Lecture 19 Rscript, Hamiltonian Monte Carlo

Example dataset

Consider a logistic regression,

$$\begin{aligned} y_i &\sim \text{Bin}(n_i, p_i) \\ \log(p_i/(1-p_i)) &= \mathbf{x}_i'\boldsymbol{\beta}. \end{aligned}$$

Clinical researchers are trying to determine the optimal dose level of a new medication for influenza. A group of influenza patients were recruited, and administered various doses of the experimental medication. One week after administration of medication, the researcher examined the patients to see if symptoms had improved. You were asked to investigate if there is a linear relationship between the log-odds of symptom improvement and medication dose.

Assume a flat prior for $\boldsymbol{\beta}$ so that your results are more comparable to a standard logistic GLM.

	Dosage: x						
y_i	0	1	2	3	4	5	6
No improvement: 0	8	5	6	6	3	5	1
Improvement: 1	2	5	4	4	7	5	9

Methods to consider

In this document, we will compare the performance of the Metropolis alogrithm to the Hamiltonian Monte Carlo algorithm when fitting a logistic regression.

Function one: Metropolis algorithm for fitting logistic regression

```
#Part one: function for performing Metropolis sampling for logistic regression normal random walk.
#Inputs:
#y: vector of responses
#n: vector (or scalar) of trial sizes.
#X: predictor matrix including intercept.
#c: rescaling for variance-covariance matrix, scalar J(theta*|theta(t-1)) = N(theta(t-1), c^2*Sigma)
#Sigma is variance covariance matrix for parameters in J()
#iter: number of iterations
#burnin: number of initial iterations to throw out.

Metropolis.fn<-function(y,n,X,c,Sigma,iter,burnin){
p <-dim(X)[2]   #number of parameters
library(mvtnorm)
theta0<-rnorm(p) #initial values.
theta.sim<-matrix(0,iter,p+1) #matrix to store iterations
```

```
theta.sim[1,1:p]<-theta0

for(i in 1:(iter-1)){
theta.cand <-rmvnorm(1,mean=theta.sim[i,1:p],sigma=c^2*Sigma) #draw candidate (jointly)
theta.cand <-as.numeric(theta.cand)
xbc        <-X%*%theta.cand
p.c        <-(1+exp(-xbc))^(-1)   #Calculating probability of success for candidates.
xb         <-X%*%theta.sim[i,1:p]
p.b        <-(1+exp(-xb))^(-1)     #Calculating probability of success for theta(t-1).
#difference of log joint distributions.
r<-sum( dbinom(y,size=n,prob=p.c,log=TRUE)-dbinom(y,size=n,prob=p.b,log=TRUE) )
#Draw an indicator whether to accept/reject candidate
ind<-rbinom(1,1,exp( min(c(r,0)) ) )
theta.sim[i+1,1:p]<- ind*theta.cand + (1-ind)*theta.sim[i,1:p]
theta.sim[i+1,p+1]<- ind #counter for acceptances.
}

#Removing the iterations in burnin phase
results<-theta.sim[-c(1:burnin),]
names(results)<-c('beta0','beta1','accept') #column names
return(results)
}
```

Function two: Hamiltonian Monte Carlo algorithm for fitting logistic regression

```
#Part one: function for performing Hamiltonian Monte Carlo for logistic regression.
#Inputs:
#y: vector of responses
#n: vector (or scalar) of trial sizes.
#X: predictor matrix including intercept.
#L: number of leapfrog steps.
#M is variance covariance matrix for normal prior of momentum variable \phi. Ideally diagonal.
#iter: number of iterations
#burnin: number of initial iterations to throw out.
HMC.fn<-function(y,n,X,L,M,iter,burnin){
p <-dim(X)[2]   #number of parameters
library(mvtnorm)
theta0<-rnorm(p) #initial values of beta
theta.sim<-matrix(0,iter,p+1) #matrix to store iterations plus acceptance.
theta.sim[1,1:p]<-theta0       #initial values in matrix.
epsilon<-1/L                #epsilon assuming epsilon*L =1.
Minv   <-solve(M)


for(i in 1:(iter-1)){
phi       <-rmvnorm(1,mean=rep(0,p),sigma=M)   #draw momentum variable.
phi       <-as.numeric(phi)
phi0      <-phi                              #saving starting phi for calculation of r.
theta     <-theta.sim[i,1:p]                  #current state of theta.

p.b         <-(1+exp(-X%*%theta))^(-1)  #calculate probabilities of success at current state.
gradtheta <- crossprod(X,y-n*p.b)
         #Gradient of posterior = joint distribution with respect to theta.
```

```
#leapfrog steps.
for(j in 1:L){
  phi   <- phi + 0.5*epsilon*gradtheta   #first half step for phi
  theta <- theta + epsilon*(Minv%*%phi)  #full step for theta

p.c <-(1+exp(-X%*%theta))^(-1) #calculate probabilities of success at candidate (sub) state.
gradtheta <- crossprod(X,y-n*p.c)
 #Gradient of posterior = joint distribution with respect to theta.

phi   <- phi + 0.5*epsilon*gradtheta #second half step for phi.
phi   <- as.numeric(phi)
}

#difference of log joint distributions at final iteration of leap.frog vs current state.
r<-sum( dbinom(y,size=n,prob=p.c,log=TRUE))+dmvnorm(phi,mean=rep(0,p),sigma=M,log=TRUE)-
  sum(dbinom(y,size=n,prob=p.b,log=TRUE) )-dmvnorm(phi0,mean=rep(0,p),sigma=M,log=TRUE)
#Draw an indicator whether to accept/reject candidate
ind<-rbinom(1,1,exp( min(c(r,0)) ) )
theta.sim[i+1,1:p]<- ind*theta + (1-ind)*theta.sim[i,1:p]
theta.sim[i+1,p+1] <- ind
}

#Removing the iterations in burnin phase
results<-theta.sim[-c(1:burnin),]
names(results)<-c('beta0','beta1','accept') #column names

return(results)
}
```

Formatting the data and running the code

```
x <-0:6                 #doses
y<-c(2,5,4,4,5,7,9)     #responses
n<-10                   #number in each group.



#formatting data into the correct format.
#Build predictor matrix.
pred.mat <-cbind(rep(1,length(y)),x) #First column intercept, second column dose levels

#Estimating a good choice of Sigma for the proposal distribution.
pest    <-y/n                   #Estimates of probability of success
logitest<-log(pest/(1-pest))    #Estimates of logits.
modest  <-lm(logitest~x)        #fit lm with estimated logits as response.
#variance-covariance matrix extracted from lm object.
sigma  <-vcov(modest)

#Running Metropolis algorithm
system.time(metro1<-Metropolis.fn(y=y,n=n,X=pred.mat,c=2.4/sqrt(2),Sigma=sigma,iter=10000,burnin=1000))
```

```
## Warning: package 'mvtnorm' was built under R version 4.3.1
```

```
##    user  system elapsed
##   0.806   0.016   0.822
```

```
system.time(metro2<-Metropolis.fn(y=y,n=n,X=pred.mat,c=2.4/sqrt(2),Sigma=sigma,iter=10000,burnin=1000))
```

```
##    user  system elapsed
##   0.757   0.013   0.770
```

```
system.time(metro3<-Metropolis.fn(y=y,n=n,X=pred.mat,c=2.4/sqrt(2),Sigma=sigma,iter=10000,burnin=1000))
```

```
##    user  system elapsed
##   0.758   0.011   0.770
```

```
#Choosing candidate M
#Md <-1/diag(sigma)
M  <- 1*solve(sigma)

system.time(HMC1<-HMC.fn(y=y,n=n,X=pred.mat,L=1,M=M,iter=10000,burnin=1000))
```

```
##    user  system elapsed
##   1.801   0.028   1.829
```

```
system.time(HMC2<-HMC.fn(y=y,n=n,X=pred.mat,L=1,M=M,iter=10000,burnin=1000))
```

```
##    user  system elapsed
##   1.772   0.014   1.787
```

```
system.time(HMC3<-HMC.fn(y=y,n=n,X=pred.mat,L=1,M=M,iter=10000,burnin=1000))
```

```
##    user  system elapsed
##   1.815   0.030   1.846
```

```
#Posterior means of beta0, beta1, Acceptance rate comparison
#Metropolis
metro.all<-rbind(metro1,metro2,metro3)
colMeans(metro.all)
```

```
## [1] -1.1992547  0.4224638  0.4157407
```

```
#Hamiltonian Monte Carlo
HMC.all <- rbind(HMC1,HMC2,HMC3)
colMeans(HMC.all)
```

```
## [1] -1.2061869  0.4244635  0.9233333
```

```
#Posterior standard deviations
apply(metro.all,2,FUN=sd)
```

```
## [1] 0.4874671 0.1395160 0.4928584
```

```r
apply(HMC.all,2,FUN=sd)
```

```
## [1] 0.4950979 0.1425928 0.2660667
```

```r
#95 % credible intervals
apply(metro.all,2,FUN=function(x) quantile(x,c(0.025,0.975)) )
```

```
##              [,1]      [,2] [,3]
## 2.5%  -2.1999567 0.1577053    0
## 97.5% -0.2625892 0.7108926    1
```

```r
apply(HMC.all,2,FUN=function(x) quantile(x,c(0.025,0.975)) )
```

```
##              [,1]      [,2] [,3]
## 2.5%  -2.2199719 0.1609075    0
## 97.5% -0.2769197 0.7173635    1
```

```r
#Convergence diagnostics.

library(coda)
```

```
## Warning: package 'coda' was built under R version 4.3.1
```

```r
#splitting metropolis chains for beta0,beta1 and converting into mcmc objects.
ml1<-as.mcmc.list(as.mcmc((metro1[1:4500,1:2])))
ml2<-as.mcmc.list(as.mcmc((metro2[1:4500,1:2])))
ml3<-as.mcmc.list(as.mcmc((metro3[1:4500,1:2])))
ml4<-as.mcmc.list(as.mcmc((metro1[4500+1:4500,1:2])))
ml5<-as.mcmc.list(as.mcmc((metro2[4500+1:4500,1:2])))
ml6<-as.mcmc.list(as.mcmc((metro3[4500+1:4500,1:2])))
ml<-c(ml1,ml2,ml3,ml4,ml5,ml6)

#Gelman-Rubin diagnostic.
gelman.diag(ml)[[1]]
```

```
##      Point est. Upper C.I.
## [1,]   1.000792   1.001723
## [2,]   1.002886   1.006623
```

```r
#effective sample size.
effectiveSize(ml)
```

```
##     var1     var2
## 3497.187 3269.565
```

```r
#splitting Hamiltonian Monte Carlo chains for beta0,beta1 and converting into mcmc objects.
hl1<-as.mcmc.list(as.mcmc((HMC1[1:4500,1:2])))
hl2<-as.mcmc.list(as.mcmc((HMC2[1:4500,1:2])))
hl3<-as.mcmc.list(as.mcmc((HMC3[1:4500,1:2])))
```

```
hl4<-as.mcmc.list(as.mcmc((HMC1[4500+1:4500,1:2])))
hl5<-as.mcmc.list(as.mcmc((HMC2[4500+1:4500,1:2])))
hl6<-as.mcmc.list(as.mcmc((HMC3[4500+1:4500,1:2])))
hl<-c(hl1,hl2,hl3,hl4,hl5,hl6)

#Gelman-Rubin diagnostic.
gelman.diag(hl)[[1]]
```

```
##      Point est. Upper C.I.
## [1,]   1.001126   1.003026
## [2,]   1.001155   1.003011
```

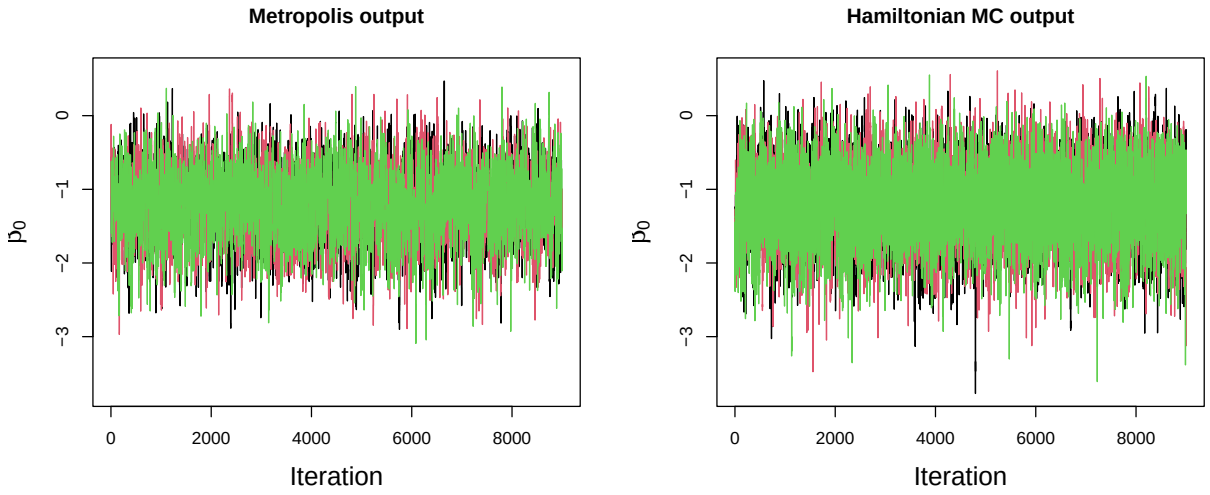
```
#effective sample size.
effectiveSize(hl)
```

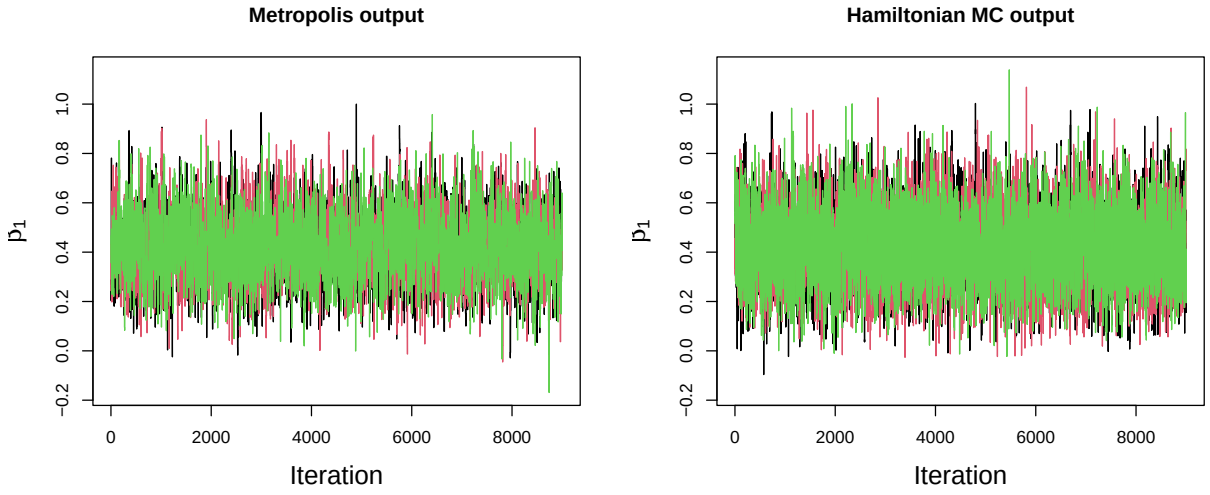
```
##     var1     var2
## 5763.638 5353.791
```

```
#Checking convergence of combined Metropolis, Hamiltonian MC.
#Gelman-Rubin diagnostic.
gelman.diag(c(ml,hl))[[1]]
```

```
##      Point est. Upper C.I.
## [1,]   1.000815    1.00169
## [2,]   1.001870    1.00374
```

```
#plotting HMC vs Metropolis iterations.

ylim1<-apply(rbind(metro.all,HMC.all),2,min)
ylim2<-apply(rbind(metro.all,HMC.all),2,max)
par(mfrow=c(2,2),mar=c(9.2,4.1,4.1,2.5))
for(i in 1:2){
plot(metro1[,i],type='l',ylim=c(ylim1[i],ylim2[i]),main='Metropolis output',
     xlab='Iteration',ylab=bquote( beta[.(i-1)] ),cex.lab=1.5)
lines(metro2[,i],col=2)
lines(metro3[,i],col=3)
plot(HMC1[,i],type='l',ylim=c(ylim1[i],ylim2[i]),main='Hamiltonian MC output',
     xlab='Iteration',ylab=bquote( beta[.(i-1)] ),cex.lab=1.5)
lines(HMC2[,i],col=2)
lines(HMC3[,i],col=3)
}
```





What happens if we do not re-sample ϕ from $p(\phi)$.

```
#Incorrect HMC. In this version, the Hamiltonian is always conserved.

#Inputs:
#y: vector of responses
#n: vector (or scalar) of trial sizes.
#X: predictor matrix including intercept.
#L: number of leapfrog steps.
#M is variance covariance matrix for normal prior of momentum variable \phi. Ideally diagonal.
#iter: number of iterations
#burnin: number of initial iterations to throw out.
HMCwrong.fn<-function(y,n,X,L,M,iter,burnin){
p <-dim(X)[2]    #number of parameters
```

```
library(mvtnorm)
theta0<-c(-1.2,0.425)#rnorm(p) #initial values of beta
theta.sim<-matrix(0,iter+1,p+1) #matrix to store iterations plus acceptance.
theta.sim[1,1:p]<-theta0       #initial values in matrix.
epsilon<-1/L                #epsilon assuming epsilon*L =1.
Minv   <-solve(M)

phi       <-rmvnorm(1,mean=rep(0,p),sigma=M)   #draw momentum variable.
phi       <-as.numeric(phi)
phi0      <-phi                              #saving starting phi for calculation of r.
theta     <-theta.sim[1,1:p]                 #Starting value for theta.

for(i in 2:(L*iter)){
                #current state of theta.

p.b         <-(1+exp(-X%*%theta))^(-1)  #calculate probabilities of success at current state.
gradtheta <- crossprod(X,y-n*p.b)
  #Gradient of posterior = joint distribution with respect to theta.

#leapfrog steps.
phi   <- phi + 0.5*epsilon*gradtheta   #first half step for phi
theta <- theta + epsilon*(Minv%*%phi)  #full step for theta

p.c         <-(1+exp(-X%*%theta))^(-1) #calculate probabilities of success at candidate (sub) state.
gradtheta <- crossprod(X,y-n*p.c)
  #Gradient of posterior = joint distribution with respect to theta.

phi   <- phi + 0.5*epsilon*gradtheta #second half step for phi.
phi   <- as.numeric(phi)

#difference of log joint distributions at final iteration of leap.frog vs current state.
r<-sum( dbinom(y,size=n,prob=p.c,log=TRUE))+dmvnorm(phi,mean=rep(0,p),sigma=M,log=TRUE)-
  sum(dbinom(y,size=n,prob=p.b,log=TRUE) )-dmvnorm(phi0,mean=rep(0,p),sigma=M,log=TRUE)

#Draw an indicator whether to accept/reject candidate
if(abs(i/L - floor(i/L)) == 0 ){
ind<-rbinom(1,1,exp( min(c(r,0)) ) )
theta.sim[i/L+1,1:p]<- ind*theta + (1-ind)*theta.sim[i/L,1:p]
theta.sim[i/L+1,p+1] <- ind
phi0  <- phi
theta <- theta.sim[i/L+1,1:p]
}
}


#Removing the iterations in burnin phase
results<-theta.sim[-c(1:burnin),]
names(results)<-c('beta0','beta1','accept') #column names

return(results)
}
```

Running incorrect HMC to see the impact

```
M  <- 50*diag(2)

system.time(HMCwrong1<-HMCwrong.fn(y=y,n=n,X=pred.mat,L=2,M=M,iter=10000,burnin=1000))
```

```
##    user  system elapsed
##   1.945   0.024   1.970
```

```
system.time(HMCwrong2<-HMCwrong.fn(y=y,n=n,X=pred.mat,L=2,M=M,iter=10000,burnin=1000))
```

```
##    user  system elapsed
##   1.919   0.027   1.947
```

```
system.time(HMCwrong3<-HMCwrong.fn(y=y,n=n,X=pred.mat,L=2,M=M,iter=10000,burnin=1000))
```

```
##    user  system elapsed
##   1.930   0.029   1.960
```

```
#Posterior means of beta0, beta1, Acceptance rate comparison
#Metropolis
metro.all<-rbind(metro1,metro2,metro3)
colMeans(metro.all)
```

```
## [1] -1.1992547  0.4224638  0.4157407
```

```
#Hamiltonian Monte Carlo
HMC.all <- rbind(HMC1,HMC2,HMC3)
colMeans(HMC.all)
```

```
## [1] -1.2061869  0.4244635  0.9233333
```

```
#Incorrect Hamiltonian Monte Carlo
HMCwrong.all <- rbind(HMCwrong1,HMCwrong2,HMCwrong3)
colMeans(HMCwrong.all)
```

```
## [1]  2.4300449 -0.5104629  0.0000000
```

```
#Posterior standard deviations
apply(metro.all,2,FUN=sd)
```

```
## [1] 0.4874671 0.1395160 0.4928584
```

```
apply(HMC.all,2,FUN=sd)
```

```
## [1] 0.4950979 0.1425928 0.2660667
```

```r
apply(HMCwrong.all,2,FUN=sd)
```

```
## [1] 2.5037436 0.4947792 0.0000000
```

```r
#95 % credible intervals
apply(metro.all,2,FUN=function(x) quantile(x,c(0.025,0.975)) )
```

```
##              [,1]      [,2] [,3]
## 2.5%  -2.1999567 0.1577053    0
## 97.5% -0.2625892 0.7108926    1
```

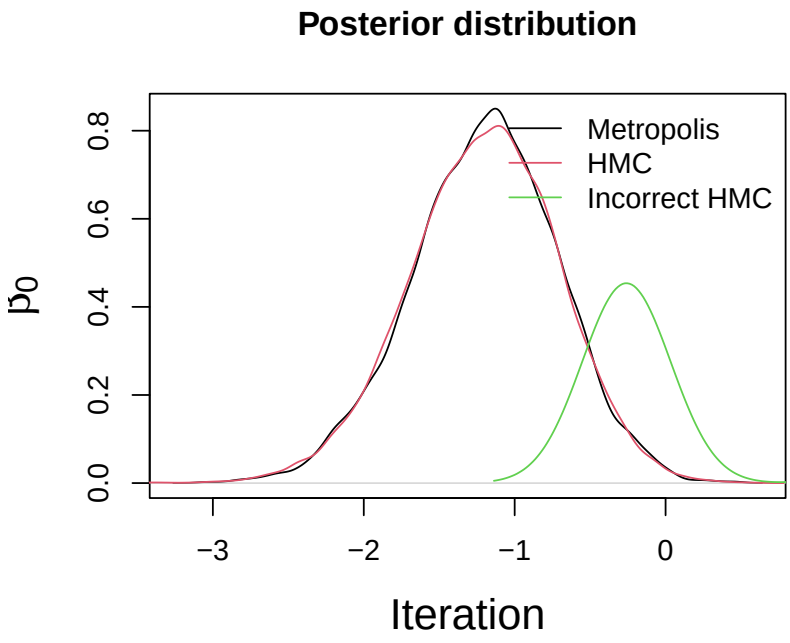
```r
apply(HMC.all,2,FUN=function(x) quantile(x,c(0.025,0.975)) )
```

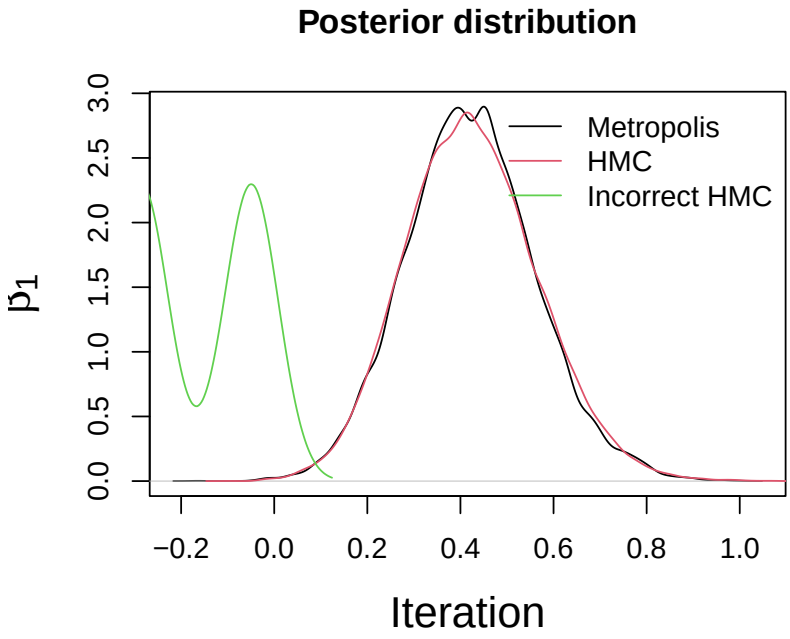
```
##              [,1]      [,2] [,3]
## 2.5%  -2.2199719 0.1609075    0
## 97.5% -0.2769197 0.7173635    1
```

```r
apply(HMCwrong.all,2,FUN=function(x) quantile(x,c(0.025,0.975)) )
```

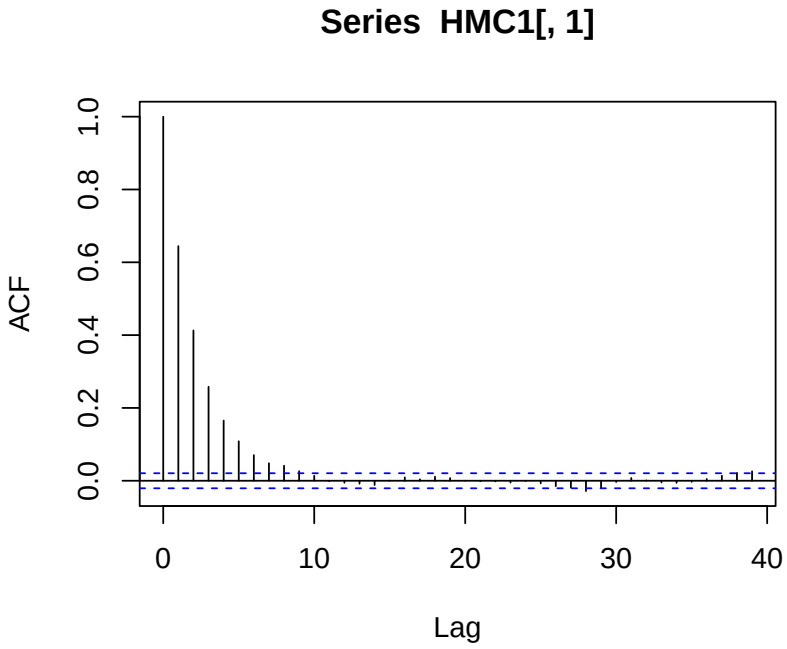
```
##              [,1]        [,2] [,3]
## 2.5%  -0.2594823 -1.19678450    0
## 97.5%  5.7691963 -0.04932706    0
```

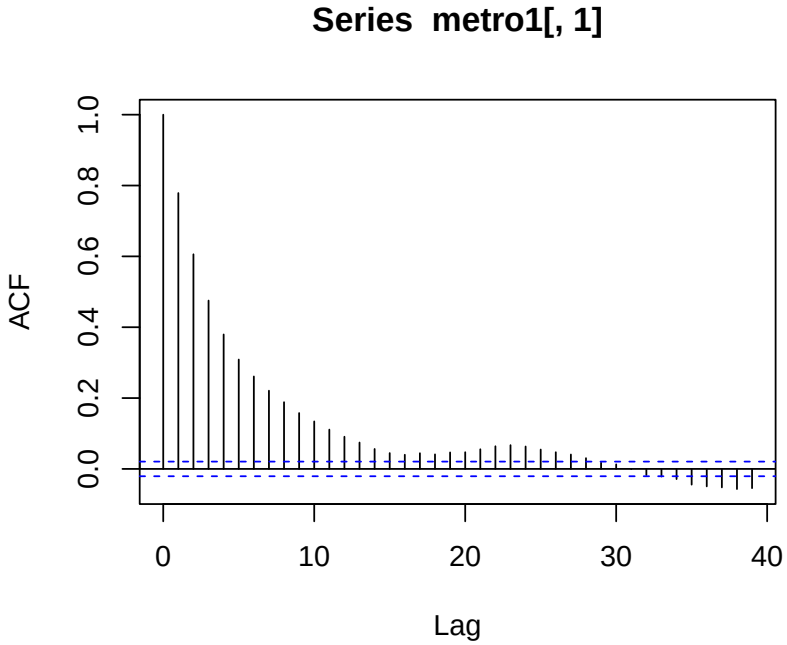
```r
#plotting HMC vs Metropolis emprical densities.


par(mfrow=c(2,1),mar=c(9.2,4.1,4.1,2.5))
for(i in 1:2){
plot(density(metro.all[,i]),main='Posterior distribution',xlab='Iteration',
     ylab=bquote( beta[.(i-1)] ),cex.lab=1.5)
lines(density(HMC.all[,i]),col=2)
lines(density(HMCwrong.all[,i]),col=3)
legend('topright',legend=c('Metropolis','HMC','Incorrect HMC'),col=1:3,lty=1,bty='n')
}
```


Posterior distribution
Metropolis
HMC
Incorrect HMC
β0
0.0
0.2
0.4
0.6
0.8
−3
−2
−1
0
Iteration


Posterior distribution
Metropolis
HMC
Incorrect HMC
β1
0.0
0.5
1.0
1.5
2.0
2.5
3.0
−0.2
0.0
0.2
0.4
0.6
0.8
1.0
Iteration

```
par(mfrow=c(2,1),mar=c(9.2,4.1,4.1,2.5))
acf(HMC1[,1])
acf(metro1[,1])
```


Series HMC1[, 1]
ACF
0.0
0.2
0.4
0.6
0.8
1.0
0
10
20
30
40
Lag


Series metro1[, 1]
ACF
0.0
0.2
0.4
0.6
0.8
1.0
0
10
20
30
40
Lag