

## Coding your own MCMC algorithm

Chelsea Loomis Lofland, University of California, Santa Cruz, CA

### ABSTRACT

In Bayesian statistics, Markov chain Monte Carlo (MCMC) algorithms are an essential tool for sampling from probability distributions. PROC MCMC is useful for these algorithms, however it is often desirable to code an algorithm from scratch. This is especially present in academia where students are expected to be able to understand and code an MCMC. The ability of SAS to accomplish this is relatively unknown yet quite straightforward. We use SAS IML to demonstrate methods for coding an MCMC algorithm with examples of a Gibbs sampler and Metropolis-Hastings random walk.

### INTRODUCTION

Markov chain Monte Carlo (MCMC) are algorithms used to sample from probability distributions and are very common in Bayesian analysis. Coding MCMC algorithms requires simulation tools such as random draws from a probability density and writing functions. These algorithms tend to be performed more frequently with R than with SAS. This is partly due to the underrepresentation of discussion regarding the abilities of SAS for performing simulations. Another contributing factor is the belief in academic circles that SAS is clunky or solely PROCs. However this was the SAS of days ago, and in this paper we seek to dispel this belief by showing techniques available in SAS along with comparisons to R.

This paper will cover simulation, sampling and Bayesian techniques using SAS 9.3, SAS/IML 12.1 and R 2.13.2. Our focus will be to discuss the SAS coding examples using functions and call subroutines, SAS/IML, and PROC MCMC. Comparable R code will also be provided for examples in this paper.

### IML

Throughout the paper we make heavy use of SAS Interactive Matrix Language (IML). This environment behaves quite differently from the SAS DATA step and other PROCs. Once the IML session is started it is interactive so that the code can be run line by line. The PRINT statement then allows the user to view variables while still in the session, which very closely mimics the behavior of an R session. In academia this environment is desired for writing coding examples from scratch rather than using predefined procedures. As the name suggests, IML is designed to handle vectors and matrices rather than datasets. As we will show with examples, the syntax of IML is intuitive and familiar to R users.

### RANDOM NUMBER GENERATION

The foundation of all simulation is random number generation. The simplest form of random number generation is to pull values at random from a known distribution. The [sascommunity.org](http://sascommunity.org) site (2009) has an excellent discussion of random number generators in SAS. In brief, they describe that there are RANxxx() functions such as the function RANUNI(), and CALL RANxxx() subroutines. RANUNI() is a pseudorandom number generator that generates random numbers from the Uniform distribution. The main difference between the RANxxx() functions and the CALL RANxxx() subroutines is how they use the seed that is the starting point for the random number stream. The RANUNI() function will generate a sequence of  $2^{31}-2$  pseudorandom numbers before it repeats the same sequence of numbers again. In addition all subsequent RANUNI() functions, and changes in subsequent seeds, will not alter the sequence of random numbers.

### STANDARD DISTRIBUTIONS

The RANxxx() functions and CALL RANxxx() subroutines extend far back in SAS software versions. In SAS 9.1 a function called RAND() was added to SAS. This random number generator is based on the complicated Mersenne-Twister algorithm, which is similar to the default random number generator that many use in R (WikiBooks 2011).

When using the RAND() function in SAS it is recommended to also use the CALL STREAMINIT() subroutine with a positive seed if reproducibility of the random number stream is desired. If CALL STREAMINIT() is not included the RAND() function will use the system clock for the seed. In the following code we generate 10 random numbers from the standard normal distribution using the Mersenne-Twister algorithm.

```

DATA normal;
  CALL STREAMINIT(12345);
  DO i=1 to 10;
    Z=RAND('NORMAL',0,1);
    OUTPUT;
  END;
RUN;

PROC PRINT DATA=normal;
RUN;

```

Obs	i	Z
1	1	0.26423
2	2	1.07473
3	3	0.81792
4	4	-0.55277
5	5	1.54014
6	6	-1.23382
7	7	-0.14154
8	8	1.04200
9	9	0.06573
10	10	1.22526

Figure 3. Random numbers from the standard normal distribution

In comparison to SAS, the R script for generating random numbers from a standard normal with the Mersenne-Twister algorithm is shown below followed by the result.

```

set.seed(12345)
print(samples<-rnorm(10,0,1))

[1] 0.5855288 0.7094660 -0.1093033 -0.4534972 0.6058875 -1.8179560
[7] 0.6300986 -0.2761841 -0.2841597 -0.9193220

```

It is important to note that the examples above show that the output of SAS and R have very different appearances. One key difference between the software packages is how the data is stored. In traditional SAS data are stored in data set tables with rows and columns, and the underlying loop processes data row by row. In R data are stored in vectors and matrices.

Another difference between SAS and R is the random number functions themselves. In R each distribution has its own function, while in SAS these distributions may have corresponding RANxxx() functions, or they can be passed in directly as an argument to the RAND() function. The RAND() function in SAS can call from other known distributions as shown in Figure 4 (support.sas.com 2013). Many of the various distributions available in R are listed in Figure 5 for comparison.

Distribution	Argument
Bernoulli	BERNOULLI
Beta	BETA
Binomial	BINOMIAL
Cauchy	CAUCHY
Chi-Square	CHISQUARE
Erlang	ERLANG
Exponential	EXPONENTIAL
F	F
Gamma	GAMMA
Geometric	GEOMETRIC
Hypergeometric	HYPERGEOMETRIC
Lognormal	LOGNORMAL
Negative binomial	NEGBINOMIAL
Normal	NORMAL   GAUSSIAN
Poisson	POISSON
T	T
Tabled	TABLE
Triangular	TRIANGLE
Uniform	UNIFORM
Weibull	WEIBULL

Figure 4. Distributions for the RAND() function

Note that in SAS the Uniform and Gamma distributions have minimal parameter specification. The Uniform argument always generates Uniform(0,1) so for Uniform(a,b) we must use the property: If  $u$  is a random uniform variate in  $[0,1]$ , then  $x = a + (b-a)*u$  is random uniform on  $[a,b]$ . Similarly, a Gamma argument produces Gamma( $\alpha,1$ ) so for a Gamma( $\alpha, \beta$ ) use the property:  $\beta*Gamma(\alpha,1) = Gamma(\alpha, \beta)$  for any  $\beta>0$ .

Another option for generating random numbers with SAS is to use SAS/IML. For a simple task like generating random numbers IML might be overkill. However the object oriented syntax and ability to code data with vectors and matrices lends itself to more complicated simulations to be discussed later in this paper. CALL RANDSEED() sets the seed and the CALL RANDGEN() subroutine fills up an entire matrix at once.

Distribution	Function
Uniform	runif
Poisson	rpois
Multivariate normal	rmvnorm
Negative binomial	rnbinom
Binomial	rbinom
Beta	rbeta
Chi-Square	rchisq
Exponential	rexp
Gamma	rgamma
Logistic	rlogis
Normal	rnorm
T	rt
Geometric	rgeom
Hypergeometric	rhyper
Wilcoxon Rank Sum Statistic	rwilcox
Weibull	rweibull

Figure 5. Distributions in R

```

PROC IML;
  CALL RANDSEED(12345);
  z = J(10,1);
  CALL RANDGEN(z,'Normal',0,1);
  PRINT z;
QUIT;

```

z
0.2642335
1.0747269
0.8179241
-0.552775
1.5401449
-1.233822
-0.141535
1.0420036
0.0657322
1.225259

Figure 6. Random numbers using IML

Other functions worth mentioning are: the PDF() function that returns the probability density at a given time point, the CDF() function that returns the probability that an observation from a given distribution is less than or equal to a particular value, and the QUANTILE() function that given a probability will return the smallest value for which the CDF of that value is greater than or equal to p. These functions mentioned for SAS correspond to the pnorm, qnorm, and dnorm scripts in R (for the normal distribution).

## WRITING A FUNCTION

Writing a function is a key tool for Bayesian methods yet is a relatively unknown capability of SAS. One option to writing your own functions in SAS is to take advantage of modules in SAS/IML, however it is also possible to do this by writing a function through PROC FCMP which will not be presented in this paper.

In the following example we have data that follows a Cauchy distribution with location parameter  $\mu$  and scale parameter  $\sigma$ . Using a non-informative prior  $1/\sigma$  we wish to find the maximum a posteriori. The START and FINISH commands in SAS/IML define a module describing the distribution we wish to maximize. The input for the module is a vector containing the values to optimize. Variables that are used in the module, but are not parameters to it (i.e. defined outside of the module), are part of the GLOBAL statement. We will also show later in the paper how to draw samples from a distribution defined in this way.

```

PROC IML;

y={0.09 0.07 0.08 0.05 0.06 0.09 0.07 0.09 0.05 0.07 0.06 0.05 0.03 0.04 0.03 0.04};
n=NCOL(y);

/*log posterior module*/
START logpost(parm) GLOBAL (y, n);
  mu=parm[1];
  sigma=parm[2];
  pi=CONSTANT('pi');
  ll=-n*LOG(pi)+(n-1)*LOG(sigma)-SUM(LOG(sigma**2+(y-mu)##2));
  RETURN(ll);
FINISH;

```

SAS has many optimization calls that use different methods but have similar syntax. The CALL NLPNRA() subroutine uses the Newton-Raphson method to compute an optimum value of this function.

```

constraint= { . 1e-8 , . . }; /*lower bounds first*/
/*1e-8 since function can't evaluate at zero*/
initial={0.1 0.1}; /*where to start*/
opt={1, 4}; /*1: maximize, 4: level for the amount of output*/
CALL NLPNRA(rc,xres,'logpost',initial,opt,constraint);
QUIT;

```

A portion of the output is shown below in Figure 15. This shows that MU is maximized at approximately 0.06 and SIGMA at approximately 0.01, and that the value of the log posterior at these values is approximately 39.86.

Optimization Results			
Parameter Estimates			
N	Parameter	Estimate	Gradient Objective Function
1	X1	0.059428	-0.000049510
2	X2	0.013102	-0.000034830

Value of Objective Function = 39.862737423

Figure 15. Optimization results for mu and sigma

Coding and output for this example in R is shown below.

```
y=c(0.09, 0.07, 0.08, 0.05, 0.06,
    0.09, 0.07, 0.09, 0.05, 0.07, 0.06, 0.05, 0.03, 0.04, 0.03, 0.04)
n=length(y)
neg.log.posterior<-function(grid){
  mu=grid[1]
  sigma=grid[2]
  if (sigma>0){lp=-n*log(pi)+(n-1)*log(sigma)-sum(log(sigma^2+(y-mu)^2))}
  else lp=NA
  return(-lp) #negative since optim minimizes
}

optim(c(0.1,0.1),neg.log.posterior)
$par
[1] 0.05942726 0.01310100

$value
[1] -39.86274
```

## EXAMPLES

### PROC MCMC

Consider again the data distributed Cauchy with location and scale parameters. We want to obtain posterior samples of the parameters. This can be done very simply with the use of PROC MCMC, as shown below. PROC MCMC works similarly to WinBugs where the user only has to specify the likelihood and priors.

```
DATA accidents;
INPUT deathrate @@;
DATALINES;
0.09 0.07 0.08 0.05 0.06 0.09 0.07 0.09 0.05 0.07 0.06 0.05 0.03 0.04 0.03 0.04
;
RUN;

ODS GRAPHICS ON;

/*Cauchy likelihood and noninformative prior*/
PROC MCMC DATA=accidents NMC=50000 OUTPOST=postout1;
  PARMs mu 1 sigma 1;
  IF sigma>0 THEN lp=-log(sigma);
  ELSE lp=.;
  PRIOR mu sigma ~ GENERAL(lp); /* 1/sigma */
  MODEL deathrate ~ CAUCHY(mu, sigma);
  PREDDIST OUTPRED=predout1 NSIM=1000; /*posterior predictive summaries and
intervals - output not shown*/
RUN;
```

PROC MCMC produces a large amount of output and only a subset is included below in Figure 16. This displays the densities for the parameters, along with trace plots and autocorrelation function (ACF) plots to be used in diagnostics. The trace plot shows convergence of the chain and the ACF does not show a significant amount of correlation, both of which are good.

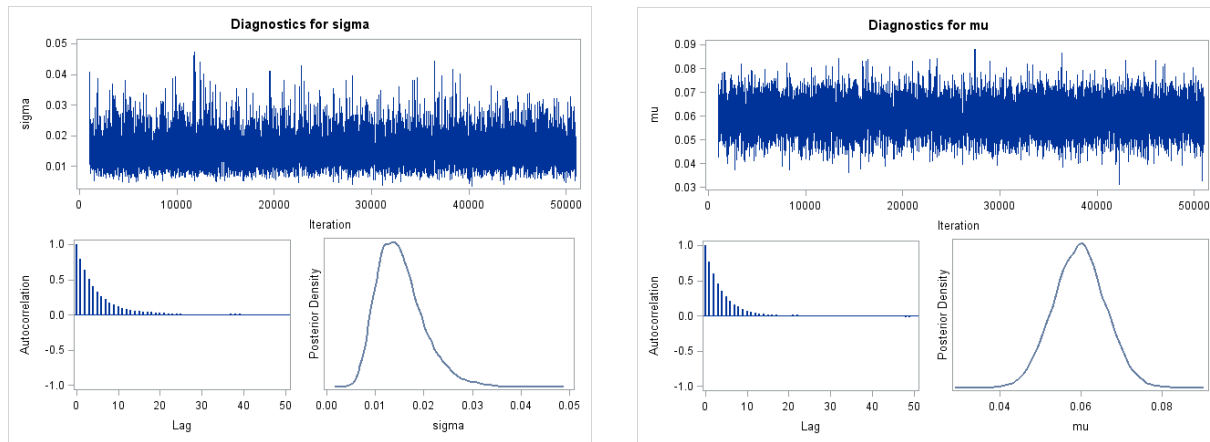


Figure 16. Results from PROC MCMC

There are also packages in R that can automate some of an MCMC, such as the 'metrop' function in the 'mcmc' package that uses a Metropolis algorithm to reach the desired specified equilibrium distribution. We will not show examples from these packages in this paper. A more informative way to do this problem in R is to use a scale mixture of normal so that the full conditionals can be defined and a Gibbs sampler can be used. This also means that we are sampling from the full conditionals exactly, rather than approximating with a Metropolis step which is the default with PROC MCMC. This approach can also be done in SAS using Gibbs sampling methods in SAS/IML or by implementing a new sampling algorithm with PROC MCMC (support.sas.com 2013). We will demonstrate a similar SAS/IML method later in example 2 of this paper.

We use latent variables  $\lambda_i$  to augment the problem and make sampling easier. Recall that a Cauchy distribution is a special case of the Student t distribution with 1 degree of freedom. We can then use the fact that

$$\int_0^{\infty} N\left(x|\mu, \frac{\sigma}{\lambda}\right) Ga\left(\lambda|\frac{\nu}{2}, \frac{\nu}{2}\right) d\lambda = St_{\nu}(x|\mu, \sigma)$$

and set  $\nu = 1$  to achieve a *Cauchy*( $x|\mu, \sigma$ ).  
With this model we have

$$\begin{aligned} x_1, \dots, x_n &\sim \text{Cauchy}(\mu, \sigma) \\ p(\mu, \sigma) &\propto \frac{1}{\sigma} \\ x_i|\lambda_i &\sim N\left(\mu, \frac{\sigma}{\lambda_i}\right) \\ \lambda_i &\sim Ga\left(\frac{\nu}{2}, \frac{\nu}{2}\right) \end{aligned}$$

and the full conditionals are

$$\begin{aligned} p(\lambda_i|\mu, \sigma, x) &\sim Ga\left(\frac{\nu+1}{2}, \frac{1}{2}\left(\nu + \frac{(x_i - \mu)^2}{\sigma}\right)\right) \\ p(\mu|\lambda_1, \dots, \lambda_n, \sigma, x) &\sim N\left(\frac{\sum \lambda_i y_i}{\sum \lambda_i}, \frac{\sigma}{\sum \lambda_i}\right) \\ p(\sigma|\lambda_1, \dots, \lambda_n, \mu, x) &\sim \text{InvGa}\left(\frac{n}{2}, \frac{\sum (x_i - \mu)^2 \lambda_i}{2}\right) \end{aligned}$$

with  $i=1, \dots, 16$ .

This can be used to implement a Gibbs sampler to obtain samples from  $\mu$  and  $\sigma$ .

The code to implement this sampler in R is given below.

```
library(MCMCpack) #for rinvgamma, could use 1/rgamma

year<- c(1987, 1988, 1989, 1990, 1991, 1992,1993,1994,
         1995, 1996, 1997, 1998, 1999, 2000, 2001, 2002)
rate<-c(0.09, 0.07, 0.08, 0.05, 0.06, 0.09, 0.07, 0.09,
        0.05, 0.07, 0.06, 0.05, 0.03, 0.04, 0.03, 0.04)
deaths<-c(900, 742, 879, 544, 638, 1076, 864, 1171, 711, 1146, 929,
          904, 499, 755, 577, 791)
n<-16

#####gibbs sampler

ntot <- 1000000 ###number of total iteration
nburn <- 500000 ####burn-in
thin <- 100 ###thinning (if dont need, thin=1)
nsave <- (ntot-nburn)/thin ###number of iteration actually saved
nsave

#####vector or matrix where save the iterations
muvett <- rep(NA, nsave)
sig2vett <- rep(NA, nsave)
lambdamatr <- matrix(NA, nrow=nsave, ncol=n)
predmatr <- matrix(NA, nrow=nsave, ncol=n)
###for predictive
predvett <- rep(NA, n)
#####

y<-rate

###initialization parameters##
mu<-6
sig2<-4
lambdavett<-rep(1,n)

for (j in 1:ntot)
{
  for (i in 1:n)
  {
    beta <- (1+((y[i]-mu)^2)/sig2)/2
    lambdavett[i] <- rgamma(1,1,beta)
  }

  meanmu <- sum(lambdavett*y) / sum(lambdavett)
  varmu <- sig2 / sum(lambdavett)
  mu <- rnorm(1, mean=meanmu, sd=sqrt(varmu))

  min(predmatr[,1])
  asig2 <- n/2
  bsig2 <- sum((y-mu)^2*lambdavett)/2
  sig2 <- rinvgamma(1,asig2,bsig2)

  for (i in 1:n)
  {
    varpred <- sig2 / lambdavett[i]
    predvett[i] <- rnorm(1, mu, sd=sqrt(varpred))
  }

  if (j> nburn)
  {
    if((j-nburn) %% thin == 0 )
    {
      muvett[(j-nburn)/thin] <- mu
      sig2vett[(j-nburn)/thin] <- sig2
      predmatr[(j-nburn)/thin,] <- predvett
      lambdamatr[(j-nburn)/thin,] <- lambdavett
      print((j-nburn)/thin)
    }
  }
}
}
```



```

sigvett <- sqrt(sig2vett)

layout(matrix(c(1,1,2,3), 2, 2, byrow = TRUE))
traceplot(as.mcmc(muvelt),main='Mu')
acf(muvelt)
hist(muvelt)

layout(matrix(c(1,1,2,3), 2, 2, byrow = TRUE))
traceplot(as.mcmc(sigvett),main='Sigma')
acf(sigvett)
hist(sigvett)

```

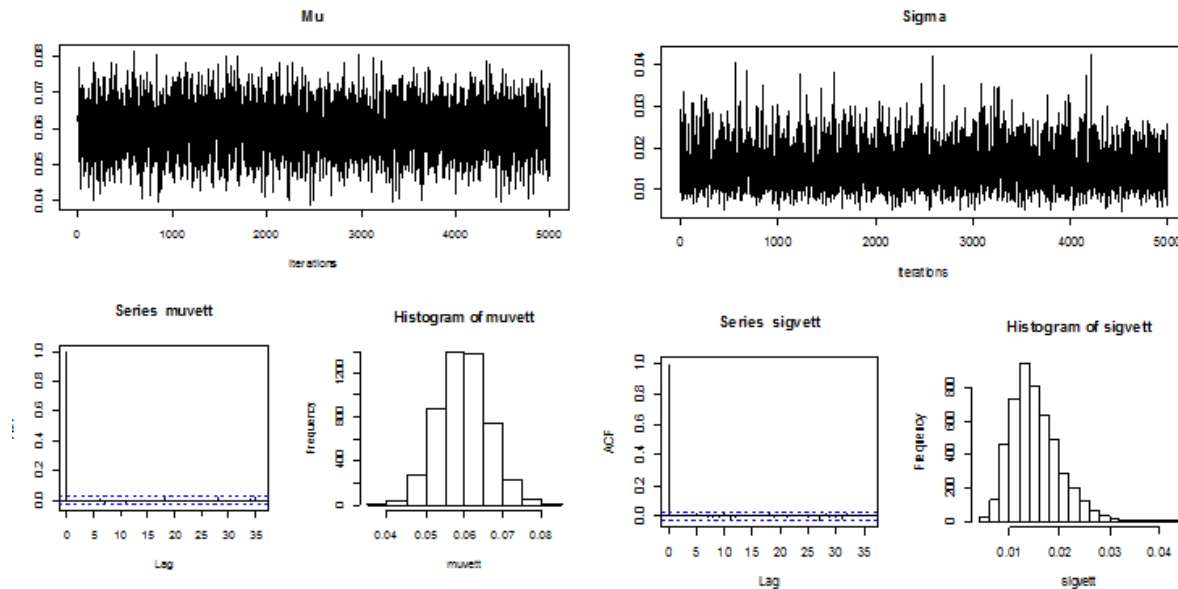


Figure 17. Results from R

Although PROC MCMC is a very useful tool, coding an MCMC from scratch is a desired ability as a teaching tool at University. SAS procedures are essential for work in industry due to being fine-tuned, thoroughly tested and efficient; however, students are often encouraged to work without PROCs in order to fully understand the methodology behind the code. An MCMC algorithm requires heavy use of vectors and matrices, where at each iteration the parameters are sampled using values from the previous iteration, then stored in a vector or matrix. This quality makes it ideal for SAS/IML, which is designed well for matrix operations.

The next two examples using SAS/IML are different MCMC algorithms that are frequently employed. We have also included example code for processes that are commonly performed, such as a trace plot to assess convergence and subsetting samples based on burn-in and thinning. These diagnostics are presented more as a reference rather than a requirement of the problem. Again it is important to note the potential use of PROC PRINTTO in SAS for exercises that may produce a large volume of log messages or output. By directing the log and/or output to a file we can avoid having to sit with the computer during large runs and avoid having to clear these windows by hand.

## GIBBS SAMPLER

The second example demonstrates a Gibbs sampler, which is a simpler algorithm due to the straight forward coding and not requiring tuning like a random walk. However, it does require that all full conditionals can be sampled exactly (i.e. Normal, Gamma, Beta, etc.)

(Carlin, Gelfand and Smith, 1992) Let  $y_1, \dots, y_n$  be a sample from a Poisson distribution for which there is a suspicion of a change point  $m$  along the observation process where the means change,  $m = 1, \dots, n$ . Given  $m, y_i \sim \text{Poisson}(\theta)$ , for  $i = 1, \dots, m$  and  $y_i \sim \text{Poisson}(\phi)$ , for  $i = m+1, \dots, n$ . The model is completed with independent prior distributions  $\theta \sim \text{Gamma}(\alpha, \beta)$ ,  $\phi \sim \text{Gamma}(\gamma, \delta)$  and  $m$  uniformly distributed over  $1, \dots, n$  where  $\alpha, \beta, \gamma, \delta$  are known constants. Implement a Gibbs sampling algorithm to obtain samples from the joint posterior distribution. Run the Gibbs sampler to apply this model to mining data which consists of counts of coal mining disasters in Great Britain by year from 1851 to 1962.

From the conjugacy of Poisson and Gamma we have the posterior distributions for  $\theta$  and  $\phi$  as

$$\theta|y_1, \dots, y_m, m \sim \text{Gamma}\left(\sum_{i=1}^m y_i + \alpha, m + \beta\right)$$

$$\phi|y_{m+1}, \dots, y_n, m \sim \text{Gamma}\left(\sum_{i=m+1}^n y_i + \gamma, n - m + \delta\right)$$

The posterior probability mass function for  $m$  is

$$\begin{aligned} pr(M = m) &\propto \theta^{\sum_{i=1}^m y_i} \phi^{\sum_{i=m+1}^n y_i} e^{-m\theta} e^{m\phi} \\ &\propto \theta^{\sum_{i=1}^m y_i} \phi^{\sum_{i=1}^n y_i - \sum_{i=1}^m y_i} e^{-m(\theta - \phi)} \\ &\propto \left(\frac{\theta}{\phi}\right)^{\sum_{i=1}^m y_i} e^{-m(\theta - \phi)} \end{aligned}$$

This gives the (unnormalized) weights for sampling  $m$  from the values  $1, \dots, n$ .

To set up the algorithm we first read in the data and initialize the vectors where we need to save data.

```
PROC IML;
y={4,5,4,1,0,4,3,4,0,6,3,3,4,0,2,6,3,3,5,4,5,3,1,4,4,1,5,5,3,4,2,5,2,2,3,4,2,1,3,2,2,
,1,1,1,1,3,0,0,1,0,1,1,0,0,3,1,0,3,2,2,0,1,1,1,0,1,0,1,0,0,0,2,1,0,0,0,1,1,0,2,3,3,1,
,1,2,1,1,1,1,2,4,2,0,0,0,1,4,0,0,0,1,0,0,0,0,0,1,0,0,1,0,1};

sum=SUM(y);
n=NROW(y);
r=10000; /*number of samples*/
s=T(1:n); /*values to sample m from*/

/*set constants*/
alpha=1;
beta=1;
gamma=1;
delta=1;

/*initialize*/
theta=J(r,1,0);
phi=J(r,1,0);
m=J(r,1,0);
prob=J(n,1,0);
iter=J(r,1,0);

/*set first value*/
theta[1]=1;
phi[1]=1;
m[1]=n/2;
iter[1]=1;
```

We can now run the algorithm in a loop as shown below with the RAND() function. The syntax for the loop matches the syntax that would be used in a DATA step. The following SAMPLE() function requires only weights for sampling, not probabilities, this means that the values do not need to add to 1. The draws from the Gamma distribution follow from the parameterization with mean  $\alpha/\beta$

```

/*run the algorithm*/
DO i=2 TO r;

  theta[i] = RAND('GAMMA',sum(y[1:m[i-1]])+alpha)/(m[i-1]+beta);
  phi[i]    = RAND('GAMMA',sum(y[(m[i-1]+1):n])+gamma)/(n-m[i-1]+delta);

  DO k=1 TO n; /*calculate weights for sampling m*/
    prob[k] = ( (theta[i]**sum(y[1:k])) / (phi[i]**sum(y[1:k])) ) * exp(-
      k*(theta[i]-phi[i]));
  END;

  m[i]=SAMPLE(s,1,'Replace',prob); /*SAMPLE will normalize prob*/
  iter[i]=i;

END;

CREATE samples VAR {theta phi m iter}; /*save to dataset samples*/
APPEND;
CLOSE samples;

QUIT;

```

Finally we exit the IML session and use SGPLOT to produce plots for the samples from the posterior distributions of Theta, Phi and M as shown in Figures 18 and 19.

```

PROC SGPLOT DATA=samples;
  HISTOGRAM theta;
  HISTOGRAM phi;
RUN;

```

```

PROC SGPLOT DATA=samples;
  VBAR m;
RUN;

```

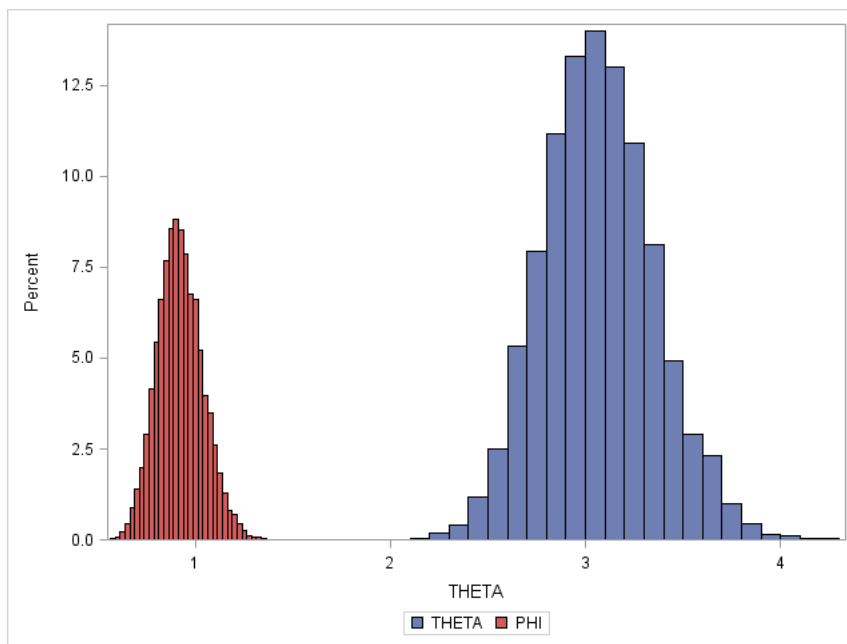


Figure 18. Samples of Theta and Phi

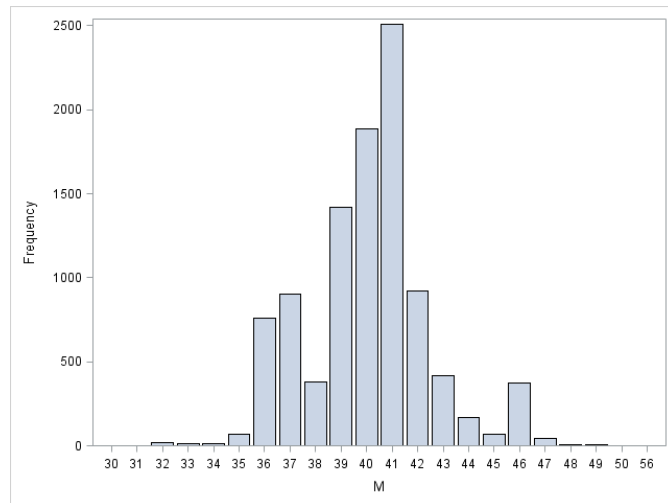


Figure 19. Samples of M

Coding this example in R is very similar to the approach used in SAS/IML in defining the parameters and running the algorithm, there are only subtle coding differences. This is an important point to make in that coding a sampler algorithm from scratch in SAS is in fact not clunky, and uses a very similar approach to that of R. The R code is shown below with similar plots of the results in figures 20 and 21. If it is desired to have both plots on the same graph the `par(mfrow=c(1,2))` option can be used.

```

mining=
c(4,5,4,1,0,4,3,4,0,6,3,3,4,0,2,6,3,3,5,4,5,3,1,4,4,1,5,5,3,4,2,5,2,2,3,4,2,1,3,2,2
,1,1,1,1,3,0,0,1,0,1,1,0,0,3,1,0,3,2,2,0,1,1,1,0,1,0,1,0,0,0,2,1,0,0,0,1,1,0,2,3,3,
1,1,2,1,1,1,1,2,4,2,0,0,0,1,4,0,0,0,1,0,0,0,0,0,1,0,0,1,0,1)
sum=sum(mining)
n=length(mining)
R=10000

alpha=1
beta=1
gamma=1
delta=1

theta=NULL
phi=NULL
m=NULL
prob.m=NULL
theta[1]=1
phi[1]=1
m[1]=n/2

for(i in 2:R){
  theta[i] = rgamma(1,sum(mining[1:m[i-1]])+alpha,m[i-1]+beta)
  phi[i]   = rgamma(1,sum(mining[(m[i-1]+1):n])+gamma,n-m[i-1]-delta)

  for(k in 1:n){
    prob.m[k] = (theta[i]^sum(mining[1:k])/phi[i]^sum(mining[1:k]))*exp(-
      k*(theta[i]-phi[i]))
  }

  m[i]=sample(c(1:n),size=1,prob=prob.m/sum)
}

hist(theta,breaks=100)
hist(phi, breaks=100)

```

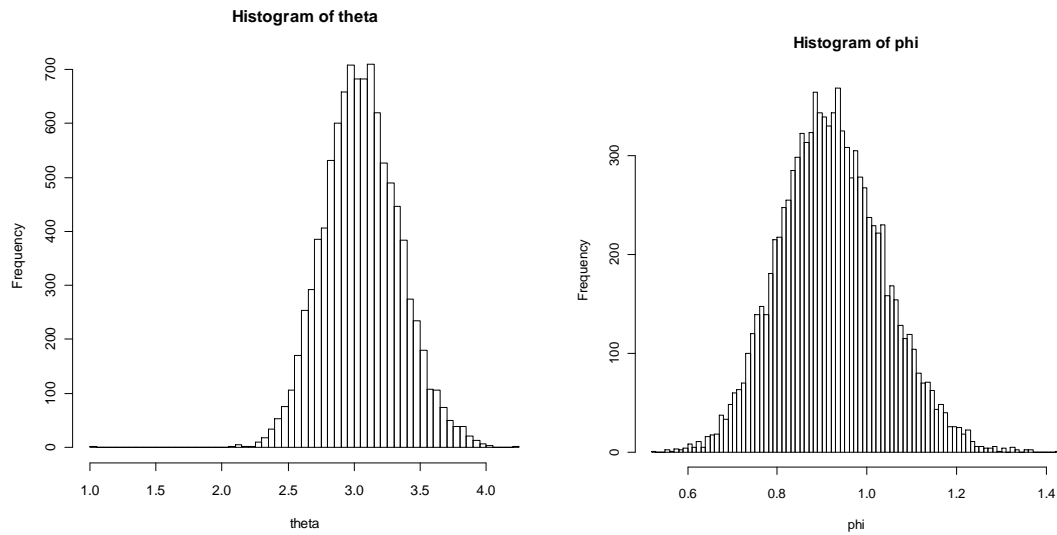


Figure 20. Samples of Theta and Phi

```
counts<-table(m.sample)
barplot(counts)
```

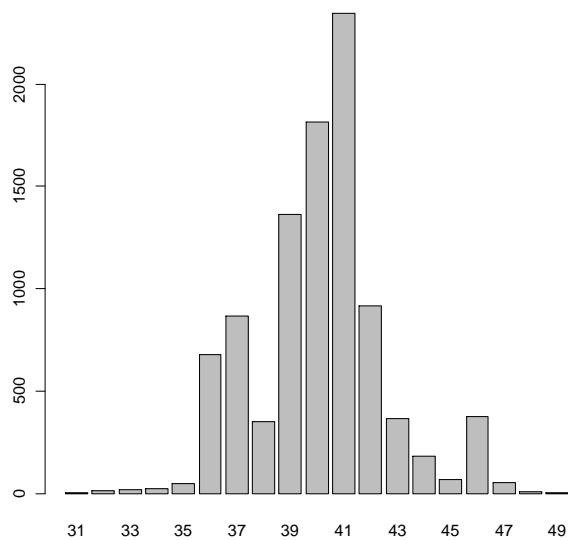


Figure 21. Samples of M

## RANDOM WALK

The last example demonstrates a random walk algorithm. This uses a proposed density with a defined method of rejecting draws in order to explore the parameter space.

A random variable  $Z$  has an *inverse Gaussian distribution* if it has density

$$f(z) \propto z^{-3/2} \exp\left\{-\theta_1 z - \frac{\theta_2}{z} + 2\sqrt{\theta_1 \theta_2} + \log(\sqrt{2\theta_2})\right\}, z > 0$$

where  $\theta_1 > 0$  and  $\theta_2 > 0$  are parameters.

Let  $\theta_1 = 1.5$  and  $\theta_2 = 2$ . Draw a sample using the random-walk Metropolis method. Since  $z > 0$  we cannot just use a Normal density, so use  $W = \log(Z)$ . Using change of variables, we have

$$f(w) \propto e^w e^{-3w/2} \exp\left\{-\theta_1 e^w - \frac{\theta_2}{e^w} + 2\sqrt{\theta_1 \theta_2} + \log(\sqrt{2\theta_2})\right\}$$

This example requires us to code the undefined distribution to be sampled from, therefore we cannot use the RAND() function. This is accomplished easily with SAS/IML using the START and FINISH commands to create a module, which is a very similar idea to task of creating a function in R. If theta1 and theta2 are defined outside of the module they can be called inside with the use of the GLOBAL option in the START statement.

```
PROC IML;
START dist(w); /*dist for w = log(z)*/
  thetal=1.5;
  theta2=2;
  d=exp(-thetal*exp(w)-theta2/exp(w)+2*sqrt(thetal*theta2)+log(sqrt(2*theta2))-w/2);
  RETURN(d);
FINISH;
```

Once the module is defined we proceed with the initialization and running of the algorithm. We do not show tuning of the variance of the random walk as it is trivial to the code, so an arbitrary value is used instead for demonstration purposes. Note that the IF THEN statement is used to incorporate the probability of accepting the proposed value "star".

```
r=10000; /*number of samples to generate*/
/*Initialize*/
z=J(r,1,0);
w=J(r,1,0);
iter=J(r,1,0);
accept=0;

z[1]=1;
w[1]=log(z[1]);
iter[1]=1;

DO m=2 TO r;
  CALL RANDGEN(star,'NORMAL',w[m-1],2);
  ratio=dist(star)/dist(w[m-1]);
  rho=MIN(1,ratio);
  CALL RANDGEN(unif,'UNIFORM');
  IF rho>=unif THEN do;
    w[m]=star;
    accept=accept+1;
  END;
  ELSE w[m]=w[m-1];
  z[m]=exp(w[m]);
  iter[m]=m;
END;
rate=accept/r; /*acceptance rate, should be about 0.3*/
PRINT rate;
CREATE samples VAR {z iter}; /*save to dataset samples*/
APPEND;
CLOSE samples;

QUIT;
```

rate
0.3058

Figure 22 . Result of the random walk algorithm

The trace plot, shown in Figure 24, is an important tool for assessing convergence of the algorithm. This plot displays the value of the parameter over the course of the algorithm.

```
PROC SGPLOT DATA=samples;
  HISTOGRAM z;
RUN;

ODS GRAPHICS / ANTIALIASMAX=10000;

PROC SGPLOT DATA=samples; /*trace plot*/
  TITLE 'Diagnostic Trace Plot';
  SERIES X=iter Y=z;
  YAXIS VALUES=(0 TO 6 BY 1);
  XAXIS VALUES=(0 TO 10000 BY 2000) LABEL='iterations';
RUN;
```

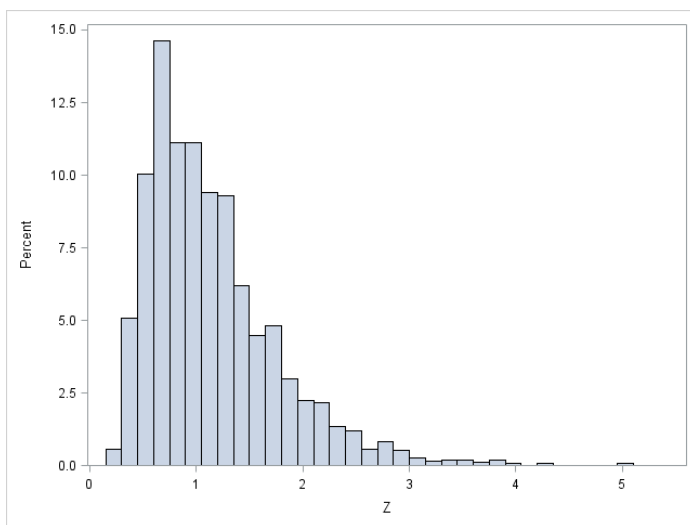


Figure 23 . Samples of Z

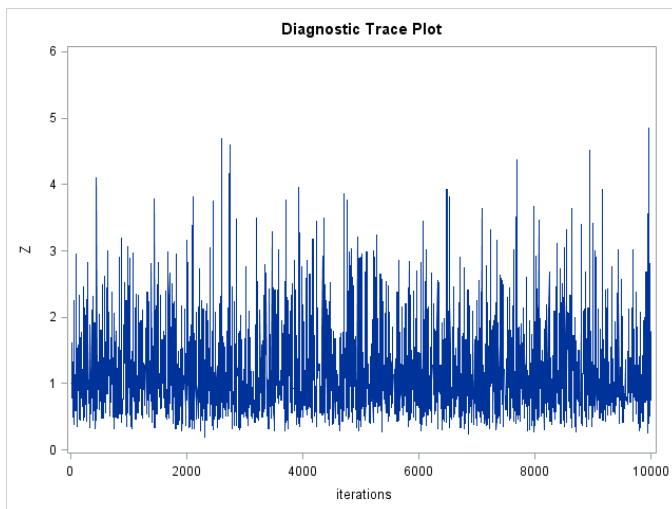


Figure 24 . Trace plot

The corresponding R code and output for this example is shown below. From both SAS and R versions we can see that SAS/IML and R operate very similarly and treat vectors in the same manner with minor differences in syntax.

```
theta1=1.5
theta2=2
w.dist<-function(w){
  dist = exp(-theta1*exp(w) - theta2/exp(w) + 2*sqrt(theta1*theta2) +
    log(sqrt(2*theta2)) - w/2)
  return(dist)
}

z=NULL
z[1]=1
w=NULL
w[1]=log(z[1])
accept=0
r=10000

for(m in 2:r){
  w.star=rnorm(1,w[m-1],2)
  ratio=w.dist(w.star)/w.dist(w[m-1])
  rho=min(1,ratio)
  unif=runif(1,0,1)
  w[m]=w[m-1]
  if(rho>=unif){
    w[m]=w.star
    accept=accept+1
  }
  z[m]=exp(w[m])
}

accept/r
[1] 0.3048

hist(z,freq=FALSE)
traceplot(as.mcmc(z))
```

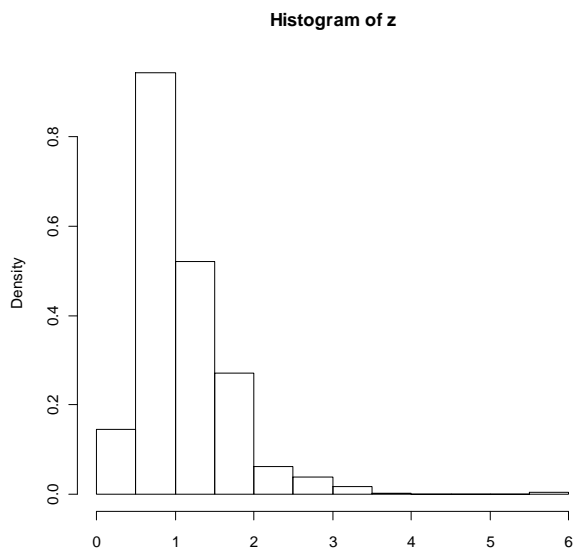


Figure 25. Sample of Z



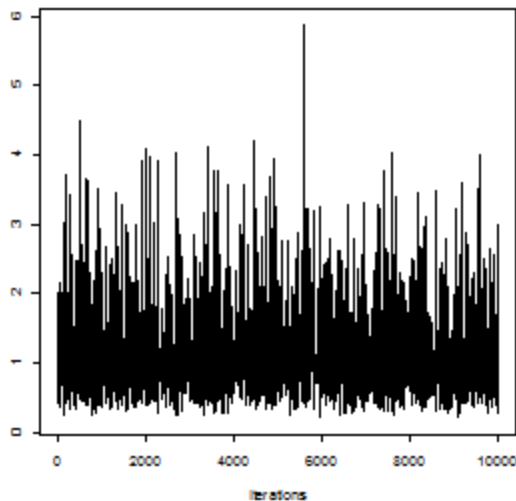


Figure 26. Trace plot

While the similarity between the SAS/IML and R code is remarkable, the simulations suggested an enormous difference in run time. Both software packages had negligible differences until approximately 10,000 repetitions and beyond. At 100,000 repetitions SAS took 0.9 seconds, while R took 1.25 seconds. At 1,000,000 repetitions SAS took 9.1 seconds, while R took 4 hours, 9 minutes, 29 seconds. MCMC algorithms are notorious for long run times so this is of high interest to statistical programmers.

C is a common alternative for R users seeking improved run times from R (see Appendix for corresponding C code for this algorithm). We find the syntax in IML more straightforward, especially for statistical programmers. Additionally, SAS/IML provides advantages of statistical software such as straightforward data management, graphics, and statistical analysis procedures. Run times in C suggest similarities to SAS, however a more in depth investigation into run times would need to take place for further conclusions.

## CONCLUSION

For Bayesian statistics PROC MCMC is a powerful tool, however the ability for users to write their own MCMC algorithms is a relatively unknown and undocumented topic in SAS literature. We have shown that SAS has strong capabilities for users to code their own algorithms and provided a guide to carrying these out. Examples of common algorithms such as a Gibbs sampler and a random-walk algorithm were demonstrated using SAS/IML. Random number generation and utilizing probability distributions is trivial through both the SAS DATA step and SAS/IML. Methods used to code the samplers were straightforward and did not require large amounts of complicated code. The syntax of these algorithms is simple and very closely related to R and displayed improved runs times compared to R. Due to the sluggish nature of R and the computationally intensive nature of MCMC algorithms, SAS presents a powerful punch.

## APPENDIX

Below is C code for the random walk algorithm.

```
double sample_w(double w){
double theta1=1.5;
double theta2=2;
double dist;

dist = exp(-theta1*exp(w)-theta2/exp(w)+2*sqrt(theta1*theta2)+log(sqrt(2*theta2))-w/2);

return(dist);
}

int main(void)
{
```

```
int MM    = 10000;    //total number iterations

double z=1;
double w=log(z);
double star;
double ratio;
double rho;
double unif;
int accept=0;

FILE *fp_z;
fp_z=fopen("Z.txt","w");

int m;
for(m=0; m<MM; m++){
  star = gsl_ran_gaussian(r,2) + w;
  ratio = sample_w(star)/sample_w(w);
  rho = min(1,ratio);
  unif = gsl_ran_flat(r,0,1);
  if(rho>=unif){
    w = star;
    accept = accept + 1;
  }
  z = exp(w);

  if(m>=burn){
    if(m % thin == 0){
      fprintf(fp_z,"%f \r\n",z);
    }
  }

}

printf("Accepted: %i, Out of: %i", accept, MM);

fclose(fp_z);
printf("\n");
return 0;
}
```

## REFERENCES

- Carlin, BP., Gelfand AE., and Smith AFM. 1992. "Hierarchical Bayesian analysis of changepoint problems." Applied Statistics: 389-405.
- Lofland, C. and Ottesen, R. 2013. "Simulations in SAS with Comparisons to R." Western Users of SAS Software 2013 Proceedings. Available at: [wuss.org/Proceedings13/128\\_Paper.pdf](http://wuss.org/Proceedings13/128_Paper.pdf)
- Lofland, C. and Ottesen, R. 2014. "Using SAS to do it all." Western Users of SAS Software 2014 Proceedings.
- sasCommunity.org. "How the SAS Random Number Generators Work." April 2009. Available at: [http://www.sascommunity.org/wiki/How\\_the\\_SAS\\_Random\\_Number\\_Generators\\_Work](http://www.sascommunity.org/wiki/How_the_SAS_Random_Number_Generators_Work)
- Support.sas.com. "RAND function." September 2013. Available at: <http://support.sas.com/documentation/cdl/en/lrdict/64316/HTML/default/viewer.htm#a001466748.htm>
- Support.sas.com. "Example 52.11 Implement a New Sampling Algorithm." September 2013. Available at: [http://support.sas.com/documentation/cdl/en/statug/63033/HTML/default/viewer.htm#statug\\_mcmc\\_sect053.htm](http://support.sas.com/documentation/cdl/en/statug/63033/HTML/default/viewer.htm#statug_mcmc_sect053.htm)
- WikiBooks. "R Programming/Random Number Generation." November 2011. Available at: [http://en.wikibooks.org/wiki/R\\_Programming/Random\\_Number\\_Generation](http://en.wikibooks.org/wiki/R_Programming/Random_Number_Generation)

## ACKNOWLEDGEMENTS

Thank you to Becky Ottesen for the endless guidance, support, and help.

Thank you to Leanne Goldstein and Annalisa Cadonna for helping.

## RECOMMENDED READING

Lofland, C. and Ottesen, R. 2013. "Simulations in SAS with Comparisons to R." Western Users of SAS Software 2013 Proceedings. Available at: [wuss.org/Proceedings13/128\\_Paper.pdf](http://wuss.org/Proceedings13/128_Paper.pdf)

Wicklin R. April 2013. "*Simulating Data with SAS*." Cary, NC: SAS Institute.

## CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Chelsea Loomis Lofland  
UC Santa Cruz, Department of Applied Math and Statistics  
1156 High St.  
Santa Cruz, CA 95060  
E-mail: [Chelsea.Loomis.Lofland@gmail.com](mailto:Chelsea.Loomis.Lofland@gmail.com)

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.