Gene Expression Array Simulator

by

Yan Li

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
May 10, 2002

©Yan Li. All rights reserved.
The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part, and to grant others the right to do so.

Author

Department of Electrical Engineering and Computer Science

Certified by

Co-Director of Biological Engineering Division

Accepted by

Chairman, Department Committee on Graduate Theses

BARKER

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

JUL 31 2002

LIBRARIES
Gene Expression Array Simulator

by

Yan Li

Submitted to the Department of Electrical Engineering and Computer Science

May 10, 2002

In Partial Fulfillment of the Requirements for the Degree of Master of Engineering in Electrical Engineering and Computer Science

Abstract

DNA microarrays are a promising biotechnology, which allows the monitoring of gene expression levels in cells for thousands of genes in parallel. DNA microarray experiments are providing large quantities of genome-wide data to be analyzed. A newly developed Bayesian analytic scheme based on image analysis may offer substantial improvement over current analysis techniques. However, this method requires further enhancement and evaluation. This thesis endeavors to create a microarray simulator based on the physical processes involved in microarray fabrication to help evaluate the inference power of the new analysis tool. In addition, it proposes to use the combination of the simulator and analysis machine to explore gene expression variation as a function of physical properties.

Thesis Supervisor: Douglas Lauffenburger
Title: Professor of Chemical Engineering and Bioengineering, Co-Director of Biological Engineering Division
Acknowledgements

My advisor Douglas Lauffenburger was very helpful in giving suggestion and guidance to the thesis work. My co-advisor Keith Duggar was also very helpful for providing detailed explanation and guidance to the project, as well as for being there when I was stuck on problems throughout the thesis work.

I would also like to thank my friends who were essential for convincing me that I would eventually finish my thesis.
## Contents

1 Introduction

1.1 Background and motivation .................................................... 8
1.2 Binomial Dual Poisson model .................................................. 10
1.3 Binomial Dual Poisson inference machine ................................. 13
1.4 Gene array simulator .......................................................... 13
1.5 Overview of the thesis ......................................................... 14

2 Objectives

2.1 Gene array variability ......................................................... 15
2.2 Physical processes .............................................................. 15
2.3 Parameter interface ............................................................. 16
2.4 Extensibility ................................................................. 16

3 User’s Guide

3.1 The purpose of the simulator .................................................. 18
3.2 How to run the simulator ....................................................... 18
3.3 Simulator requirements .......................................................... 19
   3.3.1 Parameter XML file specification ................................... 19
   3.3.2 Supported parameters ................................................ 20
   3.3.3 Supported functions .................................................. 23
   3.3.4 Adding and changing parameters ................................ 23
   3.3.5 Output intensity file .................................................. 24
3.4 Generating graphics output .................................................... 25

4 Simulator Structure
4.1 Interface for the XML parameter file.................................27
4.2 Parameter file parser..........................................................29
4.3 Functor inheritance...............................................................29
  4.3.1 Functor types.............................................................31
  4.3.2 Distributions.............................................................31
  4.3.3 Standard mapping Functor........................................32
4.4 Visitor pattern.................................................................32
  4.4.1 Benefits of the Visitor pattern.......................................34
  4.4.2 Gene array structure..................................................34
  4.4.3 Visitors ..................................................................34
  4.4.4 Reporters..................................................................36
4.5 Simulating the array...........................................................36

5 Uses of the simulator

5.1 Explore effect of parameter changes on simulation outcome........38
  5.1.1 Comparing real data and simulated data........................38
    5.1.1.1 Initial trials......................................................42
    5.1.1.2 Matching real and simulated data.........................44
  5.2 Evaluation of the inference machine..................................47
    5.2.1 Analysis of false positive........................................48
    5.2.2 Analysis of false negative.......................................50

6 Discussion

6.1 Summary of simulations..................................................54
6.2 Related work..................................................................55
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.3</td>
<td>Future work</td>
<td>55</td>
</tr>
<tr>
<td>Appendix A</td>
<td>XML parameter file specification</td>
<td>57</td>
</tr>
<tr>
<td>Appendix B</td>
<td>Code</td>
<td>62</td>
</tr>
<tr>
<td>B.1</td>
<td>ArrayParam</td>
<td>62</td>
</tr>
<tr>
<td>B.2</td>
<td>ArrayParamFileParser</td>
<td>68</td>
</tr>
<tr>
<td>B.3</td>
<td>ArrayParamFunctor</td>
<td>73</td>
</tr>
<tr>
<td>B.4</td>
<td>ArrayParamFunctorConst</td>
<td>77</td>
</tr>
<tr>
<td>B.5</td>
<td>Distributions</td>
<td>78</td>
</tr>
<tr>
<td>B.6</td>
<td>Table Functor</td>
<td>95</td>
</tr>
<tr>
<td>B.7</td>
<td>Gene Array Data Structure</td>
<td>97</td>
</tr>
<tr>
<td>B.8</td>
<td>Visitors</td>
<td>102</td>
</tr>
<tr>
<td>B.9</td>
<td>Reporters</td>
<td>116</td>
</tr>
<tr>
<td>B.10</td>
<td>Simulator main() program</td>
<td>121</td>
</tr>
<tr>
<td>B.11</td>
<td>Utility Functions</td>
<td>122</td>
</tr>
<tr>
<td>B.12</td>
<td>Makefile</td>
<td>124</td>
</tr>
<tr>
<td>B.13</td>
<td>Auxiliary Functions</td>
<td>125</td>
</tr>
<tr>
<td>Appendix C</td>
<td>Sample parameter specification file (simulate.xml)</td>
<td>137</td>
</tr>
<tr>
<td>Appendix D</td>
<td>Running Xerces on Athena</td>
<td>138</td>
</tr>
<tr>
<td>Bibliography</td>
<td>141</td>
<td></td>
</tr>
</tbody>
</table>
List of Figures

Figure 1.2: BDP model........................................................................................................ 11
Figure 3.3.2: Variable computations.................................................................................. 22
Figure 3.4: Simulated null-delta-expression array............................................................. 26
Figure 4.3: Functor inheritance tree.................................................................................... 30
Figure 4.4a: Visitor and object structure........................................................................... 33
Figure 4.4b: Visitor and object collaboration...................................................................... 33
Figure 5.1.1a: Real array statistics plot: standard deviation vs. mean.............................. 39
Figure 5.1.1b: Simulated array statistics when meanNumDNAs is not optimal.................. 40
Figure 5.1.1c: Simulated array statistics when cascadeMeanR is not optimal.................... 41
Figure 5.1.1d: Best simulated array statistics plot: standard deviation vs. mean.............. 42
Figure 5.1.1.1: Spot intensity statistics with annotation.................................................. 44
Figure 5.1.1.2a: Real array histogram of spot signal log ratio........................................... 47
Figure 5.1.1.2b: Simulated array histogram of spot signal log ratio.................................. 47
Figure 5.2.1a: False positives as a function of number of pixels per spot........................... 49
Figure 5.2.1b: Percentage of false positives for all thresholds.......................................... 50
Figure 5.2.2a: False negatives for thresholds 50%, 95%, and 99%..................................... 51
Figure 5.2.2b: False negatives for all thresholds............................................................... 52
Figure 5.2.2c: Delta-expression ratios for which false negatives is 5% and 95% for all thresholds...................................................................................................................................................................................... 53

List of Tables

Table 3.3.2a: Primary parameters..................................................................................... 20
Table 3.3.2b: Deduced variables....................................................................................... 21
Table 4.1: ArrayParam interface class............................................................................... 28

List of Listings

List 3.3.5: Output intensity file.......................................................................................... 24
List 5.2: Sample result of the inference machine............................................................... 48
Chapter 1

Introduction

Gene expression provides a step toward functional characterization and grouping of new entities revealed by DNA sequencing. A commonly used tool in expression profiling is DNA microarray technology. A theory of the physical processes involved in hybridization and fluorescence imaging of DNA microarrays is developed at M.I.T., which generates a Bayesian inference machine for analysis of microarray expression measurements [Duggar et al, 2002]. The Gene Expression Array Simulator enables fabrication of array data to serve as a useful tool for evaluating the performance of the inference machine, as well as a tool for exploring gene expression variations based on physical properties.

1.1 Background and motivation

Gene expression levels in a number of different tissues or situations allow for phenotypic screening, as well as investigation of cell dynamics such as regulatory pathways and regulatory interactions. The basic principle of measurement is based on hybridization of a mixed probe derived from tissue RNA to large sets of DNA fragments representing many genes. The probe is produced by reverse transcription of mRNA and radioactive or fluorescent labeling. It contains many different sequences of cDNA in various amounts, corresponding to the numbers of copies of the original mRNA species extracted from the sample. The experiment provides simultaneous measurement of abundance for each of
the sequences represented on the array as spots, indicating the expression levels of the corresponding genes in the original sample.

This principle is implemented in a number of different forms. Current implementations include nylon macroarrays, nylon microarrays, glass microarrays and oligonucleotide chips. The ones with most promising throughput are glass microarrays and oligonucleotide chips, which have higher sensitivity and require smaller hybridization volume. However, they also have higher initial cost and high cost of instrumentation. Therefore, they are the least accessible to academic scientists. Nylon macroarrays continue to be used because of their accessibility and flexibility. Nylon microarrays, with both colorimetric and radioactive probes, offer a flexible alternative in a number of situations [Granjeaud et al, 1999].

In order to analyze the massive amount of DNA microarray data, many sophisticated analysis techniques have been developed. There are usually two steps involved in the analysis. The first step is to get meaningful values from raw image files, such as pixel-by-pixel statistical analysis of individual array spots [Brown et al, 2001]. Then the next step is to analyze those values and try to extract findings from them. These higher level methods include hierarchical clustering, mutual information schemes, and self-organizing maps [Quackenbush, 2001]. These current analysis methods offer opportunities to generate functional data on a genome-wide scale, but they have limitations that hinder rigorous analysis of array data. They are limited to simple thresholds, which results in arbitrariness, as well as in false positives and negatives. In addition, they do not allow the determination of the probability of delta-expression.
(Delta-expression is defined as either the over or under expression of a gene [Duggar et al, 2002].)

To overcome current limitations, a new logically consistent method of array quantification is developed at MIT, which allows for direct computation of delta-expression probabilities. This method uses a Bayesian analytic scheme based on image analysis. It avoids arbitrary selection of thresholds and captures all assumptions in a mathematical model -- the Binomial Dual Poisson (BDP) model [Duggar et al, 2002]. Since the method is new and offers substantial improvements over current analysis techniques for gene expression profiling, further enhancement and evaluation of the new analysis tool is necessary. This thesis designs and implements a simulation tool for evaluation of the inference machine, as well as exploring gene-expression variations based on the physical properties of the array.

### 1.2 Binomial Dual Poisson model

Binomial Dual Poisson model is based on the gene expression profiling experiment, in which cDNA samples produced from test and control cells are labeled with different colored fluors and hybridized simultaneously to microarrays. Relative levels of cDNA for each gene are then determined by comparing red and green signal intensities from the images of microarrays for each spot [Brown et al, 2001]. As illustrated in the figure below, BDP models the process of cDNA binding to microarray spots, and subsequent Poisson processes including fluorescent emission and detection.
Binomial binding stems from the presence of a two-colored cDNA population. The Poisson cascades result from the nature of fluorescence imaging. [Duggar et al, 2002]

A pool of red and green cDNA molecules reverse transcribed from mRNA cell extracts are simultaneously bound to the gene microarray chip. Each cDNA sequence binding to the microarray can either be red or green; thus the binding is binomial in nature. Then for each color, the theory models the subsequent Poisson processes including fluorescent emission and detection. Since there are two colors, the model is therefore dual Poisson. The mathematical equations for the model are shown below. Probability equation [4] holds for the entire array. It assigns the conditional probability of observing red and green signals given all the unknowns. We can thus start by examine a single pixel, then proceed outward to a spot of many pixels, and to a set of many spots. In the equations, $f$ stands for the fraction of red cDNA in the pool; $n$ stands for the total number of cDNA molecules bound to the chip; $r$ stands for the number of red cDNA molecules bound to the chip; $K$ represents all of our prior knowledge, including information such as the physical theory, the BDP model, and other auxiliary information such as the knowledge of mathematics and probability; $\mu$ stands for the average
brightness of red cDNAs; $\gamma$ stands for the ratio of red to green optical properties such as quantum efficiency and fluorescent detection; $g_r$ and $g_G$ stand for the gains of red and green signals respectively [Duggar et al, 2002].

$$B = P[r | n, f, K] = \binom{n}{r} f^r (1 - f)^{n-r}$$  \[1\]

$$P_R = P[R | \mu, r, f, K] = \frac{e^{-\mu} (\mu r)^{g_r}}{(R / g_r)!}$$  \[2\]

$$P_G = P[G | \mu, n, r, f, K] = \frac{e^{-\mu} (\mu(n-r))^G}{(G / g_G)!}$$  \[3\]

$$P[R, G, n, r | \mu, f, K] = B \times P_R \times P_G \times P[n | K]$$  \[4\]

As BDP describes, hybridization deposits cDNA on the chip. Thus, a pixel contains $n$ number of bound cDNA molecules, some of which are red. Equation [1] models this binomial distribution by giving the probability that $r$ out of $n$ cDNA molecules in a pixel are red. The remaining cDNA molecules are green. Each of these cDNA molecules then participate in the Poisson cascade ultimately producing red and green signals, $R$ and $G$ respectively. Equations [2] and [3] model the Poisson cascade, where the parameters $\mu, \gamma, g_r, g_G$ are specified as above. The form of the equations follows the well-known Poisson probability mass function. Equation [4] combines the first three equations and gives the likelihood of observing red and green signals giving all the unknowns, where $P[n | K]$ is the prior probability of $n$ [Duggar et al, 2001].
1.3 Binomial Dual Poisson inference machine

The implementation of the Binomial Dual Poisson model forms a suite of software collectively called the BDP inference machine. The inference machine is implemented entirely in ANSI compliant portable C++ and uses Bayesian analysis to infer the unknowns and assign the conditional probability of delta-expression. The machine requires as input an ASCII text file that specifies the intensity level of the red and green signals for every pixel of every spot. The number of pixels for each spot is allowed to vary. The method states that expression probability and expression degree are independent. As a result, the analysis can determine delta-expression for genes two orders of magnitudes higher in number than previous methods, because thresholds easily fail to identify genes with high delta-expression probability but low delta-expression degree [Duggar et al, 2002]. The method offers substantial improvement over current analysis techniques, simultaneously maintaining strict logical consistency; therefore, it is highly favorable. Further enhancement of its performance and applicability is to be developed.

1.4 Gene array simulator

We develop an informatic chain for DNA microarray data to describe specific scientific hypotheses, so that we can infer parameters from observed quantities. The informatic chain assesses the experimental variability in mRNA and cDNA preparation, hybridization, and array imaging. To gain further insight into the informatic chain, we designed and implemented an array simulator. The simulator incorporates all of the physical processes known to us, including the binomial binding and the Poisson cascade components, as well as hypothetical processes such as array surface variation. The
simulator takes parameters that characterize these processes as input and simulates a microarray signal intensity data. The BDP inference machine can then process the simulated signal intensity output, and infer the probabilities of the simulation parameters. Additional tools can generate graphical representations of the simulated array. Constructing the simulator allows us to evaluate the accuracy of the inferences since the values of these parameters are known beforehand. The simulator helps to gauge the efficiency and performance of the analysis engine. It allows the evaluation of various approximation schemes as well as exploration of their range of validity. Furthermore, it allows estimation of the rate of false positives and negatives as a function of the array parameters. This information is useful for experimental design of future chips.

1.5 Overview of the thesis

This thesis is organized as follows: Chapter 2 states the objectives of designing and implementing the simulator in detail. Chapter 3 is a user’s guide to the simulator. It describes how to operate the simulator and various requirements. Chapter 4 explains the structure of the simulator and contains design and implementation details. Chapter 5 explores the uses of the simulator. Chapter 6 contains a conclusion and presents suggestion for future work.
Chapter 2

Objectives

This chapter describes the objectives for designing and implementing the simulator. The simulator is designed to capture the observed variability of real gene array. It also incorporates known physical processes such as the binomial cDNA binding process and the Poisson cascade processes of fluorescence imaging. In addition, the simulator provides an interface to the parameters governing these known processes, supports extensions of the physical processes, and provides hooks for hypothetical processes.

2.1 Gene array variability

Real gene expression array measurements have certain degree of variability, which results from the nature of the physical process producing the array and actual biological variation. Such processes include production of the microarray, hybridization, fluorescent emission and detection. In order to generate data that are comparable to those collected from real microarrays, the simulator must be able to reproduce the variability in a systematic and controlled manner. This can be achieved by having the simulator model a series of defined processes.

2.2 Physical processes

The known processes of gene expression data generation can be broken down to three steps. The first step is the manufacturing of the DNA array chip. During this step, DNA
in solution is transferred to glass slides to create a dense array of spots. The number of spots on the chip is known; however, the number of DNA strands in a particular spot is non-deterministic and must be modeled.

The second step is hybridization, during which the test and control cDNA molecules in solution bind with complimentary spots on the DNA array. The binding process can be characterized as binomial -- each DNA strand on the array can either bind or not bind to a cDNA molecule in the solution [Duggar et al, 2002]. The fraction of test cDNA molecules in the solution influences the ratio of test to control cDNA bound to a particular spot. The more test cDNA molecules there are in the solution, the more likely they will bind to a spot on the array.

2.3 Parameter interface

Another purpose of the simulator is to provide an interface to the parameters governing the physical processes. The ability to specify parameter values provides control when producing simulated arrays. Examples of these parameters are the total number of cDNA molecules in a pixel and the mean brightness of a red cDNA under certain imaging conditions. These parameters are not directly modeled by the inference machine, but they do partially determine the observed outcome. As an interface to these parameters, we define an Extensible Markup Language (XML) document to structure them.

2.4 Extensibility

To aid the scientific investigation and analysis of gene expression, the simulator supports various extensions. Hooks are provided, so users can easily modify relevant sections of
code. For example, additional probability distributions can be implemented modularly and plugged easily into the existing system. In addition, computing is confined to a small set of functions via the Visitor pattern [Gamma et al., 1995], so that additional statistics and other analysis methods can be implemented as separate classes. Finally, the user can define hypothetical processes that we haven't considered by utilizing these existing structures. The extensibility aids the incorporation of new knowledge gained from additional scientific research.
Chapter 3

User’s Guide

This chapter contains a brief introduction to the simulator, instructions on how to run the simulation program and generate graphical output, and requirements for the simulator, such as the input and output files specifications.

3.1 The purpose of the simulator

As described in Chapter 2, the design and implementation of the simulator has several objectives. The initial objective is to serve as a useful tool for evaluating the BDP inference machine. Using the simulator, we can determine the reliability of the inference machine. In addition, the simulator can be used as a tool for further gene expression investigation because it captures the physical processes producing array measurements, allows users to alter parameter values, and attempts to achieve simulation results comparable to real data.

3.2 How to run the simulator

The executable program can run in two modes: normal execution mode and debug mode. Both modes are run on the command line. In debug mode, the command is

```
%sim  input_file_name  output_file_name  test_file_name
```

In normal mode, the command is

```
%sim  input_file_name  output_file_name
```
In debug mode, a test file is constructed which contains the numerical information for each pixel of the simulated array. This information includes the number of DNA molecules of the pixel, the red and green intensity and other parameter values for each pixel. The size of the test file can potentially be huge if the number of genes is large. It is not recommended to run the program in debug mode unless the number of genes is small.

In normal mode no test file is specified or generated. For ten thousand genes, the program finishes execution in about three minutes on a 1Ghz machine. Program execution generates an output file with the red and green intensity levels for each pixel of the array. The inference machine can use this intensity file as its input for gene expression analysis.

3.3 Simulator requirements

The simulator has specific requirements for the input and output files, parameters and functions needed for simulation.

3.3.1 Parameter input file specification

Input parameters are specified as Extensible Markup Language (XML). XML is a universal format for structured documents and data. It is a set of rules for designing text markup. It specifies unambiguous structures that are easily parsed. Since many parameters for the simulator have different properties and data structures, XML is an ideal format as our parameter input file. A detailed specification for the input parameter file can be found in Appendix A.
By choosing XML as the input file format, we gain access to a large and growing community of tools and engineers experienced in the technology. An example is Xerces developed by Apache. It is an open source software package designed for parsing XML files. We utilize this package for our own implementation.

3.3.2 Supported parameters

The primary parameters in the input file are listed in the following table.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>numGenes</td>
<td>number of genes of an array</td>
</tr>
<tr>
<td>numSpots</td>
<td>number of spots for each gene</td>
</tr>
<tr>
<td>numPixels</td>
<td>number of pixels for each spot</td>
</tr>
<tr>
<td>meanNumDNAs</td>
<td>mean number of DNA molecules for each pixel</td>
</tr>
<tr>
<td>fractionR</td>
<td>fraction of red cDNA molecules in the pool</td>
</tr>
<tr>
<td>probBind</td>
<td>probability of a DNA on the chip binding to a cDNA in the pool</td>
</tr>
<tr>
<td>cascadeMeanR</td>
<td>Poisson cascade mean value of red photons emitted per pixel</td>
</tr>
<tr>
<td>cascadeMeanRatio</td>
<td>cascade mean ratio of green to red photons emitted per pixel</td>
</tr>
<tr>
<td>probFilterRtoR, probFilterGtoG</td>
<td>probability of filtering a red/green photon into the red/green channel</td>
</tr>
<tr>
<td>probDetectR, probDetectG</td>
<td>probability of detecting a red/green photon in the channels</td>
</tr>
<tr>
<td>gainR, gainG</td>
<td>gain of red/green channel signals</td>
</tr>
</tbody>
</table>

Table 3.3.2a Primary parameters
Besides these parameters, there are variables deduced from the parameters via a mathematical model or function. Following is a list of these variables and their description (Table 3.3.2b). A flow chart of the sequence of steps of how the variables are computed is also shown below in Figure 3.3.2.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n_{DNA}</td>
<td>number of DNA molecules per pixel</td>
</tr>
<tr>
<td>n_{cDNA}</td>
<td>number of cDNA molecules per pixel</td>
</tr>
<tr>
<td>n_{RcDNA}</td>
<td>number of red cDNA molecules per pixel</td>
</tr>
<tr>
<td>n_{GcDNA}</td>
<td>number of green cDNA molecules per pixel</td>
</tr>
<tr>
<td>\mu_{nRphot}</td>
<td>mean number of red photons emitted from a pixel ,</td>
</tr>
<tr>
<td>\mu_{nGphot}</td>
<td>mean number of green photons emitted from a pixel</td>
</tr>
<tr>
<td>n_{Rphot}</td>
<td>number of red photons emitted from a pixel</td>
</tr>
<tr>
<td>n_{Gphot}</td>
<td>number of green photons emitted from a pixel</td>
</tr>
<tr>
<td>n^f_{filt}</td>
<td>number of red photons filtered to red channel,</td>
</tr>
<tr>
<td>n^g_{filt}</td>
<td>number of red photons filtered to green channel,</td>
</tr>
<tr>
<td>n^g_{filt}</td>
<td>number of green photons filtered to red channel,</td>
</tr>
<tr>
<td>n^f_{filt}</td>
<td>number of green photons filtered to green channel</td>
</tr>
<tr>
<td>n^r_{det}</td>
<td>number of red photons detected in red channel,</td>
</tr>
<tr>
<td>n^g_{det}</td>
<td>number of red photons detected in green channel,</td>
</tr>
<tr>
<td>n_{Rchan}</td>
<td>total number of photons detected in the red channel</td>
</tr>
<tr>
<td>n_{Gchan}</td>
<td>total number of photons detected in the green channel</td>
</tr>
<tr>
<td>R</td>
<td>red intensity counts for a pixel</td>
</tr>
<tr>
<td>G</td>
<td>green intensity counts for a pixel</td>
</tr>
</tbody>
</table>

Table 3.3.2b Deduced variables
meanNumDNAs ($\mu_{nDNA}$)

$n_{DNA} = \text{LogNormal} (\mu_{nDNA}, \gamma \mu_{nDNA})$,
(\gamma is a multiplier to be explained later)

$n_{cDNA} = \text{Binomial} (n_{DNA}, \text{probBind})$

$n_{RcDNA} = \text{Binomial} (n_{cDNA}, \text{fractionR})$

$n_{GcDNA} = n_{cDNA} - n_{RcDNA}$

$\mu_{nRphot} = n_{RcDNA} \times \text{cascadeMeanR}$

$n_{Rphot} = \text{Poisson} (\mu_{nRphot})$

$n^{R}_{filt} = \text{Binomial} (n_{Rphot}, \text{probFilterRtoR})$

$n^{G}_{filt} = \text{Binomial} (n_{Gphot}, 1 - \text{probFilterGtoG})$

$n^{R}_{det} = \text{Binomial} (n^{R}_{filt}, \text{probDetectR})$

$n^{G}_{det} = \text{Binomial} (n^{G}_{filt}, \text{probDetectG})$

$n_{Rchan} = n^{R}_{det} + n^{G}_{det}$

$R = n_{Rchan} \times \text{gainR}$

$\mu_{nGphot} = n_{GcDNA} \times \text{cascadeMeanR} \times \text{cascadeMeanRatio}$

$n_{Gphot} = \text{Poisson} (\mu_{nGphot})$

$n^{G}_{filt} = \text{Binomial} (n_{Gphot}, \text{probFilterGtoG})$

$n^{R}_{filt} = \text{Binomial} (n_{Rphot}, 1 - \text{probFilterRtoR})$

$n^{G}_{det} = \text{Binomial} (n^{G}_{filt}, \text{probDetectG})$

$n^{R}_{det} = \text{Binomial} (n^{R}_{filt}, \text{probDetectR})$

$n_{Gchan} = n^{R}_{det} + n^{G}_{det}$

$G = n_{Gchan} \times \text{gainG}$

Figure 3.3.2 Variable computation

22
In the chart, we represent a random deviate by specifying the distribution name, then the relevant distribution parameters. For example, \( n_{cDNA} = \text{Binomial}(n_{DNA}, \text{probBind}) \), means \( n_{cDNA} \) is a binomial deviate with \( n_{DNA} \) as the number of trials and \( \text{probBind} \) as probability of success for the distribution.

### 3.3.3 Supported functions

The simulator currently supports three types of functions for specifying parameter values in the input file. The simplest function is a constant function, which returns the value read in from the parameter file for all arguments. The second type of function is a distribution. The implemented distributions include uniform, binomial, beta, Poisson, Gaussian (normal), and lognormal. Distributions return pseudo random deviates based on their given parameters. The third type of function is a standard mapping that takes an argument and returns a result based on the argument value. An implemented example of this type of function is a table function, which takes in a series of key and value pairs and step-interpolates values for omitted keys. This type of function is useful when the user wants to specify a list of values for a parameter.

### 3.3.4 Adding and changing structures

Since the simulator is designed to aid scientific investigation of gene array analysis, adding and changing its structure should be made as simple as possible. To add new parameters, the user needs to add features to the parameter file parser for it to recognize those additional parameters. Because the simulator is implemented modularly by integrating a number of different classes, new structures can be added modularly based
on inheritance. For example, to add a new distribution model, a new distribution class can
be created separately, as long as it supports the distribution function interface.

To change an existing parameter’s name, the user needs to edit the parameter file
parser to recognize the change in parameter name. Since the name of a parameter and its
type are defined separately, if the structure of the parameter remains unchanged, then no
additional changes to the implementation need to be made. A parameter can be defined as
a constant function, a distribution, or a function with arguments. This flexibility reduces
possible future changes to the implementation. To change an existing structure, for
example a distribution, one only needs to edit the corresponding distribution class. In
general, changing interfaces should be avoided, as multiple classes may rely upon them
for access.

3.3.5 Output intensity file

The simulator generates an output file, which contains the red and green signal intensity
counts for each pixel. The user specifies the name of this file when running the program
at the command line. The format of this intensity file adheres to the format required by
the inference machine. Currently, the intensity file is a simple text file that contains the
intensity counts for each pixel of each spot of each gene cohort. A sample intensity file is
shown below, where the number following “Intensities” is the total number of pixels for
that spot.

<table>
<thead>
<tr>
<th>Gene 1</th>
<th>Spot 1</th>
<th>Intensities 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1450</td>
<td>1166</td>
<td></td>
</tr>
<tr>
<td>1862</td>
<td>1634</td>
<td></td>
</tr>
<tr>
<td>2096</td>
<td>1236</td>
<td></td>
</tr>
<tr>
<td>1710</td>
<td>2186</td>
<td></td>
</tr>
<tr>
<td>1390</td>
<td>1134</td>
<td></td>
</tr>
</tbody>
</table>
3.4 Generating graphics output

To generate a graphical representation from the intensity file, we can run a bitmap construction program [Duggar et al, 2002].

`%outlier intensity.bmp < intensity_file_name`

The bitmap file `intensity.bmp` is generated based on the red and green intensity for each pixel in the `intensity file`. The bitmap file can be opened using any image viewer. As an example, we utilized the simulator to fabricate a hypothetical data set from 256 spots, with `fractionR` equals to 0.5 for every spot in the array. Since this corresponds to an expectation of zero delta-expression, the bit map image of this null-delta-expression array can been seen as generally yellow, which represents an equal mixture of red and green intensities.
Figure 3.4  Simulated null-delta-expression array
The variability in both total intensity and color ratio is apparent and similar to real arrays.
Chapter 4

Simulator Structure

This chapter explains the design and implementation of the simulator. It gives a detailed description of the C++ implementation of class structures. This chapter is intended for readers with programming experience who wish to gain a deeper understanding of and extend features for the simulator.

4.1 Interface for the XML parameter file

An interface (ArrayParam) is designed to hold parameter values directly read from the XML input file. The purpose of this interface is to expose parameter values to other simulator structures. The XML file not only specifies parameter type and values, but also contains extra tagging and comments. The simulator program itself only cares about the numerical information of the parameters. So the interface is designed to hold only this information, discarding the tags and other extra information of the XML file. The interface also allows certain flexibility in the XML file, since access of parameters by other structures of the simulator is constrained to the well-defined public methods of the interface class. The code for the parameter interface class (ArrayParam.hh, ArrayParam.cc) can be found in Appendix B.1. A drawing for the ArrayParam class, including its access methods is shown below in Table 4.1.
ArrayParam interface class

A copy constructor is implemented for the interface class. The copy constructor is implemented with destructive copy semantics for robustness because only one copy of
the array parameters is useful for the simulator at a time. Because each private member of
the ArrayParam class is dynamically allocated, the class copy constructor must take care
of deleting the pointers before assigning new ownership for the private members.

4.2 Parameter file parser

The first step of simulation is to parse the input XML file. To accomplish this task, we
choose to use Xerces (developed by Apache, http://xml.apache.org/xerces-c/index.html),
an existing XML parser, due to the complexity of parsing XML. XML enhances
robustness because it is standardized and supports many features. Some of these features
may not be relevant to the simulator. Xerces handles these extra features and provides
easy access to the information we need.

Xerces parses the XML file and produces a representation to the document
structure. We can then access this representation via the DOM interface. We construct a
wrapper (ArrayParamFileParser) to provide a syntax that is more convenient to simulator
users. The code for the parameter file parser can be found in Appendix B.2
(ArrayParamFileParser.hh, ArrayParamFileParser.cc). To use Xerces on Athena File
Systems at MIT, please see Appendix D.

4.3 Functor inheritance

Since the input parameters can be defined as different functions, and these functions have
certain common features, we define an inheritance tree for the functions. A diagram of
the Functor inheritance tree is illustrated below:
We define an abstract interface ArrayParamFunctor class to be the super class that all other concrete Functors inherit from. The overloaded operator() is the most important universal feature of the class, and it is implemented by all concrete Functors. ArrayParamFunctor is a template class since the return type of the overloaded operator can be configured. In our case, the operator returns either an int or a double. The public methods createDistr() and createFunc() dynamically allocate instances of concrete Functors. These Functors can then be passed as arguments to the accessor methods of the parameter interface class ArrayParam. The code for ArrayParamFunctor (ArrayParamFunctor.hh, ArrayParamFunctor.cc) can be found in Appendix B.3.
4.3.1 Functor types

There are three types or groups of concrete Functors that inherit from the interface class ArrayParamFunctor. They are the constant Functor (ArrayParamFunctorConst), the distribution classes, and the table Functor (TableFunc). The constant Functor as mentioned before, ignores the arguments. Its operator() returns the value read directly from the parameter input file. The code for the constant Functor (ArrayParamFunctorConst.hh, ArrayParamFunctorConst.cc) can be found in Appendix B.4.

4.3.2 Distributions

We have implemented a number of distributions for gene array investigation. Each distribution class inherits directly from ArrayParamFunctor, although they can be grouped together as distributions. We eliminated the intermediate abstract interface ArrayParamFunctorDistr in our initial design because this additional interface provides no actual implementation benefit, although it makes the grouping of various distributions conceptually clearer. The distribution classes include uniform (UniformDistr), binomial (BinomDistr), beta (BetaDistr), Poisson (PoissDistr), Gaussian (GaussDistr), and lognormal (LogNormalDistr). The core implementations for the binomial, Poisson and Gaussian distributions use variants of methods described in the book “Numerical Recipes in C” (second edition). Implementations for other distributions, as well as some auxiliary helper functions (Random.hh, Random.cc, functions.hh in Appendix B.13) are derived in house with the help of my advisor Keith Duggar. All of the distribution classes ignore their argument values. Their overloaded operator(), which contains the core mathematical
derivations, returns a deviate of the distribution. The code for all the distributions can be found in Appendix B.5.

4.3.3 Standard mapping Functor

Currently, the only Functor that utilizes its argument values is the table Functor (TableFunc). The constructor of TableFunc reads in key-value pairs as specified by a table in the parameter input file. The overloaded operator $(i)$ then step-interpolates values for the missing keys and returns a value given a specific key. The specific key used to retrieve a value from the table is indicated by argument $i$. Current implementation of the table Functor accepts all arguments but looks only at one -- the key. The code for the table Functor (TableFunc.hh) can be found in Appendix B.6.

4.4 Visitor pattern

We use a programming design pattern [Gamma et al, 1995] -- Visitor pattern -- for computations given the parameter values. Visitor pattern represents an operation to be performed on the elements of an object structure. It lets the user define a new operation without changing the classes of the elements on which it operates. The Visitor pattern packages related operations for each class in a separate object, called a Visitor, and passes it to elements of the data tree as it is traversed. When an element accepts the Visitor, it sends a request to the Visitor that encodes the element's class. The Visitor then executes the operation for that element. With the Visitor pattern, two class hierarchies are usually defined: one for the elements being operated on and one for the Visitors that define operations on the elements. A new operation can be created by adding a new
subclass to the Visitor class hierarchy, so that we can add new functionality simply by defining new Visitor subclasses. Examples of Visitor and object hierarchy and collaboration are shown below.

![Object Structure Diagram](image)

<table>
<thead>
<tr>
<th>Element</th>
<th>Accept(Visitor)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConcreteElementA</td>
<td>Accept(Visitor v) OperationA()</td>
</tr>
</tbody>
</table>

![Visitor Structure Diagram](image)

<table>
<thead>
<tr>
<th>Visitor</th>
<th>VisitElementA(Visitor)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConcreteVisitor</td>
<td>VisitElementA(Visitor)</td>
</tr>
</tbody>
</table>

Figure 4.4a Visitor and object structure [Gamma et al, 1995]

![Visitor and Object Collaboration Diagram](image)

<table>
<thead>
<tr>
<th>anObjectStructure</th>
<th>ElementA</th>
<th>aConcreteVisitor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accept(aVisitor)</td>
<td>VisitElementA(Visitor)</td>
<td>OperationA()</td>
</tr>
</tbody>
</table>

Figure 4.4b Visitor and object collaboration [Gamma et al, 1995]
4.4.1 Benefits of the Visitor pattern

Visitor pattern is usually applied when the classes defining the object structure rarely change, but new operations are often defined over the structure. In our case, we know that gene array data structure is fairly static and unlikely to change; on the other hand, we may need to keep adding different computations for our investigations. Visitors make adding new operations easy, gather related operations and separate unrelated ones, work across object structures with different types of elements, avoid the use of global variables by accumulating states as they visit each element in the object structure [Gamma et al, 1995].

4.4.2 Gene array structure

The data structure for the simulated array is constructed via a series of classes, GeneArray, Gene, GeneSpot, and GenePixel. The definition of the classes is obvious from their names. A gene array contains a vector of genes, a gene contains a vector of spots, and a spot contains a vector of pixels. Each of the classes implements the accept() methods based on the Visitor pattern. The accept() method takes as argument a Visitor, and calls the Visitor’s visit() method, asking the Visitor to visit the structure (array, gene, spot, or pixel) itself. The code for the data structure classes can be found in Appendix B.7.

4.4.3 Visitors

All the concrete Visitors inherit from the Visitor super class. The Visitor super class has default implementation for its visit() methods. Its visit() methods steps through the data structure tree until it gets to the leaf node – pixel. The default visit() methods are
implemented to avoid code replication because some concrete Visitors also need to step through the tree until they get to the pixels. A concrete Visitor class can overwrite the default visit() methods.

The concrete Visitors that are relevant for array simulation are ConstructArrayVisitor, ComputeCDNAVisitor, CompChanPhotonsVisitor, and ComputeRGVisitor. ConstructArrayVisitor builds the whole gene array structure based on the parameters read from the input file. By calling the ArrayParam accessor methods, ConstructArrayVisitor fills the data structure with necessary values such as constants, deviates from the distributions, or step-interpolated values from a table. It constructs the array with specified number of genes, spots, pixels, and number of DNA molecules per pixel. The Visitor also stores parameter values such as probBind and fractionR in each pixel for further computations.

Once the gene array is constructed, a series of computations are done via ComputeCDNAVisitor, CompChanPhotonsVisitor, and ComputeRGVisitor to generate the red and green intensity counts for each simulated pixel. ComputeCDNAVisitor and CompChanPhotonsVisitor are intermediate steps for computing intensities. Breaking the computation of intensity counts into different steps gives the flexibility of retrieving intermediate values from the computation. The code for the Visitors can be found in Appendix B.8.

4.4.4 Reporters

Another type of objects that works with Visitors is Reporter. Reporters accumulate the results of Visitors and when called, reports the accumulated result. Currently
implementation of Reporter has a \textit{reportStats()} method, which accepts a Visitor as its argument, and simply pushes the Visitor onto a stack to keep track of which Visitors' results it should later report. The actual reporting is done via the overloaded output insertion operator\texttt{\textless\textless}. The output insertion operator\texttt{\textless\textless} writes the results of the various Visitors into a report file with a desired format. The file can then be opened to view the report. The code for the Reporters can be found in \textit{Appendix B.9}.

\section*{4.5 Simulating the array}

We can now proceed to simulate a gene array by wrapping everything together to create a top-level simulation program. This program has a \textit{main()} function that accomplishes the simulation in three steps: parses the input parameter file, computes array data, and outputs a result intensity file.

To parse the input parameter file, we first instantiated an ArrayParamFileParser. Then call the \texttt{parse()} method of the parser, passing in the input file as an argument. After the input file is parsed, we are ready to simulate the gene array. A function \textit{simulate()} is implemented to group the steps necessary for computing array data.

To simulate, we first need to construct the gene array given the input parameters. Then we need to compute the pixel intensities based on the gene array's structural data and parameters. The calculation for intensities is broken down to three steps: computing the cDNA molecules for each pixel, computing the number of photons in the red and green channels of every pixel, and computing the red and green intensity counts for each pixel. These three steps are done via the three concrete Visitors as described above in the Visitors section 4.4.3.
The last step of simulation is to output the pixel intensities to a file whose name is specified by the user on the command line. The output file writing is done via an overloaded output insertion operator<<. The operator takes a gene array element (GeneArray, Gene, GeneSpot or GenePixel) as one of the arguments, and writes relevant contents of that element to the file. Writing is done recursively as the output operator<< for GeneArray is called first in the main program. The code for the simulation program (sim.cc) can be found in Appendix B.10.

In addition, we can also use a StatsReporter to report statistics of the calculation and output a report file. This will be discussed later when we talk about the use of the simulator.
Chapter 5

Uses of the Simulator

This chapter explores the uses of the simulator. Current simulator uses include helping us gain theoretical insight into the informatic chain modeling the production of microarray data and evaluating the performance of the inference machine. Furthermore, by analyzing the effect of parameter changes on simulation outcome, we can gain useful information that helps to improve future microarray experimental design.

5.1 Explore effect of parameter changes on simulation outcome

Sources and variability affect the data we observe. To gain insight to the physical processes involved in the informatic chain, we want to first get intuition from the raw data. In order to generate data that are suitable for inference machine analysis, a necessary requirement for simulation outcome is to have the data qualitatively similar to real data. We achieve this by first comparing the spot total intensity variation between real and simulated data, then varying parameter values to make simulation outcome as close to real data as possible.

5.1.1 Comparing real and simulated data

As a representative source of microarray data, we use the image of a real spotted microarray containing 6624 elements as our simulation target. To analyze the variation of its spot intensities, we plotted a diagram (Figure 5.1.1a) of total spot intensity standard
deviation versus total spot intensity mean. Total spot intensity mean and standard deviation are defined as

\[
\mu = \frac{\sum x_i}{N} \quad \sigma = \sqrt{\frac{\sum (x_i - \mu)^2}{N}}
\]  

[5]

where \( x_i \) is the total intensity of a pixel, \( \mu \) is the mean intensity of the spot, and \( N \) is the number of pixels in that spot.

Figure 5.1.1a Real array statistics plot: standard deviation vs. mean

This plot allows us to observe that pixel-to-pixel variation in fluorescence intensity scales linearly with total spot signal intensity, rather than with the square root of intensity, as is expected for normally distributed signals subject to "shot noise" [Brown et al, 1998]. To capture the origin of this variation, we vary parameters for the simulator to gain intuition of their effect and generate data closer to the real data. Using the simulator, we discovered that variability in the cascade-mean and number of cDNA molecules per
pixel dominate the overall distribution of these statistics. Assigning different variations for these two parameters can dramatically change the spread and location of statistics distribution.

We show some of these changes in the plots below. In plot 5.1.1b, we set meanNumDNAs to be a Poisson distribution (mean = 30) and cascadeMeanR to be a lognormal distribution (M = 2.0, S = 0.85). From this plot, we can see that most of the spot intensity statistics distribution is tightly clustered where both the mean and standard deviation are below 1000 as compared to the real data.

![Figure 5.1.1b Simulated array statistics when meanNumDNAs is not optimal](image)

We show another plot where meanNumDNAs is the same Poisson distribution (mean = 30) and cascadeMeanR is also a Poisson distribution (mean = 25). We can see that changing cascadeMeanR has the effect of increasing the spot intensity mean. In addition, the change also increases the slope of the statistics distribution plot slightly.
Finally, after a number of trials, we have the plot (Figure 5.1.1d) below which resembles real data statistics the most. Both $meanNumDNAs$ and $cascadeMeanR$ in this plot are lognormal. For $meanNumDNAs$, $M = 5.56$ and $S = 0.8$. For $cascadeMeanR$, $M = 2.0$, $S = 0.85$. 

Figure 5.1.1c  Simulate array statistics when $cascadeMeanR$ is not optimal
5.1.1.1 Initial trials

We first formulate the hypotheses that the two main factors contributing to the shape of the real data are cascade mean and number of cDNA molecules per pixel. We then test our hypotheses by keeping all other parameters constant and varying only one parameter at a time. During this initial step, we introduce no distribution in our input specification. We vary a parameter only by increasing its constant value. The standard deviation vs. mean plots based on these parameter changes show a tight cluster of dots moving along the increasing X-direction. From the result of this experiment, we conclude that increasing cascade mean or number of cDNA per pixel when both parameters are specified as constants instead of distributions, increases the spot signal intensity but not its variation. Increasing the gain only increase the signal variation slightly, its main effect is scaling the signal mean.
After gaining intuition about how to specify the mode for spot intensity means, the next step is to achieve the signal variation similar to the real data. From numerous trials of varying parameters, we find that the scale of the cascade mean affects the slope of the statistics distribution slightly, while the width of the statistics distribution is controlled by the scale of pixel-to-pixel cDNA distribution for a spot. We model signal intensity variation by probability distributions. As real data seems to posses the standard deviations of a spot intensity often greater than their mean, we must find a distribution whose deviate is always positive and whose standard deviation can be greater than its mean. This distribution cannot be uniform because we observe clustering and spreading of intensity statistics in real data, which means that signal intensities are not uniformly distributed. In light of all these, the most appropriate distribution that models pixel-to-pixel variation is the lognormal distribution, whose probability distribution function is specified as

\[ P(x) = \frac{1}{Sx\sqrt{2\pi}} e^{-\frac{(\ln x - M)^2}{2S^2}} \]
5.1.1.2 Matching real and simulated data

Gaining intuition about the effect of various parameters forms a basis for us to analyze real data in greater detail. Our goal is to find a relationship between the standard deviation of a spot intensity and its mean. Since each dot in Figure 5.1.1a represents data for a spot in the array, we start by analyzing a spot. For simplicity, we assume probBind is one, so that the number of cDNAs in a pixel is equal to the number of DNAs in a pixel. This is just so that we can directly specify the number of cDNAs per pixel. From the initial trials, we know that two factors, the mean number of DNAs per pixel and the cascade mean affect the spot intensity mean. The standard deviation of the spot intensity is affected by the standard deviation of the number of DNAs per spot, scaled by the cascade mean, plus the standard deviation of the cascade mean. So we have
To have a linear relationship, the ratio of spot intensity standard deviation($\sigma_i$) and mean($\mu_i$) must be roughly a constant.

$$\frac{\sigma_i}{\mu_i} = \frac{\sigma_{nDNAs}}{\mu_{nDNAs}} + \frac{\sigma_{cas}}{\mu_{cas} \cdot \mu_{nDNAs}}$$  \[8\]

$$\frac{\sigma_i}{\mu_i} = \gamma + \frac{\sigma_{cas}}{\mu_{cas} \cdot \mu_{nDNAs}}$$  \[9\]

So as long as $\sigma_{cas} / (\mu_{cas} \cdot \mu_{nDNAs})$ is small, $\gamma$, the ratio of $\sigma_{nDNAs} / \mu_{nDNAs}$ dominates the relationship between $\sigma_i$ and $\mu_i$. We find an estimation of $\gamma$ by plotting $\sigma_i$ vs $\mu_i$ for the real data. It appears that we can approximate the plot of $\log(\sigma_i/\mu_i)$ for the real data by partitioning it to be 90% lognormal ($M = -1.56$, $S = 0.5$) and 10% uniform ($a = -1$, $b = 0.5$). So we define the multiplier $\gamma$ for pixel-by-pixel variation to be 90% of the time lognormal and 10% of the time uniform. After $\gamma$ is found for the spot, we can compute $M$ and $S$ of the lognormal distribution for nDNA pixel-to-pixel variation, such that the standard deviation for this distribution is $\gamma$ times its mean.

$$\sigma = \gamma \cdot \mu_{nDNAs}, \quad \mu = \mu_{nDNAs}$$  \[10\]

$$S = \sqrt{(\ln(\gamma^2 + 1))}$$  \[11\]

$$M = 0.5 \cdot \ln\left(\frac{\mu^2}{(\gamma^2 + 1)}\right)$$  \[12\]

An implementation of this pixel-to-pixel variation is in ConstructArrayVisitor.cc (see Appendix B.4). For the number of DNAs in a pixel in this spot, we draw a deviate from this lognormal distribution.

Specifying number of DNAs distribution to be lognormal allows us to widen the spot intensity distribution, generating a plot similar to the real data. We then set both
parameters \textit{cascadeMeanR} and \textit{meanNumDNAs} to be lognormal. Doing this gives us a wide possibility of variations as well as clustering and spreading of spot intensity statistics distribution. To center the mode of spot intensity statistics around 1000 as in the real data, \textit{cascadeMeanR} and \textit{meanNumDNAs} must be changed together so that their product remains approximately the same.

Another aspect that we have considered when matching simulated data to real data is the variation in spot signal ratio (red counts / green counts). We want the simulated array to have a similar variation in ratio comparing to the real array. To achieve this, we plot histograms of the standard deviation of the log ratio. We find that the shape of the histogram mainly depends on the mean of the \textit{meanNumDNAs} distribution. As the mean of \textit{meanNumDNAs} increases, the peak of the histogram becomes narrower. We therefore need to select parameter values that give us the best match for both the array spot intensity statistics plot (Figure 5.1.1d) and spot signal ratio histogram (5.1.1.2b). The complete input parameter specification file that generated the best simulated plots is \textit{simulate.xml} in Appendix C.
5.2 Evaluation of the inference machine

In addition to explore parameter changes, the simulator can be used to evaluate the performance of the Bayesian inference machine. The command to run the inference machine with an intensity file is
The program *hermite* is the inference machine [Duggar et al, 2002]. The result file holds the analysis result given the intensity file. We extract the probability of delta-expression and the inferred delta-expression ratio for each gene from the result. We can check these data to see if the inference machine gives reasonable inferences. For example, since one of the input parameters of the simulator is the fraction of test cDNA molecules in the pool (*fractionR*), we can compare the inferred delta-expression fraction with *fractionR* to see how closely they are related. A sample of inferred result is shown below in List 5.2 when *fractionR* is 0.5. A more advanced evaluation can be done by analyzing the false positives and false negatives of the inference machine.

<table>
<thead>
<tr>
<th>Prob(Δ-exp)</th>
<th>Δ-exp fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0171614</td>
<td>0.497844</td>
</tr>
<tr>
<td>0.0256777</td>
<td>0.496977</td>
</tr>
<tr>
<td>0.0337316</td>
<td>0.505735</td>
</tr>
<tr>
<td>0.0260224</td>
<td>0.496761</td>
</tr>
<tr>
<td>0.0579892</td>
<td>0.489331</td>
</tr>
</tbody>
</table>

List 5.2 Sample result of the inference machine

### 5.2.1 Analysis of false positive

To analyze false positives of the inference machine, we set *fractionR* to be 0.5, and calculate the percentage of positive spots out of all spots. The thresholds that indicate positiveness can be arbitrary. It is the probability of delta-expression that we are more interested in. As shown in List 5.2, the probabilities of delta expression for the spots are mostly below 0.1 and the inferred degrees of delta-expression fractions are very close to 0.5. A plot of the percentage of false positives as a function of the number of pixels per
spot is shown below in Figure 5.2.1a. We plotted the function for conventional thresholds of 50%, 95%, and 99% probability of delta-expression. As shown, the percentage of false positives is exponentially decreasing as the number of pixels increases. This is because the more pixels in a spot, the more information the inference machine utilizes, and the more accurate the result should be. With 113 pixels per spot same as real array data, we can say with 99% confidence that the percentage of false positives is zero. Information such as this may be useful for improving future microarray technology, since we have shown reducing the number of pixels per spot may not significantly increase false positives.

Figure 5.2.1a  False positives as a function of number of pixels per spot

Since the thresholds that define false positives can be arbitrarily set, we also plotted false positives as a function of number of pixels for all possible thresholds (Figure 5.2.1b). As we can see, percentage of false positives decreases as threshold increases, which means higher restrictions for being positive.
5.2.2 Analysis of false negative

In addition to analyze false positives of the inferred result, we also need to analyze false negatives to complete the evaluation of the inference machine. The way we analyze false negatives is by computing the percentage of false negative spots among all spots as a function of delta-expressions ratios. We plot the false negative percentage as a function of log$_2$ of the delta-expression ratio, with the incremental step being 0.01 of the log ratio. We use the simulator to generate about 300 intensity files with $fractionR$ incrementing with the log ratios. The inference machine then accepts these intensity files to analyze. As shown in Figure 5.2.2a, we selected three conventional thresholds to show that the
percentage of false negatives decreases exponentially as log ratio increases. The plot shows that as degree of delta-expression becomes higher, the number of false negatives approaches zero. The main cut-off of \( \log_2(\text{ratio}) \) is 0.5 (ratio = 1.414, \( \text{fraction}_R = 0.586 \)), at this value the inference machine is able to infer with a very low percentage of false negatives.

![Graph showing false negatives for thresholds 50%, 95%, and 99%](image)

**Figure 5.2.2a** False negatives for thresholds 50%, 95%, and 99%

As the thresholds for the above plot is arbitrarily selected, a plot for all thresholds with percentage of false negatives as a function delta-expression ratio is shown below in Figure 5.2.2b.
If we set the false negative percentage for all thresholds to be about 5% and 95%, and plot the delta-expression log ratio, we get a plot as in Figure 5.2.2c. From this plot we can see that as threshold increases, the restriction of negative becomes looser, and the delta-expression ratio to maintain the same false negative percentage increases as a result.
Figure 5.2.2c Delta-expression ratios for which false negatives is 5% and 95% for all thresholds
Chapter 6

Discussion

In this thesis we described the design and implementation of a microarray simulator, as well as demonstrated its use and relation with the Bayesian inference machine. This chapter summarizes the work of the thesis and enumerates ideas for future work to improve and utilize the simulator.

6.1 Summary of work

The simulator incorporates all known physical processes involved in microarray hybridization and imaging. This thesis has described the design and implementation of the simulator in detail, as well as provided a user's guide, which allows interested readers to use and extend simulator functionality. Extensibility is one of our major concerns during simulator design phase. Examples are also provided to demonstrate how to use the simulator to aid scientific investigations, such as evaluating the inference machine and getting insight about the effect of parameter variations. We compared the simulated data and real data, and showed that they are qualitatively similar in terms of spot intensity statistics distribution and spot signal ratio variations. Through various test trials, we gained intuitive sense of the effect of several parameters, such as cascade mean and number of cDNAs per pixel. We found that the lognormal distribution can best model pixel-to-pixel cDNAs variation, as well as the variation of cascade mean and number of cDNAs per pixel. When evaluating the inference machine, we analyzed false positives...
and false negatives of the probability of delta-expressions, and found that both are negligible within the range of the set parameter values. We also analyzed at how the inferred degree of delta-expression relates to the input parameter $\text{fraction}_R$, and found that the inferred result is very close to the input value. The result of evaluation shows that the inference machine is performing well and is sensitive to actual biological and unknown variations but not to experimental variations incorporated by the simulator, such as shot noise and background fluorescence.

6.2 Related Work

The relevant work to the simulator is the design and implementation of the Bayesian inference machine [Duggar et al, 2002]. The inference machine inspired the creation of the simulator. The inference machine allows independent, non-arbitrary simultaneous determination of the probability of delta-expression. It demonstrates the advantage of using a first principles informatic chain to analyze DNA microarrays. Another useful program that helps to present the simulated data visually is the array image generator [Duggar et al, 2002]. The array image generator takes simulated intensity file as input and produces a visual layout of the simulated array. Visual representation is very helpful for providing a qualitative sense of the spot intensities.

6.3 Future Work

Potential additions to increase simulator capabilities can be implemented. Such additions include implementing more probability distributions and functions to be used by the simulator, and specifying more parameters. The extensibility of the simulator makes it
easy to incorporate these additional changes to the program structure. We may also want to extend the informatic chain to include additional experimental steps. This will bring more sources of variation to the physical model.

The simulator can help to explore more effects of parameter changes. For example, we can set the gain parameter to be a distribution, instead of just a constant, and study its effect on the outcome. Since there are many parameters, all of which have some effects on the outcome of simulation, a complete study of these parameters will require considerable time and future effort.

Finally, we can evaluate the performance of the inference machine in more detail by generating more simulated data with varying parameter values and analyze how false positives and false negatives change according to these variations. For example, we can compute the percentage false negatives for different number of pixels per spot given a range of delta-expression ratios, and see whether we can reduce the number of pixels per spot while still maintaining a low false negative rate.
Appendix A

XML parameter input file specification

Specify a parameter variable by indicating with label <param>, followed by attributes such as name and type.

Parameter types: Constant, Functor or Distribution.

For type Functor, attributes: type (functor type: Table or BTree), numArgs (number of arguments), argType (type of argument: Integer or Float). Interpolation is applied to Functor argument values. We have only implemented a standard mapping functor (TableFunc), which uses one argument. A BTree functor, which uses more than one argument, has not been implemented yet, but a possible XML specification for it is provided below.

For type Distribution, parameter has sub-labels <mean>, <scale>, plus others, depending on the distribution type. Attributes: type (type of distribution: Uniform, Binomial, Beta, Poisson, Gaussian, Lognormal,).

<table>
<thead>
<tr>
<th>ID</th>
<th>Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNA Arrays</td>
<td></td>
<td></td>
</tr>
<tr>
<td>numGenes</td>
<td>Constant</td>
<td>Number of genes for the array. May be a constant fixed value for all genes or a function of i.</td>
</tr>
<tr>
<td>numSpots</td>
<td>Constant Functor(i)</td>
<td>Number of spots for each gene. May be a constant fixed value for all genes or a function of i.</td>
</tr>
</tbody>
</table>

Sample Structures:

Constant:
<param name="numSpots" type="Const">
  nnnn
</param>

Functor:
<param name="numSpots" type="Functor">
  <Functor type="Table">
    <entry><key> i </key> <val>nnnn</val></entry>
  </Functor>
</param>

<table>
<thead>
<tr>
<th>ID</th>
<th>Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>numPixels</td>
<td>Constant Functor(i,j)</td>
<td>Number of pixels for the (j^{th}) spot of gene (i). Maybe a constant fixed value for all spots or a function of ((i,j)).</td>
</tr>
</tbody>
</table>
Sample Structures:
Constant Integer:
<param name="numPixels" type="Const">
  nnnn
</param>

Functor:
<param name="numPixels" type="Functor">
  <Functor type="BTree" numArgs="2" argType="Integer">
    <node><key>i</key><key>j</key><val>nnnn</val></node>
    <key>i</key> (this allows interpolation)
    <node><key>j</key><val>nnnn</val></node>
  </Functor>
</param>

<table>
<thead>
<tr>
<th>ID</th>
<th>Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>meanNumDNAs</td>
<td>Constant</td>
<td>Number of DNA molecules bound to the $k^{th}$ pixel of the $(i, j)^{th}$ spot. May be a constant fixed value for all pixels, a function of $(i, j, k)$, or a distribution</td>
</tr>
<tr>
<td>meanNumDNAs</td>
<td>Functor(i, j, k)</td>
<td></td>
</tr>
<tr>
<td>meanNumDNAs</td>
<td>Distribution</td>
<td></td>
</tr>
</tbody>
</table>

Sample Structures:
Constant:
<param name="meanNumDNAs" type="Const">
  ffff
</param>

Functor:
<param name="meanNumDNAs" type="Functor">
  <Functor type="BTree" numArgs="3" argType="Integer">
    <node><key>i</key><key>j</key><key>k</key><val>fff</val></node>
  </Functor>
</param>

Distribution:
<param name="meanNumDNAs" type="Distribution">
  <Distribution type="Poisson">
  </Distribution>
</param>
Sample Structures:
Constant:
<param name="fractionR" type="Const">
  ffff
</param>

Distribution:
<param name="fractionR" type="Distribution">
  <Distribution type="Gaussian">
    <mean>ffff</mean>
    <scale>ffff</scale>
  </Distribution>
</param>

Sample Structures:
Constant:
<param name="probBind" type="Const">
  ffff
</param>

Functor:
<param name="probBind" type="Functor">
  <Functor type="Table">
    <entry>
      <key> i </key>
      <val>ffff</val>
    </entry>
  </Functor>
</param>
### photonics

<table>
<thead>
<tr>
<th>ID</th>
<th>Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cascadeMeanR</td>
<td>Constant</td>
<td>Poisson cascade mean and ratio. May be a constant floating value for all genes, a function of ((i,j)), or a random deviate.</td>
</tr>
<tr>
<td>cascadeMeanRatio</td>
<td>Functor(i,j)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Distribution</td>
<td></td>
</tr>
</tbody>
</table>

Sample Structures:

**Constant:**

```xml
<param name="cascadeMeanR" type="Const">
  ffff
</param>
```

**Functor:**

```xml
<param name="cascadeMeanR" type="Functor">
  <Functor type="BTree" numArgs="2" argType="Integer">
    <node><key>i</key>
      <node><key>j</key><val>ffff</val>
    </node>
  </node>
</param>
```

**Distribution:**

```xml
<param name="cascadeMeanR" type="Distribution">
  <Distribution type="Poisson">
    <mean>nnnn</mean>
  </Distribution>
</param>
```

<table>
<thead>
<tr>
<th>ID</th>
<th>Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>probFilterRtoR</td>
<td>Constant</td>
<td>The probability of filtering a red photon to the red channel and a green to the green channel. The complement gives the probability of filtering red into green and green into red respectively. They are constant floating values.</td>
</tr>
<tr>
<td>probFilterGtoG</td>
<td>Constant</td>
<td></td>
</tr>
</tbody>
</table>

Sample Structures:

**Constant:**

```xml
<param name="probFilterRtoR" type="Const">
  ffff
</param>
```

<table>
<thead>
<tr>
<th>ID</th>
<th>Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>probDetectR</td>
<td>Constant</td>
<td>The probability of a detector detecting a red or green photon upon entering the detector channel. They are constant floating values.</td>
</tr>
<tr>
<td>probDetectG</td>
<td>Constant</td>
<td></td>
</tr>
</tbody>
</table>

60
Sample Structures:
Constant:
<param name="probDetectR" type="Const"> 
   ffff
</param>

<table>
<thead>
<tr>
<th>ID</th>
<th>Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gainR/gainG</td>
<td>Constant</td>
<td>The gain for the red or green channel.</td>
</tr>
</tbody>
</table>

Sample Structures:
Constant:
<param name="gainR" type="Const"> 
   nnnn
</param>
Appendix B

Code

B.1 ArrayParam

------------------------------ ArrayParam.hh ------------------------------

#ifndef ARRAYPARAM_HH
#define ARRAYPARAM_HH
#include "ArrayParamFunctor.hh"

class ArrayParam {
  public:
    ArrayParam();
    ArrayParam(ArrayParam const&);
    static void erasePointers(ArrayParam&);
    ~ArrayParam();

    int getNumGenes() const;
    void setNumGenes(ArrayParamFunctor<int>*){

    int getNumTags(int i) const;
    void setNumTags(ArrayParamFunctor<int>*){

    int getNumSpots(int i) const;
    void setNumSpots(ArrayParamFunctor<int>*){

    int getNumPixels(int i, int j) const;
    void setNumPixels(ArrayParamFunctor<int>*){

    double getMeanNumDNAs(int i, int j) const;
    void setMeanNumDNAs(ArrayParamFunctor<double>*){

    int getNumDNAs(int i, int j, int k) const;
    void setNumDNAs(ArrayParamFunctor<int>*){

    double getFractionR(int i) const;
    void setFractionR(ArrayParamFunctor<double>*){

    double getProbBind(int i) const;
    void setProbBind(ArrayParamFunctor<double>*){

    double getCascadeMeanR(int i, int j) const;
    void setCascadeMeanR(ArrayParamFunctor<double>*){

    double getCascadeMeanRatio(int i, int j) const;
    void setCascadeMeanRatio(ArrayParamFunctor<double>*){

    double getProbFilterRtoR() const;
    void setProbFilterRtoR(ArrayParamFunctor<double>*){

    double getProbFilterGtoG() const;
    void setProbFilterGtoG(ArrayParamFunctor<double>*){


double getProbDetectR() const;
void setProbDetectR(ArrayParamFunctor<double>*);

double getProbDetectG() const;
void setProbDetectG(ArrayParamFunctor<double>*);

int getGainR() const;
void setGainR(ArrayParamFunctor<int>*);

int getGainG() const;
void setGainG(ArrayParamFunctor<int>*);

private:
  ArrayParamFunctor<int>* numGenes;
  ArrayParamFunctor<int>* numTags;
  ArrayParamFunctor<int>* numSpots;
  ArrayParamFunctor<int>* numPixels;
  ArrayParamFunctor<int>* numDNAs;
  ArrayParamFunctor<double>* meanNumDNAs;
  ArrayParamFunctor<double>* fractionR;
  ArrayParamFunctor<double>* probBind;
  ArrayParamFunctor<double>* cascadeMeanR;
  ArrayParamFunctor<double>* cascadeMeanRatio;
  ArrayParamFunctor<double>* probFilterRtoR;
  ArrayParamFunctor<double>* probFilterGtoG;
  ArrayParamFunctor<double>* probDetectR;
  ArrayParamFunctor<double>* probDetectG;
  ArrayParamFunctor<int>* gainR;
  ArrayParamFunctor<int>* gainG;
};
ap.probDetectG = 0;
ap.gainR = 0;
ap.gainG = 0;
}

// Copy constructor
ArrayParam::ArrayParam(ArrayParam const& _ap) {
    ArrayParam& ap = const_cast<ArrayParam&>(_ap);
    numGenes = ap.numGenes;
    numTags = ap.numTags;
    numSpots = ap.numSpots;
    numPixels = ap.numPixels;
    numDNAs = ap.numDNAs;
    meanNumDNAs = ap.meanNumDNAs;
    fractionR = ap.fractionR;
    probBind = ap.probBind;
    cascadeMeanR = apcascadeMeanR;
    cascadeMeanRatio = ap.cascadeMeanRatio;
    probFilterRtoR = ap.probFilterRtoR;
    probFilterGtoG = ap.probFilterGtoG;
    probDetectR = ap.probDetectR;
    probDetectG = ap.probDetectG;
    gainR = ap.gainR;
    gainG = ap.gainG;

    erasePointers(ap);
}

ArrayParam::ArrayParam(){
    delete numGenes;
    delete numTags;
    delete numSpots;
    delete numPixels;
    delete numDNAs;
    delete meanNumDNAs;
    delete fractionR;
    delete probBind;
    delete cascadeMeanR;
    delete cascadeMeanRatio;
    delete probFilterRtoR;
    delete probFilterGtoG;
    delete probDetectR;
    delete probDetectG;
    delete gainR;
    delete gainG;
}

int ArrayParam::getNumGenes() const{
    return (*numGenes)();
}

void ArrayParam::setNumGenes(ArrayParamFunctor<int>* func) {
    delete numGenes;
    numGenes = func;
}

int ArrayParam::getNumTags(int i) const{
std::cerr <<"inside getNumTags"<<endl;
std::cerr <<numTags<<endl;
int n=(*numTags)(i);
std::cerr<<"after n"<<endl;
return (*numTags)(i);
}

void ArrayParam::setNumTags(ArrayParamFunctor<int>* func){
  cerr<<"reached setNumTags:"<<func<<endl;
  delete numTags;
  numTags = func;
}

int ArrayParam::getNumSpots(int i) const{
  return (*numSpots)(i);
}

void ArrayParam::setNumSpots(ArrayParamFunctor<int>* func){
  delete numSpots;
  numSpots = func;
}

int ArrayParam::getNumPixels(int i, int j) const{
  return (*numPixels)(i, j);
}

void ArrayParam::setNumPixels(ArrayParamFunctor<int>* func){
  delete numPixels;
  numPixels = func;
}

double ArrayParam::getMeanNumDNAs(int i, int j) const{
  return (*meanNumDNAs)(i, j);
}

void ArrayParam::setMeanNumDNAs(ArrayParamFunctor<double>* func){
  delete meanNumDNAs;
  meanNumDNAs = func;
}

int ArrayParam::getNumDNAs(int i, int j, int k) const{
  return (*numDNAs)(i, j, k);
}

void ArrayParam::setNumDNAs(ArrayParamFunctor<int>* func){
  delete numDNAs;
  numDNAs = func;
}

double ArrayParam::getFractionR(int i) const{
  return (*fractionR)(i);
}

void ArrayParam::setFractionR(ArrayParamFunctor<double>* func) {
  delete fractionR;
  fractionR = func;
}
double ArrayParam::getProbBind(int i) const{
    return (*probBind)(i);
}

void ArrayParam::setProbBind(ArrayParamFunctor<double>* func){
    delete probBind;
    probBind = func;
}

double ArrayParam::getCascadeMeanR(int i, int j) const{
    return (*cascadeMeanR)(i, j);
}

void ArrayParam::setCascadeMeanR(ArrayParamFunctor<double>* func) {
    delete cascadeMeanR;
    cascadeMeanR = func;
}

double ArrayParam::getCascadeMeanRatio(int i, int j) const{
    return (*cascadeMeanRatio)(i, j);
}

void ArrayParam::setCascadeMeanRatio(ArrayParamFunctor<double>* func) {
    delete cascadeMeanRatio;
    cascadeMeanRatio = func;
}

double ArrayParam::getProbFilterRtoR() const{
    return (*probFilterRtoR());
}

void ArrayParam::setProbFilterRtoR(ArrayParamFunctor<double>* func) {
    delete probFilterRtoR;
    probFilterRtoR = func;
}

double ArrayParam::getProbFilterGtoG() const{
    return (*probFilterGtoG());
}

void ArrayParam::setProbFilterGtoG(ArrayParamFunctor<double>* func) {
    delete probFilterGtoG;
    probFilterGtoG = func;
}

double ArrayParam::getProbDetectR() const{
    return (*probDetectR());
}

void ArrayParam::setProbDetectR(ArrayParamFunctor<double>* func) {
    delete probDetectR;
    probDetectR = func;
}

double ArrayParam::getProbDetectG() const{
    return (*probDetectG());
}
void ArrayParam::setProbDetectG(ArrayParamFunctor<double>* func) {
    delete probDetectG;
    probDetectG = func;
}

int ArrayParam::getGainR() const {
    return (*gainR());
}

void ArrayParam::setGainR(ArrayParamFunctor<int>* func) {
    delete gainR;
    gainR = func;
}

int ArrayParam::getGainG() const {
    return (*gainG());
}

void ArrayParam::setGainG(ArrayParamFunctor<int>* func) {
    delete gainG;
    gainG = func;
}
B.2 ArrayParamFileParser

--- ArrayParamFileParser.hh ---

```cpp
#ifndef ARRAY_PARAM_FILE_PARSER_HH
#define ARRAY_PARAM_FILE_PARSER_HH

#include <util/PlatformUtils.hpp>
#include <util/XMLString.hpp>
#include <util/XMLUniDefs.hpp>
#include <framework/XMLFormatter.hpp>
#include <util/TranscodingException.hpp>
#include <dom/DOM_DOMException.hpp>
#include <parsers/DOMParser.hpp>
#include <dom/DOM.hpp>
#include "DOMTreeErrorReporter.hpp"
#include <string.h>
#include <stdlib.h>
#include <string>
#include "ArrayParam.hh"

class ArrayParamFileParser {
public:
  void parse(char* xmlFile);
  ArrayParam getArrayParam() const;
  //Declare a new type for an enumerated string
  enum ParamName {
    NUM_GENES,
    NUM_TAGS,
    NUM_SPOTS,
    NUM_PIXELS,
    MEAN_NUM_DNAS,
    FRACTION_R,
    PROB_BIND,
    CASCADE_MEAN_R,
    CASCADE_MEAN_RATIO,
    PROB_FILTER_R_TO_R,
    PROB_FILTER_G_TO_G,
    PROB_DETECT_R,
    PROB_DETECT_G,
    GAIN_R,
    GAIN_G,
    XML_SPEC_ERROR
  };
  
  enum ParamType {
    CONST,
    FUNCTOR,
    DISTRIBUTION,
    SPEC_ERROR
  };

  class Exception {};
  class ParamTypeNotFound : public Exception {};
  class UnknownParameter: public Exception {};
};
```

---
ParamName stringToParamName(std::string);
ParamType stringToParamType(std::string);

private:
    ArrayParam arrayparam;
    void parseNodeTree(DOMNode const&);
    void parseDocument(DOMNode const&);
    void parseElement(DOMNode const&);
    template<class T> ArrayParamFunctor<T>* parseParam(DOMNode const&);
};
#endif

--------------------------- ArrayParamFileParser.cc---------------------
#include "ArrayParamFileParser.hh"
#include "ArrayParamFunctor.hh"
#include "ArrayParamFunctorConst.hh"
#include <string>
#include "util.hh"
#include <iostream>
#include "ArrayParam.hh"

using std::string;

ArrayParam ArrayParamFileParser::getArrayParam() const{
    return arrayparam;
}

ArrayParamFileParser::ParamName
ArrayParamFileParser::stringToParamName(std::string string) {
    if (string == "numTags") return NUM_TAGS;
    if (string == "numGenes") return NUM_GENES;
    if (string == "numSpots") return NUM_SPOTS;
    if (string == "numPixels") return NUM_PIXELS;
    if (string == "meanNumDNAs") return MEAN_NUM_DNAS;
    if (string == "fractionR") return FRACTION_R;
    if (string == "probBind") return PROB_BIND;
    if (string == "cascadeMeanR") return CASCADE_MEAN_R;
    if (string == "cascadeMeanRatio") return CASCADE_MEAN_RATIO;
    if (string == "probFilterRtoR") return PROB_FILTER_R_TO_R;
    if (string == "probFilterGtoG") return PROB_FILTER_G_TO_G;
    if (string == "probDetectR") return PROB_DETECT_R;
    if (string == "probDetectG") return PROB_DETECT_G;
    if (string == "gainR") return GAIN_R;
    if (string == "gainG") return GAIN_G;
    return XML_SPEC_ERROR;
}

ArrayParamFileParser::ParamType
ArrayParamFileParser::stringToParamType(std::string string) {
    if (string == "Const") return CONST;
    if (string == "Functor") return FUNCTOR;
    if (string == "Distribution") return DISTRIBUTION;
}
return SPEC_ERROR;

void ArrayParamFileParser::parse(char* xmlFile) {  
  try {  
    XMLPlatformUtils::Initialize();  
  }  
  catch(const XMLException& toCatch) {  
    cerr << "Error during Xerces-c Initialization.\n"  
    << " Exception message:"  
    << DOMString(toCatch.getMessage()) << endl;  
  }  
  // Create DOM parser, then attach an error handler to the parser.  
  // The parser will call back to methods of the ErrorHandler if it  
  // discovers errors during the course of parsing the XML document.  
  DOMParser parser;  
  DOMTreeErrorReporter errReporter;  
  parser.setErrorHandler(&errReporter);  
  // Parse the XML file, catching any XML exceptions that might  
  // propogate out of it.  
  bool errorsOccured = false;  
  try {  
    parser.parse(xmlFile); //parsing  
    int errorCount = parser.getErrorCount();  
    if (errorCount > 0)  
      errorsOccured = true;  
  }  
  catch (...) {  
    cerr << "An error occured during parsing\n" << endl;  
    errorsOccured = true;  
  }  
  // If the parse was successful, construct arrayparam  
  if (!errorsOccured && !errReporter.getSawErrors()) {  
    DOM_Node doc = parser.getDocument();  
    try {  
      parseDocument(doc);  
    }  
    catch (XMLException& e) {  
      cerr << "An error occurred during creation of output  
      transcoder. Msg is:"  
      << endl  
      << DOMString(e.getMessage()) << endl;  
    }  
  }  
  else
cerr<<"An error occurred during parsing. Output trancoder not called.\n"<<endl;
}

//-------------------------------------------------------------------------------

void ArrayParamFileParser::parseDocument(DOMNode const& theNode)
{
  switch(theNode.getNodeType())
  {
    case DOMNode::DOCUMENT_NODE :
    {
      DOMNode child = theNode.getFirstChild();
      while( child != 0)
      {
        parseDocument(child);
        child = child.getNextSibling();
      }
      break;
    }
    case DOMNode::ELEMENT_NODE :
    {
      parseElement(theNode);
      DOMNode child = theNode.getFirstChild();
      while( child != 0)
      {
        parseDocument(child);
        child = child.getNextSibling();
      }
      break;
    }
  }
}

void ArrayParamFileParser::parseElement(DOMNode const& theNode)
{
  DOMString param("param");

  if (!theNode.getNodeName().equals(param))
    return;

  ParamName name = stringToParamName(DOMUtil::getAttribute(theNode, "name"));
  //construct Array Params
  switch (name)
  {
  case NUM_GENES : arrayparam.setNumGenes (parseParam<int>(theNode)); break;
  case NUM_SPOTS : arrayparam.setNumSpots (parseParam<int>(theNode)); break;
  case NUM_PIXELS : arrayparam.setNumPixels (parseParam<int>(theNode)); break;
  case MSAN_NUM_DNAS : arrayparam.setMeanNumDNAs (parseParam<double>(theNode)); break;
  case FRACTION_R : arrayparam.setFractionR (parseParam<double>(theNode)); break;
  case PROB_BIND : arrayparam.setProbBind (parseParam<double>(theNode)); break;
  }
case CASCADE_MEAN_R  : arrayParam.setCascadeMeanR  
      (parseParam<double>(theNode)); break;    
  case CASCADE_MEAN_RATIO  : arrayParam.setCascadeMeanRatio(
parseParam<double>(theNode)); break;    
  case PROB_FILTER_R_TO_R  : arrayParam.setProbFilterRtoR  
      (parseParam<double>(theNode)); break;    
  case PROB_FILTER_G_TO_G  : arrayParam.setProbFilterGtoG  
      (parseParam<double>(theNode)); break;    
  case probDetectR  : arrayParam.setProbDetectR  
      (parseParam<double>(theNode)); break;    
  case probDetectG  : arrayParam.setProbDetectG  
      (parseParam<double>(theNode)); break;    
  case gainR  : arrayParam.setGainR  
      (parseParam<int>(theNode)); break;    
  case gainG  : arrayParam.setGainG  
      (parseParam<int>(theNode)); break;    
    default:    
      throw UnknownParameter();    
    }
}  

template<class T>  
ArrayParamFunctor<T>* ArrayParamFileParser::parseParam(DOMNode const& node)  
{    
  ParamType type = stringToParamType(DOMUtil::getAttribute(node,  
                                    "type"));    
    
    switch(type) {    
      case CONST:    
        return new ArrayParamFunctorConst<T>(node.getFirstChild());    
      case DISTRIBUTION:    
        return ArrayParamFunctor<T>::createDistr(DOMUtil::findNode(node,  
                                      "Distribution"));    
      case FUNCTOR:    
        return ArrayParamFunctor<T>::createFunc(DOMUtil::findNode(node,  
                                      "Functor"));    
      default:    
        std::cerr<<"Throwing ParamTypeNotFound Exception\n";    
        throw ParamTypeNotFound();    
    }  
}
B.3 ArrayParamFunctor

---ArrayParamFunctor.hh---

```cpp
#ifndef ARRAYPARAMFUNCTOR_HH
#define ARRAYPARAMFUNCTOR_HH
#include <string>
#include "util.hh"
#include <iostream>
#include "dom/DOM.hpp"

template<class ResT> class ArrayParamFunctor {
  public:
    typedef ResT Result;
    virtual ResT operator()(int = 0, int = 0, int = 0) const {};
    static ArrayParamFunctor<ResT>* createDistr(DOMNode const&);
    static ArrayParamFunctor<ResT>* createFunc(DOMNode const&);
  //virtual ~ArrayParamFunctor();
  enum EnumString {
    BINOMIAL,
    POISSON,
    GAUSSIAN,
    LOGNORMAL,
    UNIFORM,
    BETA,
    TABLE,
    BTREE,
    INTERGER,
    FLOAT
  };
  static EnumString stringToEnum(std::string const&);
  class Exception {};
  class DistrTypeError: public Exception{};
  class DistrNodeNotFoundError: public Exception{};
  class FuncTypeError: public Exception{};
  class FuncNodeNotFoundError: public Exception{};
  class FuncArgTypeError: public Exception{};
};

template<class ResT> ArrayParamFunctor<ResT>::EnumString
ArrayParamFunctor<ResT>::stringToEnum(std::string const& string) {
  if(string == "Binomial") return BINOMIAL;
  if(string == "Gaussian") return GAUSSIAN;
  if(string == "LogNormal") return LOGNORMAL;
  if(string == "Poisson") return POISSON;
  if(string == "Beta") return BETA;
  if(string == "Table") return TABLE;
  if(string == "BTree") return BTREE;
}
#endif
```
#include "ArrayParamFunctor.hh"
#include "BetaDistr.hh"
#include "BinomDistr.hh"
#include "PoissDistr.hh"
#include "GaussDistr.hh"
#include "LogNormalDistr.hh"
#include "UniformDistr.hh"
#include "TableFunc.hh"
#include <iostream>
#include <string>
#include "dom/DOM.hpp"

using std::string;

ArrayParamFunctor<int>*
ArrayParamFunctor<int>::createFunc(DOMNode const& funcNode) {
    string attrName;
    string funcType;
    string funcNumArgs;
    string funcArgType;

    //retrieve Functor attribute values
    if (toString(funcNode.getNodeName()) == "Functor"){
        DOMNamedNodeMap attributes = funcNode.getAttributes();
        int attrCount = attributes.getLength();
        for (int i = 0; i < attrCount; i++)
        {
            DOMNode attribute = attributes.item(i);
            attrName = toString(attribute.getNodeName());
            if (attrName == "type"){
                funcType = toString(attribute.getNodeValue());
            } else if (attrName == "numArgs"){
                funcNumArgs = toString(attribute.getNodeValue());
            }
        }

        switch(stringToEnum(funcType)){
            case TABLE:
                return new TableFunc<int>(funcNode);
            default:
                throw FuncTypeNotFound();
        }
    }
    throw FuncNodeNotFound();
}

ArrayParamFunctor<double>*
ArrayParamFunctor<double>::createFunc(DOMNode const& funcNode) {

    string attrName;
    string funcType;
    string funcNumArgs;
    string funcArgType;

    //retrieve Functor attribute values
    if (toString(funcNode.getNodeName()) == "Functor"){
        DOMNamedNodeMap attributes = funcNode.getAttributes();
        int attrCount = attributes.getLength();
        for (int i = 0; i < attrCount; i++)
        {
            DOMNode attribute = attributes.item(i);
            attrName = toString(attribute.getNodeName());
            if (attrName == "type"){
                funcType = toString(attribute.getNodeValue());
            } else if (attrName == "numArgs"){
                funcNumArgs = toString(attribute.getNodeValue());
            }
        }

        switch(stringToEnum(funcType)){
            case TABLE:
                return new TableFunc<double>(funcNode);
            default:
                throw FuncTypeNotFound();
        }
    }
    throw FuncNodeNotFound();
}
// retrieve Functor attribute values
if (toString(funcNode.getNodeName()) == "Functor") {
    DOMNamedNodeMap attributes = funcNode.getAttributes();
    int attrCount = attributes.getLength();
    for (int i = 0; i < attrCount; i++)
    {
        DOMNode attribute = attributes.item(i);
        attrName = toString(attribute.getNodeName());
        if (attrName == "type"){
            funcType = toString(attribute.getNodeValue());
        } else if (attrName == "numArgs"){
            funcNumArgs = toString(attribute.getNodeValue());
        }
    }
    switch(stringToEnum(funcType)){
    case TABLE:
        return new TableFunc<double>(funcNode);
    default:
        throw FuncTypeNotFound();
    }
}
throw FuncNodeNotFound();

ArrayParamFunctor<int>*
ArrayParamFunctor<int>::createDistr(DOMNode const& distrNode) {
    string attrName;
    string distrType;
    // retrieve param attribute values
    if (toString(distrNode.getNodeName()) == "Distribution"){
        DOMNamedNodeMap attributes = distrNode.getAttributes();
        int attrCount = attributes.getLength();
        for (int i = 0; i < attrCount; i++)
        {
            DOMNode attribute = attributes.item(i);
            attrName = toString(attribute.getNodeName());
            if (attrName == "type"){
                distrType = toString(attribute.getNodeValue());
            }
        }
        switch(stringToEnum(distrType)){
    case BINOMIAL:
        return new BinomDistr(distrNode);
    case BETA:
        return new BetaDistr<int>(distrNode);
    case POISSON:
        return new PoissDistr<int>(distrNode);
    }
case UNIFORM: {
    return new UniformDistr<int>(distrNode);
}
case GAUSSIAN: {
    return new GaussDistr<int>(distrNode);
}
case LOGNORMAL: {
    return new LogNormalDistr<int>(distrNode);
}
default: 
    throw DistrTypeNotFound();
}
throw DistrNodeNotFound();

ArrayParamFunctor<double>*
ArrayParamFunctor<double>::createDistr(DOMNode const& distrNode) {
    DOMString Distribution("Distribution");
    string attrName;
    string distrType;

    // retrieve param attribute values
    if (distrNode.getNodeName().equals(Distribution)) {
        DOMNamedNodeMap attributes = distrNode.getAttributes();
        int attrCount = attributes.getLength();
        for (int i = 0; i < attrCount; i++) {
            DOMNode attribute = attributes.item(i);
            attrName = toString(attribute.getNodeName());
            if (attrName == "type") {
                distrType = toString(attribute.getNodeValue());
            }
        }
    }
    switch (stringToEnum(distrType)) {
    case POISSON: {
        return new PoissDistr<double>(distrNode);
    }
    case GAUSSIAN: {
        return new GaussDistr<double>(distrNode);
    }
    case LOGNORMAL: {
        return new LogNormalDistr<double>(distrNode);
    }
    case UNIFORM: {
        return new UniformDistr<double>(distrNode);
    }
    case BETA: {
        return new BetaDistr<double>(distrNode);
    }
    default: 
        throw DistrTypeNotFound();
    }
}
throw DistrNodeNotFound();
}
B.4 ArrayParamFunctorConst

-----------------------------ArrayParamFunctorConst-----------------------------

#ifndef ARRAYPARAMFUNCTORCONSTHH
#define ARRAYPARAMFUNCTORCONSTHH
#include "ArrayParamFunctor.hh"
#include <sstream>
#include "util.hh"

template<class ResT> class ArrayParamFunctorConst : public ArrayParamFunctor<ResT>
{
public:
  typedef ResT Result;
  ArrayParamFunctorConst(ResT _result) : result(_result) {}
  ArrayParamFunctorConst(DOMNode const& textNode) {
    std::stringstream stream(toString(textNode.getNodeValue()));
    stream >> result;
  }
  Result operator()(int = 0, int = 0, int = 0) const {return result;}
  void setResult(ResT _result) {result = _result;}
private:
  ResT result;
};

template<class ResT>
ArrayParamFunctorConst<ResT>::ArrayParamFunctorConst(DOMNode const& textNode) {
  std::stringstream stream(toString(textNode.getNodeValue()));
  stream >> result;
}
#endif


B.5 Distributions

B.5.1 Uniform Distribution

--- UniformDistr.hh ---

```cpp
#ifndef UNIFORMDISTRHH
#define UNIFORMDISTRHH
#include "ArrayParamFunctor.hh"
#include "dom/DOM.hpp"
#include <cmath>
#include "Random.hh"
#include <sstream>
#include <string>
#include "util.hh"
#include <iostream>

// This class generates uniform deviates in the range of (a, b).
// \[ p(y) = \frac{1}{b-a} \]

template<class ResT> class UniformDistr: public ArrayParamFunctor<ResT> {
    public:
        UniformDistr(double a = 0, double b = 1);
        UniformDistr(DOMNode const&);
        double getA() const;
        double getB() const;
        void setA(double);
        void setB(double);
        ResT operator()(int=0, int=0, int=0) const;
    private:
        double a;
        double b;
    }

template<class ResT>
UniformDistr<ResT>::UniformDistr(double _a, double _b) :
    a(_a), b(_b){} // initialization

template<class ResT>
UniformDistr<ResT>::UniformDistr(DOMNode const& distrNode):
    a(0), b(1) { // read distribution parameters
    try{
        DOM_Node aNode = DOMUtil::findNode(distrNode, "a");
        DOM_Node avalueNode=aNode.getFirstChild();
        std::string avalue = toString(avalueNode.getNodeValue());
        double ta;
        std::stringstream(avalue)>> ta;
        setA(ta);
    }
    catch(DOMUtil::NodeNotFound) {
        std::cerr<<"Uniform: no 'a' specified by user. Use default b: "
        <<getA()<<endl;
    }
    try{
```

78
DOMNode bNode = DOMUtil::findNode(distrNode, "b");
DOMNode bvalueNode = bNode.getFirstChild();
std::string bvalue = toString(bvalueNode.getNodeValue());
double tb;
std::stringstream(bvalue) >> tb;
setB(tb);
}
catch(DOMUtil::NodeNotFound) {
    std::cerr << "Uniform: no 'b' specified by user. Use default b: " << getB() << endl;
}

}
B.5.2 Binomial Distribution

//This class generates binomial deviates. The binomial distribution is
//given by:
//  \[ P[m] = \binom{n}{m} \cdot p^m \cdot (1-p)^{n-m} \]
//where \( n \) is the number of trials, \( m \) is the number of successes, and \( p \)
//is the probability of success. It uses a variant of the method
// Foremost, we are talking about the following constraints apply:
// (1) \( 0 \leq p \leq 1 \)
// (2) \( 0 \leq n \)

class BinomDistr : public ArrayParamFunctor<int> {
    public:
        BinomDistr(); //Default trials = 1 and prob = 0.5
        BinomDistr(int); //Sets trials and defaults prob = 0.5
        BinomDistr(double); //Sets prob and defaults trials to 1
        BinomDistr(int, double); //Sets trials and prob
        BinomDistr(DOMNode const&); //Default trials = 1 and prob = 0.5

        double getProb () const; //Returns the probability of success
        int getTrials () const; //Returns the number of trials

        void setProb (double); //Set the probability of success
        void setTrials (int); //Set the number of trials

        int operator()(int = 0, int = 0, int = 0) const; //replaces next()

        //These error types are thrown when parameters specified by the client
        //violate any of constraints (1)-(2) listed above.
        class InvalidParam {};
        class InvalidProb : public InvalidParam {};
        class InvalidTrials : public InvalidParam {};

        //The following private utility function operates on the private
        //variables. It is responsible for setting the variables to a
        //consistent state. This function is called after a client changes a
        //parameter via the set functions or after the constructors initialize
        //a subset of the private parameters.

        private:
            void update();

        //The following parameters are not independent. Defining any one of
        //the prob* variables defines the rest. Defining either of the trials*
        //variables determines the other. Thus, care must be taken to insure
        //that the set functions and constructors establish a valid and
        //consistent set of values. Why bother with such a mess? Efficiency
//is paramount in this implementation. Thus, we sacrifice clarity, 
//robustness, and space to insure the highest temporal efficiency.

private:
    int trials;  //The number of trials
    double prob;  //The probability of success
    double probln;
    double probnot;
    double probnotln;
    double mean;
    double scale;
    double trialsfactln;
    bool polarized;
    bool reflected;
};
#endif

-------------------------- BinomDistr.cc--------------------------------
#include "BinomDistr.hh"
#include <cmath>
#include <Random.hh>
#include "constants.hh"
#include "functions.hh"
#include <sstream>
#include "util.hh"
#include <string>
#include <iostream>

using std::string;

BinomDistr::BinomDistr() :
    prob(0.5), trials(1), reflected(false) {update();}
BinomDistr::BinomDistr(int _trials) :
    prob(0.5), trials(_trials), reflected(false) {update();}
BinomDistr::BinomDistr(double _prob) :
    prob(_prob), trials(1), reflected(false) {update();}
BinomDistr::BinomDistr(int _trials, double _prob) :
    prob(_prob), trials(_trials), reflected(false) {update();}
BinomDistr::BinomDistr(DOMNode const& distrNode) :
    prob(0.5), trials(1), reflected(false) {
        update();

    //read distribution parameters
    try{
        DOMNode trialsNode = DOMUtil::findOneNodeIfPresent(distrNode, "trials");
        DOMNode tvalueNode = trialsNode.getFirstChild();
        string tvalue = toString(tvalueNode.getNodeValue());
        int ttrials;
        std::stringstream(tvalue)>> ttrials;
        setTrials(ttrials);
    }
    catch(DOMUtil::NodeNotFound) {
        std::cerr<<"Binomial: no 'trials' specified by user. Use default trials"
                 <<getTrials()<<endl;
try{
    DOM_Node probNode = DOMUtil::findNode(distrNode, "prob");
    DOM_Node pvalueNode=probNode.getFirstChild();
    string pvalue = toString(pvalueNode.getNodeValue());
    double tprob;
    std::stringstream(pvalue)>> tprob;
    setProb(tprob);
} catch(DOMUtil::NodeNotFound) {
    std::cerr<<"Binomial: no 'prob' specified by user. Use default prob"
    <<getProb()<<endl;
}

double BinomDistr::getProb () const { return reflected ? probnot : prob; }
int BinomDistr::getTrials() const { return trials; }
void BinomDistr::setProb (double _prob) { prob = _prob ; reflected = false; update();}
void BinomDistr::setTrials(int _trials) { trials = _trials; update();}
void BinomDistr::update() {
    if(prob < 0.0) throw InvalidProb();
    if(trials < 0) throw InvalidTrials();
    if(prob > 0.5) {
        prob = 1.0 - prob;
        reflected = true;
    }
    if(prob == 0.0) {
        polarized = false;
    } else {
        polarized = false;
        probln = std::log(prob);
        probnot = 1.0 - prob;
        probnotln = std::log(probnot);
        mean = trials*prob;
        scale = std::sqrt(2*mean*probnot);
        trialsfactln = KHD::gammaln(trials + 1);
    }
}

int BinomDistr::operator()(int,int,int) const {
    static Random& generator = *Random::instance();
    if(polarized) return reflected ? trials : 0;
    int result (0);
    if(trials < 25) {
        for(int i = 0; i < trials; ++i)
        if(generator.nextFloat() < prob) ++result;
} else if(mean < 1.0) {
    double goal = std::exp(-mean);
    double curr = 1.0;
    int i;
    for(i = 0; i <= trials; ++i) {
        curr *= generator.nextFloat();
        if(curr < goal) break;
    }
    result = i <= trials ? i : trials;
} else {
    double temp;
    double ratio;
    double angle;
    double tangle;
    do {
        do {
            angle = KHD::Math::PI * generator.nextFloat();
            tangle = std::tan(angle);
            temp = scale*tangle + mean;
        } while(temp < 0.0 || temp >= (trials + 1.0));
        temp = std::floor(temp);
        ratio = trialsfactln;
        ratio -= KHD::gammaln(temp+1.0) + KHD::gammaln(trials - temp + 1.0);
        ratio += temp*probln + (trials - temp)*probnotln;
        ratio = std::exp(ratio);
        ratio *= 1.2*scale*(1.0+tangle*tangle);
    } while(generator.nextFloat() > ratio);
    result = static_cast<int>(temp);
    return reflected ? trials - result : result;
}
B.5.3 Beta Distribution

--- BetaDistr.hh ---

```cpp
#ifndef BETADISTR_HH
#define BETADISTR_HH
#include "ArrayParamFunctor.hh"
#include "dom/DOM.hpp"
#include "UniformDistr.hh"
#include <cmath>
#include <sstream>
#include <string>
#include "dom/DOM.hpp"
#include "util.hh"
#include <iostream>
#include "constants.hh"
#include "functions.hh"

// This class generates beta deviates. The beta distribution is given by:
// B(x) = \frac{\gamma(a + b)}{\gamma(a) \times \gamma(b)} * x^{a-1} * (1-x)^{b-1}

template<class ResT> class BetaDistr: public ArrayParamFunctor<ResT> {
    public:
        BetaDistr(double alpha = 1, double beta = 1, double a=0, double b=1);
        BetaDistr(DOMNode const&);
        double getA () const;
        double getB () const;
        double getAlpha () const;
        double getBeta () const;
        void setAlpha (double);
        void setBeta (double);
        void setA (double);
        void setB (double);
        ResT operator()(int = 0, int =0, int=0) const;
    private:
        double alpha;
        double beta;
        double a;
        double b;
        double Beta(double) const;
};

template<class ResT>
BetaDistr<ResT>::BetaDistr(double _alpha, double _beta, double _a, double _b) :
    alpha(_alpha), beta(_beta), a(_a), b(_b){} // initialization

template<class ResT>
BetaDistr<ResT>::BetaDistr(DOMNode const& distrNode):
    alpha(1), beta(1), a(0), b(1) {
    // read distribution parameters
    try{
        DOMNode alphaNode = DOMUtil::findNode(distrNode, "alpha");
        DOMNode alphavalueNode=alphaNode.getFirstChild();
        std::string alphavalue = toString(alphavalueNode.getNodeValue());
```
double talpha;
std::stringstream(alphavalue) >> talpha;
setAlpha(talpha);
}
catch(DOMUtil::NodeNotFound) {
  std::cerr << "Beta: no 'alpha' specified by user. Use default alpha: "
             << getAlpha() << endl;
}

try{
  DOM_Node betaNode = DOMUtil::findNode(distrNode, "beta");
  DOM_Node betavalueNode = betaNode.getFirstChild();
  std::string betavalue = toString(betavalueNode.getNodeValue());
  double tbeta;
  std::stringstream(betavalue) >> tbeta;
  setBeta(tbeta);
} catch(DOMUtil::NodeNotFound) {
  std::cerr << "Beta: no 'beta' specified by user. Use default beta: "
            << getBeta() << endl;
}

try{
  DOM_Node aNode = DOMUtil::findNode(distrNode, "a");
  DOM_Node avalueNode = aNode.getFirstChild();
  std::string avalue = toString(avalueNode.getNodeValue());
  double ta;
  std::stringstream(avalue) >> ta;
  setA(ta);
} catch(DOMUtil::NodeNotFound) {
  std::cerr << "Beta: no 'a' specified by user. Use default a: "
            << getA() << endl;
}

try{
  DOM_Node bNode = DOMUtil::findNode(distrNode, "b");
  DOM_Node bvalueNode = bNode.getFirstChild();
  std::string bvalue = toString(bvalueNode.getNodeValue());
  double tb;
  std::stringstream(bvalue) >> tb;
  setB(tb);
} catch(DOMUtil::NodeNotFound) {
  std::cerr << "Beta: no 'b' specified by user. Use default b: "
            << getB() << endl;
}
}

template<class ResT>
double BetaDistr<ResT>::getAlpha() const { return alpha; }

template<class ResT>
double BetaDistr<ResT>::getA() const { return a; }

template<class ResT>
double BetaDistr<ResT>::getBeta () const { return beta; }

template<class ResT>
double BetaDistr<ResT>::getB () const { return b; }

template<class ResT>
void BetaDistr<ResT>::setAlpha (double _alpha) {
    alpha = _alpha;
}

template<class ResT>
void BetaDistr<ResT>::setA (double _a) {
    a = _a;
}

template<class ResT>
void BetaDistr<ResT>::setBeta (double _beta) {
    beta = _beta;
}

template<class ResT>
void BetaDistr<ResT>::setB (double _b) {
    b = _b;
}

template<class ResT>
ResT BetaDistr<ResT>::operator()(int,int, int) const {
    double mode = (alpha - 1) / (alpha + beta - 2);
    double Bmax = Beta(mode);
    UniformDistr<double> UDistr;
    double U1, U2;
    double B1;

    do {
        U1 = UDistr();
        U2 = UDistr();
        B1 = Beta(U1);
    } while ((U2 * Bmax) >= B1);
    return static_cast<ResT>(U1*(b-a) + a);
}

template<class ResT>
double BetaDistr<ResT>::Beta(double x) const{
    double lnB;
    lnB = KHD::gammaln(alpha + beta) - KHD::gammaln(alpha)
        - KHD::gammaln(beta)
        + (alpha - 1) * log(x) + (beta-1) * log(1 - x);
    return exp(lnB);
}

#endif
B.5.4 Poisson Distribution

This class generates poisson deviates. The binomial distribution is given by:

\[ P(m) = \lambda^m \cdot \exp(-\lambda) / m! \]

where \( \lambda \) is mean, \( m \) is the number of successes.

It uses a variant of the method described in "Numerical Recipes in C" Second Edition pages 293-295. Foremost, we are talking about the following constraints apply:

1. \( 0 \leq \lambda \)

Factorial is taken to be the gamma function when \( m \) is not an integer.

```cpp
template<class ResT>
class PoissDistr : public ArrayParamFunctor<ResT> {
    public:
        PoissDistr(double mean = 5);  // Sets default mean to 5
        PoissDistr(DOMNode const&);
        double getMean() const;  // Returns the mean
        void setMean(double);  // Set the mean

        ResT operator()(int=0, int=0, int=0) const;

        class InvalidParam {};
        class InvalidMean : public InvalidParam {};

    private:
        double mean;
        double sq, alxm, goal, g;
};

template<class ResT>
PoissDistr<ResT>::PoissDistr(double _mean):
    mean(_mean)
{
    goal = exp(-_mean);
    sq = sqrt(2.0 * _mean);
    alxm = log(_mean);
    g = _mean * alxm - KHD::gammaln(_mean + 1.0);
}

template<class ResT>
PoissDistr<ResT>::PoissDistr(DOMNode const& distrNode):
```
mean(5.0) {
    goal = exp (-mean);
    sq = sqrt(2.0 * mean);
    alxm = log(mean);
    g = mean*alxm - KHD::gammaln(mean + 1.0);
}

try{
    DOM_Node meanNode = DOMUtil::findNode(distrNode, "mean");
    DOM_Node valueNode=meanNode.getFirstChild();
    std::string value = toString(valueNode.getNodeValue());
    double tmean = mean;
    std::stringstream(value)>> tmean;
    setMean(tmean);
}
catch(DOMUtil::NodeNotFound){
    std::cerr<< "Poisson: no 'mean' specified by user. Use default mean: "
        <<getMean()<<endl;
}
}

template<class ResT>
double PoissDistr<ResT>::getMean () const { return mean; }

template<class ResT>
void PoissDistr<ResT>::setMean (double _mean) {
    mean = _mean ;
    goal = exp (-mean);
    sq = sqrt(2.0 * mean);
    alxm = log(mean);
    g = mean*alxm - KHD::gammaln(mean + 1.0);
}

inline int PoissDistr<int>::operator()(int, int, int) const {
    Random& generator = *Random::instance();
    if(mean == 0) return 0;
    double em, y, t;
    if(mean < 12.0) {
        t = 1.0;
        em = -1;
        do {
            ++em;
            t *= generator.nextFloat();
        } while (t > goal);
        do {
            do{
                y = tan(KHD::Math::PI * generator.nextFloat());
                em = sq * y + mean;
            } while (em < 0.0);
            em = floor (em);
            t = 0.9*(1.0 + y*y)*exp(em*alxm - KHD::gammaln(em + 1.0) - g);
        } while (generator.nextFloat() > t);
}
} return static_cast<int> (em); }

inline double PoissDistr<double>::operator()(int, int, int) const {
    Random& generator = *Random::instance();
    if(mean == 0) return 0;
    double em, y, t;
    do { do {
        y = tan(KHD::Math::PI * generator.nextFloat());
        em = sq * y + mean;
    } while (em < 0.0);
    em = floor (em);
    t = 0.9*(1.0 + y*y) * exp(em*alxm - KHD::gammaln(em + 1.0) - g);
    } while (generator.nextFloat() > t);
    return em;
}
#endif
B.5.5 Gaussian Distribution

//This class generates gaussian deviates. The Gaussian distribution is given by:
// p(y) = 1/ (SQRT(2 * PI)* stddev)* exp(- (y - mean)^2 / (2* stddev^2))

template<class ResT> class GaussDistr: public ArrayParamFunctor<ResT> { 
public:
    GaussDistr(double mean = 0, double scale = 10);
    GaussDistr(DOMNode const&);
    double getMean () const;
    double getScale () const;
    void setMean (double);
    void setScale (double);
    ResT operator() (int = 0, int=0, int=0) const;

private:
    double mean;
    double scale;
    mutable double iset; //flag to see if extra deviate is handy
    mutable double gset; //place to keep extra deviate
    ResT convert(double) const;
};

template<class ResT>
GaussDistr<ResT>::GaussDistr(double _mean, double _scale) :
    mean(_mean), scale(_scale) {iset = 0;} //initialization

template<class ResT>
GaussDistr<ResT>::GaussDistr(DOMNode const& distrNode):
    mean(0), scale(10) {
        iset = 0;
        //read distribution parameters
        try{
            DOM_Node meanNode = DOMUtil::findNode(distrNode, "mean");
            DOM_Node mvalueNode = meanNode.getFirstChild();
            std::string mvalue = toString(mvalueNode.getNodeValue());
            double tmean;
            std::stringstream(mvalue)>> tmean;
            setMean(tmean);
        }
        catch (DOMUtil::NodeNotFound) {
            std::cerr<<"Gaussian: no 'mean' specified by user. Use default mean: "
                     <<getMean()<<endl;
        }
} try{
DOMNode scaleNode = DOMUtil::findNode(distrNode, "scale");
DOMNode svalueNode = scaleNode.getFirstChild();
std::string svalue = toString(svalueNode.getNodeValue());
double tscale;
std::stringstream(svalue)>> tscale;
setScale(tscale);
} catch(DOMUtil::NodeNotFound) {
    std::cerr<<"Gaussian: no 'scale' specified by user. Use default scale: " <<getScale()<<endl;
}

} } // end namespace Gaussian

} // end namespace Random

#include <g++-tutorial/Random.h>

template<class ResT>
double GaussDistr<ResT>::getMean () const { return mean; } 

template<class ResT>
double GaussDistr<ResT>::getScale () const { return scale; }

template<class ResT>
void GaussDistr<ResT>::setMean (double _mean) {
    mean = _mean ;
    iset = 0;
}

template<class ResT>
void GaussDistr<ResT>::setScale (double _scale) {
    scale = _scale ;
    iset = 0;
}

template<class ResT>
ResT GaussDistr<ResT>::operator()(int, int, int) const {
    Random& generator = *Random::instance();
double v1, v2, rsq, fac;
if (iset == 0){
    do{
        v1= 2.0 * generator.nextFloat() - 1.0;
        v2= 2.0 * generator.nextFloat() - 1.0;
        rsq = v1*v1 + v2*v2;
    } while (rsq >= 1.0 || rsq == 0.0);
    fac = sqrt(-2.0 * log(rsq)/rsq);
gset = v1 * fac;
    iset = 1;
    return convert(v2*fac * scale + mean);
} else{
    iset = 0;
    return convert(gset * scale + mean);
}
}
inline int GaussDistr<int>::convert(double val) const
{
    return static_cast<int>(val+0.5);
}

inline double GaussDistr<double>::convert(double val) const
{
    return val;
}

#endif

B.5.5.6 Log-Normal Distribution

---

```cpp
#ifndef LOGNORMALDISTR_HH
#define LOGNORMALDISTR_HH

#include "ArrayParamFunctor.hh"
#include "GaussDistr.hh"
#include "dom/DOM.hpp"
#include <cmath>
#include "Random.hh"
#include <sstream>
#include <string>
#include "util.hh"
#include <iostream>
// This class generates lognormal deviates.
// P(x) = 1 / \( (S \cdot x \cdot \sqrt{2 \cdot \pi}) \cdot \exp(- (\ln(x) - M)^2 / (S^2)) \)

template<class ResT> class LogNormalDistr: public ArrayParamFunctor<ResT> {
public:
    LogNormalDistr(double mean = 0, double scale = 10);
    LogNormalDistr(DOMNode const&);
    double getMean () const;
    double getScale () const;
    void setMean (double);
    void setScale (double);
    ResT operator()(int = 0, int =0, int=0) const;
private:
    double mean;
    double scale;
    mutable double iset; // flag to see if extra deviate is handy
    mutable double gset; // place to keep extra deviate
    ResT convert(double) const;
};

template<class ResT>
LogNormalDistr<ResT>::LogNormalDistr(double _mean, double _scale) :
    mean(_mean), scale(_scale) {iset = 0;} // initialization

template<class ResT>
LogNormalDistr<ResT>::LogNormalDistr(DOMNode const& distrNode):
    mean(0), scale(10) {
    iset = 0;
    // read distribution parameters
    // The 'mean' and 'scale' are actually the 'M' and 'S' parameter of
    // the LogNormal where M is the mean of the normal, and S is the scale
    // of the normal
    try{
        DOM_Node meanNode = DOMUtil::findNode(distrNode, "mean");
        DOM_Node mvalueNode = meanNode.getFirstChild();
        std::stringstream mvalue = toString(mvalueNode.getNodeValue());
        double tmean;
        std::stringstream(mvalue)>> tmean;
        setMean(tmean);
    }catch(...){}
};
```

---

93
catch(DOMUtil::NodeNotFound) {
    std::cerr<<"LogNormal: no 'mean' specified by user. Use default mean: " <<getMean()<<endl;
}

try{
    DOM_Node scaleNode = DOMUtil::findNode(distrNode, "scale");
    DOM_Node svalueNode = scaleNode.getFirstChild();
    std::string svalue = toString(svalueNode.getNodeValue());
    double tscale;
    std::stringstream(svalue) >> tscale;
   .setScale(tscale);
} catch(DOMUtil::NodeNotFound) {
    std::cerr<<"LogNormal: no 'scale' specified by user. Use default scale: " <<getScale()<<endl;
}

} //template

template<class ResT>
double LogNormalDistr<ResT>::getMean () const { return mean; }

template<class ResT>
double LogNormalDistr<ResT>::getScale () const { return scale; }

template<class ResT>
void LogNormalDistr<ResT>::setMean (double _mean) {
    mean = _mean ;
    iset = 0;
}

template<class ResT>
void LogNormalDistr<ResT>::setScale (double _scale) {
    scale = _scale ;
    iset = 0;
}

template<class ResT>
ResT LogNormalDistr<ResT>::operator()(int, int, int) const {
    GaussDistr<double> gauss;
    //double S = sqrt(log(scale*scale/(mean*mean) + 1));
    //double M = log(mean) - S*S/2;
    gauss.setMean(mean);
    gauss.setScale(scale);
    return static_cast<ResT>(exp(gauss()));
}
#ifndef TABLE_FUNCHH
#define TABLE_FUNCHH
#include "ArrayParamFunctor.hh"
#include "util.hh"
#include <iostream>
#include "dom/DOM.hpp"
#include <map>
#include <string>
#include <sstream>
using std::string;
template<class ResT> class TableFunc: public ArrayParamFunctor<ResT> {

public:
    typedef ResT Result;
    TableFunc(DOMNode const&); //constructor
    ResT operator()(int, int = 0, int = 0) const;

private:
    std::map<int, ResT> table;
    typedef std::map<int, ResT>::constiterator constiterator;
    void readEntry(DOMNode const&);
};

template<class ResT> inline
TableFunc<ResT>::TableFunc(DOMNode const& funcNode) {
    try{
        table.clear();
        DOMNode entryNode = DOMUtil::findNode(funcNode, "entry");
        while(entryNode != 0){
            readEntry(entryNode);
            //look for the next entry
            entryNode = DOMUtil::findNextNode(entryNode, "entry");
        }
    }
    catch(DOMUtil::NodeNotFound){
        std::cerr<<"TableFunc: no entry found"<<endl;
    }
    catch(DOMUtil::NextNodeNotFound){}
}

template<class ResT> inline
void TableFunc<ResT>:::readEntry(DOMNode const& entryNode){
    try{
        DOMNode keyNode = DOMUtil::findNode(entryNode, "key");
        DOMNode valNode = DOMUtil::findNode(entryNode, "val");

        string keystr = toString(keyNode.getFirstChild().getNodeValue());
        string valstr = toString(valNode.getFirstChild().getNodeValue());

        int key;
        ResT val;
std::stringstream(keystr) >> key;
std::stringstream(valstr) >> val;

table[key] = val;
}
catch(DOMUtil::NodeNotFound){
    std::cerr<<"TableFunc: no key or val in entry found"<<endl;
}
}

template<class ResT> inline
ResT TableFunc<ResT>::operator()(int argument, int, int) const{
    const_iterator result;
    result = table.lower_bound(argument);
    if(table.size() == 0) return ResT(); //Empty table return default value
    if(result == table.end()) return (*--result).second; //past end of table
    if(result == table.begin()) return result->second; //before beginning
    if(result->first == argument) return result->second; //found exact match
    //Otherwise interpolate
    return (*--result).second;
}
#endif
B.7 Gene Array Data Structure

B.7.1 Gene Array

---------------GeneArray.hh------------------------------------

#ifndef GENEARRAY_HH
#define GENEARRAY_HH
#include <vector>
#include "Visitor.hh"
#include "Gene.hh"
#include <iostream>

class GeneArray : public std::vector<Gene>{
public:
    GeneArray(int numGenes = 6624) : std::vector<Gene>(numGenes) {} 
    virtual void accept(Visitor& v){
        v.visitGeneArray(this);
    }
    void setNumGenes(int numGenes) {resize(numGenes);}
    int getNumGenes() const {return size();}
    void setFractionR(double fR) {fractionR = fR;}
    double getFractionR() const {return fractionR;}
private:
    double fractionR;
};

std::ostream& operator<< (std::ostream&, GeneArray const&);
#endif

---------------GeneArray.cc--------------------------------------------------

#include "GeneArray.hh"
#include <iostream>
#include "Gene.hh"

std::ostream& operator<< (std::ostream& outfile, GeneArray const& array)
{
    for(GeneArray::const_iterator i = array.begin(); i != array.end(); ++i) {
        Gene gene = *i;
        outfile << gene;
    }
    return outfile;
}
B.7.2 Gene

----------------------------------------Gene.hh-----------------------------------------
#ifndef GENE_HH
#define GENE_HH
#include <vector>
#include "Visitor.hh"
#include "GeneSpot.hh"
#include <iostream>

class Gene : public std::vector<GeneSpot>{
    public:
        typedef int ID;
        Gene(int numSpots = 1) : std::vector<GeneSpot>(numSpots) {}
        virtual void accept(Visitor& v){
            v.visitGene(this);
        }
        void setNumSpots(int numSpots) {resize(numSpots);}
        int getNumSpots() const {return size();}
        void setId(ID _id) {_id = _id;}
        ID getId() const {return id;}
        void setProbBind(double pb) {probBind = pb;}
        double getProbBind() const {return probBind;}
        void setFractionR(double fR) {fractionR = fR;}
        double getFractionR() const {return fractionR;}
    private:
        ID id;
        double probBind;
        double fractionR;
    }

    std::ostream& operator<< (std::ostream&, Gene const&);
#endif

----------------------------------------Gene.cc----------------------------------------
#include "Gene.hh"
#include "GeneSpot.hh"
#include <iostream>

std::ostream& operator<< (std::ostream& outfile, Gene const& gene)
{
    for(Gene::const_iterator i = gene.begin(); i != gene.end(); ++i){
        GeneSpot spot = *i;
        outfile<<Gene "<<gene.getId()<<endl;
        outfile << spot;
    }
    return outfile;
}
B.7.3 Gene Spot

-------------------------- GeneSpot.hh----------------------------------

#ifndef GENESPOTHH
#define GENESPOT_HH
#include <vector>
#include "Visitor.hh"
#include "GenePixel.hh"
#include <iostream>

class GeneSpot : public std::vector<GenePixel>{
  public:
    typedef int ID;
    GeneSpot(int numPixels = 113):
      std::vector<GenePixel>(numPixels) {}
    virtual void accept(Visitor& v){
      v.visitGeneSpot(this);
    }
    void setNumPixels(int numPixels) {resize(numPixels);}
    int getNumPixels() const {return size();}
    void setId(ID _id) {id = _id;}
    ID getId() const {return id;}
    void setCascadeMeanR(double mR) {cascadeMeanR = mR;}
    double getCascadeMeanR() const {return cascadeMeanR;}
    void setCascadeMeanG(double mG) {cascadeMeanG = mG;}
    double getCascadeMeanG() const {return cascadeMeanG;}
    void setMeanNumDNAs(double mN) {meanNumDNAs = mN;}
    double getMeanNumDNAs() const {return meanNumDNAs;}
  private:
    ID id;
    double cascadeMeanR;
    double cascadeMeanG;
    double meanNumDNAs;
};

std::ostream& operator<< (std::ostream& outfile, GeneSpot const& spot)
{
  outfile<<"Spot \"<< spot.getId()<<'\t'
    "Intensities\t"<<spot.getNumPixels()<<endl;
  for(GeneSpot::const_iterator i = spot.begin(); i != spot.end(); ++i) {
    GenePixel pixel = *i;
    outfile << pixel;
  }
  outfile<<endl;
  return outfile;
}

--------------------------- GeneSpot.cc---------------------------------

#include "GeneSpot.hh"
#include <iostream>
#include "GenePixel.hh"

std::ostream& operator<< (std::ostream& outfile, GeneSpot const& spot)
{
  outfile<<"Spot " << spot.getId()<<'\t'
    "Intensities\t"<<spot.getNumPixels()<<endl;
  for(GeneSpot::const_iterator i = spot.begin(); i != spot.end(); ++i) {
    GenePixel pixel = *i;
    outfile << pixel;
  }
  outfile<<endl;
  return outfile;
}
B.7.4 Gene Pixel

#ifndef GENEPIXEL_HH
#define GENEPIXEL_HH
#include "Visitor.hh"
#include <iostream>

class GenePixel {
public:
    typedef int ID;
    virtual void accept(Visitor& v) {
        v.visitGenePixel(this);
    }
    int getNumDNAs() const { return numDNA; }
    int getNumRcDNAs() const { return numRcDNA; }
    int getNumGcDNAs() const { return numGcDNA; }
    int getRCount() const { return redCount; }
    int getGCount() const { return greenCount; }
    int getRChanPhotons() const { return rchanPhoton; }
    int getGChanPhotons() const { return gchanPhoton; }
    double getCascadeMeanR() const { return cascadeMeanR; }
    double getCascadeMeanG() const { return cascadeMeanG; }
    double getFractionR() const { return fractionR; }
    double getProbBind() const { return probBind; }

    void setNumDNAs(int nDNA) { numDNA = nDNA; }
    void setNumRcDNAs(int nR) { numRcDNA = nR; }
    void setNumGcDNAs(int nG) { numGcDNA = nG; }
    void setRCount(int R) { redCount = R; }
    void setGCount(int G) { greenCount = G; }
    void setRChanPhotons(int Rpho) { rchanPhoton = Rpho; }
    void setGChanPhotons(int Gpho) { gchanPhoton = Gpho; }
    void setCascadeMeanR(double mR) { cascadeMeanR = mR; }
    void setCascadeMeanG(double mG) { cascadeMeanG = mG; }
    void setFractionR(double fR) { fractionR = fR; }
    void setProbBind(double pb) { probBind = pb; }
    void setID(ID _id) { id = _id; }
    ID getID() const { return id; }

private:
    int numRcDNA;
    int numGcDNA;
    int redCount;
    int greenCount;
    int numDNA;
    int rchanPhoton;
    int gchanPhoton;
    double cascadeMeanR;
    double cascadeMeanG;
    double fractionR;
    double probBind;
    ID id;
};
std::ostream& operator<< (std::ostream&, GenePixel const&);
#endif
#include "GenePixel.hh"
#include <iostream>

std::ostream& operator<< (std::ostream& outfile, GenePixel const& genepixel) {
    outfile<<genepixel.getRCount()<<'\t'
    <<genepixel.getGCount()<<endl;
    return outfile;
}
B.8 Visitors

B.8.1 Visitor Super Class

---Visitor.hh---

```c++
#ifndef VISITORHH
#define VISITORHH

class GeneArray;
class Gene;
class GeneSpot;
class GenePixel;

class Visitor{
    public:
        virtual ~Visitor() {};
        virtual void visitGeneArray(GeneArray*);
        virtual void visitGene(Gene*);
        virtual void visitGeneSpot(GeneSpot*);
        virtual void visitGenePixel(GenePixel*);
    
};
#endif
```

---Visitor.cc---

```c++
#include "Visitor.hh"
#include "GeneArray.hh"
#include "Gene.hh"
#include "GeneSpot.hh"
#include "GenePixel.hh"

void Visitor::visitGeneArray(GeneArray* data) {
    for(GeneArray::iterator i = data->begin(); i != data->end(); ++i)
        i->accept(*this);
}

void Visitor::visitGene(Gene* cohort) {
    for(Gene::iterator i = cohort->begin(); i != cohort->end(); ++i)
        i->accept(*this);
}

void Visitor::visitGeneSpot(GeneSpot* spot) {
    for(GeneSpot::iterator i = spot->begin(); i != spot->end(); ++i)
        i->accept(*this);
}

void Visitor::visitGenePixel(GenePixel* pixel) {}
```

102
B.8.2 ConstructArrayVisitor

------------ConstructArrayVisitor.hh--------------------------
#ifndef CONSTRUCTARRAYVISITORHH
#define CONSTRUCTARRAYVISITORHH
#include "Visitor.hh"
#include "Gene.hh"
#include "GeneSpot.hh"
#include "GenePixel.hh"
#include "ArrayParam.hh"
#include <vector>

// Construct the gene array data structure
class GeneArray;

class ConstructArrayVisitor: public Visitor {
public:
    ConstructArrayVisitor(ArrayParam const*);
    virtual void visitGeneArray(GeneArray*);
    virtual void visitGene(Gene*);
    virtual void visitGeneSpot(GeneSpot*);
    virtual void visitGenePixel(GenePixel*);
private:
    ArrayParam const* ap;
    Gene::ID geneID;
    GeneSpot::ID spotID;
    GenePixel::ID pixelID;
    Gene* genePtr;
    GeneSpot* spotptr;
    double spotS;
    double spotM;
};
#endif

------------ConstructArrayVisitor.cc-----------------------
#include "ConstructArrayVisitor.hh"
#include "ArrayParam.hh"
#include "GeneArray.hh"
#include "Gene.hh"
#include "GeneSpot.hh"
#include "GenePixel.hh"
#include <iostream>
#include <vector>
#include "PoissDistr.hh"
#include "UniformDistr.hh"
#include "LogNormalDistr.hh"

ConstructArrayVisitor::ConstructArrayVisitor(ArrayParam const* _ap):
    ap(_ap){}

void ConstructArrayVisitor::visitGeneArray(GeneArray* array)
{
    array->setNumGenes(ap->getNumGenes());
}
int genemod = static_cast<int> (0.01*array->getNumGenes());
int linemod = 20;
int lcounter = 0;
geneID = 1;
for (GeneArray::iterator i = array->begin(); i != array->end(); ++i) {
    i->setID(geneID);
    i->accept(*this);
    ++geneID;
    if (!(geneID % genemod)) {
        std::cerr<<'*';
        ++lcounter;
        if (!(lcounter % linemod))
            std::cerr<<endl;
    }
}
std::cerr<<endl;

void ConstructArrayVisitor::visitGene(Gene* gene) {
    gene->setNumSpots(ap->getNumSpots(gene->getID()));
    gene->setProbBind(ap->getProbBind(gene->getID()));
    gene->setFractionR(ap->getFractionR(gene->getID()));
    geneptr = gene;
    spotID = 1;
    for (Gene::iterator i = gene->begin(); i != gene->end(); ++i) {
        i->setID(spotID);
        i->accept(*this);
        ++spotID;
    }
}

void ConstructArrayVisitor::visitGeneSpot(GeneSpot* spot) {
    spot->setNumPixels(ap->getNumPixels(geneID, spot->getID()));
    spot->setCascadeMeanR(ap->getCascadeMeanR(geneID, spot->getID()));
    spot->setCascadeMeanG((spot->getCascadeMeanR())
        * (ap->getCascadeMeanRatio(geneID, spot->getID())));
    spotptr = spot;
    //Compute gamma -- multiplier for pixel LogNormal mean
    UniformDistr<double> selectdistr;
    double gamma;
    //choose LogNormal 90% of the time,and Uniform 10% of the time
    if (selectdistr() <= 0.9) {
        double M = -1.56;
        double S = 0.5;
        LogNormalDistr<double> gdistr(M, S);
        gamma = gdistr();
    } else{
        double a = -1;
        double b = 0.5;
        UniformDistr<double> gdistr(a, b);
        gamma = exp(gdistr());
    }
// solve for pixel LogNormal M, S
double u = spot->getMeanNumDNAs();
spotS = sqrt(log(gamma*gamma + 1));
spotM = 0.5*log(u*u / (gamma*gamma +1));

pixelID = 1;
for(GeneSpot::iterator i = spot->begin(); i != spot->end(); ++i) {
    i->setID(pixelID);
    i->accept(*this);
    ++pixelID;
}

void ConstructArrayVisitor::visitGenePixel(GenePixel* pixel)
{
    LogNormalDistr<int> distr(spotM, spotS);
pixel->setNumDNAs(distr());
pixel->setFractionR(geneptr->getFractionR());
pixel->setProbBind(geneptr->getProbBind());
pixel->setCascadeMeanR(spotptr->getCascadeMeanR());
pixel->setCascadeMeanG(spotptr->getCascadeMeanG());
}
B.8.3 ComputeCDNAVisitor

---------------------------- ComputeCDNAVisitor.hh----------------------------

#ifndef COMPUTE_CDNA_VISITOR_HH
#define COMPUTE_CDNA_VISITOR_HH
#include "Visitor.hh"
class GeneArray;
class Gene;
class GeneSpot;
class GenePixel;
class ComputeCDNAVisitor: public Visitor {
  public:
    ComputeCDNAVisitor();
    virtual void visitGeneArray(GeneArray*);
    virtual void visitGene(Gene*);
    virtual void visitGeneSpot(GeneSpot*);
    virtual void visitGenePixel(GenePixel*);
};
#endif

---------------------------- ComputeCDNAVisitor.cc----------------------------

#include "ComputeCDNAVisitor.hh"
#include "GeneArray.hh"
#include "Gene.hh"
#include "GeneSpot.hh"
#include "GenePixel.hh"
#include "BinomDistr.hh"
ComputeCDNAVisitor::ComputeCDNAVisitor(){}

void ComputeCDNAVisitor::visitGeneArray(GeneArray* array)
{
  int genemod = static_cast<int>(0.01*array->getNumGenes());
  int gcountr = 0;
  int lcounter = 0;
  for(GeneArray::iterator i = array->begin(); i != array->end(); ++i)
  {
    i->accept(*this);
    // progress meter
    if (!(gcountr % genemod))
    {
      std::cerr<<'+';
      ++lcounter;
    }
    if (!(lcounter % linemod))
      std::cerr<<endl;
    ++gcountr;
  }
  std::cerr<<endl;
}
void ComputeCDNAVisitor::visitGene(Gene* gene)
{
    for(Gene::iterator i = gene->begin(); i != gene->end(); ++i) {
        i->accept(*this);
    }
}

void ComputeCDNAVisitor::visitGeneSpot(GeneSpot* spot)
{
    for(GeneSpot::iterator i = spot->begin(); i != spot->end(); ++i) {
        i->accept(*this);
    }
}

void ComputeCDNAVisitor::visitGenePixel(GenePixel* pixel)
{
    int nDNAs = pixel->getNumDNAs();
    double probBind = pixel->getProbBind();
    double fR = pixel->getFractionR();
    BinomDistr binom(nDNAs, probBind);
    int ncDNAs = binom();
    BinomDistr binomR(ncDNAs, fR);
    pixel->setNumRcDNAs(binomR());
    pixel->setNumGcDNAs(ncDNAs - pixel->getNumRcDNAs());
}
B.8.4 ComputeChanPhotonsVisitor

-------------------- ComputeChanPhotonsVisitor.hh -------------------

#ifndef COMPCHANPHOTONS_VISITOR_HH
#define COMPCHANPHOTONS_VISITOR_HH

#include "Visitor.hh"
#include "ArrayParam.hh"

class GeneArray;
class Gene;
class GeneSpot;
class GenePixel;

class CompChanPhotonsVisitor: public Visitor {
  public:
    CompChanPhotonsVisitor(ArrayParam const*);
    virtual void visitGeneArray(GeneArray*);
    virtual void visitGene(Gene*);
    virtual void visitGeneSpot(GeneSpot*);
    virtual void visitGenePixel(GenePixel*);
  private:
    ArrayParam const* ap;
};

#endif

-------------------- ComputeChanPhotonsisitor.cc ---------------------

#include "CompChanPhotonsVisitor.hh"
#include "GeneArray.hh"
#include "Gene.hh"
#include "GeneSpot .hh"
#include "GenePixel.hh"
#include "ArrayParam.hh"
#include "BinomDistr.hh"
#include "PoisDistr.hh"

CompChanPhotonsVisitor::CompChanPhotonsVisitor(ArrayParam const* _ap): ap(_ap) {};

void CompChanPhotonsVisitor::visitGeneArray(GeneArray* array)
{
  int genemod = static_cast<int>(0.01*array->getNumGenes());
  int gcounter = 0;
  int linemod = 20;
  int lcounter = 0;
  for(GeneArray::iterator i = array->begin; i != array->end(); ++i)
    {
      i->accept(*this);
      // progress meter
      if (!(gcounter % genemod)){
        std::cerr<<',';
        ++lcounter;
      }
      if (!(lcounter % linemod))
        std::cerr<<endl;
    }
++gcounter;
}
std::cerr<<endl;
}

void CompChanPhotonsVisitor::visitGene(Gene* gene)
{
    for(Gene::iterator i = gene->begin(); i != gene->end(); ++i) {
        i->accept(*this);
    }
}

void CompChanPhotonsVisitor::visitGeneSpot(GeneSpot* spot)
{
    for(GeneSpot::iterator i = spot->begin(); i != spot->end(); ++i) {
        i->accept(*this);
    }
}

void CompChanPhotonsVisitor::visitGenePixel(GenePixel* pixel)
{
    int nCDNAR = pixel->getNumRcDNAs();
    int nCDNAG = pixel->getNumGcDNAs();
    double probBind = pixel->getProbBind();
    double fractionR = pixel->getFractionR();
    double cascadeMeanR = pixel->getCascadeMeanR();
    double cascadeMeanG = pixel->getCascadeMeanG();
    double probFilterRtoR = ap->getProbFilterRtoR();
    double probFilterRtoG = 1 - probFilterRtoR;
    double probFilterGtoG = ap->getProbFilterGtoG();
    double probFilterGtoR = 1 - probFilterGtoG;
    double probDetectR = ap->getProbDetectR();
    double probDetectG = ap->getProbDetectG();
    double mR = nCDNAR * cascadeMeanR; // mean # of R photons emitted from pixel
    double mG = nCDNAG * cascadeMeanG; // mean # of G photons emitted from pixel
    PoissDistr<int> poissR(mR);
    int nR = poissR();  // # of R photons emitted from pixel
    PoissDistr<int> poissG(mG);
    int nG = poissG();  // # of G photons emitted from pixel

    // Compute R channel photons detected
    BinomDistr binomRR(nR, probFilterRtoR);
    int nrr = binomRR();  // # of R photons filtered into R channel
    BinomDistr binomGR(nG, probFilterGtoR);
    int ngr = binomGR();  // # of G photons filtered into R channel
    BinomDistr binomDetectRR(nrr, probDetectR);
    int nrrDetect = binomDetectRR();  // # of R photons detected in R channel of pixel
BinomDistr binomDetectGR(ngr, probDetectG);
    int ngrDetect = binomDetectGR(); // # of G photons detected in R channel of pixel
    pixel->setRChanPhotons(nrrDetect + ngrDetect);

    // Compute G channel photons detected
    BinomDistr binomRG(nR, probFilterRtoG);
    int nrg = binomRG(); // # of R photons filtered into G channel of pixel
    BinomDistr binomGG(nG, probFilterGtoG);
    int ngg = binomGG(); // # of G photons filtered into G channel of pixel

    BinomDistr binomDetectRG(nrg, probDetectR);
    int nrgDetect = binomDetectRG(); // # of R photons detected in G channel of pixel
    BinomDistr binomDetectGG(ngg, probDetectG);
    int nggDetect = binomDetectGG(); // # of G photons detected in G channel of pixel
    pixel->setGChanPhotons(nrgDetect + nggDetect);
B.8.5 ComputeRGVisitor

--------------------------- ComputeRGVisitor.hh------------------------

#ifndef COMP_RG_VISITOR_HH
#define COMP_RG_VISITOR_HH

#include "Visitor.hh"
#include "GeneArray.hh"
#include "Gene.hh"
#include "GeneSpot.hh"
#include "GenePixel.hh"
#include "ArrayParam.hh"

class ComputeRGVisitor: public Visitor {
  public:
    ComputeRGVisitor(ArrayParam const*);
    virtual void visitGeneArray(GeneArray*);
    virtual void visitGene(Gene*);
    virtual void visitGeneSpot(GeneSpot*);
    virtual void visitGenePixel(GenePixel*);

  private:
    ArrayParam const* ap;
    Gene::ID geneID;
    GeneSpot::ID spotID;
    GenePixel::ID pixelID;

};

#endif

----------------------- ComputeRGVisitor.cc----------------------------

#include "ComputeRGVisitor.hh"
#include "GeneArray.hh"
#include "Gene.hh"
#include "GeneSpot.hh"
#include "GenePixel.hh"
#include "ArrayParam.hh"
#include "PoissDistr.hh"
#include "BinomDistr.hh"
#include <iostream>

ComputeRGVisitor::ComputeRGVisitor(ArrayParam const* _ap):
  ap(_ap){}

void ComputeRGVisitor::visitGeneArray(GeneArray* array)
{
  int genemod = static_cast<int> (0.01*array->getNumGenes());
  int linemod = 20;
  int lcounter = 0;
  geneID = 1;
  for(GeneArray::iterator i = array->begin(); i != array->end(); ++i)
  {
    i->accept(*this);
    ++geneID;
    if(!(geneID % genemod))
      std::cerr<<'.';
++lcounter;
if ( !(lcounter % linemod))
    std::cerr<<endl;
}
std::cerr<<endl;
}

void ComputeRGVisitor::visitGene(Gene* gene)
{
    for(Gene::iterator i = gene->begin(); i != gene->end(); ++i) {
        i->accept(*this);
    }
}

void ComputeRGVisitor::visitGeneSpot(GeneSpot* spot)
{
    for(GeneSpot::iterator i = spot->begin(); i != spot->end(); ++i) {
        i->accept(*this);
    }
}

void ComputeRGVisitor::visitGenePixel(GenePixel* pixel)
{
    int RchanPhotons = pixel->getRChanPhotons();
    int GchanPhotons = pixel->getGChanPhotons();

    int gainR = ap->getGainR();
    int gainG = ap->getGainG();
    pixel->setRCount(RchanPhotons * gainR);
    pixel->setGCount(GchanPhotons * gainG);
}
B.8.6 ComputeRGMeanStdevVisitor

------------------------ ComputeRGMeanStdevVisitor.hh-------------------

#ifndef COMPUTE_RG_MEAN_STDEV_VISITOR_HH
#define COMPUTE_RG_MEAN_STDEV_VISITOR_HH

#include <vector>
#include "Tuple.hh"
#include "Visitor.hh"

class ComputeRGMeanVisitor : public Visitor {
  public:
    typedef std::vector<double> Result;
    void visitGeneSpot(GeneSpot*);
    Result getResult() const { return result; }
  private:
    Result result;
};

class ComputeRGStdevVisitor : public Visitor {
  public:
    typedef std::vector<double> Result;
    void visitGeneSpot(GeneSpot*);
    Result getResult() const { return result; }
  private:
    Result result;
};

class ComputeRGMeanStdevVisitor : public Visitor {
  public:
    typedef Tuple<2, double> MeanAndStdev;
    typedef std::vector<MeanAndStdev> Result;
    void visitGeneArray(GeneArray*);
    Result getResult() const { return result; }
  private:
    Result result;
};
#endif

---------------------- ComputeRGMeanStdevVisitor.cc---------------------

#include <cmath>
#include "ComputeRGMeanStdevVisitor.hh"
#include "GeneArray.hh"
#include "GeneSpot.hh"
#include "Tuple.hh"

void ComputeRGMeanStdevVisitor::visitGeneArray(GeneArray* data) {
  ComputeRGMeanVisitor meanVis;
  ComputeRGStdevVisitor stdevVis;
  data->accept(meanVis);
  data->accept(stdevVis);
```cpp
ComputeRGMeanVisitor::Result means = meanVis.getResult();
ComputeRGStdevVisitor::Result stddev = stdevVis.getResult();

int genemod = static_cast<int>(0.01*data->getNumGenes());
int linemod = 20;
int lcounter = 0;
int gcounter = 0;
for(unsigned int i = 0; i < means.size() && i < stddev.size(); ++i)
{
    result.push_back(MeanAndStdev(means[i], stddev[i]));
    // progress meter
    if (!(gcounter % genemod)){
        std::cerr << '#';
        ++lcounter;
    }
    if (!(lcounter % linemod))
        std::cerr << endl;
    ++gcounter;
}

std::cerr << endl;

void ComputeRGMeanVisitor::visitGeneSpot(GeneSpot* spot) {
    // Compute total intensity mean
    double mean = 0;
    for(GeneSpot::iterator i = spot->begin(); i != spot->end(); ++i) {
        mean += i->getRCount();
        mean += i->getGCount();
    }
    mean /= spot->getNumPixels();
    result.push_back(mean);
}

void ComputeRGStdevVisitor::visitGeneSpot(GeneSpot* spot) {
    // Compute total intensity mean
    double mean = 0;
    for(GeneSpot::iterator i = spot->begin(); i != spot->end(); ++i) {
        mean += i->getRCount();
        mean += i->getGCount();
    }
    mean /= spot->getNumPixels();

    // Compute total intensity stdev
    double stdev = 0;
    for(GeneSpot::iterator i = spot->begin(); i != spot->end(); ++i) {
        stdev += (i->getRCount() + i->getGCount() - mean)*
            (i->getRCount() + i->getGCount() - mean);
    }
    stdev /= spot->getNumPixels();
    stdev = sqrt(stdev);
    result.push_back(stdev);
}
```
B.8.7 ComputeRGRatioStdevVisitor

ifndef COMPUTE_RG_RATIO_STDEV_VISITOR_HH
#define COMPUTE_RG_RATIO_STDEV_VISITOR_HH

#include <vector>
#include "Visitor.hh"

class ComputeRGRatioStdevVisitor : public Visitor {
public:
  typedef std::vector<double> Result;
  void visitGeneSpot(GeneSpot*);
  Result getResult() const { return result; }

private:
  Result result;
};
#endif

#include <cmath>
#include "ComputeRGRatioStdevVisitor.hh"
#include "GeneSpot.hh"
#include <iostream>

void ComputeRGRatioStdevVisitor::visitGeneSpot(GeneSpot* spot) {
  //Compute total intensity mean
  double meanLogRatio = 0;
  double logratio = 0;
  double r = 0;
  double g = 0;
  for(GeneSpot::iterator i = spot->begin(); i != spot->end(); ++i) {
    r = i->getRCount(); //make sure r and g are not zero
    g = i->getGCount();
    logratio = log(r/g) / log(2); //log base 2
    meanLogRatio += logratio;
  }
  meanLogRatio /= spot->getNumPixels();

  //Compute total intensity std dev
  double stdev = 0;
  for(GeneSpot::iterator i = spot->begin(); i != spot->end(); ++i) {
    r = i->getRCount(); //make sure r and g are not zero
    g = i->getGCount();
    logratio = log(r/g) / log(2);
    stdev += (logratio - meanLogRatio) * (logratio - meanLogRatio);
  }
  stdev /= spot->getNumPixels();
  stdev = sqrt(stdev);
  result.push_back(stdev);
}
B.9 Reporters

B.9.1 Reporter Super Class

```
#ifndef REPORTER_HH
#define REPORTER_HH
#include <iosfwd>

class ComputeRGMeanStdevVisitor;

class Reporter {
    public:
        virtual void reportStats(ComputeRGMeanStdevVisitor const &)
    }

inline std::ostream& operator<< (std::ostream& stream, Reporter const&) {return stream;}
#endif
```
B.9.2 StatsReporter

------------------------ StatsReporter.hh-------------------------------

#ifndef STATS_REPORTER_HH
#define STATS_REPORTER_HH
#include <iosfwd>
#include <vector>
#include "Reporter.hh"
#include "ComputeRGMeanStdevVisitor.hh"

class StatsReporter: public Reporter {
    friend std::ostream& operator<< (std::ostream& outfile,
                        StatsReporter const& reporter);

    public:
        virtual void reportStats(ComputeRGMeanStdevVisitor const &);

    private:
        std::vector<ComputeRGMeanStdevVisitor> mean_stdev;
};

std::ostream& operator<< (std::ostream &,
                        StatsReporter const&)

#endif

----------------------- StatsReporter.cc--------------------------------

#include "StatsReporter.hh"
#include <iostream>
#include <vector>
#include "ComputeRGMeanStdevVisitor.hh"
#include "Tuple.hh"

void StatsReporter::reportStats(ComputeRGMeanStdevVisitor const& v)
{
    mean_stdev.push_back(v);
}

std::ostream& operator<< (std::ostream &ofile,
                        StatsReporter const& reporter)
{
    int maxRows = reporter.mean_stdev[0].getResult().size(); //first
    visitor result size
    int irows = 0;
    int Icounter = 0;
    for (int row = 0; row < maxRows; ++row) {
        //progress meter
        //for every 100 spots, print ^
        if (! (row % 100)) {
            std::cerr<<'^';
            ++Icounter;
        //for every 20 ^, goto the next line
            if (! (Icounter % 20)) {
                std::cerr<< endl;
            }
        }
    }
}
// go through each visitor
for (std::vector<ComputeRGMeanStdevVisitor>::const_iterator i=
    reporter.mean_stdev.begin(); i!=reporter.mean_stdev.end() ; ++i)
{
    irows = i->getResult().size();

    if (row > irows){
        outfile << 0 <<'\t' << 0 << '\t';
    }else{
        outfile <<i->getResult()[row][0] <<'\t'
               <<i->getResult()[row][1] <<'\t';
    }
}

return outfile;
B.9.3 RatioReporter

------------------------ RatioReporter.hh ------------------------

#ifndef RATIOREPORTERHH
#define RATIOREPORTERHH
#include <iosfwd>
#include <vector>
#include "Reporter.hh"
#include "ComputeRGRatioStdevVisitor.hh"

class RatioReporter: public Reporter {
   friend std::ostream& operator<< (std::ostream& outfile, RatioReporter const& reporter);

   public:
      virtual void reportStats(ComputeRGRatioStdevVisitor const &);

   private:
      std::vector<ComputeRGRatioStdevVisitor> ratio_stdev;
};

std::ostream& operator<< (std::ostream& outfile, RatioReporter const& reporter);

#endif

---------------------- RatioReporter.cc --------------------------------

#include "RatioReporter.hh"
#include <iostream>
#include <vector>
#include "ComputeRGRatioStdevVisitor.hh"
#include "Tuple.hh"

void RatioReporter::reportStats(ComputeRGRatioStdevVisitor const& v) {
   ratio_stdev.push_back(v);
}

std::ostream& operator<< (std::ostream& outfile, RatioReporter const& reporter) {
   int maxRows = reporter.ratio_stdev[0].getResult().size(); //first visitor result size
   int irows = 0;
   int lcounter = 0;
   for (int row = 0; row < maxRows; ++row) {
      //progress meter
      //for every 100 spots, print ^
      if (! (row % 100)) {
         std::cerr<<'^';
         ++lcounter;
         //for every 20 ^, goto the next line
         if (! (lcounter % 20)) {
            std::cerr<< endl;
         }
      }
   }

}
// go through each visitor
for (std::vector<ComputeRGRatioStdVisitor>::const_iterator i=
     reporter.ratio_stdev.begin(); i!=reporter.ratio_stdev.end(); ++i)
{
    irows = i->getResult().size();
    if (row > irows){
        outfile << 0 <<'\t';
    }
    else{
        outfile <<i->getResult()[row] <<'\t';
    }
}
outfile<<endl;
return outfile;
# B.10 Simulator main() program

```cpp
#include "ArrayParamFileParser.hh"
#include "ArrayParam.hh"
#include <iostream>
#include "ConstructArrayVisitor.hh"
#include "ComputeCDNAVisitor.hh"
#include "CompChanPhotonsVisitor.hh"
#include "ComputeRGVisitor.hh"
#include "GeneArray.hh"
#include <fstream>
#include "FileWriter.hh"

// Forward declaration
void simulate(ArrayParam*, GeneArray*);

// main
int main(int argC, char* argV[])
{
    char* inputFile;
    inputFile = argV[1];
    std::ofstream outputFile(argV[2]);
    // for debugging
    std::ofstream testFile(argV[3]);

    ArrayParamFileParser parser;
    parser.parse(inputFile);
    ArrayParam arrayparam = parser.getArrayParam();
    GeneArray array;
    simulate(&arrayparam, &array);
    outputFile << array;
    FileWriter::writeFile(array, testFile); // for debugging write out
    std::cerr << "done mysim" << endl;
    return 0;
}

void simulate(ArrayParam* ap, GeneArray* arrayptr)
{
    ConstructArrayVisitor constructV(ap);
    ComputeCDNAVisitor cDNAV;
    CompChanPhotonsVisitor chanV(ap);
    ComputeRGVisitor rgV(ap);
    arrayptr->accept(constructV);
    arrayptr->accept(cDNAV);
    arrayptr->accept(chanV);
    arrayptr->accept(rgV);
}
```
B.11 Utility Functions

--- util.hh ---

```cpp
#ifndef UTIL_HH
#define UTIL_HH
#include "dom/DOM.hpp"
#include <string>

std::string toString(const DOMString& s);

ostream& operator<<(ostream& target, const DOMString& toWrite);

namespace DOMUtil {
    class Exception {};
    class NodeNotFound : public Exception {};
    class NextNodeNotFound : public Exception {};
    DOM_Node findNode(DOM_Node const& strParentNode, std::string const& tString);
    DOM_Node findNextNode(DOM_Node const& strSiblingNode, std::string const& tString);
    std::string getAttribute(DOM_Node const& strNode, std::string const& strAttribute);
}
#endif
--- util.cc ---

```
DOMString childName = child.getNodeName();
if (toString(childName) == tString){
    strFound = true;
    strNode = child;
} else{
    child = child.getNextSibling();
}
}
if (strFound)
    return strNode;
else{
    throw NodeNotFound();
}

DOMNode DOMUtil::findNextNode(DOMNode const& strSibNode, std::string const& tString)
{
DOMNode strNode;
DOMNode nextSib = strSibNode.getNextSibling();
bool strFound = false;
while((nextSib != 0) && (! strFound))
{
    DOMString nextSibName = nextSib.getNodeName();
    if (toString(nextSibName) == tString){
        strFound = true;
        strNode = nextSib;
    } else{
        nextSib = nextSib.getNextSibling();
    }
}
if (strFound)
    return strNode;
else{
    throw NextNodeNotFound();
}
}

std::string DOMUtil::getAttribute(DOMNode const& theNode, std::string const& name)
{
DOMNode NamedNodeMap attributes = theNode.getAttributes();
int attrCount = attributes.getLength();
std::string attr;
std::string attrName;
for (int i = 0; i < attrCount; i++)
{
    DOMNode attribute = attributes.item(i);
    attrName = toString(attribute.getNodeName());
    if (attrName == name){
        attr = toString(attribute.getNodeValue());
    }
}
return attr;
}
B.12 Makefile

------------------------------make_sim---------------------------------

include -/Thesis/Xerces/xerces-c-srcl_6_0/version.incl
include ~/Thesis/Xerces/xerces-c-srcl_6_0/samples/Makefile.incl

PLATFORM = LINUX
CXXFLAGS = -w -O $(INCLUDES)
CFLAGS = -w -O
LDFLAGS =
PREFIX = /usr/local

COMPILER = g+

APP_NAME = sim

OBJS = FileWriter.o \
    GeneArray.o \
    Gene.o \
    GeneSpot.o \
    GenePixel.o \
    Visitor.o \
    ConstructArrayVisitor.o \
    ComputeCDNAVisitor.o \
    CompChanPhotonsVisitor.o \
    ComputerRGVisitor.o \
    ComputerRGMeanStdevVisitor.o \
    util.o \
    DOMTreeErrorReporter.o \
    ArrayParam.o \
    ArrayParamFileParser.o \
    ArrayParamFunctor.o \
    BinomDistr.o \
    Random.o \
    StatsReporter.o \
    sim.o \

${APP_NAME}:: ${OBJS}
g++ ${PLATFORM_LIB_LINK_OPTIONS} ${OBJS} -o $@
${LIBRARY_SEARCH_PATHS} ${LIBRARY_NAMES} ${EXTRA_LINK_OPTIONS}

clean::
    rm -f ${OBJS} ${APP_NAME}

distclean:: clean
    rm -f Makefile
B.13 Auxiliary Functions

B.13.1 FileWriter

---------------------------- FileWriter.hh--------------------------------
#ifndef FILEWRITERHH
#define FILEWRITERHH
#include "GeneArray.hh"
#include <iostream>
class FileWriter {
  public:
    static void writeFile(GeneArray const&, ostream&);
};
#endif
---------------------------- FileWriter.cc-------------------------------
#include "FileWriter.hh"
#include <iostream>
#include "GeneArray.hh"
#include "Gene.hh"
#include "GeneSpot.hh"
#include "GenePixel.hh"
#include <iostream>
void FileWriter::writeFile(GeneArray const& array, ostream& outfile)
{
  if (!outfile)
  {
    std::cerr << "Debug file not specified." << std::endl;
  }
  else{
    //read from array, write to output file
    outfile << "Array - numGenes: " << array.getNumGenes() << std::endl;
    for (GeneArray::const_iterator i = array.begin(); i != array.end(); ++i)
    {
      Gene gene = *i;
      outfile << "Gene " << gene.getID() << std::endl;
      outfile << "numSpots " << gene.getNumSpots() << std::endl;
      for (Gene::iterator i = gene.begin(); i != gene.end(); ++i)
      {
        GeneSpot spot = *i;
        outfile << "Spot " << spot.getID() << std::endl;
        outfile << "numPixels " << spot.getNumPixels() << std::endl;
        for (GeneSpot::iterator i = spot.begin(); i != spot.end(); ++i)
        {
          GenePixel pixel = *i;
          outfile << "Pixel " << pixel.getID() << std::endl;
          outfile << "numDNAs " << pixel.getNumDNAs() << std::endl;
          outfile << "probBind " << pixel.getProbBind() << std::endl;
          outfile << "fractionR " << pixel.getFractionR() << std::endl;
          outfile << "numRcDNAs " << pixel.getNumRcDNAs() << std::endl;
          outfile << "numGcDNAs " << pixel.getNumGcDNAs() << std::endl;
        }
      }
    }
  }
}
outfile<<"cascadeMeanR"<<pixel.getCascadeMeanR()<<endl;
outfile<<"cascadeMeanG"<<pixel.getCascadeMeanG()<<endl;
outfile<<"RChanPhotons"<<pixel.getRChanPhotons()<<endl;
outfile<<"GChanPhotons"<<pixel.getGChanPhotons()<<endl;
outfile<<"RCount "<<pixel.getRCount()<<endl;
outfile<<"GCount "<<pixel.getGCount()<<endl;
B.13.2 Random

---Random.hh---

 ifndef Random_HH
 define Random_HH

 /**********************************************************************
 Name: Random
 Author: Keith H. Duggar
 Modify: 2001.06.16
 Purpose: Class Random provides an interface for generating random
 numbers. In its current implementation it uses a shuffled prime
 modulus multiplicative linear congruential generator (PMMLCG). It uses
 the following constants recommended by Park and Miller (1988):
 modulus = 2^31 - 1
 multiplier = 69621
 Random is implemented as a singleton. A singleton allows only one copy
 of its self to exist. Clients do not have access to the default
 constructors. Instead, a pointer to the single existing is accessed by
 the instance() function. Generating pseudo-random numbers
 algorithmically is a delicate and tricky matter. The algorithm must be
 able to insure the order of output. If multiple copies of the
 generator existed, it would be possible for clients to generate
 identical sequences of numbers from different generators. The clients
 would have no way of knowing this was actually occurring. By making
 Random a singleton, the random ordering of the generated numbers can
 only be defeated by the client through explicit means. This would mean
 that at least they have knowledge of such explicit violation of
 randomness.

 As a note, the private constructors also prevent inheritance from
 Random. This is by design. I could see no reason for extension of
 Random except to provide alternate behavior for the fundamental
 interface (nextInt() for example). This would then require overloading
 of the fundamental interface methods. However, these are non-virtual
 functions. Overloading of non-virtual base member functions is
 generally poor style. Better would be to make the methods virtual.
 However, virtual functions would introduce both space and performance
 penalties into the entire Random derived tree.

 As random generators are typically used in high performance
 applications (as is my case) this would be unacceptable. Thus, the
 gains of a virtual base class seemed not to outweigh the associated
 penalties. Hence the private constructors and the intent that Random
 is a final class. If clients absolutely wish to subclass Random then
 the constructors should be made protected.

 **********************************************************************

 class Random { //FINAL
   public:
     int nextInt() const; //Return a number in the interval
     [0 , maxint] (closed)
double nextFloat () const;  //Return a number in the interval [0.0, 1.0] (closed)
double nextAngle () const;  //Return a number in the interval [0.0, 1.0) (semi-open)
double nextSlice () const;  //Return a number in the interval (0.0, 1.0) (open)

int getMaxInt () const;
double getMaxAngle () const;
double getMaxSlice () const;
double getMinSlice () const;

static Random* instance();  //Access the single copy of Random

private:
Random();  //Prevent client construction
Random(Random const&);  //Prevent client construction
Random& operator=(Random const&);  //Only one copy so prevent self-assignment

static int const maxInt;  //0x7FFFFFFD (2^31 - 3)
static int const modulus;  //0x7FFFFFFF (2^31 - 1)
static int const multiplier;  //69621
static int const quotient;  //30845
static int const remainder;  //23902

static int prevInt;  //Previous random number
static Random* instance_;  //The single copy of Random

//Calculate the next integer
inline int Random::nextInt() const {
    int nextInt;
    nextInt = multiplier * (prevInt % quotient);  //Calculate the next integer from
    nextInt -= remainder * (prevInt / quotient);  //previously generated integer using
    if(nextInt < 1) nextInt += modulus;  //Schrage's method (1983)
    prevInt = nextInt;
    return nextInt;
}

//Return a uniform deviate by normalizing nextInt() to the appropriate interval
inline double Random::nextFloat() const {return (double(nextInt())) / double(maxInt);}
inline double Random::nextAngle() const {return (double(nextInt())) / double(maxInt + 1));
inline double Random::nextSlice() const {return (double(nextInt()) + 1) / double(maxInt + 2));

#endif
#include <ctime>
#include "Random.hh"

int const Random::maxInt = (0x7FFFFFFD);
int const Random::modulus = (0x7FFFFFFF);
int const Random::multiplier = 69621);
int const Random::quotient = 30845);
int const Random::remainder = 23902);

int Random::prevInt (0);
Random* Random::instance_ = 0;

Random* Random::instance() {
    if(!instance_) instance_ = new Random();
    return instance_;
}

Random::Random() {
    time_t theTime;
    time(&theTime);
    int mySeed (theTime);
    prevInt = mySeed;
}

int Random::getMaxInt() const {
    return maxInt;
}

double Random::getMaxAngle() const {
    return static_cast<double>(getMaxInt()) /
           static_cast<double>(getMaxInt() + 1);
}

double Random::getMaxSlice() const {
    return static_cast<double>(getMaxInt() + 1) /
           static_cast<double>(getMaxInt() + 2);
}

double Random::getMinSlice() const {
    return 1.0 / static_cast<double>(getMaxInt() + 1);
}
B.13.3 Functions

--- functions.hh ---

// Implemented by Keith Duggar

#ifndef FUNCTIONS_HH
#define FUNCTIONS_HH

#include <cmath>
#include <complex>

namespace KHD {

    // Gamma function
    template<typename FloatT> FloatT gammaln(FloatT z) {
        static const FloatT coef[7] = {
            +1.000000000190015e+0,  
            +7.618009172947146e+1,  
            -8.650532032941677e+1, 
            +2.401409824083091e+1, 
            -1.231739572450155e+0, 
            +1.208650973866179e-3,  
            -5.395239384953000e-5  
        };  
        FloatT tmp = z + 5.5;
        tmp -= log(tmp) * (z + 0.5);
        FloatT res = coef[0];
        FloatT zcpy = z;
        for(int i = 1; i <= 6; ++i)
            res += coef[i] / ++zcpy;
        res *= 2.5066282746310005;  // Square root of 2PI
        res /= z;
        res = log(res);
        res -= tmp;
        return res;
    }

    // Integer specialization
    inline double gammaln(int z) {
        return gammaln<double>(z);
    }

    // Digamma function
    template<typename FloatT> FloatT digamma(FloatT z) {
        static const FloatT coef[7] = {
            +1.000000000190015e+0,  
            +7.618009172947146e+1,  
            -8.650532032941677e+1, 
            +2.401409824083091e+1, 
            -1.231739572450155e+0, 
            +1.208650973866179e-3,  
            -5.395239384953000e-5  
        };  
    }

}}

130
-5.395239384953000e-5

--z;

FloatT res;
    res = z + 0.5;
    res /= z + 5.5;
    res += log(z+5.5);
    res -= 1;

FloatT zcpy = z;
FloatT sum0 = coef[0];
for(int i = 1; i <= 6; ++i)
    sum0 += coef[i] / ++zcpy;

zcpy = z;
FloatT suml = 0.0;
for(int i = 1; i <= 6; ++i) {
    ++zcpy;
    suml += coef[i]/(zcpy*zcpy);
}
	suml /= sum0;
    res -= sum1;

return res;
}

//Trigamma function
template<typename FloatT> FloatT trigamma(FloatT z) {
    static const FloatT coef[7] = {
        +1.00000000000190015e+0 ,
        +7.61809172947146e+1 ,
        -8.650532032941677e+1 ,
        +2.401409824083091e+0 ,
        -1.231739572450155e+0 ,
        +1.208650973866179e-3 ,
        -5.395239384953000e-5
    };

    --z;

    FloatT res;
    res = z + 10.5;
    res /= z + 5.5;
    res /= z + 5.5;

    FloatT zcpy = z;
    FloatT sum0 = coef[0];
    for(int i = 1; i <= 6; ++i)
        sum0 += coef[i] / ++zcpy;

    zcpy = z;
    FloatT suml = 0.0;
    for(int i = 1; i <= 6; ++i) {
        ++zcpy;
        suml += coef[i]/(zcpy*zcpy);
    }

    return res;
}
zcpy = z;
FloatT sum2 = 0.0;
for(int i = 1; i <= 6; ++i) {
    ++zcpy;
    sum2 += coef[i]*(zcpy*zcpy*zcpy);
}

sum1 *= sum1;
sum1 /= sum0*sum0;
sum2 *= 2/sum0;
sum2 -= sum1;
res += sum2;

return res;
}

//Computes the roots of a cubic polynomial of the form:
// a0 + a1 x + a2 x^2 + a3 = 0

template<typename FloatT>
class CubicRoots {

public:

    //Typedefs and local result type
    typedef std::complex<FloatT> Complex;
    class Result {
    public:
        Complex & operator[](int);
        Complex const& operator[](int) const;
    private:
        Complex root1;
        Complex root2;
        Complex root3;
    }

    static Result compute(FloatT const& a0, FloatT const& a1, FloatT const& a2);
};

template<typename FloatT> typename CubicRoots<FloatT>::Result
CubicRoots<FloatT>::compute
(FloatT const& a0, FloatT const& a1, FloatT const& a2) {

    FloatT q = a1/3.0 - a2*a2/9.0;
    FloatT r = (a1*a2 - 3.0*a0)/6.0 - a2*a2*a2/27.0;
    Complex t = q*q*q+r*r;
    t = sqrt(t);
    Complex s1 = r + t;
    s1 = pow(s1, 1.0/3.0);
    Complex s2 = r - t;
    s2 = pow(s2, 1.0/3.0);
    t = Complex(0.0, 1.0) * sqrt(3.0)/2.0;

    Result result;
    result[0] = s1+s2-a2/3.0;
result[1] = -(s1+s2)/2.0 - a2/3.0 + t*(s1-s2);
result[2] = -(s1+s2)/2.0 - a2/3.0 - t*(s1-s2);
return result;
}

template< typename FloatT > inline typename CubicRoots<FloatT>::Complex& CubicRoots<FloatT>::Result::operator[](int i) {
    switch(i) {
        case 0: return root1;
        case 1: return root2;
        case 2: return root3;
    }
    throw(0);
}

template< typename FloatT > inline typename CubicRoots<FloatT>::Complex const& CubicRoots<FloatT>::Result::operator[](int i) const {
    switch(i) {
        case 0: return root1;
        case 1: return root2;
        case 2: return root3;
    }
    throw(0);
}

#endif
B.13.4 Base Types

---basetypes.hh---

// author: Keith H. Duggar
// email : duggar@mit.edu
// purpose: This provides some simple redefinitions of builtin types.
//          The idea is to provide uniform (though not pretty) names
//          for types of specific size and polarity

#ifndef basetypes_hh
#define basetypes_hh

namespace KHD {
    typedef unsigned char byte;
    typedef signed char sint01;
    typedef unsigned char uint01;
    typedef signed short sint02;
    typedef unsigned short uint02;
    typedef signed long sint04;
    typedef unsigned long uint04;
}
#endif
B.13.5 Constants

--- constants.hh ---

// Implemented by Keith Duggar
#ifndef constants.hh
#define constants.hh

namespace KHD {
    namespace Math {
        double const PI = 3.14159265358979;
        double const EE = 2.71828182845905;
        double const GOLD = 1.61803398874990;
        double const SQRT2 = 1.41421356237310;
        double const SQRTPI = 1.77245385090552;
    }
}
#endif
B.13.6 Tuple

---TUPLE.hh---

//Implemented by Keith Duggar
#ifndef TUPLE_HH
#define TUPLE_HH

template<int i, typename T> class Tuple;

template<typename T> class Tuple<2, T> {

public:
    typedef T Value;
    Tuple() {}
    Tuple(T const& _field0, T const& _field1) : field0(_field0), field1(_field1) {} 
    T& operator[](int i) { if(i == 0) return field0; return field1; }
    T const& operator[](int i) const { if(i == 0) return field0; return field1; }

private:
    T field0;
    T field1;
};

template<typename T> class Tuple<3, T> {

public:
    typedef T Value;
    Tuple() {}
    Tuple(T const& _field0, T const& _field1, T const& _field2) :
    field0(_field0), field1(_field1), field2(_field2) {}
    T& operator[](int i) { if(i == 0) return field0; }
    T const& operator[](int i) const { if(i == 0) return field0; }

private:
    T field0;
    T field1;
    T field2;
};

#endif
Appendix C

Sample simulation parameter specification file (simulate.xml)

<?xml version="1.0"?>
<!--SPECIFICATION -->
<!-- Gene Expression Array Simulator Input Parameters-->  
<!-- cDNA Structure -->
<data>  
<!-- DNA Array Structure -->
<param name="numGenes" type="Const"> 6624 </param>
<param name="numSpots" type="Const"> 1 </param>
<param name="numPixels" type="Const"> 113 </param>
<param name="meanNumDNAs" type="Distribution">  
  <Distribution type="LogNormal">  
    <mean> 5.56 </mean>  
    <scale> 0.8 </scale>  
  </Distribution>  
</param>  
<!-- Hybridization -->
<param name="fractionR" type="Const"> 0.5 </param>
<param name="probBind" type="Const"> 1 </param>
<!-- Photonics -->
<param name="cascadeMeanR" type="Distribution">  
  <Distribution type="LogNormal">  
    <mean> 2.0 </mean>  
    <scale> 0.85 </scale>  
  </Distribution>  
</param>
<param name="cascadeMeanRatio" type="Const"> 1 </param>
<param name="probFilterRtoR" type="Const"> 1 </param>
<param name="probFilterGtoG" type="Const"> 1 </param>
<param name="probDetectR" type="Const"> 1 </param>
<param name="probDetectG" type="Const"> 1 </param>
<param name="gainR" type="Const"> 2 </param>
<param name="gainG" type="Const"> 2 </param>  
</data>
Appendix D

Running Xerces on Athena

In order to use the Xerces XML parser, we first have to build the Xerces system on our available platforms. We have the option of installing binary drop or download the source code and build Xerces ourselves. Working with the source code makes more sense since we only have to download the source code (xerces-c-src1_6_0.tar.gz) once, and are able to install Xerces on various platforms. Besides, it is helpful to study the open source for our own implementation. Binary drop is platform dependent; therefore we need to download various binary drops if we decide to change platforms. The tar file for the source code resides in the 'Xerces' directory under the home directory. In order to use the code, first we need to unzip the tar file to the 'xerces-src-c1_6_0' directory, so that the source code has the path '~/Xerces/xerces-src-c1_6_0'.

The basic instructions for building Xerces can be found on the web (http://xml.apache.org/xerces-c/build-winunix.html#UNIX). However, the actual build procedure is a little different on our Athena File Systems. It consists of three steps: setting build environment variables, building Xerces libraries, and building the samples.

Setting up the environment variables is the step that changes the most from web instructions. First we need to change the PATH environment variable. On our Athena File Systems, we need to change athena_path instead of PATH, because during system initialization, the .environment file runs before the .path file runs. Just changing the PATH variable will not have the effect we want, since the PATH will be reset to default when the .path file runs. In the .environment file, do

\[
\text{set athena\_path = ( $athena\_path ~/Xerces/xerces-c-src1_6_0/bin)}
\]

In addition, we also have to set the environment variable XERCESROOT and the library search path so that the program can pick up the shared libraries at runtime.
setenv XERCESROOT "$HOME/Xerces/xerces-c-src1_6_0"
if ("$?LD_LIBRARY_PATH") then
    setenv LD_LIBRARY_PATH "$XERCESCROOT/lib:$LD_LIBRARY_PATH"
else
    setenv LD_LIBRARY_PATH "$XERCESCROOT/lib"
endif

We need to use the “if ... then...” statement because LD_LIBRARY_PATH variable is not defined on our Linux machines, while LD_LIBRARY_PATH is defined on our Solaris machines.

After all the environment variables are set, the next step is to build the Xerces-C++ libraries. Building the libraries requires GNU tools such as `autoconf` and `gmake`, these are free software available on the web. In addition, Apache has provided a wrapper script (`runConfigure`) that exports a few more environment variables. All we need to tell the script is what our compiler is and what options we are going to use inside our build, and the script does everything for us. However, there is a small glitch in the `runConfigure` file provided by Apache – its executable bit is not set. So to before running `runConfigure`, we have to first set permission of `runConfigure` to be executable by doing

```
% chmod a+x
```

, then

```
% ./runConfigure -plinux -cgcc -xg++
```

The `runConfigure` file invokes `configure` to create Makefiles in the individual subdirectories. In addition to that, it sets a few environment variables to correctly configure our compiler and compiler flags. After all the Makefiles are created, one can do the actual build by

```
% gmake
```
Building the samples is the same as building the Xerces-C++ libraries. We only need to call `runConfigure`, then `gmake`. This creates the object files in each sample directory and the executables in the 'xerces-src-c1_6_0/bin' directory.
Bibliography


Gamma, E., Helm, R., Johnson, R., Vlissides, J. Design Patterns - Elements of Reusable Object-Oriented Software. Addison-Wesley (1995).

