# Random Variate Generation for Bayesian Nonparametric Reliability Analysis

**by**

**Patrick John Munson, B.S.**

**REPORT**

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

**Master of Science in Engineering**

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2005

# Report Documentation Page

*Form Approved*
*OMB No. 0704-0188*



| 1. REPORT DATE | 2. REPORT TYPE | 3. DATES COVERED |
|---|---|---|
| **02 JUN 2005** | **N/A** | **-** |

| 4. TITLE AND SUBTITLE | 5a. CONTRACT NUMBER |
|---|---|
| **Random Variate Generation for Bayesian Nonparametric Reliability Analysis** | 5b. GRANT NUMBER |
| | 5c. PROGRAM ELEMENT NUMBER |
| 6. AUTHOR(S) | 5d. PROJECT NUMBER |
| | 5e. TASK NUMBER |
| | 5f. WORK UNIT NUMBER |
| 7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **The University of Texas at Austin** | 8. PERFORMING ORGANIZATION REPORT NUMBER |
| 9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) | 10. SPONSOR/MONITOR'S ACRONYM(S) |
| | 11. SPONSOR/MONITOR'S REPORT NUMBER(S) |

12. DISTRIBUTION/AVAILABILITY STATEMENT
**Approved for public release, distribution unlimited**

13. SUPPLEMENTARY NOTES
**The original document contains color images.**

14. ABSTRACT

15. SUBJECT TERMS

| 16. SECURITY CLASSIFICATION OF: | | | 17. LIMITATION OF ABSTRACT | 18. NUMBER OF PAGES | 19a. NAME OF RESPONSIBLE PERSON |
|---|---|---|---|---|---|
| a. REPORT **unclassified** | b. ABSTRACT **unclassified** | c. THIS PAGE **unclassified** | **UU** | **70** | |

**Random Variate Generation for Bayesian Nonparametric Reliability Analysis**

APPROVED BY

SUPERVISING COMMITTEE:

Elmira Popova, Supervisor

Paul Damien, Supervisor

# Random Variate Generation for Bayesian Nonparametric Reliability Analysis

Patrick John Munson, M.S.E.
The University of Texas at Austin, 2005

Supervisors: Elmira Popova
Paul Damien

Simulation modeling requires accurate input analysis to ensure validity of the study. Hence, the mantra "garbage in = garbage out." Much of the research and simulation code that has been written to date has been focused on traditional parametric methods. Here we investigate Bayesian nonparametric methods for input modeling and reliability analysis. Bayesian nonparametric methods have been shown in many cases to produce better predictive models. Also, for use in a Bayesian setting, we have written C++ classes for random variate generation. These contain functions for standard and truncated distributions as well as functions for statistical data handling. Although we have written the code for Bayesian algorithms, the functions can be used anywhere a good source of random variates is needed. Included is a detailed description of class implementation and usage along with complete source code.

# Table of Contents

# List of Figures

# Chapter 1

# Introduction

Many real-world systems of interest to operations research analysts have inherently stochastic components. These include, but are certainly not limited to, financial markets, mechanical systems with uncertain lifetimes, and queuing systems. To gain a better understanding of the underlying mechanisms of such systems, analysts build statistical simulation models. The focus of this report is on input analysis for statistical simulation modeling. Input analysis is extremely important to simulation modeling as wrong input distributions will result in misleading simulation output and incorrect decisions.

In simulation input analysis we must first develop accurate probability models to represent real-world random phenomena, then simulate draws from these predictive distributions for input to the simulation model. To date, the primary focus for input analysis has been on traditional, parametric statistical methods [see Law and Kelton, 2000]. Here we discuss an alternative method for input modeling in a reliability setting, Bayesian nonparametrics, and a C++ class library that we've developed primarily for Bayesian simulation.

We've chosen to investigate Bayesian nonparametric methods for input modeling in an attempt to make data models more robust and, as a result,

make more accurate predictions. With a Bayesian nonparametric model we do not force in any undue assumptions about the characteristics of the underlying distribution, yet we can inject prior notions about the location and spread of the data. Bayesian models also have a nice feature in that we can "learn from experience". We can continually update the model with new data and therefore capture more and more information about the underlying mechanism.

The focus here is on a Bayesian nonparametric technique to simulate cumulative hazard functions using Beta stochastic processes [see Hjort, 1990]. Cumulative hazard functions can then be used either for direct analysis or to recover a predictive distribution. We give an overview of Beta processes and, more generally, the stochastic process approach to Bayesian nonparametrics. We also include a practical description of how Beta processes can be simulated.

Simulation in a Bayesian setting requires accurate generators that can produce variates from standard and truncated probability distributions for use in specialized algorithms. We have therefore written a C++ class library that will generate random variates for use in a variety of contexts. We needed a source that is reliable, easy to use, and flexible. There are many commercial packages available that can produce quality random variates, but these are generally not flexible enough for use in new (Bayesian) research projects. Our C++ class was primarily developed for Bayesian statistical modeling but can be used anywhere simulated random variates are needed.

Included is a complete description of the C++ class usage and implementation. We give specific examples and discuss each function in detail with

general explanations and references to the underlying algorithm. The code has been thoroughly tested and debugged, but we assume that the user has a basic understanding of probability theory. Given the usage description, references, and attached source code, one should be able to incorporate these functions into almost any simulation study.

The report is organized as follows: in chapter 2 we discuss random variate generation and the implementation and usage of our C++ class library. Chapter 3 includes the theory and computational algorithms associated with some Bayesian nonparametric methods, and chapter 4 concludes the report with our summary and possible areas for future research. The appendices include source code from our C++ class library.

# Chapter 2

# Pseudo-Random Variate Generation in C++

When performing Bayesian statistical analysis, nonparametric, parametric, and mixed, one frequently must simulate draws from standard probability distributions, predictive distributions, and/or stochastic processes. The need arises in building simulation models and while using sampling techniques when non-conjugate set-ups render posterior distributions intractable. In either case one needs a good source of uniform and non-uniform pseudo-random variates from known distributions. We use the term "pseudo" here as true randomness is impossible from a computer; generated data that appears random is sufficient when trying to glean information in a stochastic setting.

We wanted a random variate (RV) source that is reusable, adaptable, and reliable so that it can be used in future specialized Bayesian simulation projects. Thus we built a class library in C++ to handle all random number generation, some statistical processing of data, and specialized Bayesian functions to include variate generation from truncated densities using *auxiliary variable* techniques [see Damien et al., 1999]. C++ was chosen for its versatility and object oriented nature. For example, all statistical functions, as well as multivariate generator functions, can be passed vectors from the

C++ standard library "vector" class. Using vectors in lieu of arrays results in code that is faster/more efficient and does not require the user to pass vector sizes, which makes functions more general. In this chapter we discuss the implementation and usage of functions from our random variate class library.

Section 2.1 contains a general description of how to implement the class and its member functions. This includes detailed sample code that should make the usage easier to understand. Sections 2.2 and 2.3 discuss the class member functions associated with uniform and non-uniform variate generation respectively. In section 2.4 we discuss algorithms and associated functions for sampling from truncated densities. We conclude the chapter in section 2.5 with the statistical data handling functions.

## 2.1 Class Implementation and Usage

Our random variate class library consists of four files: RanV.h, RanV.cpp, mrand_seeds.h, and BayesRV.cpp. In the usual manner RanV.h contains the class declarations, and RanV.cpp and BayesRV.cpp contain the associated source code. The header file, mrand_seeds.h, contains the 10,000 six vector seeds needed for the random number generators. To implement, place the header files (*.h) in an appropriate directory so that the compiler can locate them, and compile and link RanV.cpp and BayesRV.cpp with the .cpp file containing function `main()`.

Once RanV.h is included in the main .cpp file with the statement `#include ‘‘RanV.h’’`, merely instantiate objects of type RanV or BayesRV

(with the statements `RanV object()` or `BayesRV object()`) to use the classes' member functions. To set a particular seed value, instantiate objects with input in the interval [1,10000]. The objects' constructors will set the appropriate seed value. At any other point the seed can be set to a new value with a call to `object.setSeed(int)`[1]. Seed will default to 12 otherwise. C++ source code from RanV.cpp is attached in appendix A and code from BayesRV.cpp in appendix B. Header file RanV.h is in appendix C.

The following is complete sample C++ program that shows exactly how classes RanV and BayesRV are implemented. The program instantiates objects of type `RanV` and `BayesRV`, calls functions from each class, and outputs results to a computer monitor.

```
1  //Samp1.cpp, Sample program that implements
2  //classes RanV and BayesRV
3
4  #include<iostream>
5  #include<vector>
6
7  using namespace std;
8
9  #include "RanV.h"
10
11 int main()
12 {
13         //declare variables:
14         double U, N, Chi, Gam, TRe;
15
```

[1]We assume "object" to be the general object instantiated with the statements `RanV object` or `BayesRV object`.

```
16          RanV rv(55);                //instantiate RanV object
17
18          BayesRV bay(155);        //instantiate BayesRV object
19
20          //call class member functions
21
22          U = rv.mrand();                //uniform(0,1) variate
23          N = rv.stdnorm();        //normal(0,1) variate
24          Chi = rv.ChiSquare(8);        //ChiSquare n = 8 d.f.
25          Gam = rv.gamma(2.4);        //Gamma(a), a = 2.4
26
27          //return truncated exponential with rate = 3.0
28          //and truncation limits [1,3]
29          TRe = bay.TRexpn(3.0,1,3);
30
31          //output results
32          cout << U << endl << N << endl << Chi << endl << Gam
33                   << endl << endl
34                   << TRe << endl;
35
36          return 0;        //successful termination of program
37  }//end function main
```

In lines 4-5 we include header files from the standard template library. We need "iostream" since we are using `cout` and `endl`, and "vector" is necessary since class RanV employs vectors. Line 7 is necessary since `cout` and `endl` are both from the "std" namespace. Line 9 contains the `#include` statement for the RanV header file, and line 11 is the official start of the program.

The program itself is well commented (lines beginning with "//"). The only items of note are in lines 16 and 18. These are where our RanV and BayesRV objects are instantiated. Note here that the seeds for the uniform

generator for each object are set to 55 and 155 respectively. At any subsequent point during the program, one could set the seeds to a new value with `rv.setSeed(int)` or `bay.setSeed2(int)`.

## 2.2 Uniform Variate Functions

All non-uniform random variate (RV) generators in this class library are implemented using various methods (e.g. transformations, acceptance-rejection) involving Uniform(0,1) RVs. It was therefore imperative that we start by developing a good source of uniform variates. "Good" uniform variates are those that are evenly distributed across (0,1) in all dimensions and appear to be independent.

For Uniform(0,1) random numbers we employed a composite Multiple Recursive Generator (MRG) as presented in Law and Kelton [2000] and developed by L'Ecuyer [1999]. The uniform RV source combines 2 MRGs and is defined by:

$$
\begin{aligned}
Z_{1,i} &= (1,403,580\, Z_{1,i-2} - 810,728\, Z_{1,i-3})[mod(2^{32} - 209)] \\
Z_{2,i} &= (527,612\, Z_{2,i-1} - 1,370,589\, Z_{2,i-3})[mod(2^{32} - 22,853)] \\
Y_i &= (Z_{1,i} - Z_{2,i})[mod(2^{32} - 209)] \\
U_i &= \frac{Y_i}{2^{32} - 209}
\end{aligned}
$$

The parameters were chosen carefully by L'Ecuyer [1999], and the generator has period of approximately $2^{191}$ with, according to Law and Kelton [2000], good statistical properties through dimension 32. The resulting variates passed

all statistical tests for uniformity and independence for which we subjected them. The source code includes a header file with 10,000 6-vector seeds spaced $10^{16}$ apart which makes the generator a prime candidate for parallel computing (although parallel processors were not used in this application).

There are four functions associated with Uniform variate generation: `mrand(), unif(), mrandst(), and mrandgt()`. When calling `object.mrand()` no input is required; the seed value is set when ‘‘`object`’’ is instantiated, or with the function `setSeed(stream)` where “stream” is an integer on [1,10000]. The output will be a uniform RV on the interval (0,1). Figure 2.1 shows a histogram of 64,000 variates from `mrand()`[2]. Simply use `object.unif(a, b)` with real valued (a,b) to obtain general uniform RVs on the interval (a,b).

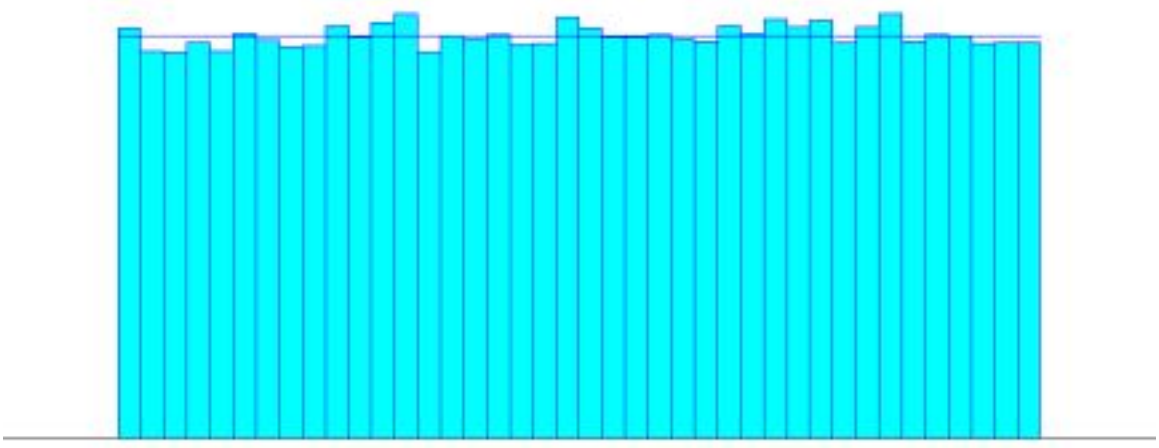

Figure 2.1: Uniform(0,1) random variates

Functions `mrandst()` and `mrandgt()` can be used to set and get re-

[2]The horizontal axis contains partitioned $X$ values, and the vertical axis contains the respective frequencies of the sample.

spectively the *actual* 6-vector seed used in the uniform generator. Call the function `object.mrandst (vector<double> seed, str)` with a vector containing 6 real numbers and an integer on [0,10000] to set 6-vector stream "str" to that contained in vector "seed". To get the latest 6-vector used in the generator, pass an empty vector and the stream number using `object.mrandgt (vector<double> seed, str)`.

## 2.3 Non-Uniform Variate Functions

Given a good source of Uniform(0,1) variates, most common non-uniform RVs are easily generated. Published research in this area is plentiful, robust, and well developed. For an excellent, comprehensive source one can turn to Devroye's *Non-uniform Random Variate Generation* [see Devroye, 1986]. The book is now out of print, but an Adobe pdf version has been posted by the author and can be found at `http://jeff.cs.mcgill.ca/~luc/rnbookindex.html`.

Our class library can be implemented to generate data from the following well-known continuous and discrete distributions: exponential, Weibull, gamma, beta, Dirichlet, normal, lognormal, Chi-square, Bernoulli, binomial, multinomial, and Poisson. The only continuous distribution that is implemented via direct inversion is the exponential. This comes from the result that Exponential($\alpha$) variates can be generated by $X = -\frac{1}{\alpha} ln(U)$ where $U$ is Uniform(0,1). Sums of exponentials can then be used to generate Gamma($\alpha$) where $\alpha \in \mathbb{N}$ and Weibull RVs are simply $X^{\frac{1}{\beta}}$ where $X$ is an exponential RV.

The respective nonuniform probability density and mass functions (pdf's and pmf's) implemented in this package are:

1. Exponential($\lambda$): $f_X(x) = \lambda\, e^{-\lambda\, x},\ \ x > 0$
2. Weibull($\alpha, \lambda$): $f_X(x) = \alpha\, \lambda^{\alpha}\, x^{\alpha-1}\, e^{-(\lambda\, x)^{\alpha}},\ \ x > 0$
3. Gamma($\alpha$): $f_X(x) = \frac{1}{\Gamma(\alpha)}\, x^{\alpha-1}\, e^{-x},\ \ x > 0$
4. Beta($\alpha, \beta$): $f_X(x) = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\,\Gamma(\beta)}\, x^{\alpha-1}\, (1-x)^{\beta-1},\ \ 0 < x < 1$
5. Dirichlet($\alpha_1, \ldots, \alpha_k$): $f_{X_1,\ldots,X_k}(x_1, \ldots, x_k) \propto x_1^{\alpha_1-1} \cdots x_k^{\alpha_k-1},\ \ \sum X_i = 1$
6. Normal($\mu, \sigma^2$): $f_X(x) = \frac{1}{\sigma\,\sqrt{2\,\pi}}\, e^{-\frac{(x-\mu)^2}{2\,\sigma^2}},\ \ -\infty < x < \infty$
7. Lognormal($\mu, \sigma^2$): $f_X(x) = \frac{1}{x\,\sigma\,\sqrt{2\,\pi}}\, e^{-\frac{(\ln(x)-\mu)^2}{2\,\sigma^2}},\ \ x > 0$
8. Chi-square($r$): $f_X(x) = \frac{1}{\Gamma(r/2)\, 2^{(r/2)}}\, x^{(r/2)-1}\, e^{-x/2},\ \ x > 0$
9. Bernoulli($p$): $f_X(x) = p^x\, (1-p)^{1-x},\ \ x = 0, 1$
10. Binomial($n, p$): $f_X(x) = \frac{n!}{x!\,(n-x)!}\, p^x\, (1-p)^{1-x},\ \ x = 0, 1, \ldots, n$
11. Multinomial($p_1, \ldots, p_k$): $f_{X_1,\ldots,X_k}(x_1, \ldots, x_k) \propto p_1^{x_1} \cdots p_k^{x_k},\ \ \sum p_i = 1$
12. Poisson($\lambda$): $f_X(x) = e^{-\lambda}\, \frac{\lambda^x}{x!},\ \ x = 0, 1, \ldots$

To obtain exponential variates, use the function `expn(lambda)`[3] with any *lambda* $> 0$. Weibull variates are obtained with the function `weibull(shape,`

[3]From this point forward, we will assume the reader knows to call a function using "object" and drop the `object.function()` notation.

`scale)` where the *shape* and *scale* parameters are both real valued and greater than 0. Figure 2.2 contains a histogram of 64,000 exponential(2.0) variates and figure 2.3 a plot of Weibull(2.0,4.0) variates[4].

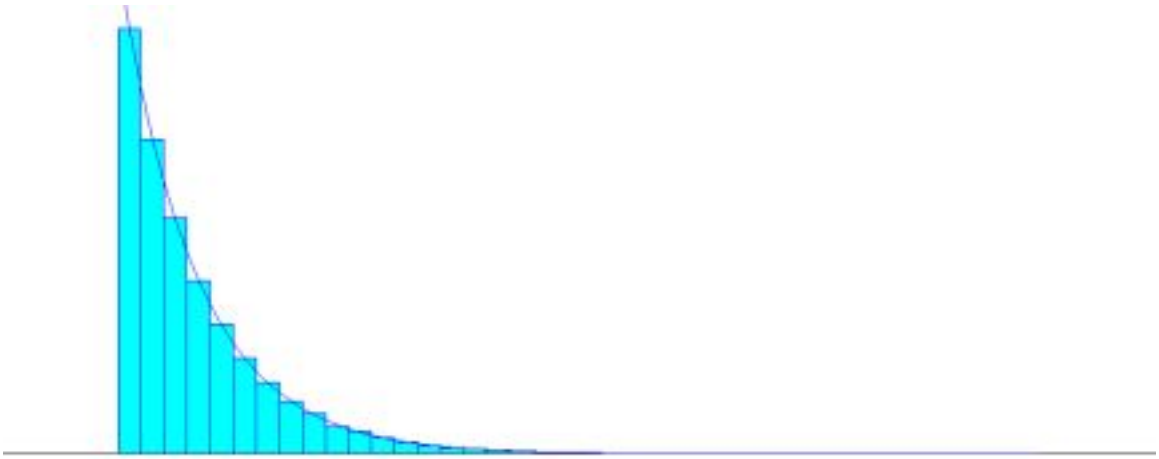

Figure 2.2: Exponential variates: rate = 2.0

Standard normal variates (Normal(0,1)) are generated using the *polar method* [see Law and Kelton, 2000] and are easily converted to the general case, Normal($\mu, \sigma^2$), via $Y = X * \sigma + \mu$. Standard normals can also be used to generate lognormal and chi-square variates through $Y = e^X$ and $Y = \sum X^2$ respectively.

Function `stdnorm()` requires no input and returns a Normal(0,1) random variate. See figure 2.4 for a histogram of generated standard normal variates. Normal($\mu, \sigma^2$) random variates are generated using `norm(mu, sig2)`,

[4]All histogram plots presented in this chapter use 64,000 generated values. Also, horizontal axes contain partitioned $X$ values, and vertical axes contain respective frequencies of the sample.

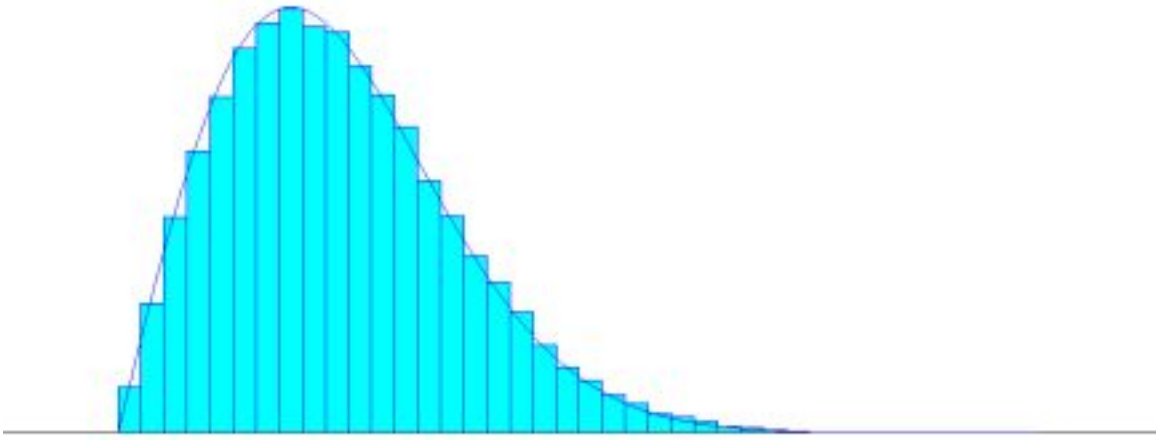

Figure 2.3: Weibull variates: shape = 2.0, scale = 4.0

and lognormal variates are generated using function `logn(mu, sig2)`. Parameters $mu$ and $sig2$ must be real valued with $sig2 > 0$ in both cases and $mu \geq 0$ for lognormal. To obtain a Chi Squared random variate with $n$ degrees of freedom, call function `ChiSquare(n)` with integer valued $n \geq 1$.

For the generalized Gamma($\alpha$), $\alpha \in \mathbb{R}, \alpha > 0$, we use a fast acceptance-rejection routine[5]. For $\alpha > 1$ we use a routine developed by Cheng [1977] and presented in Law and Kelton [2000]. For $\alpha < 1$ we use a routine attributed to Ahrens and Dieter (1974) [see Law and Kelton, 2000]. Gamma($\alpha$) RVs can then be used to obtain Beta($\alpha_1, \alpha_2$) through $Y = \frac{X_1}{X_1+X_2}$ and Dirichlet through $Y = \frac{X_i}{\sum X_i}$. Gamma($\alpha, \beta$) is simply $\beta X$, where X = Gamma($\alpha$). See figure 2.5 for a histogram plot of Gamma(5.4) variates.

[5]Note that in the case of very small $\alpha$, no acceptance-rejection routine will be fast. We cover an algorithm for this special case in section 2.4.

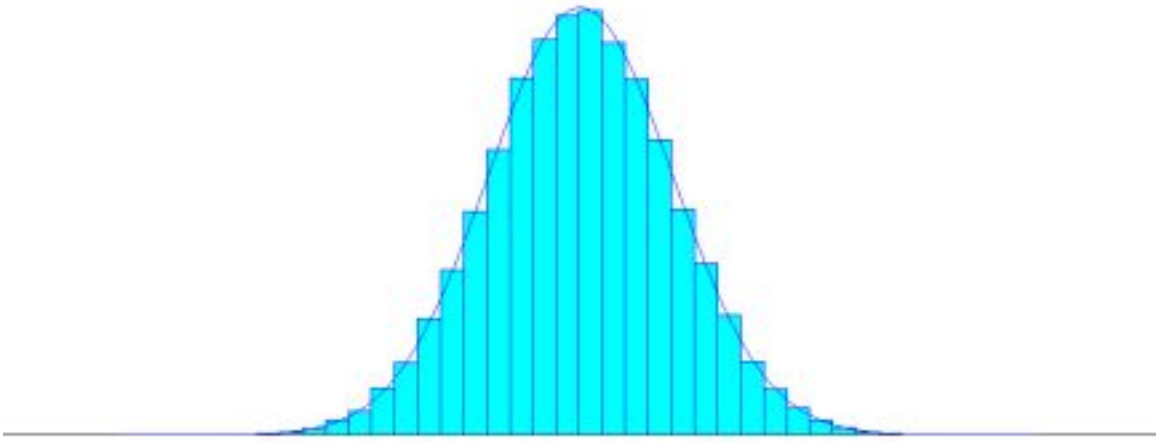

Figure 2.4: Standard Normal Variates: mean = 0, variance = 1

A Gamma($\alpha$) variate is returned with the function `gamma(alpha)` where input *alpha* must be real valued and greater than 0. Again, to obtain a Gamma($\alpha, \beta$) variate, multiply a Gamma($\alpha$) by $\beta$. Beta variates are obtained using function `beta(a1, a2)` with real valued $a1 \geq 0$ and $a2 \geq 0$.[6] Now, the Dirichlet distribution is a multivariate generalization of the Beta distribution. For a vector of realizations from a Dirichlet distribution you must pass a parameter vector and an empty vector for $x$ values with the function `Dirichlet(vector<double> P, vector<double> X)`. The parameters in vector P must sum to 1. After the function is called, vector X will contain the desired random vector.

The discrete distributions that we implemented here are Bernoulli, binomial, multinomial, and Poisson. Bernoulli RVs are simple transformations

[6]If a1 and/or a2 are very small, use the beta function described in section 2.4.

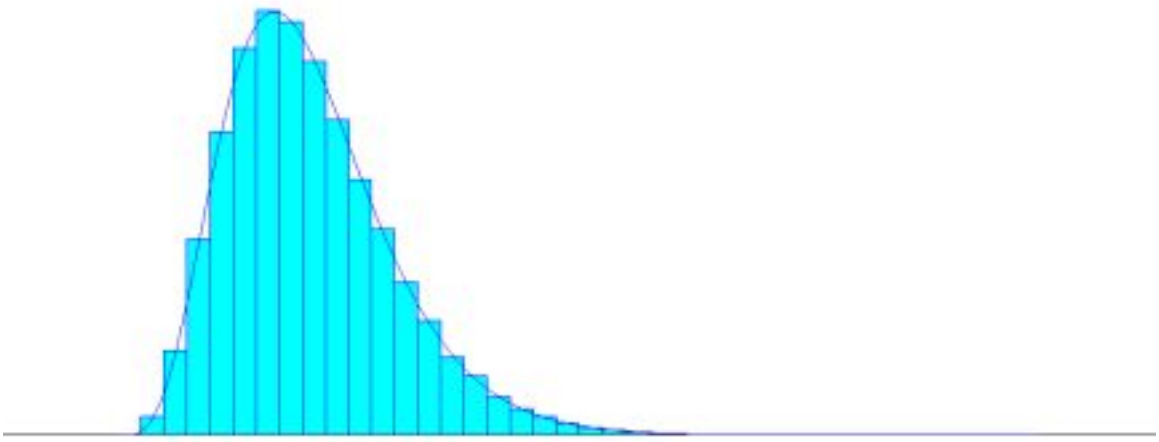

Figure 2.5: Standard Gamma Variates: alpha = 5.4, beta = 1

of uniforms, binomials are sums of Bernoullis, and multinomial random vectors are obtained using binomials. Poisson RVs are a little bit tricky. We employ an algorithm developed by Press et al. [2002] that separates cases into $\lambda < 12.0$ and $\lambda >= 12.0$. For $\lambda < 12.0$ the code directly exploits the Poisson distribution's relationship with the exponential. When $\lambda >= 12.0$ we use an acceptance-rejection routine to control processing time as $\lambda$ gets large. See figure 2.6 for a plot of generated Poisson variates.

Function `bern(p)` is used to generate a Bernoulli variate with parameter $0 \leq p \leq 1$. To generate a Binomial variate call function `binom(n, p)` with integer input $n \geq 2$ and, again, $0 \leq p \leq 1$. Now, the multinomial function is multivariate and also requires two vectors for input: one for parameters such that all sum to 1 and one for return of $x$ values. Calling `multnom(vector<double> P, vector<int> X)` results in the desired random

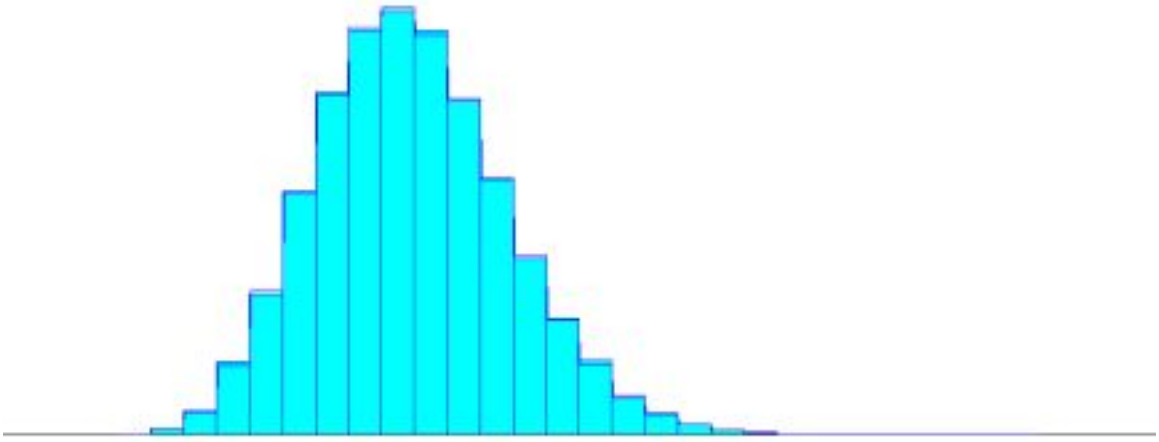

Figure 2.6: Poisson Random Variates: mean = 8.4

vector being stored in vector X. Finally, to obtain Poisson variates (as in figure 2.6), use the function `Poisson(lambda)` with $lambda > 0$.

## 2.4 Sampling from Truncated Densities

In Bayesian simulation one frequently must sample from truncated densities and/or distributions with exceptionally narrow spread. In both cases, using an acceptance-rejection routine is not practical. In fact, if the truncation interval and/or spread is very small, obtaining a sample using acceptance-rejection may be impossible. To sample from such distributions we have implemented in C++ algorithms employing the *auxiliary variable* technique developed by Damien et al. [1999]. In this section we describe three algorithms from Damien and Walker [2001] for sampling truncated Beta, Gamma, and Bivariate Normal densities.

The *auxiliary variable* technique was developed for ease of use in a Gibbs sampler [see Casella and George, 1992]. In general, latent (auxiliary) variables can be strategically added to probability distributions so that the full conditionals are known and easy to sample. The latent variables must be added such that the marginal densities include the desired distribution. In this manner a Gibbs sampler is used with the full conditionals to produce variate(s) from the original distribution.

Introducing one auxiliary variable to the Beta distribution reduces the complicated task of generating truncated Beta variates to merely sampling a Uniform and an exponential function. The truncated Beta distribution is given, up to proportionality, by

$$f_X(x) \propto x^{\alpha-1}\,(1-x)^{\beta-1}\,I\big(x \in (a,b)\big)$$

where $\alpha, \beta > 0$ and $0 \leq a < b \leq 1$ and $I$ is the indicator function. Damien and Walker [2001] introduce latent variable $Y$ such that the joint density is given by

$$f_{X,Y}(x,y) \propto x^{\alpha-1}\,I\big(y < (1-x)^{\beta-1}, x \in (a,b)\big). \tag{2.1}$$

Note that

$$\int_0^{(1-x)^{\beta-1}} x^{\alpha-1}\,I\big(y < (1-x)^{\beta-1}\big)\,dy = x^{\alpha-1}\,(1-x)^{\beta-1},$$

i.e. the marginal density of $X$ is again the Beta distribution.

Given the joint density in (2.1), the full conditional for $Y$ given $X$ is simply Uniform$(0,(1-x)^{\beta-1})$. The full conditional for $X$ given $Y$ depends

on whether $\beta < 1$ or $\beta > 1$; if $\beta = 1$ the Beta distribution can be sampled directly using inversion. So for $\beta > 1$ the full conditional of $X$ is

$$f_{X|Y}(x|y) \propto x^{\alpha-1}\, I\big(x \in \big(a, \min\{b, 1-y^{1/(\beta-1)}\}\big)\big). \tag{2.2}$$

Now when $\beta < 1$ the full conditional of $X$ becomes

$$f_{X|Y}(x|y) \propto x^{\alpha-1}\, I\big(x \in \big(\max\{a, 1-y^{1/(\beta-1)}\}, b\big)\big). \tag{2.3}$$

Both can be sampled directly using the inverse CDF technique with

$$F(x) = \frac{x^\alpha - a^\alpha}{b^\alpha - a^\alpha}\, I\big(x \in (a,b)\big).$$

Now, the Gibbs sampler algorithm for generating a truncated Beta reduces to:

1. INITIALIZE X
2. FOR 5 ITERATIONS
   GENERATE Y AS UNIFORM$(0, (1-x)^{\beta-1})$
   GENERATE X FROM (2.2) OR (2.3)
3. RETURN X

The biggest advantage of this algorithm over acceptance-rejection techniques, if there are no truncation limits, occurs when $\alpha$ and/or $\beta$ are very small. See figure 2.7 for a sample of Beta(0.2,0.2) variates. These are generated using function `TRbeta(alpha, beta, a, b)` from the "BayesRV" class where `a` and

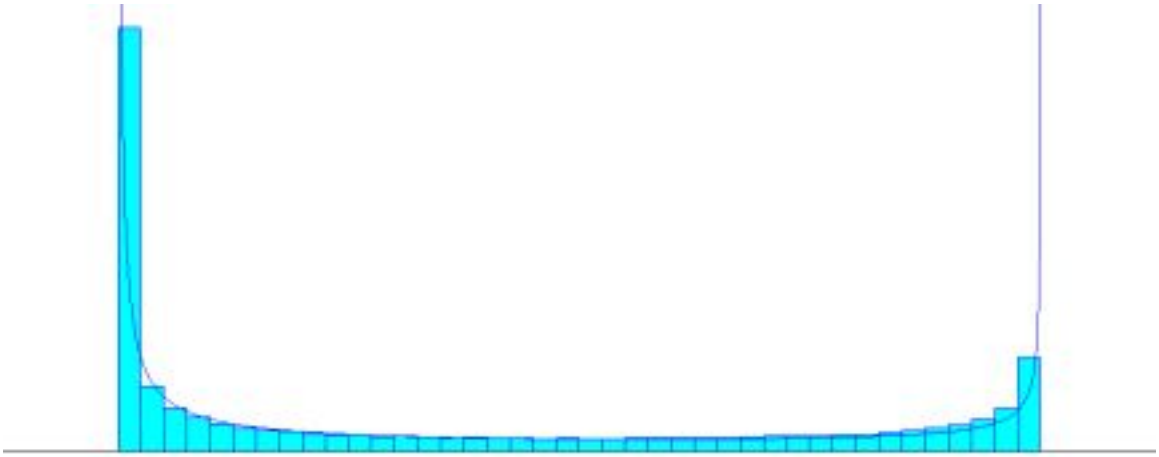

Figure 2.7: Beta Random Variates: alpha = 0.2, beta = 0.2

`b` are the truncation limits. To obtain non truncated variates, merely set `a` = 0 and `b` = 1.

The method for truncated Gamma variates is very similar to that of Beta. The Gibbs sampler algorithm is exactly the same as above but with slightly different conditionals. The truncated standard ($\beta = 1$) Gamma density is given to proportionality by:

$$f_X(x) \propto x^{\alpha-1} \exp(-x)\, I\big(x \in (a,b)\big)$$

where $0 \leq a < b \leq \infty$. Damien and Walker [2001] introduce latent variable $Y$ such that the joint density is given as:

$$f_{X,Y}(x,y) \propto x^{\alpha-1}\, I\big(y < \exp(-x), x \in (a,b)\big). \tag{2.4}$$

Again, note that, integrating with respect to y:

$$\int_0^{\exp(-x)} x^{\alpha-1}\, I\big(y < \exp(-x)\big)\, dy = x^{\alpha-1} \exp(-x).$$

Now, given the joint density in (2.4) we have the full conditional for $Y$ given $X$ as Uniform$(0, \exp(-x))$. The full conditional for $X$ given $Y$ is given as

$$f_{X|Y}(x|y) \propto x^{\alpha-1}\, I\big(x \in \big(a, \min\{b, -log(y)\}\big)\big). \tag{2.5}$$

Equation (2.5) is very similar to that of (2.2) and (2.3) and can be sampled using inversion.

To obtain a truncated Gamma variate call function `TRgamma(alpha, a, b)` from the "BayesRV" class. To obtain non truncated variates simply call `TRgamma()` with `a` $= 0$ and `b` arbitrarily high compared to `alpha`. Again, the advantages over the standard Gamma function occur when `alpha` is very small. See figure 2.8 for a sample of Gamma(0.57) variates.

Like the truncated Beta and Gamma distributions, sampling from a truncated multivariate Normal density can be greatly simplified using the auxiliary variable technique of Damien et al. [1999]. Without this technique the task can be arduous at best. Here we have employed an algorithm from Damien and Walker [2001] to handle the bivariate case. Given a good matrix handling C++ library, this method can easily be extended to larger random vectors.

The truncated multivariate Normal density is given to proportionality by

$$f_{X_1,\ldots,X_p}(x_1, \ldots, x_p) \propto \exp\big(-\frac{1}{2}\,(\mathbf{x}-\mu)'\,\Sigma^{-1}\,(\mathbf{x}-\mu)\big)\, I\big(x \in A\big)$$

where $\mu$ is the mean vector and $\Sigma$ is the covariance matrix. Damien and Walker

[2001] introduce only one latent variable $Y$ and define the joint density as

$$f_{X_1,\ldots,X_p,Y}(x_1,\ldots,x_p,y) \propto \exp\Big(-\frac{y}{2}\Big)\, I\big(y > (\mathbf{x}-\mu)'\,\Sigma^{-1}\,(\mathbf{x}-\mu), x \in A\big)$$

where the full conditional for each $X_i$ is given as

$$f_{X_i|X_{-i},Y}(x_i|x_{-i},y) \propto I(x_i \in A_i).$$

The conditionals are then uniform densities with $A_i = (a_i, b_i) \cap B_i$ where $B_i$ is the set $\{x_i|x_{-i} \ : \ (\mathbf{x}-\mu)'\,\Sigma^{-1}\,(\mathbf{x}-\mu) < y\}$. The bounds for $B_i$ are therefore found by solving a quadratic equation. Also, the full conditional for $Y$ given $X$ is a truncated exponential distribution.

Figure 2.8: Gamma Random Variates: alpha = 0.57, beta = 1

Since the truncated exponential distribution is used frequently in simulation algorithms, other than just sampling truncated multivariate normal, we created the function `TRexpn(lambda, a, b)`. Call this function with real

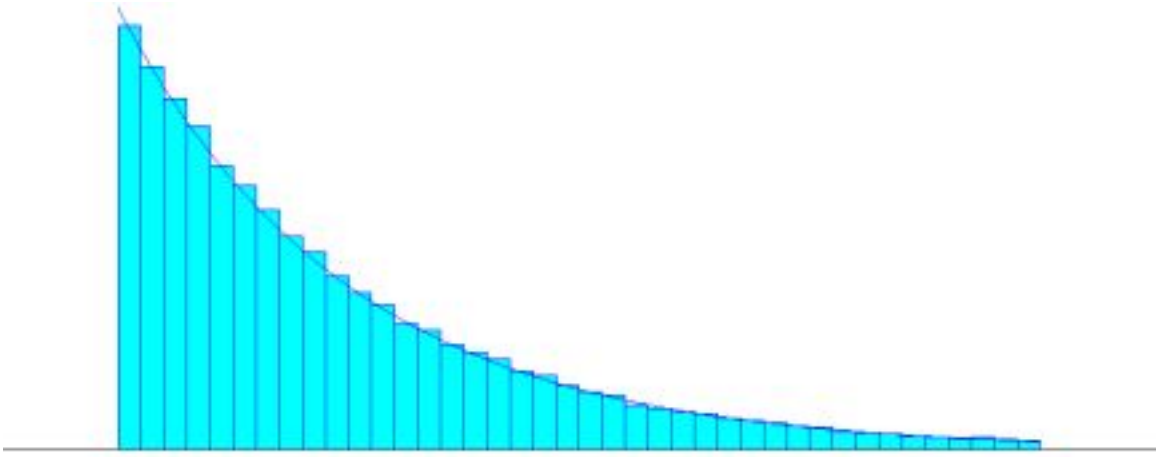

Figure 2.9: Truncated Exponential Random Variates: lambda = 2.0, a = 1.0, b = 3.0

valued `lambda` greater than 0 and $0 \leq$ `a` $<$ `b` $\leq \infty$ to obtain truncated exponential variates as in figure 2.9.

Now, to obtain truncated bivariate Normal variates call the function `TRbiNorm(X, Mu, Sigma, A)`. `X` is an empty `vector<double>` to hold the returned variates, `Mu` is a `vector<double>` containing the means, and `Sigma` is a 2x2 double subscripted array containing the covariance matrix. `A` is again a 2x2 double subscripted array. The first row contains the truncation limits for $X_1$ and the second row contains the truncation limits for $X_2$.

Using function `TRbiNorm()` we reproduce a simulation described in Damien and Walker [2001] of 10,000 samples from a bivariate Normal density truncated on the unit circle centered at (0.5,0.5). The means here are 0 and $\Sigma$ is given with unit variances and covariance = 0.9. Figure 2.10 contains a graph of the produced variates. Notice the truncation limits and the positive

correlation between $X_1$ and $X_2$.

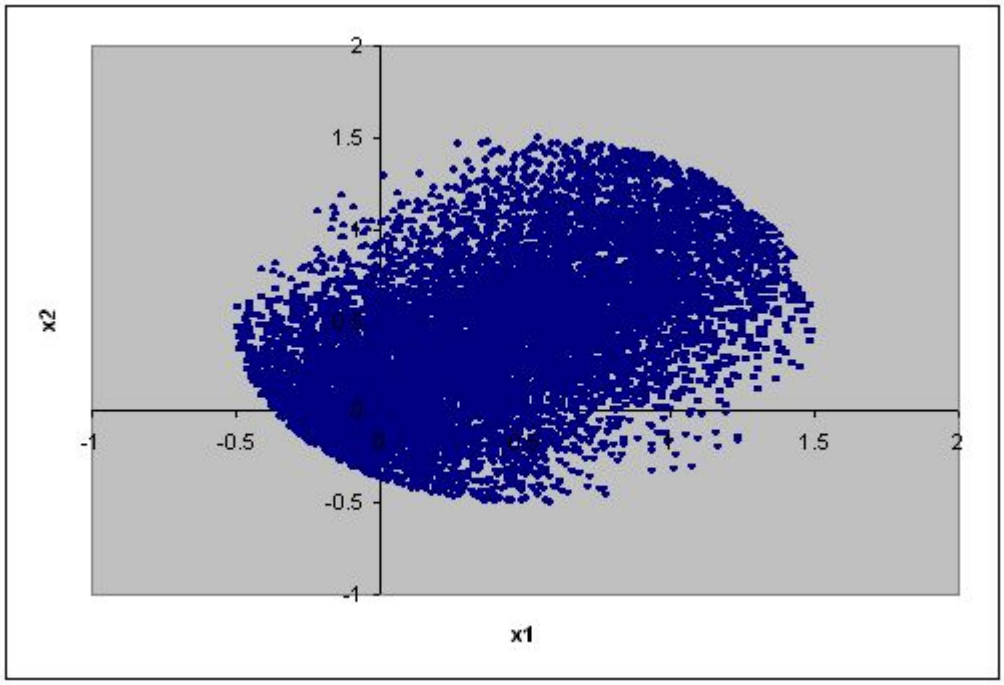


Figure 2.10: 10,000 Truncated Bivariate Normal Variates

## 2.5 Statistical Data Handling in C++

In addition to pseudo-random variate generation, we can use our class library to input vectors of data from sequential, delimited data files and perform empirical statistical analysis. Analysis includes calculating population mean and variance, covariance between two vectors of data, and empirical quartiles. Input is stored in vectors so all data handling functions can be used with internally simulated values as well.

External data files can be delimited in almost any manner to include

spaces, commas, lines, and tabs. To get external data, call the function `getdata(vector<> D, filename)` with an empty vector and filename to include the file extension, i.e. "filename.txt" or "filename.dat". The data file must be in the same directory as the compiled executable. The data will be stored in vector "D".

Data handling functions include `cmom(vector<> D, mean, var)`, `cov(vector<> D1, vector<> D2, cov)`, and `quant(vector<> D, vector<> Q)`. The inputs to `cmom()` are a data vector, `D`, and double valued variables `mean` and `var` to hold the outputs for population mean and population variance respectively. Function `cov()` requires two data vectors, `D1` and `D2`, and double valued `cov` to hold the output sample covariance between the two data vectors.

Finally, function `quant()` requires a data vector, `D`, and a vector of quantiles, `Q`. For output, the data vector is returned numerically sorted, and function `quant()` replaces the percentile values in the quantile vector with the desired data value. For example, `Q` might contain as input {`0.1,0.5,0.9`} to return the 10th, 50th, and 90th percentiles respectively. If the data vector contained standard Normal variates, the quantile vector above would be returned as approximately {`-1.282,0,1.282`}.

# Chapter 3

# Bayesian Nonparametrics

## 3.1 Introduction

The focus of the Bayes portion of this report is on the use of Bayesian nonparametric techniques for reliability (lifetime data) analysis. In general we would like to glean useful information and make accurate predictions using lifetime data collected about an observed set of items. In a *traditional parametric* setting we would make certain assumptions about the family of distributions from which these lifetimes arise and use collected data to estimate the unknown, but fixed, parameter(s) from the chosen distribution family. In *Bayesian parametric* analysis we view the parameter(s) as random and assign prior distribution(s) from which we make inferences about the parameter(s) in question. We then use Bayes theorem to make data conditional updates to the prior distribution(s).

*Traditional nonparametric* techniques [see Siegel and Castellan Jr., 1988] involve not making any assumptions about the family of distributions and using the data to build unknown, yet fixed, distributions from which to make predictions. Now, in parallel with the parametric setting, *Bayesian nonparametric* methods "randomize" the distribution in question and place a prior to

which we can make data conditional updates (posterior). We can then use the posterior to make inferences about the distribution and in turn make predictions about lifetimes in question. A current survey of many popular Bayesian nonparametric techniques with follow-on discussions can be found in Walker et al. [1999].

We've chosen to investigate the use of stochastic process priors in Bayesian nonparametric models. Since Ferguson [see Ferguson, 1973] introduced the Dirichlet process prior in his seminal 1973 paper, the stochastic process approach to Bayesian nonparametrics has become very popular for its stable theoretical grounding and ease of use [see Walker et al., 1999]. Ferguson [1973] showed that the Dirichlet process has large support and is conjugate to the space of cumulative distribution functions (CDFs), i.e. the data conditional posterior process is again a Dirichlet process.

In the stochastic process approach to Bayesian nonparametrics, we define a stochastic process with index set on the sample space, $\Omega$, where specific realizations are cumulative distribution functions. In this setting, if the stochastic process is carefully chosen (as in the Dirichlet process introduced above), updates can be made using observations from the distribution in question that will result in a well defined posterior process. The increments of the posterior process can then be simulated to obtain realizations of the underlying CDF, which can in turn be used to simulate predictive data sets.

The Dirichlet process prior has one drawback in that it leads to the class of discrete distributions almost surely. To overcome this (while handling

censored lifetime data), Doksum [1974], Ferguson [1974], and Ferguson and Phadia [1979] developed a general class of priors called *Neutral to the Right* (NTR) processes of which the Dirichlet is a specific case. They showed that if the random distribution function $F(x), x \in \Omega$ is NTR, then $Z(x) = -log(1 - F(x))$ is a Lévy process where

1. $Z(x)$ has non-negative independent increments,
2. $Z(x)$ is non-decreasing almost surely,
3. $Z(x)$ is right continuous almost surely,
4. $lim_{x \to -\infty}[Z(x)] = 0$ almost surely and
5. $lim_{x \to +\infty}[Z(x)] = \infty$ almost surely.

Ferguson and Phadia [1979] also proved that if $F(x)$ is NTR, then $F(x)$ given a sample $X_1, X_2, \ldots, X_n$, possibly right censored, from $F(x)$ is again NTR and $Z(x)$ is again an independent increments Lévy process. As a result, possibly right censored data can be used to make updates to the selected prior process, then focus can be turned to the Lévy process $Z(x)$ for realizations of the random distribution $F(x)$.

In this chapter we turn our attention to a specific class of Lévy processes developed by Hjort [1990], beta processes, which are used as priors for the space of cumulative hazard functions (CHFs). The distribution function $F(t), t \geq 0$ can be recovered from the CHF, $A(t)$, as $A(t) = -log(1 - F(t))$ in the case of

continuous $F(t)$. Notice the parallel of $A(t)$ to $Z(x)$ above. In section 3.2 we discuss the use of the cumulative hazard function in a Bayesian nonparametric setting. Section 3.3 introduces the theoretical framework behind the beta process, and section 3.4 addresses the computational issues associated with simulating the increments of the beta process.

## 3.2 Random Cumulative Hazard Function

Most of the emphasis in the Bayesian nonparametric literature has been placed on developing priors for the random distribution function (CDF). To that effect the 1970s saw the development of Neutral to the Right (NTR) processes as priors for the CDF, $F(t)$, where $Z(t) = -log(1-F(t))$ is an independent increments Lévy process. Hjort [1990], noticing a natural relationship of hazard functions to $Z(t)$, chose to focus on developing a stochastic process prior not for the CDF, $F(t)$, but for the cumulative hazard function (CHF), $A(t)$, instead. According to Hjort [1990] the CHF is as basic to understanding survival phenomenon as the CDF, and the hazard rate concept is easily generalized to more complicated models.

We've chosen only to address the time continuous CHF model. Let $T$ be a random variable with CDF $F(t) = Pr(T \leq t)$ on $[0, \infty)$ and $F(0) = 0$. The cumulative hazard rate for T is a nonnegative, nondecreasing, right continuous function $A$ on $[0, \infty)$ where

$$dA(s) = A[s, s+ds) = Pr\{T \in [s, s+ds) | T \geq s\} = \frac{dF(s)}{F[s, \infty)}$$

Now Hjort defines for $0 \leq a \leq b < \infty$

$$A[a,b) = \int_{[a,b)} \frac{dF(s)}{F[s,\infty)}$$

and requires

$$F[a,b) = \int_{[a,b)} F[s,\infty)\, dA(s).$$

We have that $A(t)$ is the value of the function $A$ at the point $t$ such that $F(t)$ is recoverable from $A(t)$ by using *product integrals*[Gill and Johansen, 1990]:

$$F(t) = 1 - \prod_{s\in[0,t]} \{1 - dA(s)\}, \qquad t \geq 0. \tag{3.1}$$

If $F$ is continuous this reduces to the familiar

$$A(t) = -log(1 - F(t)).$$

We now have the framework for developing Hjort's prior distributions for $A$. Let $\mathbf{F}$ be the set of all CDFs $F$ on $[0,\infty)$ having $F(0) = 0$ and let $\mathbf{B}$ be the set of all nondecreasing, right continuous functions $B$ on $[0,\infty)$ having $B(0) = 0$. Define the space of cumulative hazard rates as

$$\mathbf{A} = \{A \in \mathbf{B} \text{ for which (3.1) leads to an } F \in \mathbf{F}\}.$$

To place probability distribution on $\mathbf{A}$, it is natural to consider the non-negative, nondecreasing Lévy processes, $Z(t)$, defined above where $Z(t) = -log(1 - F(t))$. However, not every Lévy process can be used as a prior for $A$. To address this limitation Hjort constructs the beta process, whose sample paths lie in $\mathbf{A}$ almost surely.

## 3.3 The Beta Process

In this section we outline the key definitions and theorems associated with the beta process. These include complicated Lévy representations that define the increments of the process. The Lévy representation is a formula denoting the moment generating function (MGF) or characteristic function (CF) of the process. We will show in the next section how these Lévy formulas can be used in practice. Many stochastic processes, including most of the general processes developed as Bayes priors for random CDFs and CHFs, have no closed-form representation defining the distribution of the increments–only an MGF or CF. Hjort's beta process is no exception. We should point out that this is merely an overview of the beta process theory. Further development and proofs for all theorems can be found in Hjort [1990].

We now define the beta process. Note that the goal of this definition is to split the process into a *continuous* and a *discrete* component. This lends itself to computational practicality as discussed in the next section.

**Definition 3.3.1.** Let $A_0$ be a CHF with a finite number of jumps taking place at $t_1, t_2, \ldots$, and let $c(\cdot)$ be a piecewise continuous, non-negative function on $[0, \infty)$. A beta process with parameters $c(\cdot), A(\cdot)$ is a Lévy process denoted by

$$A \sim beta\{c(\cdot), A_0(\cdot)\}, \tag{3.2}$$

if $A$ has Lévy representation

$$E[e^{-\theta A(t)}] = \left\{ \prod_{j:t_j \leq t} E[e^{-\theta S_j}] \right\} exp\left\{ -\int_0^\infty (1 - e^{-\theta s})\, dL_t(s) \right\}, \tag{3.3}$$

with

$$S_j = A\{t_j\} \sim beta\{c(t_j)\, A_0\{t_j\}, c(t_j)\,(1 - A_0\{t_j\})\}, \tag{3.4}$$

and

$$dL_t(s) = \int_0^t c(z)\, s^{-1}\,(1-s)^{c(z)-1}\, dA_{0,c}(z)ds \tag{3.5}$$

for $t \geq 0$ and $0 < s < 1$, in which $A_{0,c}(t) = A_0(t) - \sum_{t_j \leq t} A_0\{t_j\}$ is $A_0(t)$ with its jumps removed.

We can restate the last part of the definition as

$$A(t) = \sum_{t_j \leq t} S_j + A_c(t),$$

where the jumps, $S_j$, are independent beta random variables from (3.4), and $A_c(t)$ is $beta\{c(\cdot), A_{0,c}(\cdot)\}$ as defined in (3.2).

Assume that $A_0$ is the prior guess at the cumulative hazard, and $c(s)$ can be interpreted as the number at risk at time $s$ in an imagined prior sample with hazard rate corresponding to $A_0$. The prior guess at the cumulative hazard can be chosen from a parametric model such as the Weibull, i.e. $A_0(t) = \lambda^\alpha\, t^\alpha$ where $\alpha$ is an assumed prior *shape* parameter and $\lambda$ is an assumed prior *scale* parameter.

To define a Bayes posterior as developed by Hjort for the random CHF $A(t)$, let $X_1, X_2, \ldots, X_n$ be a random sample from $F(t)$ with CHF $A \in \mathbf{A}$ (as defined above). Assume that $(T_1, \delta_1), \ldots, (T_n, \delta_n)$ are observed where $T_i = \min(X_i, c_i)$, $\delta_i = I\{X_i \leq c_i\}$ and $c_1, \ldots, c_n$ are censoring times. Define the

counting process $N$ and the left-continuous at-risk process Y by

$$N(t) = \sum_{i=1}^{n} I\{T_i \leq t \ \& \ \delta_i = 1\}, \ Y(t) = \sum_{t=1}^{n} I\{T_i \geq t\}, \tag{3.6}$$

where $I$ is the indicator function. Assume that the censoring times are either fixed or independent of the lifetimes $X_i$. Also note that $dN(t) = N(t)$ is the number of observed $X_i$'s at a particular time $t$. The posterior process given the data will then be

**Theorem 3.3.1.** *(Hjort's corollary 4.1) Let* $A \sim beta\{c(\cdot), A_0(\cdot)\}$ *as in (3.2). Then*

$$A \,|\, (T_1, \delta_1), \ldots, (T_n, \delta_n) \sim beta\left\{c(\cdot) + Y(\cdot), \int_0^{(\cdot)} \frac{c(s)\, dA_0(s) + dN(s)}{c(s) + Y(s)}\right\} \tag{3.7}$$

The posterior process contains fixed points of discontinuity (at each of the observed times, censor and survival) even if the prior does not.

## 3.4 Computational Issues

Simulating sample paths from stochastic processes with no closed form representation for their increments can be quite tedious. The stochastic processes described earlier in this chapter can be separated into a *continuous* component and a *jump* component. The *jump* components are straightforward in most cases, as they are usually drawn from standard probability distributions. The *continuous* component is much more complicated in general. Damien et al. [1995] gives a general method for sampling from such distributions (denoted

*infinitely divisible*) and Damien et al. [1996] apply this method to Hjort's beta process.

Sampling from posterior beta processes can be simplified, though, by approximating the *continuous* component with standard beta distributions. This is a result of the fact that the accompanying Lévy measures $L_t(\cdot)$ associated with the beta process as defined in (3.2) are concentrated on the interval (0,1). The *jump* component is already given as a beta random variable (see equation (3.4)).

To simulate a beta process let $H(t)$ denote a random cumulative hazard function and partition the time axis into times $t = 0, \ldots, T$ (a finer grid results in a smoother curve). We have that $H(t) = \sum_{t_j \leq t} S_j + A_c(t)$ where $S_j$ are *jumps* at time $t_j$ and $A_c(t)$ is the *continuous* component. Assume that the prior process is absolutely continuous, i.e. $\sum_{t_j \leq t} S_j = 0$ for all $t$. For the continuous component we assume $A_c \sim Beta(a(t), b(t))$, where $a = c(t)$ and $b = A_0(t)$ (the prior guess). Note that "Beta" here denotes the beta *distribution* not the beta *process*.

Take $c(t) = k$, where $k$ is an integer, say, 1 or 2 and $A_0(t) = \alpha t$. This means the underlying failure rate is linear, with mean $\frac{1}{\alpha}$. Choose $\alpha = 0.1$ for example. Now, to sample the beta process PRIOR, at each time interval sample from the appropriate Beta (a(t),b(t)) defined above. Note that the $A_0$ parameter will change if the time intervals are of unequal lengths. You can also make it more general by setting $c(t) = \lambda t$, where $\lambda$ is a real number.

Given data, the process will have *jumps* at each failure and censor time $t_j$. Each jump $S_j$ will have a beta distribution with parameters $a = c(t)\,A_0(t_j) + dN(t)$ and $b = c(t)\,(1 - A_0(t_j) + Y(t) - dN(t))$ where $dN(t)$ is the number of failures by time $t$ and $Y(t)$ is the number still at risk by time $t$ as defined in (3.6).

Now, the $a(t)$ parameter in the beta distribution for the *continuous* component is updated by simply adding the number of at risk observations (right censored) from that interval to $c(t)$. And so $a(t) = c(t) + Y(t)$. Updates to the $b(t)$ parameter of the *continuous* component are slightly more complicated. First define a couple of variables:

1. $dA_0(t) = \alpha$; the differential of $A_0$ given above. Note if you change the choice of $A_0$ above, then the differential will obviously change.

2. $dN(t)$ is the exact observations in the interval of interest.

3. $Numerator = c(s)\,dA_0(s) + dN(s)$

4. $Denominator = c(s) + Y(s)$

For the first time interval from time $t = 0$ to the first point on the grid, say, $t_1$, integrate: $\int_{t=0}^{t_1} \frac{Numerator}{Denominator}$. This integrated quantity is your $b(t)$ parameter in the POSTERIOR beta process in the first interval. To get the $b(t)$ for the second interval integrate from $t_1$ to $t_2$, and so on.

Key point: Note that the $a(t)$ and $b(t)$ parameters will likely change from one time interval to the next. As you can see, with complicated choices for $c$ and $A_0$, the updates can get nasty because the integral can become awful.

Using these parameters, simulate jump components and a continuous component at each time $t_i$ in the partition of the time axis. The final simulated value for $H(t)$ will now be $\sum_{t_j \leq t} S_j + A_c(t)$. Selecting the time axis partition to coincide with the EXACT and RIGHT CENSORED data points will simplify things. To obtain a smoother estimate of the hazard function include more grid points. An estimate of the CDF $F(t)$ can now be recovered using $F(t_j) = 1 - [(1 - F(t_{j-1}) * (1 - A_c(t_j)) * (1 - S(t_j))]$ as derived from the *product integrals* in (3.1).

# Chapter 4

# Summary and Areas for Continued Research

Using Bayesian nonparametrics for simulation input modeling has exceptional potential. Until recently such methods were impractical due to limited computing power and relatively immature computational methods. There now exists multitudes of new published research in the area, and computing power is virtually unlimited. We have given an overview of Bayesian nonparametric methods with specific focus in hazard rate modeling for reliability analysis. We have also written and presented two C++ classes that can be used as a foundation for almost any Bayesian simulation project.

Computer programs inherently are evolving entities. Any program or algorithm has potential for improvement. The C++ classes presented in chapter 2 are no exception. They have been thoroughly debugged and employ reliable, proven algorithms but are still a work in progress. Also, they can be tailored to fit almost any project where random variates are needed. The classes are a foundation and are meant to fill the void between published methodologies and actual working simulation programs.

In addition, the Bayesian theory presented in chapter 3 presents a starting point for future research. The methods can be applied to a "real world"

reliability problem and compared to existing procedures. Section 3.4 presents a usable technique with examples for simulating Beta processes, but the algorithm was not actually coded. The examples should be tailored to a specific problem since prior knowledge will be different in every case. The Bayesian nonparametric method presented here could be easily implemented in an existing reliability study where failure events are volatile and difficult to model. Censored data is also easily managed in this context.

For simulation input modeling, the hazard rates sampled from the Bayesian nonparametric model can be used to recover a discrete cumulative distribution function (CDF) as described in section 3.4. This CDF can, in turn, be used to sample failure times for input to a simulation. Techniques for sampling from a discrete CDF are well defined and can be found in Law and Kelton [2000]. To obtain a smoother curve with more possible failure times, simply create a finer grid for sampling the hazard rates.

# Appendices

# Appendix A

# Source Code: RanV.cpp

The following is verbatim source code from RanV.cpp as described in Chapter 2.

```
//RanV.cpp, Patrick J. Munson, 1 Apr 05
//Member function definitions for class RanV
//generates uniform and non-uniform pseudo-random variates
//and performs some statistical data handling

#include <iostream>
#include <cmath>
#include <vector>
#include <algorithm>
#include <fstream>
#include "RanV.h"
#include "mrand_seeds.h"

using namespace std;

/**************************************************************************
All RV functions employ uniform RVs from combined MRG as presented by
L'Ecuyer (1999).  10,000 streams are supported, with seed vectors spaced
10^16 apart.  Class RanV object must be initialized with a seed value in
the interval [1,10000].  If not seed defaulted to 12.
**************************************************************************/
RanV::RanV( int seed )
{
        setSeed( seed );
}//end RanV constructor

void RanV::setSeed( int s )
{
        stream = ( s >= 1 && s <= 10000 ) ? s : 12;
}//end function setSeed
```

```
//UNIFORM RVs
/**************************************************************************
Function mrand():  Returns uniform(0,1).
**************************************************************************/
double RanV::mrand()
{
        //variables
        const double m1=4294967087.0, m2=4294944443.0;
        const double norm=1.0/(m1+1);
        int k;
        double p;
        double s10=drng[stream][0], s11=drng[stream][1], s12=drng[stream][2];
        double s20=drng[stream][3], s21=drng[stream][4], s22=drng[stream][5];

        //calculate next "random" number
        p = 1403580.0 * s11 - 810728.0 * s10;
        k = (int)(p/m1);
        p -= k*m1;  //p (mod m1)
        if (p < 0.0) p += m1;
        s10 = s11;  s11 = s12;  s12 = p;

        p = 527612.0 * s22 - 1370589.0 * s20;
        k = (int)(p/m2);
        p -= k*m2;  //p (mod m2)
        if (p < 0.0) p += m2;
        s20 = s21;  s21 = s22;  s22 = p;

        //update stream
        drng[stream][0] = s10;  drng[stream][1] = s11;  drng[stream][2] = s12;
        drng[stream][3] = s20;  drng[stream][4] = s21;  drng[stream][5] = s22;

        if (s12 <= s22) return ((s12 - s22 + m1) * norm);
        else return ((s12 - s22) * norm);
}//end function mrand()

/**************************************************************************
Function unif():  Returns uniform(a,b).
**************************************************************************/
double RanV::unif( double a, double b )
{
        double U;
        U = mrand();

        return U*(b-a) + a;
```

```
}//end function unif()

/*************************************************************************
Function mrandst():  Sets seed vector "stream" to desired 6-vector.
*************************************************************************/
void RanV::mrandst( vector<double> &seed, int str )
{
        for ( int i=0; i<6; i++ )
                drng[str][i] = seed[i];
}//end function mrandst()

/*************************************************************************
Function mrandgt():  Returns the most current 6-vector of integers.
*************************************************************************/
void RanV::mrandgt( vector<double> &seed, int str )
{
        seed.clear();
        for ( int i=0; i<6; i++ )
                seed.push_back( drng[str][i] );
}//end function mrandgt()
//END UNIFORM RVs


//NONUNIFORM RVs
/*****************************************************************
Function "stdnorm()":  Returns a N(0,1) random variate using the
mrand() function and the polar method.
*****************************************************************/
double RanV::stdnorm()
{
        //variables
        static int flag=0;
        static double X2;
        double W, Y, V1, V2;

        //avoid duplicating effort using a flag, utilize both variates
        if ( flag == 0 ) {
                do {
                        V1 = 2.0*mrand()-1.0;
                        V2 = 2.0*mrand()-1.0;
                        W = V1*V1 + V2*V2;
                } while ( W > 1.0 || W == 0.0 );

                Y = sqrt( (-2.0*log(W))/W );
                X2 = V2*Y;
```

```
                flag = 1;
                return V1*Y;
        } else {  //we have an extra variate handy--return it
                flag = 0;
                return X2;
        }//end if/else
}//end function stdnorm()

/*******************************************************
Function norm():  Returns N(mu, sig^2), using stdnorm()
*******************************************************/
double RanV::norm( double mu, double sig2 )
{
        double n0, nm;

        n0 = stdnorm();
        nm = sqrt(sig2)*n0 + mu;

        return nm;
}//end function norm()

/*********************************************************************
Function logn():  Returns lognormal using norm() and a transformation.
*********************************************************************/
double RanV::logn( double mu, double sig2 )
{
        double n1, Nmu, Nsig2, lg;

        Nmu = log( mu*mu/(sqrt(mu*mu+sig2)) );
        Nsig2 = log( 1 + sig2/(mu*mu) );

        n1 = norm(Nmu,Nsig2);
        lg = exp(n1);

        return lg;
}//end function logn()

/*********************************************************************
Function expn():  Returns exponential(lamda) random variate using mrand()
and the inversion method.
*********************************************************************/
double RanV::expn( double lambda )
{
        double U, ed;
```

```
        U = mrand();
        ed = (-1.0/lambda)*log(U);

        return ed;
}//end function expn()

/*************************************************************************
Function weibull():  Returns weibull(a,b) random variate using expn(a)
and a transformation.
*************************************************************************/
double RanV::weibull( double shape, double scale )
{
        double a, w;

        a = pow( scale,shape );

        w = pow( expn(a),(1.0/shape) );

        return w;
}//end function weibull()

/*************************************************************************
Function bern():  Returns bernoulli using mrand().
*************************************************************************/
int RanV::bern( double p )
{
        double u;
        u = mrand();

        if ( u <= p ) return(1);
        else return(0);
}//end function bern()

/*************************************************************************
Function binom():  Returns binomial R.V. using bern().
*************************************************************************/
int RanV::binom( int n, double p )
{
        int sum=0;

        for ( int i=0; i<n; i++ )
                sum += bern(p);

        return sum;
}//end function binom()
```

```
/**************************************************************************
Function multnom():  Returns multinomial R. vector using binom().  Takes
advantage of conditional Xi given others is binomial.
**************************************************************************/
void RanV::multnom( vector<double> &mp, vector<int> &Xi )
{
        //initialize variables and vectors
        int m, r, j=1;
        m = (int)mp[0];
        r = mp.size() - 1;
        Xi.clear();
        for ( int i=0; i<r; i++ )
                Xi.push_back(0);

        double q=1.0;

        do{
                Xi[j-1] = binom(m,mp[j]/q);
                m -= Xi[j-1];
                q -= mp[j];
                j++;
        }while( m != 0 );
}//end function multnom()

/**************************************************************************
Function ChiSquare():  Returns ChiSquare R.V. using stdnorm().
**************************************************************************/
double RanV::ChiSquare( int df )
{
        double yLocal, sum=0.0;

        for ( int i=0; i<df; i++ ){
                yLocal = stdnorm();
                sum += yLocal*yLocal;
        }//end for

        return sum;
}//end function ChiSquare()

/**************************************************************************
Function gamma():  Returns gamma(a) R.V. using GB algorithm (Cheng 1977)
if a > 1.  If a = 1 then reduces to exponential.  If a < 1 then uses
rejection algorithm due to Ahrens and Dieter(1974).
**************************************************************************/
```

```
double RanV::gamma( double alpha )
{
        if ( alpha < 1.0 )
                return gam1(alpha);
        else if ( alpha == 1.0 )
                return expn(alpha);
        else
                return gam2(alpha);
}//end function gamma()

//PRIVATE FUNCTION
double RanV::gam1( double alpha1 )
{
        //See Law and Kelton for description of algorithm
        double b, P, U1, U2, Y;
        double Gam1=-1.0;

        b = (exp(1.0) + alpha1) / exp(1.0);

        do {
                U1 = mrand();
                P = b * U1;

                if ( P > 1.0 ) {
                        Y = -log((b - P)/alpha1);
                        U2 = mrand();
                        if ( U2 <= pow(Y, alpha1-1.0) ) {
                                Gam1 = Y;
                                break;
                        }//end if
                }else {
                        Y = pow(P, 1.0/alpha1);
                        U2 = mrand();
                        if ( U2 <= exp(-Y) )
                                Gam1 = Y;
                }//end if/else
        }while ( Gam1 < 0.0 );

        return Gam1;
}//end function gam1()

//PRIVATE FUNCTION
double RanV::gam2( double alpha2 )
{
        //See Law and Kelton for description of algorithm
```

```
        double a,b,q,theta,d,u1,u2;
        double V,Y,Z,W, Gam2=-1.0;

        a = 1.0/( sqrt(2.0*alpha2-1.0) );
        b = alpha2 - log(4.0);
        q = alpha2 + 1.0/a;
        theta = 4.5;
        d = 1.0 + log(theta);

        if ( alpha2 == 0.0 )
                Gam2 = alpha2;

        do{
                u1 = mrand();
                u2 = mrand();
                V = a*log(u1/(1.0-u1));
                Y = alpha2*exp(V);
                Z = u1*u1*u2;
                W = b + q*V - Y;
                if ( W + d - theta*Z >= 0.0 ){
                        Gam2 = Y;
                        break;
                }//end if
                if ( W >= log(Z) ){
                        Gam2 = Y;
                }//end if
        }while ( Gam2 < 0.0 );

        return Gam2;
}//end function gam2()

/*************************************************************************
Function beta():  Returns beta(a1,a2) R.V. using gamma().
*************************************************************************/
double RanV::beta( double a1, double a2 )
{
        double g1, g2;
        g1 = gamma(a1);
        g2 = gamma(a2);

        return( g1/(g1+g2) );
}//end function beta()

/*************************************************************************
Function Dirichlet():  Returns Dirichlet R.Vector using gamma().
```

```
*************************************************************************/
void RanV::Dirichlet( vector<double> &Dp, vector<double> &XiD )
{
        int k;
        k = Dp.size();

        vector<double> yLocal;
        yLocal.clear();
        XiD.clear();
        for ( int i=0; i<k; i++ ){
                XiD.push_back(0.0);
                yLocal.push_back(0.0);
        }//end for

        double sum=0.0;

        for ( int i=0; i<k; i++ ){
                yLocal[i] = gamma(Dp[i]);
                sum += yLocal[i];
        }//end for

        for ( int i=0; i<k; i++ )
                XiD[i] = yLocal[i]/sum;
}//end function Dirichlet()

/*************************************************************************
Function Poisson():  Returns Poisson RV.  For small lamda (<=12) exploits
relationship with exponential.  For lamda > 12, uses code as developed in
"Numerical Recipes", Press, et al.
*************************************************************************/
int RanV::Poisson( double lambda )
{
        const double PI = 3.141592653589793238;
        static double sq, alxm, g, oldl=(-1.0);
        //oldl is a flag for whether lamda has changed since last call
        int em;
        double t, tm, y;

        if ( lambda < 12.0 ) {                          //use direct method
                if ( lambda != oldl ) {
                        oldl = lambda;
                        g = exp( -lambda );        //if lambda is new, compute exponential
                }//end if
                em = -1;
                t = 1.0;
```

```
                do {
                        ++em;
                        t *= mrand();
                }while ( t > g );
        }else {
                if ( lambda != oldl ) {
                        oldl = lambda;
                        sq = sqrt( 2.0*lambda );
                        alxm = log( lambda );
                        g = lambda*alxm - gammln(lambda+1.0);
                }//end if

                do {
                        do {
                                y = tan( PI*mrand() );
                                tm = sq*y + lambda;
                        }while ( tm < 0.0 );

                        em = (int)floor(tm);
                        t = 0.9*(1.0+y*y)*exp( em*alxm - gammln(em+1.0) - g );
                }while ( mrand() > t );
        }//end else

        return em;
}//end function Poisson()

//PRIVATE FUNCTION
//Returns the natural log of the Gamma function of xx for xx>0
double RanV::gammln( const double xx )
{
        double x, y, tmp, ser;
        static const double cof[6] = {76.18009172947146, -86.50532032941677,
                24.01409824083091, -1.231739572450155, 0.001208650973866179,
                -0.000005395239384953};

        y = x = xx;
        tmp = x + 5.5;
        tmp -= (x+0.5)*log(tmp);
        ser = 1.000000000190015;
        for ( int j=0; j<6; j++ )
                ser += cof[j]/++y;

        return -tmp + log( 2.5066282746310005*ser/x );
}//end function gammln()
```

```
//DATA HANDLING FUNCTIONS
/*************************************************************************
Fuction getdata():  Retrieves data from file and stores in referenced
vector. Data files must be sequential and line, space, or tab delimited.
*************************************************************************/
void RanV::getdata( vector<double> &data, char* filename )
{
        //open input file
        ifstream inData( filename, ios::in );

        //exit program if ifstream could not open file
        if ( !inData ) {
                cerr << "Data file could not be opened" << endl;
                exit( 1 );
        }//end if

        //get data
        data.clear();
        double dvalue;

        while ( inData >> dvalue )
                data.push_back( dvalue );

        //close file
        inData.close();
}//end function getdata()

/*************************************************************************
Fuction cmom():  Calculates 1st and 2nd central moments/population
mean and variance of vector of data.
*************************************************************************/
void RanV::cmom( vector<double> &d, double &mean, double &var )
{
        //initialize variables
        int n = d.size();
        mean = 0.0;
        var = 0.0;

        //calculate population mean and variance
        for ( int i=0; i<n; i++ )
                mean += d[i];

        mean /= n;
```

```
        for ( int i=0; i<n; i++ )
                var += (d[i]-mean)*(d[i]-mean);

        var /= (n-1);
}//end function cmom()

/*************************************************************************
Fuction cov():  Calculates covariance for 2 vectors of data.
*************************************************************************/
void RanV::cov( vector<double> &d1, vector<double> &d2, double &cvar )
{
        //initialize variables
        int n1=d1.size(), n2=d2.size(), n;
        n = min(n1,n2);

        double mean1=0.0, mean2=0.0;

        //calculate means for vectors
        for ( int i=0; i<n; i++ ){
                mean1 += d1[i];
                mean2 += d2[i];
        }//end for

        mean1 /= n;
        mean2 /= n;

        //calculate covariance
        for ( int i=0; i<n; i++ )
                cvar += (d1[i]-mean1)*(d2[i]-mean2);

        cvar /= (n-1);
}//end function cov()

/*************************************************************************
Fuction quant():  Calculates specified quantiles for vector of data.
Supply data vector and vector with specified quantiles in (0,1).  Returns
sorted data vector; quantile values in place of passed quantiles.
*************************************************************************/
void RanV::quant( vector<double> &dat, vector<double> &q )
{
        //variables
        int sd = dat.size();
        int sq = q.size();
        int loc;
```

```
        //find quantiles
        sort( dat.begin(), dat.end() );
        for ( int i=0; i<sq; i++ ){
                loc = (int)ceil( q[i]*sd );
                q[i] = dat[loc-1];
        }//end for
}//end function quant()
```

# Appendix B

# Source Code: BayesRV.cpp

The following is verbatim source code from BayesRV.cpp as described in chapters 2 and 3.

```
//BayesRV.cpp, Patrick J. Munson, 25 Apr 05
//Member function definitions for class BayesRV
//generates specialized RVs for Bayesian analysis

#include <iostream>
#include <cmath>
#include <vector>
#include <algorithm>
#include "RanV.h"

using namespace std;

//constructor
BayesRV::BayesRV( int seed )
{
        setSeed2( seed );
}//end RanV constructor

//assigns seed value to stream2 for use in RanV objects
void BayesRV::setSeed2( int s2 )
{
        stream2 = ( s2 >= 1 && s2 <= 10000 ) ? s2 : 12;
}//end function setSeed



//PUBLIC FUNCTIONS
/*************************************************************************
Function TRexpn():  Returns truncated exponential(a) random variate using
truncated uniform and the inversion method.
```

```
*************************************************************************/
double BayesRV::TRexpn( double lambda, double a, double b )
{
        //variables for truncated uniform
        double TU, Ua, Ub;

        RanV r(stream2);   //RanV object for uniform variates

        //Generate Uniform with inverted truncation limits
        //We can remove 3 subtraction operations by inverting the limits
        //and subtracting from 1
        Ua = exp(-lambda*b);
        Ub = exp(-lambda*a);
        TU = r.unif(Ua,Ub);

        //return truncated exponential
        return( (-1.0/lambda) * log(TU) );
}//end function TRexpn()

/*************************************************************************
Function TRbeta():  Returns truncated Beta(a,b) random variate using
auxilliary variable technique and Gibbs sampler.
*************************************************************************/
double BayesRV::TRbeta( double alpha, double beta, double a, double b )
{
        //variables
        double X, Y;       //X given Y; Y is latent variable given X
        double xCL;        //truncation limit for conditional X

        //initialize X
        X = alpha / (alpha+beta);

        RanV r(stream2);   //RanV object for uniform variates

        //different cases for beta=1, beta<1, and beta>1
        if ( beta == 1.0 ) {
                //simply return f(x) = a*x^(a-1)
                X = TBx(alpha,a,b);
        }else if ( beta < 1.0 ) {
                //run Gibbs sampler for 5 iterations
                for ( int i=0; i<5; i++ ) {
                        //obtain Y given X
                        Y = r.unif( 0.0,(pow( (1.0-X),(beta-1.0) )) );
                        //now obtain X given Y
                        xCL = 1.0 - pow( Y,(1.0/(beta-1.0)) );
```

```
                        xCL = max( a,xCL );
                        X = TBx(alpha,xCL,b);
                }//end for
        }else if ( beta > 1.0 ) {
                //run Gibbs sampler for 5 iterations
                for ( int i=0; i<5; i++ ) {
                        //obtain Y given X
                        Y = r.unif( 0.0,(pow( (1.0-X),(beta-1.0) )) );
                        //now obtain X given Y
                        xCL = 1.0 - pow( Y,(1.0/(beta-1.0)) );
                        xCL = min( b,xCL );
                        X = TBx(alpha,a,xCL);
                }//end for
        }//end if else

        return X;
}//end function TRBeta()

/*************************************************************************
Function TRgamma():  Returns truncated gamma(a) random variate using
auxilliary variable technique and Gibbs sampler.
*************************************************************************/
double BayesRV::TRgamma( double alpha, double a, double b )
{
        //variables
        double X, Y;   //X given Y; Y is latent variable given X
        double bY;     //to store new truncation limit for X given Y

        //initialize X
        X = alpha;

        RanV r(stream2);   //RanV object for uniform variates

        //run Gibbs sampler for 5 iterations
        for ( int i=0; i<5; i++ ) {
                //obtain Y given X
                Y = r.unif( 0.0,(exp( -X )) );

                //now obtain X given Y
                bY = -log( Y );
                bY = min( b,bY );
                //can use the same function (TBx) as in Beta
                X = TBx(alpha,a,bY);
        }//end for
```

```
        return X;
}//end function TRgamma

/**************************************************************************
Function TRbiNorm():  Returns truncated bivariate normal random vector
using auxilliary variable technique and Gibbs sampler.  Pass vector for
X values, mean vector, sigma matrix, and truncation matrix.
**************************************************************************/
void BayesRV::TRbiNorm( double X[], double Mu[], double Sig[][2], double A[][2] )
{
        //variables
        double x1L[2], x2L[2];               //uniform limits, X given Y is U(a,b)
        double B1[2], B2[2];                //B is used to calculate X limits
        double xM1, xM2;                 //stores (X-mu) to simplify calculations
        double Sinv[2][2];                 //for storing Sig inverse
        double Sa, Sb, Sc, Sd;                //values of Sig inverse--simplified eqns
        double Y, Ya, Yb;                 //stores Y and its truncation limits
        Yb = 10000.0;                        //right limit for Y given X is infinity

        //initialize X and xM
        X[0] = Mu[0];
        X[1] = Mu[1];
        xM1 = X[0] - Mu[0];
        xM2 = X[1] - Mu[1];

        /**************************************************************************
        //for Damien(2001) example only
        A[1][0] = 0.5 - sqrt( 1-(X[0]-0.5)*(X[0]-0.5) );
        A[1][1] = 0.5 + sqrt( 1-(X[0]-0.5)*(X[0]-0.5) );
        A[0][0] = 0.5 - sqrt( 1-(X[1]-0.5)*(X[1]-0.5) );
        A[0][1] = 0.5 + sqrt( 1-(X[1]-0.5)*(X[1]-0.5) );
        **************************************************************************/

        //calculate the inverse of the Sig matrix
        inv2x2(Sig,Sinv);
        Sa = Sinv[0][0];
        Sb = Sinv[0][1];
        Sc = Sinv[1][0];
        Sd = Sinv[1][1];

        RanV r(stream2);                        //RanV object for uniform variates

        //run Gibbs sampler for 5 iterations
        for ( int i=0; i<5; i++ ) {
```

```
//first obtain Y given X
Ya = Sa*xM1*xM1 + (Sb+Sc)*xM1*xM2 + Sd*xM2*xM2;
Y = TRexpn(0.5,Ya,Yb);

//for X1
//solution of quadratic eq Y > (X-Mu)'[SigInv](X-Mu)
B1[0] = (1.0/(2.0*Sa)) * ( -xM2*(Sb+Sc) -
        sqrt( 4*Sa*(Y-xM2*xM2*Sd) + xM2*xM2*(Sb+Sc)*(Sb+Sc) ) ) + Mu[0];
B1[1] = (1.0/(2.0*Sa)) * ( -xM2*(Sb+Sc) +
        sqrt( 4*Sa*(Y-xM2*xM2*Sd) + xM2*xM2*(Sb+Sc)*(Sb+Sc) ) ) + Mu[0];
//truncation limits for X given Y
x1L[0] = max( A[0][0],B1[0] );
x1L[1] = min( A[0][1],B1[1] );
//now obtain X1
X[0] = r.unif(x1L[0],x1L[1]);
//recalculate xM1
xM1 = X[0] - Mu[0];

/*********************************************************************
//for Damien(2001) example only
A[1][0] = 0.5 - sqrt( 1-(X[0]-0.5)*(X[0]-0.5) );
A[1][1] = 0.5 + sqrt( 1-(X[0]-0.5)*(X[0]-0.5) );
*********************************************************************/

//for X2
//solution of quadratic eq Y > (X-Mu)'SigInv(X-Mu)
B2[0] = (1.0/(2.0*Sa)) * ( -xM1*(Sb+Sc) -
        sqrt( 4*Sa*(Y-xM1*xM1*Sd) + xM1*xM1*(Sb+Sc)*(Sb+Sc) ) ) + Mu[1];
B2[1] = (1.0/(2.0*Sa)) * ( -xM1*(Sb+Sc) +
        sqrt( 4*Sa*(Y-xM1*xM1*Sd) + xM1*xM1*(Sb+Sc)*(Sb+Sc) ) ) + Mu[1];
//truncation limits for X given Y
x2L[0] = max( A[1][0],B2[0] );
x2L[1] = min( A[1][1],B2[1] );
//now obtain X2
X[1] = r.unif(x2L[0],x2L[1]);
//recalculate xM2
xM2 = X[1] - Mu[1];

/*********************************************************************
//for Damien(2001) example only
A[0][0] = 0.5 - sqrt( 1-(X[1]-0.5)*(X[1]-0.5) );
A[0][1] = 0.5 + sqrt( 1-(X[1]-0.5)*(X[1]-0.5) );
*********************************************************************/

}//end for
```

```
}//end function TRbiNorm



//PRIVATE FUNCTIONS
/*************************************************************************
Function TBx():  Returns X given Y in the truncated Beta and truncated
Gamma examples.  Here F(x) = (x^alpha-a^alpha)/(b^alpha-a^alpha) with
X on the set (a,b).
*************************************************************************/
double BayesRV::TBx( double &alpha, double &a, double &b )
{
        //variables
        double U, tmp;

        RanV r(stream2);  //RanV object

        U = r.mrand();

        tmp = U*(pow(b,alpha)-pow(a,alpha)) + pow(a,alpha);

        //return X, given latent variable Y, for truncated Beta and Gamma
        return( pow(tmp,(1.0/alpha)) );
}//end function TBx()

//calculate and return the inverse of a 2x2 matrix
void BayesRV::inv2x2( double In[][2], double Out[][2] )
{
        double detIn;   //for storing the determinant of In[][]

        detIn = In[0][0]*In[1][1] - In[0][1]*In[1][0];
        //calculate the inverse and return
        Out[0][0] = In[1][1]/detIn;
        Out[0][1] = -In[0][1]/detIn;
        Out[1][0] = -In[1][0]/detIn;
        Out[1][1] = In[0][0]/detIn;
}//end function inv2x2()
```

# Appendix C

# Header File: RanV.h

The following is the verbatim header file RanV.h as described in chapter 2.

```
//RanV.h
//Includes declarations for classes RanV and BayesRV
#ifndef RANV_H
#define RANV_H

//Declaration of class RanV, member functions are defined in RanV.cpp
class RanV {

public:

        //DEFAULT CONSTRUCTOR
        RanV( int = 12 );
        //set seed
        void setSeed( int );

        //UNIFORM
        double mrand();
        double unif( double a, double b );
        void mrandst( std::vector<double> &, int str );
        void mrandgt( std::vector<double> &, int str );

        //NONUNIFORM
        double stdnorm();
        double norm( double mu, double sig2 );
        double logn( double mu, double sig2 );
        double expn( double lambda );
        double weibull( double shape, double scale );
        int bern( double p );
        int binom( int n, double p );
        void multnom( std::vector<double> &mp, std::vector<int> &Xi );
```

```
        double ChiSquare( int df );
        double gamma( double alpha );
        double beta( double a1, double a2 );
        void Dirichlet( std::vector<double> &Dp, std::vector<double> &XiD );
        int Poisson( double lambda );

        //DATA HANDLING
        void getdata( std::vector<double> &, char* );
        void cmom( std::vector<double> &, double &mean, double &var );
        void cov( std::vector<double> &, std::vector<double> &, double &cvar );
        void quant( std::vector<double> &, std::vector<double> & );

private:

        //Seed variable
        int stream;

        //Gamma functions: depends on alpha
        double gam1( double alpha1 );
        double gam2( double alpha2 );

        //for use in Poisson()
        double gammln( const double xx );


};//end class RanV

//Declaration of class BayesRV, member functions are defined in BayesRV.cpp
class BayesRV {

public:

        BayesRV( int = 12 );
        void setSeed2( int );

        double TRexpn( double lambda, double a, double b );
        double TRbeta( double alpha, double beta, double a, double b );
        double TRgamma( double alpha, double a, double b );
        void TRbiNorm( double X[], double Mu[], double Sig[][2], double A[][2] );

private:

        int stream2;

        double TBx( double &, double &, double & );
```

```
        void inv2x2( double [][2], double [][2] );

};//end class BayesRV

#endif
```

# Bibliography

George Casella and Edward I. George. Explaining the Gibbs sampler. *The American Statistician*, 46(3):167–174, 1992.

R.C.H. Cheng. The generation of gamma variables with non-integral shape parameter. *Applied Statistics*, 26:71–75, 1977.

Paul Damien. Some Bayesian nonparametric models. In C. R. Rao and Dipak Dey, editors, *Handbook of Statistics, 25*. Elsevier, In Press.

Paul Damien et al. Approximate random variate generation from infinitely divisible distributions with applications to Bayesian inference. *J. R. Stat. Soc. B*, 57(3):547–563, 1995.

Paul Damien et al. Implementation of Bayesian non-parametric inference based on beta processes. *Scandinavian Journal of Statistics*, 23:27–36, 1996.

Paul Damien et al. Gibbs sampling for Bayesian non-conjugate and hierarchical models by using auxiliary variables. *J. R. Stat. Soc. B*, 61(2):331–344, 1999.

Paul Damien and Stephen G. Walker. Sampling truncated normal, beta, and gamma densities. *Journal of Computational and Graphical Statistics*, 10(2): 206, 2001.

Luc Devroye. *Non-Uniform Random Variate Generation*. Springer-Verlag, New York, 1986.

K. Doksum. Tailfree and neutral random probabilities and their posterior distributions. *The Annals of Probability*, 2:183–201, 1974.

Thomas S. Ferguson. A Bayesian analysis of some nonparametric problems. *The Annals of Statistics*, 1(2):209–230, 1973.

Thomas S. Ferguson. Prior distributions on spaces of probability measures. *The Annals of Statistics*, 2(4):615–629, 1974.

Thomas S. Ferguson and Eswar G. Phadia. Bayesian nonparametric estimation based on censored data. *The Annals of Statistics*, 7(1):163–186, 1979.

Richard D. Gill and Soren Johansen. A survey of product-integration with a view toward application in survival analysis. *The Annals of Statistics*, 18 (4):1501–1555, 1990.

Nils Lid Hjort. Nonparametric Bayes estimators based on beta processes in models for life history data. *The Annals of Statistics*, 18(3):1259–1294, 1990.

Averill M. Law and W. David Kelton. *Simulation Modeling and Analysis*. McGraw-Hill, 3rd edition, 2000.

Pierre L'Ecuyer. Good parameters and implementations for combined multiple recursive random number generators. *Operations Research*, 47(1):159–164, 1999.

William H. Press et al. *Numerical Recipes in C++*. Cambridge, 2nd edition, 2002.

Sidney Siegel and N. John Castellan Jr. *Nonparametric Statistics for the Behavioral Sciences*. McGraw-Hill, 2nd edition, 1988.

Stephen G. Walker et al. Bayesian nonparametric inference for random distributions and related functions. *J. R. Stat. Soc. B*, 61(3):485–527, 1999.

# Vita

Patrick John Munson was born in Summertown, Tennessee. He spent his early childhood in Western North Carolina before moving to Charleston, South Carolina in 1993 and graduated from Hanahan High School in 1996. He received the Bachelor of Science degree in Mathematics from the College of Charleston in December 1999 and was commissioned an Officer in the United States Air Force in 2000. He was stationed in Albuquerque, New Mexico prior to entering the University of Texas at Austin in January 2004.

This report was typeset with $\LaTeX$† by the author.

†$\LaTeX$ is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's $\TeX$ Program.