar = ( factorial(p+q-j)*factorial(p) )/(
factorial(p+q)*factorial(j)*factorial(p-j) );
    matMult(lda,Atemp,A);
    for(i=0;i<(lda*lda);i++){
      *(temp+i) = *(Atemp+i);
    }
    matScale(lda,temp,scalar);
    matAdd(lda,N,temp);
  }
  free(Atemp);
  free(temp);
  return N;
}

double *DBlock(int lda, double *A, int p, int q)
{
  int j,i;
  double scalar;
  double *D = calloc(lda*lda, sizeof(double));
  double *Atemp = calloc(lda*lda, sizeof(double));
  double *temp = calloc(lda*lda, sizeof(double));

  for(j=0;j<lda;j++){
    *(D+(j*(lda+1))) = 1.0;
    *(Atemp+(j*(lda+1))) = 1.0;
  }
  for(j=1;j<q;j++){
```

```
    scalar = ( factorial(p+q-j)*factorial(q) )/(
factorial(p+q)*factorial(j)*factorial(q-j) );
    matMult(lda,Atemp,A);
    if(j%2 == 0){
      for(i=0;i<(lda*lda);i++){
      *(temp+i) = *(Atemp+i);
      }
    }else{
      for(i=0;i<(lda*lda);i++){
      *(temp+i) = -*(Atemp+i);
      }
    }
    matScale(lda,temp,scalar);
    matAdd(lda,D,temp);
  }
  free(Atemp);
  free(temp);
  return D;
}

void Pade(int lda, double *A, int p, int q)
{
  double *N;
  double *D;
  double norm;
  double m = 0.0;

  norm = Norm(lda,A);

  if(norm > 1.0){
    m = 1.0;
    while(m < norm){
      m *= 2.0;
    }
    matScale(lda,A,1.0/m);
  }

  N = NBlock(lda,A,p,q);
  D = DBlock(lda,A,p,q);
  matInvert(lda,D);
  matMult(lda,D,N);

  while(m > 1){
    m /= 2;
    matMult(lda,D,D);
  }

  free(N);
  memcpy(A,D,sizeof(double)*lda*lda);

  free(D);
}
```

matrices.h

```
/*matrices.h*/

double factorial(int);
double Norm(int, double *);
double vecNorm(int, double *);
void rowSwitch(int, double *, int, int);
void printMat(int, double *);
void matMult(int, double *, double *);
void matAdd(int, double *, double *);
void matScale(int, double *, double);
void vecMult(int, double *, double *);
void matInvert(int, double *);
double *NBlock(int, double *, int, int);
double *DBlock(int, double *, int, int);
void Pade(int, double *, int, int);
```

matTest.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "matrices.h"

double *addStates(int size, int kp, int gamma)
{
  int i;
  double *A = calloc(size*size,sizeof(double));
  *(A) = -kp;
  *(A+1) = gamma;
  for(i=1;i<size;i++){
    *(A+(size*i+i)) = -kp-(gamma*i);
    *(A+(size*i+i-1)) = kp;
    if(i != (size-1)){
      *(A+(size*i+i+1)) = (i+1)*gamma;
    }
  }
  return A;
}

int main(int argc, char **argv)
{
  int kp = 6;
  double tf = 10.0;
  int gamma = 1;
  int currentSize = 4;
  int p = 10;
  int q = 10;

  double *A;
  double *B = calloc(currentSize*currentSize,sizeof(double));
  A = addStates(currentSize,kp,gamma);
  printf("A - %d\n",sizeof(A));
  printMat(currentSize,A);

  memcpy(B,A,sizeof(double)*currentSize*currentSize);
  printf("here\n");
```

```
  printf("B - %d\n",sizeof(B));
  printMat(currentSize,B);

  matScale(currentSize,A,tf);
  printf("A*tf\n");
  printMat(currentSize,A);

  Pade(currentSize,A,p,q);
  printf("exponentiated A*tf\n");
  printMat(currentSize,A);
  free(A);
}
```

serialBlasFSPDouble.c

```
#include <stdio.h>
#include <stdlib.h>
#include <cblas.h>
#include <time.h>

/*select device*/


double scalar_N[10];
double scalar_D[10];

void printMat(int lda, double *A)
{
      int i,j;
      for(i=0;i<lda;i++){
            printf("[");
            for(j=0;j<lda;j++){
                  printf("%4.10f ",*(A+(lda*i+j)));
            }
            printf("]\n");
      }
      printf("\n");
}

double *addStates(int size, int kp, int gammap, int km, int gammam)
{
  int i;
  int m,p;
  int side = size*size;
  double *A = calloc(side*side,sizeof(double));
  for(i=0;i<side;i++){
    m = i/size;
    p = i%size;
    *(A+i*side+i) = (double)(-km - m*gammam - m*kp - p*gammap);
    if(i != (side-1)){
      *(A+(i+1)*side+i) = (double)(m*kp);
      *(A+i*side+i+1) = (double)(((p+1)%size)*gammap);
    }
    if((m > 0) && (m < size)){
      *(A+(i-size)*side+i) = (double)(gammam*m);
      *(A+i*side+(i-size)) = (double)km;
```

```
    }
  }
  return A;
}

void NBlock(int N, double *d_N, double *d_reacMat, double *d_tempMat, int p)
{
      int i;
      double *Atemp = (double *)calloc(N*N,sizeof(double));;
      cblas_dscal(N*N,scalar_N[0],d_N,1);
      for(i=1;i<p;i++){

      cblas_dgemm(CblasRowMajor,CblasNoTrans,CblasNoTrans,N,N,N,1.0,d_tempMat
,N,d_reacMat,N,0.0,Atemp,N);
            cblas_dcopy(N*N,Atemp,1,d_tempMat,1);
            cblas_daxpy(N*N,scalar_N[i],d_tempMat,1,d_N,1);
      }
      free(Atemp);
}

void DBlock(int N, double *d_D, double *d_reacMat, double *d_tempMat, int p)
{
      int i;
      double *Atemp = (double *)calloc(N*N,sizeof(double));
      cblas_dscal(N,scalar_D[0],d_D,N+1);
      for(i=1;i<p;i++){
            cblas_dgemm(CblasRowMajor,CblasNoTrans,CblasNoTrans,N,N,N,-
1.0,d_tempMat,N,d_reacMat,N,0.0,Atemp,N);
            cblas_dcopy(N*N,Atemp,1,d_tempMat,1);
            cblas_daxpy(N*N,scalar_D[i],d_tempMat,1,d_D,1);
      }
      free(Atemp);
}

void matInvert(int N, double *d_A, double *d_tempMat)
{
      int i,j;
      double multterm, topelement;

      for(i=0;i<N;i++){
            topelement = *(d_A+(i*N+i));
            //cublasGetVector(1,sizeof(double),(d_A+(i*N+i)),1,topelement,1);
            if(topelement == 0){
                  printf("Matrix may be singular.\n");
            }else{
                  cblas_dscal(N,(1.0 / topelement),(d_A+(i*N)),1);
                  cblas_dscal(N,(1.0 / topelement),(d_tempMat+(i*N)),1);
            }
            for(j=0;j<N;j++){
                  if(j == i){
                        j++;
                  }
                  if(j != N){
                        multterm = *(d_A+(j*N+i));

      //cublasGetVector(1,sizeof(double),(d_A+(j*N+i)),1,multterm,1);
                        multterm *= -1.0;
```

```
                        cblas_daxpy(N,multterm,(d_A+(i*N)),1,(d_A+(j*N)),1);

        cblas_daxpy(N,multterm,(d_tempMat+(i*N)),1,(d_tempMat+(j*N)),1);
                }
            }
        }
        cblas_dswap(N*N,d_A,1,d_tempMat,1);
}

void FSP(int kp, int gammap, int km, int gammam, int p, int q, int startProb,
double targetError, int tf)
{

        int i;
        int currentSize = 60;
        int N;
        int n2;
        double tempNorm, norm, m;
        double tempError;
        /*double *reacMat;
        double *probVec;
        double *identMat;*/
        double *d_reacMat;
        double *d_probVec;
        double *d_D;
        double *d_N;
        double *d_tempMat;

        time_t now;

        time(&now);

        printf("%s\n",ctime(&now));

        while(1){
              m = 0.0;
              N = currentSize*currentSize;
              n2 = N*N;
              d_reacMat = addStates(currentSize,kp,gammap,km,gammam);
              d_probVec = (double *)calloc(N,sizeof(double));
              d_tempMat = (double *)calloc(n2,sizeof(double));
              d_D = (double *)calloc(n2,sizeof(double));
              d_N = (double *)calloc(n2,sizeof(double));
              *(d_probVec+startProb) = 1.0;
              for(i=0;i<N;i++){
                    *(d_tempMat+(N*i)+i) = 1.0;
                    *(d_D+(N*i)+i) = 1.0;
                    *(d_N+(N*i)+i) = 1.0;
              }

              /* multiply by tf */
              cblas_dscal(n2,(double)tf,d_reacMat,1);

              /* find matrix norm */
              norm = 0.0;
              for(i=0;i<N;i++){
                    tempNorm = cblas_dnrm2(N,d_reacMat+i,N);
```

```
                if(tempNorm > norm){
                        norm = tempNorm;
                }
        }

        /* scaling */
        if(norm > 1.0){
                m = 1.0;
                while(m < norm){
                        m *= 2.0;
                }
                cblas_dscal(n2, 1.0/m, d_reacMat, 1);
        }

        /* find blocks */
        NBlock(N,d_N,d_reacMat,d_tempMat,p);
        cblas_dcopy(n2,d_D,1,d_tempMat,1);
        DBlock(N,d_D,d_reacMat,d_tempMat,p);

        free(d_tempMat);
        d_tempMat = (double *)calloc(n2,sizeof(double));

        for(i=0;i<N;i++){
                *(d_tempMat+(N*i)+i) = 1.0;
        }

        /* Invert D Block */
        matInvert(N,d_D,d_tempMat);

        /* Pade approximation result */

    cblas_dgemm(CblasRowMajor,CblasNoTrans,CblasNoTrans,N,N,N,1.0,d_D,N,d_N
,N,0.0,d_tempMat,N);
        cblas_dcopy(n2,d_tempMat,1,d_D,1);

        /* squaring */
        while(m > 1){
                m /= 2;

    cblas_dgemm(CblasRowMajor,CblasNoTrans,CblasNoTrans,N,N,N,1.0,d_D,N,d_D
,N,0.0,d_tempMat,N);
                cblas_dcopy(n2,d_tempMat,1,d_D,1);
        }

        /*multiply probability vector*/

    cblas_dgemv(CblasRowMajor,CblasTrans,N,N,1.0,d_D,N,d_probVec,1,0.0,d_te
mpMat,1);
        cblas_dcopy(N,d_tempMat,N,d_probVec,1);

        /* compute error */
        tempError = 1.0 - cblas_dasum(N,d_probVec,1);

        //if(tempError < targetError){
        break;
        if(currentSize > 100){
                break;
```

```
            }else{
                  free(d_reacMat);
                  free(d_probVec);
                  free(d_D);
                  free(d_N);
                  free(d_tempMat);
                  currentSize++;
            }
      }

      time(&now);

      printf("%s\n",ctime(&now));

      printf("\ncurrentSize - %d\n",currentSize);
      printf("error - %4.10f -> %4.10f\n\n",tempError,targetError);

      for(i=0;i<N;i++){
            printf("%f\n",*(d_probVec+i));
      }

      free(d_reacMat);
      free(d_probVec);
      free(d_D);
      free(d_N);
      free(d_tempMat);
}

double factorial(int n)
{
  double result = 1.0;
  while(n > 1){
    result *= (double)n;
    n--;
  }
  return result;
}

int main(int argc, char* argv[]) {
      int kp = 12;
      int gammap = 1;
      int km = 12;
      int gammam = 1;
      int p = 10;
      int q = 10;
      int tf  = 10;
      int j;
      int startProb = 0;
      double targetError = 10e-7;

      for(j=0;j<p;j++){
            scalar_N[j] = ( factorial(p+q-j)*factorial(p) )/(
factorial(p+q)*factorial(j)*factorial(p-j) );
            scalar_D[j] = ( factorial(p+q-j)*factorial(q) )/(
factorial(p+q)*factorial(j)*factorial(q-j) );
      }
```

```
	FSP(kp, gammap, km, gammam, p, q, startProb, targetError, tf);
	return EXIT_SUCCESS;
}
```

serialBlasFSPDoubleSweep.c

```
#include <stdio.h>
#include <stdlib.h>
#include <cblas.h>
#include <time.h>

/*select device*/


double scalar_N[10];
double scalar_D[10];

void printMat(int lda, double *A)
{
	int i,j;
	for(i=0;i<lda;i++){
		printf("[");
		for(j=0;j<lda;j++){
			printf("%4.10f ",*(A+(lda*i+j)));
		}
		printf("]\n");
	}
	printf("\n");
}

double *addStates(int size, int kp, int gammap, int km, int gammam)
{
  int i;
  int m,p;
  int side = size*size;
  double *A = calloc(side*side,sizeof(double));
  for(i=0;i<side;i++){
    m = i/size;
    p = i%size;
    *(A+i*side+i) = (double)(-km - m*gammam - m*kp - p*gammap);
    if(i != (side-1)){
      *(A+(i+1)*side+i) = (double)(m*kp);
      *(A+i*side+i+1) = (double)(((p+1)%size)*gammap);
    }
    if((m > 0) && (m < size)){
      *(A+(i-size)*side+i) = (double)(gammam*m);
      *(A+i*side+(i-size)) = (double)km;
    }
  }
  return A;
}

void NBlock(int N, double *d_N, double *d_reacMat, double *d_tempMat, int p)
{
	int i;
	double *Atemp = (double *)calloc(N*N,sizeof(double));;
	cblas_dscal(N*N,scalar_N[0],d_N,1);
```

```
	for(i=1;i<p;i++){

	cblas_dgemm(CblasRowMajor,CblasNoTrans,CblasNoTrans,N,N,N,1.0,d_tempMat
,N,d_reacMat,N,0.0,Atemp,N);
		cblas_dcopy(N*N,Atemp,1,d_tempMat,1);
		cblas_daxpy(N*N,scalar_N[i],d_tempMat,1,d_N,1);
	}
	free(Atemp);
}

void DBlock(int N, double *d_D, double *d_reacMat, double *d_tempMat, int p)
{
	int i;
	double *Atemp = (double *)calloc(N*N,sizeof(double));
	cblas_dscal(N,scalar_D[0],d_D,N+1);
	for(i=1;i<p;i++){
		cblas_dgemm(CblasRowMajor,CblasNoTrans,CblasNoTrans,N,N,N,-
1.0,d_tempMat,N,d_reacMat,N,0.0,Atemp,N);
		cblas_dcopy(N*N,Atemp,1,d_tempMat,1);
		cblas_daxpy(N*N,scalar_D[i],d_tempMat,1,d_D,1);
	}
	free(Atemp);
}

void matInvert(int N, double *d_A, double *d_tempMat)
{
	int i,j;
	double multterm, topelement;

	for(i=0;i<N;i++){
		topelement = *(d_A+(i*N+i));
		//cublasGetVector(1,sizeof(double),(d_A+(i*N+i)),1,topelement,1);
		if(topelement == 0){
			printf("Matrix may be singular.\n");
		}else{
			cblas_dscal(N,(1.0 / topelement),(d_A+(i*N)),1);
			cblas_dscal(N,(1.0 / topelement),(d_tempMat+(i*N)),1);
		}
		for(j=0;j<N;j++){
			if(j == i){
				j++;
			}
			if(j != N){
				multterm = *(d_A+(j*N+i));

	//cublasGetVector(1,sizeof(double),(d_A+(j*N+i)),1,multterm,1);
				multterm *= -1.0;
				cblas_daxpy(N,multterm,(d_A+(i*N)),1,(d_A+(j*N)),1);

	cblas_daxpy(N,multterm,(d_tempMat+(i*N)),1,(d_tempMat+(j*N)),1);
			}
		}
	}
	cblas_dswap(N*N,d_A,1,d_tempMat,1);
}
```

```
void FSP(int kp, int gammap, int km, int gammam, int p, int q, int startProb,
double targetError, long int tf, int currentSize)
{

	int i;
	int N;
	int n2;
	double tempNorm, norm, m;
	double tempError;
	/*double *reacMat;
	double *probVec;
	double *identMat;*/
	double *d_reacMat;
	double *d_probVec;
	double *d_D;
	double *d_N;
	double *d_tempMat;

	time_t now;

		m = 0.0;
		N = currentSize*currentSize;
		n2 = N*N;
		d_reacMat = addStates(currentSize,kp,gammap,km,gammam);

		time(&now);

		printf("%s\n",ctime(&now));

		d_probVec = (double *)calloc(N,sizeof(double));
		d_tempMat = (double *)calloc(n2,sizeof(double));
		d_D = (double *)calloc(n2,sizeof(double));
		d_N = (double *)calloc(n2,sizeof(double));
		*(d_probVec+startProb) = 1.0;
		for(i=0;i<N;i++){
			*(d_tempMat+(N*i)+i) = 1.0;
			*(d_D+(N*i)+i) = 1.0;
			*(d_N+(N*i)+i) = 1.0;
		}

		/* multiply by tf */
		cblas_dscal(n2,(double)tf,d_reacMat,1);

		/* find matrix norm */
		norm = 0.0;
		for(i=0;i<N;i++){
			tempNorm = cblas_dnrm2(N,d_reacMat+i,N);
			if(tempNorm > norm){
				norm = tempNorm;
			}
		}

		/* scaling */
		if(norm > 1.0){
			m = 1.0;
			while(m < norm){
				m *= 2.0;
```

```
                }
                cblas_dscal(n2, 1.0/m, d_reacMat, 1);
        }

        /* find blocks */
        NBlock(N,d_N,d_reacMat,d_tempMat,p);
        cblas_dcopy(n2,d_D,1,d_tempMat,1);
        DBlock(N,d_D,d_reacMat,d_tempMat,p);

        free(d_tempMat);
        d_tempMat = (double *)calloc(n2,sizeof(double));

        for(i=0;i<N;i++){
                *(d_tempMat+(N*i)+i) = 1.0;
        }

        /* Invert D Block */
        matInvert(N,d_D,d_tempMat);

        /* Pade approximation result */

    cblas_dgemm(CblasRowMajor,CblasNoTrans,CblasNoTrans,N,N,N,1.0,d_D,N,d_N
,N,0.0,d_tempMat,N);
        cblas_dcopy(n2,d_tempMat,1,d_D,1);

        /* squaring */
        while(m > 1){
                m /= 2;

    cblas_dgemm(CblasRowMajor,CblasNoTrans,CblasNoTrans,N,N,N,1.0,d_D,N,d_D
,N,0.0,d_tempMat,N);
                cblas_dcopy(n2,d_tempMat,1,d_D,1);
        }

        /*multiply probability vector*/

    cblas_dgemv(CblasRowMajor,CblasTrans,N,N,1.0,d_D,N,d_probVec,1,0.0,d_te
mpMat,1);
        cblas_dcopy(N,d_tempMat,N,d_probVec,1);

        /* compute error */
        tempError = 1.0 - cblas_dasum(N,d_probVec,1);

    time(&now);

    printf("%s\n",ctime(&now));

    printf("\ncurrentSize - %d\n",currentSize);
    printf("error - %4.10f -> %4.10f\n\n",tempError,targetError);

    for(i=0;i<N;i++){
        printf("%f\n",*(d_probVec+i));
    }

    free(d_reacMat);
    free(d_probVec);
    free(d_D);
```

```
      free(d_N);
      free(d_tempMat);
}

double factorial(int n)
{
  double result = 1.0;
  while(n > 1){
    result *= (double)n;
    n--;
  }
  return result;
}

int main(int argc, char* argv[]) {
      int kp;
      int currentSize;
      int gammap = 1;
      int km;
      int gammam = 1;
      int p = 10;
      int q = 10;
      long int tf;
      int j;
      int startProb = 0;
      double targetError = 10e-7;

      if(argc != 5){
            printf("usage - %s [currentSize] [km] [kp] [tf]\n",argv[0]);
            return EXIT_FAILURE;
      }
      sscanf(argv[1],"%d",&currentSize);
      sscanf(argv[2],"%d",&km);
      sscanf(argv[3],"%d",&kp);
      sscanf(argv[4],"%ld",&tf);

      for(j=0;j<p;j++){
            scalar_N[j] = ( factorial(p+q-j)*factorial(p) )/(
factorial(p+q)*factorial(j)*factorial(p-j) );
            scalar_D[j] = ( factorial(p+q-j)*factorial(q) )/(
factorial(p+q)*factorial(j)*factorial(q-j) );
      }

      FSP(kp, gammap, km, gammam, p, q, startProb, targetError, tf,
currentSize);
      return EXIT_SUCCESS;
}
```

serialBlasFSPFloat.c

```
#include <stdio.h>
#include <stdlib.h>
#include <cblas.h>
#include <time.h>

/*select device*/
```

```
float scalar_N[10];
float scalar_D[10];

void printMat(int lda, float *A)
{
      int i,j;
      for(i=0;i<lda;i++){
            printf("[");
            for(j=0;j<lda;j++){
                  printf("%4.10f ",*(A+(lda*i+j)));
            }
            printf("]\n");
      }
      printf("\n");
}

float *addStates(int size, int kp, int gammap, int km, int gammam)
{
  int i;
  int m,p;
  int side = size*size;
  float *A = calloc(side*side,sizeof(float));
  for(i=0;i<side;i++){
    m = i/size;
    p = i%size;
    *(A+i*side+i) = (float)(-km - m*gammam - m*kp - p*gammap);
    if(i != (side-1)){
      *(A+(i+1)*side+i) = (float)(m*kp);
      *(A+i*side+i+1) = (float)(((p+1)%size)*gammap);
    }
    if((m > 0) && (m < size)){
      *(A+(i-size)*side+i) = (float)(gammam*m);
      *(A+i*side+(i-size)) = (float)km;
    }
  }
  return A;
}

void NBlock(int N, float *d_N, float *d_reacMat, float *d_tempMat, int p)
{
      int i;
      float *Atemp = (float *)calloc(N*N,sizeof(float));;
      cblas_sscal(N,scalar_N[0],d_N,N+1);
      for(i=1;i<p;i++){

      cblas_sgemm(CblasRowMajor,CblasNoTrans,CblasNoTrans,N,N,N,1.0,d_tempMat
,N,d_reacMat,N,0.0,Atemp,N);
            cblas_scopy(N*N,Atemp,1,d_tempMat,1);
            cblas_saxpy(N*N,scalar_N[i],d_tempMat,1,d_N,1);
      }
      free(Atemp);
}

void DBlock(int N, float *d_D, float *d_reacMat, float *d_tempMat, int p)
{
      int i;
```

```
	float *Atemp = (float *)calloc(N*N,sizeof(float));
	cblas_sscal(N,scalar_D[0],d_D,N+1);
	for(i=1;i<p;i++){
		cblas_sgemm(CblasRowMajor,CblasNoTrans,CblasNoTrans,N,N,N,-
1.0,d_tempMat,N,d_reacMat,N,0.0,Atemp,N);
		cblas_scopy(N*N,Atemp,1,d_tempMat,1);
		cblas_saxpy(N*N,scalar_D[i],d_tempMat,1,d_D,1);
	}
	free(Atemp);
}

void matInvert(int N, float *d_A, float *d_tempMat)
{
	int i,j;
	float multterm, topelement;

	for(i=0;i<N;i++){
		topelement = *(d_A+(i*N+i));
		//cublasGetVector(1,sizeof(float),(d_A+(i*N+i)),1,topelement,1);
		if(topelement == 0){
			printf("Matrix may be singular.\n");
		}else{
			cblas_sscal(N,(1.0 / topelement),(d_A+(i*N)),1);
			cblas_sscal(N,(1.0 / topelement),(d_tempMat+(i*N)),1);
		}
		for(j=0;j<N;j++){
			if(j == i){
				j++;
			}
			if(j != N){
				multterm = *(d_A+(j*N+i));

	//cublasGetVector(1,sizeof(float),(d_A+(j*N+i)),1,multterm,1);
				multterm *= -1.0;
				cblas_saxpy(N,multterm,(d_A+(i*N)),1,(d_A+(j*N)),1);

	cblas_saxpy(N,multterm,(d_tempMat+(i*N)),1,(d_tempMat+(j*N)),1);
			}
		}
	}
	cblas_sswap(N*N,d_A,1,d_tempMat,1);
}

void FSP(int kp, int gammap, int km, int gammam, int p, int q, int startProb,
float targetError, int tf)
{

	int i;
	int currentSize = 2;
	int N;
	int n2;
	float tempNorm, norm, m;
	float tempError;
	/*float *reacMat;
	float *probVec;
	float *identMat;*/
	float *d_reacMat;
```

```
float *d_probVec;
float *d_D;
float *d_N;
float *d_tempMat;

time_t now;

while(1){
	m = 0.0;
	N = currentSize*currentSize;
	n2 = N*N;
	d_reacMat = addStates(currentSize,kp,gammap,km,gammam);
	d_probVec = (float *)calloc(N,sizeof(float));
	d_tempMat = (float *)calloc(n2,sizeof(float));
	d_D = (float *)calloc(n2,sizeof(float));
	d_N = (float *)calloc(n2,sizeof(float));
	*(d_probVec+startProb) = 1.0;
	for(i=0;i<N;i++){
		*(d_tempMat+(N*i)+i) = 1.0;
		*(d_D+(N*i)+i) = 1.0;
		*(d_N+(N*i)+i) = 1.0;
	}

	time(&now);

	printf("%s\n",ctime(&now));

	/* multiply by tf */
	cblas_sscal(n2,(float)tf,d_reacMat,1);

	/* find matrix norm */
	norm = 0.0;
	for(i=0;i<N;i++){
		tempNorm = cblas_snrm2(N,d_reacMat+i,N);
		if(tempNorm > norm){
			norm = tempNorm;
		}
	}

	/* scaling */
	if(norm > 1.0){
		m = 1.0;
		while(m < norm){
			m *= 2.0;
		}
		cblas_sscal(n2, 1.0/m, d_reacMat, 1);
	}


	/* find blocks */
	NBlock(N,d_N,d_reacMat,d_tempMat,p);

	/*printMat(N,d_N);*/

	cblas_scopy(n2,d_D,1,d_tempMat,1);
	DBlock(N,d_D,d_reacMat,d_tempMat,p);
```

```
            free(d_tempMat);
            d_tempMat = (float *)calloc(n2,sizeof(float));

            for(i=0;i<N;i++){
                  *(d_tempMat+(N*i)+i) = 1.0;
            }

            /* Invert D Block */
            matInvert(N,d_D,d_tempMat);

            /* Pade approximation result */

      cblas_sgemm(CblasRowMajor,CblasNoTrans,CblasNoTrans,N,N,N,1.0,d_D,N,d_N
,N,0.0,d_tempMat,N);
            cblas_scopy(n2,d_tempMat,1,d_D,1);

            /* squaring */
            while(m > 1){
                  m /= 2;

      cblas_sgemm(CblasRowMajor,CblasNoTrans,CblasNoTrans,N,N,N,1.0,d_D,N,d_D
,N,0.0,d_tempMat,N);
                  cblas_scopy(n2,d_tempMat,1,d_D,1);
            }

            /*multiply probability vector*/

      cblas_sgemv(CblasRowMajor,CblasTrans,N,N,1.0,d_D,N,d_probVec,1,0.0,d_te
mpMat,1);
            cblas_scopy(N,d_tempMat,N,d_probVec,1);

            /* compute error */
            tempError = 1.0 - cblas_sasum(N,d_probVec,1);

            time(&now);

            printf("%s\n",ctime(&now));

            printf("\ncurrentSize - %d\n",currentSize);
            printf("error - %4.10f -> %4.10f\n\n",tempError,targetError);

            if(tempError < targetError){
                  break;
            }else{
                  free(d_reacMat);
                  free(d_probVec);
                  free(d_D);
                  free(d_N);
                  free(d_tempMat);
                  currentSize++;
            }
      }

      free(d_reacMat);
      free(d_probVec);
      free(d_D);
      free(d_N);
```

```
      free(d_tempMat);

      printf("\ncurrentSize - %d\n",currentSize);
      printf("error - %4.10f -> %4.10f\n",tempError,targetError);
}

float factorial(int n)
{
  float result = 1.0;
  while(n > 1){
    result *= (float)n;
    n--;
  }
  return result;
}

int main(int argc, char* argv[]) {
      int kp = 2;
      int gammap = 1;
      int km = 2;
      int gammam = 1;
      int p = 10;
      int q = 10;
      int tf  = 10;
      int j;
      int startProb = 0;
      float targetError = 10e-7;

      for(j=0;j<p;j++){
            scalar_N[j] = ( factorial(p+q-j)*factorial(p) )/(
factorial(p+q)*factorial(j)*factorial(p-j) );
            scalar_D[j] = ( factorial(p+q-j)*factorial(q) )/(
factorial(p+q)*factorial(j)*factorial(q-j) );
      }

      FSP(kp, gammap, km, gammam, p, q, startProb, targetError, tf);
      return EXIT_SUCCESS;
}
```

SingleModel.c

```
#include <stdio.h>
#include <stdlib.h>
#include "matrices.h"

double *addStates(int size, int kp, int gamma)
{
  int i;
  double *A = calloc(size*size,sizeof(double));
  *(A) = -kp;
  *(A+1) = gamma;
  for(i=1;i<size;i++){
    *(A+(size*i+i)) = -kp-(gamma*i);
    *(A+(size*i+i-1)) = kp;
    if(i != (size-1)){
      *(A+(size*i+i+1)) = (i+1)*gamma;
    }
```

```
  }
  return A;
}

int main(int argc, char **argv)
{
  int kp = 2;
  int i;
  int gamma = 1;
  int tf = 10;
  int currentSize = 2;
  int p = 10;
  int q = 10;
  double targetError = 10e-3;
  double tempError = 1.0;

  double *reacMat;
  double *probVec;

  while(1){
    reacMat = addStates(currentSize,kp,gamma);
    probVec = calloc(currentSize,sizeof(double));
    *probVec = 1.0;

    matScale(currentSize,reacMat,tf);
    Pade(currentSize,reacMat,p,q);
    vecMult(currentSize,reacMat,probVec);
    tempError = 1.0 - vecNorm(currentSize,reacMat);
    if(tempError < targetError){
      break;
    }else{
      currentSize++;
      free(reacMat);
      free(probVec);
    }
    printf("\ncurrentSize - %d\n",currentSize);
    printf("error - %4.10f -> %4.10f\n",tempError,targetError);
  }
  printf("\n\n");
  for(i=0;i<currentSize;i++){
    printf("%4.10f\n",*(reacMat+i));
  }
  printf("\n%d\n",currentSize);
  printf("\n%f\n",tempError);

  free(probVec);
  free(reacMat);
}
```

Appendix C: CUBLAS source code

addVectors.cu

```
#include "stdio.h"
#include "stdlib.h"
#include "sys/time.h"

__global__ void add_arrays_gpu(float* in1, float* in2, float* out, int N) {
	int idx = blockIdx.x * blockDim.x + threadIdx.x;
	if(idx < N) {
		out[idx] = in1[idx] + in2[idx];
	}
}

int main(int argc, char* argv[]) {
	int N = 18;
	int i;
	float *a, *b, *c;
	float *a_d, *b_d, *c_d;
	int block_size = 8;
	int num_blocks, threads_per_block;
	if(argc != 3) {
		printf("Usage error: ./addVectors <vector_size> <block_size>\n");
		exit(1);
	}

	N = atoi(argv[1]);
	//block_size = atoi(argv[2]);
	num_blocks = atoi(argv[2]);
	threads_per_block = (N/num_blocks)+1;
	printf("Total threads: %d\n",(num_blocks*threads_per_block));
	//dim3 dimBlock(block_size);
	//dim3 dimGrid((N/dimBlock.x) + (!(N%dimBlock.x)?0:1));

	struct timeval tstart, tend;
	double start, end;

	a = (float*) malloc(N*sizeof(float));
	b = (float*) malloc(N*sizeof(float));
	c = (float*) malloc(N*sizeof(float));

	cudaMalloc((void **) & a_d, sizeof(float)*N);
	cudaMalloc((void **) & b_d, sizeof(float)*N);
	cudaMalloc((void **) & c_d, sizeof(float)*N);

	gettimeofday(&tstart,0);
	srand(tstart.tv_usec);

	for(i = 0; i < N; i ++) {
		a[i] = (rand() % 2000 - 1000) / 10;
		b[i] = (rand() % 2000 - 1000) / 10;
	}

	cudaMemcpy(a_d, a, sizeof(float)*N,cudaMemcpyHostToDevice);
	cudaMemcpy(b_d, a, sizeof(float)*N,cudaMemcpyHostToDevice);
```

```
	gettimeofday(&tstart,0);
	//add_arrays_gpu<<<dimGrid,dimBlock>>>(a_d,b_d,c_d,N);
	add_arrays_gpu<<<num_blocks,threads_per_block>>>(a_d,b_d,c_d,N);
	gettimeofday(&tend,0);
	cudaMemcpy(c,c_d,sizeof(float)*N,cudaMemcpyDeviceToHost);

	//for(i = 0; i < N; i ++) {
	//    printf(" c[%d]=%f\n",i,c[i]);
	//}

	start = tstart.tv_sec*1000000 + tstart.tv_usec;
	end = tend.tv_sec*1000000 + tend.tv_usec;
	printf("Vector add time: %f microseconds\n", (end - start));

	free(a);
	free(b);
	free(c);
	cudaFree(a_d);
	cudaFree(b_d);
	cudaFree(c_d);

	return 0;
}
```

blasFSPDouble.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <cublas.h>
#include <time.h>

/*select device*/


double scalar_N[10];
double scalar_D[10];

__host__ void setToTommyDevice() {
	cudaError_t setDeviceErrorCode;
	setDeviceErrorCode = cudaSetDevice(2);
	if(setDeviceErrorCode != cudaSuccess) {
		printf("Error setting to Tommy's
device:\n%s\n",cudaGetErrorString(setDeviceErrorCode));
		printf("Program will now exit.\n");
		exit(EXIT_FAILURE);
	}
}

__host__ void printMat(int lda, double *A)
{
	int i,j;
	for(i=0;i<lda;i++){
		printf("[");
		for(j=0;j<lda;j++){
			printf("%4.10f ",*(A+(lda*i+j)));
		}
```

```
            printf("]\n");
      }
      printf("\n");
}

__host__ double *addStates(int size, int kp, int gammap, int km, int gammam)
{
  int i;
  int m,p;
  int side = size*size;
  double *A = (double *)calloc(side*side,sizeof(double));
  for(i=0;i<side;i++){
    m = i/size;
    p = i%size;
    *(A+i*side+i) = (double)(-km - m*gammam - m*kp - p*gammap);
    if(i != (side-1)){
      *(A+(i+1)*side+i) = (double)(m*kp);
      *(A+i*side+i+1) = (double)(((p+1)%size)*gammap);
    }
    if((m > 0) && (m < size)){
      *(A+(i-size)*side+i) = (double)(gammam*m);
      *(A+i*side+(i-size)) = (double)km;
    }
  }
  return A;
}

__host__ void NBlock(int N, double *d_N, double *d_reacMat, double
*d_tempMat, int p)
{
      int i;
      double *Atemp;

      cublasAlloc(N*N, sizeof(double), (void**)&Atemp);
      cublasDscal(N,scalar_N[0],d_N,N+1);
      for(i=1;i<p;i++){

      cublasDgemm('n','n',N,N,N,1.0,d_tempMat,N,d_reacMat,N,0.0,Atemp,N);
            cublasDcopy(N*N,Atemp,1,d_tempMat,1);
            cublasDaxpy(N*N,scalar_N[i],d_tempMat,1,d_N,1);
      }
      cublasFree(Atemp);
}

__host__ void DBlock(int N, double *d_D, double *d_reacMat, double
*d_tempMat, int p)
{
      int i;
      double *Atemp;

      cublasAlloc(N*N, sizeof(double), (void**)&Atemp);
      cublasDscal(N,scalar_D[0],d_D,N+1);
      for(i=1;i<p;i++){
            cublasDgemm('n','n',N,N,N,-
1.0,d_tempMat,N,d_reacMat,N,0.0,Atemp,N);
            cublasDcopy(N*N,Atemp,1,d_tempMat,1);
            cublasDaxpy(N*N,scalar_D[i],d_tempMat,1,d_D,1);
```

```
	}
	cublasFree(Atemp);
}

__host__ void matInvert(int N, double *d_A, double *d_tempMat)
{
	int i,j;
	double *multterm, *topelement;

	multterm = (double *)calloc(1,sizeof(double));
	topelement = (double *)calloc(1,sizeof(double));

	for(i=0;i<N;i++){
		cublasGetVector(1,sizeof(double),(d_A+(i*N+i)),1,topelement,1);
		if(*topelement == 0){
			printf("Matrix may be singular.\n");
		}else{
			cublasDscal(N,(1.0 / *topelement),(d_A+(i*N)),1);
			cublasDscal(N,(1.0 / *topelement),(d_tempMat+(i*N)),1);
		}
		for(j=0;j<N;j++){
			if(j == i){
				j++;
			}
			if(j != N){

	cublasGetVector(1,sizeof(double),(d_A+(j*N+i)),1,multterm,1);
				*multterm *= -1.0;
				cublasDaxpy(N,*multterm,(d_A+(i*N)),1,(d_A+(j*N)),1);

	cublasDaxpy(N,*multterm,(d_tempMat+(i*N)),1,(d_tempMat+(j*N)),1);
			}
		}
	}
	cublasDswap(N*N,d_A,1,d_tempMat,1);
}

__host__ void FSP(int kp, int gammap, int km, int gammam, int p, int q, int
startProb, double targetError, int tf)
{
	cublasStatus status;

	int i;
	int currentSize = 50;
	int N;
	int n2;
	double tempNorm, norm, m;
	double tempError;
	double *reacMat;
	double *probVec;
	double *identMat;
	double *d_reacMat;
	double *d_probVec;
	double *d_D;
	double *d_N;
	double *d_tempMat;
```

```
	time_t now;

	time(&now);

	printf("%s\n",ctime(&now));

	status = cublasInit();
	if(status != CUBLAS_STATUS_SUCCESS){
		fprintf(stderr,"CUBLAS won't turn on...\n");
	}

	while(1){
		m = 0.0;
		N = currentSize*currentSize;
		n2 = N*N;
		reacMat = addStates(currentSize,kp,gammap,km,gammam);
		probVec = (double *)calloc(N,sizeof(double));
		identMat = (double *)calloc(n2,sizeof(double));
		*(probVec+startProb) = 1.0;
		for(i=0;i<N;i++){
			*(identMat+(N*i)+i) = 1.0;
		}

		/* reserve and copy matrices and vectors */
		status = cublasAlloc(n2, sizeof(double), (void**)&d_reacMat);
		if(status != CUBLAS_STATUS_SUCCESS){
			fprintf(stderr,"can't fit the reaction matrix on the
card!\n");
		}
		status = cublasAlloc(N, sizeof(double), (void**)&d_probVec);
		if(status != CUBLAS_STATUS_SUCCESS){
			fprintf(stderr,"can't fit Vi on the card!\n");
		}
		status = cublasAlloc(n2, sizeof(double), (void**)&d_D);
		if(status != CUBLAS_STATUS_SUCCESS){
			fprintf(stderr,"can't fit D on the card!\n");
		}
		status = cublasAlloc(n2, sizeof(double), (void**)&d_N);
		if(status != CUBLAS_STATUS_SUCCESS){
			fprintf(stderr,"can't fit N on the card!\n");
		}
		status = cublasAlloc(n2, sizeof(double), (void**)&d_tempMat);
		if(status != CUBLAS_STATUS_SUCCESS){
			fprintf(stderr,"can't fit temporary workspace on the
card!\n");
		}

		status = cublasSetVector(n2, sizeof(double), reacMat, 1,
d_reacMat, 1);
		if(status != CUBLAS_STATUS_SUCCESS){
			fprintf(stderr,"can't copy the reaction matrix to the
card!\n");
		}
		status  = cublasSetVector(N, sizeof(double), probVec, 1,
d_probVec, 1);
		if(status != CUBLAS_STATUS_SUCCESS){
			fprintf(stderr,"can't copy Vi to the card!\n");
```

```
            }
            status  = cublasSetVector(n2, sizeof(double), identMat, 1, d_N,
1);
            if(status != CUBLAS_STATUS_SUCCESS){
                  fprintf(stderr,"can't copy N to the card!\n");
            }
            status  = cublasSetVector(n2, sizeof(double), identMat, 1, d_D,
1);
            if(status != CUBLAS_STATUS_SUCCESS){
                  fprintf(stderr,"can't copy D to the card!\n");
            }
            status  = cublasSetVector(n2, sizeof(double), identMat, 1,
d_tempMat, 1);
            if(status != CUBLAS_STATUS_SUCCESS){
                  fprintf(stderr,"can't copy D to the card!\n");
            }

            /* multiply by tf */
            cublasDscal(n2,(double)tf,d_reacMat,1);

            /* find matrix norm */
            norm = 0.0;
            for(i=0;i<N;i++){
                  tempNorm = cublasDnrm2(N,d_reacMat+i,N);
                  status = cublasGetError();
                  if(status != CUBLAS_STATUS_SUCCESS){
                        printf("error - %d\n",status);
                  }
                  if(tempNorm > norm){
                        norm = tempNorm;
                  }
            }

            /* scaling */
            if(norm > 1.0){
                  m = 1.0;
                  while(m < norm){
                        m *= 2.0;
                  }
                  cublasDscal(n2, 1.0/m, d_reacMat, 1);
            }

            /* find blocks */
            NBlock(N,d_N,d_reacMat,d_tempMat,p);
            cublasDcopy(n2,d_D,1,d_tempMat,1);
            DBlock(N,d_D,d_reacMat,d_tempMat,p);

            status  = cublasSetVector(n2, sizeof(double), identMat, 1,
d_tempMat, 1);
            if(status != CUBLAS_STATUS_SUCCESS){
                  fprintf(stderr,"can't copy D to the card!\n");
            }

            /* Invert D Block */
            matInvert(N,d_D,d_tempMat);

            /* Pade approximation result */
```

```
        cublasDgemm('n','n',N,N,N,1.0,d_D,N,d_N,N,0.0,d_tempMat,N);
        cublasDcopy(n2,d_tempMat,1,d_D,1);

        /* squaring */
        while(m > 1){
              m /= 2;
              cublasDgemm('n','n',N,N,N,1.0,d_D,N,d_D,N,0.0,d_tempMat,N);
              cublasDcopy(n2,d_tempMat,1,d_D,1);
        }

        /*multiply probability vector*/
        cublasDgemv('t',N,N,1.0,d_D,N,d_probVec,1,0.0,d_tempMat,1);
        cublasDcopy(N,d_tempMat,N,d_probVec,1);

        /* compute error */
        tempError = 1.0 - cublasDasum(N,d_probVec,1);

        //if(tempError < targetError){
        break;
        if(currentSize > 100){
              break;
        }else{
              currentSize++;
              free(reacMat);
              free(identMat);
              free(probVec);
              status = cublasFree(d_reacMat);
              status = cublasFree(d_D);
              status = cublasFree(d_N);
              status = cublasFree(d_tempMat);
              status = cublasFree(d_probVec);
        }
}

time(&now);

printf("%s\n",ctime(&now));

printf("\ncurrentSize - %d\n",currentSize);
printf("error - %4.10f -> %4.10f\n\n",tempError,targetError);

/*cublasGetVector(N,sizeof(double),d_probVec,1,probVec,1);
for(i=0;i<N;i++){
      printf("%f\n",*(probVec+i));
}*/

free(reacMat);
free(identMat);
free(probVec);
status = cublasFree(d_reacMat);
status = cublasFree(d_D);
status = cublasFree(d_N);
status = cublasFree(d_tempMat);
status = cublasFree(d_probVec);

status = cublasShutdown();
if(status != CUBLAS_STATUS_SUCCESS){
```

```
            printf("can't turn it off\n");
      }
}

__host__ double factorial(int n)
{
  double result = 1.0;
  while(n > 1){
    result *= (double)n;
    n--;
  }
  return result;
}

int main(int argc, char* argv[]) {
      int kp = 10;
      int gammap = 1;
      int km = 10;
      int gammam = 1;
      int p = 10;
      int q = 10;
      int tf  = 10;
      int j;
      int startProb = 0;
      double targetError = 10e-7;

      for(j=0;j<p;j++){
            scalar_N[j] = ( factorial(p+q-j)*factorial(p) )/(
factorial(p+q)*factorial(j)*factorial(p-j) );
            scalar_D[j] = ( factorial(p+q-j)*factorial(q) )/(
factorial(p+q)*factorial(j)*factorial(q-j) );
      }

      setToTommyDevice();
      FSP(kp, gammap, km, gammam, p, q, startProb, targetError, tf);
      return EXIT_SUCCESS;
}
```

blasFSPDoublespinv.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <cublas.h>
#include <cblas.h>
#include <time.h>

/*select device*/


double scalar_N[10];
double scalar_D[10];

inline void __cudaSafeCall( int err, const char *file, const int line )
{
  do {
    if( err != 0) {
```

```
      fprintf(stderr, "cudaSafeCall() Runtime API error in file <%s>, line %i
: %s.\n",
              file, line, cudaGetErrorString((cudaError_t) err) );
      exit(-1);
    }
  } while (0);
}

#define SAFECALL(err) __cudaSafeCall(err, __FILE__, __LINE__)
#define CUDACHECK {cudaError_t error = cudaGetLastError(); if(error !=
0){fprintf(stderr, "Error code %d: %s file %s line
%i.\n",error,cudaGetErrorString(error),__FILE__,__LINE__);}}


void mathdispAI(const double *mat, int lda, int MAT_SIZE_h) {
    fprintf(stderr, "\n");
      int i,j;
    for (j=0;j<MAT_SIZE_h;j++)  {
            for (i=0;i<MAT_SIZE_h*2;i++) {
            fprintf(stderr, "%6.3f",mat[j*lda*2+i]);
      }
      fprintf(stderr, "\n");
    }
    fprintf(stderr, "\n");
} // mathdisp2

void mathdispAId(const double * AId, int lda, int n) {
      double * AI = new double[n*lda*2];
      cudaMemcpy(AI,AId,sizeof(double)*n*lda*2,cudaMemcpyDeviceToHost);
      mathdispAI(AI, lda, n);
      delete [] AI;
}

__global__ void GEStep1A(double * AI, int i, int n2, int lda2) {
      int k = blockIdx.x * blockDim.x + threadIdx.x;

  if (k>i && k < n2 && AI[i*lda2+k]!=0) {
    double multiplyer = -AI[i*lda2+k]/AI[i*lda2+i];
      int n = n2 / 2;
    for (int j = i+1; j < n; j++) {
      AI[j*lda2+k] += multiplyer*AI[j*lda2+i];
    }
  }
}

__global__ void GEStep2(double * AI,double diag,int i, int n2, int lda2) {
      int k = blockIdx.x * blockDim.x + threadIdx.x;
      if (k < n2) {
            AI[i*lda2+k] /= diag;
      }
}

__global__ void GEStep3(double * AI,int i, int n2, int lda2) {
  int k = blockIdx.x * blockDim.x + threadIdx.x;
  if (k > i && k < n2) {
    double multiplyer = -AI[i*lda2+k];
    for (int j = 0; j < i; j++) {
```

```
      AI[j*lda2+k] += multiplyer*AI[j*lda2+i];
    }
  }
}


void invertge(double * AI_d, int lda, int n) {
      int lda2 = lda * 2;
  // perform elementary row operations till A in AI becomes identity matrix
  for (int i = 0; i < n; i++) {
    GEStep1A<<<(int)ceil(1+(2*n-1)/32),32>>>(AI_d,i,n*2, lda2);
    CUDACHECK;
    cudaThreadSynchronize();
  }


  for (int i = n-1; i >= 0; i--) {
    double diag = 1.0;
    SAFECALL(cudaMemcpy(&diag, &AI_d[i*lda2+i], sizeof(double),
cudaMemcpyDeviceToHost));
    GEStep2<<<(int)ceil(1+(n*2-1)/32),32>>>(AI_d,diag,i,n*2, lda2);
    CUDACHECK;

    GEStep3<<<(int)ceil(1+(n*2-1)/32),32>>>(AI_d,i,n*2, lda2);
    CUDACHECK;
    cudaThreadSynchronize();
    CUDACHECK;
  }
} // invertge

__host__ void setToTommyDevice() {
      cudaError_t setDeviceErrorCode;
      setDeviceErrorCode = cudaSetDevice(2);
      if(setDeviceErrorCode != cudaSuccess) {
            printf("Error setting to Tommy's
device:\n%s\n",cudaGetErrorString(setDeviceErrorCode));
            printf("Program will now exit.\n");
            exit(EXIT_FAILURE);
      }
}

__host__ void printMat(int lda, double *A)
{
      int i,j;
      for(i=0;i<lda;i++){
            printf("[");
            for(j=0;j<lda;j++){
                  printf("%4.10f ",*(A+(lda*i+j)));
            }
            printf("]\n");
      }
      printf("\n");
}

__host__ double *addStates(int size, int kp, int gammap, int km, int gammam)
{
  int i;
```

```
  int m,p;
  int side = size*size;
  double *A = (double *)calloc(side*side,sizeof(double));
  for(i=0;i<side;i++){
    m = i/size;
    p = i%size;
    *(A+i*side+i) = (double)(-km - m*gammam - m*kp - p*gammap);
    if(i != (side-1)){
      *(A+(i+1)*side+i) = (double)(m*kp);
      *(A+i*side+i+1) = (double)(((p+1)%size)*gammap);
    }
    if((m > 0) && (m < size)){
      *(A+(i-size)*side+i) = (double)(gammam*m);
      *(A+i*side+(i-size)) = (double)km;
    }
  }
  return A;
}

__host__ void NBlock(int N, double *d_N, double *d_reacMat, double
*d_tempMat, int p)
{
      int i;
      double *Atemp;

      cublasAlloc(N*N, sizeof(double), (void**)&Atemp);
      cublasDscal(N,scalar_N[0],d_N,N+1);
      for(i=1;i<p;i++){

      cublasDgemm('n','n',N,N,N,1.0,d_tempMat,N,d_reacMat,N,0.0,Atemp,N);
            cublasDcopy(N*N,Atemp,1,d_tempMat,1);
            cublasDaxpy(N*N,scalar_N[i],d_tempMat,1,d_N,1);
      }
      cublasFree(Atemp);
}

__host__ void DBlock(int N, double *d_D, double *d_reacMat, double
*d_tempMat, int p)
{
      int i;
      double *Atemp;

      cublasAlloc(N*N, sizeof(double), (void**)&Atemp);
      cublasDscal(N,scalar_D[0],d_D,N+1);
      for(i=1;i<p;i++){
            cublasDgemm('n','n',N,N,N,-
1.0,d_tempMat,N,d_reacMat,N,0.0,Atemp,N);
            cublasDcopy(N*N,Atemp,1,d_tempMat,1);
            cublasDaxpy(N*N,scalar_D[i],d_tempMat,1,d_D,1);
      }
      cublasFree(Atemp);
}
__host__ void FSP(int kp, int gammap, int km, int gammam, int p, int q, int
startProb, double targetError, int tf)
{
      cublasStatus status;
```

```
	int i;
	int currentSize = 60;
	int N;
	int n2;
	int lda;
	double tempNorm, norm, m;
	double tempError;
	double *reacMat;
	double *probVec;
	double *identMat;
	double *d_reacMat;
	double *d_probVec;
	double *d_D;
	double *d_N;
	double *d_tempMat;

	time_t now;

	time(&now);

	printf("%s\n",ctime(&now));

	status = cublasInit();
	if(status != CUBLAS_STATUS_SUCCESS){
		fprintf(stderr,"CUBLAS won't turn on...\n");
	}

	while(1){
		m = 0.0;
		N = currentSize*currentSize;
		n2 = N*N;
		reacMat = addStates(currentSize,kp,gammap,km,gammam);
		probVec = (double *)calloc(N,sizeof(double));
		identMat = (double *)calloc(n2,sizeof(double));
		*(probVec+startProb) = 1.0;
		for(i=0;i<N;i++){
			*(identMat+(N*i)+i) = 1.0;
		}

		/* reserve and copy matrices and vectors */
		status = cublasAlloc(n2, sizeof(double), (void**)&d_reacMat);
		if(status != CUBLAS_STATUS_SUCCESS){
			fprintf(stderr,"can't fit the reaction matrix on the
card!\n");
		}
		status = cublasAlloc(N, sizeof(double), (void**)&d_probVec);
		if(status != CUBLAS_STATUS_SUCCESS){
			fprintf(stderr,"can't fit Vi on the card!\n");
		}
		status = cublasAlloc(n2, sizeof(double), (void**)&d_D);
		if(status != CUBLAS_STATUS_SUCCESS){
			fprintf(stderr,"can't fit D on the card!\n");
		}
		status = cublasAlloc(n2, sizeof(double), (void**)&d_N);
		if(status != CUBLAS_STATUS_SUCCESS){
			fprintf(stderr,"can't fit N on the card!\n");
		}
```

```
            status = cublasAlloc(n2, sizeof(double), (void**)&d_tempMat);
            if(status != CUBLAS_STATUS_SUCCESS){
                  fprintf(stderr,"can't fit temporary workspace on the
card!\n");
            }

            status = cublasSetVector(n2, sizeof(double), reacMat, 1,
d_reacMat, 1);
            if(status != CUBLAS_STATUS_SUCCESS){
                  fprintf(stderr,"can't copy the reaction matrix to the
card!\n");
            }
            status  = cublasSetVector(N, sizeof(double), probVec, 1,
d_probVec, 1);
            if(status != CUBLAS_STATUS_SUCCESS){
                  fprintf(stderr,"can't copy Vi to the card!\n");
            }
            status  = cublasSetVector(n2, sizeof(double), identMat, 1, d_N,
1);
            if(status != CUBLAS_STATUS_SUCCESS){
                  fprintf(stderr,"can't copy N to the card!\n");
            }
            status  = cublasSetVector(n2, sizeof(double), identMat, 1, d_D,
1);
            if(status != CUBLAS_STATUS_SUCCESS){
                  fprintf(stderr,"can't copy D to the card!\n");
            }
            status  = cublasSetVector(n2, sizeof(double), identMat, 1,
d_tempMat, 1);
            if(status != CUBLAS_STATUS_SUCCESS){
                  fprintf(stderr,"can't copy D to the card!\n");
            }

            /* multiply by tf */
            cublasDscal(n2,(double)tf,d_reacMat,1);

            /* find matrix norm */
            norm = 0.0;
            for(i=0;i<N;i++){
                  tempNorm = cublasDnrm2(N,d_reacMat+i,N);
                  status = cublasGetError();
                  if(status != CUBLAS_STATUS_SUCCESS){
                        printf("error - %d\n",status);
                  }
                  if(tempNorm > norm){
                        norm = tempNorm;
                  }
            }

            /* scaling */
            if(norm > 1.0){
                  m = 1.0;
                  while(m < norm){
                        m *= 2.0;
                  }
                  cublasDscal(n2, 1.0/m, d_reacMat, 1);
            }
```

```
            /* find blocks */
            NBlock(N,d_N,d_reacMat,d_tempMat,p);
            cublasDcopy(n2,d_D,1,d_tempMat,1);
            DBlock(N,d_D,d_reacMat,d_tempMat,p);

            /* Invert D Block */
            cublasGetVector(n2,sizeof(double),d_D,1,reacMat,1);
            lda = ((N+15)&~15|16);
            free(identMat);
            cublasFree(d_tempMat);
            cublasFree(d_reacMat);

            identMat = (double *)calloc(N*lda*2,sizeof(double));
            status = cublasAlloc(N*lda*2, sizeof(double),
(void**)&d_tempMat);
            for(i=0;i<N;i++) {
                  //memcpy(&identMat[lda*i*2], &reacMat[N*i],
sizeof(double)*N);
                  cblas_dcopy(N,(reacMat+(N*i)),1,(identMat+(lda*i*2)),1);
                  identMat[lda*i*2+N+i] = 1.0;
            }
            cublasSetVector(lda*N*2,sizeof(double),identMat,1,d_tempMat,1);
            invertge(d_tempMat,lda,N);
            cublasGetVector(lda*N*2,sizeof(double),d_tempMat,1,identMat,1);

            cublasFree(d_tempMat);

            status = cublasAlloc(N*N, sizeof(double), (void**)&d_tempMat);
            for(i=0;i<N;i++){
                  cblas_dcopy(N,(identMat+(lda*i*2+N)),1,(reacMat+(N*i)),1);
            }

            cublasSetVector(n2,sizeof(double),reacMat,1,d_D,1);

            /* Pade approximation result */
            cublasDgemm('n','n',N,N,N,1.0,d_D,N,d_N,N,0.0,d_tempMat,N);
            cublasDcopy(n2,d_tempMat,1,d_D,1);

            /* squaring */
            while(m > 1){
                  m /= 2;
                  cublasDgemm('n','n',N,N,N,1.0,d_D,N,d_D,N,0.0,d_tempMat,N);
                  cublasDcopy(n2,d_tempMat,1,d_D,1);
            }

            /*multiply probability vector*/
            cublasDgemv('t',N,N,1.0,d_D,N,d_probVec,1,0.0,d_tempMat,1);
            cublasDcopy(N,d_tempMat,N,d_probVec,1);

            /* compute error */
            tempError = 1.0 - cublasDasum(N,d_probVec,1);

            //if(tempError < targetError){
                break;
            if(currentSize > 100){
                  break;
```

```
            }else{
                  currentSize++;
                  free(reacMat);
                  free(identMat);
                  free(probVec);
                  status = cublasFree(d_D);
                  status = cublasFree(d_N);
                  status = cublasFree(d_tempMat);
                  status = cublasFree(d_probVec);
            }
      }

      time(&now);

      printf("%s\n",ctime(&now));

      printf("\ncurrentSize - %d\n",currentSize);
      printf("error - %4.10f -> %4.10f\n\n",tempError,targetError);

      cublasGetVector(N,sizeof(double),d_probVec,1,probVec,1);
      for(i=0;i<N;i++){
            printf("%f\n",*(probVec+i));
      }

      free(reacMat);
      free(identMat);
      free(probVec);
      status = cublasFree(d_D);
      status = cublasFree(d_N);
      status = cublasFree(d_tempMat);
      status = cublasFree(d_probVec);

      status = cublasShutdown();
      if(status != CUBLAS_STATUS_SUCCESS){
            printf("can't turn it off\n");
      }
}

__host__ double factorial(int n)
{
  double result = 1.0;
  while(n > 1){
    result *= (double)n;
    n--;
  }
  return result;
}

int main(int argc, char* argv[]) {
      int kp = 20;
      int gammap = 1;
      int km = 20;
      int gammam = 1;
      int p = 10;
      int q = 10;
      int tf  = 10;
      int j;
```

```
      int startProb = 0;
      double targetError = 10e-7;

      for(j=0;j<p;j++){
            scalar_N[j] = ( factorial(p+q-j)*factorial(p) )/(
factorial(p+q)*factorial(j)*factorial(p-j) );
            scalar_D[j] = ( factorial(p+q-j)*factorial(q) )/(
factorial(p+q)*factorial(j)*factorial(q-j) );
      }

      setToTommyDevice();
      FSP(kp, gammap, km, gammam, p, q, startProb, targetError, tf);
      return EXIT_SUCCESS;
}
```

blasFSPDoubleSweep.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <cublas.h>
#include <cblas.h>
#include <time.h>

/*select device*/


double scalar_N[10];
double scalar_D[10];

inline void __cudaSafeCall( int err, const char *file, const int line )
{
  do {
    if( err != 0) {
      fprintf(stderr, "cudaSafeCall() Runtime API error in file <%s>, line %i
: %s.\n",
            file, line, cudaGetErrorString((cudaError_t) err) );
      exit(-1);
    }
  } while (0);
}

#define SAFECALL(err) __cudaSafeCall(err, __FILE__, __LINE__)
#define CUDACHECK {cudaError_t error = cudaGetLastError(); if(error !=
0){fprintf(stderr, "Error code %d: %s file %s line
%i.\n",error,cudaGetErrorString(error),__FILE__,__LINE__);}}


void mathdispAI(const double *mat, int lda, int MAT_SIZE_h) {
    fprintf(stderr, "\n");
      int i,j;
    for (j=0;j<MAT_SIZE_h;j++)  {
            for (i=0;i<MAT_SIZE_h*2;i++) {
            fprintf(stderr, "%6.3f",mat[j*lda*2+i]);
      }
      fprintf(stderr, "\n");
    }
    fprintf(stderr, "\n");
```

```
} // mathdisp2

void mathdispAId(const double * AId, int lda, int n) {
      double * AI = new double[n*lda*2];
      cudaMemcpy(AI,AId,sizeof(double)*n*lda*2,cudaMemcpyDeviceToHost);
      mathdispAI(AI, lda, n);
      delete [] AI;
}

__global__ void GEStep1A(double * AI, int i, int n2, int lda2) {
      int k = blockIdx.x * blockDim.x + threadIdx.x;

  if (k>i && k < n2 && AI[i*lda2+k]!=0) {
    double multiplyer = -AI[i*lda2+k]/AI[i*lda2+i];
      int n = n2 / 2;
    for (int j = i+1; j < n; j++) {
      AI[j*lda2+k] += multiplyer*AI[j*lda2+i];
    }
  }
}

__global__ void GEStep2(double * AI,double diag,int i, int n2, int lda2) {
      int k = blockIdx.x * blockDim.x + threadIdx.x;
      if (k < n2) {
            AI[i*lda2+k] /= diag;
      }
}

__global__ void GEStep3(double * AI,int i, int n2, int lda2) {
  int k = blockIdx.x * blockDim.x + threadIdx.x;
  if (k > i && k < n2) {
    double multiplyer = -AI[i*lda2+k];
    for (int j = 0; j < i; j++) {
      AI[j*lda2+k] += multiplyer*AI[j*lda2+i];
    }
  }
}


void invertge(double * AI_d, int lda, int n) {
      int lda2 = lda * 2;
  // perform elementary row operations till A in AI becomes identity matrix
  for (int i = 0; i < n; i++) {
    GEStep1A<<<(int)ceil(1+(2*n-1)/32),32>>>(AI_d,i,n*2, lda2);
    CUDACHECK;
    cudaThreadSynchronize();
  }


  for (int i = n-1; i >= 0; i--) {
    double diag = 1.0;
    SAFECALL(cudaMemcpy(&diag, &AI_d[i*lda2+i], sizeof(double),
cudaMemcpyDeviceToHost));
    GEStep2<<<(int)ceil(1+(n*2-1)/32),32>>>(AI_d,diag,i,n*2, lda2);
    CUDACHECK;

    GEStep3<<<(int)ceil(1+(n*2-1)/32),32>>>(AI_d,i,n*2, lda2);
```

```
    CUDACHECK;
    cudaThreadSynchronize();
    CUDACHECK;
  }
} // invertge

__host__ void setToTommyDevice() {
      cudaError_t setDeviceErrorCode;
      setDeviceErrorCode = cudaSetDevice(2);
      if(setDeviceErrorCode != cudaSuccess) {
            printf("Error setting to Tommy's
device:\n%s\n",cudaGetErrorString(setDeviceErrorCode));
            printf("Program will now exit.\n");
            exit(EXIT_FAILURE);
      }
}

__host__ void printMat(int lda, double *A)
{
      int i,j;
      for(i=0;i<lda;i++){
            printf("[");
            for(j=0;j<lda;j++){
                  printf("%4.10f ",*(A+(lda*i+j)));
            }
            printf("]\n");
      }
      printf("\n");
}

__host__ double *addStates(int size, int kp, int gammap, int km, int gammam)
{
  int i;
  int m,p;
  int side = size*size;
  double *A = (double *)calloc(side*side,sizeof(double));
  for(i=0;i<side;i++){
    m = i/size;
    p = i%size;
    *(A+i*side+i) = (double)(-km - m*gammam - m*kp - p*gammap);
    if(i != (side-1)){
      *(A+(i+1)*side+i) = (double)(m*kp);
      *(A+i*side+i+1) = (double)(((p+1)%size)*gammap);
    }
    if((m > 0) && (m < size)){
      *(A+(i-size)*side+i) = (double)(gammam*m);
      *(A+i*side+(i-size)) = (double)km;
    }
  }
  return A;
}

__host__ void NBlock(int N, double *d_N, double *d_reacMat, double
*d_tempMat, int p)
{
      int i;
      double *Atemp;
```

```
	cublasAlloc(N*N, sizeof(double), (void**)&Atemp);
	cublasDscal(N,scalar_N[0],d_N,N+1);
	for(i=1;i<p;i++){

	cublasDgemm('n','n',N,N,N,1.0,d_tempMat,N,d_reacMat,N,0.0,Atemp,N);
		cublasDcopy(N*N,Atemp,1,d_tempMat,1);
		cublasDaxpy(N*N,scalar_N[i],d_tempMat,1,d_N,1);
	}
	cublasFree(Atemp);
}

__host__ void DBlock(int N, double *d_D, double *d_reacMat, double
*d_tempMat, int p)
{
	int i;
	double *Atemp;

	cublasAlloc(N*N, sizeof(double), (void**)&Atemp);
	cublasDscal(N,scalar_D[0],d_D,N+1);
	for(i=1;i<p;i++){
		cublasDgemm('n','n',N,N,N,-
1.0,d_tempMat,N,d_reacMat,N,0.0,Atemp,N);
		cublasDcopy(N*N,Atemp,1,d_tempMat,1);
		cublasDaxpy(N*N,scalar_D[i],d_tempMat,1,d_D,1);
	}
	cublasFree(Atemp);
}
__host__ void FSP(int kp, int gammap, int km, int gammam, int p, int q, int
startProb, double targetError, long int tf, int currentSize)
{
	cublasStatus status;

	int i;
	int N;
	int n2;
	int lda;
	double tempNorm, norm, m;
	double tempError;
	double *reacMat;
	double *probVec;
	double *identMat;
	double *d_reacMat;
	double *d_probVec;
	double *d_D;
	double *d_N;
	double *d_tempMat;

	time_t now;

	status = cublasInit();
	if(status != CUBLAS_STATUS_SUCCESS){
		fprintf(stderr,"CUBLAS won't turn on...\n");
	}

		m = 0.0;
		N = currentSize*currentSize;
```

```
            n2 = N*N;
            reacMat = addStates(currentSize,kp,gammap,km,gammam);
            probVec = (double *)calloc(N,sizeof(double));
            identMat = (double *)calloc(n2,sizeof(double));
            *(probVec+startProb) = 1.0;
            for(i=0;i<N;i++){
                  *(identMat+(N*i)+i) = 1.0;
            }

            time(&now);

            printf("%s\n",ctime(&now));

            /* reserve and copy matrices and vectors */
            status = cublasAlloc(n2, sizeof(double), (void**)&d_reacMat);
            if(status != CUBLAS_STATUS_SUCCESS){
                  fprintf(stderr,"can't fit the reaction matrix on the
card!\n");
            }
            status = cublasAlloc(N, sizeof(double), (void**)&d_probVec);
            if(status != CUBLAS_STATUS_SUCCESS){
                  fprintf(stderr,"can't fit Vi on the card!\n");
            }
            status = cublasAlloc(n2, sizeof(double), (void**)&d_D);
            if(status != CUBLAS_STATUS_SUCCESS){
                  fprintf(stderr,"can't fit D on the card!\n");
            }
            status = cublasAlloc(n2, sizeof(double), (void**)&d_N);
            if(status != CUBLAS_STATUS_SUCCESS){
                  fprintf(stderr,"can't fit N on the card!\n");
            }
            status = cublasAlloc(n2, sizeof(double), (void**)&d_tempMat);
            if(status != CUBLAS_STATUS_SUCCESS){
                  fprintf(stderr,"can't fit temporary workspace on the
card!\n");
            }

            status = cublasSetVector(n2, sizeof(double), reacMat, 1,
d_reacMat, 1);
            if(status != CUBLAS_STATUS_SUCCESS){
                  fprintf(stderr,"can't copy the reaction matrix to the
card!\n");
            }
            status  = cublasSetVector(N, sizeof(double), probVec, 1,
d_probVec, 1);
            if(status != CUBLAS_STATUS_SUCCESS){
                  fprintf(stderr,"can't copy Vi to the card!\n");
            }
            status  = cublasSetVector(n2, sizeof(double), identMat, 1, d_N,
1);
            if(status != CUBLAS_STATUS_SUCCESS){
                  fprintf(stderr,"can't copy N to the card!\n");
            }
            status  = cublasSetVector(n2, sizeof(double), identMat, 1, d_D,
1);
            if(status != CUBLAS_STATUS_SUCCESS){
                  fprintf(stderr,"can't copy D to the card!\n");
```

```
            }
            status  = cublasSetVector(n2, sizeof(double), identMat, 1,
d_tempMat, 1);
            if(status != CUBLAS_STATUS_SUCCESS){
                  fprintf(stderr,"can't copy D to the card!\n");
            }

            /* multiply by tf */
            cublasDscal(n2,(double)tf,d_reacMat,1);

            /* find matrix norm */
            norm = 0.0;
            for(i=0;i<N;i++){
                  tempNorm = cublasDnrm2(N,d_reacMat+i,N);
                  status = cublasGetError();
                  if(status != CUBLAS_STATUS_SUCCESS){
                        printf("error - %d\n",status);
                  }
                  if(tempNorm > norm){
                        norm = tempNorm;
                  }
            }

            /* scaling */
            if(norm > 1.0){
                  m = 1.0;
                  while(m < norm){
                        m *= 2.0;
                  }
                  cublasDscal(n2, 1.0/m, d_reacMat, 1);
            }

            /* find blocks */
            NBlock(N,d_N,d_reacMat,d_tempMat,p);
            cublasDcopy(n2,d_D,1,d_tempMat,1);
            DBlock(N,d_D,d_reacMat,d_tempMat,p);

            /* Invert D Block */
            cublasGetVector(n2,sizeof(double),d_D,1,reacMat,1);
            lda = ((N+15)&~15|16);
            free(identMat);
            cublasFree(d_tempMat);
            cublasFree(d_reacMat);

            identMat = (double *)calloc(N*lda*2,sizeof(double));
            status = cublasAlloc(N*lda*2, sizeof(double),
(void**)&d_tempMat);
            for(i=0;i<N;i++) {
                  //memcpy(&identMat[lda*i*2], &reacMat[N*i],
sizeof(double)*N);
                  cblas_dcopy(N,(reacMat+(N*i)),1,(identMat+(lda*i*2)),1);
                  identMat[lda*i*2+N+i] = 1.0;
            }
            cublasSetVector(lda*N*2,sizeof(double),identMat,1,d_tempMat,1);
            invertge(d_tempMat,lda,N);
            cublasGetVector(lda*N*2,sizeof(double),d_tempMat,1,identMat,1);
```

```
            cublasFree(d_tempMat);

            status = cublasAlloc(N*N, sizeof(double), (void**)&d_tempMat);
            for(i=0;i<N;i++){
                  cblas_dcopy(N,(identMat+(lda*i*2+N)),1,(reacMat+(N*i)),1);
            }

            cublasSetVector(n2,sizeof(double),reacMat,1,d_D,1);

            /* Pade approximation result */
            cublasDgemm('n','n',N,N,N,1.0,d_D,N,d_N,N,0.0,d_tempMat,N);
            cublasDcopy(n2,d_tempMat,1,d_D,1);

            /* squaring */
            while(m > 1){
                  m /= 2;
                  cublasDgemm('n','n',N,N,N,1.0,d_D,N,d_D,N,0.0,d_tempMat,N);
                  cublasDcopy(n2,d_tempMat,1,d_D,1);
            }

            /*multiply probability vector*/
            cublasDgemv('t',N,N,1.0,d_D,N,d_probVec,1,0.0,d_tempMat,1);
            cublasDcopy(N,d_tempMat,N,d_probVec,1);

            /* compute error */
            tempError = 1.0 - cublasDasum(N,d_probVec,1);


      time(&now);

      printf("%s\n",ctime(&now));

      printf("\ncurrentSize - %d\n",currentSize);
      printf("error - %4.10f -> %4.10f\n\n",tempError,targetError);

      cublasGetVector(N,sizeof(double),d_probVec,1,probVec,1);
      for(i=0;i<N;i++){
            printf("%f\n",*(probVec+i));
      }

      free(reacMat);
      free(identMat);
      free(probVec);
      status = cublasFree(d_D);
      status = cublasFree(d_N);
      status = cublasFree(d_tempMat);
      status = cublasFree(d_probVec);

      status = cublasShutdown();
      if(status != CUBLAS_STATUS_SUCCESS){
            printf("can't turn it off\n");
      }
}

__host__ double factorial(int n)
{
  double result = 1.0;
```

```
  while(n > 1){
    result *= (double)n;
    n--;
  }
  return result;
}

int main(int argc, char* argv[]) {
      int kp;
      int gammap = 1;
      int km;
      int gammam = 1;
      int p = 10;
      int q = 10;
      long int tf;
      int j;
      int startProb = 0;
      int currentSize;
      double targetError = 10e-7;

      if(argc != 5) {
            printf("usage - %s [currentSize] [km] [kp] [tf]\n",argv[0]);
            return EXIT_FAILURE;
      }
      sscanf(argv[1],"%d",&currentSize);
      sscanf(argv[2],"%d",&km);
      sscanf(argv[3],"%d",&kp);
      sscanf(argv[4],"%ld",&tf);

      for(j=0;j<p;j++){
            scalar_N[j] = ( factorial(p+q-j)*factorial(p) )/(
factorial(p+q)*factorial(j)*factorial(p-j) );
            scalar_D[j] = ( factorial(p+q-j)*factorial(q) )/(
factorial(p+q)*factorial(j)*factorial(q-j) );
      }

      setToTommyDevice();
      FSP(kp, gammap, km, gammam, p, q, startProb, targetError, tf,
currentSize);
      return EXIT_SUCCESS;
}
```

blasFSPFloat.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <cublas.h>
#include <time.h>

/*select device*/


float scalar_N[10];
float scalar_D[10];

__host__ void setToTommyDevice() {
      cudaError_t setDeviceErrorCode;
```

```
	setDeviceErrorCode = cudaSetDevice(2);
	if(setDeviceErrorCode != cudaSuccess) {
		printf("Error setting to Tommy's
device:\n%d\n",cudaGetErrorString(setDeviceErrorCode));
		printf("Program will now exit.\n");
		exit(EXIT_FAILURE);
	}
}

__host__ void printMat(int lda, float *A)
{
	int i,j;
	for(i=0;i<lda;i++){
		printf("[");
		for(j=0;j<lda;j++){
			printf("%4.10f ",*(A+(lda*i+j)));
		}
		printf("]\n");
	}
	printf("\n");
}

__host__ float *addStates(int size, int kp, int gammap, int km, int gammam)
{
  int i;
  int m,p;
  int side = size*size;
  float *A = (float *)calloc(side*side,sizeof(float));
  for(i=0;i<side;i++){
    m = i/size;
    p = i%size;
    *(A+i*side+i) = (float)(-km - m*gammam - m*kp - p*gammap);
    if(i != (side-1)){
      *(A+(i+1)*side+i) = (float)(m*kp);
      *(A+i*side+i+1) = (float)(((p+1)%size)*gammap);
    }
    if((m > 0) && (m < size)){
      *(A+(i-size)*side+i) = (float)(gammam*m);
      *(A+i*side+(i-size)) = (float)km;
    }
  }
  return A;
}

__host__ void NBlock(int N, float *d_N, float *d_reacMat, float *d_tempMat,
int p)
{
	int i;
	float *Atemp;

	cublasAlloc(N*N, sizeof(float), (void**)&Atemp);
	cublasSscal(N,scalar_N[0],d_N,N+1);
	for(i=1;i<p;i++){

	cublasSgemm('n','n',N,N,N,1.0,d_tempMat,N,d_reacMat,N,0.0,Atemp,N);
		cublasScopy(N*N,Atemp,1,d_tempMat,1);
		cublasSaxpy(N*N,scalar_N[i],d_tempMat,1,d_N,1);
```

```
	}
	cublasFree(Atemp);
}

__host__ void DBlock(int N, float *d_D, float *d_reacMat, float *d_tempMat,
int p)
{
	int i;
	float *Atemp;

	cublasAlloc(N*N, sizeof(float), (void**)&Atemp);
	cublasSscal(N,scalar_D[0],d_D,N+1);
	for(i=1;i<p;i++){
		cublasSgemm('n','n',N,N,N,-
1.0,d_tempMat,N,d_reacMat,N,0.0,Atemp,N);
		cublasScopy(N*N,Atemp,1,d_tempMat,1);
		cublasSaxpy(N*N,scalar_D[i],d_tempMat,1,d_D,1);
	}
	cublasFree(Atemp);
}

__host__ void matInvert(int N, float *d_A, float *d_tempMat)
{
	int i,j;
	float *multterm, *topelement;

	multterm = (float *)calloc(1,sizeof(float));
	topelement = (float *)calloc(1,sizeof(float));

	for(i=0;i<N;i++){
		cublasGetVector(1,sizeof(float),(d_A+(i*N+i)),1,topelement,1);
		if(*topelement == 0){
			printf("Matrix may be singular.\n");
		}else{
			cublasSscal(N,(1.0 / *topelement),(d_A+(i*N)),1);
			cublasSscal(N,(1.0 / *topelement),(d_tempMat+(i*N)),1);
		}
		for(j=0;j<N;j++){
			if(j == i){
				j++;
			}
			if(j != N){

	cublasGetVector(1,sizeof(float),(d_A+(j*N+i)),1,multterm,1);
				*multterm *= -1.0;
				cublasSaxpy(N,*multterm,(d_A+(i*N)),1,(d_A+(j*N)),1);

	cublasSaxpy(N,*multterm,(d_tempMat+(i*N)),1,(d_tempMat+(j*N)),1);
			}
		}
	}
	cublasSswap(N*N,d_A,1,d_tempMat,1);
}

__host__ void FSP(int kp, int gammap, int km, int gammam, int p, int q, int
startProb, float targetError, int tf)
{
```

```
	cublasStatus status;

	int i;
	int currentSize = 2;
	int N;
	int n2;
	float tempNorm, norm, m;
	float tempError;
	float *reacMat;
	float *probVec;
	float *identMat;
	float *d_reacMat;
	float *d_probVec;
	float *d_D;
	float *d_N;
	float *d_tempMat;

	time_t now;

	status = cublasInit();
	if(status != CUBLAS_STATUS_SUCCESS){
		fprintf(stderr,"CUBLAS won't turn on...\n");
	}

	while(1){
		m = 0.0;
		N = currentSize*currentSize;
		n2 = N*N;
		reacMat = addStates(currentSize,kp,gammap,km,gammam);
		probVec = (float *)calloc(N,sizeof(float));
		identMat = (float *)calloc(n2,sizeof(float));
		*(probVec+startProb) = 1.0;
		for(i=0;i<N;i++){
			*(identMat+(N*i)+i) = 1.0;
		}

		time(&now);

		printf("%s\n",ctime(&now));

		/* reserve and copy matrices and vectors */
		status = cublasAlloc(n2, sizeof(float), (void**)&d_reacMat);
		if(status != CUBLAS_STATUS_SUCCESS){
			fprintf(stderr,"can't fit the reaction matrix on the
card!\n");
		}
		status = cublasAlloc(N, sizeof(float), (void**)&d_probVec);
		if(status != CUBLAS_STATUS_SUCCESS){
			fprintf(stderr,"can't fit Vi on the card!\n");
		}
		status = cublasAlloc(n2, sizeof(float), (void**)&d_D);
		if(status != CUBLAS_STATUS_SUCCESS){
			fprintf(stderr,"can't fit D on the card!\n");
		}
		status = cublasAlloc(n2, sizeof(float), (void**)&d_N);
		if(status != CUBLAS_STATUS_SUCCESS){
			fprintf(stderr,"can't fit N on the card!\n");
```

```
            }
            status = cublasAlloc(n2, sizeof(float), (void**)&d_tempMat);
            if(status != CUBLAS_STATUS_SUCCESS){
                  fprintf(stderr,"can't fit temporary workspace on the
card!\n");
            }

            status = cublasSetVector(n2, sizeof(float), reacMat, 1,
d_reacMat, 1);
            if(status != CUBLAS_STATUS_SUCCESS){
                  fprintf(stderr,"can't copy the reaction matrix to the
card!\n");
            }
            status  = cublasSetVector(N, sizeof(float), probVec, 1,
d_probVec, 1);
            if(status != CUBLAS_STATUS_SUCCESS){
                  fprintf(stderr,"can't copy Vi to the card!\n");
            }
            status  = cublasSetVector(n2, sizeof(float), identMat, 1, d_N,
1);
            if(status != CUBLAS_STATUS_SUCCESS){
                  fprintf(stderr,"can't copy N to the card!\n");
            }
            status  = cublasSetVector(n2, sizeof(float), identMat, 1, d_D,
1);
            if(status != CUBLAS_STATUS_SUCCESS){
                  fprintf(stderr,"can't copy D to the card!\n");
            }
            status  = cublasSetVector(n2, sizeof(float), identMat, 1,
d_tempMat, 1);
            if(status != CUBLAS_STATUS_SUCCESS){
                  fprintf(stderr,"can't copy D to the card!\n");
            }

            /* multiply by tf */
            cublasSscal(n2,(float)tf,d_reacMat,1);

            /* find matrix norm */
            norm = 0.0;
            for(i=0;i<N;i++){
                  tempNorm = cublasSnrm2(N,d_reacMat+i,N);
                  status = cublasGetError();
                  if(status != CUBLAS_STATUS_SUCCESS){
                        printf("error - %d\n",status);
                  }
                  if(tempNorm > norm){
                        norm = tempNorm;
                  }
            }

            /* scaling */
            if(norm > 1.0){
                  m = 1.0;
                  while(m < norm){
                        m *= 2.0;
                  }
                  cublasSscal(n2, 1.0/m, d_reacMat, 1);
```

```
		}

		/* find blocks */
		NBlock(N,d_N,d_reacMat,d_tempMat,p);

		/*status = cublasGetVector(n2,sizeof(float),d_N,1,reacMat,1);

		printMat(N,reacMat);*/

		cublasScopy(n2,d_D,1,d_tempMat,1);
		DBlock(N,d_D,d_reacMat,d_tempMat,p);

		status  = cublasSetVector(n2, sizeof(float), identMat, 1,
d_tempMat, 1);
		if(status != CUBLAS_STATUS_SUCCESS){
			fprintf(stderr,"can't copy D to the card!\n");
		}

		/* Invert D Block */
		matInvert(N,d_D,d_tempMat);

		/* Pade approximation result */
		cublasSgemm('n','n',N,N,N,1.0,d_D,N,d_N,N,0.0,d_tempMat,N);
		cublasScopy(n2,d_tempMat,1,d_D,1);

		/* squaring */
		while(m > 1){
			m /= 2;
			cublasSgemm('n','n',N,N,N,1.0,d_D,N,d_D,N,0.0,d_tempMat,N);
			cublasScopy(n2,d_tempMat,1,d_D,1);
		}

		/*multiply probability vector*/
		cublasSgemv('t',N,N,1.0,d_D,N,d_probVec,1,0.0,d_tempMat,1);
		cublasScopy(N,d_tempMat,N,d_probVec,1);

		/* compute error */
		tempError = 1.0 - cublasSasum(N,d_probVec,1);

		time(&now);

		printf("%s\n",ctime(&now));

		printf("\ncurrentSize - %d\n",currentSize);
		printf("error - %4.10f -> %4.10f\n\n",tempError,targetError);

		free(reacMat);
		free(identMat);
		free(probVec);
		status = cublasFree(d_reacMat);
		status = cublasFree(d_D);
		status = cublasFree(d_N);
		status = cublasFree(d_tempMat);
		status = cublasFree(d_probVec);

		if(tempError < targetError){
			break;
```

```
            }else{
                  currentSize++;
            }
      }
      status = cublasShutdown();
      if(status != CUBLAS_STATUS_SUCCESS){
            printf("can't turn it off\n");
      }

      printf("\ncurrentSize - %d\n",currentSize);
      printf("error - %4.10f -> %4.10f\n",tempError,targetError);
}

__host__ float factorial(int n)
{
  float result = 1.0;
  while(n > 1){
    result *= (float)n;
    n--;
  }
  return result;
}

int main(int argc, char* argv[]) {
      int kp = 2;
      int gammap = 1;
      int km = 2;
      int gammam = 1;
      int p = 10;
      int q = 10;
      int tf  = 10;
      int j;
      int startProb = 0;
      float targetError = 10e-7;

      for(j=0;j<p;j++){
            scalar_N[j] = ( factorial(p+q-j)*factorial(p) )/(
factorial(p+q)*factorial(j)*factorial(p-j) );
            scalar_D[j] = ( factorial(p+q-j)*factorial(q) )/(
factorial(p+q)*factorial(j)*factorial(q-j) );
      }

      setToTommyDevice();
      FSP(kp, gammap, km, gammam, p, q, startProb, targetError, tf);
      return EXIT_SUCCESS;
}
```

Blastest.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <cublas.h>
#include <sys/time.h>

/*select device*/
__host__ void setToTommyDevice() {
      cudaError_t setDeviceErrorCode;
```

```
	setDeviceErrorCode = cudaSetDevice(2);
	if(setDeviceErrorCode != cudaSuccess) {
		printf("Error setting to Tommy's
device:\n%d\n",cudaGetErrorString(setDeviceErrorCode));
		printf("Program will now exit.\n");
		exit(EXIT_FAILURE);
	}
}

__global__ void multiply_gpu(int N, float *A, float *B, float *C)
{
	int i;
	int rows = blockIdx.x * blockDim.x + threadIdx.x;
	int cols = blockIdx.y * blockDim.y + threadIdx.y;


	*(C+(N*rows+cols)) = 0.0;
	for(i=0;i<N;i++){
		*(C+(N*rows+cols)) += *(A+(N*rows+i)) * *(B+(N*i+cols));
	}
}

void printTime(struct timeval *start) {
     struct timeval end;
     gettimeofday(&end,0);
     long secs  = end.tv_sec  - start->tv_sec;
     long usecs = end.tv_usec - start->tv_usec;

printf("%ld.%06ld\n",(secs*1000000+usecs)/1000000,(secs*1000000+usecs)%100000
0);
     *start = end;
}

int main(int argc, char* argv[]) {
	cublasStatus status;

	float *A;
	float *B;
	float *C;
	float *C_2;

	float *d_A;
	float *d_B;
	float *d_C;

	float *d2_A;
	float *d2_B;
	float *d2_C;

	int N;
	int n2;
	int i;

	struct timeval tommy;
	gettimeofday(&tommy,0);

	int blas_time,func_time,blas_load,func_load;
```

```
struct timeval my_start, my_end;
struct timeval blas_start, blas_end;
struct timeval my_load_start, my_load_end;
struct timeval blas_load_start, blas_load_end;

setToTommyDevice();

if(argc != 2){
      printf("./prog_name [mat size]\n");
      exit(EXIT_FAILURE);
}


printf("Loading random data into matrices\n");

N = atoi(argv[1]);
n2 = N*N;

dim3 threadsPerBlock(1);
dim3 numBlocks(N/threadsPerBlock.x, N/threadsPerBlock.y);

/*host memory allocation*/
A = (float*)calloc(n2,sizeof(float));
B = (float*)calloc(n2,sizeof(float));
C = (float*)calloc(n2,sizeof(float));
C_2 = (float*)calloc(n2,sizeof(float));

srand(3);

for(i=0;i<n2;i++){
      *(A+i) = (float)rand()/RAND_MAX;
      *(B+i) = (float)rand()/RAND_MAX;
}
/*for(i=0;i<N;i++){
      *(B+(N*i+i)) = 1.0;
}*/

printTime(&tommy);
printf("cudaMalloc\n");

gettimeofday(&my_load_start,0);

cudaMalloc(&d2_A, n2*sizeof(float));
cudaMalloc(&d2_B, n2*sizeof(float));
cudaMalloc(&d2_C, n2*sizeof(float));

printTime(&tommy);
printf("cudaMemcpy\n");

cudaMemcpy(d2_A, A, n2*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d2_B, B, n2*sizeof(float), cudaMemcpyHostToDevice);

gettimeofday(&my_load_end,0);

printTime(&tommy);
printf("multiply_gpu\n");
```

```
gettimeofday(&my_start,0);

multiply_gpu<<<numBlocks, threadsPerBlock>>>(N, d2_A, d2_B, d2_C);
cudaThreadSynchronize();

gettimeofday(&my_end,0);

printTime(&tommy);
printf("cudaMemcpy\n");

cudaMemcpy(C, d2_C, n2*sizeof(float), cudaMemcpyDeviceToHost);

printTime(&tommy);
printf("cudaMemcpy\n");

cudaFree(d2_A);
cudaFree(d2_B);
cudaFree(d2_C);

printTime(&tommy);
printf("cublasInit\n");

gettimeofday(&blas_load_start,0);

status = cublasInit();
if (status != CUBLAS_STATUS_SUCCESS) {
      fprintf (stderr, "!!!! CUBLAS initialization error\n");
        return EXIT_FAILURE;
}


printTime(&tommy);
printf("cublasAlloc\n");

/*device memory allocation*/
status = cublasAlloc(n2, sizeof(float), (void**)&d_A);
status = cublasAlloc(n2, sizeof(float), (void**)&d_B);
status = cublasAlloc(n2, sizeof(float), (void**)&d_C);

printTime(&tommy);
printf("cublasSetVector\n");

/*initialize device*/
status = cublasSetVector(n2,sizeof(float),A,1,d_A,1);
status = cublasSetVector(n2,sizeof(float),B,1,d_B,1);

gettimeofday(&blas_load_end,0);

printTime(&tommy);
printf("cublasSgemm\n");

gettimeofday(&blas_start,0);

cublasSgemm('n', 'n', N, N, N, 1.0, d_A, N, d_B, N, 0.0, d_C, N);

gettimeofday(&blas_end,0);
```

```
	status = cublasGetError();
	status = cublasGetVector(n2,sizeof(float),d_C,1,C_2,1);

	printTime(&tommy);
	printf("Verify\n");

	for(i=0;i<n2;i++){
		if(*(C+i) != *(C_2+i)){
			printf("whoops at %d!\n",i);
			printf("A - %f\n",*(A+i));
			printf("C - %f\n",*(C+i));
			printf("C_2 - %f\n",*(C_2+i));
			free(A);
			free(B);
			free(C);
			status = cublasFree(d_A);
			status = cublasFree(d_B);
			status = cublasFree(d_C);

			status = cublasShutdown();
			if (status != CUBLAS_STATUS_SUCCESS) {
				fprintf (stderr, "!!!! shutdown error (A)\n");
			return EXIT_FAILURE;
		  }
			return EXIT_FAILURE;
		}
	}

	free(A);
	free(B);
	free(C);
	status = cublasFree(d_A);
	status = cublasFree(d_B);
	status = cublasFree(d_C);

	status = cublasShutdown();
	if (status != CUBLAS_STATUS_SUCCESS) {
			fprintf (stderr, "!!!! shutdown error (A)\n");
		return EXIT_FAILURE;
	  }

	printf("blas_end:		%ld.%6ld\n",blas_end.tv_sec,blas_end.tv_usec);
	printf("blas_start:
%ld.%6ld\n",blas_start.tv_sec,blas_start.tv_usec);
	printf("my_end:		%ld.%6ld\n",my_end.tv_sec,my_end.tv_usec);
	printf("my_start:		%ld.%6ld\n",my_start.tv_sec,my_start.tv_usec);
	printf("blas_load_end:
%ld.%6ld\n",blas_load_end.tv_sec,blas_load_end.tv_usec);
	printf("blas_load_start:
%ld.%6ld\n",blas_load_start.tv_sec,blas_load_start.tv_usec);
	printf("my_load_end:
%ld.%6ld\n",my_load_end.tv_sec,my_load_end.tv_usec);
	printf("my_load_start:
%ld.%6ld\n",my_load_start.tv_sec,my_load_start.tv_usec);
```

```
      blas_time = (int)((blas_end.tv_sec*1000000.0 + blas_end.tv_usec) -
(blas_start.tv_sec*1000000.0 + blas_start.tv_usec));
      func_time = (int)((my_end.tv_sec*1000000.0 + my_end.tv_usec) -
(my_start.tv_sec*1000000.0 + my_start.tv_usec));
      blas_load = (int)((blas_load_end.tv_sec*1000000.0 +
blas_load_end.tv_usec) - (blas_load_start.tv_sec*1000000.0 +
blas_load_start.tv_usec));
      func_load = (int)((my_load_end.tv_sec*1000000.0 + my_load_end.tv_usec)
- (my_load_start.tv_sec*1000000.0 + my_load_start.tv_usec));

      printf("woohoo!\n");
      printf("function time - %d useconds\n",func_time);
      printf("function memory load time - %d useconds\n",func_load);
      printf("BLAS time - %d useconds\n",blas_time);
      printf("BLAS memory load time - %d useconds\n",blas_load);
      printTime(&tommy);
      return EXIT_SUCCESS;
}
```

geneExpmRNA.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <cublas.h>
#include <cblas.h>
#include <math.h>

/*select device*/


double scalar_N[10];
double scalar_D[10];


int *p1s;
int *m1s;

inline void __cudaSafeCall( int err, const char *file, const int line )
{
  do {
    if( err != 0) {
      fprintf(stderr, "cudaSafeCall() Runtime API error in file <%s>, line %i
: %s.\n",
              file, line, cudaGetErrorString((cudaError_t) err) );
      exit(-1);
    }
  } while (0);
}

#define SAFECALL(err) __cudaSafeCall(err, __FILE__, __LINE__)
#define CUDACHECK {cudaError_t error = cudaGetLastError(); if(error !=
0){fprintf(stderr, "Error code %d: %s file %s line
%i.\n",error,cudaGetErrorString(error),__FILE__,__LINE__);}}


void mathdispAI(const double *mat, int lda, int MAT_SIZE_h) {
```

```
  fprintf(stderr, "\n");
  int i,j;
  for (j=0;j<MAT_SIZE_h;j++)  {
    for (i=0;i<MAT_SIZE_h*2;i++) {
      fprintf(stderr, "%6.3f",mat[j*lda*2+i]);
    }
    fprintf(stderr, "\n");
  }
  fprintf(stderr, "\n");
} // mathdisp2

void mathdispAId(const double * AId, int lda, int n) {
  double * AI = new double[n*lda*2];
  cudaMemcpy(AI,AId,sizeof(double)*n*lda*2,cudaMemcpyDeviceToHost);
  mathdispAI(AI, lda, n);
  delete [] AI;
}

__global__ void GEStep1A(double * AI, int i, int n2, int lda2) {
  int k = blockIdx.x * blockDim.x + threadIdx.x;

  if (k>i && k < n2 && AI[i*lda2+k]!=0) {
    double multiplyer = -AI[i*lda2+k]/AI[i*lda2+i];
    int n = n2 / 2;
    for (int j = i+1; j < n; j++) {
      AI[j*lda2+k] += multiplyer*AI[j*lda2+i];
    }
  }
}

__global__ void GEStep2(double * AI,double diag,int i, int n2, int lda2) {
  int k = blockIdx.x * blockDim.x + threadIdx.x;
  if (k < n2) {
    AI[i*lda2+k] /= diag;
  }
}

__global__ void GEStep3(double * AI,int i, int n2, int lda2) {
  int k = blockIdx.x * blockDim.x + threadIdx.x;
  if (k > i && k < n2) {
    double multiplyer = -AI[i*lda2+k];
    for (int j = 0; j < i; j++) {
      AI[j*lda2+k] += multiplyer*AI[j*lda2+i];
    }
  }
}


void invertge(double * AI_d, int lda, int n) {
  int lda2 = lda * 2;
  // perform elementary row operations till A in AI becomes identity matrix
  for (int i = 0; i < n; i++) {
    GEStep1A<<<(int)ceil(1+(2*n-1)/32),32>>>(AI_d,i,n*2, lda2);
    CUDACHECK;
    cudaThreadSynchronize();
  }
```

```
  for (int i = n-1; i >= 0; i--) {
    double diag = 1.0;
    SAFECALL(cudaMemcpy(&diag, &AI_d[i*lda2+i], sizeof(double),
cudaMemcpyDeviceToHost));
    GEStep2<<<(int)ceil(1+(n*2-1)/32),32>>>(AI_d,diag,i,n*2, lda2);
    CUDACHECK;

    GEStep3<<<(int)ceil(1+(n*2-1)/32),32>>>(AI_d,i,n*2, lda2);
    CUDACHECK;
    cudaThreadSynchronize();
    CUDACHECK;
  }
} // invertge

__host__ void setToTommyDevice(int device) {
  cudaError_t setDeviceErrorCode;
  setDeviceErrorCode = cudaSetDevice(device);
  if(setDeviceErrorCode != cudaSuccess) {
    printf("Error setting to Tommy's
device:\n%s\n",cudaGetErrorString(setDeviceErrorCode));
    printf("Program will now exit.\n");
    exit(EXIT_FAILURE);
  }
}

__host__ void printMat(int lda, double *A)
{
  int i,j;
  for(i=0;i<lda;i++){
    printf("[");
    for(j=0;j<lda;j++){
      printf("%4.10f ",*(A+(lda*i+j)));
    }
    printf("]\n");
  }
  printf("\n");
}

__host__ double *addStates(double km, double kp, double g1, double g2, int
size)
{
  int i,j,num,num1,num2;

  double *A = (double *)calloc(size*size,sizeof(double));

  for(i=0;i<size;i++){
    for(j=0;j<size;j++){
      if(i == j){
      *(A+(j*size+i)) = -km - (double)*(m1s+i)*kp - (double)*(p1s+i)*g1 -
(double)*(m1s+i)*g2;
      }else{
      num = abs(*(p1s+i) - *(p1s+j)) + abs(*(m1s+i) - *(m1s+j));
      if(num != 1){
        *(A+(j*size+i)) = 0.0;
      }else if(num == 1){
        num1 = *(p1s+i) - *(p1s+j);
```

```
        num2 = *(m1s+i) - *(m1s+j);
        if(num1 != 0){
          if(num1 > 0){
            *(A+(j*size+i)) = (double)*(p1s+i)*g1;
          }else{
            *(A+(j*size+i)) = (double)*(m1s+i)*kp;
          }
        }else if(num2 != 0){
          if(num2 > 0){
            *(A+(j*size+i)) = (double)*(m1s+i)*g2;
          }else{
            *(A+(j*size+i)) = km;
          }
        }
      }
      }
    }
  }
  return A;
}

/*smart growth*/
__host__ void growStates(int &mRNA, int &protein, int &size)
{
  int i;
  mRNA++;
  protein++;
  for(i=0;i<protein;i++){
    size++;

    m1s = (int*)realloc(m1s,size*sizeof(int));
    *(m1s+(size-1)) = (double)mRNA;

    p1s = (int*)realloc(p1s,size*sizeof(int));
    *(p1s+(size-1)) = (double)i;
  }

  for(i=0;i<=mRNA;i++){
    size++;

    m1s = (int*)realloc(m1s,size*sizeof(int));
    *(m1s+(size-1)) = (double)i;

    p1s = (int*)realloc(p1s,size*sizeof(int));
    *(p1s+(size-1)) = (double)protein;
  }
}


__host__ void NBlock(int N, double *d_N, double *d_reacMat, double
*d_tempMat, int p)
{
  int i;
  double *Atemp;

  cublasAlloc(N*N, sizeof(double), (void**)&Atemp);
  cublasDscal(N,scalar_N[0],d_N,N+1);
```

```
  for(i=1;i<p;i++){
    cublasDgemm('n','n',N,N,N,1.0,d_tempMat,N,d_reacMat,N,0.0,Atemp,N);
    cublasDcopy(N*N,Atemp,1,d_tempMat,1);
    cublasDaxpy(N*N,scalar_N[i],d_tempMat,1,d_N,1);
  }
  cublasFree(Atemp);
}

__host__ void DBlock(int N, double *d_D, double *d_reacMat, double
*d_tempMat, int p)
{
  int i;
  double *Atemp;

  cublasAlloc(N*N, sizeof(double), (void**)&Atemp);
  cublasDscal(N,scalar_D[0],d_D,N+1);
  for(i=1;i<p;i++){
    cublasDgemm('n','n',N,N,N,-1.0,d_tempMat,N,d_reacMat,N,0.0,Atemp,N);
    cublasDcopy(N*N,Atemp,1,d_tempMat,1);
    cublasDaxpy(N*N,scalar_D[i],d_tempMat,1,d_D,1);
  }
  cublasFree(Atemp);
}
__host__ void FSP(double km, double g1, double kp, double g2, int p, int q,
int startProb, double targetError, long int tf)
{
  cublasStatus status;

  int i;
  int N = 4;
  int n2;
  int lda;
  int mRNA = 1;
  int protein = 1;
  double tempNorm, norm, m;
  double tempError = 1.0;
  double *reacMat;
  double *probVec;
  double *identMat;

  double *d_reacMat;
  double *d_probVec;
  double *d_D;
  double *d_N;
  double *d_tempMat;

  FILE *file;

  char fileName[20];

  status = cublasInit();
  if(status != CUBLAS_STATUS_SUCCESS){
    fprintf(stderr,"CUBLAS won't turn on...\n");
  }

  p1s = (int*)calloc(N,sizeof(int));
  m1s = (int*)calloc(N,sizeof(int));
```

```
for(i=0;i<N;i++){
  *(p1s+i) = i%2;
  *(m1s+i) = (i/2)%2;
}

while(1){
  m = 0.0;
  n2 = N*N;
  reacMat = addStates(km,kp,g1,g2,N);
  probVec = (double *)calloc(N,sizeof(double));
  identMat = (double *)calloc(n2,sizeof(double));
  *(probVec+startProb) = 1.0;
  for(i=0;i<N;i++){
    *(identMat+(N*i)+i) = 1.0;
  }

  /* reserve and copy matrices and vectors */
  status = cublasAlloc(n2, sizeof(double), (void**)&d_reacMat);
  if(status != CUBLAS_STATUS_SUCCESS){
    fprintf(stderr,"can't fit the reaction matrix on the card!\n");
  }
  status = cublasAlloc(N, sizeof(double), (void**)&d_probVec);
  if(status != CUBLAS_STATUS_SUCCESS){
    fprintf(stderr,"can't fit Vi on the card!\n");
  }
  status = cublasAlloc(n2, sizeof(double), (void**)&d_D);
  if(status != CUBLAS_STATUS_SUCCESS){
    fprintf(stderr,"can't fit D on the card!\n");
  }
  status = cublasAlloc(n2, sizeof(double), (void**)&d_N);
  if(status != CUBLAS_STATUS_SUCCESS){
    fprintf(stderr,"can't fit N on the card!\n");
  }
  status = cublasAlloc(n2, sizeof(double), (void**)&d_tempMat);
  if(status != CUBLAS_STATUS_SUCCESS){
    fprintf(stderr,"can't fit temporary workspace on the card!\n");
  }

  status = cublasSetVector(n2, sizeof(double), reacMat, 1, d_reacMat, 1);
  if(status != CUBLAS_STATUS_SUCCESS){
    fprintf(stderr,"can't copy the reaction matrix to the card!\n");
  }
  status  = cublasSetVector(N, sizeof(double), probVec, 1, d_probVec, 1);
  if(status != CUBLAS_STATUS_SUCCESS){
    fprintf(stderr,"can't copy Vi to the card!\n");
  }
  status  = cublasSetVector(n2, sizeof(double), identMat, 1, d_N, 1);
  if(status != CUBLAS_STATUS_SUCCESS){
    fprintf(stderr,"can't copy N to the card!\n");
  }
  status  = cublasSetVector(n2, sizeof(double), identMat, 1, d_D, 1);
  if(status != CUBLAS_STATUS_SUCCESS){
    fprintf(stderr,"can't copy D to the card!\n");
  }
  status  = cublasSetVector(n2, sizeof(double), identMat, 1, d_tempMat, 1);
  if(status != CUBLAS_STATUS_SUCCESS){
```

```
  fprintf(stderr,"can't copy D to the card!\n");
}

/* multiply by tf */
cublasDscal(n2,(double)tf,d_reacMat,1);

/* find matrix norm */
norm = 0.0;
for(i=0;i<N;i++){
  tempNorm = cublasDnrm2(N,d_reacMat+i,N);
  status = cublasGetError();
  if(status != CUBLAS_STATUS_SUCCESS){
  printf("error - %d\n",status);
  }
  if(tempNorm > norm){
  norm = tempNorm;
  }
}

/* scaling */
if(norm > 1.0){
  m = 1.0;
  while(m < norm){
  m *= 2.0;
  }
  cublasDscal(n2, 1.0/m, d_reacMat, 1);
}

/* find blocks */
NBlock(N,d_N,d_reacMat,d_tempMat,p);
cublasDcopy(n2,d_D,1,d_tempMat,1);
DBlock(N,d_D,d_reacMat,d_tempMat,p);

/* Invert D Block */
cublasGetVector(n2,sizeof(double),d_D,1,reacMat,1);
lda = ((N+15)&~15|16);
free(identMat);
cublasFree(d_tempMat);
cublasFree(d_reacMat);

identMat = (double *)calloc(N*lda*2,sizeof(double));
status = cublasAlloc(N*lda*2, sizeof(double), (void**)&d_tempMat);
for(i=0;i<N;i++) {
  cblas_dcopy(N,(reacMat+(N*i)),1,(identMat+(lda*i*2)),1);
  identMat[lda*i*2+N+i] = 1.0;
}
cublasSetVector(lda*N*2,sizeof(double),identMat,1,d_tempMat,1);
invertge(d_tempMat,lda,N);
cublasGetVector(lda*N*2,sizeof(double),d_tempMat,1,identMat,1);

cublasFree(d_tempMat);

status = cublasAlloc(N*N, sizeof(double), (void**)&d_tempMat);
for(i=0;i<N;i++){
  cblas_dcopy(N,(identMat+(lda*i*2+N)),1,(reacMat+(N*i)),1);
}
```

```
  cublasSetVector(n2,sizeof(double),reacMat,1,d_D,1);

  /* Pade approximation result */
  cublasDgemm('n','n',N,N,N,1.0,d_D,N,d_N,N,0.0,d_tempMat,N);
  cublasDcopy(n2,d_tempMat,1,d_D,1);

  /* squaring */
  while(m > 1){
    m /= 2;
    cublasDgemm('n','n',N,N,N,1.0,d_D,N,d_D,N,0.0,d_tempMat,N);
    cublasDcopy(n2,d_tempMat,1,d_D,1);
  }

  /*multiply probability vector*/
  cublasDgemv('t',N,N,1.0,d_D,N,d_probVec,1,0.0,d_tempMat,1);
  cublasDcopy(N,d_tempMat,N,d_probVec,1);

  /* compute error */
  tempError = 1.0 - cublasDasum(N,d_probVec,1);
  printf("error - %f N - %d\n\n",tempError,N);

  cublasGetVector(N,sizeof(double),d_probVec,1,probVec,1);

  if((tempError < targetError) || (N > 10000)){
    sprintf(fileName,"run.txt");
    file = fopen(fileName,"w");
    for(i=0;i<N;i++){
    fprintf(file,"%d\t%d\t%lf\n",*(p1s+i),*(m1s+i),*(probVec+i));
    }
    fclose(file);
    free(reacMat);
    free(identMat);
    free(probVec);
    status = cublasFree(d_D);
    status = cublasFree(d_N);
    status = cublasFree(d_tempMat);
    status = cublasFree(d_probVec);
    free(p1s);
    free(m1s);
    break;
  }

  growStates(mRNA,protein,N);

  free(reacMat);
  free(identMat);
  free(probVec);
  status = cublasFree(d_D);
  status = cublasFree(d_N);
  status = cublasFree(d_tempMat);
  status = cublasFree(d_probVec);
}

status = cublasShutdown();
if(status != CUBLAS_STATUS_SUCCESS){
  printf("can't turn it off\n");
}
```

```
}

__host__ double factorial(int n)
{
  double result = 1.0;
  while(n > 1){
    result *= (double)n;
    n--;
  }
  return result;
}

int main(int argc, char* argv[]) {
  double kp;
  double g1 = 1.0;
  double km;
  double g2 = 1.0;
  int p = 10;
  int q = 10;
  long int tf;
  int j;
  int startProb = 0;
  int device;
  double targetError;

  if(argc != 6) {
    printf("usage - %s [kp] [km] [tf] [error] [device #]\n",argv[0]);
    return EXIT_FAILURE;
  }
  sscanf(argv[1],"%lf",&kp);
  sscanf(argv[2],"%lf",&km);
  sscanf(argv[3],"%ld",&tf);
  sscanf(argv[4],"%lf",&targetError);
  sscanf(argv[5],"%d",&device);

  for(j=0;j<p;j++){
    scalar_N[j] = ( factorial(p+q-j)*factorial(p) )/(
factorial(p+q)*factorial(j)*factorial(p-j) );
    scalar_D[j] = ( factorial(p+q-j)*factorial(q) )/(
factorial(p+q)*factorial(j)*factorial(q-j) );
  }

  setToTommyDevice(device);
  FSP(km, g1, kp, g2, p, q, startProb, targetError, tf);
  return EXIT_SUCCESS;
}
```

geneExpmRNASmart.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <cublas.h>
#include <cblas.h>
#include <math.h>

/*select device*/
```

```
double scalar_N[10];
double scalar_D[10];


int *p1s;
int *m1s;
int *edgeStates;

inline void __cudaSafeCall( int err, const char *file, const int line )
{
  do {
    if( err != 0) {
      fprintf(stderr, "cudaSafeCall() Runtime API error in file <%s>, line %i
: %s.\n",
              file, line, cudaGetErrorString((cudaError_t) err) );
      exit(-1);
    }
  } while (0);
}

#define SAFECALL(err) __cudaSafeCall(err, __FILE__, __LINE__)
#define CUDACHECK {cudaError_t error = cudaGetLastError(); if(error !=
0){fprintf(stderr, "Error code %d: %s file %s line
%i.\n",error,cudaGetErrorString(error),__FILE__,__LINE__);}}


void mathdispAI(const double *mat, int lda, int MAT_SIZE_h) {
  fprintf(stderr, "\n");
  int i,j;
  for (j=0;j<MAT_SIZE_h;j++)  {
    for (i=0;i<MAT_SIZE_h*2;i++) {
      fprintf(stderr, "%6.3f",mat[j*lda*2+i]);
    }
    fprintf(stderr, "\n");
  }
  fprintf(stderr, "\n");
} // mathdisp2

void mathdispAId(const double * AId, int lda, int n) {
  double * AI = new double[n*lda*2];
  cudaMemcpy(AI,AId,sizeof(double)*n*lda*2,cudaMemcpyDeviceToHost);
  mathdispAI(AI, lda, n);
  delete [] AI;
}

__global__ void GEStep1A(double * AI, int i, int n2, int lda2) {
  int k = blockIdx.x * blockDim.x + threadIdx.x;

  if (k>i && k < n2 && AI[i*lda2+k]!=0) {
    double multiplyer = -AI[i*lda2+k]/AI[i*lda2+i];
    int n = n2 / 2;
    for (int j = i+1; j < n; j++) {
      AI[j*lda2+k] += multiplyer*AI[j*lda2+i];
    }
  }
}
```

```
__global__ void GEStep2(double * AI,double diag,int i, int n2, int lda2) {
  int k = blockIdx.x * blockDim.x + threadIdx.x;
  if (k < n2) {
    AI[i*lda2+k] /= diag;
  }
}

__global__ void GEStep3(double * AI,int i, int n2, int lda2) {
  int k = blockIdx.x * blockDim.x + threadIdx.x;
  if (k > i && k < n2) {
    double multiplyer = -AI[i*lda2+k];
    for (int j = 0; j < i; j++) {
      AI[j*lda2+k] += multiplyer*AI[j*lda2+i];
    }
  }
}


void invertge(double * AI_d, int lda, int n) {
  int lda2 = lda * 2;
  // perform elementary row operations till A in AI becomes identity matrix
  for (int i = 0; i < n; i++) {
    GEStep1A<<<(int)ceil(1+(2*n-1)/32),32>>>(AI_d,i,n*2, lda2);
    CUDACHECK;
    cudaThreadSynchronize();
  }


  for (int i = n-1; i >= 0; i--) {
    double diag = 1.0;
    SAFECALL(cudaMemcpy(&diag, &AI_d[i*lda2+i], sizeof(double),
cudaMemcpyDeviceToHost));
    GEStep2<<<(int)ceil(1+(n*2-1)/32),32>>>(AI_d,diag,i,n*2, lda2);
    CUDACHECK;

    GEStep3<<<(int)ceil(1+(n*2-1)/32),32>>>(AI_d,i,n*2, lda2);
    CUDACHECK;
    cudaThreadSynchronize();
    CUDACHECK;
  }
} // invertge

__host__ void setToTommyDevice(int device) {
  cudaError_t setDeviceErrorCode;
  setDeviceErrorCode = cudaSetDevice(device);
  if(setDeviceErrorCode != cudaSuccess) {
    printf("Error setting to Tommy's
device:\n%s\n",cudaGetErrorString(setDeviceErrorCode));
    printf("Program will now exit.\n");
    exit(EXIT_FAILURE);
  }
}

__host__ void printMat(int lda, double *A)
{
  int i,j;
```

```
  for(i=0;i<lda;i++){
    printf("[");
    for(j=0;j<lda;j++){
      printf("%4.10f ",*(A+(lda*i+j)));
    }
    printf("]\n");
  }
  printf("\n");
}

__host__ double *addStates(double km, double kp, double g1, double g2, int
size)
{
  int i,j,num,num1,num2;

  double *A = (double *)calloc(size*size,sizeof(double));

  for(i=0;i<size;i++){
    for(j=0;j<size;j++){
      if(i == j){
      *(A+(j*size+i)) = -km - (double)*(m1s+i)*kp - (double)*(p1s+i)*g1 -
(double)*(m1s+i)*g2;
      }else{
      num = abs(*(p1s+i) - *(p1s+j)) + abs(*(m1s+i) - *(m1s+j));
      if(num != 1){
        *(A+(j*size+i)) = 0.0;
      }else if(num == 1){
        num1 = *(p1s+i) - *(p1s+j);
        num2 = *(m1s+i) - *(m1s+j);
        if(num1 != 0){
          if(num1 > 0){
            *(A+(j*size+i)) = (double)*(p1s+i)*g1;
          }else{
            *(A+(j*size+i)) = (double)*(m1s+i)*kp;
          }
        }else if(num2 != 0){
          if(num2 > 0){
            *(A+(j*size+i)) = (double)*(m1s+i)*g2;
          }else{
            *(A+(j*size+i)) = km;
          }
        }
      }
      }
    }
  }
  return A;
}

/*smart growth*/
__host__ void growStates(double *probVec, int &edgeSize, int &size, double
currentError, double targetError)
{
  int numAdded = 0;
  double maxNum;
  int maxIndex,maxEdgeIndex,i;
  int numFound1,numFound2,numFound3,numFound4;
```

```
  int tempSize = size;
  int tempEdgeSize = edgeSize;
  //int numToAdd = (int)round(pow((log10(currentError)-
log10(targetError)),2.0)*1000.0 + 20.0);
  int numToAdd = 10;

  while( (numAdded < tempSize) && (numAdded < numToAdd) && (tempEdgeSize > 0)
){
    maxNum = -1.0;
    maxIndex = -1;
    maxEdgeIndex = -1;

    for(i=0;i<tempEdgeSize;i++){
      if( *(probVec+*(edgeStates+i)) > maxNum){
      maxIndex = *(edgeStates+i);
      maxNum = *(probVec+maxIndex);
      maxEdgeIndex = i;
      }
    }
    numFound1 = 0;
    numFound2 = 0;
    numFound3 = 0;
    numFound4 = 0;
    for(i=0;i<size;i++){
      if( (*(p1s+i) == *(p1s+maxIndex)+1) && (*(m1s+i) == *(m1s+maxIndex)) ){
      numFound1 = 1;
      }
      if( ((*(p1s+i) == *(p1s+maxIndex)-1) && (*(m1s+i) == *(m1s+maxIndex)))
|| (*(p1s+maxIndex)-1 < 0) ){
      numFound2 = 1;
      }
      if( (*(p1s+i) == *(p1s+maxIndex)) && (*(m1s+i) == *(m1s+maxIndex)+1) ){
      numFound3 = 1;
      }
      if( ((*(p1s+i) == *(p1s+maxIndex)) && (*(m1s+i) == *(m1s+maxIndex)-1))
|| (*(m1s+maxIndex)-1 < 0) ){
      numFound4 = 1;
      }
    }
    if( (numFound1 == 0) ){
      size++;
      numAdded++;
      edgeSize++;

      p1s = (int*)realloc(p1s,size*sizeof(int));
      *(p1s+(size-1)) = *(p1s+maxIndex)+1;

      m1s = (int*)realloc(m1s,size*sizeof(int));
      *(m1s+(size-1)) = *(m1s+maxIndex);

      edgeStates = (int*)realloc(edgeStates,edgeSize*sizeof(int));
      *(edgeStates+(edgeSize-1)) = size-1;
    }else if(numFound2 == 0){
      size++;
      numAdded++;
      edgeSize++;
```

```
      p1s = (int*)realloc(p1s,size*sizeof(int));
      *(p1s+(size-1)) = *(p1s+maxIndex)-1;

      m1s = (int*)realloc(m1s,size*sizeof(int));
      *(m1s+(size-1)) = *(m1s+maxIndex);

      edgeStates = (int*)realloc(edgeStates,edgeSize*sizeof(int));
      *(edgeStates+(edgeSize-1)) = size-1;
    }else if( (numFound3 == 0) ){
      size++;
      numAdded++;
      edgeSize++;

      p1s = (int*)realloc(p1s,size*sizeof(int));
      *(p1s+(size-1)) = *(p1s+maxIndex);

      m1s = (int*)realloc(m1s,size*sizeof(int));
      *(m1s+(size-1)) = *(m1s+maxIndex)+1;

      edgeStates = (int*)realloc(edgeStates,edgeSize*sizeof(int));
      *(edgeStates+(edgeSize-1)) = size-1;
    }else if(numFound4 == 0){
      size++;
      numAdded++;
      edgeSize++;

      p1s = (int*)realloc(p1s,size*sizeof(int));
      *(p1s+(size-1)) = *(p1s+maxIndex);

      m1s = (int*)realloc(m1s,size*sizeof(int));
      *(m1s+(size-1)) = *(m1s+maxIndex)-1;

      edgeStates = (int*)realloc(edgeStates,edgeSize*sizeof(int));
      *(edgeStates+(edgeSize-1)) = size-1;
    }else{
      edgeSize--;
      tempEdgeSize--;
      for(i=maxEdgeIndex;i<edgeSize;i++){
      *(edgeStates+i) = *(edgeStates+i+1);
      }
      edgeStates = (int*)realloc(edgeStates,edgeSize*sizeof(int));
    }
  }
}


__host__ void NBlock(int N, double *d_N, double *d_reacMat, double
*d_tempMat, int p)
{
  int i;
  double *Atemp;

  cublasAlloc(N*N, sizeof(double), (void**)&Atemp);
  cublasDscal(N,scalar_N[0],d_N,N+1);
  for(i=1;i<p;i++){
    cublasDgemm('n','n',N,N,N,1.0,d_tempMat,N,d_reacMat,N,0.0,Atemp,N);
    cublasDcopy(N*N,Atemp,1,d_tempMat,1);
```

```
    cublasDaxpy(N*N,scalar_N[i],d_tempMat,1,d_N,1);
  }
  cublasFree(Atemp);
}

__host__ void DBlock(int N, double *d_D, double *d_reacMat, double
*d_tempMat, int p)
{
  int i;
  double *Atemp;

  cublasAlloc(N*N, sizeof(double), (void**)&Atemp);
  cublasDscal(N,scalar_D[0],d_D,N+1);
  for(i=1;i<p;i++){
    cublasDgemm('n','n',N,N,N,-1.0,d_tempMat,N,d_reacMat,N,0.0,Atemp,N);
    cublasDcopy(N*N,Atemp,1,d_tempMat,1);
    cublasDaxpy(N*N,scalar_D[i],d_tempMat,1,d_D,1);
  }
  cublasFree(Atemp);
}
__host__ void FSP(double km, double g1, double kp, double g2, int p, int q,
int startProb, double targetError, long int tf)
{
  cublasStatus status;

  int i;
  int N = 4;
  int n2;
  int lda;
  int edgeSize = 0;
  double tempNorm, norm, m;
  double tempError = 1.0;
  double *reacMat;
  double *probVec;
  double *identMat;

  double *d_reacMat;
  double *d_probVec;
  double *d_D;
  double *d_N;
  double *d_tempMat;

  FILE *file;

  char fileName[20];

  status = cublasInit();
  if(status != CUBLAS_STATUS_SUCCESS){
    fprintf(stderr,"CUBLAS won't turn on...\n");
  }

  p1s = (int*)calloc(N,sizeof(int));
  m1s = (int*)calloc(N,sizeof(int));
  edgeStates = (int*)calloc(0,sizeof(int));

  for(i=0;i<N;i++){
    *(p1s+i) = i%2;
```

```
  *(m1s+i) = (i/2)%2;
}

for(i=0;i<N;i++){
  edgeSize++;
  edgeStates = (int*)realloc(edgeStates,edgeSize*sizeof(int));
  *(edgeStates+(edgeSize-1)) = i;
}

while(1){
  m = 0.0;
  n2 = N*N;
  reacMat = addStates(km,kp,g1,g2,N);
  probVec = (double *)calloc(N,sizeof(double));
  identMat = (double *)calloc(n2,sizeof(double));
  *(probVec+startProb) = 1.0;
  for(i=0;i<N;i++){
    *(identMat+(N*i)+i) = 1.0;
  }

  /* reserve and copy matrices and vectors */
  status = cublasAlloc(n2, sizeof(double), (void**)&d_reacMat);
  if(status != CUBLAS_STATUS_SUCCESS){
    fprintf(stderr,"can't fit the reaction matrix on the card!\n");
  }
  status = cublasAlloc(N, sizeof(double), (void**)&d_probVec);
  if(status != CUBLAS_STATUS_SUCCESS){
    fprintf(stderr,"can't fit Vi on the card!\n");
  }
  status = cublasAlloc(n2, sizeof(double), (void**)&d_D);
  if(status != CUBLAS_STATUS_SUCCESS){
    fprintf(stderr,"can't fit D on the card!\n");
  }
  status = cublasAlloc(n2, sizeof(double), (void**)&d_N);
  if(status != CUBLAS_STATUS_SUCCESS){
    fprintf(stderr,"can't fit N on the card!\n");
  }
  status = cublasAlloc(n2, sizeof(double), (void**)&d_tempMat);
  if(status != CUBLAS_STATUS_SUCCESS){
    fprintf(stderr,"can't fit temporary workspace on the card!\n");
  }

  status = cublasSetVector(n2, sizeof(double), reacMat, 1, d_reacMat, 1);
  if(status != CUBLAS_STATUS_SUCCESS){
    fprintf(stderr,"can't copy the reaction matrix to the card!\n");
  }
  status  = cublasSetVector(N, sizeof(double), probVec, 1, d_probVec, 1);
  if(status != CUBLAS_STATUS_SUCCESS){
    fprintf(stderr,"can't copy Vi to the card!\n");
  }
  status  = cublasSetVector(n2, sizeof(double), identMat, 1, d_N, 1);
  if(status != CUBLAS_STATUS_SUCCESS){
    fprintf(stderr,"can't copy N to the card!\n");
  }
  status  = cublasSetVector(n2, sizeof(double), identMat, 1, d_D, 1);
  if(status != CUBLAS_STATUS_SUCCESS){
    fprintf(stderr,"can't copy D to the card!\n");
```

```
}
status  = cublasSetVector(n2, sizeof(double), identMat, 1, d_tempMat, 1);
if(status != CUBLAS_STATUS_SUCCESS){
  fprintf(stderr,"can't copy D to the card!\n");
}

/* multiply by tf */
cublasDscal(n2,(double)tf,d_reacMat,1);

/* find matrix norm */
norm = 0.0;
for(i=0;i<N;i++){
  tempNorm = cublasDnrm2(N,d_reacMat+i,N);
  status = cublasGetError();
  if(status != CUBLAS_STATUS_SUCCESS){
  printf("error - %d\n",status);
  }
  if(tempNorm > norm){
  norm = tempNorm;
  }
}

/* scaling */
if(norm > 1.0){
  m = 1.0;
  while(m < norm){
  m *= 2.0;
  }
  cublasDscal(n2, 1.0/m, d_reacMat, 1);
}

/* find blocks */
NBlock(N,d_N,d_reacMat,d_tempMat,p);
cublasDcopy(n2,d_D,1,d_tempMat,1);
DBlock(N,d_D,d_reacMat,d_tempMat,p);

/* Invert D Block */
cublasGetVector(n2,sizeof(double),d_D,1,reacMat,1);
lda = ((N+15)&~15|16);
free(identMat);
cublasFree(d_tempMat);
cublasFree(d_reacMat);

identMat = (double *)calloc(N*lda*2,sizeof(double));
status = cublasAlloc(N*lda*2, sizeof(double), (void**)&d_tempMat);
for(i=0;i<N;i++) {
  cblas_dcopy(N,(reacMat+(N*i)),1,(identMat+(lda*i*2)),1);
  identMat[lda*i*2+N+i] = 1.0;
}
cublasSetVector(lda*N*2,sizeof(double),identMat,1,d_tempMat,1);
invertge(d_tempMat,lda,N);
cublasGetVector(lda*N*2,sizeof(double),d_tempMat,1,identMat,1);

cublasFree(d_tempMat);

status = cublasAlloc(N*N, sizeof(double), (void**)&d_tempMat);
for(i=0;i<N;i++){
```

```
    cblas_dcopy(N,(identMat+(lda*i*2+N)),1,(reacMat+(N*i)),1);
  }

  cublasSetVector(n2,sizeof(double),reacMat,1,d_D,1);

  /* Pade approximation result */
  cublasDgemm('n','n',N,N,N,1.0,d_D,N,d_N,N,0.0,d_tempMat,N);
  cublasDcopy(n2,d_tempMat,1,d_D,1);

  /* squaring */
  while(m > 1){
    m /= 2;
    cublasDgemm('n','n',N,N,N,1.0,d_D,N,d_D,N,0.0,d_tempMat,N);
    cublasDcopy(n2,d_tempMat,1,d_D,1);
  }

  /*multiply probability vector*/
  cublasDgemv('t',N,N,1.0,d_D,N,d_probVec,1,0.0,d_tempMat,1);
  cublasDcopy(N,d_tempMat,N,d_probVec,1);

  /* compute error */
  tempError = 1.0 - cublasDasum(N,d_probVec,1);
  printf("error - %f N - %d\n\n",tempError,N);

  cublasGetVector(N,sizeof(double),d_probVec,1,probVec,1);

  if((tempError < targetError) || (N > 10000)){
    sprintf(fileName,"runSmart.txt");
    file = fopen(fileName,"w");
    for(i=0;i<N;i++){
    fprintf(file,"%d\t%d\t%lf\n",*(p1s+i),*(m1s+i),*(probVec+i));
    }
    fclose(file);
    free(reacMat);
    free(identMat);
    free(probVec);
    status = cublasFree(d_D);
    status = cublasFree(d_N);
    status = cublasFree(d_tempMat);
    status = cublasFree(d_probVec);
    free(p1s);
    free(m1s);
    free(edgeStates);
    break;
  }

  growStates(probVec,edgeSize,N,tempError,targetError);

  free(reacMat);
  free(identMat);
  free(probVec);
  status = cublasFree(d_D);
  status = cublasFree(d_N);
  status = cublasFree(d_tempMat);
  status = cublasFree(d_probVec);
}
```

```
  status = cublasShutdown();
  if(status != CUBLAS_STATUS_SUCCESS){
    printf("can't turn it off\n");
  }
}

__host__ double factorial(int n)
{
  double result = 1.0;
  while(n > 1){
    result *= (double)n;
    n--;
  }
  return result;
}

int main(int argc, char* argv[]) {
  double kp;
  double g1 = 1.0;
  double km;
  double g2 = 1.0;
  int p = 10;
  int q = 10;
  long int tf;
  int j;
  int startProb = 0;
  int device;
  double targetError;

  if(argc != 6) {
    printf("usage - %s [kp] [km] [tf] [error] [device #]\n",argv[0]);
    return EXIT_FAILURE;
  }
  sscanf(argv[1],"%lf",&kp);
  sscanf(argv[2],"%lf",&km);
  sscanf(argv[3],"%ld",&tf);
  sscanf(argv[4],"%lf",&targetError);
  sscanf(argv[5],"%d",&device);

  for(j=0;j<p;j++){
    scalar_N[j] = ( factorial(p+q-j)*factorial(p) )/(
factorial(p+q)*factorial(j)*factorial(p-j) );
    scalar_D[j] = ( factorial(p+q-j)*factorial(q) )/(
factorial(p+q)*factorial(j)*factorial(q-j) );
  }

  setToTommyDevice(device);
  FSP(km, g1, kp, g2, p, q, startProb, targetError, tf);
  return EXIT_SUCCESS;
}
```

repressCSLGPU.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <cublas.h>
#include <cblas.h>
```

```
#include <math.h>

/*select device*/


double scalar_N[10];
double scalar_D[10];


int *p1s;
int *p2s;
int *p3s;
int *g1s;
int *g2s;
int *g3s;
int *edgeStates;

inline void __cudaSafeCall( int err, const char *file, const int line )
{
  do {
    if( err != 0) {
      fprintf(stderr, "cudaSafeCall() Runtime API error in file <%s>, line %i
: %s.\n",
              file, line, cudaGetErrorString((cudaError_t) err) );
      exit(-1);
    }
  } while (0);
}

#define SAFECALL(err) __cudaSafeCall(err, __FILE__, __LINE__)
#define CUDACHECK {cudaError_t error = cudaGetLastError(); if(error !=
0){fprintf(stderr, "Error code %d: %s file %s line
%i.\n",error,cudaGetErrorString(error),__FILE__,__LINE__);}}


void mathdispAI(const double *mat, int lda, int MAT_SIZE_h) {
  fprintf(stderr, "\n");
  int i,j;
  for (j=0;j<MAT_SIZE_h;j++)  {
    for (i=0;i<MAT_SIZE_h*2;i++) {
      fprintf(stderr, "%6.3f",mat[j*lda*2+i]);
    }
    fprintf(stderr, "\n");
  }
  fprintf(stderr, "\n");
} // mathdisp2

void mathdispAId(const double * AId, int lda, int n) {
  double * AI = new double[n*lda*2];
  cudaMemcpy(AI,AId,sizeof(double)*n*lda*2,cudaMemcpyDeviceToHost);
  mathdispAI(AI, lda, n);
  delete [] AI;
}

__global__ void GEStep1A(double * AI, int i, int n2, int lda2) {
  int k = blockIdx.x * blockDim.x + threadIdx.x;
```

```
  if (k>i && k < n2 && AI[i*lda2+k]!=0) {
    double multiplyer = -AI[i*lda2+k]/AI[i*lda2+i];
    int n = n2 / 2;
    for (int j = i+1; j < n; j++) {
      AI[j*lda2+k] += multiplyer*AI[j*lda2+i];
    }
  }
}

__global__ void GEStep2(double * AI,double diag,int i, int n2, int lda2) {
  int k = blockIdx.x * blockDim.x + threadIdx.x;
  if (k < n2) {
    AI[i*lda2+k] /= diag;
  }
}

__global__ void GEStep3(double * AI,int i, int n2, int lda2) {
  int k = blockIdx.x * blockDim.x + threadIdx.x;
  if (k > i && k < n2) {
    double multiplyer = -AI[i*lda2+k];
    for (int j = 0; j < i; j++) {
      AI[j*lda2+k] += multiplyer*AI[j*lda2+i];
    }
  }
}


void invertge(double * AI_d, int lda, int n) {
  int lda2 = lda * 2;
  // perform elementary row operations till A in AI becomes identity matrix
  for (int i = 0; i < n; i++) {
    GEStep1A<<<(int)ceil(1+(2*n-1)/32),32>>>(AI_d,i,n*2, lda2);
    CUDACHECK;
    cudaThreadSynchronize();
  }


  for (int i = n-1; i >= 0; i--) {
    double diag = 1.0;
    SAFECALL(cudaMemcpy(&diag, &AI_d[i*lda2+i], sizeof(double),
cudaMemcpyDeviceToHost));
    GEStep2<<<(int)ceil(1+(n*2-1)/32),32>>>(AI_d,diag,i,n*2, lda2);
    CUDACHECK;

    GEStep3<<<(int)ceil(1+(n*2-1)/32),32>>>(AI_d,i,n*2, lda2);
    CUDACHECK;
    cudaThreadSynchronize();
    CUDACHECK;
  }
} // invertge

__host__ void setToTommyDevice(int device) {
  cudaError_t setDeviceErrorCode;
  setDeviceErrorCode = cudaSetDevice(device);
  if(setDeviceErrorCode != cudaSuccess) {
    printf("Error setting to Tommy's
device:\n%s\n",cudaGetErrorString(setDeviceErrorCode));
```

```
    printf("Program will now exit.\n");
    exit(EXIT_FAILURE);
  }
}

__host__ void printMat(int lda, double *A)
{
  int i,j;
  for(i=0;i<lda;i++){
    printf("[");
    for(j=0;j<lda;j++){
      printf("%4.10f ",*(A+(lda*i+j)));
    }
    printf("]\n");
  }
  printf("\n");
}

__host__ double *addStates(double k1, double k2, double k3, double g1, double
g2, double g3, double z, int size)
{
  int i,j,num,num1,num2,num3,num4,num5,num6;

  double *A = (double *)calloc(size*size,sizeof(double));
  double k1f = z;
  double k2f = z;
  double k3f = z;
  double k1r = z/k1;
  double k2r = z/k2;
  double k3r = z/k3;

  for(i=0;i<size;i++){
    for(j=0;j<size;j++){
      if(i == j){
      *(A+(j*size+i)) = -(double)*(g1s+i)*k1 - (double)*(g2s+i)*k2 -
(double)*(g3s+i)*k3 - (double)*(p1s+i)*g1 - (double)*(p2s+i)*g2 -
(double)*(p3s+i)*g3 - (double)*(p3s+i)*(double)*(g1s+i)*k1r -
(double)*(p1s+i)*(double)*(g2s+i)*k2r - (double)*(p2s+i)*(double)*(g3s+i)*k3r
- abs((double)*(g1s+i)-1.0)*k1f - abs((double)*(g2s+i)-1.0)*k2f -
abs((double)*(g3s+i)-1.0)*k3f;
      }else{
      num = abs(*(p1s+i) - *(p1s+j)) + abs(*(p2s+i) - *(p2s+j)) +
abs(*(p3s+i) - *(p3s+j)) + abs(*(g1s+i) - *(g1s+j)) + abs(*(g2s+i) -
*(g2s+j)) + abs(*(g3s+i) - *(g3s+j));
      if(num != 1){
        *(A+(j*size+i)) = 0.0;
      }else if(num == 1){
        num1 = *(p1s+i) - *(p1s+j);
        num2 = *(p2s+i) - *(p2s+j);
        num3 = *(p3s+i) - *(p3s+j);
        num4 = *(g1s+i) - *(g1s+j);
        num5 = *(g2s+i) - *(g2s+j);
        num6 = *(g3s+i) - *(g3s+j);
        if(num1 != 0){
          if(num1 > 0){
            *(A+(j*size+i)) = (double)*(p1s+i)*g1;
          }else{
```

```
            *(A+(j*size+i)) = (double)*(g1s+i)*k1;
          }
        }else if(num2 != 0){
          if(num2 > 0){
            *(A+(j*size+i)) = (double)*(p2s+i)*g2;
          }else{
            *(A+(j*size+i)) = (double)*(g2s+i)*k2;
          }
        }else if(num3 != 0){
          if(num3 > 0){
            *(A+(j*size+i)) = (double)*(p3s+i)*g3;
          }else{
            *(A+(j*size+i)) = (double)*(g3s+i)*k3;
          }
        }else if(num4 != 0){
          *(A+(j*size+i)) = (double)*(p3s+i)*(double)*(g1s+i)*k1r +
abs((double)*(g1s+i)-1.0)*k1f;
        }else if(num5 != 0){
          *(A+(j*size+i)) = (double)*(p1s+i)*(double)*(g2s+i)*k2r +
abs((double)*(g2s+i)-1.0)*k2f;
        }else if(num6 != 0){
          *(A+(j*size+i)) = (double)*(p2s+i)*(double)*(g3s+i)*k3r +
abs((double)*(g3s+i)-1.0)*k3f;
        }
      }
      }
    }
  }
  return A;
}

/*smart growth*/
__host__ void growStates(double *probVec, int &edgeSize, int &size, double
currentError, double targetError)
{
  int numAdded = 0;
  double maxNum;
  int maxIndex,maxEdgeIndex,i;
  int
numFound1,numFound2,numFound3,numFound4,numFound5,numFound6,numFound7,numFoun
d8,numFound9;
  int tempSize = size;
  int tempEdgeSize = edgeSize;
  int numToAdd = (int)round(pow((log10(currentError)-
log10(targetError)),2.0)*1000.0 + 20.0);

  while( (numAdded < tempSize) && (numAdded < numToAdd) && (tempEdgeSize > 0)
){
    maxNum = -1.0;
    maxIndex = -1;
    maxEdgeIndex = -1;

    for(i=0;i<tempEdgeSize;i++){
      if( *(probVec+*(edgeStates+i)) > maxNum){
      maxIndex = *(edgeStates+i);
      maxNum = *(probVec+maxIndex);
      maxEdgeIndex = i;
```

```
      }
    }
    numFound1 = 0;
    numFound2 = 0;
    numFound3 = 0;
    numFound4 = 0;
    numFound5 = 0;
    numFound6 = 0;
    numFound7 = 0;
    numFound8 = 0;
    numFound9 = 0;
    for(i=0;i<size;i++){
      if( (*(p1s+i) == *(p1s+maxIndex)+1) && (*(p2s+i) == *(p2s+maxIndex)) &&
(*(p3s+i) == *(p3s+maxIndex)) && (*(g1s+i) == *(g1s+maxIndex)) && (*(g2s+i)
== *(g2s+maxIndex)) && (*(g3s+i) == *(g3s+maxIndex)) ){
      numFound1 = 1;
      }
      if( ((*(p1s+i) == *(p1s+maxIndex)-1) && (*(p2s+i) == *(p2s+maxIndex))
&& (*(p3s+i) == *(p3s+maxIndex)) && (*(g1s+i) == *(g1s+maxIndex)) &&
(*(g2s+i) == *(g2s+maxIndex)) && (*(g3s+i) == *(g3s+maxIndex))) ||
(*(p1s+maxIndex)-1 < 0) ){
      numFound2 = 1;
      }
      if( (*(p1s+i) == *(p1s+maxIndex)) && (*(p2s+i) == *(p2s+maxIndex)+1) &&
(*(p3s+i) == *(p3s+maxIndex)) && (*(g1s+i) == *(g1s+maxIndex)) && (*(g2s+i)
== *(g2s+maxIndex)) && (*(g3s+i) == *(g3s+maxIndex)) ){
      numFound3 = 1;
      }
      if( ((*(p1s+i) == *(p1s+maxIndex)) && (*(p2s+i) == *(p2s+maxIndex)-1)
&& (*(p3s+i) == *(p3s+maxIndex)) && (*(g1s+i) == *(g1s+maxIndex)) &&
(*(g2s+i) == *(g2s+maxIndex)) && (*(g3s+i) == *(g3s+maxIndex))) ||
(*(p2s+maxIndex)-1 < 0) ){
      numFound4 = 1;
      }
      if( (*(p1s+i) == *(p1s+maxIndex)) && (*(p2s+i) == *(p2s+maxIndex)) &&
(*(p3s+i) == *(p3s+maxIndex)+1) && (*(g1s+i) == *(g1s+maxIndex)) && (*(g2s+i)
== *(g2s+maxIndex)) && (*(g3s+i) == *(g3s+maxIndex)) ){
      numFound5 = 1;
      }
      if( ((*(p1s+i) == *(p1s+maxIndex)) && (*(p2s+i) == *(p2s+maxIndex)) &&
(*(p3s+i) == *(p3s+maxIndex)-1) && (*(g1s+i) == *(g1s+maxIndex)) && (*(g2s+i)
== *(g2s+maxIndex)) && (*(g3s+i) == *(g3s+maxIndex))) || (*(p3s+maxIndex)-1 <
0) ){
      numFound6 = 1;
      }
      if( (*(p1s+i) == *(p1s+maxIndex)) && (*(p2s+i) == *(p2s+maxIndex)) &&
(*(p3s+i) == *(p3s+maxIndex)) && (*(g1s+i) == abs(*(g1s+maxIndex)-1)) &&
(*(g2s+i) == *(g2s+maxIndex)) && (*(g3s+i) == *(g3s+maxIndex)) ){
      numFound7 = 1;
      }
      if( (*(p1s+i) == *(p1s+maxIndex)) && (*(p2s+i) == *(p2s+maxIndex)) &&
(*(p3s+i) == *(p3s+maxIndex)) && (*(g1s+i) == *(g1s+maxIndex)) && (*(g2s+i)
== abs(*(g2s+maxIndex)-1)) && (*(g3s+i) == *(g3s+maxIndex)) ){
      numFound8 = 1;
      }
```

```
      if( (*(p1s+i) == *(p1s+maxIndex)) && (*(p2s+i) == *(p2s+maxIndex)) &&
(*(p3s+i) == *(p3s+maxIndex)) && (*(g1s+i) == *(g1s+maxIndex)) && (*(g2s+i)
== *(g2s+maxIndex)) && (*(g3s+i) == abs(*(g3s+maxIndex)-1)) ){
      numFound9 = 1;
      }
    }
    if( (numFound1 == 0) && (*(g1s+maxIndex) == 1) ){
      size++;
      numAdded++;
      edgeSize++;

      p1s = (int*)realloc(p1s,size*sizeof(int));
      *(p1s+(size-1)) = *(p1s+maxIndex)+1;

      p2s = (int*)realloc(p2s,size*sizeof(int));
      *(p2s+(size-1)) = *(p2s+maxIndex);

      p3s = (int*)realloc(p3s,size*sizeof(int));
      *(p3s+(size-1)) = *(p3s+maxIndex);

      g1s = (int*)realloc(g1s,size*sizeof(int));
      *(g1s+(size-1)) = *(g1s+maxIndex);

      g2s = (int*)realloc(g2s,size*sizeof(int));
      *(g2s+(size-1)) = *(g2s+maxIndex);

      g3s = (int*)realloc(g3s,size*sizeof(int));
      *(g3s+(size-1)) = *(g3s+maxIndex);

      edgeStates = (int*)realloc(edgeStates,edgeSize*sizeof(int));
      *(edgeStates+(edgeSize-1)) = size-1;
    }else if(numFound2 == 0){
      size++;
      numAdded++;
      edgeSize++;

      p1s = (int*)realloc(p1s,size*sizeof(int));
      *(p1s+(size-1)) = *(p1s+maxIndex)-1;

      p2s = (int*)realloc(p2s,size*sizeof(int));
      *(p2s+(size-1)) = *(p2s+maxIndex);

      p3s = (int*)realloc(p3s,size*sizeof(int));
      *(p3s+(size-1)) = *(p3s+maxIndex);

      g1s = (int*)realloc(g1s,size*sizeof(int));
      *(g1s+(size-1)) = *(g1s+maxIndex);

      g2s = (int*)realloc(g2s,size*sizeof(int));
      *(g2s+(size-1)) = *(g2s+maxIndex);

      g3s = (int*)realloc(g3s,size*sizeof(int));
      *(g3s+(size-1)) = *(g3s+maxIndex);

      edgeStates = (int*)realloc(edgeStates,edgeSize*sizeof(int));
      *(edgeStates+(edgeSize-1)) = size-1;
    }else if( (numFound3 == 0) && (*(g2s+maxIndex) == 1) ){
```

```
  size++;
  numAdded++;
  edgeSize++;

  p1s = (int*)realloc(p1s,size*sizeof(int));
  *(p1s+(size-1)) = *(p1s+maxIndex);

  p2s = (int*)realloc(p2s,size*sizeof(int));
  *(p2s+(size-1)) = *(p2s+maxIndex)+1;

  p3s = (int*)realloc(p3s,size*sizeof(int));
  *(p3s+(size-1)) = *(p3s+maxIndex);

  g1s = (int*)realloc(g1s,size*sizeof(int));
  *(g1s+(size-1)) = *(g1s+maxIndex);

  g2s = (int*)realloc(g2s,size*sizeof(int));
  *(g2s+(size-1)) = *(g2s+maxIndex);

  g3s = (int*)realloc(g3s,size*sizeof(int));
  *(g3s+(size-1)) = *(g3s+maxIndex);

  edgeStates = (int*)realloc(edgeStates,edgeSize*sizeof(int));
  *(edgeStates+(edgeSize-1)) = size-1;
}else if(numFound4 == 0){
  size++;
  numAdded++;
  edgeSize++;

  p1s = (int*)realloc(p1s,size*sizeof(int));
  *(p1s+(size-1)) = *(p1s+maxIndex);

  p2s = (int*)realloc(p2s,size*sizeof(int));
  *(p2s+(size-1)) = *(p2s+maxIndex)-1;

  p3s = (int*)realloc(p3s,size*sizeof(int));
  *(p3s+(size-1)) = *(p3s+maxIndex);

  g1s = (int*)realloc(g1s,size*sizeof(int));
  *(g1s+(size-1)) = *(g1s+maxIndex);

  g2s = (int*)realloc(g2s,size*sizeof(int));
  *(g2s+(size-1)) = *(g2s+maxIndex);

  g3s = (int*)realloc(g3s,size*sizeof(int));
  *(g3s+(size-1)) = *(g3s+maxIndex);

  edgeStates = (int*)realloc(edgeStates,edgeSize*sizeof(int));
  *(edgeStates+(edgeSize-1)) = size-1;
}else if( (numFound5 == 0) && (*(g3s+maxIndex) == 1) ){
  size++;
  numAdded++;
  edgeSize++;

  p1s = (int*)realloc(p1s,size*sizeof(int));
  *(p1s+(size-1)) = *(p1s+maxIndex);
```

```
  p2s = (int*)realloc(p2s,size*sizeof(int));
  *(p2s+(size-1)) = *(p2s+maxIndex);

  p3s = (int*)realloc(p3s,size*sizeof(int));
  *(p3s+(size-1)) = *(p3s+maxIndex)+1;

  g1s = (int*)realloc(g1s,size*sizeof(int));
  *(g1s+(size-1)) = *(g1s+maxIndex);

  g2s = (int*)realloc(g2s,size*sizeof(int));
  *(g2s+(size-1)) = *(g2s+maxIndex);

  g3s = (int*)realloc(g3s,size*sizeof(int));
  *(g3s+(size-1)) = *(g3s+maxIndex);

  edgeStates = (int*)realloc(edgeStates,edgeSize*sizeof(int));
  *(edgeStates+(edgeSize-1)) = size-1;
}else if(numFound6 == 0){
  size++;
  numAdded++;
  edgeSize++;

  p1s = (int*)realloc(p1s,size*sizeof(int));
  *(p1s+(size-1)) = *(p1s+maxIndex);

  p2s = (int*)realloc(p2s,size*sizeof(int));
  *(p2s+(size-1)) = *(p2s+maxIndex);

  p3s = (int*)realloc(p3s,size*sizeof(int));
  *(p3s+(size-1)) = *(p3s+maxIndex)-1;

  g1s = (int*)realloc(g1s,size*sizeof(int));
  *(g1s+(size-1)) = *(g1s+maxIndex);

  g2s = (int*)realloc(g2s,size*sizeof(int));
  *(g2s+(size-1)) = *(g2s+maxIndex);

  g3s = (int*)realloc(g3s,size*sizeof(int));
  *(g3s+(size-1)) = *(g3s+maxIndex);

  edgeStates = (int*)realloc(edgeStates,edgeSize*sizeof(int));
  *(edgeStates+(edgeSize-1)) = size-1;
}else if(numFound7 == 0){
  size++;
  numAdded++;
  edgeSize++;

  p1s = (int*)realloc(p1s,size*sizeof(int));
  *(p1s+(size-1)) = *(p1s+maxIndex);

  p2s = (int*)realloc(p2s,size*sizeof(int));
  *(p2s+(size-1)) = *(p2s+maxIndex);

  p3s = (int*)realloc(p3s,size*sizeof(int));
  *(p3s+(size-1)) = *(p3s+maxIndex);

  g1s = (int*)realloc(g1s,size*sizeof(int));
```

```
  *(g1s+(size-1)) = abs(*(g1s+maxIndex)-1);

  g2s = (int*)realloc(g2s,size*sizeof(int));
  *(g2s+(size-1)) = *(g2s+maxIndex);

  g3s = (int*)realloc(g3s,size*sizeof(int));
  *(g3s+(size-1)) = *(g3s+maxIndex);

  edgeStates = (int*)realloc(edgeStates,edgeSize*sizeof(int));
  *(edgeStates+(edgeSize-1)) = size-1;
}else if(numFound8 == 0){
  size++;
  numAdded++;
  edgeSize++;

  p1s = (int*)realloc(p1s,size*sizeof(int));
  *(p1s+(size-1)) = *(p1s+maxIndex);

  p2s = (int*)realloc(p2s,size*sizeof(int));
  *(p2s+(size-1)) = *(p2s+maxIndex);

  p3s = (int*)realloc(p3s,size*sizeof(int));
  *(p3s+(size-1)) = *(p3s+maxIndex);

  g1s = (int*)realloc(g1s,size*sizeof(int));
  *(g1s+(size-1)) = *(g1s+maxIndex);

  g2s = (int*)realloc(g2s,size*sizeof(int));
  *(g2s+(size-1)) = abs(*(g2s+maxIndex)-1);

  g3s = (int*)realloc(g3s,size*sizeof(int));
  *(g3s+(size-1)) = *(g3s+maxIndex);

  edgeStates = (int*)realloc(edgeStates,edgeSize*sizeof(int));
  *(edgeStates+(edgeSize-1)) = size-1;
}else if(numFound9 == 0){
  size++;
  numAdded++;
  edgeSize++;

  p1s = (int*)realloc(p1s,size*sizeof(int));
  *(p1s+(size-1)) = *(p1s+maxIndex);

  p2s = (int*)realloc(p2s,size*sizeof(int));
  *(p2s+(size-1)) = *(p2s+maxIndex);

  p3s = (int*)realloc(p3s,size*sizeof(int));
  *(p3s+(size-1)) = *(p3s+maxIndex);

  g1s = (int*)realloc(g1s,size*sizeof(int));
  *(g1s+(size-1)) = *(g1s+maxIndex);

  g2s = (int*)realloc(g2s,size*sizeof(int));
  *(g2s+(size-1)) = *(g2s+maxIndex);

  g3s = (int*)realloc(g3s,size*sizeof(int));
  *(g3s+(size-1)) = abs(*(g3s+maxIndex)-1);
```

```
      edgeStates = (int*)realloc(edgeStates,edgeSize*sizeof(int));
      *(edgeStates+(edgeSize-1)) = size-1;
    }else{
      edgeSize--;
      tempEdgeSize--;
      for(i=maxEdgeIndex;i<edgeSize;i++){
      *(edgeStates+i) = *(edgeStates+i+1);
      }
      edgeStates = (int*)realloc(edgeStates,edgeSize*sizeof(int));
    }
  }
}


__host__ void NBlock(int N, double *d_N, double *d_reacMat, double
*d_tempMat, int p)
{
  int i;
  double *Atemp;

  cublasAlloc(N*N, sizeof(double), (void**)&Atemp);
  cublasDscal(N,scalar_N[0],d_N,N+1);
  for(i=1;i<p;i++){
    cublasDgemm('n','n',N,N,N,1.0,d_tempMat,N,d_reacMat,N,0.0,Atemp,N);
    cublasDcopy(N*N,Atemp,1,d_tempMat,1);
    cublasDaxpy(N*N,scalar_N[i],d_tempMat,1,d_N,1);
  }
  cublasFree(Atemp);
}

__host__ void DBlock(int N, double *d_D, double *d_reacMat, double
*d_tempMat, int p)
{
  int i;
  double *Atemp;

  cublasAlloc(N*N, sizeof(double), (void**)&Atemp);
  cublasDscal(N,scalar_D[0],d_D,N+1);
  for(i=1;i<p;i++){
    cublasDgemm('n','n',N,N,N,-1.0,d_tempMat,N,d_reacMat,N,0.0,Atemp,N);
    cublasDcopy(N*N,Atemp,1,d_tempMat,1);
    cublasDaxpy(N*N,scalar_D[i],d_tempMat,1,d_D,1);
  }
  cublasFree(Atemp);
}
__host__ void FSP(double k1, double g1, double k2, double g2, double k3,
double g3, double z, int p, int q, int startProb, double targetError, long
int tf)
{
  cublasStatus status;

  int i;
  int N = 64;
  int n2;
  int lda;
  int edgeSize = 0;
```

```
double tempNorm, norm, m;
double tempError = 1.0;
double *reacMat;
double *probVec;
double *identMat;

double *d_reacMat;
double *d_probVec;
double *d_D;
double *d_N;
double *d_tempMat;

FILE *file;

char fileName[20];

status = cublasInit();
if(status != CUBLAS_STATUS_SUCCESS){
  fprintf(stderr,"CUBLAS won't turn on...\n");
}

p1s = (int*)calloc(N,sizeof(int));
p2s = (int*)calloc(N,sizeof(int));
p3s = (int*)calloc(N,sizeof(int));
g1s = (int*)calloc(N,sizeof(int));
g2s = (int*)calloc(N,sizeof(int));
g3s = (int*)calloc(N,sizeof(int));
edgeStates = (int*)calloc(0,sizeof(int));

for(i=0;i<N;i++){
  *(p1s+i) = i%2;
  *(p2s+i) = (i/2)%2;
  *(p3s+i) = (i/4)%2;
  *(g1s+i) = (i/8)%2;
  *(g2s+i) = (i/16)%2;
  *(g3s+i) = (i/32)%2;
}

for(i=0;i<N;i++){
  edgeSize++;
  edgeStates = (int*)realloc(edgeStates,edgeSize*sizeof(int));
  *(edgeStates+(edgeSize-1)) = i;
}

while(1){
  m = 0.0;
  n2 = N*N;
  reacMat = addStates(k1,k2,k3,g1,g2,g3,z,N);
  probVec = (double *)calloc(N,sizeof(double));
  identMat = (double *)calloc(n2,sizeof(double));
  *(probVec+startProb) = 1.0;
  for(i=0;i<N;i++){
    *(identMat+(N*i)+i) = 1.0;
  }

  /* reserve and copy matrices and vectors */
  status = cublasAlloc(n2, sizeof(double), (void**)&d_reacMat);
```

```
if(status != CUBLAS_STATUS_SUCCESS){
  fprintf(stderr,"can't fit the reaction matrix on the card!\n");
}
status = cublasAlloc(N, sizeof(double), (void**)&d_probVec);
if(status != CUBLAS_STATUS_SUCCESS){
  fprintf(stderr,"can't fit Vi on the card!\n");
}
status = cublasAlloc(n2, sizeof(double), (void**)&d_D);
if(status != CUBLAS_STATUS_SUCCESS){
  fprintf(stderr,"can't fit D on the card!\n");
}
status = cublasAlloc(n2, sizeof(double), (void**)&d_N);
if(status != CUBLAS_STATUS_SUCCESS){
  fprintf(stderr,"can't fit N on the card!\n");
}
status = cublasAlloc(n2, sizeof(double), (void**)&d_tempMat);
if(status != CUBLAS_STATUS_SUCCESS){
  fprintf(stderr,"can't fit temporary workspace on the card!\n");
}

status = cublasSetVector(n2, sizeof(double), reacMat, 1, d_reacMat, 1);
if(status != CUBLAS_STATUS_SUCCESS){
  fprintf(stderr,"can't copy the reaction matrix to the card!\n");
}
status  = cublasSetVector(N, sizeof(double), probVec, 1, d_probVec, 1);
if(status != CUBLAS_STATUS_SUCCESS){
  fprintf(stderr,"can't copy Vi to the card!\n");
}
status  = cublasSetVector(n2, sizeof(double), identMat, 1, d_N, 1);
if(status != CUBLAS_STATUS_SUCCESS){
  fprintf(stderr,"can't copy N to the card!\n");
}
status  = cublasSetVector(n2, sizeof(double), identMat, 1, d_D, 1);
if(status != CUBLAS_STATUS_SUCCESS){
  fprintf(stderr,"can't copy D to the card!\n");
}
status  = cublasSetVector(n2, sizeof(double), identMat, 1, d_tempMat, 1);
if(status != CUBLAS_STATUS_SUCCESS){
  fprintf(stderr,"can't copy D to the card!\n");
}

/* multiply by tf */
cublasDscal(n2,(double)tf,d_reacMat,1);

/* find matrix norm */
norm = 0.0;
for(i=0;i<N;i++){
  tempNorm = cublasDnrm2(N,d_reacMat+i,N);
  status = cublasGetError();
  if(status != CUBLAS_STATUS_SUCCESS){
  printf("error - %d\n",status);
  }
  if(tempNorm > norm){
  norm = tempNorm;
  }
}
```

```
/* scaling */
if(norm > 1.0){
  m = 1.0;
  while(m < norm){
  m *= 2.0;
  }
  cublasDscal(n2, 1.0/m, d_reacMat, 1);
}

/* find blocks */
NBlock(N,d_N,d_reacMat,d_tempMat,p);
cublasDcopy(n2,d_D,1,d_tempMat,1);
DBlock(N,d_D,d_reacMat,d_tempMat,p);

/* Invert D Block */
cublasGetVector(n2,sizeof(double),d_D,1,reacMat,1);
lda = ((N+15)&~15|16);
free(identMat);
cublasFree(d_tempMat);
cublasFree(d_reacMat);

identMat = (double *)calloc(N*lda*2,sizeof(double));
status = cublasAlloc(N*lda*2, sizeof(double), (void**)&d_tempMat);
for(i=0;i<N;i++) {
  cblas_dcopy(N,(reacMat+(N*i)),1,(identMat+(lda*i*2)),1);
  identMat[lda*i*2+N+i] = 1.0;
}
cublasSetVector(lda*N*2,sizeof(double),identMat,1,d_tempMat,1);
invertge(d_tempMat,lda,N);
cublasGetVector(lda*N*2,sizeof(double),d_tempMat,1,identMat,1);

cublasFree(d_tempMat);

status = cublasAlloc(N*N, sizeof(double), (void**)&d_tempMat);
for(i=0;i<N;i++){
  cblas_dcopy(N,(identMat+(lda*i*2+N)),1,(reacMat+(N*i)),1);
}

cublasSetVector(n2,sizeof(double),reacMat,1,d_D,1);

/* Pade approximation result */
cublasDgemm('n','n',N,N,N,1.0,d_D,N,d_N,N,0.0,d_tempMat,N);
cublasDcopy(n2,d_tempMat,1,d_D,1);

/* squaring */
while(m > 1){
  m /= 2;
  cublasDgemm('n','n',N,N,N,1.0,d_D,N,d_D,N,0.0,d_tempMat,N);
  cublasDcopy(n2,d_tempMat,1,d_D,1);
}

/*multiply probability vector*/
cublasDgemv('t',N,N,1.0,d_D,N,d_probVec,1,0.0,d_tempMat,1);
cublasDcopy(N,d_tempMat,N,d_probVec,1);

/* compute error */
tempError = 1.0 - cublasDasum(N,d_probVec,1);
```

```
    printf("error - %f N - %d\n\n",tempError,N);

    cublasGetVector(N,sizeof(double),d_probVec,1,probVec,1);
    sprintf(fileName,"run%d.txt",N);
    file = fopen(fileName,"w");
    for(i=0;i<N;i++){

fprintf(file,"%d\t%d\t%d\t%d\t%d\t%d\t%lf\n",*(p1s+i),*(p2s+i),*(p3s+i),*(g1s
+i),*(g2s+i),*(g3s+i),*(probVec+i));
    }
    fclose(file);
    growStates(probVec,edgeSize,N,tempError,targetError);

    free(reacMat);
    free(identMat);
    free(probVec);
    status = cublasFree(d_D);
    status = cublasFree(d_N);
    status = cublasFree(d_tempMat);
    status = cublasFree(d_probVec);
    if((tempError < targetError) || (N > 10000)){
      free(p1s);
      free(p2s);
      free(p3s);
      free(g1s);
      free(g2s);
      free(g3s);
      free(edgeStates);
      break;
    }
  }

  status = cublasShutdown();
  if(status != CUBLAS_STATUS_SUCCESS){
    printf("can't turn it off\n");
  }
}

__host__ double factorial(int n)
{
  double result = 1.0;
  while(n > 1){
    result *= (double)n;
    n--;
  }
  return result;
}

int main(int argc, char* argv[]) {
  double k1;
  double g1 = 1.0;
  double k2;
  double g2 = 1.0;
  double k3;
  double g3 = 1.0;
  double z;
  int p = 10;
```

```
  int q = 10;
  long int tf;
  int j;
  int startProb = 0;
  int device;
  double targetError;

  if(argc != 8) {
    printf("usage - %s [k1] [k2] [k3] [z] [tf] [error] [device
#]\n",argv[0]);
    return EXIT_FAILURE;
  }
  sscanf(argv[1],"%lf",&k1);
  sscanf(argv[2],"%lf",&k2);
  sscanf(argv[3],"%lf",&k3);
  sscanf(argv[4],"%lf",&z);
  sscanf(argv[5],"%ld",&tf);
  sscanf(argv[6],"%lf",&targetError);
  sscanf(argv[7],"%d",&device);

  for(j=0;j<p;j++){
    scalar_N[j] = ( factorial(p+q-j)*factorial(p) )/(
factorial(p+q)*factorial(j)*factorial(p-j) );
    scalar_D[j] = ( factorial(p+q-j)*factorial(q) )/(
factorial(p+q)*factorial(j)*factorial(q-j) );
  }

  setToTommyDevice(device);
  FSP(k1, g1, k2, g2, k3, g3, z, p, q, startProb, targetError, tf);
  return EXIT_SUCCESS;
}
```

repressGPU.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <cublas.h>
#include <cblas.h>
#include <math.h>

/*select device*/


double scalar_N[10];
double scalar_D[10];


int *p1s;
int *p2s;
int *p3s;
int *edgeStates;

inline void __cudaSafeCall( int err, const char *file, const int line )
{
  do {
    if( err != 0) {
```

```
      fprintf(stderr, "cudaSafeCall() Runtime API error in file <%s>, line %i
: %s.\n",
              file, line, cudaGetErrorString((cudaError_t) err) );
      exit(-1);
    }
  } while (0);
}

#define SAFECALL(err) __cudaSafeCall(err, __FILE__, __LINE__)
#define CUDACHECK {cudaError_t error = cudaGetLastError(); if(error !=
0){fprintf(stderr, "Error code %d: %s file %s line
%i.\n",error,cudaGetErrorString(error),__FILE__,__LINE__);}}


void mathdispAI(const double *mat, int lda, int MAT_SIZE_h) {
  fprintf(stderr, "\n");
  int i,j;
  for (j=0;j<MAT_SIZE_h;j++)  {
    for (i=0;i<MAT_SIZE_h*2;i++) {
      fprintf(stderr, "%6.3f",mat[j*lda*2+i]);
    }
    fprintf(stderr, "\n");
  }
  fprintf(stderr, "\n");
} // mathdisp2

void mathdispAId(const double * AId, int lda, int n) {
  double * AI = new double[n*lda*2];
  cudaMemcpy(AI,AId,sizeof(double)*n*lda*2,cudaMemcpyDeviceToHost);
  mathdispAI(AI, lda, n);
  delete [] AI;
}

__global__ void GEStep1A(double * AI, int i, int n2, int lda2) {
  int k = blockIdx.x * blockDim.x + threadIdx.x;

  if (k>i && k < n2 && AI[i*lda2+k]!=0) {
    double multiplyer = -AI[i*lda2+k]/AI[i*lda2+i];
    int n = n2 / 2;
    for (int j = i+1; j < n; j++) {
      AI[j*lda2+k] += multiplyer*AI[j*lda2+i];
    }
  }
}

__global__ void GEStep2(double * AI,double diag,int i, int n2, int lda2) {
  int k = blockIdx.x * blockDim.x + threadIdx.x;
  if (k < n2) {
    AI[i*lda2+k] /= diag;
  }
}

__global__ void GEStep3(double * AI,int i, int n2, int lda2) {
  int k = blockIdx.x * blockDim.x + threadIdx.x;
  if (k > i && k < n2) {
    double multiplyer = -AI[i*lda2+k];
    for (int j = 0; j < i; j++) {
```

```
      AI[j*lda2+k] += multiplyer*AI[j*lda2+i];
    }
  }
}


void invertge(double * AI_d, int lda, int n) {
  int lda2 = lda * 2;
  // perform elementary row operations till A in AI becomes identity matrix
  for (int i = 0; i < n; i++) {
    GEStep1A<<<(int)ceil(1+(2*n-1)/32),32>>>(AI_d,i,n*2, lda2);
    CUDACHECK;
    cudaThreadSynchronize();
  }


  for (int i = n-1; i >= 0; i--) {
    double diag = 1.0;
    SAFECALL(cudaMemcpy(&diag, &AI_d[i*lda2+i], sizeof(double),
cudaMemcpyDeviceToHost));
    GEStep2<<<(int)ceil(1+(n*2-1)/32),32>>>(AI_d,diag,i,n*2, lda2);
    CUDACHECK;

    GEStep3<<<(int)ceil(1+(n*2-1)/32),32>>>(AI_d,i,n*2, lda2);
    CUDACHECK;
    cudaThreadSynchronize();
    CUDACHECK;
  }
} // invertge

__host__ void setToTommyDevice(int device) {
  cudaError_t setDeviceErrorCode;
  setDeviceErrorCode = cudaSetDevice(device);
  if(setDeviceErrorCode != cudaSuccess) {
    printf("Error setting to Tommy's
device:\n%s\n",cudaGetErrorString(setDeviceErrorCode));
    printf("Program will now exit.\n");
    exit(EXIT_FAILURE);
  }
}

__host__ void printMat(int lda, double *A)
{
  int i,j;
  for(i=0;i<lda;i++){
    printf("[");
    for(j=0;j<lda;j++){
      printf("%4.10f ",*(A+(lda*i+j)));
    }
    printf("]\n");
  }
  printf("\n");
}

__host__ double *addStates(double k1, double k2, double k3, double g1, double
g2, double g3, double b1, double b2, double b3, int size)
{
```

```
  int i,j,num,num1,num2,num3;

  double *A = (double *)calloc(size*size,sizeof(double));

  for(i=0;i<size;i++){
    for(j=0;j<size;j++){
      if(i == j){
      *(A+(j*size+i)) = -k1/(1.0+pow((double)*(p3s+i),b1)) -
k2/(1.0+pow((double)*(p1s+i),b2)) - k3/(1.0+pow((double)*(p2s+i),b3)) -
(double)*(p1s+i)*g1 - (double)*(p2s+i)*g2 - (double)*(p3s+i)*g3;
      }else{
      num = abs(*(p1s+i) - *(p1s+j)) + abs(*(p2s+i) - *(p2s+j)) +
abs(*(p3s+i) - *(p3s+j));
      if(num != 1){
        *(A+(j*size+i)) = 0.0;
      }else if(num == 1){
        num1 = *(p1s+i) - *(p1s+j);
        num2 = *(p2s+i) - *(p2s+j);
        num3 = *(p3s+i) - *(p3s+j);
        if(num1 != 0){
          if(num1 > 0){
            *(A+(j*size+i)) = (double)*(p1s+i)*g1;
          }else{
            *(A+(j*size+i)) = k1/(1.0+pow((double)*(p3s+i),b1));
          }
        }else if(num2 != 0){
          if(num2 > 0){
            *(A+(j*size+i)) = (double)*(p2s+i)*g2;
          }else{
            *(A+(j*size+i)) = k2/(1.0+pow((double)*(p1s+i),b2));
          }
        }else if(num3 != 0){
          if(num3 > 0){
            *(A+(j*size+i)) = (double)*(p3s+i)*g3;
          }else{
            *(A+(j*size+i)) = k3/(1.0+pow((double)*(p2s+i),b3));
          }
        }
      }
      }
    }
  }
  return A;
}

/*smart growth*/
__host__ void growStates(double *probVec, int &edgeSize, int &size, double
currentError, double targetError)
{
  int numAdded = 0;
  double maxNum;
  int maxIndex,maxEdgeIndex,i;
  int numFound1,numFound2,numFound3,numFound4,numFound5,numFound6;
  int tempSize = size;
  int tempEdgeSize = edgeSize;
  int numToAdd = (int)round(pow((log10(currentError)-
log10(targetError)),2.0)*1000.0 + 20.0);
```

```
  while( (numAdded < tempSize) && (numAdded < numToAdd) && (tempEdgeSize > 0)
){
    maxNum = -1.0;
    maxIndex = -1;
    maxEdgeIndex = -1;

    for(i=0;i<tempEdgeSize;i++){
      if( *(probVec+*(edgeStates+i)) > maxNum){
      maxIndex = *(edgeStates+i);
      maxNum = *(probVec+maxIndex);
      maxEdgeIndex = i;
      }
    }
    numFound1 = 0;
    numFound2 = 0;
    numFound3 = 0;
    numFound4 = 0;
    numFound5 = 0;
    numFound6 = 0;
    for(i=0;i<size;i++){
      if( (*(p1s+i) == *(p1s+maxIndex)+1) && (*(p2s+i) == *(p2s+maxIndex)) &&
(*(p3s+i) == *(p3s+maxIndex)) ){
      numFound1 = 1;
      }
      if( ((*(p1s+i) == *(p1s+maxIndex)-1) && (*(p2s+i) == *(p2s+maxIndex))
&& (*(p3s+i) == *(p3s+maxIndex))) || (*(p1s+maxIndex)-1 < 0) ){
      numFound2 = 1;
      }
      if( (*(p1s+i) == *(p1s+maxIndex)) && (*(p2s+i) == *(p2s+maxIndex)+1) &&
(*(p3s+i) == *(p3s+maxIndex)) ){
      numFound3 = 1;
      }
      if( ((*(p1s+i) == *(p1s+maxIndex)) && (*(p2s+i) == *(p2s+maxIndex)-1)
&& (*(p3s+i) == *(p3s+maxIndex))) || (*(p2s+maxIndex)-1 < 0) ){
      numFound4 = 1;
      }
      if( (*(p1s+i) == *(p1s+maxIndex)) && (*(p2s+i) == *(p2s+maxIndex)) &&
(*(p3s+i) == *(p3s+maxIndex)+1) ){
      numFound5 = 1;
      }
      if( ((*(p1s+i) == *(p1s+maxIndex)) && (*(p2s+i) == *(p2s+maxIndex)) &&
(*(p3s+i) == *(p3s+maxIndex)-1)) || (*(p3s+maxIndex)-1 < 0) ){
      numFound6 = 1;
      }
    }
    if( (numFound1 == 0) ){
      size++;
      numAdded++;
      edgeSize++;

      p1s = (int*)realloc(p1s,size*sizeof(int));
      *(p1s+(size-1)) = *(p1s+maxIndex)+1;

      p2s = (int*)realloc(p2s,size*sizeof(int));
      *(p2s+(size-1)) = *(p2s+maxIndex);
```

```
  p3s = (int*)realloc(p3s,size*sizeof(int));
  *(p3s+(size-1)) = *(p3s+maxIndex);

  edgeStates = (int*)realloc(edgeStates,edgeSize*sizeof(int));
  *(edgeStates+(edgeSize-1)) = size-1;
}else if(numFound2 == 0){
  size++;
  numAdded++;
  edgeSize++;

  p1s = (int*)realloc(p1s,size*sizeof(int));
  *(p1s+(size-1)) = *(p1s+maxIndex)-1;

  p2s = (int*)realloc(p2s,size*sizeof(int));
  *(p2s+(size-1)) = *(p2s+maxIndex);

  p3s = (int*)realloc(p3s,size*sizeof(int));
  *(p3s+(size-1)) = *(p3s+maxIndex);

  edgeStates = (int*)realloc(edgeStates,edgeSize*sizeof(int));
  *(edgeStates+(edgeSize-1)) = size-1;
}else if( (numFound3 == 0) ){
  size++;
  numAdded++;
  edgeSize++;

  p1s = (int*)realloc(p1s,size*sizeof(int));
  *(p1s+(size-1)) = *(p1s+maxIndex);

  p2s = (int*)realloc(p2s,size*sizeof(int));
  *(p2s+(size-1)) = *(p2s+maxIndex)+1;

  p3s = (int*)realloc(p3s,size*sizeof(int));
  *(p3s+(size-1)) = *(p3s+maxIndex);

  edgeStates = (int*)realloc(edgeStates,edgeSize*sizeof(int));
  *(edgeStates+(edgeSize-1)) = size-1;
}else if(numFound4 == 0){
  size++;
  numAdded++;
  edgeSize++;

  p1s = (int*)realloc(p1s,size*sizeof(int));
  *(p1s+(size-1)) = *(p1s+maxIndex);

  p2s = (int*)realloc(p2s,size*sizeof(int));
  *(p2s+(size-1)) = *(p2s+maxIndex)-1;

  p3s = (int*)realloc(p3s,size*sizeof(int));
  *(p3s+(size-1)) = *(p3s+maxIndex);

  edgeStates = (int*)realloc(edgeStates,edgeSize*sizeof(int));
  *(edgeStates+(edgeSize-1)) = size-1;
}else if(numFound5 == 0){
  size++;
  numAdded++;
  edgeSize++;
```

```
      p1s = (int*)realloc(p1s,size*sizeof(int));
      *(p1s+(size-1)) = *(p1s+maxIndex);

      p2s = (int*)realloc(p2s,size*sizeof(int));
      *(p2s+(size-1)) = *(p2s+maxIndex);

      p3s = (int*)realloc(p3s,size*sizeof(int));
      *(p3s+(size-1)) = *(p3s+maxIndex)+1;

      edgeStates = (int*)realloc(edgeStates,edgeSize*sizeof(int));
      *(edgeStates+(edgeSize-1)) = size-1;
    }else if(numFound6 == 0){
      size++;
      numAdded++;
      edgeSize++;

      p1s = (int*)realloc(p1s,size*sizeof(int));
      *(p1s+(size-1)) = *(p1s+maxIndex);

      p2s = (int*)realloc(p2s,size*sizeof(int));
      *(p2s+(size-1)) = *(p2s+maxIndex);

      p3s = (int*)realloc(p3s,size*sizeof(int));
      *(p3s+(size-1)) = *(p3s+maxIndex)-1;

      edgeStates = (int*)realloc(edgeStates,edgeSize*sizeof(int));
      *(edgeStates+(edgeSize-1)) = size-1;
    }else{
      edgeSize--;
      tempEdgeSize--;
      for(i=maxEdgeIndex;i<edgeSize;i++){
      *(edgeStates+i) = *(edgeStates+i+1);
      }
      edgeStates = (int*)realloc(edgeStates,edgeSize*sizeof(int));
    }
  }
}


__host__ void NBlock(int N, double *d_N, double *d_reacMat, double
*d_tempMat, int p)
{
  int i;
  double *Atemp;

  cublasAlloc(N*N, sizeof(double), (void**)&Atemp);
  cublasDscal(N,scalar_N[0],d_N,N+1);
  for(i=1;i<p;i++){
    cublasDgemm('n','n',N,N,N,1.0,d_tempMat,N,d_reacMat,N,0.0,Atemp,N);
    cublasDcopy(N*N,Atemp,1,d_tempMat,1);
    cublasDaxpy(N*N,scalar_N[i],d_tempMat,1,d_N,1);
  }
  cublasFree(Atemp);
}
```

```
__host__ void DBlock(int N, double *d_D, double *d_reacMat, double
*d_tempMat, int p)
{
  int i;
  double *Atemp;

  cublasAlloc(N*N, sizeof(double), (void**)&Atemp);
  cublasDscal(N,scalar_D[0],d_D,N+1);
  for(i=1;i<p;i++){
    cublasDgemm('n','n',N,N,N,-1.0,d_tempMat,N,d_reacMat,N,0.0,Atemp,N);
    cublasDcopy(N*N,Atemp,1,d_tempMat,1);
    cublasDaxpy(N*N,scalar_D[i],d_tempMat,1,d_D,1);
  }
  cublasFree(Atemp);
}
__host__ void FSP(double k1, double g1, double k2, double g2, double k3,
double g3, double b1, double b2, double b3, int p, int q, int startProb,
double targetError, long int tf)
{
  cublasStatus status;

  int i;
  int N = 8;
  int n2;
  int lda;
  int edgeSize = 0;
  double tempNorm, norm, m;
  double tempError = 1.0;
  double *reacMat;
  double *probVec;
  double *identMat;

  double *d_reacMat;
  double *d_probVec;
  double *d_D;
  double *d_N;
  double *d_tempMat;

  FILE *file;

  char fileName[20];

  status = cublasInit();
  if(status != CUBLAS_STATUS_SUCCESS){
    fprintf(stderr,"CUBLAS won't turn on...\n");
  }

  p1s = (int*)calloc(N,sizeof(int));
  p2s = (int*)calloc(N,sizeof(int));
  p3s = (int*)calloc(N,sizeof(int));
  edgeStates = (int*)calloc(0,sizeof(int));

  for(i=0;i<N;i++){
    *(p1s+i) = i%2;
    *(p2s+i) = (i/2)%2;
    *(p3s+i) = (i/4)%2;
  }
```

```
for(i=0;i<N;i++){
  edgeSize++;
  edgeStates = (int*)realloc(edgeStates,edgeSize*sizeof(int));
  *(edgeStates+(edgeSize-1)) = i;
}

while(1){
  m = 0.0;
  n2 = N*N;
  reacMat = addStates(k1,k2,k3,g1,g2,g3,b1,b2,b3,N);
  probVec = (double *)calloc(N,sizeof(double));
  identMat = (double *)calloc(n2,sizeof(double));
  *(probVec+startProb) = 1.0;
  for(i=0;i<N;i++){
    *(identMat+(N*i)+i) = 1.0;
  }

  /* reserve and copy matrices and vectors */
  status = cublasAlloc(n2, sizeof(double), (void**)&d_reacMat);
  if(status != CUBLAS_STATUS_SUCCESS){
    fprintf(stderr,"can't fit the reaction matrix on the card!\n");
  }
  status = cublasAlloc(N, sizeof(double), (void**)&d_probVec);
  if(status != CUBLAS_STATUS_SUCCESS){
    fprintf(stderr,"can't fit Vi on the card!\n");
  }
  status = cublasAlloc(n2, sizeof(double), (void**)&d_D);
  if(status != CUBLAS_STATUS_SUCCESS){
    fprintf(stderr,"can't fit D on the card!\n");
  }
  status = cublasAlloc(n2, sizeof(double), (void**)&d_N);
  if(status != CUBLAS_STATUS_SUCCESS){
    fprintf(stderr,"can't fit N on the card!\n");
  }
  status = cublasAlloc(n2, sizeof(double), (void**)&d_tempMat);
  if(status != CUBLAS_STATUS_SUCCESS){
    fprintf(stderr,"can't fit temporary workspace on the card!\n");
  }

  status = cublasSetVector(n2, sizeof(double), reacMat, 1, d_reacMat, 1);
  if(status != CUBLAS_STATUS_SUCCESS){
    fprintf(stderr,"can't copy the reaction matrix to the card!\n");
  }
  status  = cublasSetVector(N, sizeof(double), probVec, 1, d_probVec, 1);
  if(status != CUBLAS_STATUS_SUCCESS){
    fprintf(stderr,"can't copy Vi to the card!\n");
  }
  status  = cublasSetVector(n2, sizeof(double), identMat, 1, d_N, 1);
  if(status != CUBLAS_STATUS_SUCCESS){
    fprintf(stderr,"can't copy N to the card!\n");
  }
  status  = cublasSetVector(n2, sizeof(double), identMat, 1, d_D, 1);
  if(status != CUBLAS_STATUS_SUCCESS){
    fprintf(stderr,"can't copy D to the card!\n");
  }
  status  = cublasSetVector(n2, sizeof(double), identMat, 1, d_tempMat, 1);
```

```
if(status != CUBLAS_STATUS_SUCCESS){
  fprintf(stderr,"can't copy D to the card!\n");
}

/* multiply by tf */
cublasDscal(n2,(double)tf,d_reacMat,1);

/* find matrix norm */
norm = 0.0;
for(i=0;i<N;i++){
  tempNorm = cublasDnrm2(N,d_reacMat+i,N);
  status = cublasGetError();
  if(status != CUBLAS_STATUS_SUCCESS){
  printf("error - %d\n",status);
  }
  if(tempNorm > norm){
  norm = tempNorm;
  }
}

/* scaling */
if(norm > 1.0){
  m = 1.0;
  while(m < norm){
  m *= 2.0;
  }
  cublasDscal(n2, 1.0/m, d_reacMat, 1);
}

/* find blocks */
NBlock(N,d_N,d_reacMat,d_tempMat,p);
cublasDcopy(n2,d_D,1,d_tempMat,1);
DBlock(N,d_D,d_reacMat,d_tempMat,p);

/* Invert D Block */
cublasGetVector(n2,sizeof(double),d_D,1,reacMat,1);
lda = ((N+15)&~15|16);
free(identMat);
cublasFree(d_tempMat);
cublasFree(d_reacMat);

identMat = (double *)calloc(N*lda*2,sizeof(double));
status = cublasAlloc(N*lda*2, sizeof(double), (void**)&d_tempMat);
for(i=0;i<N;i++) {
  cblas_dcopy(N,(reacMat+(N*i)),1,(identMat+(lda*i*2)),1);
  identMat[lda*i*2+N+i] = 1.0;
}
cublasSetVector(lda*N*2,sizeof(double),identMat,1,d_tempMat,1);
invertge(d_tempMat,lda,N);
cublasGetVector(lda*N*2,sizeof(double),d_tempMat,1,identMat,1);

cublasFree(d_tempMat);

status = cublasAlloc(N*N, sizeof(double), (void**)&d_tempMat);
for(i=0;i<N;i++){
  cblas_dcopy(N,(identMat+(lda*i*2+N)),1,(reacMat+(N*i)),1);
}
```

```
    cublasSetVector(n2,sizeof(double),reacMat,1,d_D,1);

    /* Pade approximation result */
    cublasDgemm('n','n',N,N,N,1.0,d_D,N,d_N,N,0.0,d_tempMat,N);
    cublasDcopy(n2,d_tempMat,1,d_D,1);

    /* squaring */
    while(m > 1){
      m /= 2;
      cublasDgemm('n','n',N,N,N,1.0,d_D,N,d_D,N,0.0,d_tempMat,N);
      cublasDcopy(n2,d_tempMat,1,d_D,1);
    }

    /*multiply probability vector*/
    cublasDgemv('t',N,N,1.0,d_D,N,d_probVec,1,0.0,d_tempMat,1);
    cublasDcopy(N,d_tempMat,N,d_probVec,1);

    /* compute error */
    tempError = 1.0 - cublasDasum(N,d_probVec,1);
    printf("error - %f N - %d\n\n",tempError,N);

    cublasGetVector(N,sizeof(double),d_probVec,1,probVec,1);
    sprintf(fileName,"run%d.txt",N);
    file = fopen(fileName,"w");
    for(i=0;i<N;i++){

fprintf(file,"%d\t%d\t%d\t%lf\n",*(p1s+i),*(p2s+i),*(p3s+i),*(probVec+i));
    }
    fclose(file);
    growStates(probVec,edgeSize,N,tempError,targetError);

    free(reacMat);
    free(identMat);
    free(probVec);
    status = cublasFree(d_D);
    status = cublasFree(d_N);
    status = cublasFree(d_tempMat);
    status = cublasFree(d_probVec);
    if((tempError < targetError) || (N > 10000)){
      free(p1s);
      free(p2s);
      free(p3s);
      free(edgeStates);
      break;
    }
  }

  status = cublasShutdown();
  if(status != CUBLAS_STATUS_SUCCESS){
    printf("can't turn it off\n");
  }
}

__host__ double factorial(int n)
{
  double result = 1.0;
```

```
  while(n > 1){
    result *= (double)n;
    n--;
  }
  return result;
}

int main(int argc, char* argv[]) {
  double k1;
  double g1 = 1.0;
  double k2;
  double g2 = 1.0;
  double k3;
  double g3 = 1.0;
  double b1;
  double b2;
  double b3;
  int p = 10;
  int q = 10;
  long int tf;
  int j;
  int startProb = 0;
  int device;
  double targetError;

  if(argc != 10) {
    printf("usage - %s [k1] [k2] [k3] [b1] [b2] [b3] [tf] [error] [device
#]\n",argv[0]);
    return EXIT_FAILURE;
  }
  sscanf(argv[1],"%lf",&k1);
  sscanf(argv[2],"%lf",&k2);
  sscanf(argv[3],"%lf",&k3);
  sscanf(argv[4],"%lf",&b1);
  sscanf(argv[5],"%lf",&b2);
  sscanf(argv[6],"%lf",&b3);
  sscanf(argv[7],"%ld",&tf);
  sscanf(argv[8],"%lf",&targetError);
  sscanf(argv[9],"%d",&device);

  for(j=0;j<p;j++){
    scalar_N[j] = ( factorial(p+q-j)*factorial(p) )/(
factorial(p+q)*factorial(j)*factorial(p-j) );
    scalar_D[j] = ( factorial(p+q-j)*factorial(q) )/(
factorial(p+q)*factorial(j)*factorial(q-j) );
  }

  setToTommyDevice(device);
  FSP(k1, g1, k2, g2, k3, g3, b1, b2, b3, p, q, startProb, targetError, tf);
  return EXIT_SUCCESS;
}
```

toggleMike.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <cublas.h>
```

```
#include <cblas.h>
#include <math.h>

/*select device*/


double scalar_N[10];
double scalar_D[10];


int *p1s;
int *p2s;
int *g1s;
int *g2s;
int *edgeStates;

inline void __cudaSafeCall( int err, const char *file, const int line )
{
  do {
    if( err != 0) {
      fprintf(stderr, "cudaSafeCall() Runtime API error in file <%s>, line %i
: %s.\n",
              file, line, cudaGetErrorString((cudaError_t) err) );
      exit(-1);
    }
  } while (0);
}

#define SAFECALL(err) __cudaSafeCall(err, __FILE__, __LINE__)
#define CUDACHECK {cudaError_t error = cudaGetLastError(); if(error !=
0){fprintf(stderr, "Error code %d: %s file %s line
%i.\n",error,cudaGetErrorString(error),__FILE__,__LINE__);}}


void mathdispAI(const double *mat, int lda, int MAT_SIZE_h) {
  fprintf(stderr, "\n");
  int i,j;
  for (j=0;j<MAT_SIZE_h;j++)  {
    for (i=0;i<MAT_SIZE_h*2;i++) {
      fprintf(stderr, "%6.3f",mat[j*lda*2+i]);
    }
    fprintf(stderr, "\n");
  }
  fprintf(stderr, "\n");
} // mathdisp2

void mathdispAId(const double * AId, int lda, int n) {
  double * AI = new double[n*lda*2];
  cudaMemcpy(AI,AId,sizeof(double)*n*lda*2,cudaMemcpyDeviceToHost);
  mathdispAI(AI, lda, n);
  delete [] AI;
}

__global__ void GEStep1A(double * AI, int i, int n2, int lda2) {
  int k = blockIdx.x * blockDim.x + threadIdx.x;

  if (k>i && k < n2 && AI[i*lda2+k]!=0) {
```

```
    double multiplyer = -AI[i*lda2+k]/AI[i*lda2+i];
    int n = n2 / 2;
    for (int j = i+1; j < n; j++) {
      AI[j*lda2+k] += multiplyer*AI[j*lda2+i];
    }
  }
}

__global__ void GEStep2(double * AI,double diag,int i, int n2, int lda2) {
  int k = blockIdx.x * blockDim.x + threadIdx.x;
  if (k < n2) {
    AI[i*lda2+k] /= diag;
  }
}

__global__ void GEStep3(double * AI,int i, int n2, int lda2) {
  int k = blockIdx.x * blockDim.x + threadIdx.x;
  if (k > i && k < n2) {
    double multiplyer = -AI[i*lda2+k];
    for (int j = 0; j < i; j++) {
      AI[j*lda2+k] += multiplyer*AI[j*lda2+i];
    }
  }
}


void invertge(double * AI_d, int lda, int n) {
  int lda2 = lda * 2;
  // perform elementary row operations till A in AI becomes identity matrix
  for (int i = 0; i < n; i++) {
    GEStep1A<<<(int)ceil(1+(2*n-1)/32),32>>>(AI_d,i,n*2, lda2);
    CUDACHECK;
    cudaThreadSynchronize();
  }


  for (int i = n-1; i >= 0; i--) {
    double diag = 1.0;
    SAFECALL(cudaMemcpy(&diag, &AI_d[i*lda2+i], sizeof(double),
cudaMemcpyDeviceToHost));
    GEStep2<<<(int)ceil(1+(n*2-1)/32),32>>>(AI_d,diag,i,n*2, lda2);
    CUDACHECK;

    GEStep3<<<(int)ceil(1+(n*2-1)/32),32>>>(AI_d,i,n*2, lda2);
    CUDACHECK;
    cudaThreadSynchronize();
    CUDACHECK;
  }
} // invertge

__host__ void setToTommyDevice(int device) {
  cudaError_t setDeviceErrorCode;
  setDeviceErrorCode = cudaSetDevice(device);
  if(setDeviceErrorCode != cudaSuccess) {
    printf("Error setting to Tommy's
device:\n%s\n",cudaGetErrorString(setDeviceErrorCode));
    printf("Program will now exit.\n");
```

```
    exit(EXIT_FAILURE);
  }
}

__host__ void printMat(int lda, double *A)
{
  int i,j;
  for(i=0;i<lda;i++){
    printf("[");
    for(j=0;j<lda;j++){
      printf("%4.10f ",*(A+(lda*i+j)));
    }
    printf("]\n");
  }
  printf("\n");
}

__host__ double *addStates(int *p1s, int *p2s, int *g1s, int *g2s, double k1,
double k2, double g1, double g2, double z, int size)
{
  int i,j,num,num1,num2,num4,num5;
  double k1f = z;
  double k1r = z/k1;
  double k2f = z;
  double k2r = z/k2;

  double *A = (double *)calloc(size*size,sizeof(double));

  for(i=0;i<size;i++){
    for(j=0;j<size;j++){
      if(i == j){
      *(A+(j*size+i)) = -(double)*(g1s+i)*k1 - (double)*(g2s+i)*k2 -
(double)*(p1s+i)*g1 - (double)*(p2s+i)*g2 -
(double)*(p2s+i)*(double)*(g1s+i)*k1r - (double)*(p1s+i)*(double)*(g2s+i)*k2r
- fabs((double)*(g1s+i)-1.0)*k1f - fabs((double)*(g2s+i)-1.0)*k2f;
      }else{
      num = abs(*(p1s+i) - *(p1s+j)) + abs(*(p2s+i) - *(p2s+j)) +
abs(*(g1s+i) - *(g1s+j)) + abs(*(g2s+i) - *(g2s+j));
      if(num != 1){
        *(A+(j*size+i)) = 0.0;
      }else if(num == 1){
        num1 = *(p1s+i) - *(p1s+j);
        num2 = *(p2s+i) - *(p2s+j);
        num4 = *(g1s+i) - *(g1s+j);
        num5 = *(g2s+i) - *(g2s+j);
        if(num1 != 0){
          if(num1 > 0){
            *(A+(j*size+i)) = (double)*(p1s+i)*g1;
          }else{
            *(A+(j*size+i)) = (double)*(g1s+i)*k1;
          }
        }else if(num2 != 0){
          if(num2 > 0){
            *(A+(j*size+i)) = (double)*(p2s+i)*g2;
          }else{
            *(A+(j*size+i)) = (double)*(g2s+i)*k2;
          }
```

```
        }else if(num4 != 0){
          *(A+(j*size+i)) = (double)*(p2s+i)*(double)*(g1s+i)*k1r +
fabs((double)*(g1s+i)-1.0)*k1f;
        }else if(num5 != 0){
          *(A+(j*size+i)) = (double)*(p1s+i)*(double)*(g2s+i)*k1r +
fabs((double)*(g2s+i)-1.0)*k2f;
        }
      }
      }
    }
  }
  return A;
}

/*smart growth*/
__host__ void growStates(double *probVec, int &edgeSize, int &size, double
currentError, double targetError)
{
  int numAdded = 0;
  double maxNum;
  int maxIndex,maxEdgeIndex,i;
  int numFound1,numFound2,numFound3,numFound4,numFound5,numFound6;
  int tempSize = size;
  int tempEdgeSize = edgeSize;
  int numToAdd = (int)round(pow((log10(currentError)-
log10(targetError)),2.0)*1000.0 + 20.0);

  while( (numAdded < tempSize) && (numAdded < numToAdd) && (tempEdgeSize > 0)
){
    maxNum = -1.0;
    maxIndex = -1;
    maxEdgeIndex = -1;

    for(i=0;i<tempEdgeSize;i++){
      if( *(probVec+*(edgeStates+i)) > maxNum){
      maxIndex = *(edgeStates+i);
      maxNum = *(probVec+maxIndex);
      maxEdgeIndex = i;
      }
    }
    numFound1 = 0;
    numFound2 = 0;
    numFound3 = 0;
    numFound4 = 0;
    numFound5 = 0;
    numFound6 = 0;
    for(i=0;i<size;i++){
      if( (*(p1s+i) == *(p1s+maxIndex)+1) && (*(p2s+i) == *(p2s+maxIndex)) &&
(*(g1s+i) == *(g1s+maxIndex)) && (*(g2s+i) == *(g2s+maxIndex)) ){
      numFound1 = 1;
      }
      if( ((*(p1s+i) == *(p1s+maxIndex)-1) && (*(p2s+i) == *(p2s+maxIndex))
&& (*(g1s+i) == *(g1s+maxIndex)) && (*(g2s+i) == *(g2s+maxIndex))) ||
(*(p1s+maxIndex)-1 < 0) ){
      numFound2 = 1;
      }
```

```
      if( (*(p1s+i) == *(p1s+maxIndex)) && (*(p2s+i) == *(p2s+maxIndex)+1) &&
(*(g1s+i) == *(g1s+maxIndex)) && (*(g2s+i) == *(g2s+maxIndex)) ){
      numFound3 = 1;
      }
      if( ((*(p1s+i) == *(p1s+maxIndex)) && (*(p2s+i) == *(p2s+maxIndex)-1)
&& (*(g1s+i) == *(g1s+maxIndex)) && (*(g2s+i) == *(g2s+maxIndex))) ||
(*(p2s+maxIndex)-1 < 0) ){
      numFound4 = 1;
      }
      if( (*(p1s+i) == *(p1s+maxIndex)) && (*(p2s+i) == *(p2s+maxIndex)) &&
(*(g1s+i) == abs(*(g1s+maxIndex)-1)) && (*(g2s+i) == *(g2s+maxIndex)) ){
      numFound5 = 1;
      }
      if( (*(p1s+i) == *(p1s+maxIndex)) && (*(p2s+i) == *(p2s+maxIndex)) &&
(*(g1s+i) == *(g1s+maxIndex)) && (*(g2s+i) == abs(*(g2s+maxIndex)-1)) ){
      numFound6 = 1;
      }
    }
    if( (numFound1 == 0) && (*(g1s+maxIndex) != 0) ){
      size++;
      numAdded++;
      edgeSize++;

      p1s = (int*)realloc(p1s,size*sizeof(int));
      *(p1s+(size-1)) = *(p1s+maxIndex)+1;

      p2s = (int*)realloc(p2s,size*sizeof(int));
      *(p2s+(size-1)) = *(p2s+maxIndex);

      g1s = (int*)realloc(g1s,size*sizeof(int));
      *(g1s+(size-1)) = *(g1s+maxIndex);

      g2s = (int*)realloc(g2s,size*sizeof(int));
      *(g2s+(size-1)) = *(g2s+maxIndex);

      edgeStates = (int*)realloc(edgeStates,edgeSize*sizeof(int));
      *(edgeStates+(edgeSize-1)) = size-1;
    }else if(numFound2 == 0){
      size++;
      numAdded++;
      edgeSize++;

      p1s = (int*)realloc(p1s,size*sizeof(int));
      *(p1s+(size-1)) = *(p1s+maxIndex)-1;

      p2s = (int*)realloc(p2s,size*sizeof(int));
      *(p2s+(size-1)) = *(p2s+maxIndex);

      g1s = (int*)realloc(g1s,size*sizeof(int));
      *(g1s+(size-1)) = *(g1s+maxIndex);

      g2s = (int*)realloc(g2s,size*sizeof(int));
      *(g2s+(size-1)) = *(g2s+maxIndex);

      edgeStates = (int*)realloc(edgeStates,edgeSize*sizeof(int));
      *(edgeStates+(edgeSize-1)) = size-1;
    }else if( (numFound3 == 0) && (*(g2s+maxIndex) != 0) ){
```

```
  size++;
  numAdded++;
  edgeSize++;

  p1s = (int*)realloc(p1s,size*sizeof(int));
  *(p1s+(size-1)) = *(p1s+maxIndex);

  p2s = (int*)realloc(p2s,size*sizeof(int));
  *(p2s+(size-1)) = *(p2s+maxIndex)+1;

  g1s = (int*)realloc(g1s,size*sizeof(int));
  *(g1s+(size-1)) = *(g1s+maxIndex);

  g2s = (int*)realloc(g2s,size*sizeof(int));
  *(g2s+(size-1)) = *(g2s+maxIndex);

  edgeStates = (int*)realloc(edgeStates,edgeSize*sizeof(int));
  *(edgeStates+(edgeSize-1)) = size-1;
}else if(numFound4 == 0){
  size++;
  numAdded++;
  edgeSize++;

  p1s = (int*)realloc(p1s,size*sizeof(int));
  *(p1s+(size-1)) = *(p1s+maxIndex);

  p2s = (int*)realloc(p2s,size*sizeof(int));
  *(p2s+(size-1)) = *(p2s+maxIndex)-1;

  g1s = (int*)realloc(g1s,size*sizeof(int));
  *(g1s+(size-1)) = *(g1s+maxIndex);

  g2s = (int*)realloc(g2s,size*sizeof(int));
  *(g2s+(size-1)) = *(g2s+maxIndex);

  edgeStates = (int*)realloc(edgeStates,edgeSize*sizeof(int));
  *(edgeStates+(edgeSize-1)) = size-1;
}else if(numFound5 == 0){
  size++;
  numAdded++;
  edgeSize++;

  p1s = (int*)realloc(p1s,size*sizeof(int));
  *(p1s+(size-1)) = *(p1s+maxIndex);

  p2s = (int*)realloc(p2s,size*sizeof(int));
  *(p2s+(size-1)) = *(p2s+maxIndex);

  g1s = (int*)realloc(g1s,size*sizeof(int));
  *(g1s+(size-1)) = abs(*(g1s+maxIndex)-1);

  g2s = (int*)realloc(g2s,size*sizeof(int));
  *(g2s+(size-1)) = *(g2s+maxIndex);

  edgeStates = (int*)realloc(edgeStates,edgeSize*sizeof(int));
  *(edgeStates+(edgeSize-1)) = size-1;
}else if(numFound6 == 0){
```

```
      size++;
      numAdded++;
      edgeSize++;

      p1s = (int*)realloc(p1s,size*sizeof(int));
      *(p1s+(size-1)) = *(p1s+maxIndex);

      p2s = (int*)realloc(p2s,size*sizeof(int));
      *(p2s+(size-1)) = *(p2s+maxIndex);

      g1s = (int*)realloc(g1s,size*sizeof(int));
      *(g1s+(size-1)) = *(g1s+maxIndex);

      g2s = (int*)realloc(g2s,size*sizeof(int));
      *(g2s+(size-1)) = abs(*(g2s+maxIndex)-1);

      edgeStates = (int*)realloc(edgeStates,edgeSize*sizeof(int));
      *(edgeStates+(edgeSize-1)) = size-1;
    }else{
      edgeSize--;
      tempEdgeSize--;
      for(i=maxEdgeIndex;i<edgeSize;i++){
      *(edgeStates+i) = *(edgeStates+i+1);
      }
      edgeStates = (int*)realloc(edgeStates,edgeSize*sizeof(int));
    }
  }
}


__host__ void NBlock(int N, double *d_N, double *d_reacMat, double
*d_tempMat, int p)
{
  int i;
  double *Atemp;

  cublasAlloc(N*N, sizeof(double), (void**)&Atemp);
  cublasDscal(N,scalar_N[0],d_N,N+1);
  for(i=1;i<p;i++){
    cublasDgemm('n','n',N,N,N,1.0,d_tempMat,N,d_reacMat,N,0.0,Atemp,N);
    cublasDcopy(N*N,Atemp,1,d_tempMat,1);
    cublasDaxpy(N*N,scalar_N[i],d_tempMat,1,d_N,1);
  }
  cublasFree(Atemp);
}

__host__ void DBlock(int N, double *d_D, double *d_reacMat, double
*d_tempMat, int p)
{
  int i;
  double *Atemp;

  cublasAlloc(N*N, sizeof(double), (void**)&Atemp);
  cublasDscal(N,scalar_D[0],d_D,N+1);
  for(i=1;i<p;i++){
    cublasDgemm('n','n',N,N,N,-1.0,d_tempMat,N,d_reacMat,N,0.0,Atemp,N);
    cublasDcopy(N*N,Atemp,1,d_tempMat,1);
```

```
    cublasDaxpy(N*N,scalar_D[i],d_tempMat,1,d_D,1);
  }
  cublasFree(Atemp);
}
__host__ void FSP(double k1, double g1, double k2, double g2, double z, int
p, int q, int startProb, double targetError, long int tf)
{
  cublasStatus status;

  int i;
  int N = 16;
  int n2;
  int lda;
  int edgeSize = 0;
  double tempNorm, norm, m;
  double tempError = 1.0;
  double *reacMat;
  double *probVec;
  double *identMat;

  double *d_reacMat;
  double *d_probVec;
  double *d_D;
  double *d_N;
  double *d_tempMat;

  FILE *file;

  char fileName[20];

  status = cublasInit();
  if(status != CUBLAS_STATUS_SUCCESS){
    fprintf(stderr,"CUBLAS won't turn on...\n");
  }

  p1s = (int*)calloc(N,sizeof(int));
  p2s = (int*)calloc(N,sizeof(int));
  g1s = (int*)calloc(N,sizeof(int));
  g2s = (int*)calloc(N,sizeof(int));
  edgeStates = (int*)calloc(0,sizeof(int));

  for(i=0;i<N;i++){
    *(p1s+i) = i%2;
    *(p2s+i) = (i/2)%2;
    *(g1s+i) = (i/4)%2;
    *(g2s+i) = (i/8)%2;
  }

  for(i=0;i<N;i++){
    edgeSize++;
    edgeStates = (int*)realloc(edgeStates,edgeSize*sizeof(int));
    *(edgeStates+(edgeSize-1)) = i;
  }

  while(1){
    m = 0.0;
    n2 = N*N;
```

```
reacMat = addStates(p1s,p2s,g1s,g2s,k1,k2,g1,g2,z,N);
probVec = (double *)calloc(N,sizeof(double));
identMat = (double *)calloc(n2,sizeof(double));
*(probVec+startProb) = 1.0;
for(i=0;i<N;i++){
  *(identMat+(N*i)+i) = 1.0;
}

/* reserve and copy matrices and vectors */
status = cublasAlloc(n2, sizeof(double), (void**)&d_reacMat);
if(status != CUBLAS_STATUS_SUCCESS){
  fprintf(stderr,"can't fit the reaction matrix on the card!\n");
}
status = cublasAlloc(N, sizeof(double), (void**)&d_probVec);
if(status != CUBLAS_STATUS_SUCCESS){
  fprintf(stderr,"can't fit Vi on the card!\n");
}
status = cublasAlloc(n2, sizeof(double), (void**)&d_D);
if(status != CUBLAS_STATUS_SUCCESS){
  fprintf(stderr,"can't fit D on the card!\n");
}
status = cublasAlloc(n2, sizeof(double), (void**)&d_N);
if(status != CUBLAS_STATUS_SUCCESS){
  fprintf(stderr,"can't fit N on the card!\n");
}
status = cublasAlloc(n2, sizeof(double), (void**)&d_tempMat);
if(status != CUBLAS_STATUS_SUCCESS){
  fprintf(stderr,"can't fit temporary workspace on the card!\n");
}

status = cublasSetVector(n2, sizeof(double), reacMat, 1, d_reacMat, 1);
if(status != CUBLAS_STATUS_SUCCESS){
  fprintf(stderr,"can't copy the reaction matrix to the card!\n");
}
status  = cublasSetVector(N, sizeof(double), probVec, 1, d_probVec, 1);
if(status != CUBLAS_STATUS_SUCCESS){
  fprintf(stderr,"can't copy Vi to the card!\n");
}
status  = cublasSetVector(n2, sizeof(double), identMat, 1, d_N, 1);
if(status != CUBLAS_STATUS_SUCCESS){
  fprintf(stderr,"can't copy N to the card!\n");
}
status  = cublasSetVector(n2, sizeof(double), identMat, 1, d_D, 1);
if(status != CUBLAS_STATUS_SUCCESS){
  fprintf(stderr,"can't copy D to the card!\n");
}
status  = cublasSetVector(n2, sizeof(double), identMat, 1, d_tempMat, 1);
if(status != CUBLAS_STATUS_SUCCESS){
  fprintf(stderr,"can't copy D to the card!\n");
}

/* multiply by tf */
cublasDscal(n2,(double)tf,d_reacMat,1);

/* find matrix norm */
norm = 0.0;
for(i=0;i<N;i++){
```

```
  tempNorm = cublasDnrm2(N,d_reacMat+i,N);
  status = cublasGetError();
  if(status != CUBLAS_STATUS_SUCCESS){
  printf("error - %d\n",status);
  }
  if(tempNorm > norm){
  norm = tempNorm;
  }
}

/* scaling */
if(norm > 1.0){
  m = 1.0;
  while(m < norm){
  m *= 2.0;
  }
  cublasDscal(n2, 1.0/m, d_reacMat, 1);
}

/* find blocks */
NBlock(N,d_N,d_reacMat,d_tempMat,p);
cublasDcopy(n2,d_D,1,d_tempMat,1);
DBlock(N,d_D,d_reacMat,d_tempMat,p);

/* Invert D Block */
cublasGetVector(n2,sizeof(double),d_D,1,reacMat,1);
lda = ((N+15)&~15|16);
free(identMat);
cublasFree(d_tempMat);
cublasFree(d_reacMat);

identMat = (double *)calloc(N*lda*2,sizeof(double));
status = cublasAlloc(N*lda*2, sizeof(double), (void**)&d_tempMat);
for(i=0;i<N;i++) {
  cblas_dcopy(N,(reacMat+(N*i)),1,(identMat+(lda*i*2)),1);
  identMat[lda*i*2+N+i] = 1.0;
}
cublasSetVector(lda*N*2,sizeof(double),identMat,1,d_tempMat,1);
invertge(d_tempMat,lda,N);
cublasGetVector(lda*N*2,sizeof(double),d_tempMat,1,identMat,1);

cublasFree(d_tempMat);

status = cublasAlloc(N*N, sizeof(double), (void**)&d_tempMat);
for(i=0;i<N;i++){
  cblas_dcopy(N,(identMat+(lda*i*2+N)),1,(reacMat+(N*i)),1);
}

cublasSetVector(n2,sizeof(double),reacMat,1,d_D,1);

/* Pade approximation result */
cublasDgemm('n','n',N,N,N,1.0,d_D,N,d_N,N,0.0,d_tempMat,N);
cublasDcopy(n2,d_tempMat,1,d_D,1);

/* squaring */
while(m > 1){
  m /= 2;
```

```
      cublasDgemm('n','n',N,N,N,1.0,d_D,N,d_D,N,0.0,d_tempMat,N);
      cublasDcopy(n2,d_tempMat,1,d_D,1);
    }

    /*multiply probability vector*/
    cublasDgemv('t',N,N,1.0,d_D,N,d_probVec,1,0.0,d_tempMat,1);
    cublasDcopy(N,d_tempMat,N,d_probVec,1);

    /* compute error */
    tempError = 1.0 - cublasDasum(N,d_probVec,1);
    printf("error - %f N - %d\n\n",tempError,N);

    cublasGetVector(N,sizeof(double),d_probVec,1,probVec,1);
    sprintf(fileName,"run%d.txt",N);
    file = fopen(fileName,"w");
    for(i=0;i<N;i++){

fprintf(file,"%d\t%d\t%d\t%d\t%lf\n",*(p1s+i),*(p2s+i),*(g1s+i),*(g2s+i),*(pr
obVec+i));
    }
    fclose(file);
    growStates(probVec,edgeSize,N,tempError,targetError);

    free(reacMat);
    free(identMat);
    free(probVec);
    status = cublasFree(d_D);
    status = cublasFree(d_N);
    status = cublasFree(d_tempMat);
    status = cublasFree(d_probVec);
    if((tempError < targetError) || (N > 10000)){
      free(p1s);
      free(p2s);
      free(g1s);
      free(g2s);
      free(edgeStates);
      break;
    }
  }

  status = cublasShutdown();
  if(status != CUBLAS_STATUS_SUCCESS){
    printf("can't turn it off\n");
  }
}

__host__ double factorial(int n)
{
  double result = 1.0;
  while(n > 1){
    result *= (double)n;
    n--;
  }
  return result;
}

int main(int argc, char* argv[]) {
```

```
  double k1;
  double g1 = 1.0;
  double k2;
  double g2 = 1.0;
  int p = 10;
  int q = 10;
  long int tf;
  int j;
  int device;
  int startProb = 0;
  double z;
  double targetError;

  if(argc != 7) {
    printf("usage - %s [k1] [k2] [z] [tf] [error] [dev #]\n",argv[0]);
    return EXIT_FAILURE;
  }
  sscanf(argv[1],"%lf",&k1);
  sscanf(argv[2],"%lf",&k2);
  sscanf(argv[3],"%lf",&z);
  sscanf(argv[4],"%ld",&tf);
  sscanf(argv[5],"%lf",&targetError);
  sscanf(argv[6],"%d",&device);

  for(j=0;j<p;j++){
    scalar_N[j] = ( factorial(p+q-j)*factorial(p) )/(
factorial(p+q)*factorial(j)*factorial(p-j) );
    scalar_D[j] = ( factorial(p+q-j)*factorial(q) )/(
factorial(p+q)*factorial(j)*factorial(q-j) );
  }

  setToTommyDevice(device);
  FSP(k1, g1, k2, g2, z, p, q, startProb, targetError, tf);
  return EXIT_SUCCESS;
}
```

toggleMunsky.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <cublas.h>
#include <cblas.h>
#include <math.h>

/*select device*/


double scalar_N[10];
double scalar_D[10];


int *p1s;
int *p2s;
int *edgeStates;

inline void __cudaSafeCall( int err, const char *file, const int line )
{
```

```
  do {
    if( err != 0) {
      fprintf(stderr, "cudaSafeCall() Runtime API error in file <%s>, line %i
: %s.\n",
              file, line, cudaGetErrorString((cudaError_t) err) );
      exit(-1);
    }
  } while (0);
}

#define SAFECALL(err) __cudaSafeCall(err, __FILE__, __LINE__)
#define CUDACHECK {cudaError_t error = cudaGetLastError(); if(error !=
0){fprintf(stderr, "Error code %d: %s file %s line
%i.\n",error,cudaGetErrorString(error),__FILE__,__LINE__);}}


void mathdispAI(const double *mat, int lda, int MAT_SIZE_h) {
  fprintf(stderr, "\n");
  int i,j;
  for (j=0;j<MAT_SIZE_h;j++)  {
    for (i=0;i<MAT_SIZE_h*2;i++) {
      fprintf(stderr, "%6.3f",mat[j*lda*2+i]);
    }
    fprintf(stderr, "\n");
  }
  fprintf(stderr, "\n");
} // mathdisp2

void mathdispAId(const double * AId, int lda, int n) {
  double * AI = new double[n*lda*2];
  cudaMemcpy(AI,AId,sizeof(double)*n*lda*2,cudaMemcpyDeviceToHost);
  mathdispAI(AI, lda, n);
  delete [] AI;
}

__global__ void GEStep1A(double * AI, int i, int n2, int lda2) {
  int k = blockIdx.x * blockDim.x + threadIdx.x;

  if (k>i && k < n2 && AI[i*lda2+k]!=0) {
    double multiplyer = -AI[i*lda2+k]/AI[i*lda2+i];
    int n = n2 / 2;
    for (int j = i+1; j < n; j++) {
      AI[j*lda2+k] += multiplyer*AI[j*lda2+i];
    }
  }
}

__global__ void GEStep2(double * AI,double diag,int i, int n2, int lda2) {
  int k = blockIdx.x * blockDim.x + threadIdx.x;
  if (k < n2) {
    AI[i*lda2+k] /= diag;
  }
}

__global__ void GEStep3(double * AI,int i, int n2, int lda2) {
  int k = blockIdx.x * blockDim.x + threadIdx.x;
  if (k > i && k < n2) {
```

```
    double multiplyer = -AI[i*lda2+k];
    for (int j = 0; j < i; j++) {
      AI[j*lda2+k] += multiplyer*AI[j*lda2+i];
    }
  }
}


void invertge(double * AI_d, int lda, int n) {
  int lda2 = lda * 2;
  // perform elementary row operations till A in AI becomes identity matrix
  for (int i = 0; i < n; i++) {
    GEStep1A<<<(int)ceil(1+(2*n-1)/32),32>>>(AI_d,i,n*2, lda2);
    CUDACHECK;
    cudaThreadSynchronize();
  }


  for (int i = n-1; i >= 0; i--) {
    double diag = 1.0;
    SAFECALL(cudaMemcpy(&diag, &AI_d[i*lda2+i], sizeof(double),
cudaMemcpyDeviceToHost));
    GEStep2<<<(int)ceil(1+(n*2-1)/32),32>>>(AI_d,diag,i,n*2, lda2);
    CUDACHECK;

    GEStep3<<<(int)ceil(1+(n*2-1)/32),32>>>(AI_d,i,n*2, lda2);
    CUDACHECK;
    cudaThreadSynchronize();
    CUDACHECK;
  }
} // invertge

__host__ void setToTommyDevice(int device) {
  cudaError_t setDeviceErrorCode;
  setDeviceErrorCode = cudaSetDevice(device);
  if(setDeviceErrorCode != cudaSuccess) {
    printf("Error setting to Tommy's
device:\n%s\n",cudaGetErrorString(setDeviceErrorCode));
    printf("Program will now exit.\n");
    exit(EXIT_FAILURE);
  }
}

__host__ void printMat(int lda, double *A)
{
  int i,j;
  for(i=0;i<lda;i++){
    printf("[");
    for(j=0;j<lda;j++){
      printf("%4.10f ",*(A+(lda*i+j)));
    }
    printf("]\n");
  }
  printf("\n");
}
```

```
__host__ double *addStates(double k1, double k2, double g1, double g2, double
b1, double b2, int size)
{
  int i,j,num,num1,num2;

  double *A = (double *)calloc(size*size,sizeof(double));

  for(i=0;i<size;i++){
    for(j=0;j<size;j++){
      if(i == j){
      *(A+(j*size+i)) = -k1/(1.0+pow((double)*(p2s+i),b1)) -
k2/(1.0+pow((double)*(p1s+i),b2)) - (double)*(p1s+i)*g1 -
(double)*(p2s+i)*g2;
      }else{
      num = abs(*(p1s+i) - *(p1s+j)) + abs(*(p2s+i) - *(p2s+j));
      if(num != 1){
        *(A+(j*size+i)) = 0.0;
      }else if(num == 1){
        num1 = *(p1s+i) - *(p1s+j);
        num2 = *(p2s+i) - *(p2s+j);
        if(num1 != 0){
          if(num1 > 0){
            *(A+(j*size+i)) = (double)*(p1s+i)*g1;
          }else{
            *(A+(j*size+i)) = k1/(1.0+pow((double)*(p2s+i),b1));
          }
        }else if(num2 != 0){
          if(num2 > 0){
            *(A+(j*size+i)) = (double)*(p2s+i)*g2;
          }else{
            *(A+(j*size+i)) = k2/(1.0+pow((double)*(p1s+i),b2));
          }
        }
      }
      }
    }
  }
  return A;
}

/*smart growth*/
__host__ void growStates(double *probVec, int &edgeSize, int &size, double
currentError, double targetError)
{
  int numAdded = 0;
  double maxNum;
  int maxIndex,maxEdgeIndex,i;
  int numFound1,numFound2,numFound3,numFound4;
  int tempSize = size;
  int tempEdgeSize = edgeSize;
  int numToAdd = (int)round(pow((log10(currentError)-
log10(targetError)),2.0)*1000.0 + 20.0);

  while( (numAdded < tempSize) && (numAdded < numToAdd) && (tempEdgeSize > 0)
){
    maxNum = -1.0;
    maxIndex = -1;
```

```
    maxEdgeIndex = -1;

    for(i=0;i<tempEdgeSize;i++){
      if( *(probVec+*(edgeStates+i)) > maxNum){
      maxIndex = *(edgeStates+i);
      maxNum = *(probVec+maxIndex);
      maxEdgeIndex = i;
      }
    }
    numFound1 = 0;
    numFound2 = 0;
    numFound3 = 0;
    numFound4 = 0;
    for(i=0;i<size;i++){
      if( (*(p1s+i) == *(p1s+maxIndex)+1) && (*(p2s+i) == *(p2s+maxIndex)) ){
      numFound1 = 1;
      }
      if( ((*(p1s+i) == *(p1s+maxIndex)-1) && (*(p2s+i) == *(p2s+maxIndex)))
||  (*(p1s+maxIndex)-1 < 0) ){
      numFound2 = 1;
      }
      if( (*(p1s+i) == *(p1s+maxIndex)) && (*(p2s+i) == *(p2s+maxIndex)+1) ){
      numFound3 = 1;
      }
      if( ((*(p1s+i) == *(p1s+maxIndex)) && (*(p2s+i) == *(p2s+maxIndex)-1))
||  (*(p2s+maxIndex)-1 < 0) ){
      numFound4 = 1;
      }
    }
    if( (numFound1 == 0) ){
      size++;
      numAdded++;
      edgeSize++;

      p1s = (int*)realloc(p1s,size*sizeof(int));
      *(p1s+(size-1)) = *(p1s+maxIndex)+1;

      p2s = (int*)realloc(p2s,size*sizeof(int));
      *(p2s+(size-1)) = *(p2s+maxIndex);

      edgeStates = (int*)realloc(edgeStates,edgeSize*sizeof(int));
      *(edgeStates+(edgeSize-1)) = size-1;
    }else if(numFound2 == 0){
      size++;
      numAdded++;
      edgeSize++;

      p1s = (int*)realloc(p1s,size*sizeof(int));
      *(p1s+(size-1)) = *(p1s+maxIndex)-1;

      p2s = (int*)realloc(p2s,size*sizeof(int));
      *(p2s+(size-1)) = *(p2s+maxIndex);

      edgeStates = (int*)realloc(edgeStates,edgeSize*sizeof(int));
      *(edgeStates+(edgeSize-1)) = size-1;
    }else if( (numFound3 == 0) ){
      size++;
```

```
      numAdded++;
      edgeSize++;

      p1s = (int*)realloc(p1s,size*sizeof(int));
      *(p1s+(size-1)) = *(p1s+maxIndex);

      p2s = (int*)realloc(p2s,size*sizeof(int));
      *(p2s+(size-1)) = *(p2s+maxIndex)+1;

      edgeStates = (int*)realloc(edgeStates,edgeSize*sizeof(int));
      *(edgeStates+(edgeSize-1)) = size-1;
    }else if(numFound4 == 0){
      size++;
      numAdded++;
      edgeSize++;

      p1s = (int*)realloc(p1s,size*sizeof(int));
      *(p1s+(size-1)) = *(p1s+maxIndex);

      p2s = (int*)realloc(p2s,size*sizeof(int));
      *(p2s+(size-1)) = *(p2s+maxIndex)-1;

      edgeStates = (int*)realloc(edgeStates,edgeSize*sizeof(int));
      *(edgeStates+(edgeSize-1)) = size-1;
    }else{
      edgeSize--;
      tempEdgeSize--;
      for(i=maxEdgeIndex;i<edgeSize;i++){
      *(edgeStates+i) = *(edgeStates+i+1);
      }
      edgeStates = (int*)realloc(edgeStates,edgeSize*sizeof(int));
    }
  }
}


__host__ void NBlock(int N, double *d_N, double *d_reacMat, double
*d_tempMat, int p)
{
  int i;
  double *Atemp;

  cublasAlloc(N*N, sizeof(double), (void**)&Atemp);
  cublasDscal(N,scalar_N[0],d_N,N+1);
  for(i=1;i<p;i++){
    cublasDgemm('n','n',N,N,N,1.0,d_tempMat,N,d_reacMat,N,0.0,Atemp,N);
    cublasDcopy(N*N,Atemp,1,d_tempMat,1);
    cublasDaxpy(N*N,scalar_N[i],d_tempMat,1,d_N,1);
  }
  cublasFree(Atemp);
}

__host__ void DBlock(int N, double *d_D, double *d_reacMat, double
*d_tempMat, int p)
{
  int i;
  double *Atemp;
```

```
  cublasAlloc(N*N, sizeof(double), (void**)&Atemp);
  cublasDscal(N,scalar_D[0],d_D,N+1);
  for(i=1;i<p;i++){
    cublasDgemm('n','n',N,N,N,-1.0,d_tempMat,N,d_reacMat,N,0.0,Atemp,N);
    cublasDcopy(N*N,Atemp,1,d_tempMat,1);
    cublasDaxpy(N*N,scalar_D[i],d_tempMat,1,d_D,1);
  }
  cublasFree(Atemp);
}
__host__ void FSP(double k1, double g1, double k2, double g2, double b1,
double b2, int p, int q, int startProb, double targetError, long int tf)
{
  cublasStatus status;

  int i;
  int N = 4;
  int n2;
  int lda;
  int edgeSize = 0;
  double tempNorm, norm, m;
  double tempError = 1.0;
  double *reacMat;
  double *probVec;
  double *identMat;

  double *d_reacMat;
  double *d_probVec;
  double *d_D;
  double *d_N;
  double *d_tempMat;

  FILE *file;

  char fileName[20];

  status = cublasInit();
  if(status != CUBLAS_STATUS_SUCCESS){
    fprintf(stderr,"CUBLAS won't turn on...\n");
  }

  p1s = (int*)calloc(N,sizeof(int));
  p2s = (int*)calloc(N,sizeof(int));
  edgeStates = (int*)calloc(0,sizeof(int));

  for(i=0;i<N;i++){
    *(p1s+i) = i%2;
    *(p2s+i) = (i/2)%2;
  }

  for(i=0;i<N;i++){
    edgeSize++;
    edgeStates = (int*)realloc(edgeStates,edgeSize*sizeof(int));
    *(edgeStates+(edgeSize-1)) = i;
  }

  while(1){
```

```
m = 0.0;
n2 = N*N;
reacMat = addStates(k1,k2,g1,g2,b1,b2,N);
probVec = (double *)calloc(N,sizeof(double));
identMat = (double *)calloc(n2,sizeof(double));
*(probVec+startProb) = 1.0;
for(i=0;i<N;i++){
  *(identMat+(N*i)+i) = 1.0;
}

/* reserve and copy matrices and vectors */
status = cublasAlloc(n2, sizeof(double), (void**)&d_reacMat);
if(status != CUBLAS_STATUS_SUCCESS){
  fprintf(stderr,"can't fit the reaction matrix on the card!\n");
}
status = cublasAlloc(N, sizeof(double), (void**)&d_probVec);
if(status != CUBLAS_STATUS_SUCCESS){
  fprintf(stderr,"can't fit Vi on the card!\n");
}
status = cublasAlloc(n2, sizeof(double), (void**)&d_D);
if(status != CUBLAS_STATUS_SUCCESS){
  fprintf(stderr,"can't fit D on the card!\n");
}
status = cublasAlloc(n2, sizeof(double), (void**)&d_N);
if(status != CUBLAS_STATUS_SUCCESS){
  fprintf(stderr,"can't fit N on the card!\n");
}
status = cublasAlloc(n2, sizeof(double), (void**)&d_tempMat);
if(status != CUBLAS_STATUS_SUCCESS){
  fprintf(stderr,"can't fit temporary workspace on the card!\n");
}

status = cublasSetVector(n2, sizeof(double), reacMat, 1, d_reacMat, 1);
if(status != CUBLAS_STATUS_SUCCESS){
  fprintf(stderr,"can't copy the reaction matrix to the card!\n");
}
status  = cublasSetVector(N, sizeof(double), probVec, 1, d_probVec, 1);
if(status != CUBLAS_STATUS_SUCCESS){
  fprintf(stderr,"can't copy Vi to the card!\n");
}
status  = cublasSetVector(n2, sizeof(double), identMat, 1, d_N, 1);
if(status != CUBLAS_STATUS_SUCCESS){
  fprintf(stderr,"can't copy N to the card!\n");
}
status  = cublasSetVector(n2, sizeof(double), identMat, 1, d_D, 1);
if(status != CUBLAS_STATUS_SUCCESS){
  fprintf(stderr,"can't copy D to the card!\n");
}
status  = cublasSetVector(n2, sizeof(double), identMat, 1, d_tempMat, 1);
if(status != CUBLAS_STATUS_SUCCESS){
  fprintf(stderr,"can't copy D to the card!\n");
}

/* multiply by tf */
cublasDscal(n2,(double)tf,d_reacMat,1);

/* find matrix norm */
```

```
norm = 0.0;
for(i=0;i<N;i++){
  tempNorm = cublasDnrm2(N,d_reacMat+i,N);
  status = cublasGetError();
  if(status != CUBLAS_STATUS_SUCCESS){
  printf("error - %d\n",status);
  }
  if(tempNorm > norm){
  norm = tempNorm;
  }
}

/* scaling */
if(norm > 1.0){
  m = 1.0;
  while(m < norm){
  m *= 2.0;
  }
  cublasDscal(n2, 1.0/m, d_reacMat, 1);
}

/* find blocks */
NBlock(N,d_N,d_reacMat,d_tempMat,p);
cublasDcopy(n2,d_D,1,d_tempMat,1);
DBlock(N,d_D,d_reacMat,d_tempMat,p);

/* Invert D Block */
cublasGetVector(n2,sizeof(double),d_D,1,reacMat,1);
lda = ((N+15)&~15|16);
free(identMat);
cublasFree(d_tempMat);
cublasFree(d_reacMat);

identMat = (double *)calloc(N*lda*2,sizeof(double));
status = cublasAlloc(N*lda*2, sizeof(double), (void**)&d_tempMat);
for(i=0;i<N;i++) {
  cblas_dcopy(N,(reacMat+(N*i)),1,(identMat+(lda*i*2)),1);
  identMat[lda*i*2+N+i] = 1.0;
}
cublasSetVector(lda*N*2,sizeof(double),identMat,1,d_tempMat,1);
invertge(d_tempMat,lda,N);
cublasGetVector(lda*N*2,sizeof(double),d_tempMat,1,identMat,1);

cublasFree(d_tempMat);

status = cublasAlloc(N*N, sizeof(double), (void**)&d_tempMat);
for(i=0;i<N;i++){
  cblas_dcopy(N,(identMat+(lda*i*2+N)),1,(reacMat+(N*i)),1);
}

cublasSetVector(n2,sizeof(double),reacMat,1,d_D,1);

/* Pade approximation result */
cublasDgemm('n','n',N,N,N,1.0,d_D,N,d_N,N,0.0,d_tempMat,N);
cublasDcopy(n2,d_tempMat,1,d_D,1);

/* squaring */
```

```
    while(m > 1){
      m /= 2;
      cublasDgemm('n','n',N,N,N,1.0,d_D,N,d_D,N,0.0,d_tempMat,N);
      cublasDcopy(n2,d_tempMat,1,d_D,1);
    }

    /*multiply probability vector*/
    cublasDgemv('t',N,N,1.0,d_D,N,d_probVec,1,0.0,d_tempMat,1);
    cublasDcopy(N,d_tempMat,N,d_probVec,1);

    /* compute error */
    tempError = 1.0 - cublasDasum(N,d_probVec,1);
    printf("error - %f N - %d\n\n",tempError,N);

    cublasGetVector(N,sizeof(double),d_probVec,1,probVec,1);
    sprintf(fileName,"run%d.txt",N);
    file = fopen(fileName,"w");
    for(i=0;i<N;i++){
      fprintf(file,"%d\t%d\t%lf\n",*(p1s+i),*(p2s+i),*(probVec+i));
    }
    fclose(file);
    growStates(probVec,edgeSize,N,tempError,targetError);

    free(reacMat);
    free(identMat);
    free(probVec);
    status = cublasFree(d_D);
    status = cublasFree(d_N);
    status = cublasFree(d_tempMat);
    status = cublasFree(d_probVec);
    if((tempError < targetError) || (N > 10000)){
      free(p1s);
      free(p2s);
      free(edgeStates);
      break;
    }
  }

  status = cublasShutdown();
  if(status != CUBLAS_STATUS_SUCCESS){
    printf("can't turn it off\n");
  }
}

__host__ double factorial(int n)
{
  double result = 1.0;
  while(n > 1){
    result *= (double)n;
    n--;
  }
  return result;
}

int main(int argc, char* argv[]) {
  double k1;
  double g1 = 1.0;
```

```
  double k2;
  double g2 = 1.0;
  double b1;
  double b2;
  int p = 10;
  int q = 10;
  long int tf;
  int j;
  int startProb = 0;
  int device;
  double targetError;

  if(argc != 8) {
    printf("usage - %s [k1] [k2] [b1] [b2] [tf] [error] [device
#]\n",argv[0]);
    return EXIT_FAILURE;
  }
  sscanf(argv[1],"%lf",&k1);
  sscanf(argv[2],"%lf",&k2);
  sscanf(argv[3],"%lf",&b1);
  sscanf(argv[4],"%lf",&b2);
  sscanf(argv[5],"%ld",&tf);
  sscanf(argv[6],"%lf",&targetError);
  sscanf(argv[7],"%d",&device);

  for(j=0;j<p;j++){
    scalar_N[j] = ( factorial(p+q-j)*factorial(p) )/(
factorial(p+q)*factorial(j)*factorial(p-j) );
    scalar_D[j] = ( factorial(p+q-j)*factorial(q) )/(
factorial(p+q)*factorial(j)*factorial(q-j) );
  }

  setToTommyDevice(device);
  FSP(k1, g1, k2, g2, b1, b2, p, q, startProb, targetError, tf);
  return EXIT_SUCCESS;
}
```