Paper 9080-2016

MCMC in SAS®: From Scratch or by PROC

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

ABSTRACT

Markov chain Monte Carlo (MCMC) algorithms are an essential tool in Bayesian statistics for sampling from various probability distributions. Many users prefer to use an existing procedure to code these algorithms, while others prefer to write an algorithm from scratch. We demonstrate the various capabilities in SAS® software to satisfy both of these approaches. In particular, we first illustrate the ease of using the MCMC procedure to define a structure. Then we step through the process of using SAS/IML® to write an algorithm from scratch, with examples of a Gibbs sampler and a Metropolis-Hastings random walk.

INTRODUCTION

Markov chain Monte Carlo (MCMC) algorithms are used to sample from probability distributions and are a central component to Bayesian analysis. The capabilities of SAS to code these algorithms is somewhat unknown in the Bayesian community, partly due to underrepresentation and lack of discussion in this area.

MCMC algorithms have made a grand appearance into the world of SAS through PROC MCMC. Although PROC MCMC is a very useful tool, coding an MCMC from scratch is a desired ability 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.

We present tools in SAS used to code an MCMC algorithm, followed by three examples of algorithms done both by hand and with PROC MCMC. These tools will cover simulation, sampling, and Bayesian techniques using SAS 9.3, SAS/IML 12.1. Our focus will be to discuss the SAS coding examples using functions and call subroutines, SAS/IML, and PROC MCMC. Comparable R (version 2.13.2) and C code will also be provided for the third example in the appendix as a reference. R code for the other examples is available through the author if requested.

TOOLS FOR MCMC CODING

We will first discuss tools that are useful when coding an MCMC algorithm.

SAS/IML

Coding an MCMC algorithm from scratch requires a 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 Interactive Matrix Language (IML), which is designed well for matrix operations.

This environment behaves quite differently from the SAS DATA step and other PROCs. The term “interactive” refers to the ability to run code line by line during the session. The PRINT statement then allows the user to view objects while still in the session. In academia this monitoring environment is desired for writing coding examples from scratch. As the name suggests, IML is designed to handle vectors and matrices rather than datasets, making the syntax of IML intuitive and familiar to users of other software such as R and Matlab.

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 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.

When using the RAND() function users can 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. A benefit of the RAND() function is the use of the distribution name as an argument. This simplicity is advantageous over using a different function for every distribution.

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 1. Random numbers from the standard normal distribution

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 2. 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(α,1) so for a Gamma(α, β) use the property: β*Gamma(α,1) = Gamma(α, β) for any β>0.

Another option for generating random numbers 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.

```
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 3. 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 p will return the smallest value for which the CDF of that value is greater than or equal to p.

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 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 μ and scale parameter σ. 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 4. This shows that μ is maximized at approximately 0.06 and σ 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 4. Optimization results for μ and σ

PROC MCMC

Syntax for PROC MCMC is as follows.

```
PROC MCMC options ;
ARRAY array specification ;
BEGINCNST/ENDCNST ;
BEGINNODATA/ENDNODATA ;
BY variables ;
MODEL statistical model specification ;
PARMS parameters and starting values ;
PRIOR/HYPERPRIOR prior or hyperprior specification ;
Program statements ;
UDS user defined sampler specification ;
```

SAS documentation has the usual depths of information on these options, but there are a few things worth noting. The PROC MCMC statement defines options such as burn-in, thinning, seed, and number of iterations. Each use requires the three components that form the base of a Bayesian model: PARMS for the parameters to sample, PRIOR for the prior distribution, and MODEL for the likelihood. Program statements allow for code such as IF/THEN/ELSE and DO loops. Additionally, note the ability to include an ARRAY in the PROC. The BEGINCNST/ENDCNST and UDS are powerful tools used for complex algorithms beyond the scope of this paper.

PROC MCMC is an efficient choice due to its ability to use the appropriate sampling method based on the model specified. Options for it to use are direct sampling, conjugate sampling, slice sampler, and various Metropolis algorithms (Chen 2013). This is of special interest due to the computational demand of some MCMC's.

EXAMPLE ALGORITHMS

This arsenal of tools can now be used to run some MCMC algorithms. We have also included example SAS/IML code for processes that are commonly performed, such as a trace plot to assess convergence and subsetting samples based on burn-in and thinning.

EXAMPLE 1. CAUCHY DISTRIBUTION

Consider again the data distributed as a Cauchy with location μ and scale σ. We want to obtain posterior samples of these 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.

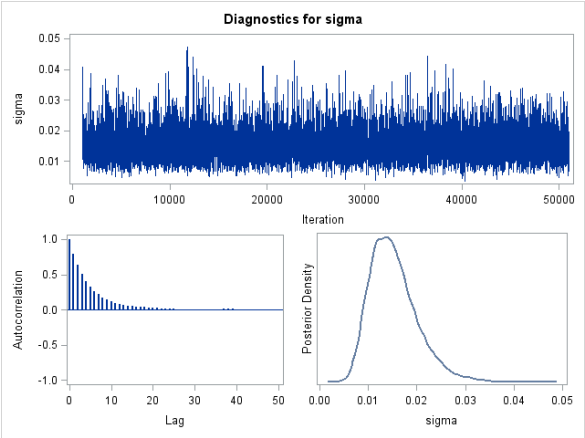
When defining a non-informative prior such as $1/\sigma$, we code the prior distribution as an element, in our case lp. Since σ cannot be negative, we implement that constraint when defining lp. Note that PROC MCMC uses distributions on the log scale, so $1/\sigma$ becomes $-\log(\sigma)$. Once this distribution is defined, we insert it into GENERAL in the PRIOR statement.

We specify 50,000 iterations and a new dataset postout1 to save the posterior samples, initialize μ and σ to 1, and simulate 1000 posterior predictive samples saved in predout1.

```
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;

/*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 of posterior statistics and convergence diagnostics, but we only include a subset below in Figure 5. This displays the density estimates 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 plot does not show a large amount of correlation, both of which are good.



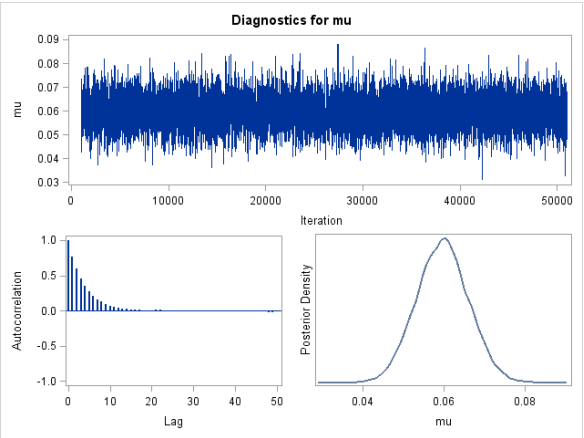


Figure 5. Results from PROC MCMC

If coding this problem outside PROC MCMC, an alternative method is to use a scale mixture of Normals so that the full conditionals can be defined and a Gibbs sampler can be used. This approach can simplify coding from scratch by eliminating the need for sampling from a Cauchy distribution.

We use latent variables λ_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 acheive a $Cauchy(x|\mu,\sigma)$.
With this model we have

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

and the full conditionals are

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

with i=1,...,16.
This can be used to implement a Gibbs sampler to obtain samples from μ and σ.

The following code implements this method using SAS/IML instead of PROC MCMC. We first create an object containing the data and initialize the parameters. Brackets define a matrix of individual values, whereas J(a,b,c) simply creates an entire matrix with a rows, b columns, and elements c. When using brackets, a comma indicates a new row and the values between commas indicate columns. The object y is a column vector, but could instead be a row vector if the commas were removed.

```
PROC IML;
y={0.09,0.07,0.08,0.05,0.06,0.09,0.07,0.09,0.05,0.07,0.06,0.05,0.03,0.0
4,0.03,0.04};
n = NROW(y);

niter = 50000;
burn = 2000;

musave = J(niter-burn,1,0);
sigmasave = J(niter-burn,1,0);

mu=1;
sigma=1;
lambda=J(1,n,1);
```

This demonstrates setting up a burn-in, however note that the value chosen is for illustration purposes. We can now run the algorithm, print the posterior means, and plot the samples we obtained. The syntax for a loop in IML matches the syntax that would be used in a DATA step.

```
DO i=1 TO niter;

  DO j=1 TO n;
    beta = (1+((y[j]-mu)##2)/sigma)/2;
    lambda[j] = RAND('GAMMA',1)/beta;
  END;

  mean = lambda*y / SUM(lambda);
  sd = sigma/SUM(lambda);
  mu = RAND('NORMAL',mean,sd);

  a = n/2;
  b = SUM((y-mu)##2*lambda)/2;
  sigma = 1/(RAND('GAMMA',a)/b);

  IF(i > burn) THEN DO;
    musave[i-burn] = mu;
    sigmasave[i-burn] = sigma;
  END;

END;

mumean = mean(musave);
smean = mean(sigmasave);

PRINT mumean;
PRINT smean;
```

mumean
0.0605818

smean
0.0146089

```
CREATE samples VAR {musave sigmasave};
APPEND;
CLOSE samples;

QUIT;

PROC SGPLOT DATA=samples;
  HISTOGRAM musave;
RUN;
PROC SGPLOT DATA=samples;
  HISTOGRAM sigmasave;
RUN;
```

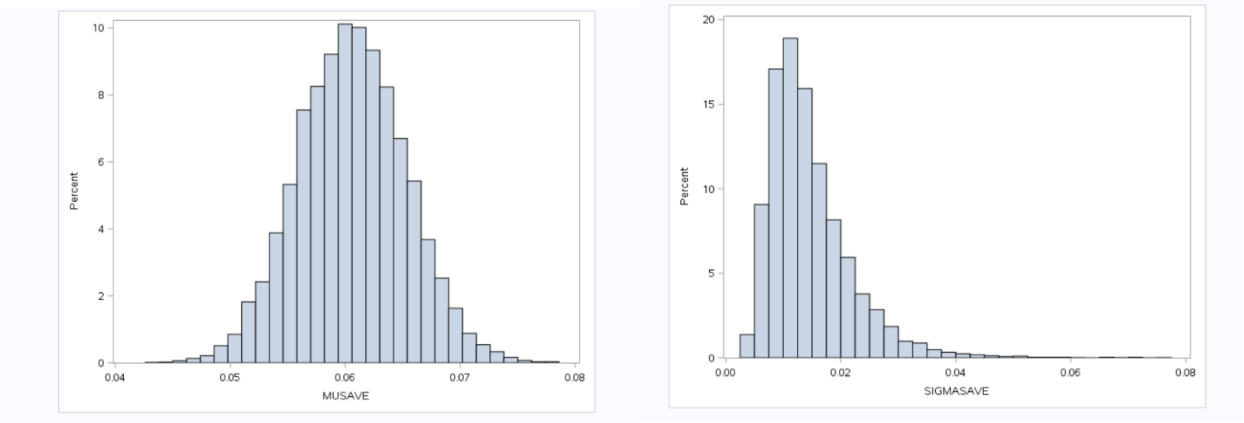


Figure 6. Results using latent variables

This approach can also be done by implementing a new sampling algorithm with PROC MCMC (support.sas.com 2013), however we feel this is unnecessary given the availability of the Cauchy distribution in PROC MCMC.

The next two examples are MCMC algorithms that are frequently employed – a Gibbs sampler and a Metropolis-Hastings random walk. 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.

EXAMPLE 2. CHANGE POINT

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 $y_1, ..., y_n$ 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, $m = 1, ..., n$. Given $m, y_i \sim Poisson(\theta)$, for $i = 1, ..., m$ and $y_i \sim Poisson(\phi)$, for $i = m+1, ..., n$. The model is completed with independent prior distributions $\theta \sim Gamma(\alpha, \beta), \phi \sim Gamma(\gamma, \delta)$ and m uniformly distributed over 1,...,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 consits 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 θ and ϕ as

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

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

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 $1, ..., n$.

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

```
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;
```

The following SAMPLE() function requires only weights for sampling, not probabilities, meaning the values do not need to add to 1. The draws from the Gamma distribution follow from the parameterization with mean α/β.

```
/*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 θ, ϕ, and m as shown in Figures 7 and 8.

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

PROC SGPLOT DATA=samples;
 VBAR m;
RUN;
```

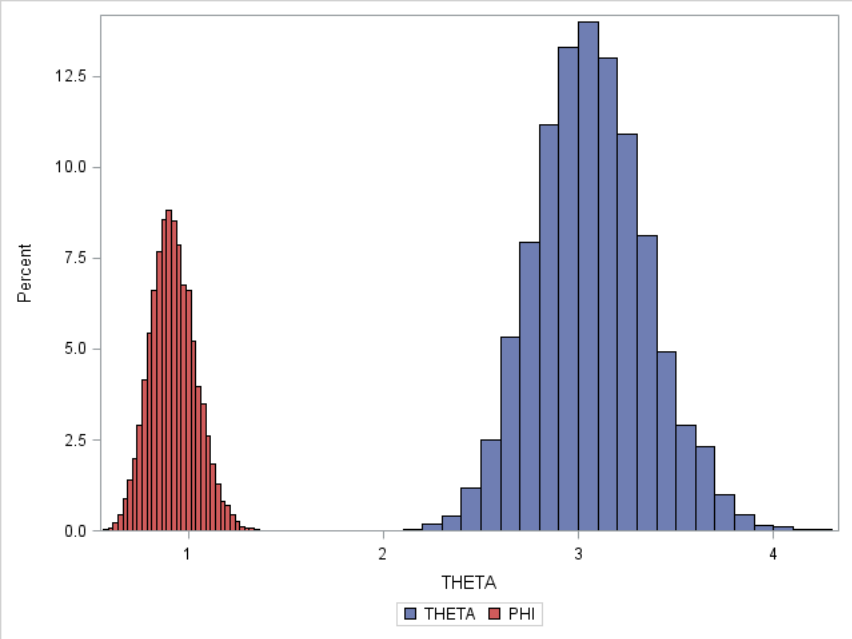


Figure 7. Samples of θ and ϕ

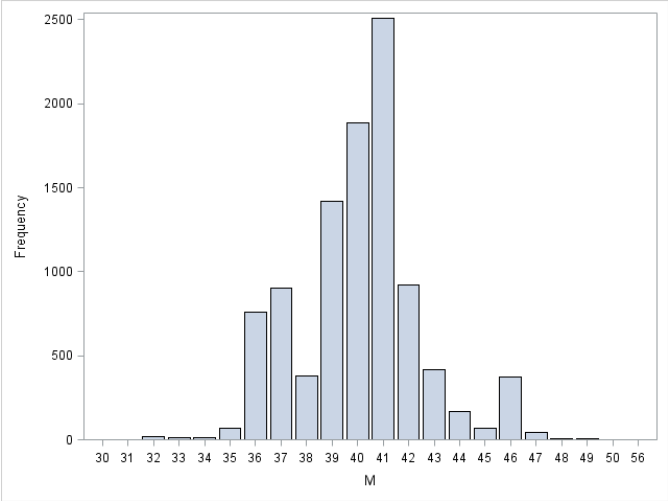


Figure 8. Samples of m

Now that we are satisfied with our code and results, let's check it with PROC MCMC. The use of DGENERAL() defines the prior on m as discrete, compared with the continuous GENERAL().

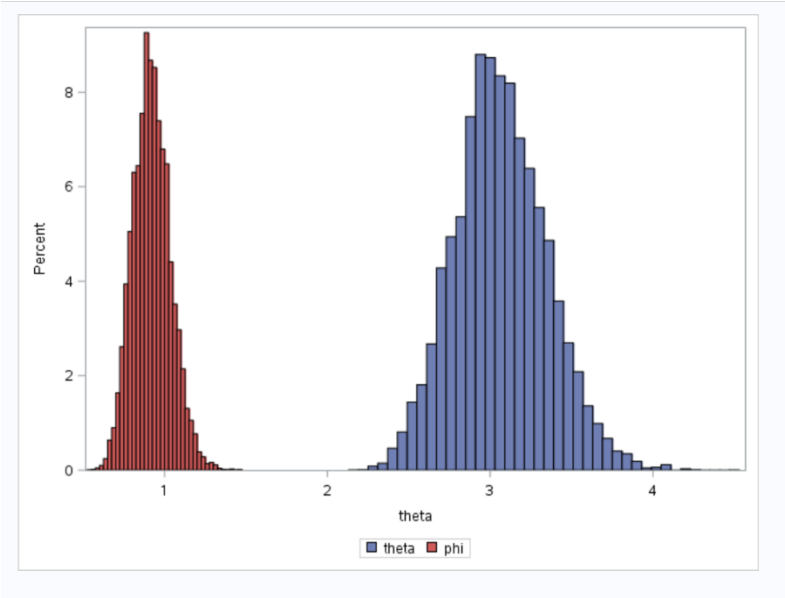
```
DATA change;
ind = _N_;
INFILE DATALINES delimiter=',';
INPUT y @@;
DATALINES;
4,5,4,1,0,4,3,4,0,6,3,3,4,0,2,6,3,3,5,4,5,3,1,4,4,1,5,5,3,4,2,5,2,2,3,4
,2,1,3,2,2,1,1,1,1,3,0,0,1,0,1,1,0,0,3,1,0,3,2,2,0,1,1,1,0,1,0,1,0,0,0,
2,1,0,0,0,1,1,0,2,3,3,1,1,2,1,1,1,1,2,4,2,0,0,0,1,4,0,0,0,1,0,0,0,0,0,1
,0,0,1,0,1
;
RUN;

PROC MCMC DATA=change NMC=25000 OUTPOST=posterior;
  PARMS theta phi;
  PARMS m 50;

  PRIOR m ~ DGENERAL(1, LOWER=1, UPPER=100);
  PRIOR theta phi ~ GAMMA(1, SCALE=1);

  indic = (ind <= m); /*indicator that returns 1 or 0*/
  mu = indic*theta + (1-indic)*phi;
  MODEL y ~ POISSON(mu);
RUN;
```

```
PROC SGPLOT DATA=posterior;
  HISTOGRAM theta;
  HISTOGRAM phi;
RUN;
PROC SGPLOT DATA=posterior;
  VBAR m;
RUN;
```



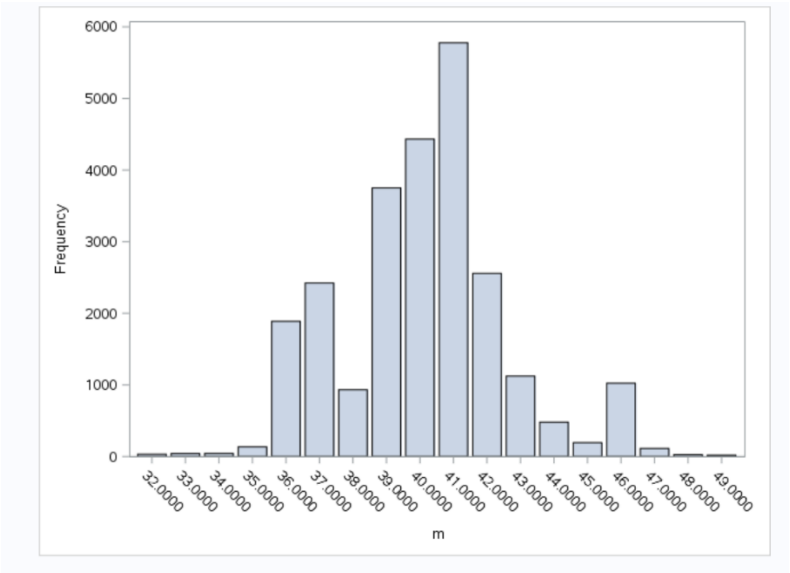


Figure 9. Change point results using PROC MCMC

We can see that both approaches lead to the same conclusions, and PROC MCMC is a useful tool to check results against with simple coding.

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

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

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

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

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. If θ_1 and θ_2 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)*/
  theta1=1.5;
  theta2=2;
  d=exp(-theta1*exp(w)-
theta2/exp(w)+2*sqrt(theta1*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 for the proposal variance 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;
```

rate
0.3058

```
CREATE samples VAR {z iter};  /*save to dataset samples*/
APPEND;
CLOSE samples;

QUIT;
```

The trace plot, shown in Figure 10, 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;
```

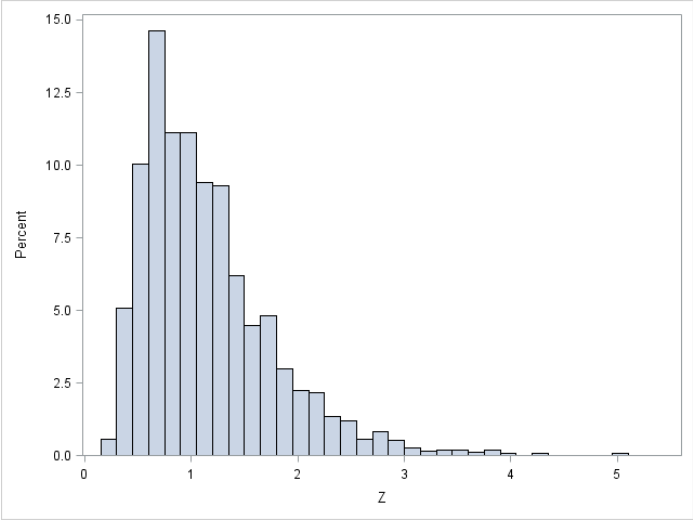


Figure 10. Samples of Z

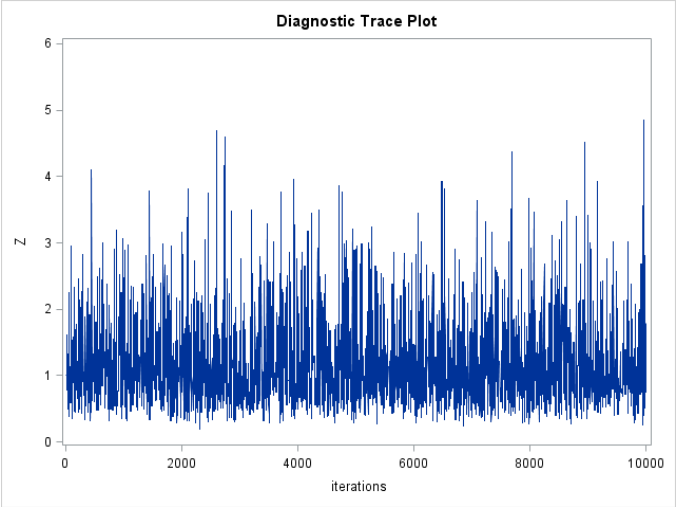


Figure 11. Trace plot

Again, we can check our results with PROC MCMC. Since we are not coding the random walk ourselves, we no longer need to transform z to have the same support as a Normal distribution. We then simply work directly with the distribution of z. With no likelihood, we enter GENERAL(0) in the MODEL statement.

```
DATA temp; run;
PROC MCMC data=temp nmc=10000;
  theta1 = 1.5;
  theta2 = 2;

 PARMS z 1;
 IF z>0 then lp = -3*log(z)/2 - theta1*z - theta2/z +
2*sqrt(theta1*theta2) + log(2*theta2)/2;
 ELSE lp=.;

 PRIOR z ~ GENERAL(lp);
 MODEL GENERAL(0);
RUN;
```

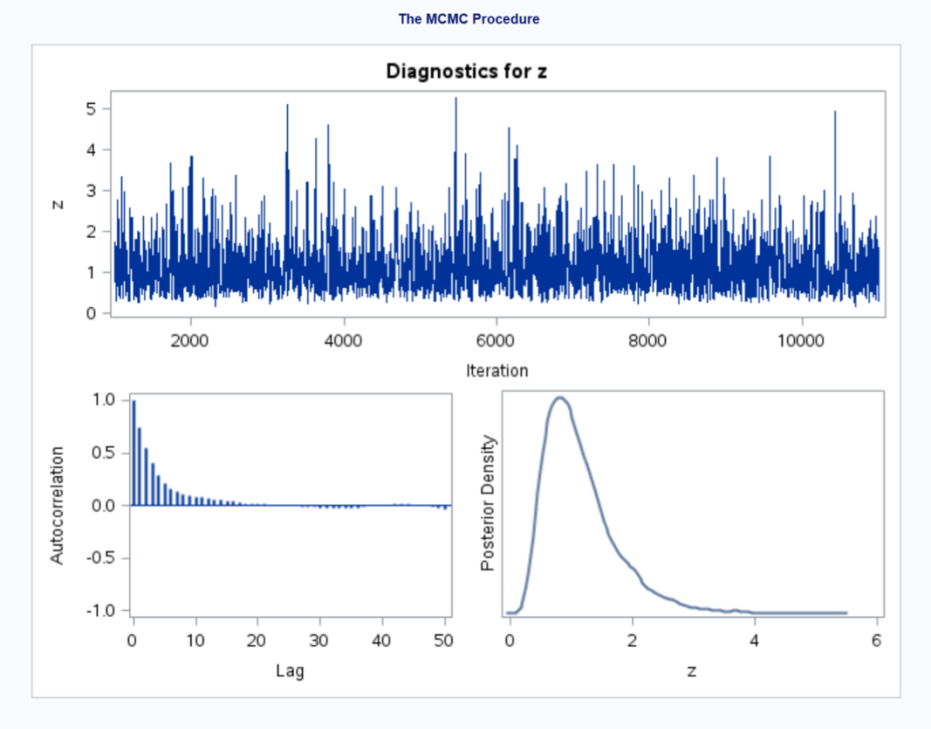


Figure 12. Inverse Gaussian results with PROC MCMC

Again, we see PROC MCMC is a tool both for use on its own or and as a simpler method to check an algorithm coded by hand.

COMPARISONS

Through preliminary work, the simulations suggested an enormous difference in run time between SAS and R, a software commonly used for MCMC's. 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 the third algorithm). We find the syntax in SAS/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. It is worthwhile to suggest that coders working from scratch can still use PROC MCMC as a sanity check for their results. We have shown that SAS has strong capabilities for users to code their own algorithms and provided a guide to carrying these out. Methods used to code the samplers were straightforward and did not require large amounts of complicated code. Although R is a common choice for Bayesians, the syntax of the SAS algorithms is very similar and had the added benefit of largely improved runs times compared to R. Additionally, the code in SAS is a much simpler option for speedy runs than C. Due to the sluggish nature of R, the complexity of C code, and the computationally intensive nature of MCMC algorithms, SAS presents a powerful punch.

APPENDIX

RANDOM WALK C CODE

```
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;
}
```

RANDOM WALK R CODE

```
library(MCMCpack)

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])
}

print(accept/r)

layout(matrix(c(1,1,2,3), 2, 2, byrow = TRUE))
traceplot(as.mcmc(z))
acf(z)
hist(z,freq=FALSE)
```

REFERENCES

Carlin, BP., Gelfand AE., and Smith AFM. 1992. "*Hierarchical Bayesian analysis of changepoint problems*." Applied Statistics: 389-405.

Chen, F. 2013. “Introduction to the MCMC Procedure in SAS/STAT Software.” SAS Global Forum 2013 Proceedings.

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

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

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

ACKNOWLEDGMENTS

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

Thank you to Fang Chen for guidance on PROC MCMC.

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

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