STA3431-Monte_Carlo_Methods-Homework_2

Name: Sergio E. Betancourt
Student Number: 998548585
Department: Statistical Sciences
Program: M.Sc. 2019
E-mail: sergio.betancourt@mail.utoronto.ca

Date: 2018-11-12

Question 1

Let A=8, B=5, C=8, and D=5, given that my student number is 998548585. Moreover, let $h(X_1, X_2, X_3, X_4, X_5) = \frac{X_1 - X_2}{2 + X_3 + X_4 * X_5}$, and consider $g(x_1, x_2, x_3, x_4, x_5)$, and $\pi(x_1, x_2, x_3, x_4, x_5) = c\, g(x_1, x_2, x_3, x_4, x_5)$, as given in the problem statement.

Below I use the **a. Metropolis** and **b. Independence Sampler** algorithms to estimate $E_\pi[\frac{(X_1 - X_2)}{(2 + X_3 + X_4 * X_5)}]$.

First, the stage is set:

```r
A=8; B=5; C=8; D=5;

g <- function(x){
  if((x[1] <= 0) | (x[1] >= 1) |
     (x[2] <= 0) | (x[2] >= 1) |
     (x[3] <= 0) | (x[3] >= 1) |
     (x[4] <= 0) | (x[4] >= 1) |
     (x[5] <= 0) | (x[5] >= 1)) {
    return(0)
  } else {
    return( ((x[1]+A+2)^(x[2]+3)) * (1+cos(2*x[2] + 3*x[3] + 4*x[4] + (B+3)*x[5])) *
              exp((12-C)*x[4]) * (abs(x[4] - 3*x[5])^(D+2)) )
  }
}

h <- function(x){
  return((x[1]-x[2])/(2+x[3]+(x[4]*x[5])))
}

set.seed(1234);
### Fixing starting value for both methods to maintain consistency
X <- c(runif(1),runif(1),runif(1),runif(1),runif(1));
```

a. Metropolis

For this algorithm I choose to propose from a vector of 5 uniform distributions, all centered at the vector X: $Y \sim Unif(X \pm \frac{1}{5})$. I considered proposing from a vector of also-symmetric standard normals, centered at X, but uniform proposals trump them here given how the standard normal has support over the entire real line, regardless of the magnitude of the scale parameter. Therefore, proposing from the standard normal could possibly lead to a lower acceptance rate.

Note how normalizing constants cancel out and we can use $A_i = \frac{\pi(Y_i)}{\pi(X_{i-1})} = \frac{g(Y_i)}{g(X_{i-1})}$ for our accept/reject step.

```r
#Metropolis Algo
set.seed(1234);
#Timing this algo
ptm <- proc.time()
```

```r
#Iterations and burn-in
M <- 100000; B <- 15000

x1list = x2list = x3list = x4list = x5list = halist = numeric(M);
accepted = 0;

for (i in 1:M) {
    Y = X + runif(5,-1/5,1/5)  # proposing from x +- runif(0,1/5)
    U = runif(1)  # for accept/reject
    alpha = g(Y) / g(X)  # for accept/reject
    if (U < alpha) {
      X = Y  # accept proposal
      accepted= accepted + 1;
    }
    x1list[i] = X[1]; x2list[i] = X[2]; x3list[i] = X[3];
    x4list[i] = X[4]; x5list[i] = X[5];
    halist[i] = h(X);
};
timer1 <- proc.time() - ptm;
```

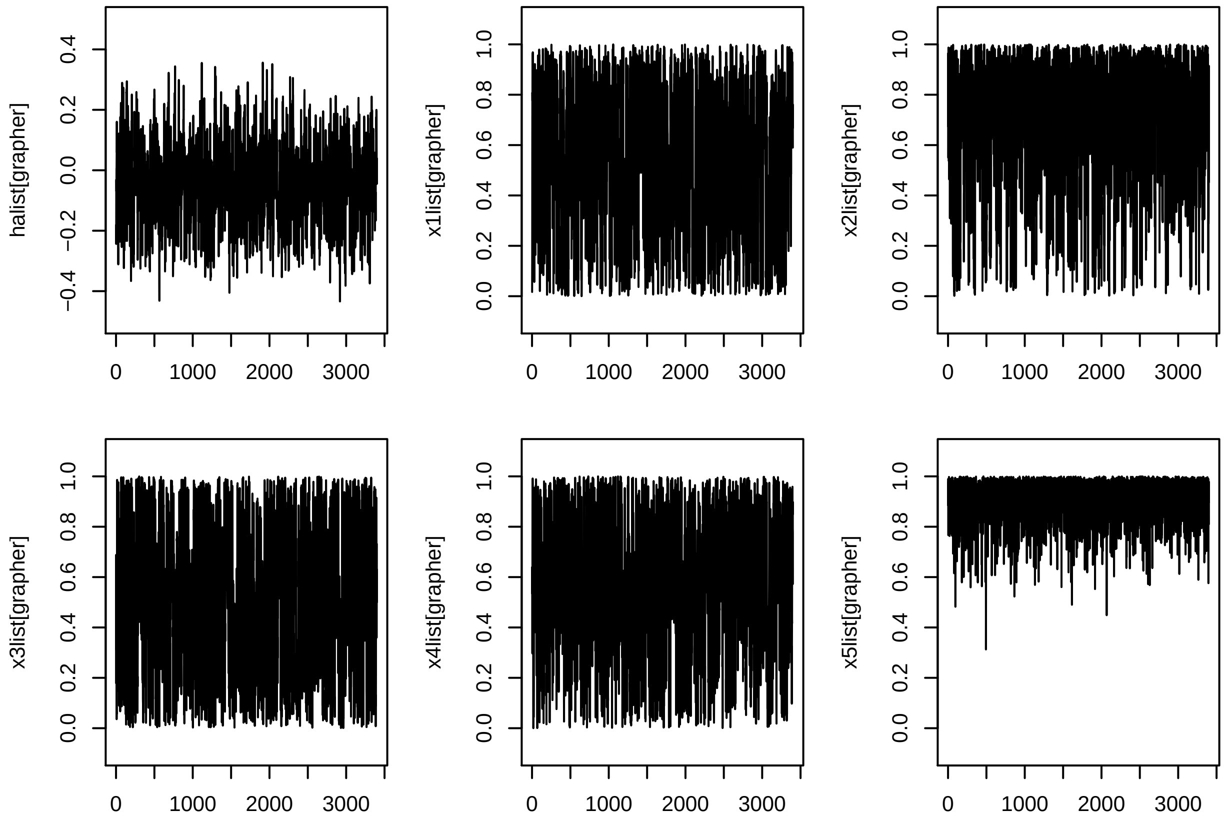
```r
cat(M, "iterations of the Metropolis algorithm, with burn-in", B, "\n");
cat("Running time:","\n"); timer1;
cat("acceptance rate =", accepted/M, "\n");
ua = mean(halist[(B+1):M]);
cat("mean of h is about", ua, "\n")

sea = sd(halist[(B+1):M]) / sqrt(M-B);
cat("iid standard error would be about", sea, "\n");

varfact <- function(xxx) { 2 * sum(acf(xxx, plot=FALSE)$acf) - 1 }
thevarfacta = varfact(halist[(B+1):M]);
sea = sea * sqrt( thevarfacta );
cat("Varfact = ", thevarfacta, "\n");
cat("True standard error is about", sea, "\n");
cat("Approximate 95% confidence interval is (", ua - 1.96 * sea, ",",
                                                ua + 1.96 * sea, ")\n\n");

#Saving values of h to show ACF in discussion section, along with ACF for Independence Sampler
metrovec <- halist[(B+1):M]
#acfplotmet <- acf(hlist[(B+1):M],main="ACF of Metropolis");
```

```
## 1e+05 iterations of the Metropolis algorithm, with burn-in 15000
## Running time:
##    user  system elapsed
##   3.725   0.071   3.960
## acceptance rate = 0.18504
## mean of h is about -0.05229475
## iid standard error would be about 0.0004460101
## Varfact =  69.02864
## True standard error is about 0.003705607
## Approximate 95% confidence interval is ( -0.05955774 , -0.04503176 )
```

Not only is the resulting estimate and associated metrics comparable to both methods from assignment 1 (same M and seed), this algorithm yields a significantly lower run-time. This allows me to increase the number of iterations in order to improve the algorithms' respective accuracies. Nonetheless, I chose not to manipulate further the number of iterations or the scale of the proposal as it can easily turn into a never-ending game of finding the optimal combo.

Given how this is an MCMC method, the varfact component must now be considered when calculating the variance of our estimates. This was not the case before when we were looking at iid sampling.

The below plots shows good mixing of the estimate of $h(X_1, X_2, X_3, X_4, X_5)$, as well as good mixing for the values of the vector X[1:3]. The indeces only go up to 4000 because I am plotting the 25th points in the series B+1 to M in order to speed up the rendering of my PDF report.



b. Independence Sampler

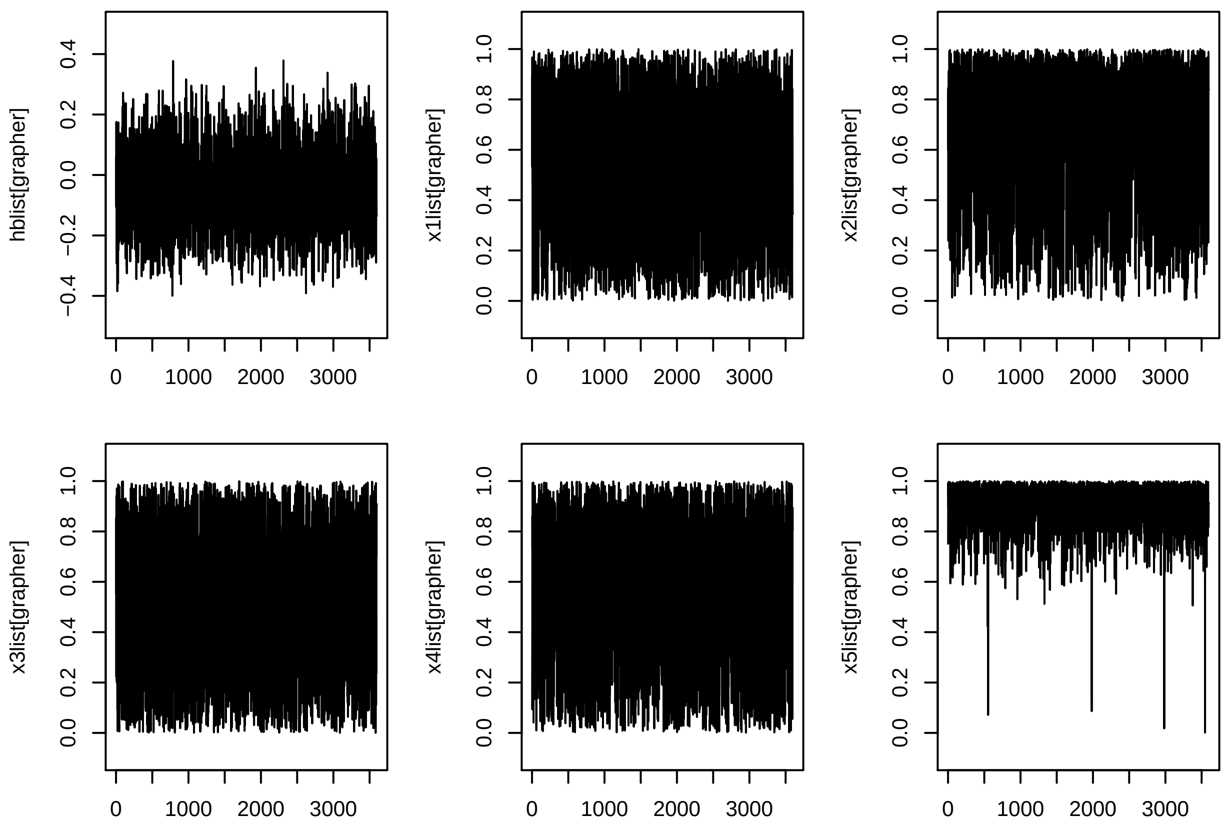
Here I use the Independence Sampler algorithm, proposing from a vector of 5 uniform distributions on [0,1]. Note that the proposals are made from a fixed density, independent of the value of X, and for all $X\ ,Y\ : q(X, Y) = 1$. Moreover, note how normalizing constants cancel out and we can use $A_i = \frac{\pi(Y_i)}{\pi(X_{i-1})} = \frac{g(Y_i)}{g(X_{i-1})}$ for our accept/reject step.

```r
ptm <- proc.time()
set.seed(1234);
#Independent Sampler
#Iterations and burn-in
M <- 100000
B <- 10000

x1list = x2list = x3list = x4list = x5list = hblist = rep(0,M)
accepted = 0;

for (i in 1:M) {
    Y = runif(5,0,1)  # proposing from independent 5-vector of unif(0,1)
    U = runif(1)  # for accept/reject
    alpha = g(Y) / g(X)  # for accept/reject
    if (U < alpha) {
      X = Y  # accept proposal
      accepted= accepted + 1;
    }
    x1list[i] = X[1]; x2list[i] = X[2]; x3list[i] = X[3];
    x4list[i] = X[4]; x5list[i] = X[5];
    hblist[i] = h(X);
};
timer2 <- proc.time() - ptm;
```

```
cat(M, "iterations of the Independence Sampler algorithm, with burn-in", B, "\n");
cat("Running time:","\n");timer2;

cat("acceptance rate =", accepted/M, "\n");
ub = mean(hblist[(B+1):M]);
cat("mean of h is about", u, "\n")

seb = sd(hblist[(B+1):M]) / sqrt(M-B);
cat("iid standard error would be about", seb, "\n");

varfact <- function(xxx) { 2 * sum(acf(xxx, plot=FALSE)$acf) - 1 }
thevarfactb = varfact(hblist[(B+1):M]);
seb = seb * sqrt( thevarfactb );
cat("Varfact = ", thevarfactb, "\n");
cat("True standard error is about", seb, "\n");
cat("Approximate 95% confidence interval is (", ub - 1.96 * seb, ",",
                                                ub + 1.96 * seb, ")\n\n");

indepvec <- hblist[(B+1):M]
#acfplotind <- acf(hlist[(B+1):M],main="ACF of Independence Sampler");
```

```
## 1e+05 iterations of the Independence Sampler algorithm, with burn-in 10000
## Running time:
##    user  system elapsed
##   3.762   0.068   3.992
## acceptance rate = 0.10831
## mean of h is about -0.05175471
## iid standard error would be about 0.0004291336
## Varfact =  21.38796
## True standard error is about 0.001984619
## Approximate 95% confidence interval is ( -0.05564456 , -0.04786485 )
```

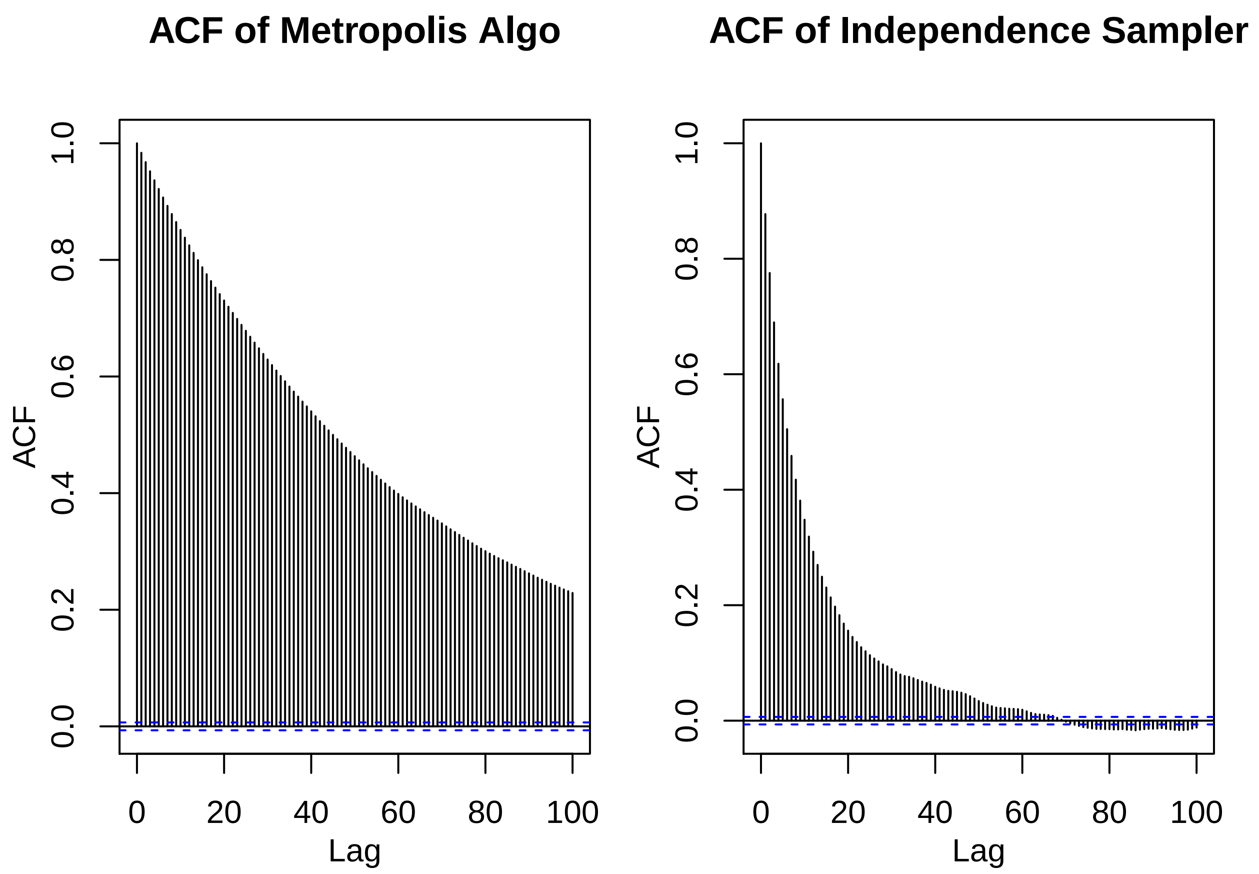
As in the previous set of graphs, the indeces only go up to 4000 because I am plotting the 25th points in the series B+1 to M in order to speed up the rendering of my PDF report.



Discussion

Although the acceptance rate is lower (11%) for this method than the previous method (approx. 18%), the varfact component is considerably smaller (approx. 20 compared to approx. 70). This makes sense given how the distribution of my proposal is independent from the previous value, and thus the correlation between moves is smaller (see below plots). This affects the calculated "true" standard error, which is smallest for the independence sampler.

```r
#ACF plots here
par(mfrow=c(1,2), mai = c(.6, .6, .6, .2),mgp=c(2,1,0));
acf(metrovec,lag.max=100,main="ACF of Metropolis Algo");
acf(indepvec,lag.max=100,main="ACF of Independence Sampler");
```



Accuracy-wise, the final estimates of these two methods are very close to each other, and their run-time is very similar. As per the below difference in "true" standard error, the Independence Sampler yielded the narrowest confidence interval. However, I can not decisively choose one method over the other in this case given how I was unable to compute the mean value of h numerically. What I can observe again is the much faster run-time of these two MC algorithms compared to the Rejection and Importance Sampler. But what I can also observe is the fact that the Importance Sampler yielded narrower confidence intervals (varfact needs not be accounted there). In reality, accuracy-wise, the importance sampler may be deemed superior in the context of this problem because it allows direct sampling from the target distribution.

row.names....names2	meanh	varfact	truese	runtime
Metropolis	-0.0522948	69.02864	0.0037056	3.960
Independence	-0.0517547	21.38796	0.0019846	3.992

Question 2

Consider the standard variance components model described in lecture, with K=6, $J_i = 5$, and $\{Y_{ij}\}$, in the context of the dyestuff data.

```
# Loading the data
# Defined so Ydye[i,j] equals yield (in grams) from j'th sample of i'th batch.
# Valid for i=1,2,3,4,5,6, and j=1,2,3,4,5, i.e. K=6 and J=5.
library(invgamma);
```

```
## Warning: package 'invgamma' was built under R version 3.2.5
```

```
Ydye = t( matrix(
    c(1545, 1440, 1440, 1520, 1580,
        1540, 1555, 1490, 1560, 1495,
        1595, 1550, 1605, 1510, 1560,
        1445, 1440, 1595, 1465, 1545,
        1595, 1630, 1515, 1635, 1625,
        1520, 1455, 1450, 1480, 1445), nrow=5) );
K <- dim(Ydye)[1];
J <- dim(Ydye)[2];
```

Utilizing priors $V \sim IG(a_1, b_1)$, $V \sim IG(a_2, b_2)$, and $\mu \sim N(a_3, b_3)$, where $IG(\ , \ )$ corresponds to the inverse gamma distribution with pdf \$,x>0,\$ we note that the joint density of V, W, μ, θ_i, and $Y_{i,j}$ corresponds to:

$$
\begin{aligned}
f(V, W, \mu, \theta_1, ..., \theta_6, Y_{11}, ..., Y_{KJ_K}) = \\
C_2 \ e^{-b_1/V} \ V^{-a_1-1} \ e^{-b_2/W} \ W^{-a_2-1} \ e^{-(\mu-a_3)^2/2b_3} \ V^{-K/2} \ W^{-\frac{1}{2}\sum_{i=1}^{K} J_i} \times \\
\exp[-\sum_{i=1}^{K}(\theta_i - \mu)^2/2V] \times \exp[-\sum_{i=1}^{K}\sum_{i=1}^{J_i}(Y_{ij} - \theta_i)^2/2W]
\end{aligned}
$$

and thus

$$
\begin{aligned}
\pi(V, W, \mu, \theta_1, ..., \theta_6) = \frac{f(V, W, \mu, \theta_1, ..., \theta_6, Y_{11}, ..., Y_{KJ_K})}{f_Y(Y_{11}, ..., Y_{KJ_K})} \\
\propto f(V, W, \mu, \theta_1, ..., \theta_6, Y_{11}, ..., Y_{KJ_K}) \\
= C_3 \ e^{-b_1/V} \ V^{-a_1-1} \ e^{-b_2/W} \ W^{-a_2-1} \ e^{-(\mu-a_3)^2/2b_3} \ V^{-K/2} \ W^{-\frac{1}{2}\sum_{i=1}^{K} J_i} \times \\
\exp[-\sum_{i=1}^{K}(\theta_i - \mu)^2/2V] \times \exp[-\sum_{i=1}^{K}\sum_{i=1}^{J_i}(Y_{ij} - \theta_i)^2/2W]
\end{aligned}
$$

and, evaluating its logarithm:

$$
\begin{aligned}
\log[\pi(V, W, \mu, \theta_1, ..., \theta_6, Y_{11}, ..., Y_{KJ_K})] = \log(C_3) + (-b_1/V) - (a_1 + 1)\log(V) + \\
(-b_2/W) - (a_2 + 1)\log(W) - (\mu - a_3)^2/2b_3 + (-K/2)\log(V) \\
-\frac{1}{2}(\sum_{i=1}^{K} J_i)\,log(W) - (\sum_{i=1}^{K}(\theta_i - \mu)^2/2V) - (\sum_{i=1}^{K}\sum_{i=1}^{J_i}(Y_{ij} - \theta_i)^2/2W)
\end{aligned}
$$

I shall employ the above logarithm of the joint density in all of the below computations in order to avoid the potential overflow problems that arise when working with large quantities, in high dimensions. Working with this algorithm means that I must also modify the accept/reject step as follows:

$$
\begin{aligned}
log(A_i) > log(U_i) \implies X_i = Y_i \ (acceptance) \\
else \ X_i = X_{i-1}, \ for \ i \ \in \{1, ..., M\}
\end{aligned}
$$

$$
where \ \log(A_i) = log(\frac{\pi(Y_i)}{\pi(X_{i-1})}) = \log(g(Y_i)) - \log(g(X_{i-1}))
$$

Now, in order to carry out the Gibbs Sampler algorithm, here I include the conditional densities for V, W, μ, and θ_i:

1. Conditional of μ :

$$\mathcal{L}(\mu|V,W,\theta_1,...,\theta_6,Y_{11},...,Y_{KJ_K}) =$$

$$C_* \; e^{-(\mu-a_3)^2/2b_3} \; \exp[-\sum_{i=1}^{K}(\theta_i-\mu)^2/2V] =$$

$$C_{**} \; \exp[-\mu^2(\frac{1}{2b_3}+\frac{K}{2V})] \; \exp[\mu(\frac{a_3}{b_3}+\frac{\sum_{i=1}^{K}\theta_i}{V})]$$

Now, assuming $\mu \sim N(m,u)$, solve for m and u and obtain the conditional distribution:

$$\mu|V,W,\theta_1,...,\theta_6,Y_{11},...,Y_{KJ_K} \sim N(\frac{a_3V+b_3\sum_{i=1}^{K}\theta_i}{V+Kb_3} \; , \; \sqrt{\frac{Vb_3}{V+Kb_3}})$$

2. Conditional of V :

$$\mathcal{L}(V|W,\mu,\theta_1,...,\theta_6,Y_{11},...,Y_{KJ_K}) =$$

$$C_{**} \; e^{-b_1/V} \; V^{-a_1-1} \; V^{-K/2} \; \exp[-\frac{1}{2V}\sum_{i=1}^{K}(\theta_i-\mu)^2] =$$

$$C_{**} \; V^{-(\frac{K}{2}+a_1)-1} \; \exp[-\frac{1}{V}(b_1+\frac{1}{2}\sum_{i=1}^{K}(\theta_i-\mu)^2)]$$

and thus, by comparing the below form with the density of an inverse gamma, the conditional distribution of V is:

$$V|W,\mu,\theta_1,...,\theta_6,Y_{11},...,Y_{KJ_K} \sim IG(\frac{K}{2}+a_1, \; b_1+\frac{1}{2}\sum_{i=1}^{K}(\theta_i-\mu)^2)$$

3. Conditional of W :

$$\mathcal{L}(W|V,\mu,\theta_1,...,\theta_6,Y_{11},...,Y_{KJ_K}) =$$

$$C_* \; e^{-b_2/W} \; W^{-a_2-1} \; W^{-\frac{1}{2}\sum_{i=1}^{K}J_i} \times \exp[-\sum_{i=1}^{K}\sum_{i=1}^{J_i}(Y_{ij}-\theta_i)^2/2W] =$$

$$C_{**} \exp[-\frac{1}{W}(b_2+\frac{1}{2}\sum_{i=1}^{K}\sum_{j=1}^{J_i}(Y_{ij}-\theta_i)^2)] \; W^{-(a_2+\frac{1}{2}\sum_{i=1}^{K}J_i)-1}$$

and thus, by comparing the above to the pdf of the inverse gamma function, the conditional distribution of W is:

$$W|V,\mu,\theta_1,...,\theta_6,Y_{11},...,Y_{KJ_K} \sim IG(a_2+\frac{1}{2}\sum_{i=1}^{K}J_i \; , \; b_2+\frac{1}{2}\sum_{i=1}^{K}\sum_{j=1}^{J_i}(Y_{ij}-\theta_i)^2)$$

4. Conditional of θ_i :

To proceed to obtain the conditional distribution of θ_i first note how I break up the product of all the exponents containing a θ_i term in the joint density $\pi(V,W,\mu,\theta_1,...,\theta_6,Y_{11},...,Y_{KJ_K})$:

$$\exp[-\sum_{i=1}^{K}(\theta_i-\mu)^2/2V] \times \exp[-\sum_{i=1}^{K}\sum_{i=1}^{J_i}(Y_{ij}-\theta_i)^2/2W] =$$

$$\exp\{(\frac{-1}{2V})[(\sum_{m\neq i}^{K}\theta_m^2-2\theta_m\mu+\mu^2)+\theta_i^2-2\theta_i\mu+\mu^2]\}\times$$

$$\exp\{(\frac{-1}{2W})[\sum_{m\neq i}^{K}(\sum_{i=1}^{J_i}Y_{mj}^2-2Y_{mj}\theta_m)+J\sum_{m\neq i}^{K}\theta_m^2+\sum_{j=1}^{J}Y_{ij}^2-2\theta_i\sum_{j=1}^{J}Y_{ij}+J\theta_i^2]\}$$

Now the conditional distribution of θ_i :

$$\mathcal{L}(\theta_i|V,\mu,\tilde{\theta}_{(-i)},Y_{11},...,Y_{KJ_K}) =$$
$$C_* \ \exp[(\frac{-1}{2V})(\theta_i^2-2\theta_i\mu+\mu^2)] \ \exp[(\frac{-1}{2W})(\sum_{j=1}^{J}Y_{ij}^2-2\theta_i\sum_{j=1}^{J}Y_{ij}+J\theta_i^2)] =$$
$$C_{**} \ \exp[\frac{-1}{2V}(\theta_i^2-2\theta_i\mu)] \ \exp[\frac{-1}{2W}(-2\theta_i\sum_{j=1}^{J}Y_{ij}+J\theta_i^2)] =$$
$$C_{**} \ \exp[-\theta_i^2(\frac{1}{2V}+\frac{J}{2W})+\theta_i(\frac{\mu}{V}+\frac{\sum_{j=1}^{J}Y_{ij}}{W})]$$

and thus, assuming $\theta_i \sim N(m,u)$, solve for m and u and obtain the corresponding conditional distribution:

$$\theta_i|V,W,\mu,\theta_{(-i)},Y_{11},...,Y_{KJ_K} \sim N(\frac{W\mu+V\sum_{j=1}^{J}Y_{ij}}{W+JV} \ , \ \sqrt{\frac{VW}{W+JV}})$$

Below find the computations of the posterior mean of W/V for the two sets of prior values, according to the three algorithms requested (**a. Random-walk Metropolis**, **b. Metropolis-within-Gibbs**, and **c. Gibbs Sampler**). In my computations I work with logarithms in order to minimize the potential overflow problems that come with working in high dimensions and large quantities.

Set 1: "Reasonable" values

Consider prior values $a_i = 1500$ and $b_i = 1500^2$ for $i \in \{1,2,3\}$.

a. Random-walk Metropolis

Recall how the Random-walk Metropolis algorithm starts from an overdispersed starting distribution for the parameters of interest. In this case, and for all of the algorithms below, my starting distributions (priors) consist of:

$$V \sim IG(a_1,b_1)$$
$$W \sim IG(a_2,b_2)$$
$$\mu \sim N(a_3,\sqrt{b_3})$$
$$\theta_i \sim N(\mu,\sqrt{V}) \ for \ i \in \{1,...,6\}$$

Setting the stage for the three algorithms with "reasonable" parameters for their priors:

```r
logjointdist <- function(Ydat,V,W,mu,theta,a,b){
  if (V <= 0 | W <= 0){
    return(0)
  }else{
    stepp <- numeric(length(theta))
    for (i in 1:length(stepp)){
      stepp[i] <- sum((Ydat[i,]-theta[i])^2)
    }
    #K=6,J=5
    output <- (-b[1]/V) - (a[1]+1)*log(V) - (b[2]/W) - (a[2]+1)*log(W) -
      ((mu-a[3])^2)/(2*b[3]) - (K/2)*log(V) - (1/2)*(J*K)*log(W) -
      (sum((theta-mu)^2)/(2*V)) - (sum(stepp)/(2*W))
    return(output)
  }
}
```

```
h2 = function(V,W) { return( W/V ) }
```

and the Metropolis algorithm itself:

```
#Timing this algo
ptm <- proc.time()

set.seed(1234);
#a <- rep(1500,3); b <- rep(1500^2,3);
a=rep(1000,3);b=rep(1000^2,3);
M = 210000  # run length
B = 30000  # amount of burn-in

#Overdispersed starting distribution
#V,W,mu,theta
Xpre <- c(rinvgamma(1,shape=a[1],rate=b[1]), #V
       rinvgamma(1,shape=a[2],rate=b[2]), #W
       rnorm(1,a[3],sqrt(b[3]))) #mu
#Constructing vector of parameters X in two steps because t
#theta[i] ~ N(mu, V) and we're sampling for mu and V in Xpre
thetas <- rnorm(6,Xpre[3],sqrt(Xpre[1]));
X <- c(Xpre,thetas);

sigma = 13;  # proposal scaling
vlist = wlist = mulist = theta1list = theta2list = theta3list = numeric(M);
theta4list = theta5list = theta6list =  h2list = numeric(M);  # for keeping track of values
numaccept = 0;

#Y,V,W,mu,theta,a,b

for (i in 1:M) {
    #I can make proposals from same dist family or all normals with same scaling
    #However, proposing from inverse gamma isn't symmetric given how it is only
    #defined on the positive real line
    Y = X + sigma * rnorm(9);
    U = runif(1)  # for accept/reject
    #logjointdist <- function(Y,V,W,mu,theta,a,b){
    alpha = logjointdist(Ydye,Y[1],Y[2],Y[3],Y[4:9],a,b) - logjointdist(Ydye,X[1],X[2],X[3],X[4:9],a,b)  #
    if (log(U) < alpha) {
        X = Y  # accept proposal
      numaccept = numaccept + 1;
    }
    vlist[i] = X[1]; wlist[i] = X[2]; mulist[i] = X[3];
    theta1list[i] = X[4]; theta2list[i] = X[5]; theta3list[i] = X[6];
    theta4list[i] = X[7]; theta5list[i] = X[8]; theta6list[i] = X[9];
    h2list[i] = h2(X[1],X[2]);
}
timer3 <- proc.time() - ptm;
```

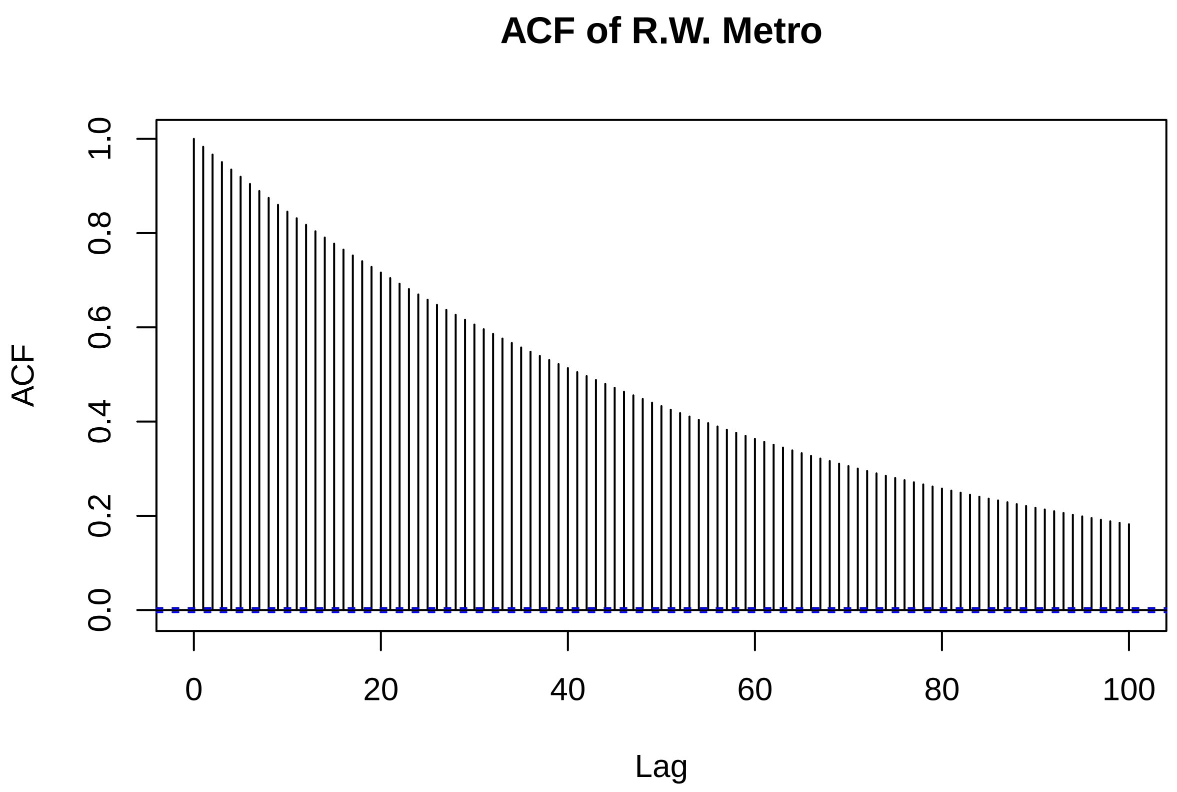
```
cat(M, "iterations of the Random-walk Metropolis Algorithm, with burn-in", B, "\n");
cat("Running time of this algo:","\n"); timer3;
cat("acceptance rate =", numaccept/M, "\n");
cat("Proposal scaling:",sigma,"\n");
u3 = mean(h2list[(B+1):M])
cat("mean of h is about", u3, "\n")

se3 =  sd(h2list[(B+1):M]) / sqrt(M-B)
```

```
cat("iid standard error would be about", se3, "\n")

varfact <- function(xxx) { 2 * sum(acf(xxx, plot=FALSE)$acf) - 1 }
thevarfact3 = varfact(h2list[(B+1):M])
se3 = se3 * sqrt( thevarfact3 )
cat("varfact = ", thevarfact3, "\n")
cat("true standard error is about", se3, "\n")
cat("approximate 95% confidence interval is (", u3 - 1.96 * se3, ",",
                                               u3 + 1.96 * se3, ")\n\n")
```

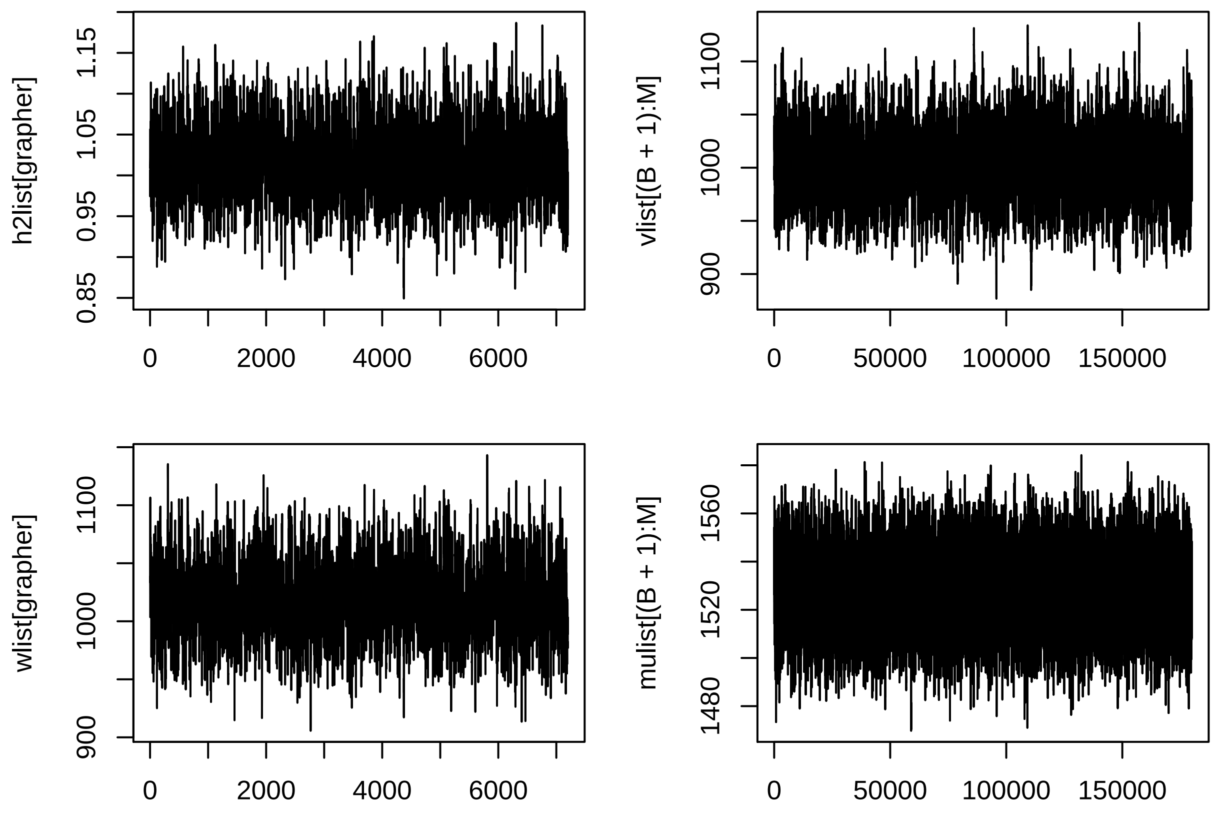
```
## 210000 iterations of the Random-walk Metropolis Algorithm, with burn-in 30000
## Running time of this algo:
##    user  system elapsed
##  15.700   0.132  16.144
## acceptance rate = 0.2147524
## Proposal scaling: 13
## mean of h is about 1.016553
## iid standard error would be about 0.0001077063
## varfact =  70.34368
## true standard error is about 0.0009033451
## approximate 95% confidence interval is ( 1.014783 , 1.018324 )
```

As in the previous set of graphs, the indeces only go up to 8000 because I am plotting the 25th points in the series B+1 to M in order to speed up the rendering of my PDF report.

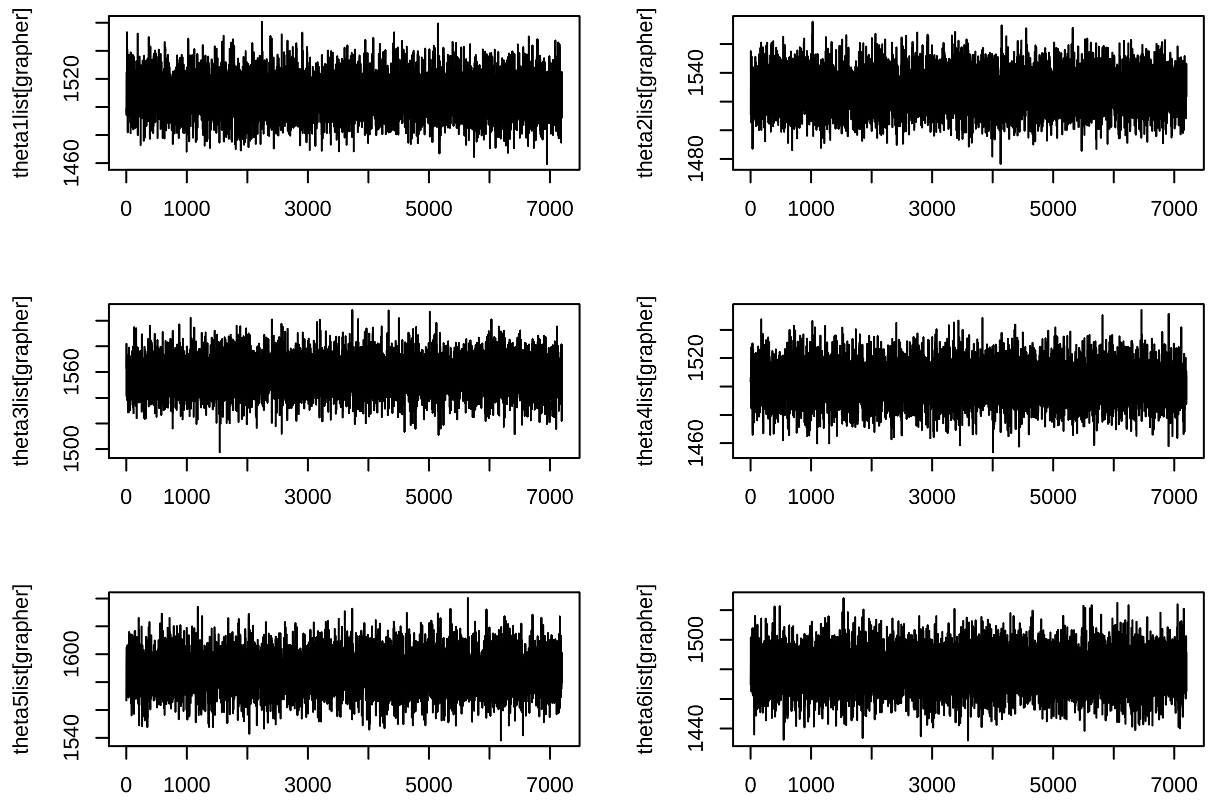
```
acf(h2list,lag.max=100,main="ACF of R.W. Metro");
```



```
par(mfrow=c(2,2), mai = c(.4, .7, .3, .2),mgp=c(3,1,0));
#Plotting only every 25th point between B+1 and M
grapher <- seq(from=B+1,M,by=25);
plot(h2list[grapher], type='l'); plot(vlist[(B+1):M], type='l');
plot(wlist[grapher], type='l'); plot(mulist[(B+1):M], type='l');
```



```r
par(mfrow=c(3,2), mai = c(.4, .6, .3, .2),mgp=c(3,1,0));
plot(theta1list[grapher], type='l'); plot(theta2list[grapher], type='l');
plot(theta3list[grapher], type='l'); plot(theta4list[grapher], type='l');
plot(theta5list[grapher], type='l'); plot(theta6list[grapher], type='l');
```



The above trace plots show good mixing throughout the algorithm for our estimates of W/V, as well as all parameters in question. Although the plots for x[3:9] (corresponding to values of μ and θ_i) exhibit very uneven moves at the very early runs (omitted values pre B+1), the swings are corrected "quickly." These wild swings may be attributed to the overdispersion of the starting distributions for these parameters.

b. Metropolis-within-Gibbs

Recall that the Metropolis-within-Gibbs works by formulating a proposal for each component at a time, leaving the rest as they are, and then applying the accept/reject step. This results in 9*M accept/reject steps, in contrast to the M steps from the regular Metropolis. Therefore, I expect the running time to increase substantially.

```r
set.seed(1234);
ptm <- proc.time()
#Metro-within-Gibbs Systematic Scan
#a <- rep(1500,3); b <- rep(1500^2,3);
M = 210000  # run length
B = 30000 # amount of burn-in

#Overdispersed starting distribution
#V,W,mu,theta
Xpre <- c(rinvgamma(1,shape=a[1],rate=b[1]),
       rinvgamma(1,shape=a[2],rate=b[2]),
       rnorm(1,a[3],sqrt(b[3])))
#Constructing X in two steps because theta[i] ~ N(mu, V) and
#we're sampling for mu and V in Xpre
#Sampling thetas
X <- c(Xpre,
       rnorm(6,Xpre[3],sqrt(Xpre[1])));

sigma = 13;  # proposal scaling
x1list = x2list = x3list = x4list = x5list = x6list = numeric(9*M);
x7list = x8list = x9list = h3list = numeric(9*M);  # for keeping track of values
numaccept = 0;

for (i in 1:M) {
  #Coordinates here are V,W,mu,theta[1:6]
  for (coord in 1:length(X)) {
    #I can make proposals from same dist family or all normals with same scaling
    #However, proposing from inverse gamma isn't symmetric given how it is only
    #defined on the positive real line
    Y = X
    Y[coord] = X[coord] + sigma * rnorm(1);
    U = runif(1)  # for accept/reject
    #Coordinate with index coord is shifted, everywhere else Y and X are identical.
    #Inheriting below from above for consistency.
    alpha = logjointdist(Ydye,Y[1],Y[2],Y[3],Y[4:9],a,b) - logjointdist(Ydye,X[1],X[2],X[3],X[4:9],a,b)  #
    if (log(U) < alpha) {
        X = Y  # accept proposal
      numaccept = numaccept + 1;
    }
    x1list[9*(i-1)+coord] = X[1]; x2list[9*(i-1)+coord] = X[2];
    x3list[9*(i-1)+coord] = X[3]; x4list[9*(i-1)+coord] = X[4];
    x5list[9*(i-1)+coord] = X[5]; x6list[9*(i-1)+coord] = X[6];
    x7list[9*(i-1)+coord] = X[7]; x8list[9*(i-1)+coord] = X[8];
    x9list[9*(i-1)+coord] = X[9];
    h3list[9*(i-1)+coord] = h2(X[1],X[2]);
  }
}
```

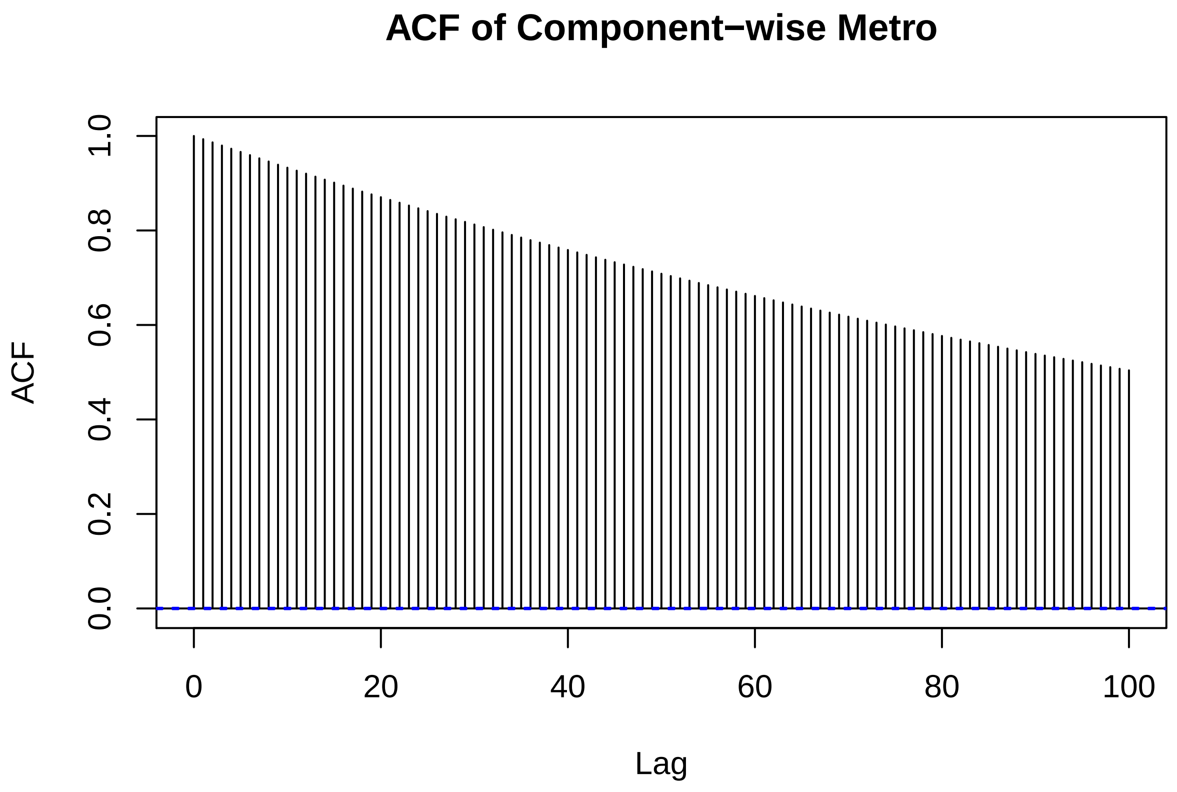
```r
timer4 <- proc.time() - ptm;
```

```r
cat(M, "iterations of the Variable-at-a-Time Metropolis Algorithm, with burn-in", B, "\n");
cat("Running time of this algo:\n"); timer4;
cat("Acceptance rate:", numaccept/(9*M), "\n");
cat("Proposal scaling:",sigma,"\n")
u4 = mean(h3list[(9*(B+1)):(9*M)])
cat("Mean of W/V is approximately", u4, "\n")

se4 =  sd(h3list[(9*(B+1)):(9*M)]) / sqrt(9*(M-B))
cat("iid standard error would be about", se4, "\n")

varfact <- function(xxx) { 2 * sum(acf(xxx, plot=FALSE)$acf) - 1 }
thevarfact4 = varfact(h3list[(9*(B+1)):(9*M)])

se4 = se4 * sqrt( thevarfact4 )
cat("varfact = ", thevarfact4, "\n")
cat("true standard error is about", se4, "\n")
cat("approximate 95% confidence interval is (", u4 - 1.96 * se4, ",",
                                                u4 + 1.96 * se4, ")\n\n")
```

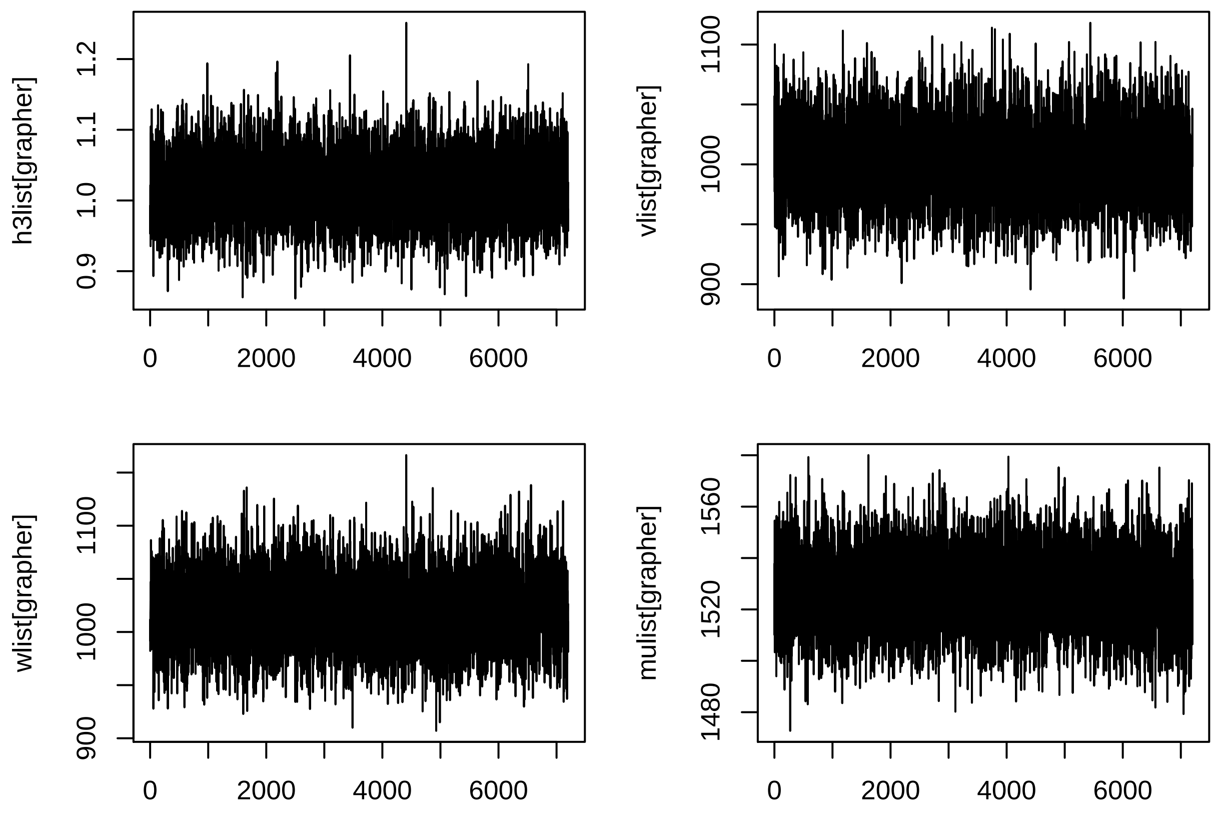
```
## 210000 iterations of the Variable-at-a-Time Metropolis Algorithm, with burn-in 30000
## Running time of this algo:
##    user  system elapsed
## 132.692   0.754 134.379
## Acceptance rate: 0.7411524
## Proposal scaling: 13
## Mean of W/V is approximately 1.017753
## iid standard error would be about 3.611122e-05
## varfact =  101.4721
## true standard error is about 0.0003637603
## approximate 95% confidence interval is ( 1.01704 , 1.018466 )
```

As in the previous set of graphs, the indeces only go up to 8000 because I am plotting the 25th points in the series B+1 to M in order to speed up the rendering of my PDF report.

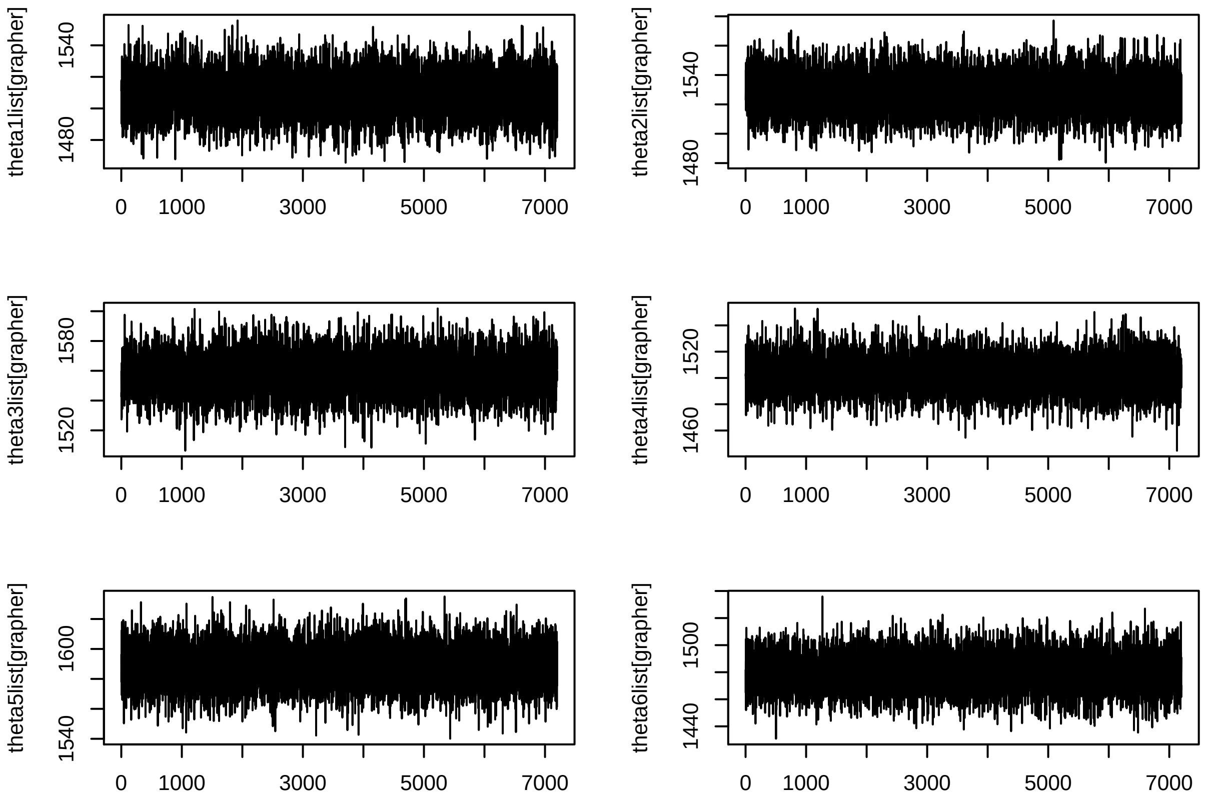
```r
acf(h3list,lag.max=100,main="ACF of Component-wise Metro");
```



```r
par(mfrow=c(2,2), mai = c(.4, .7, .3, .2),mgp=c(3,1,0));
#Plotting only every 25th point between 9*(B+1) and 9*M
grapher <- seq(from=9*B+1,to=9*M,by=9*25);

vlist <- x1list; wlist <- x2list; mulist <- x3list;
theta1list <- x4list; theta2list <- x5list;
theta3list <- x6list; theta4list <- x7list;
theta5list <- x8list; theta6list <- x9list;

plot(h3list[grapher], type='l'); plot(vlist[grapher], type='l');
plot(wlist[grapher], type='l'); plot(mulist[grapher], type='l');
```



```
# acf(hlist)

par(mfrow=c(3,2), mai = c(.4, .6, .3, .2),mgp=c(3,1,0));
plot(theta1list[grapher], type='l'); plot(theta2list[grapher], type='l');
plot(theta3list[grapher], type='l'); plot(theta4list[grapher], type='l');
plot(theta5list[grapher], type='l'); plot(theta6list[grapher], type='l');
```



```
# acf(hlist)
```

For this algorithm (executed with the same proposal scaling of 13 and seed 1234 as the previous Metropolis), the final estimate of W/V and the mixing of the parameters are very similar. However, the varfact component is larger while the iid standard error is smaller, which lead to a smaller true standard error. Another set of differences lies in the acceptance rate and running-time. The acceptance rate and running time are higher for this algorithm (rate approx 78%, running-time) than for the Metropolis (rate approx. 30%, running-time).

An interesting line of inquiry would be to vary the proposal scaling of the Variable-at-a-time algorithm to see how the diagnostic metrics would change.

c. Gibbs Sampler

Recall that the Gibbs Sampler requires that the coordinates move by sampling from their respective conditional distributions, conditional on the current values of every other coordinate. Therefore, the accept/reject step has no place here, as every proposal is "optimal".

With two ways of looping through the coordinates, I first choose systematic scan due to the fact that all coordinates will be "moved" in every round of the M iterations. As pointed out in the lecture, the alternative random scan…

Below I construct the functions to sample V, W, μ, and θ_i from their respective conditional distributions:

```
#Sampling from conditional distribution of mu
rcondmu <- function(K,a,b,V,theta){
  out <- rnorm(1,((a[3]*V+b[3]*sum(theta))/(V+K*b[3])),sqrt((b[3]*V)/(V+K*b[3])))
  return(out)
}

sum_data <- function(Ydat,theta){
  cum <- 0;
  for(i in 1:6){
    for(j in 1:5){
      cum + cum + ((Ydat[i,j]-theta[i])^2)
```

```r
    }
  }
  return(cum)
}

sum_theta <- function(theta,mu){
  cum <- 0
  for (i in 1:6){
    cum = cum + (theta[i]-mu)^2
  }
  return(cum)
}

#Sampling from conditional distribution of W
rcondw <- function(Ydat,K,J,a,b,mu,theta){
  out <- rinvgamma(1, shape=(a[2]+((K*J)/2)), rate=(b[2] + (1/2)*sum_data(Ydat,theta)))
  return(out)
}


#Sampling from conditional distribution of V

rcondv <- function(K,a,b,mu,theta){
  out <- rinvgamma(1, shape=((K/2)+a[1]), rate=(b[1] + (1/2)*sum_theta(theta,mu)))
  return(out)
}

#Sampling from conditional distribution of theta[i]
rcondtheta <- function(Ydat,index,J,V,W,mu){
  out <- rnorm(1,((mu*W+V*sum(Ydat[index,]))/(W+(J*V))),sqrt((V*W)/(W+(J*V))))
  return(out)
}
```

Here is the algorithm:

```r
#Gibbs Sampler (Systematic Scan)
a <- rep(1500,3); b <- rep(1500^2,3);
K=6;J=5;
M = 210000  # run length
B = 25000  # amount of burn-in

set.seed(1234);
ptm <- proc.time()

#Starting values
#V,W,mu,theta

V <- rinvgamma(1,shape=a[1],rate=b[1]);
W <- rinvgamma(1,shape=a[2],rate=b[2]);
mu <- rnorm(1,a[3],sqrt(b[3]));
theta <- rnorm(6,mu,sqrt(V));

#To keep track of values
vlist = wlist = mulist = numeric(M);
theta1list = theta2list = theta3list = theta4list = numeric(M)
theta5list = theta6list = h4list = numeric(M);
```

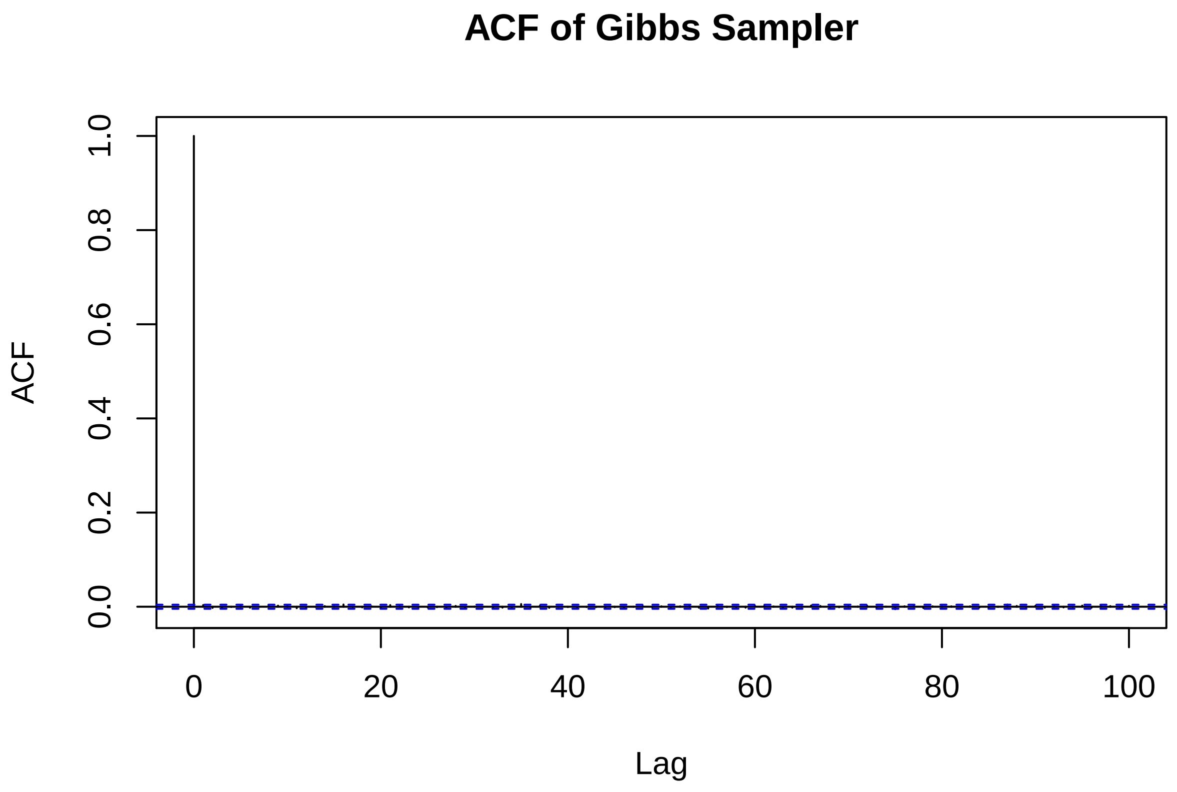
```r
for (i in 1:M) {
  #Parameters here are V,W,mu,theta[1:6]
  #Systematic scan
  V <- rcondv(K,a,b,mu,theta);
  W <- rcondw(Ydye,K,J,a,b,mu,theta)
  mu <- rcondmu(K,a,b,V,theta)
  for (j in 1:K){
    theta[j] <- rcondtheta(Ydye,j,J,V,W,mu)
  }
    #Storing values
  vlist[i] <- V; wlist[i] <- W; mulist[i] <- mu;
  theta1list[i] = theta[1]; theta2list[i] = theta[2];
  theta3list[i] = theta[3]; theta4list[i] = theta[4];
  theta5list[i] = theta[5]; theta6list[i] = theta[6];
  h4list[i] = W/V;
}

timer5 <- proc.time() - ptm;
```

```r
cat(M, "iterations of the Gibbs Sampler Algorithm, with burn-in", B, "\n");
cat("Run-time of this algo is:","\n"); timer5;
#u = mean(h4list[(9*(B+1)):(9*M)])
u5 = mean(h4list[(B+1):M])
cat("Mean of W/V is approximately", u5, "\n")

se5 =  sd(h4list[(B+1):M]) / sqrt(M-B)
cat("iid standard error would be about", se5, "\n")

varfact <- function(xxx) { 2 * sum(acf(xxx, plot=FALSE)$acf) - 1 }
thevarfact5 = varfact(h4list[(B+1):M])

s5 = se5 * sqrt( thevarfact5 )
cat("varfact = ", thevarfact5, "\n")
cat("True standard error is about", s5, "\n")
cat("Approximate 95% confidence interval is (", u5 - 1.96 * se5, ",",
                                               u5 + 1.96 * se5, ")\n\n")
```

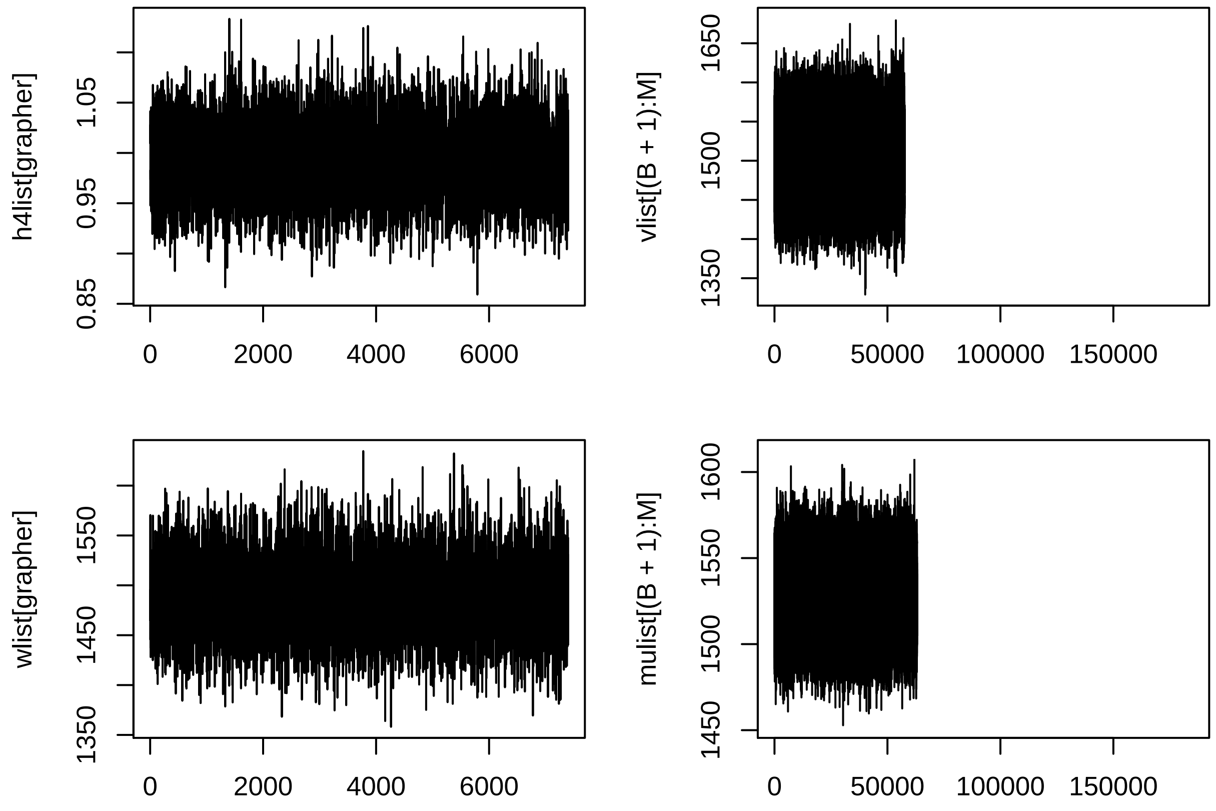
```
## 210000 iterations of the Gibbs Sampler Algorithm, with burn-in 25000
## Run-time of this algo is:
##    user  system elapsed
##  25.230   0.241  25.753
## Mean of W/V is approximately 0.9904677
## iid standard error would be about 8.382705e-05
## varfact =  0.9842602
## True standard error is about 8.316472e-05
## Approximate 95% confidence interval is ( 0.9903034 , 0.990632 )
```

As in the previous set of graphs, the indeces only go up to 8000 because I am plotting the 25th points in the series B+1 to M in order to speed up the rendering of my PDF report.

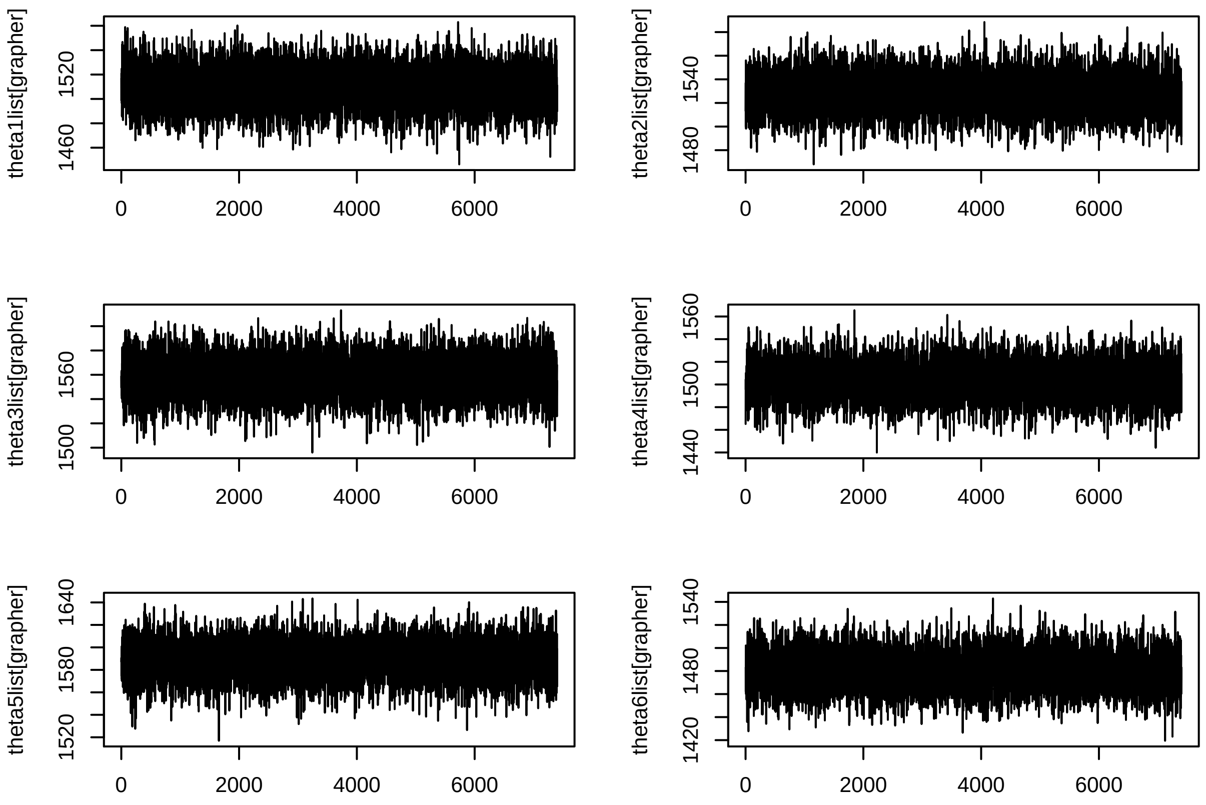
```r
acf(h4list,lag.max=100,main="ACF of Gibbs Sampler");
```



```
par(mfrow=c(2,2), mai = c(.4, .7, .3, .2),mgp=c(3,1,0));
#Plotting only every 25th point between B+1 and M
grapher <- seq(from=B+1,M,by=25);
plot(h4list[grapher], type='l'); plot(vlist[(B+1):M], type='l');
plot(wlist[grapher], type='l'); plot(mulist[(B+1):M], type='l');
```



```r
par(mfrow=c(3,2), mai = c(.4, .6, .3, .2),mgp=c(3,1,0));
plot(theta1list[grapher], type='l'); plot(theta2list[grapher], type='l');
plot(theta3list[grapher], type='l'); plot(theta4list[grapher], type='l');
plot(theta5list[grapher], type='l'); plot(theta6list[grapher], type='l');
```



```r
# acf(hlist)
```

Discussion

The below chart provides a succint comparison of the above three methods for the reasonable values $a_i = 1500$ and $b_i = 1500^2$ for $i \in \{1, 2, 3\}$.

Right away we can observe the larger running time of the Variable-at-a-Time Monte Carlo algorithm, albeit yielding smaller "true" standard error than the regular Monte Carlo.

row.names....names2	meanh	varfact	truese	runtime
Metropolis	1.0165533	70.3436791	0.0009033	17.306
Metro-within-Gibbs	1.0177533	101.4720520	0.0003638	154.653
Gibbs Sampler	0.9904677	0.9842602	0.0000838	26.892

Another interesting metric here is the acceptance rate of both Metropolis (21%) and Variable-at-a-Time Metropolis (74%). Such a large rate for the Metro-within-Gibbs suggests that the scaling of the proposal could be adjusted further to achieve greater precision. However, as mentioned before, it is also important to compare these methods for shared parameters and conditions.

The Gibbs Sampler needs not be subjected to a critique on its acceptance rate given how every proposal made is accepted. What we can observe accross the board is that the estimates from all three algorithms are very close to the value 1. The Gibbs sampler yields the narrowest confidence intervals, suggesting greater precision; however, I was unable to compute the expected value of W/V numerically and thus cannot affirm one answer to be closer to the truth than the rest. It is comforting to see that all three methods approached the same value.

Set 2: “Unreasonable” values

Consider prior values $a_i = b_i = 100$ for $i \in \{1, 2, 3\}$.

a. Random-walk Metropolis

```r
set.seed(1234);

a=b=rep(100,3);
M = 210000   # run length
B = 30000  # amount of burn-in

#Overdispersed starting distribution
#V,W,mu,theta
Xpre <- c(rinvgamma(1,a[1],b[1]), #V
       rinvgamma(1,a[2],b[2]), #W
       rnorm(1,a[3],sqrt(b[3]))) #mu
#Constructing vector of parameters X in two steps because theta[i] ~ N(mu, V)
#and we're sampling for mu and V in Xpre
thetas <- rnorm(6,Xpre[3],sqrt(Xpre[1]));
X <- c(Xpre,thetas);

sigma = 13;  # proposal scaling
x1list = x2list = x3list = x4list = x5list = x6list = numeric(M);
x7list = x8list = x9list = h5list = numeric(M);  # for keeping track of values
numaccept = 0;

#Y,V,W,mu,theta,a,b

for (i in 1:M) {
    #Make proposals from normals with same scaling
    Y = X + sigma * rnorm(9);
    U = runif(1)  # for accept/reject
    #logjointdist <- function(Y,V,W,mu,theta,a,b){
    alpha = logjointdist(Ydye,Y[1],Y[2],Y[3],Y[4:9],a,b) - logjointdist(Ydye,X[1],X[2],X[3],X[4:9],a,b)
    #accept/reject
    if (log(U) < alpha) {
        X = Y  # accept proposal
      numaccept = numaccept + 1;
    }
    x1list[i] = X[1]; x2list[i] = X[2]; x3list[i] = X[3];
    x4list[i] = X[4]; x5list[i] = X[5]; x6list[i] = X[6];
    x7list[i] = X[7]; x8list[i] = X[8]; x9list[i] = X[9];
    h5list[i] = h2(X[1],X[2]);
}
timer6 <- proc.time() - ptm;
```

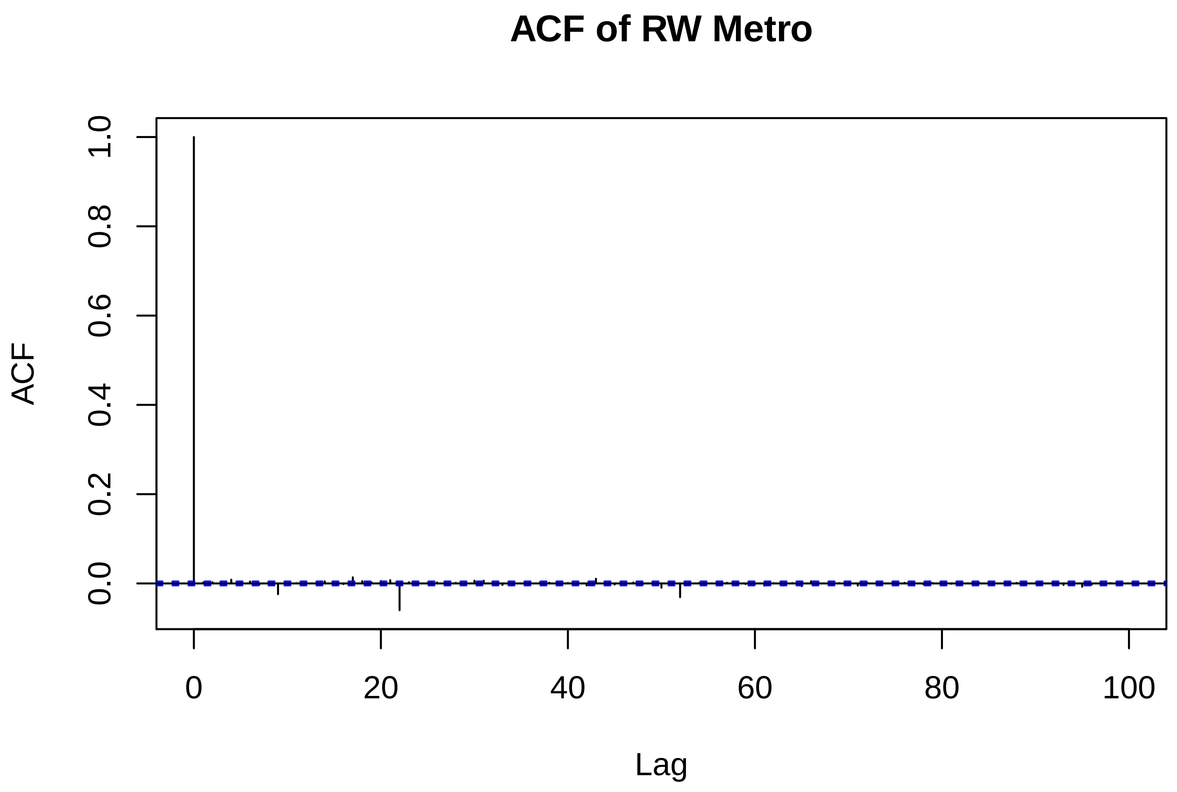
```r
cat(M, "iterations of the Random-walk Metropolis Algorithm, with burn-in", B, "\n");
cat("Running time of this algo:","\n"); timer6;
cat("acceptance rate =", numaccept/M, "\n");
cat("Proposal scaling:",sigma,"\n");
u6 = mean(h5list[(B+1):M])
cat("mean of h is about", u6, "\n")

se6 =  sd(h5list[(B+1):M]) / sqrt(M-B)
cat("iid standard error would be about", se6, "\n")
```

```
varfact <- function(xxx) { 2 * sum(acf(xxx, plot=FALSE)$acf) - 1 }
thevarfact6 = varfact(h5list[(B+1):M])
se6 = se6 * sqrt( thevarfact6 )
cat("varfact = ", thevarfact6, "\n")
cat("true standard error is about", se6, "\n")
cat("approximate 95% confidence interval is (", u6 - 1.96 * se6, ",",
                                                u6 + 1.96 * se6, ")\n\n")
```

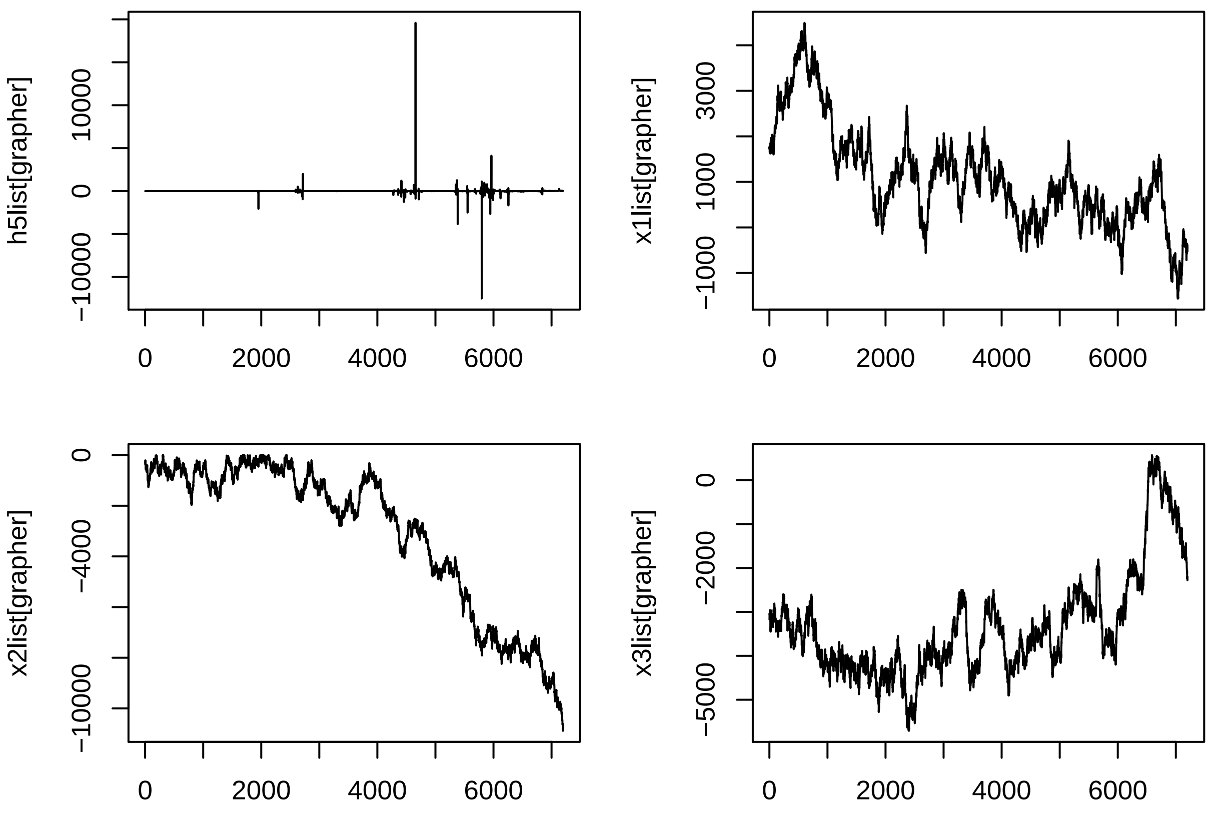
```
## 210000 iterations of the Random-walk Metropolis Algorithm, with burn-in 30000
## Running time of this algo:
##    user  system elapsed
##  33.695   0.547  34.668
## acceptance rate = 0.9983381
## Proposal scaling: 13
## mean of h is about -2.860983
## iid standard error would be about 2.5829
## varfact =  0.8863386
## true standard error is about 2.431686
## approximate 95% confidence interval is ( -7.627087 , 1.905121 )
```

As in the previous set of graphs, the indeces only go up to 8000 because I am plotting the 25th points in the series B+1 to M in order to speed up the rendering of my PDF report.

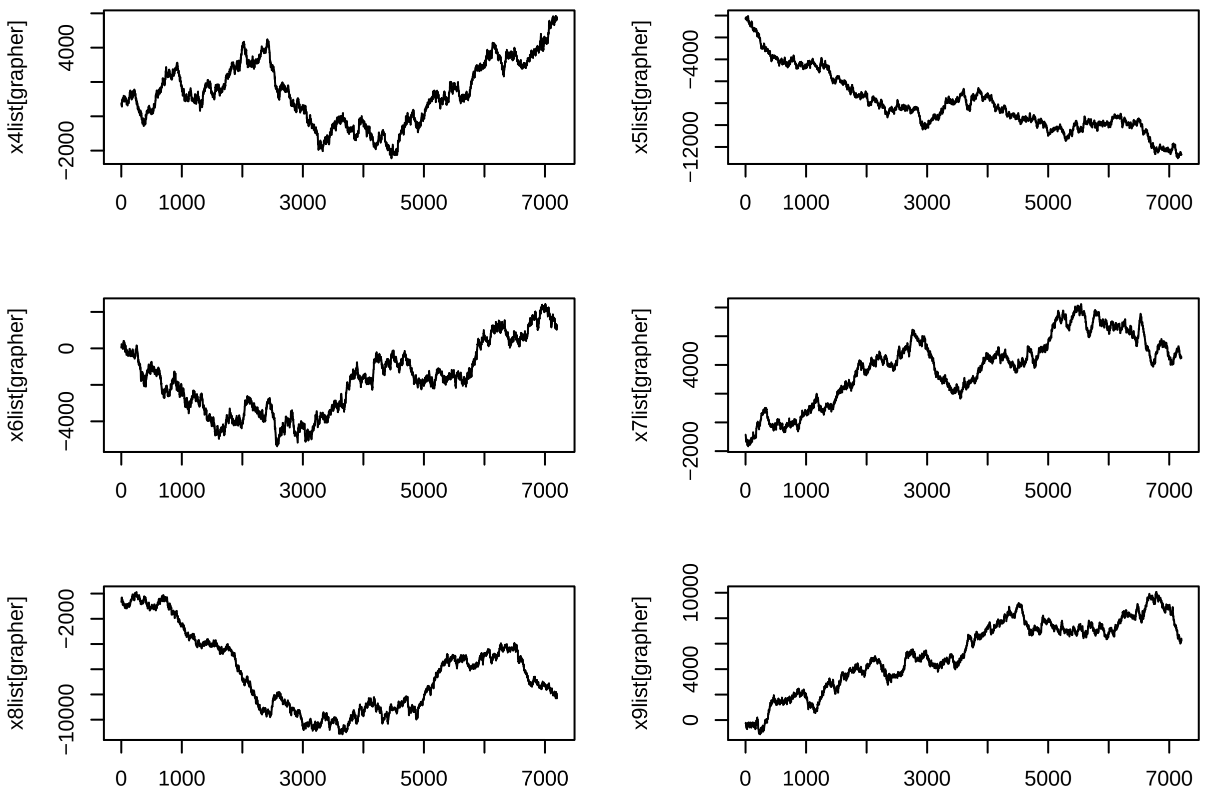
```
acf(h5list,lag.max=100,main="ACF of RW Metro");
```



```
par(mfrow=c(2,2), mai = c(.4, .7, .3, .2),mgp=c(3,1,0));
#Plotting only every 25th point between B+1 and M
grapher <- seq(from=B+1,M,by=25);
plot(h5list[grapher], type='l'); plot(x1list[grapher], type='l');
plot(x2list[grapher], type='l'); plot(x3list[grapher], type='l');
```



```r
# acf(hlist)

par(mfrow=c(3,2), mai = c(.4, .6, .3, .2),mgp=c(3,1,0));
plot(x4list[grapher], type='l'); plot(x5list[grapher], type='l');
plot(x6list[grapher], type='l'); plot(x7list[grapher], type='l');
plot(x8list[grapher], type='l'); plot(x9list[grapher], type='l');
```



```r
# acf(hlist)
```

The inclusion of these unreasonable values to the R.W. Metropolis algorithm, everything else remaining the same from the run above with reasonable values, yields very unstable estimates, as well as a significantly higher acceptance rate. Apples to apples, the values given to distributions of priors make a significant difference in the algorithm's accuracy.

The plots above show upwards and downwards trends in the estimations of the different parameters. Compare these plots to the ones from the previous run with reasonable parameters and recall how those display good mixing. The ACF plot here appears highly abnormal, because the proposals made and either accepted or rejected in this algorithm have a very well defined relationship with their previous values.

One way to ameliorate the accuracy of this algorithm is to modify the scale of the proposals, but inclusion of several extreme values did not improve the estimation. Moreover, different seed values yield wildly different results. This suggests great sensitivity of this algorithm to sensible prior in this scenario.

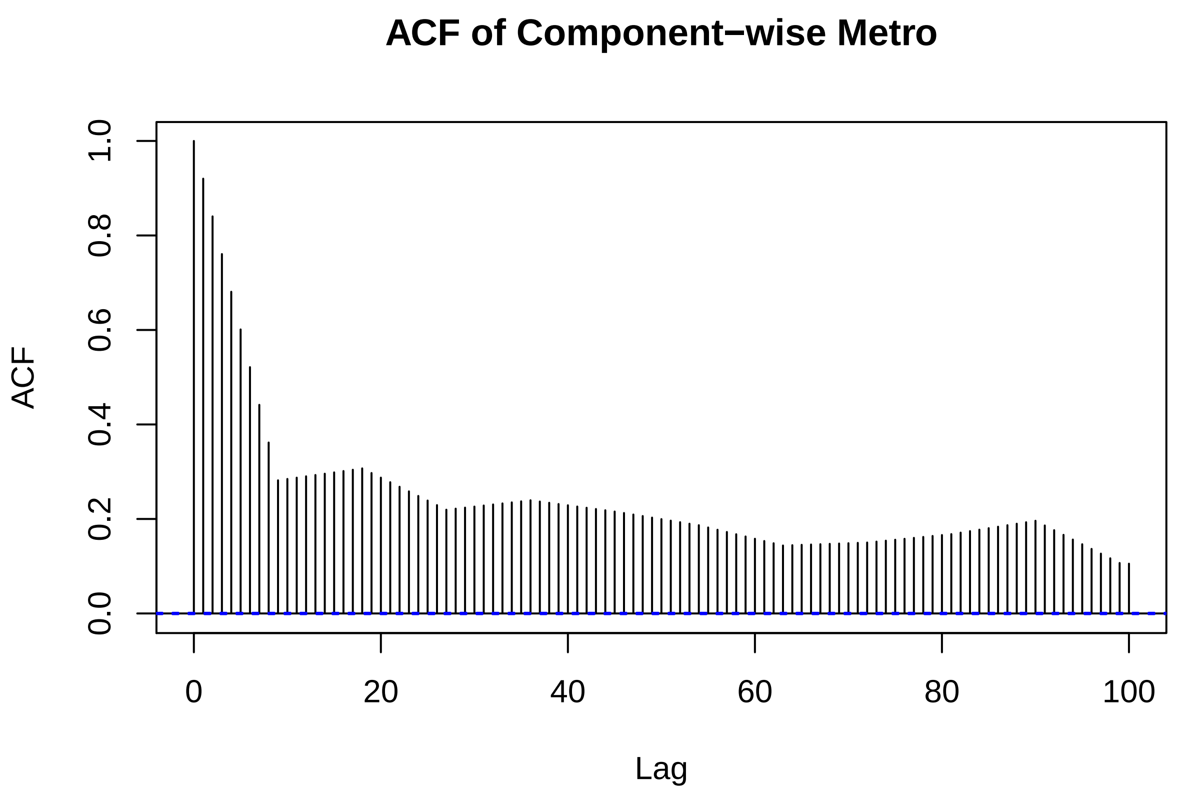
b. Metropolis-within-Gibbs

```r
a=b=rep(100,3);
set.seed(1234);
ptm <- proc.time()
#Metro-within-Gibbs Systematic Scan

M = 210000  # run length
B = 30000 # amount of burn-in

#Overdispersed starting distribution
#V,W,mu,theta
Xpre <- c(rinvgamma(1,a[1],b[1]),
       rinvgamma(1,a[2],b[2]),
       rnorm(1,a[3],sqrt(b[3])))


#Constructing X in two steps because theta[i] ~ N(mu, V) and we're sampling
#for mu and V in Xpre
#Sampling thetas
X <- c(Xpre,
       rnorm(6,Xpre[3],sqrt(Xpre[1])));




sigma = 13;  # proposal scaling
x1list = x2list = x3list = x4list = x5list = x6list = x7list = numeric(9*M);
x8list = x9list = h6list = numeric(9*M);  # for keeping track of values
numaccept = 0;

for (i in 1:M) {
  #Coordinates here are V,W,mu,theta[1:6]
  for (coord in 1:length(X)) {
    #I can make proposals from same dist family or all normals with same scaling
    #However, proposing from inverse gamma isn't symmetric given how it is only
    #defined on the positive real line
    Y = X
    Y[coord] = X[coord] + sigma * rnorm(1);
    U = runif(1)  # for accept/reject
    #Coordinate with index coord is shifted, everywhere else Y and X are identical.
    #Inheriting below from above for consistency.
    alpha = logjointdist(Ydye,Y[1],Y[2],Y[3],Y[4:9],a,b) - logjointdist(Ydye,X[1],X[2],X[3],X[4:9],a,b)
    #accept/reject
    if (log(U) < alpha) {
        X = Y  # accept proposal
      numaccept = numaccept + 1;
    }
    x1list[9*(i-1)+coord] = X[1]; x2list[9*(i-1)+coord] = X[2];
    x3list[9*(i-1)+coord] = X[3]; x4list[9*(i-1)+coord] = X[4];
    x5list[9*(i-1)+coord] = X[5]; x6list[9*(i-1)+coord] = X[6];
    x7list[9*(i-1)+coord] = X[7]; x8list[9*(i-1)+coord] = X[8];
    x9list[9*(i-1)+coord] = X[9];
    h6list[9*(i-1)+coord] = h2(X[1],X[2]);
  }
}
timer7 <- proc.time() - ptm;
```

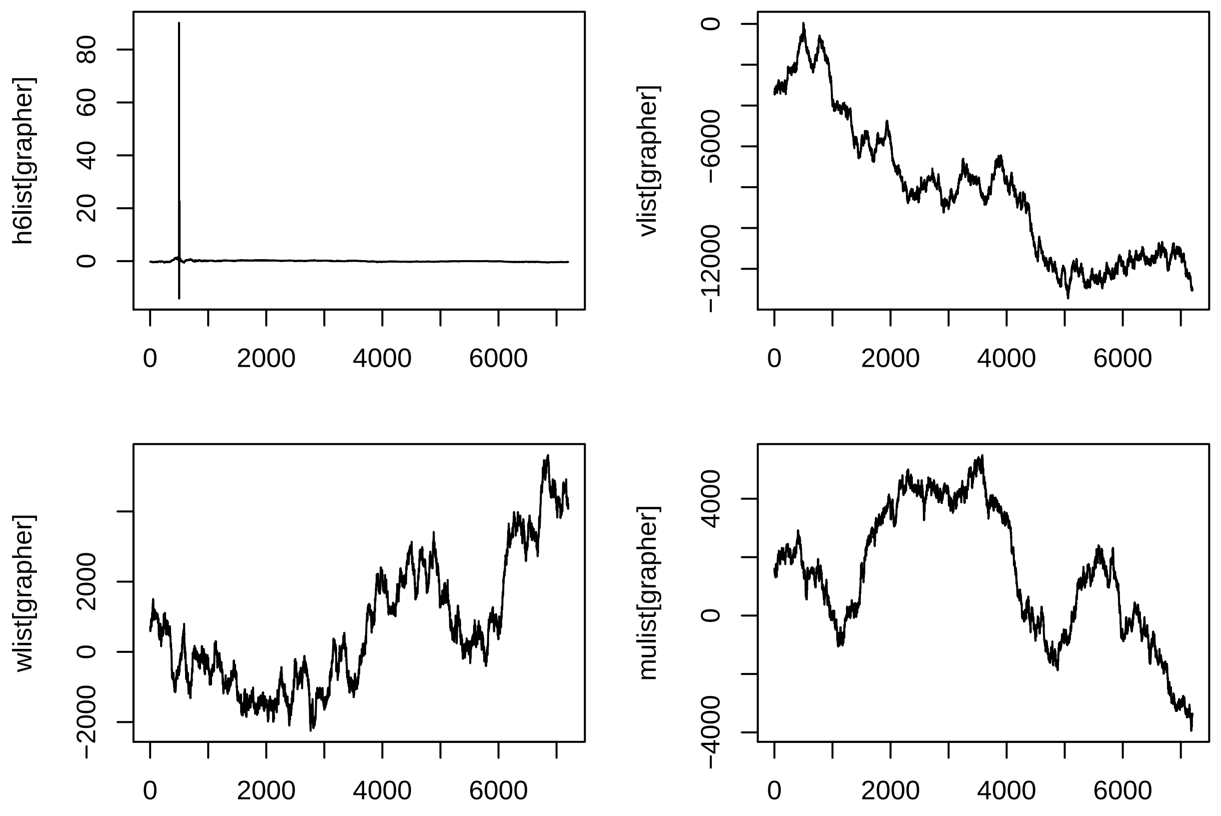
```r
cat(M, "iterations of the Random-walk Metropolis Algorithm, with burn-in", B, "\n");
cat("Running time of this algo:\n"); timer7;
cat("Acceptance rate =", numaccept/(9*M), "\n");
u7 = mean(h6list[(9*(B+1)):(9*M)])
cat("Mean of W/V is approximately", u7, "\n")

se7 =  sd(h6list[(9*(B+1)):(9*M)]) / sqrt(9*(M-B))
cat("iid standard error would be about", se7, "\n")

varfact <- function(xxx) { 2 * sum(acf(xxx, plot=FALSE)$acf) - 1 }
thevarfact7 = varfact(h6list[(9*(B+1)):(9*M)])

se7 = se7 * sqrt( thevarfact )
cat("varfact = ", thevarfact7, "\n")
cat("True standard error is about", se7, "\n")
cat("Approximate 95% confidence interval is (", u7 - 1.96 * se7, ",",
                                                u7 + 1.96 * se7, ")\n\n")
```

```
## 210000 iterations of the Random-walk Metropolis Algorithm, with burn-in 30000
## Running time of this algo:
##    user  system elapsed
##  48.487   0.479  49.539
## Acceptance rate = 0.9999931
## Mean of W/V is approximately -0.025439
## iid standard error would be about 0.004725436
## varfact =  9.431131
## True standard error is about 0.02185378
## Approximate 95% confidence interval is ( -0.06827241 , 0.01739441 )
```

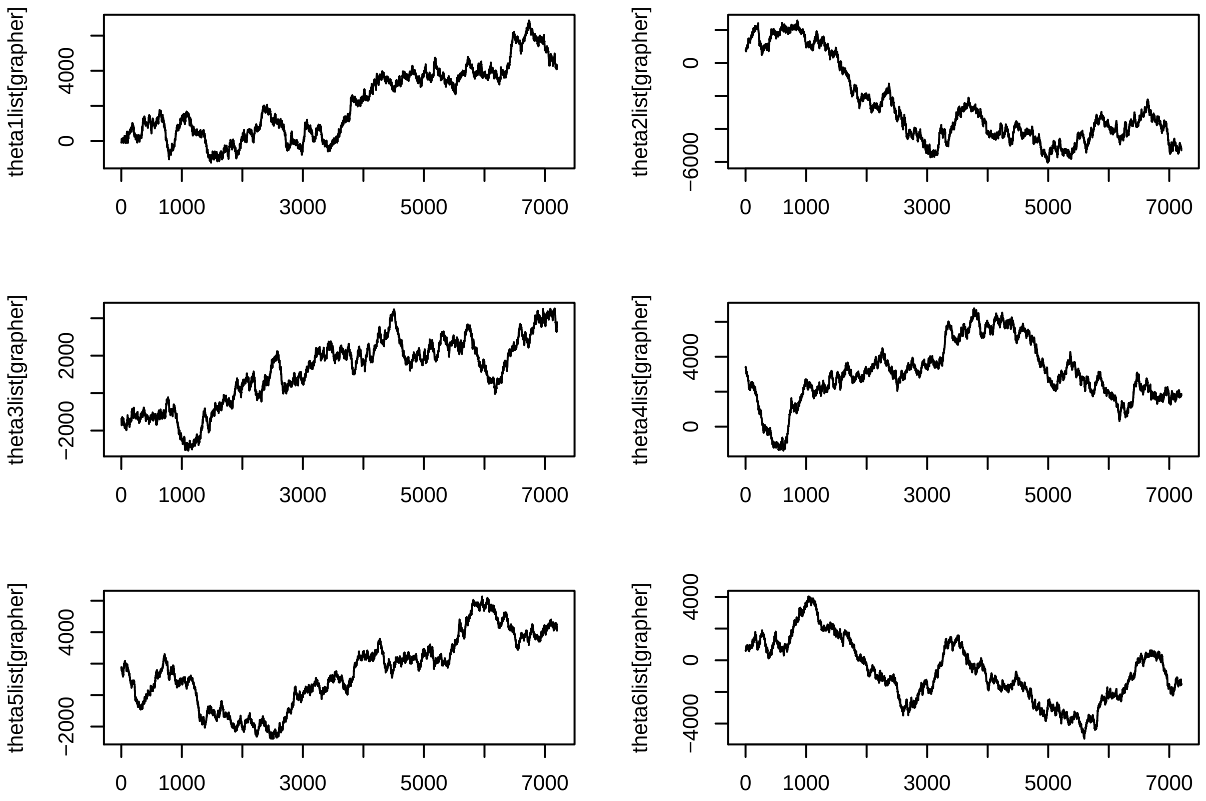
```r
acf(h6list,lag.max=100,main="ACF of Component-wise Metro");
```



```
par(mfrow=c(2,2), mai = c(.4, .7, .3, .2),mgp=c(3,1,0));
#Plotting only every 25th point between 9*(B+1) and 9*M
grapher <- seq(from=9*B+1,to=9*M,by=9*25);

vlist <- x1list; wlist <- x2list; mulist <- x3list;
theta1list <- x4list; theta2list <- x5list;
theta3list <- x6list; theta4list <- x7list;
theta5list <- x8list; theta6list <- x9list;

plot(h6list[grapher], type='l'); plot(vlist[grapher], type='l');
plot(wlist[grapher], type='l'); plot(mulist[grapher], type='l');
```



```r
# acf(hlist)

par(mfrow=c(3,2), mai = c(.4, .6, .3, .2),mgp=c(3,1,0));
plot(theta1list[grapher], type='l'); plot(theta2list[grapher], type='l');
plot(theta3list[grapher], type='l'); plot(theta4list[grapher], type='l');
plot(theta5list[grapher], type='l'); plot(theta6list[grapher], type='l');
```



c. Gibbs Sampler

Below is my algorithm for the Gibbs Sampler with unreasonable starting values:

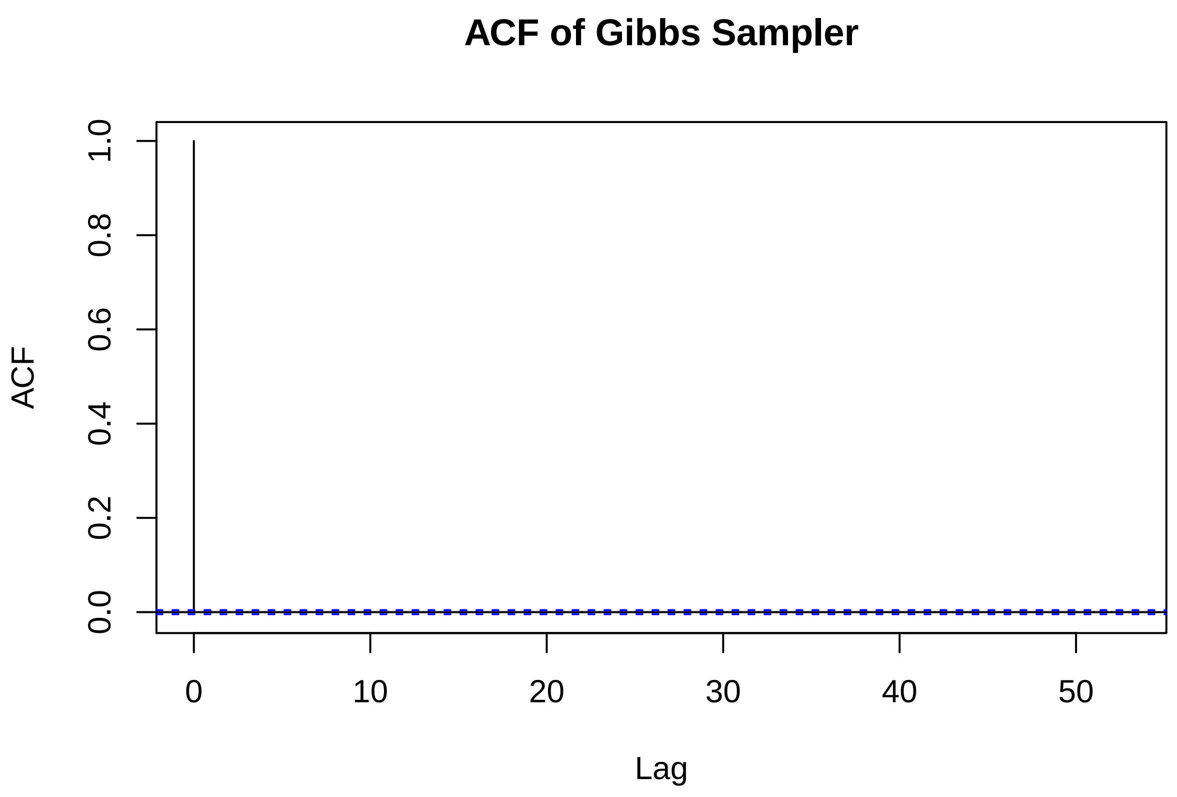
```r
#Gibbs Sampler - Systematic Scan
a=b=rep(100,3);
K=6;J=5;
M = 210000  # run length
B = 25000  # amount of burn-in

set.seed(1234);

ptm <- proc.time()

#Starting values
#V,W,mu,theta
V <- rinvgamma(1,shape=a[1],rate=b[1]);
W <- rinvgamma(1,shape=a[2],rate=b[2]);
mu <- rnorm(1,a[3],sqrt(b[3]));
theta <- rnorm(6,mu,sqrt(V));

#To keep track of values
vlist = wlist = mulist = h7list = numeric(M);
theta1list = theta2list = theta3list = theta4list = numeric(M);
theta5list = theta6list= numeric(M);


for (i in 1:M) {
  #Parameters here are V,W,mu,theta[1:6]
  #Systematic scan
  V <- rcondv(K,a,b,mu,theta);
  W <- rcondw(Ydye,K,J,a,b,mu,theta)
  mu <- rcondmu(K,a,b,V,theta)

  for (j in 1:K){
    theta[j] <- rcondtheta(Ydye,j,J,V,W,mu)

  }
  #Storing values
  vlist[i] <- V; wlist[i] <- W; mulist[i] <- mu;
  theta1list[i] = theta[1]; theta2list[i] = theta[2];
  theta3list[i] = theta[3]; theta4list[i] = theta[4];
  theta5list[i] = theta[5]; theta6list[i] = theta[6];
  h7list[i] = W/V;
}

timer8 <- proc.time() - ptm;
```

```
cat(M, "iterations of the Gibbs Sampler Algorithm, with burn-in", B, "\n");
cat("Run-time of this algo is:","\n"); timer8;
u8 = mean(h7list[(B+1):M])
cat("Mean of W/V is approximately", u8, "\n")

se8 =  sd(h7list[(B+1):M]) / sqrt(M-B)
cat("iid standard error would be about", se8, "\n")

varfact <- function(xxx) { 2 * sum(acf(xxx, plot=FALSE)$acf) - 1 }
thevarfact8 = varfact(h7list[(B+1):M])

se8 = se8 * sqrt( thevarfact8 )
cat("varfact = ", thevarfact8, "\n")
cat("True standard error is about", se8, "\n")
cat("Approximate 95% confidence interval is (", u8 - 1.96 * se8, ",",
                                                u8 + 1.96 * se8, ")\n\n")
```

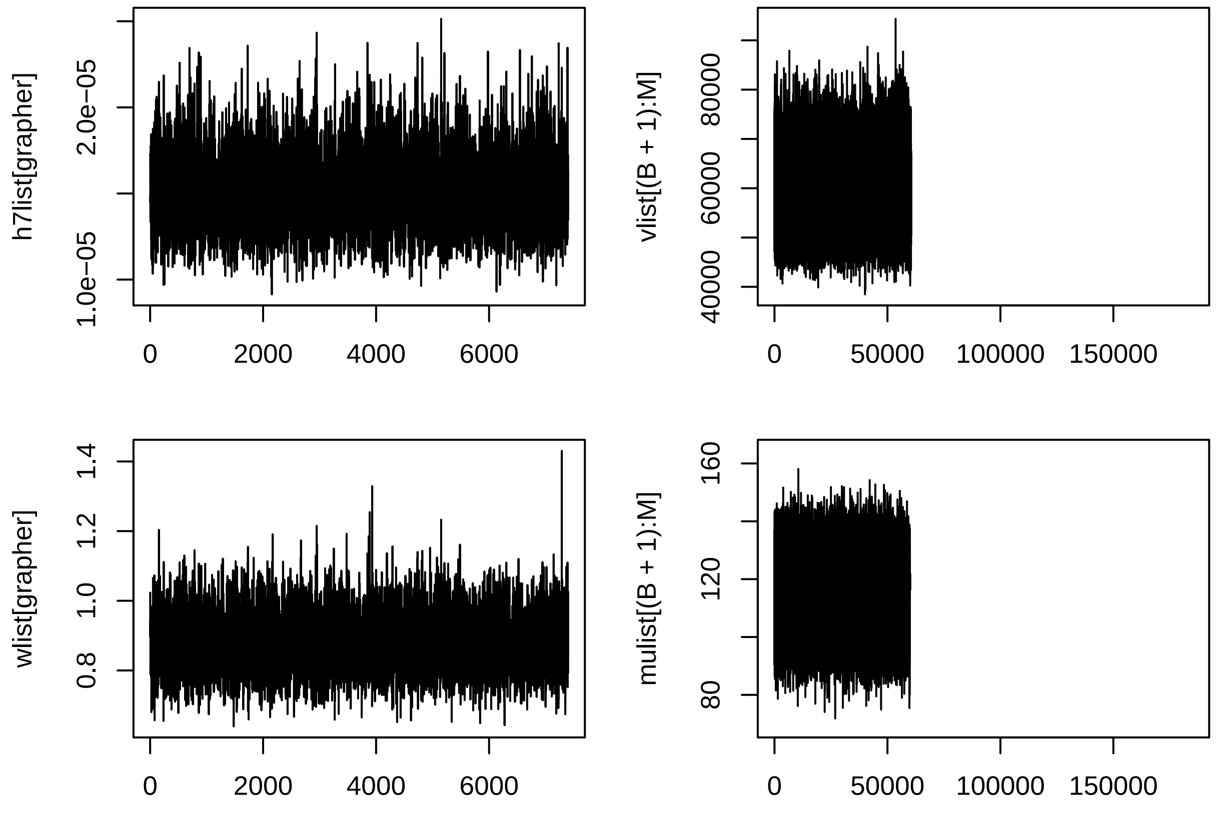
```
## 210000 iterations of the Gibbs Sampler Algorithm, with burn-in 25000
## Run-time of this algo is:
##    user  system elapsed
##  24.471   0.150  24.709
## Mean of W/V is approximately 1.507929e-05
## iid standard error would be about 4.818088e-09
## varfact =  1.056478
## True standard error is about 4.952278e-09
## Approximate 95% confidence interval is ( 1.506959e-05 , 1.5089e-05 )
```

As in the previous set of graphs, the indeces only go up to 8000 because I am plotting the 25th points in the series B+1 to M in order to speed up the rendering of my PDF report.

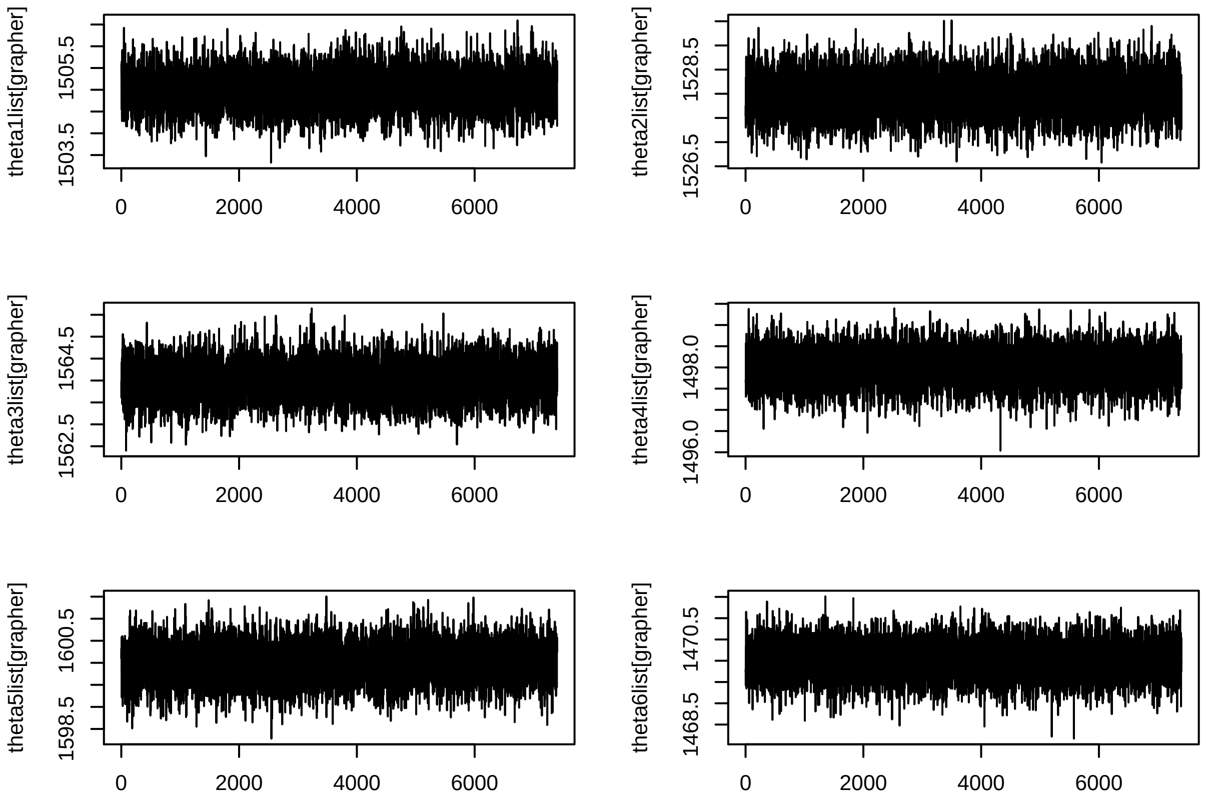
```
acf(h7list,main="ACF of Gibbs Sampler");
```



```
par(mfrow=c(2,2), mai = c(.4, .7, .3, .2),mgp=c(3,1,0));
#Plotting only every 25th point between B+1 and M
grapher <- seq(from=B+1,M,by=25);
plot(h7list[grapher], type='l'); plot(vlist[(B+1):M], type='l');
plot(wlist[grapher], type='l'); plot(mulist[(B+1):M], type='l');
```



```r
# acf(hlist)

par(mfrow=c(3,2), mai = c(.4, .6, .3, .2),mgp=c(3,1,0));
plot(theta1list[grapher], type='l'); plot(theta2list[grapher], type='l');
plot(theta3list[grapher], type='l'); plot(theta4list[grapher], type='l');
plot(theta5list[grapher], type='l'); plot(theta6list[grapher], type='l');
```



Discussion

The below chart provides a succint comparison of the above three methods for the unreasonable values $a_i = b_i = 100$ for $i \in \{1, 2, 3\}$:

row.names....names2	meanh	varfact	truese	runtime
Metropolis	-2.8609831	0.8863386	2.4316859	34.668
Metro-within-Gibbs	-0.0254390	9.4311310	0.0218538	49.539
Gibbs Sampler	0.0000151	1.0564781	0.0000000	24.709

The unreasonable values pose a challenge for all three algorithms given how much they affect the prior distributions; and although they do not yield a final estimate of the posterior mean of W/V that agrees with the algorithms ran with reasonable values, both the Gibbs Sampler and the Metropolis-Within-Gibbs approach the value 0. However, the jagged trace plots of the Metro-within-Gibbs suggest that the estimates themselves may not be reliable as there does not appear to be any approach to convergence.

The Gibbs Sampler shows the most symmetric convergence pattern for all parameters, as well as for the quantity of interest W/V, while the Metropolis and Variable-at-a-Time Metropolis algorithms show very great volatility when started on the unreasonable parameters. Recall that this was not the case when we used the reasonable starting values.

Based on the Gibbs's Sampler convergence pattern alone, coupled with the minute size of its standard errors, and its second-fast running time, the Gibbs's Sampler comes on top of the other two methods. Nonetheless, the set-up for this method was time-consuming due to the derivations needed for the conditional distributions. In the case of reasonable and unreasonable values, it appears to be the most consistent performer of these three algorithms.