2015

Acceptance-Rejection Sampling with Hierarchical Models

Christian A. Ayala
Claremont McKenna College

Claremont McKenna College

Acceptance-Rejection Sampling with Hierarchical Models

Submitted to

Professor Mark Huber

And

Dean Nicholas Warner

By

Christian Alessandro Ayala

For

Senior Thesis

Spring 2015

The Twenty-third of April in the Year of our Lord Two Thousand and Fifteen

Abstract

Hierarchical models provide a flexible way of modeling complex behavior. However, the complicated interdependencies among the parameters in the hierarchy make training such models difficult. MCMC methods have been widely used for this purpose, but can often only approximate the necessary distributions. Acceptance-rejection sampling allows for perfect simulation from these often unnormalized distributions by drawing from another distribution over the same support. The efficacy of acceptance-rejection sampling is explored through application to a small dataset which has been widely used for evaluating different methods for inference on hierarchical models. A particular algorithm is developed to draw variates from the posterior distribution. The algorithm is both verified and validated, and then finally applied to the given data, with comparisons to the results of different methods.

1 Introduction

In an empirical model, a sample of some population is used to estimate or "train" the parameters of a model for the population. The most familiar example is probably the simple OLS linear regression model,

$$y_i = \beta_0 + \beta_1 x_i + u_i,\ i = 1, ..., n$$
$$u_i \sim N(0, \sigma^2)$$

where (x_i, y_i) are the values of a dependent and an independent variable over n trials and the parameters are β_0 and β_1. In OLS, the parameters are trained by taking

$$\min_{\beta_0, \beta_1} \sum_{i=1}^{n} (y_i - \beta_0 - \beta_1 x_i)^2$$

which, in this case, can be expressed analytically:

$$\beta_0 = \bar{y} - \beta_1 \bar{x},\ \beta_1 = \frac{cov(\vec{x}, \vec{y})}{var(\vec{x})}.$$

Once the model has been trained, it can be used for a variety of predictive or inferential purposes. For example, given a new value of the independent variable, point and interval estimates can be made of the corresponding value of the dependent variable. Or, the parameters can be subjected to statistical tests to make inferences about the underlying population parameters.

A hierarchical model is simply an empirical model where a hierarchical relationship among the parameters is assumed. More precisely, for data $t_1, t_2, ..., t_m$ and parameters $\theta_1, \theta_2, ..., \theta_n$:

$$\theta_1 \sim D_1$$
$$\theta_i \sim D_i(\theta_1, \theta_2, ..., \theta_{n-1})$$
$$t_i \sim D(\theta_1, \theta_2, ..., \theta_n)$$

As with the linear model, a hierarchical model allows a variety of predictions and inferences to be made. Furthermore, the hierarchical structure allows for greater flexibility of assumptions and more sophisticated inferences. In particular, a hierarchical model is a natural solution when there are a number of related parameters to estimate, but little empirical data for each one individually; the hierarchy allows this similarity to be exploited [6]. However, due to the complicated conditional relationships among the parameters in the model, training them based on a known data set can be much more difficult than it is in the case of OLS. Monte Carlo Markov Chain (MCMC) approaches, including Gibbs samplers, are among the most popular methods for meeting these challenges [11], but these approaches introduce challenges of their own. Generally, such methods are able only to estimate the required distributions, and are not simple to implement for researchers without specialized statistical modeling expertise.

Acceptance-rejection (or AR) sampling is an alternative approach to estimating these models. Through application of Bayes' Theorem, it is usually possible to find an unnormalized distribution function for the parameters in the model based on the data and other parameters in the model. AR provides a way to sample exactly from these unnormalized distributions.

The goal of this paper will be to explore the effectiveness of AR sampling for estimating hierarchical models by applying it to a model of pump failures at the nuclear plant Farley 1 [4] that has already been used widely for testing other approaches [4, 5, 6, 8, 10]. Section 2 will provide an explanation of the model, Section 3 will explain how AR was used to estimate the model, and Section 4 will provide analysis of the results.

2 The Model

The original data for this problem consist of failure counts for ten different pumps over different time intervals. Let S_k denote the number of failures for pump k, and let T_k denote the time of operation in thousands of hours for pump k (the data themselves can be found in Appendix A). The problem is to determine the distribution of failures for each of the ten pumps. This is accomplished through the following hierarchical model, which was formulated in [10]:

$$\beta \to \lambda_k \to S_k$$

First,

$$\alpha = 1.802, \gamma = 0.01, \delta = 1.$$

The values of these parameters were determined using a method of moments argument[10].Then, the parameter β is drawn:

$$\beta \sim Gamma(\gamma, \delta).$$

λ_k for $k = 1, ..., 10$ are then drawn:

$$\lambda_k \sim Gamma(\alpha, \beta).$$

Finally, S_k is drawn, according to λ_k and T_k:

$$S_k \sim Po(\lambda_k T_k).$$

It is useful to have the distribution for each parameter of the model conditioned on the data and the other parameters. For this model, we have the following:

Fact 1.

$$[\lambda_k|\beta, S_k, T_k] \sim Gamma(\alpha + S_k, \beta + T_k)$$
$$[\beta|\vec{\lambda}] \sim Gamma(\gamma + 10\alpha, \sum \lambda_k + \delta)$$

Proof. Explicitly stated, the conditional distribution of λ given β, S and T is

$$f_{\lambda|\beta,S,T}(x) = \frac{(\beta + T)^{\alpha+S}}{\Gamma(\alpha + S)} x^{\alpha+S-1} e^{-(\beta+T)x} \mathbb{1}(x > 0)$$

dropping the subscript k for convenience. $\Gamma(x)$ is an extension of the factorial function to all complex numbers, where $\Gamma(x) = (x-1)!$ for integers, and

$$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t}\, dt.$$

Proceeding with Bayes' theorem and the Law of Total Probability,

$$\begin{aligned} f_{\lambda|\beta,S,T}(x) &= \frac{f_{\lambda|\beta}(x)\mathbb{P}(S|\beta,\lambda=x)}{\int_{-\infty}^{\infty} f_{\lambda|\beta}(u)\mathbb{P}(S|\beta,\lambda=u)du} \\ &= \frac{g(x)}{\int_{-\infty}^{\infty} g(u)du} \end{aligned}$$

where, using the fact that $S! = \Gamma(S+1)$,

$$\begin{aligned} g(x) &= \frac{(xT)^S e^{-xT}}{S!}\frac{\beta^\alpha}{\Gamma(\alpha)}x^{\alpha-1}e^{-\beta x}\mathbb{1}(x>0) \\ &= \frac{\beta^\alpha T^S}{\Gamma(s+1)\Gamma(\alpha)}x^{\alpha+S-1}e^{-(\beta+T)x}\mathbb{1}(x>0). \end{aligned}$$

Then, using the definition of $\Gamma(x)$,

$$\begin{aligned} \int_{-\infty}^{\infty} g(u)\ du &= \frac{\beta^\alpha T^S}{\Gamma(S+1)\Gamma(\alpha)}\int_0^\infty u^{\alpha+S-1}e^{-(\beta+T)u}\ du \\ &= \frac{\beta^\alpha T^S}{\Gamma(S+1)\Gamma(\alpha)}\frac{1}{(\beta+T)^{(\alpha+S)}}\int_0^\infty v^{\alpha+S-1}e^{-v}\ dv \\ &= \frac{\beta^\alpha T^S}{\Gamma(S+1)\Gamma(\alpha)}\frac{\Gamma(\alpha+S)}{(\beta+T)^{(\alpha+S)}}. \end{aligned}$$

Therefore,

$$\begin{aligned} f_{\lambda|\beta,S,T}(x) &= \frac{g(x)}{\int_{-\infty}^{\infty} g(u)du} \\ &= \frac{\beta^\alpha T^S}{\Gamma(s+1)\Gamma(\alpha)}x^{\alpha+S-1}e^{-(\beta+T)x}\mathbb{1}(x>0)\frac{\Gamma(S+1)\Gamma(\alpha)}{\beta^\alpha T^S}\frac{(\beta+T)^{(\alpha+S)}}{\Gamma(\alpha+S)} \\ &= \frac{(\beta+T)^{\alpha+S}}{\Gamma(\alpha+S)}x^{\alpha+S-1}e^{-(\beta+T)x}\mathbb{1}(x>0), \end{aligned}$$

which completes the derivation of the first conditional distribution.

In a similar fashion, the second conditional distribution is equivalent to

$$f_{\beta|\vec{\lambda}}(b) = \frac{(\sum\lambda_k+\delta)^{\gamma+10\alpha}}{\Gamma(\gamma+10\alpha)}b^{\gamma+10\alpha-1}e^{-(\sum\lambda_k+\delta)b}\mathbb{1}(b>0)$$

where $\vec{\lambda}$ denotes the vector of the λ_k.

As before, we have

$$f_{\beta|\vec{\lambda}}(b) = \frac{h(b)}{\int_{-\infty}^{\infty} h(u)\ du}$$

where, by conditional independence of the λ_k given β, we have

$$\begin{aligned} h(b) &= f_{\vec{\lambda}|\beta=b}(b)f_\beta(b) \\ &= \prod_{k=1}^{10}\frac{b^\alpha}{\Gamma(\alpha)}\lambda_k^{\alpha-1}e^{-b\lambda_k}\frac{\delta^\gamma}{\Gamma(\gamma)}b^{\gamma-1}e^{-\delta b}\mathbb{1}(b>0) \\ &= \frac{\delta^\gamma(\prod\lambda_k)^{\alpha-1}}{\Gamma(\alpha)^{10}\Gamma(\gamma)}b^{\gamma+10\alpha-1}e^{-(\sum\lambda_k+\delta)b}\mathbb{1}(b>0). \end{aligned}$$

Integrating h(b), we have

$$\begin{aligned}\int_{-\infty}^{\infty} h(u)\ du &= \frac{\delta^\gamma(\prod \lambda_k)^{\alpha-1}}{\Gamma(\alpha)^{10}\Gamma(\gamma)} \int_0^\infty u^{\gamma+10\alpha-1} e^{-(\sum \lambda_k+\delta)u}\ du \\ &= \frac{\delta^\gamma(\prod \lambda_k)^{\alpha-1}}{\Gamma(\alpha)^{10}\Gamma(\gamma)} \frac{\Gamma(\gamma+10\alpha)}{(\sum \lambda_k+\delta)^{\gamma+10\alpha}}.\end{aligned}$$

Finally,

$$\begin{aligned}f_{\beta|\vec{\lambda}}(b) &= \frac{h(b)}{\int_{-\infty}^{\infty} h(u)\ du} \\ &= \frac{\delta^\gamma(\prod \lambda_k)^{\alpha-1}}{\Gamma(\alpha)^{10}\Gamma(\gamma)} b^{\gamma+10\alpha-1} e^{-(\sum \lambda_k+\delta)b} \mathbb{1}(b>0) \frac{\Gamma(\alpha)^{10}\Gamma(\gamma)}{\delta^\gamma(\prod \lambda_k)^{\alpha-1}} \frac{(\sum \lambda_k+\delta)^{\gamma+10\alpha}}{\Gamma(\gamma+10\alpha)} \\ &= \frac{(\sum \lambda_k+\delta)^{\gamma+10\alpha}}{\Gamma(\gamma+10\alpha)} b^{\gamma+10\alpha-1} e^{-(\sum \lambda_k+\delta)b} \mathbb{1}(b>0)\end{aligned}$$

which completes the derivation of the second conditional distribution. □

The joint distribution of β and $\vec{S}$ is also of vital importance.

Fact 2. *The joint distribution of β and $\vec{S}$ is*

$$f_{\beta,\vec{S}}(b,\vec{s}) = \frac{\delta^\gamma}{\Gamma(\alpha)^{10}\Gamma(\gamma)} b^{10\alpha+\gamma-1} e^{-\delta b} \prod_{k=1}^{10} \frac{T_k^{s_k}\Gamma(\alpha+s_k)}{\Gamma(s_k+1)(T_k+b)^{\alpha+s_k}} \mathbb{1}(b>0)$$

Proof. We can write the full joint distribution as a product of the conditional distributions and integrate over all values of $\vec{\lambda}$ to write the desired joint distribution as

$$\begin{aligned}f_{\beta,\vec{S}}(b,\vec{s}) &= \int_{\vec{x}\in\mathbb{R}^{10}} f_{\beta,\vec{S},\vec{\lambda}}(b,\vec{s},\vec{x})\ d\vec{x} \\ &= \int_{\vec{x}\in\mathbb{R}^{10}} f_\beta(b) f_{\vec{\lambda}|\beta}(\vec{x}) f_{\vec{S}|\beta,\vec{\lambda}}(\vec{s})\ d\vec{x}.\end{aligned}$$

The λ_k are conditionally independent of each other given β and S_k is conditionally independent of β and the rest of $\vec{S}$ given λ_k, so $f_{\vec{\lambda}}$ and $f_{\vec{S}}$ can be expressed as products of f_{λ_k} and f_{S_k}, respectively.

$$\begin{aligned}f_{\beta,\vec{S}}(b,\vec{s}) &= \int_{\vec{x}\in\mathbb{R}^{10}} f_\beta(b) \prod_{k=1}^{10} f_{\lambda_k|\beta}(x_k) \prod_{k=1}^{10} f_{S_k|\lambda_k}(s_k)\ d\vec{x} \\ &= f_\beta(b) \int_{\vec{x}\in\mathbb{R}^{10}} \prod_{k=1}^{10} \frac{b^\alpha x_k^{\alpha-1} e^{-bx_k} (x_k T_k)^{s_k} e^{-x_k T_k}}{\Gamma(\alpha)\Gamma(s_k+1)}\ d\vec{x} \\ &= b^{10\alpha} f_\beta(b) \prod_{k=1}^{10} \frac{T_k^{s_k}}{\Gamma(\alpha)\Gamma(s_k+1)} \int_{\vec{x}\in\mathbb{R}^{10}} \prod_{k=1}^{10} x_k^{\alpha+s_k-1} e^{-(b+T_k)x_k}\ d\vec{x}.\end{aligned}$$

The integral in the above equation is an unnormalized product of Gamma distributed random variables. Therefore, the integral must be equal to the reciprocal of the product of the individual normalizing constants, or

$$\int_{\vec{x}\in\mathbb{R}^{10}} \prod_{k=1}^{10} x_k^{\alpha+s_k-1} e^{-(b+T_k)x_k}\ d\vec{x} = \prod_{k=1}^{10} \frac{\Gamma(\alpha+s_k)}{(T_k+b)^{\alpha+s_k}}.$$

Finally,

$$f_{\beta,\vec{S}}(b,\vec{s}) = \frac{b^{10\alpha}}{\Gamma(\alpha)^{10}} f_\beta(b) \prod_{k=1}^{10} \frac{T_k^{s_k}\Gamma(\alpha+s_k)}{\Gamma(s_k+1)(T_k+b)^{\alpha+s_k}}$$
$$= \frac{\delta^\gamma}{\Gamma(\alpha)^{10}\Gamma(\gamma)} b^{10\alpha+\gamma-1} e^{-\delta b} \prod_{k=1}^{10} \frac{T_k^{s_k}\Gamma(\alpha+s_k)}{\Gamma(s_k+1)(T_k+b)^{\alpha+s_k}} \mathbb{1}(b>0)$$

□

The goal with this model would be to draw β given only $\vec{S}$, and then to use Fact 2 to draw $\vec{\lambda}$ given $\vec{S}$ and β. This would require $f_{\vec{S}}(\vec{s})$, using which we could find $f_{\beta|\vec{S}}(b,\vec{s})$ by

$$f_{\beta|\vec{S}}(b,\vec{s}) = \frac{f_{\beta,\vec{S}}(b,\vec{s})}{f_{\vec{S}}(\vec{s})}$$

However, this would require integrating $f_{\beta,\vec{S}}(b,\vec{s})$ over the range of β:

$$f_{\vec{S}}(\vec{s}) = \int_0^\infty \frac{\delta^\gamma}{\Gamma(\alpha)^{10}\Gamma(\gamma)} b^{10\alpha+\gamma-1} e^{-\delta b} \prod_{k=1}^{10} \frac{T_k^{s_k}\Gamma(\alpha+s_k)}{\Gamma(s_k+1)(T_k+b)^{\alpha+s_k}}\, db.$$

Instead, since $\vec{S}$ is known, $f_{\vec{S}}(\vec{S})$ would be a normalizing constant for $f_{\beta|\vec{S}}(b,\vec{s})$, making $\hat{f}_\beta(b) = f_{\beta,\vec{S}}(b,\vec{s})$ an unnormalized version of the distribution from which we want to draw. Therefore, we can use AR sampling to complete the inference.

3 New Methods

3.1 The AR Algorithm

The AR sampling procedure hinges on two ideas. First, given draws from a certain distribution f over $A \subset \mathbb{R}$, it is possible to make a draw from f conditioned to lie in $B \subset A$ (ie: from the distribution $f(x)/\int_B f(t)\,dt$ for $x \in B$); it would simply require "rejecting" draws from f until a member of B was drawn, at which point it would be "accepted." This is formulated in Theorem 4 from [7]:

Theorem 1. *Suppose that ν is a finite measure over A, and $B \subset A$ where $\nu(B) > 0$. Then if $X_1, X_2, ... \sim \nu(A)$ and $T = inf\{t : X_t \in B\}$ then*

$$X_T \sim [X_1 | X_1 \in B]$$

The second idea is summarized in the following theorem [7]:

Fundamental Theorem of Monte Carlo Simulation. *Suppose that X has density f_X over measure ν on Ω. Then if $[Y|X] \sim \mathsf{Unif}([0, f_X])$, then (X,Y) is a draw from the product measure $\nu \times \mathsf{Unif}$ over the set $(x,y) : x \in \Omega, 0 \le y \le f_X$.*

(For an introduction to the relevant measure theory, see [2].) Intuitively, this means that for some distribution h, it is possible to draw a point (X,Y) uniformly from the area beneath $h(x)$ given a draw $X \sim h$. Furthermore, by multiplying Y by some constant c, (X, cY) becomes a draw from the area beneath $c \cdot h(x)$. To draw from some possibly unnormalized distribution f over $A \subset \mathbb{R}$, we start with a distribution h over A from which we can draw easily. Then, we fix c such that $c \cdot h(x) \ge f(x)$ for all $x \in A$. We could then draw both $X \sim h$ and $Y \sim \mathsf{Unif}([0, h(X)])$ until (X, cY) were in the area beneath $f(x)$ (ie: when $cY < f(X)$). However, for a fixed X, (X, cY) would fall in the area beneath $f(x)$ with probability $f(X)/c \cdot h(X)$. Therefore, after drawing X, it is only necessary to draw $B \sim Bern(f(X)/c \cdot h(X))$; X is a variate from $f(x)$ when $B = 1$. The AR algorithm for drawing $X \sim f$ can be summarized as follows:

1. Let h be some distribution over A and fix c such that $c \cdot h(x) \geq f(x), \forall x \in A$.
2. Draw $X \sim h$.
3. Draw $B \sim Bern(f(X)/c \cdot h(X))$.
4. Repeat 2. and 3. until $B = 1$.
5. $X \sim f$.

3.2 Implementation

For the purposes of this hierarchical model, the unnormalized distribution of interest is

$$\begin{aligned}\hat{f}_\beta(b) &= \frac{\delta^\gamma}{\Gamma(\alpha)^{10}\Gamma(\gamma)} b^{10\alpha+\gamma-1} e^{-\delta b} \prod_{k=1}^{10} \frac{t_k^{S_k}\Gamma(\alpha+S_k)}{\Gamma(S_k+1)(T_k+b)^{\alpha+S_k}} \mathbb{1}(b>0) \\ &= b^{10\alpha+\gamma-1} e^{-\delta b} \prod_{k=1}^{10} (T_k+b)^{-(\alpha+S_k)} c(\vec{S}) \mathbb{1}(b>0),\end{aligned}$$

where $c(\vec{S})$ is a constant with respect to b; dropping this term simply changes the normalizing constant for the distribution and simplifies the implementation. Therefore, for the AR algorithm

$$f(b) = b^{10\alpha+\gamma-1} e^{-\delta b} \prod_{k=1}^{10} (T_k+b)^{-(\alpha+S_k)} \mathbb{1}(b>0). \tag{1}$$

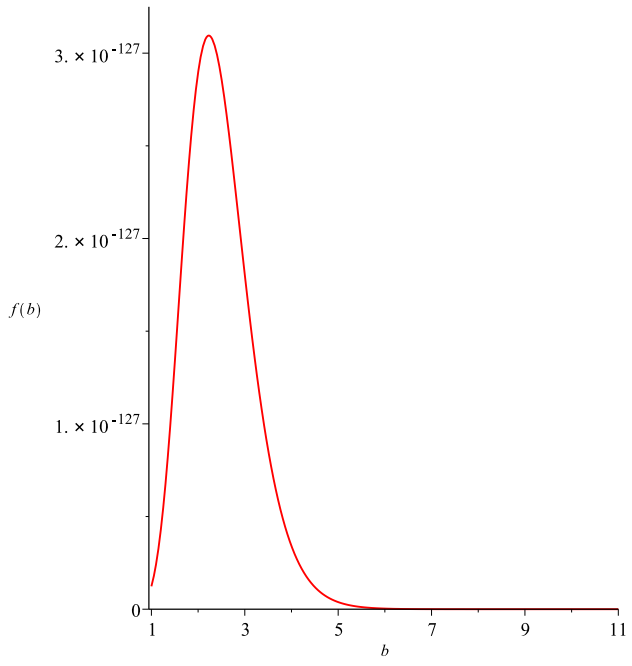


Figure 1: Plot of f(b)

The next step is to determine h, a distribution for which a sampling algorithm already exists, as well as a constant c such that $c \cdot h(x) \geq f(x), \forall x > 0$ in this case. It is also important to note that the choice of c and h determines the running time of the algorithm in the following way [7]:

Theorem 2. *Let Z_f and Z_h be the normalizing constants for f and h respectively. The AR algorithm accepts a draw from h with probability*

$$\frac{Z_f}{cZ_h} = \frac{\int_{x\in A} f(x)\ dx}{\int_{x\in A} c\cdot h(x)\ dx}$$

Therefore, it is sensible to choose h and c such that $c\cdot h(x) - f(x)$ is as small as possible for $x \in A$. For this f, no single family of distributions provides an easily-provable bound for f, so a piecewise approach for defining h is taken instead. The following fact is useful for this purpose:

Fact 3. *For all $b \in [b_1, b_2]$, $e(b_1, b_2) \le f(b) \le g(b_1, b_2)$, where*

$$g(b_1, b_2) = b_2^{10\alpha+\gamma-1} e^{-\delta b_1} \prod_{k=1}^{10} (T_k + b_1)^{-(\alpha+S_k)}$$
$$e(b_1, b_2) = g(b_2, b_1)$$

Proof. We can factor e, f, and g into the functions i, j, and k, where

$$i(b) = b^{10\alpha+\gamma-1} = b^{17.03}$$
$$j(b) = e^{-\delta b} = e^{-b}$$
$$k(b) = \prod_{k=1}^{10} (T_k + b)^{-(\alpha+S_k)}$$

and

$$e(b_1, b_2) = i(b_1)j(b_2)k(b_2)$$
$$f(b) = i(b)j(b)k(b)$$
$$g(b_1, b_2) = i(b_2)j(b_1)k(b_1).$$

For $b > 0$, the function i is monotonically increasing, while the functions j and k are monotonically decreasing. Since $b_1 \le b \le b_2$,

$$i(b_1) \le i(b) \le i(b_2)$$
$$j(b_2) \le j(b) \le j(b_1)$$
$$k(b_2) \le k(b) \le k(b_1)$$

Since each factor of e is less than the corresponding factor of f, $e(b_1, b_2) \le f(b), \forall b \in [b_1, b_2]$. A similar argument holds for f and g. □

Therefore, for b in an interval $[b_1, b_2]$, $h(b) = g(b_1, b_2)$ would be a viable bounding distribution for the purposes of AR sampling. Furthermore, the function e provides a convenient way to determine how to partition $\mathbb{R}$. The probability that AR accepts a draw in a given interval is a function of the endpoints of the interval according to the following fact:

Fact 4. *Let $h(b) = g(b_1, b_2)$ for some interval $I = [b_1, b_2]$, c be 1, and P be the probability that the AR algorithm accepts a draw from h over I as a draw from f over I. Then*

$$P \ge \frac{e(b_1, b_2)}{g(b_1, b_2)}.$$

Proof. By Theorem 2,

$$P = \frac{\int_{b_1}^{b_2} f(x)\ dx}{\int_{b_1}^{b_2} h(x)\ dx} = \frac{\int_{b_1}^{b_2} f(x)\ dx}{(b_2 - b_1)g(b_1, b_2)}$$

Since $f(b) \geq e(b_1, b_2)$ for $b \in I$,

$$\int_{b_1}^{b_2} f(x)\ dx \geq \int_{b_1}^{b_2} e(b_1, b_2)\ dx = (b_2 - b_1)e(b_1, b_2) \Rightarrow \frac{\int_{b_1}^{b_2} f(x)\ dx}{(b_2 - b_1)g(b_1, b_2)} \geq \frac{e(b_1, b_2)}{g(b_1, b_2)}.$$

□

To ensure that the AR algorithm accepts at least $A\%$ of draws from the bounding distribution, a starting point $b_1 > 0$ is chosen and b_{i+1} is chosen as the largest point where

$$\frac{e(b_i, b_{i+1})}{g(b_i, b_{i+1})} \geq A\%,$$

which can be accomplished through a simple binary search. The point b_1 must be greater than 0 because $e(0, b) = 0$ for all b, meaning the search would not terminate. This process can continue as long as desired; for this problem, the process was continued until $b_i \geq 7.5$ for some i, which is well past the peak of the distribution, with a minimum acceptance probability of 1/1.9 (see Figure 1 and global variables in Section B.1). In order to bound the tail of f, we make use of the following fact:

Fact 5. *If h is the distribution for a Gamma-distributed random variable with shape parameter $\alpha = 10\alpha + \gamma$ and rate parameter $\beta = 1$, then $\forall b_0 > 0$ and $\forall b > b_0$,*

$$f(b) \leq \Gamma(10\alpha + \gamma) \prod_{k=1}^{10} (T_k + b_0)^{-(\alpha + S_k)} h(b) = c \cdot h(b)$$

Proof. The distribution of the Gamma-distributed variable is

$$h(b) = \frac{1}{\Gamma(10\alpha + \gamma)} b^{10\alpha + \gamma - 1} e^{-\delta b} \Rightarrow c \cdot h(b) = b^{10\alpha + \gamma - 1} e^{-\delta b} \prod_{k=1}^{10} (T_k + b_0)^{-(\alpha + S_k)}.$$

For $b > b_0 > 0$, by monotonicity of k(b) in the proof of Fact 3

$$\prod_{k=1}^{10} (T_k + b)^{-(\alpha + S_k)} \leq \prod_{k=1}^{10} (T_k + b_0)^{-(\alpha + S_k)} \Rightarrow f(b) \leq b^{10\alpha + \gamma - 1} e^{-\delta b} \prod_{k=1}^{10} (T_k + b_0)^{-(\alpha + S_k)} \Rightarrow f(b) \leq c \cdot h(b).$$

□

Altogether, after the endpoints b_i for $i = 0, 1, ..., n$ have been properly selected, we have

$$h(b) = \begin{cases} g(b_i, b_{i+1}) & : b \in [b_i, b_{i+1}) \\ b^{10\alpha + \gamma - 1} e^{-\delta b} \prod_{k=1}^{10} (T_k + b_n)^{-(\alpha + S_k)} & : b \geq b_n \end{cases} \tag{2}$$

The final step is to draw variates from h, which proceeds as follows. For $\vec{L} \in \mathbb{R}^{n+1}$, let $L_i = h(b_i)$ for $i = 0, 1, ..., n - 1$ and

$$L_n = \mathbb{P}(X > b_n) \cdot \Gamma(10\alpha + \gamma) \prod_{k=1}^{10} (T_k + b_0)^{-(\alpha + S_k)} \text{ where } X \sim Gamma(10\alpha + \gamma, 1).$$

First, a random variable I is drawn from $0, 1, ..., n$ according to the distribution

$$\vec{L}' = \frac{\vec{L}}{\sum_{i=0}^{n} L_i}.$$

As the sequence $\{b_i\}$ can be made arbitrarily long, it is important that this draw be accomplished in constant time, which can be done with $O(n)$ space using Walker's alias method [3]. This was done automatically through the use of R's "sample" function [9], which uses the alias method for $n > 200$. Then, if $I < n$, $X \sim h$ is simply drawn uniformly from $[b_I, b_{I+1}]$. Conversely, if $I = n$, then a draw needs to be made from the Gamma-tail of h. This is accomplished through a slight modification of the inversion method [3]: Let d_Γ and q_Γ be the cdf and quantile functions of the Gamma random variable described in Fact 5 and draw $J \sim \mathsf{Unif}([d_\Gamma(b_n), 1])$. $X \sim h$ is then simply the value $q_\Gamma(J)$.

4 Results

To evaluate the accuracy of a particular implementation of an empirical model, it is necessary to carry out two general classes of testing:

1. verification, or confirming that the model is implemented correctly according to its design, and
2. validation, or determining whether the model can be used to achieve the goals for which it was designed.

4.1 Verification

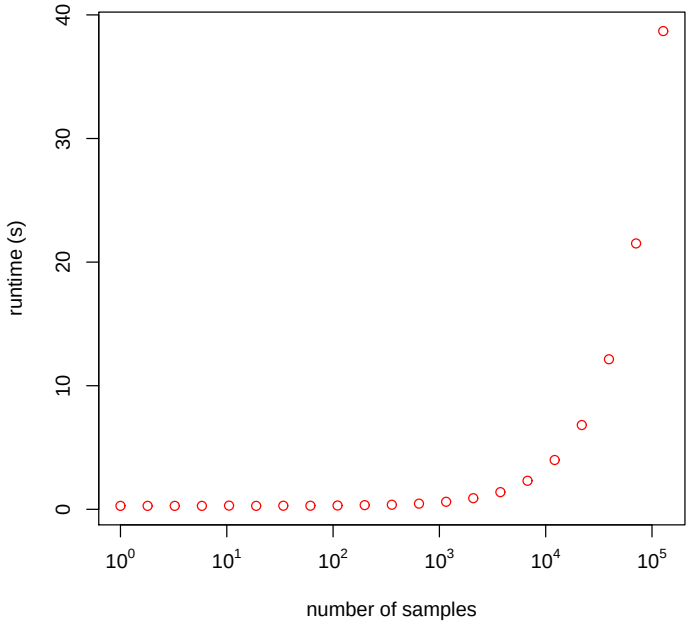


Figure 2: runtime for various numbers of samples

The first goal during verification was to measure the runtime of the AR algorithm for generating n samples. As Figure 2 demonstrates, the AR algorithm appears to run in linear time. Of course, before variates can be drawn with this implementation, the support of the distribution has to be partitioned by $\{b_n\}$. This took less than 1 second (using an Intel® Core(TM) i5-4570 CPU @ 3.20GHz with R version 3.2.0), and is dominated by the actual sampling process for large numbers of variates. A sample $\{x_k\}$ of 100,000 variates was generated for the remaining verification, which took about 30 seconds.

To determine whether the variates followed the necessary distribution, a histogram was plotted along with the original plot of f (Figure 3). In addition, the mean of f was estimated through numerical integration (using the trapezoidal rule [1]):

$$\int_0^\infty b \cdot f(b)\ db \approx 2.470975 \text{ vs } \bar{x} = 2.471813$$

Both of these suggest that this implementation samples from the unnormalized distribution correctly; the histogram matches the plot of f and the sample mean and estimated population mean are close.

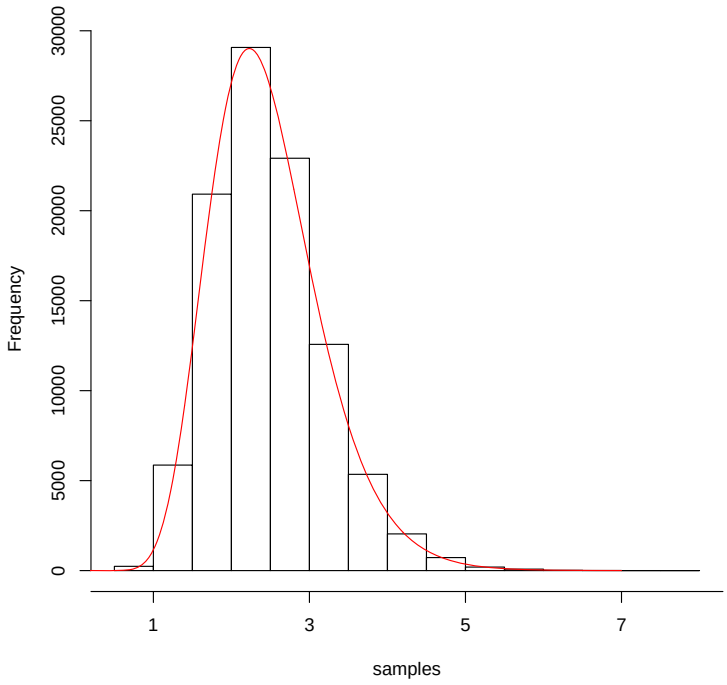


Figure 3: histogram of 100,000 samples with f(b)

4.2 Validation

The implementation provided more of a challenge with respect to validation. The intent of the model is to accurately infer the values of β and $\vec{\lambda}$. Therefore, a way to validate the implementation withe respect to inferring β would be to:

1. Let $T_k = 1$ for $k = 1, ..., 10$.
2. Fix β.
3. Draw $\lambda_k \sim Gamma(\alpha, \beta)$.
4. Draw $S_k \sim Po(\lambda_k)$.
5. Use AR to draw a sample from f.
6. Use the sample to estimate β and $\vec{\lambda}$ (ie by using the mean of the sample as an estimate for β).

7. Compare the estimates to the "ground truth" values.

However, differing values of $\vec{S}$ cause the resulting f to vary wildly in scale. This led to problems relating to numerical underflow, as well as difficulties with the partitioning process. These would not be serious issues in practice since it is straightforward to tune the implementation for a particular scale of f. However, automating the tuning process for various $\vec{S}$ proved nontrivial. Therefore, after generating $\beta, \vec{\lambda}$, and $\vec{S}$ as above, the following approach was taken instead:

1. Use numerical integration to estimate the mean of f.
2. Estimate β and $\vec{\lambda}$ using the estimated mean of f.
3. Compare the estimates to the "ground truth" values.

Murdoch and Green describe the prior on β as "relatively diffuse" [8]. However, according to this prior, $\mathbb{P}(\beta > 2.472) \approx 2.63 \times 10^{-4}$, where $\beta \approx 2.472$ was our AR estimate. Therefore, we begin by testing values of β from 0 to 3. As Figure 4 shows, the inference appears to be unbiased, though the plot also demonstrates clear heteroskedasticity in the data. A simple linear regression

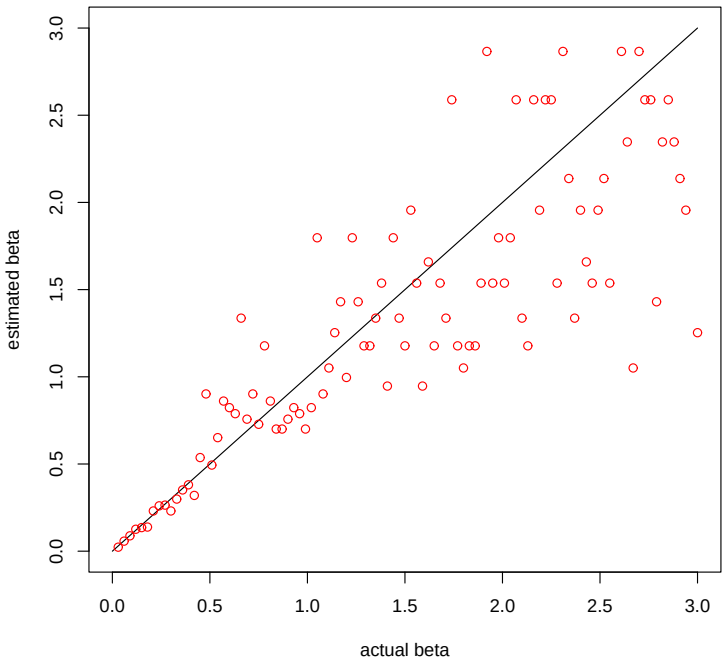


Figure 4: estimated β vs true β, with $y = x$ line

with heteroskedasticity-robust standard errors yields the following:

$$\hat{\beta} = \underset{(0.0594)}{0.1625} + \underset{(0.0531)}{0.7547}\beta$$

where both the intercept and the coefficient are significant at the 99.9% confidence level. This underestimating bias becomes even more pronounced when much higher values of β are tested (see Figure 5). This is likely due to the influence of the prior, suggesting that extreme values of β will be harder to detect if the prior reflects the reality. To validate the inference of $\vec{\lambda}$, we can use the estimate of β to generate an estimate for $\vec{\lambda}$ (which will be discussed in detail in Section 4.3). The

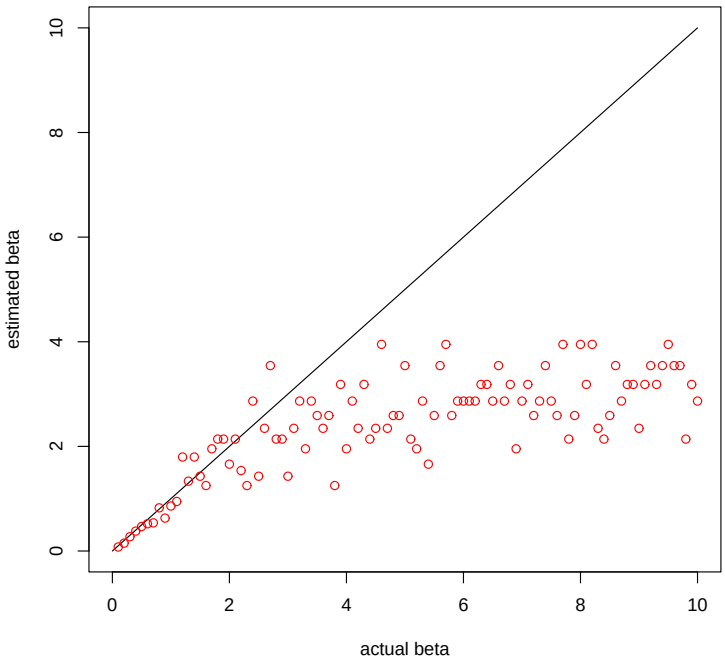


Figure 5: estimated β vs true β, with $y = x$ line

accuracy of the estimate will be assessed using the L_1 norm of the difference between the estimated and actual $\vec{\lambda}$'s, or:

$$\sum_{k=1}^{10} |\lambda_k - \hat{\lambda}_k|$$

where $\hat{\lambda}_k$ is the estimate. These differences are plotted in Figure 6, where it appears that the least extreme values with respect to the β prior yield the worst estimates of $\vec{\lambda}$.

4.3 Analysis of the original data

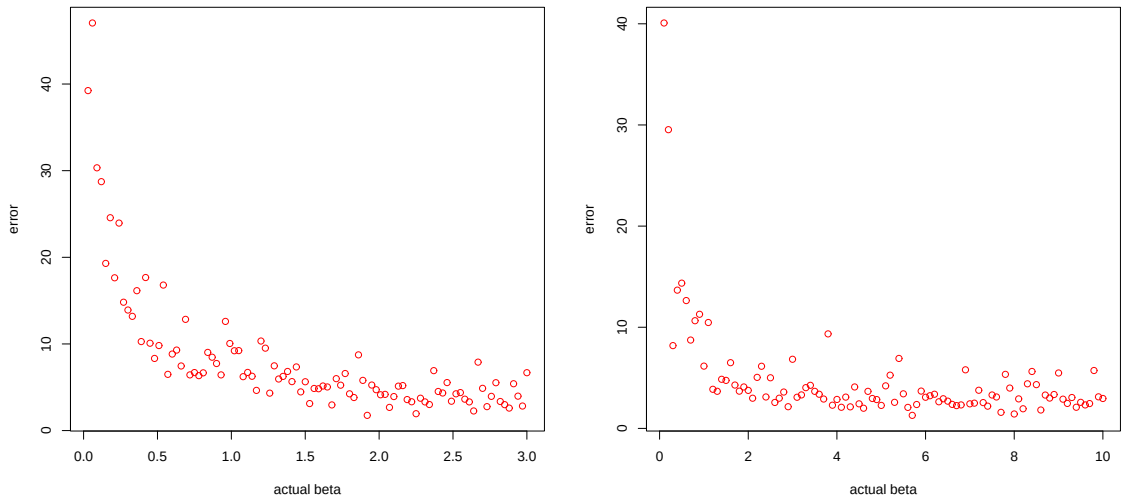
Given the generally positive results from verification and validation (at least for $\beta \approx 2.472$), the model can be used for inferences and prediction about the given data. The primary goal is to estimate $\vec{\lambda}$, which can be done with a sample of β's from f. We know from Fact 1 that

$$[\lambda_k|\beta, S_k, T_k] \sim Gamma(\alpha + S_k, \beta + T_k) \Rightarrow \mathbb{E}[\lambda_k|\beta, S_k, T_k] = \frac{\alpha + S_k}{\beta + T_k}.$$

Therefore, if x_i is a variate from the sample,

$$\mathbb{E}\left[\frac{\alpha + S_k}{x_i + T_k}\right] = \frac{\alpha + S_k}{\mathbb{E}[x_i] + T_k}$$

is an estimate for λ_k. In this way, the hierarchical model approach takes into account the entire vector $\vec{S}$ for the estimation of each λ_k. Conversely, in a naive approach that did not make use of this hierarchical model, the best point estimate for λ_k would be the empirical failure rate, S_k/T_k by the following:



(a) reasonable β values

(b) extreme β values

Figure 6: error in estimating $\vec{\lambda}$ vs true β

Fact 6.

$$\underset{x}{\operatorname{argmax}}\, \mathbb{P}(S_k = S_k | \lambda_k = x) = \frac{S_k}{T_k}$$

Proof. Dropping the subscript k again for convenience,

$$\mathbb{P}(S = S|\lambda = x) = \frac{T^S}{S!} x^S e^{-Tx}.$$

Then we maximize the log of the probability:

$$\frac{d}{dx} \log(\mathbb{P}(S = S|\lambda = x)) = \frac{d}{dx} \left(\log \frac{T^S}{S!} + S \log x - Tx \right) = \frac{S}{x} - T.$$

Setting the derivative equal to zero yields the result. □

In Figure 7, we compare our results to those that would be obtained from this naive procedure, as well as estimates provided by [6], which examined the same data using Gibbs samplers with various prior distributions on α (The actual data for this and the following charts can be found in Section A). Note that the AR estimates have a smaller effective range than the other two series of estimates.

In a similar fashion to [6], we also consider the effect on our model of differing values of α. Confidence intervals were generated by taking a 100,000-variate sample, ordering it, and then taking the 2,500th entry and the 97,500th entry. As Figure 8 demonstrates, the model is robust to changes in α over the range $[1, 10]$.

5 Conclusion

The model provided accessible conditional distributions for each parameter given the others and the data. Using this, we were able to access an unnormalized probability distribution f. Then, through the use of AR sampling with a procedurally-generated bounding distribution h, it was possible to draw from f with a high rate of acceptance. This allowed the rapid generation of large samples

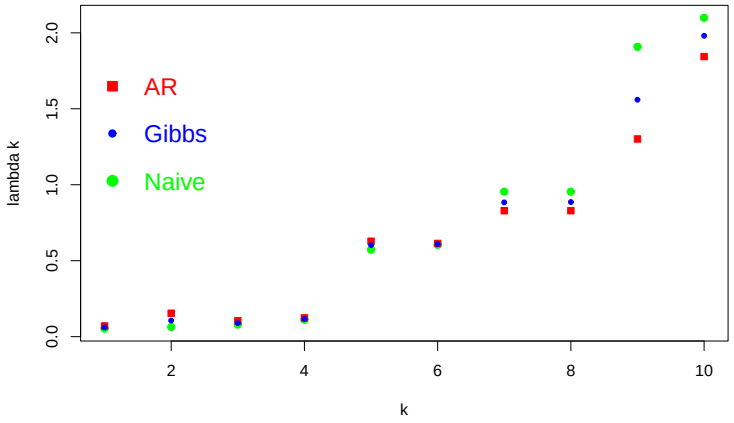


Figure 7: comparison of λ estimates

from the posterior of β, which we could then use to make inferences about $\vec{\lambda}$ using the distribution of $\vec{\lambda}$, conditioned upon β and the data S_k and T_k. Our results agreed with the results from similar inference performed using a MCMC method, and displayed somewhat less dispersion than a less sophisticated method that did not use a hierarchical model. This project highlighted a number of directions for future study. For the purposes of determining the time advantage of AR sampling, it would be good to test an R implementation of the MCMC methods that have been used with this model side-by-side with this implementation. Furthermore, with further study, it might be possible to find a single reasonable bounding distribution h for the posterior f, or to automate the partitioning process in such a way as to make possible a more detailed validation of the algorithm.

A Data

Table 1: Pump data set

k	S_k	T_k
1	5	94.320
2	1	15.720
3	5	62.880
4	14	125.760
5	3	5.240
6	19	31.440
7	1	1.048
8	1	1.048
9	4	2.096
10	22	10.480

B Code

B.1 AREstimate.R

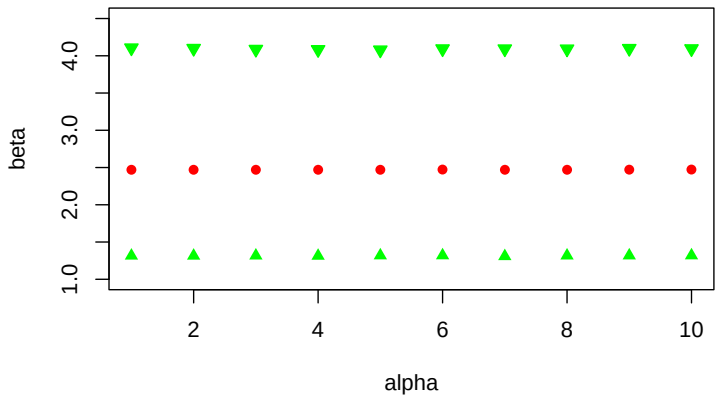


Figure 8: confidence intervals for varying α

Table 2: Lambda estimate comparison

AR	Gibbs	Naive
0.07	0.06	0.05
0.15	0.11	0.06
0.10	0.09	0.08
0.12	0.12	0.11
0.63	0.60	0.57
0.61	0.61	0.60
0.83	0.88	0.95
0.83	0.89	0.95
1.30	1.56	1.91
1.84	1.98	2.10

```
# This file implements all the functions needed to create a bounding
# distribution, sample from it, and sample from the unnormalized distribution

# initialize parameters of the model------------------------------------------
ALPHA <- 1.802
DELTA <- 1
GAMMA <- .01


# initialize global variables-------------------------------------------------
DOMINANCE <- 1.9    # The largest factor by which g(x) is allowed to dominate e(x)
END_OF_DISCRETE <- 7.5
                    # How far out to discretize
RANGEFIND <- END_OF_DISCRETE / 10
                    # The length of the first interval searchPoint tries
EPSILON <- .0001 # The interval length at which searchPoint says "Good Enough"
FIRST_OUT <- .01

# define lower and upper bound functions--------------------------------------
```

Table 3: Confidence intervals with varying α

α	lower	mean	upper
1	1.3158	2.4710	4.1060
2	1.3151	2.4709	4.0998
3	1.3172	2.4692	4.0872
4	1.3140	2.4693	4.0834
5	1.3192	2.4686	4.0784
6	1.3197	2.4722	4.0939
7	1.3087	2.4700	4.0928
8	1.3174	2.4711	4.0915
9	1.3184	2.4712	4.0989
10	1.3194	2.4734	4.0947

```
# helper function for lower and upper bound functions that
# computes the big product at the end of f(x)
lastTermF <- function(b){
        answer <- 1
        for (k in 1:10){
                term <- (T[k] + b)^(-1*(ALPHA + S[k]))
                answer <- answer * term
        }
        return(answer)
}
# the upper bound on f(x) for x in [b1,b2]
gFunc <- function(b1,b2){
        bExponent <- 10*ALPHA + GAMMA - 1
        return((b2^bExponent)*(exp(-DELTA*b1))*lastTermF(b1))
}

# logarithmically implemented versions of gFunc, eFunc, and fFunc
gFunc2 <- function(b1,b2){
        bExponent <- 10*ALPHA + GAMMA - 1
        logGFunc <- 0
        for (k in 1:10){
                term <- log(T[k] + b1)*(-1*(ALPHA + S[k]))
                logGFunc <- logGFunc + term
        }
        logGFunc <- logGFunc - (DELTA*b1) + (log(b2)*bExponent)
        return(exp(logGFunc))
}

eFunc2 <- function(b1,b2){
        return(gFunc2(b2,b1))
}

fFunc2 <- function(x){
        return (gFunc2(x,x))
}

# the lower bound on f(x) for x in [b1,b2]
```

```
eFunc <- function(b1,b2){
        return(gFunc(b2,b1))
}
# f(x) itself
fFunc <- function(x){
        return (gFunc(x,x))
}


# define functions that allow discretization ----------------------------------

# tests whether g(x) < DOMINANCE * e(x) over the interval [b1,b2]
lowEnough <- function(b1,b2)
{
g <- gFunc(b1,b2)
e <- eFunc(b1,b2)
quo <- g/e
return(quo < DOMINANCE)
}

# uses a binary search algorithm to find the greatest point b2
# for which lowEnough(b1,b2) is true
searchPoint <- function(b1){
        start <- b1
        end <- b1 + RANGEFIND
        range <- RANGEFIND
        while (range > EPSILON){
                if (lowEnough(b1,end)){
                        start <- end
                        end <- start + range
                } else {
                        end <- (range/2) + start
                }
                range <- end - start
        }

        if(start > b1){
                return(end)
        } else {
                print("searchPoint terminated without finding a successful point")
                stop
        }
}

# splits up the real line between 0 and end in the required way
discretize <- function(end){
        I <- vector(mode="numeric",length=1000)
        I[1] <- 0
        I[2] <- FIRST_OUT
        lastElement <- 2
        while(I[lastElement] < end){
                start <- I[lastElement]
                b2 <- searchPoint(start)
```

```
                I[(lastElement + 1)] <- b2
                lastElement <- lastElement + 1
        }
        I[lastElement] <- end
        return (head(I,lastElement))
}

# Given a set of intervals (provided by discretize), stepIntegrate
# calculates the area underneath h between 0 and END_OF_DISCRETE, where:
#     h(x) = gFunc(b[i],b[i+1]) for x in [b[i],b[i+1]]
stepIntegrate <- function(intervals){
        area <- 0
        for (k in 2:length(intervals)){
                b1 <- intervals[k-1]
                b2 <- intervals[k]
                intLength <- b2 - b1
                intHeight <- gFunc(b1,b2)
                area <- area + (intLength * intHeight)
        }
        return (area)
}

# normalizing constant for the gamma that bounds f(x) for x > b
scalingConstant <- function(b){
        answer <- (DELTA^(10*ALPHA+GAMMA))/gamma(10*ALPHA+GAMMA)
        return (answer / lastTermF(b))
}

# sampling functions-----------------------------------------------------

# helps sampleFromH by drawing a bucket
drawBucket <- function(){
        return (sample((1:length(probs)),replace = TRUE, size = 1, prob = probs))
}

# helps sampleFromH by drawing from a given bucket
sampleFromBucket <- function(bucket){
        if (bucket == length(probs)){
                # This is the case where we have to generate from the
                # gamma tail
                minQuant <- pgamma(END_OF_DISCRETE,shape=(10*ALPHA+GAMMA),rate=DELTA
                sample <- runif(1,min=minQuant, max=1)
                return (qgamma(sample,shape=(10*ALPHA+GAMMA),rate=DELTA))
        } else {
                b1 <- intervals[bucket]
                b2 <- intervals[bucket+1]
                return (runif(1,min=b1,max=b2))
        }
}

# helps sampleFromF by drawing from the bounding distribution H
sampleFromH <- function() {
        bucket <- drawBucket()
```

```
        return (sampleFromBucket(bucket))
}

# helper function for hFunc - determines which bucket the argument comes from
whichBucket <- function (b) {
        for (i in 1:length(intervals)){
                if (b < intervals[i]){
                        return (i-1)
                }
        }
        return (length(intervals))
}

# the pdf for the bounding distribution, h
hFunc <- function (b) {
        bucket <- whichBucket(b)
        if (bucket == length(intervals)){
                return (dgamma(b, shape=(10*ALPHA+GAMMA),rate=DELTA)/Z)
        } else {
                return (gFunc(intervals[bucket],intervals[bucket+1]))
        }
}

# does AR to draw from f(b)
sampleFromF <- function(){
        h <- sampleFromH()
        C <- rbinom(1,size=1,prob=(fFunc(h)/hFunc(h)))
        if (C == 1){
                return (h)
        } else {
                return (sampleFromF())
        }
}
```

B.2 SetUp.R

```
# This file will set up data so that "sampleFromF()" in AREstimate will work
# properly. Note that if any capitalized global variables are changed (eg
# ALPHA), this needs to be run again.

intervals <- discretize(END_OF_DISCRETE)

discArea <- stepIntegrate(intervals)
Z <- scalingConstant(END_OF_DISCRETE)
tailArea <- (1 - pgamma(END_OF_DISCRETE,shape=(10*ALPHA+GAMMA),rate=DELTA)) / Z

totalArea <- discArea + tailArea
probDiscrete <- discArea / totalArea
probDiscrete

# Generate unnormalized probability vector corresponding to each bucket,
# including an entry for the entire area past END_OF_DISCRETE
probs <- vector(mode="numeric",length=length(intervals))
```

```
for (k in 2:length(intervals)){
        b1 <- intervals[k-1]
        b2 <- intervals[k]
        intLength <- b2 - b1
        intHeight <- gFunc(b1,b2)
        probs[k-1] <- intLength * intHeight
}
probs <- append(probs,tailArea)
sum(probs)
```

B.3 Verification.R

```
# Assumes the pump data is loaded.

# Checking runtimes (takes a while)----------------------------------------
runtimes <- rep(0,21)
for (i in 0:20){
        NUM_SAMPLES <- 1.8^i
        start <- proc.time()
        timeSamples <- drawWithAlpha()
        end <- proc.time()
        x <- end - start
        runtimes[i+1] <- x[3]
}
plot(1.8^(0:20),runtimes,log='x',xaxt='n', col = 'red',
        xlab = 'number of samples', ylab = 'runtime (s)')
ticks <- seq(0,5,by=1)
labels <- sapply(ticks, function(i) as.expression(bquote(10^ .(i))))
axis(1, at=10^ticks, labels=labels)


# Confirming distribution--------------------------------------------------

# determining mean by numerical integration (using trapezoid method as laid
# out at "http://en.wikipedia.org/wiki/Trapezoidal_rule" )
x <- (0:1000)/100 # only need to go out to 10
y <- fFunc(x)
estNormConst <- (10/2000)*(2*sum(y) - y[1] - y[1001])
estNormConst
firstMomentIntegrand <- function(x){
        return (x*fFunc(x)/estNormConst)
}
z <- firstMomentIntegrand(x)
estMean <- (10/2000)*(2*sum(z) - z[1] - z[1001])
estMean
mean(samples)

# comparison histogram (this part is hard coded for NUM_SAMPLES = 100K)
x <- (1:5000)/(5000/7)
y <- fFunc(x)
y <- y *(29000/y[1607])
hist(samples,include.lowest=TRUE,xaxt="n",main=NULL)
axis(1,at=c(0,1,3,5,7,9),labels=c(0,1,3,5,7,9))
```

```
lines(x,y,col="red")



# Robustness to alpha prior (takes a while) ------------------------------------

alphasToTest <- 1:10
high95 <- vector(mode="numeric",length=10)
low95 <- vector(mode="numeric",length=10)
for (alph in 1:10){
        ordSamples <- sort(drawWithAlpha(alphasToTest[alph]))
        high95[alph] <- ordSamples[97500]
        low95[alph] <- ordSamples[2500]
}



# Checking lambda estimates ----------------------------------------------------

lambdaMeans <- matrix(nrow=NUM_SAMPLES, ncol=10)
lambdaMeanEstimates <- vector(mode="numeric", length=10)
for (lamb in 1:10){
        vector <- (ALPHA+S[lamb])/(samples + T[lamb])
        lambdaMeans[,lamb] <- vector
        lambdaMeanEstimates[lamb] <- mean(vector)
}
CLDEstimates = c(.061,.106,.09,.117,.603,.609,.884,.886,1.56,1.981)
plot(1:10,S/T,col="green",bg="green",pch=21,ylab="lambda k",xlab="k")
points(1:10,lambdaMeanEstimates,col="red",bg="red",pch=22)
points(1:10,CLDEstimates,col="blue",bg="blue",pch=20)
legend("topleft"
  , inset = c(0,0.1),
  , cex = 1.5,
  , bty = "n",
  , legend = c("AR", "Gibbs", "Naive"),
  , text.col = c("red", "blue","green"),
  , col = c("red", "blue","green"),
  , pt.bg = c("red","blue","green")
  , pch = c(22,20,21)
)
```

B.4 Validation.R

```
# save the original data
OLD_S <- S
OLD_T <- T

# Over reasonable beta values --------------------------------------------------

betasToTest <- (0:100)/(100/3)
betasToTest <- betasToTest[2:length(betasToTest)]
estimatedBetas <- (1:100)
```

```
estimatedLambdas <- matrix(nrow=length(betasToTest), ncol=10)
actualLambdas <- matrix(nrow=length(betasToTest), ncol=10)
T <- rep(1,10)
firstMomentIntegrand <- function(x){
        return (x*fFunc(x)/estNormConst)
}
for (BETA in 1:length(betasToTest)){
        # generating ground truth values
        L <- rgamma(n=10, shape=ALPHA, rate=betasToTest[BETA])
        S <- vector(mode="numeric",length=10)
        for (pump in 1:10){
                S[pump] <- rpois(n=1,lambda=L[pump]*T[pump])
        }

        # estimating beta and lambdas
        x <- (0:1000)/100 # only need to go out to 10
        y <- fFunc(x)
        estNormConst <- (10/2000)*(2*sum(y) - y[1] - y[1001])
        z <- firstMomentIntegrand(x)

        estimatedBetas[BETA] <- (10/2000)*(2*sum(z) - z[1] - z[1001])
        actualLambdas[BETA,] <- L
        estimatedLambdas[BETA,] <- (ALPHA+S)/(estimatedBetas[BETA] + T)
}
# beta plot with x=y line
curve(x*1,from=0,to=3,xlab="actual beta",ylab="estimated beta")
points(betasToTest,estimatedBetas,col="red")

#robust standard errors
model <- lm(estimatedBetas ~ betasToTest)
require("sandwich")
require("lmtest")
model$newse<-vcovHC(model)
coeftest(model,model$newse)

# lambda plot
lambdaErrors <- rowSums(abs(estimatedLambdas - actualLambdas))
plot(betasToTest,lambdaErrors,col="red",xlab="actual beta",ylab = "error")


# Over extreme beta values ----------------------------------------------

betasToTest <- (0:100)/(100/10)
betasToTest <- betasToTest[2:length(betasToTest)]
estimatedBetas <- (1:100)
estimatedLambdas <- matrix(nrow=length(betasToTest), ncol=10)
actualLambdas <- matrix(nrow=length(betasToTest), ncol=10)
T <- OLD_T # better results here using the original T

for (BETA in 1:length(betasToTest)){
        # generating ground truth values
        L <- rgamma(n=10, shape=ALPHA, rate=betasToTest[BETA])
        S <- vector(mode="numeric",length=10)
```

```
        for (pump in 1:10){
                S[pump] <- rpois(n=1,lambda=L[pump]*T[pump])
        }

        # estimating beta and lambdas
        x <- (0:1000)/100 # only need to go out to 10
        y <- fFunc(x)
        estNormConst <- (10/2000)*(2*sum(y) - y[1] - y[1001])
        z <- firstMomentIntegrand(x)

        estimatedBetas[BETA] <- (10/2000)*(2*sum(z) - z[1] - z[1001])
        actualLambdas[BETA,] <- L
        estimatedLambdas[BETA,] <- (ALPHA+S)/(estimatedBetas[BETA] + T)
}
# beta plot with x=y line
curve(x*1,from=0,to=10,xlab="actual beta",ylab="estimated beta")
points(betasToTest,estimatedBetas,col="red")

# lambda plot
lambdaErrors <- rowSums(abs(estimatedLambdas - actualLambdas))
plot(betasToTest,lambdaErrors,col="red",xlab="actual beta",ylab = "error")

# reload the original data
S <- OLD_S
T <- OLD_T
```

B.5 Thesis.R

```
## Personal working file for duration of this project


# initialize wd and data ------------------------------------------------------
# setwd("C:/Users/Christian/Desktop/Class Notes/Thesis/Simulation Files")
setwd("U:/Class2015/CAyala15/Thesis/Simulation Files")
pData <- read.csv("Pump Data.csv")
S <- pData$si
T <- pData$ti

source("AREstimate.r")
NUM_SAMPLES <- 100000 #How many variates you want

# Draw from distribution ------------------------------------------------------

drawWithAlpha <- function(alpha=1.802){
        ALPHA = alpha
        samples <- vector(mode="numeric",length=NUM_SAMPLES)
        source("SetUp.r")
        for (s in 1:length(samples)){
                samples[s] <- sampleFromF()
        }
        return(samples)
}
```

```
startTime <- proc.time()
samples = drawWithAlpha()
endTime <- proc.time()

# timing
endTime - startTime

# The following call takes awhile
# source("Verification.r")
```

References

[1] Trapezoidal rule. *Wikipedia*.

[2] Robert B. Ash and Catherine A. Doleans-Dade. *Probability and Measure Theory*. 2nd edition, 1999.

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

[4] Donald P. Gaver and I. G. O'Muircheartaigh. Robust empirical bayes analyses of event rates. *Technometrics*, 29.

[5] Alan E. Gelfand and Adrian F. M. Smith. Sampling-based approaches to calculating marginal densities. *Journal of the American Statistical Association*, 85, 1990.

[6] E.I. George, U. E. Makov, and A.F.M. Smith. Conjugate likelihood distributions. *Scandinavian Journal of Statistics*, 20:147–156, 1993.

[7] Mark Huber. *Perfect Simulation*. CRC Press. Forthcoming.

[8] D.J. Murdoch and P.J. Green. Exact sampling from a continuous state space. *Scandinavian Journal of Statistics*, 25:483–502, 1998.

[9] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2014.

[10] Alex Reutter and Johnson Valen. General strategies for assessing convergence of MCMC algorithms using coupled sample paths.

[11] Christian Robert and George Casella. *Monte Carlo Statistical Methods*. Springer-Verlag New York, 2004.