Bayesian Learning Lab 3

Mohamed Ali - Mohal954

2023-05-15

Gibbs sampler for a normal model

A

Implement (code!) a Gibbs sampler that simulates from the joint posterior $p(\mu, \sigma^2|lny_1, ..., lnyn)$. The full conditional posteriors are given on the slides from Lecture 7.
Evaluate the convergence of the Gibbs sampler by calculating the Inefficiency Factors (IFs) and by plotting the trajectories of the sampled Markov chains.
The Gibbs Sampling algorithm:

1- Choose inital values $\theta_2^0, \theta_3^0, ..., \theta_n^0$.
2- Repeat for j=1,. . . ,N:
- Draw θ_1^j from $p(\theta_1|\theta_2^{j-1}, ..., \theta_k^{j-1})$.
- Draw θ_2^j from $p(\theta_2|\theta_1^j, ..., \theta_k^{j-1})$.
.
.
.
3- Draw θ_k^j from $p(\theta_k|\theta_1^j, ..., \theta_{k-1}^j)$.
4- Return draws: θ^1,. . . ,θ^N, where θ^j=(θ_1^j,. . . ,θ_k^j).

$$\bar{\theta} = 1/N \sum_{t=1}^{N} \theta^t$$

$$var(\bar{\theta}) = \frac{\sigma^2}{N}$$

Autocorrelated samples :

$$var(\bar{\theta}) = \frac{\sigma^2}{N}(1 + 2\sum_{k=1}^{\inf} \rho_k)$$

Convergence diagnostics:

$$var(\bar{\theta}) = \frac{\sigma^2}{N}(1 + 2\sum_{k=1}^{\inf} \rho_k)$$

$$IF = 1 + 2\sum_{k=1}^{\inf} \rho_k$$

$$ESS = N/IF$$

We start by building our function using the Slides notes we can define the Normal Model with Conditionally conjugate prior as:

$$\mu \sim N(\mu_0, \tau_0^2)\sigma \quad \sim Inv - \chi^2(v_0^2, \sigma_0^2)$$

And the full conditional posteriors:

$$\mu|\sigma^2, x \sim N(\mu_n, \tau_n^2)$$
$$\sigma^2|\mu, x \sim Inv - \chi^2(v_n, \frac{v_0\sigma^2 + \sum_{i=1}^{n}(x_i - \mu)^2}{n + v_0})$$

Where we have:

$$\mu_n = w\bar{x} + (1-w)\mu_0$$
$$w = \frac{\frac{n}{\sigma^2}}{\frac{n}{\sigma^2} + \frac{1}{\tau_0^2}}$$
$$\frac{1}{\tau_n^2} = \frac{n}{\sigma^2} + \frac{1}{\tau_0^2}$$
$$v_n = v_0 + n$$

We start by defining the above variables in R, Note that as we have $p(lny_1, ..., lnyn|\mu, \sigma^2) \sim N(\mu, \sigma^2)$ we will transform our variable by taking the natural log of the daily precipitation.

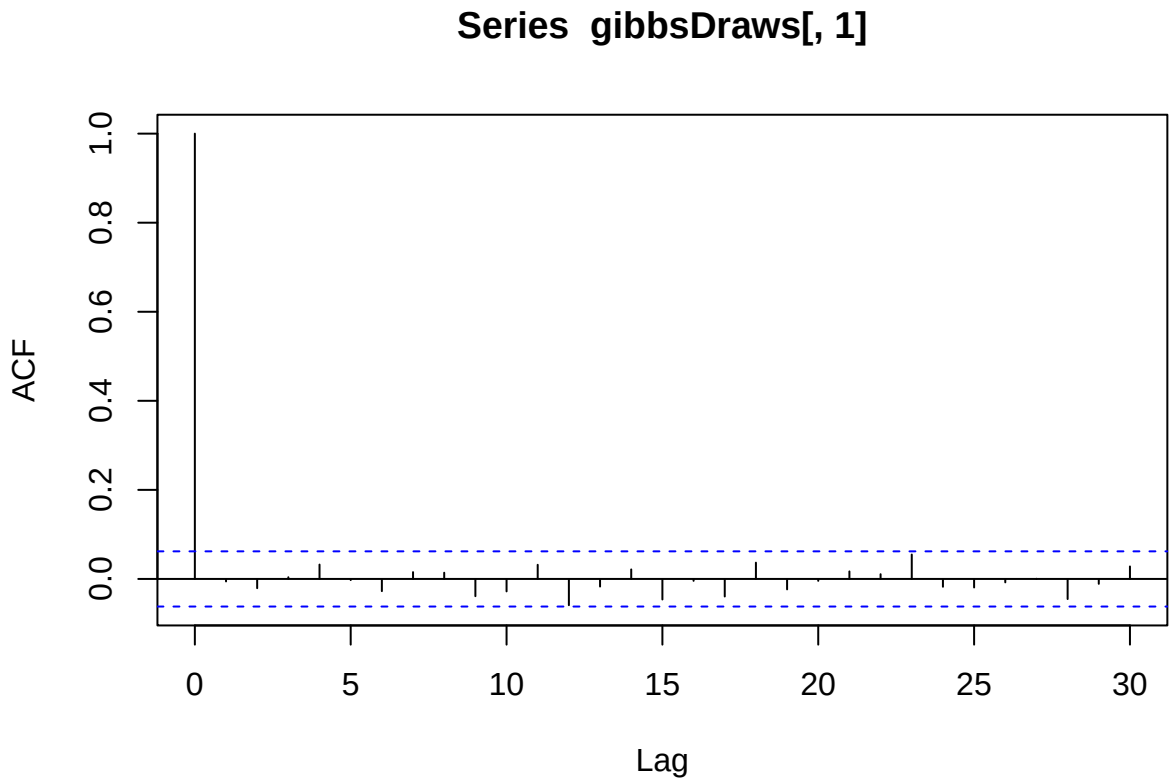
```
# we start by reading our dataset using the readRDS command in R
df <- data.frame(x=readRDS("Precipitation.rds"))
# in the task we looking at natural log of the daily precipitation
#lny1,lny2,lny3,...lnyn follows N(mu,sigma)
# We add a new var called logx
df$logx<- log(df$x)
sigma2_0<- var(df$logx)  # Sample var
tau2_0<- 1# arbirtry initail value let it be 1
n<- nrow(df)# Sample size
mu_0<- mean(df$logx)#sample mean
w<- (n/sigma2_0)/((n/sigma2_0) + (1/tau2_0))# value used to calculate mu_n
v_0<- 1 # arbirtry initail value let it be 1
v_n<- v_0+n
mu_n<- w*(mean(df$logx))+ (1-w)*mu_0
 ###### This part of the code has been modified######
tau2_n<- 1/((n/sigma2_0)+(1/tau2_0))################## shoud be 1/tau
#####################################################
nDraws<- 1000
gibbsDraws <- matrix(0,nDraws,2)
```

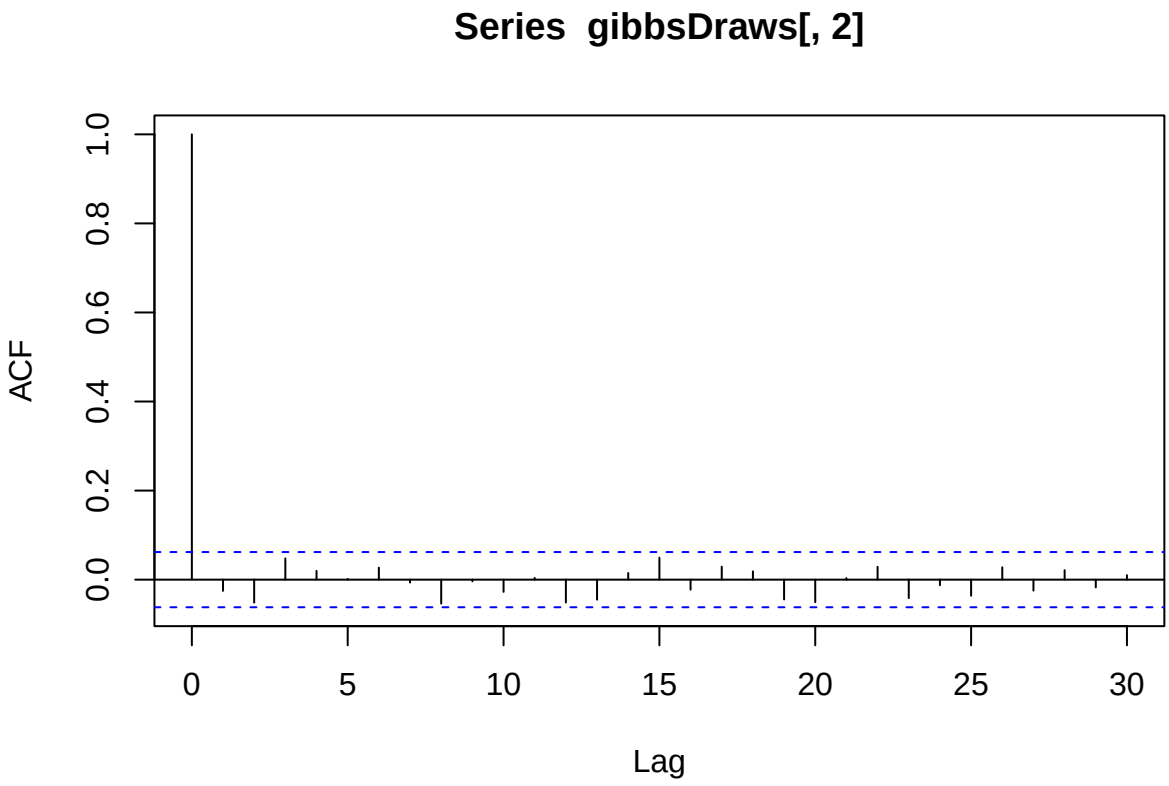
Now we write a code to simulates from the joint posterior using Gibbs sampler.

```
### From lec notes we can use this part of the code
  ###### This part of the code has been modified######
scale=sigma2_0 ########################################
for (i in 1:nDraws){###################################
  w<- (n/scale)/((n/scale) + (1/tau2_0)) #############
  mu_n<- w*(mean(df$logx))+ (1-w)*mu_0 ############### We need to update those parameters as well
  tau2_n<- 1/((n/scale)+(1/tau2_0))##################
  ####################################################
  # Update theta1 ----> mu given theta2
  theta1 <- rnorm(1, mean = mu_n, sd = (tau2_n))
```

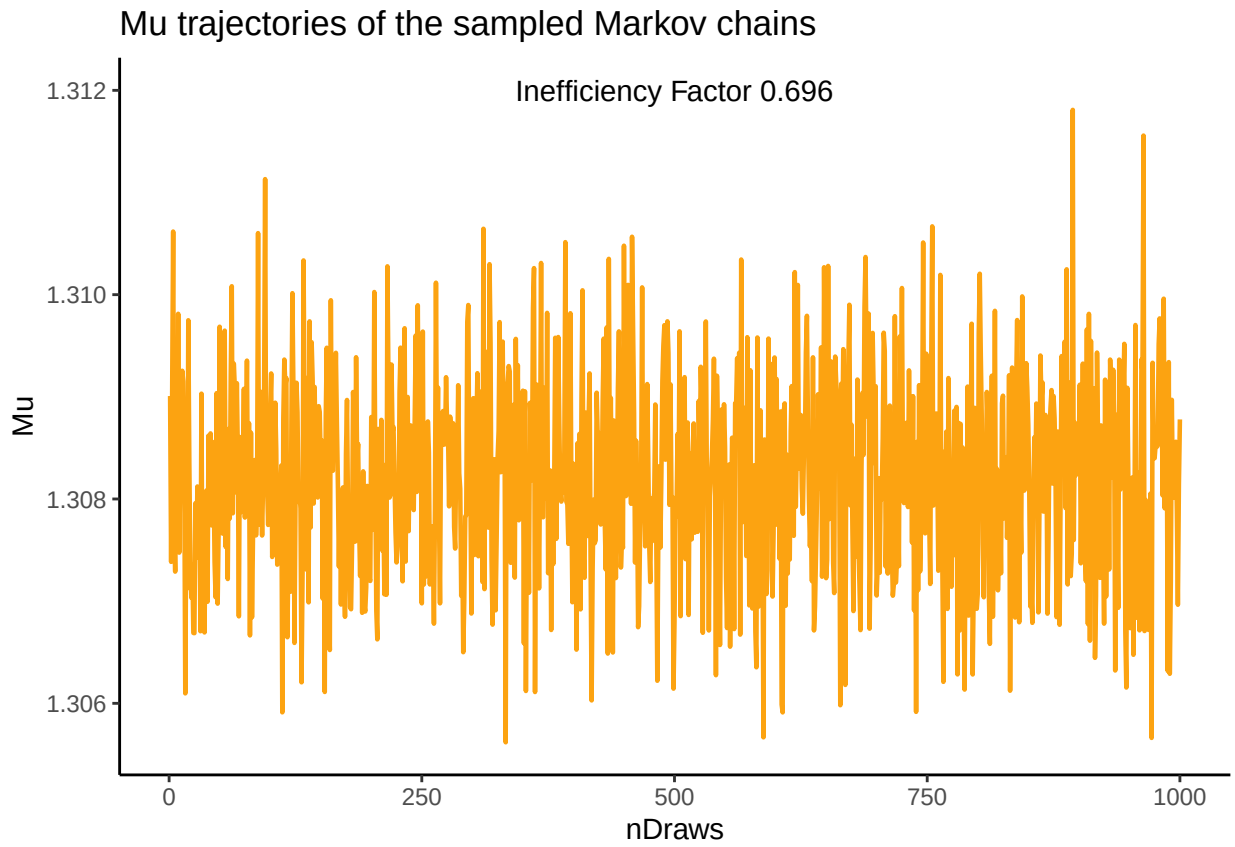
```
  gibbsDraws[i,1] <- theta1

  scale<- ((v_0*sigma2_0)+(sum((df$logx-theta1)^2)))/(n+v_0)
  # Update theta2 ----> sigma2 given theta1
  theta2 <- rinvchisq(1,v_n,scale=scale)
  gibbsDraws[i,2] <- theta2
}
```

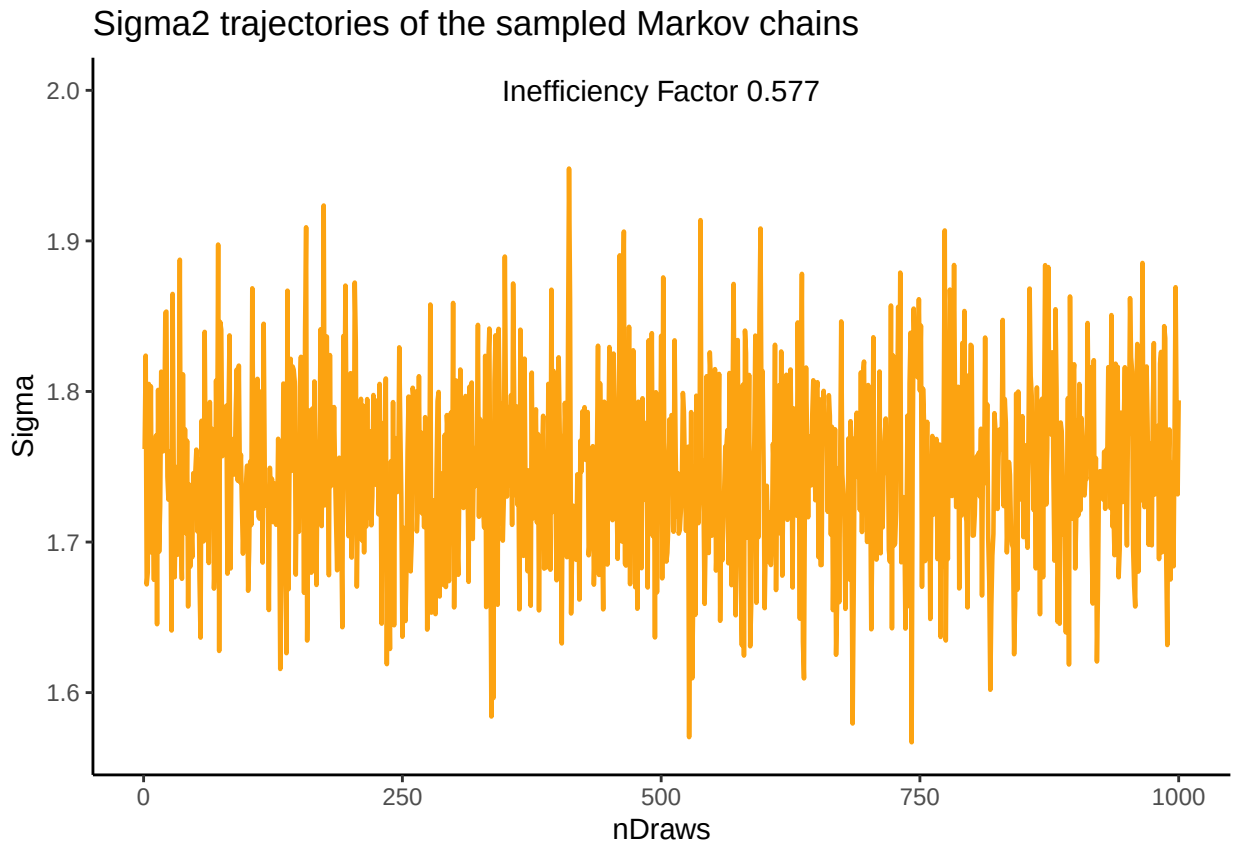
To Evaluate the convergence of the Gibbs sampler we calculate the Inefficiency Factors (IFs) $1 + 2\sum_{k=1}^{\infty} \rho_k$ - where ρ_k is the autocorrelation at lag k - and then we plot the trajectories of the sampled Markov chains.






```
## Warning: Using ‘size‘ aesthetic for lines was deprecated in ggplot2 3.4.0.
## i Please use ‘linewidth‘ instead.
## This warning is displayed once every 8 hours.
## Call ‘lifecycle::last_lifecycle_warnings()‘ to see where this warning was
## generated.
```


Mu trajectories of the sampled Markov chains
Inefficiency Factor 0.696
1.312
1.310
1.308
1.306
Mu
0
250
500
750
1000
nDraws

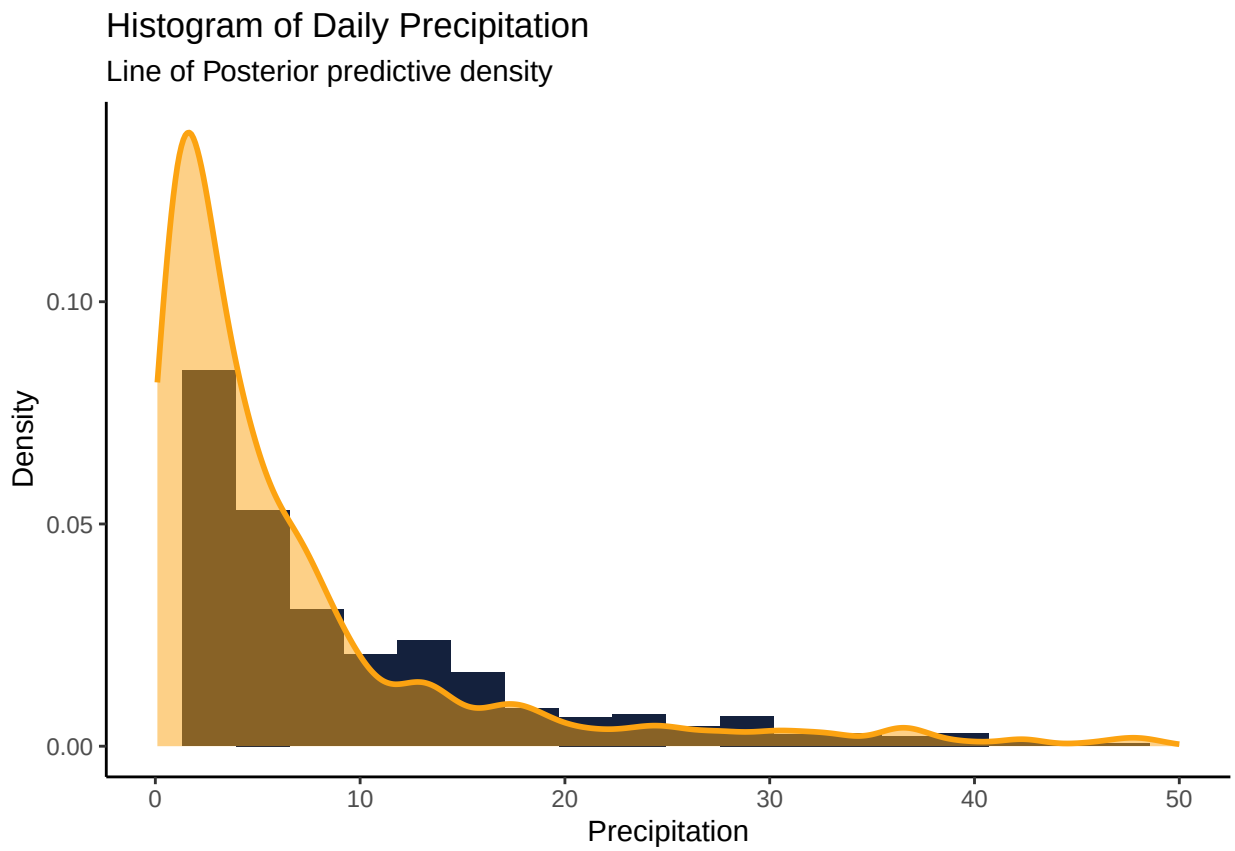


The graph above shows the convergence of the Gibbs sampler and the Inefficiency Factors, the IF tell us number of iterations needed to get an independent sample from the posterior distribution, as we can see μ and σ^2 have 0,69 and 0,57 IF respectively, which tell us a better the convergence of the Gibbs sampler for the σ^2.

B

Plot the following in one figure:
1) a histogram or kernel density estimate of the daily precipitation $(y_1, ..., y_n)$.
2)The resulting posterior predictive density $p(\tilde{y}|y_1, ..., y_n)$ using the simulated posterior draws from (a).
How well does the posterior predictive density agree with this data?

To do we draw a sample from rnorm with mean and sd from the posterior results we got in (a).



Metropolis Random Walk for Poisson regression

A

Obtain the maximum likelihood estimator of β in the Poisson regression model for the eBay data [Hint: glm.R, don't forget that glm() adds its own intercept so don't input the covariate Const].
Which covariates are significant?

```
##
## Call:
## glm(formula = nBids ~ ., family = poisson, data = df1[, -2])
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -3.5800  -0.7222  -0.0441   0.5269   2.4605
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept)  1.07244    0.03077  34.848  < 2e-16 ***
## PowerSeller -0.02054    0.03678  -0.558   0.5765
## VerifyID    -0.39452    0.09243  -4.268 1.97e-05 ***
## Sealed       0.44384    0.05056   8.778  < 2e-16 ***
## Minblem     -0.05220    0.06020  -0.867   0.3859
```

```
## MajBlem     -0.22087    0.09144  -2.416   0.0157 *
## LargNeg      0.07067    0.05633   1.255   0.2096
## LogBook     -0.12068    0.02896  -4.166 3.09e-05 ***
## MinBidShare -1.89410    0.07124 -26.588  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for poisson family taken to be 1)
##
##     Null deviance: 2151.28  on 999  degrees of freedom
## Residual deviance:  867.47  on 991  degrees of freedom
## AIC: 3610.3
##
## Number of Fisher Scoring iterations: 5
```

The model reuslts shows the coefficients and their standard errors for each predictor. The "Estimate" shows the estimated effect of each predictor on the expected number of bids. A positive coefficient indicates that the predictor is associated with an increase in the expected number of bids, while a negative coefficient indicates a decrease.
The "Pr(>|z|)" column shows the p-value for each coefficient. If this value is less than 0.05, the coefficient is considered statistically significant.
The model suggests that the variables *VerifyID, Sealed, LogBook, and MinBidShare* have a significant effect on the expected number of bids (p-value less that 0.01 (p<.01) at $\alpha = 0.01$), while the others do not.

B

Let's do a Bayesian analysis of the Poisson regression. Let the prior be $\beta \sim N(0, 100\ (X^T X)^{-1})$,where X is the n x p covariate matrix. This is a commonly used prior, which is called Zellner's g-prior. Assume frst that the posterior density is approximately multivariate normal:

$$\beta|y \sim N(\tilde{\beta}, J_y^{-1}(\tilde{\beta}))$$

where $\tilde{\beta}$ is the posterior mode and $J_y^{-1}(\tilde{\beta})$ is the negative Hessian at the posterior mode. $\tilde{\beta}$ and $J_y^{-1}(\tilde{\beta})$ can be obtained by numerical optimization (optim.R) exactly like you already did for the logistic regression in Lab 2 (but with the log posterior function replaced by the corresponding one for the Poisson model, which you have to code up.).

```
## [1] "The posterior mode is:"
```

```
##          [,1]        [,2]      [,3]      [,4]        [,5]       [,6]       [,7]
## [1,] 1.069841 -0.02051246 -0.393006 0.4435555 -0.05246627 -0.2212384 0.07069683
##            [,8]      [,9]
## [1,] -0.1202177 -1.891985
## attr(,"names")
## [1] "Const"       "PowerSeller" "VerifyID"    "Sealed"      "Minblem"
## [6] "MajBlem"     "LargNeg"     "LogBook"     "MinBidShare"
```

```
## [1] "The Hessian Matrix:"
```

```
##             [,1]        [,2]          [,3]          [,4]          [,5]
##  [1,] -3634.2841 -1574.88862 -1.284330e+02 -5.054825e+02 -3.089573e+02
##  [2,] -1574.8886 -1574.88862 -6.049186e+01 -3.260764e+02 -1.044615e+02
##  [3,]  -128.4330   -60.49186 -1.284330e+02 -6.148277e+01 -9.865299e+00
##  [4,]  -505.4825  -326.07643 -6.148277e+01 -5.054825e+02  1.705303e-07
##  [5,]  -308.9573  -104.46148 -9.865299e+00  1.705303e-07 -3.089573e+02
##  [6,]  -126.9303   -68.96966  2.273737e-07  0.000000e+00  0.000000e+00
##  [7,]  -385.7170   -53.05278 -1.136868e-07  0.000000e+00 -3.390566e+01
##  [8,]  -638.9730    71.79073 -6.887776e+01 -1.287304e+02 -2.229706e+01
##  [9,]   729.8896   146.21556  2.380017e+01  8.975677e+01  5.518223e+01
##                [,6]          [,7]        [,8]       [,9]
##  [1,] -1.269303e+02 -3.857170e+02  -638.97297  729.88956
##  [2,] -6.896966e+01 -5.305278e+01    71.79073  146.21556
##  [3,]  2.273737e-07 -1.136868e-07   -68.87776   23.80017
##  [4,]  0.000000e+00  0.000000e+00  -128.73043   89.75677
##  [5,]  0.000000e+00 -3.390566e+01   -22.29706   55.18223
##  [6,] -1.269303e+02  0.000000e+00   -36.39914   34.16904
##  [7,]  0.000000e+00 -3.857170e+02  -220.23559  115.38523
##  [8,] -3.639914e+01 -2.202356e+02 -1930.07936  534.16381
##  [9,]  3.416904e+01  1.153852e+02   534.16381 -446.88731

## [1] "The approximate posterior standard deviation is:"

##       Const PowerSeller    VerifyID      Sealed     Minblem     MajBlem
##  0.03074837  0.03678418  0.09227871  0.05057448  0.06020470  0.09146070
##     LargNeg     LogBook MinBidShare
##  0.05634767  0.02895635  0.07109682
```

C

Random walk Metropolis algorithm:

- Initialize θ^0 and iterate for i= 1,2,..
  1- Sample proposal: $\theta_p|\theta^{(i-1)} \sim N(\theta^{(i-1)}, c.\Sigma)$.
  2- Compute the acceptance probability ($\alpha = min(1, \frac{p(\theta_p|y)}{\theta^{(i-1)}|y)})$).
  3- With probability α true set $\theta^i = \theta_p$ and $\theta^i = \theta^{i-1}$.

Note

The draw backs of the MH usual find it hard to find proposal distribution and getting too small step size give us too many rejections.
Alternately we can use HM (Hamiltonian Monto Carlo), which solve this by giving distant proposal and high acceptance probabilities.

Let's simulate from the actual posterior of β using the Metropolis algorithm and compare the results with the approximate results in b).
Program a general function that uses the Metropolis algorithm to generate random draws from an arbitrary posterior density. In order to show that it is a general function for any model, we denote the vector of model parameters by θ. Let the proposal density be the multivariate normal density mentioned in Lecture

8 (random walk Metropolis):

$$\theta_p|\theta^{i-1} \sim N(\theta^{i-1}, c.\Sigma)$$

Where $\Sigma = J_y^{-1}(\tilde{\beta})$ was obtained in b). The value c is a tuning parameter and should be an input to your Metropolis function. The user of your Metropolis function should be able to supply her own posterior density function, not necessarily for the Poisson regression, and still be able to use your Metropolis function.
* Note that* on how to How to code up the Random Walk Metropolis Algorithm in R:
One of the input arguments of our *RWMSampler* function is *logPostFunc*. *logPostFunc* is a function object that computes the log posterior density at any value of the parameter vector.
This is needed when we compute the acceptance probability of the Metropolis algorithm. from (q2 a) we program the log posterior density, since logs are more stable and avoids problems with too small or large numbers (overflow).
The ratio of posterior densities in the Metropolis acceptance probability can be written as:

$$\frac{p(\theta_p|y)}{p(\theta^{i-1}|y)} = exp(log\ p(\theta_p|y) - log\ p(\theta^{i-1}|y))$$

The first argument our (log) posterior function is theta (theta_old,theta_new), the vector of parameters for which the posterior density is evaluated. You can of course use some other name for the variable, but it must be the first argument of your posterior density function.

Function RWMSampler

```
RWMSampler_old<- function(logPostFunc,nDraws,c,y,x,mu,Sigma){
  # First we buld our data frame of samples
  sample <- data.frame(matrix(nrow = nDraws, ncol = ncol(x)))
  colnames(sample) <- colnames(x)
  # The inital sample value c here represent a tuning parameter
  sample[1,] <- mvrnorm(1, posteriorMode, c*postCov)
  # Now we implement the Metropolis-Hastings
  #in which we generate samples from the proposal distribution in this case
  # We look at the results of the first sample
  #as theta_i-1 pluged in mvnorm to get theta_i and the we use the values in
  # our proposed logPostFunc
  counter <- 1
  i=1
  while (counter < nDraws) {
    theta_old<-as.numeric(sample[counter,])
    theta_new<-mvrnorm(1,theta_old,c*postCov)
  # We define th our accept/reject threshold
    th<-runif(1,0,1)
  # now we find the value of the target/proposed distribution
    proposed<- logPostFunc(theta_new,y = y,
                           x = x,
                           mu = posteriorMode,
                           Sigma = postCov)
    target<- logPostFunc(theta_old,y = y,
                         x = x,
                         mu = posteriorMode,
```

```
                        Sigma = postCov)
    # the ratio of posterior densities in the Metropolis acceptance probability
    if (th<min(1,exp(proposed-target))) {
      counter=counter+1
      sample[counter,]<-theta_new
    }
  }
 return(sample)
}
```

Old solution

```
RWMSampler<- function(logPostFunc,nDraws,c,y,x,mu,Sigma,brnin){
  # First we buld our data frame of samples
  sample <- data.frame(matrix(0,nrow = nDraws - brnin, ncol = ncol(x)))
  colnames(sample) <- colnames(x)
  # The inital sample value c here represent a tuning parameter
  # sample[1,] <- mvrnorm(1, posteriorMode, c*postCov)
  # Now we implement the Metropolis-Hastings
  #in which we generate samples from the proposal distribution in this case
  # We look at the results of the first sample
  #as theta_i-1 pluged in mvnorm to get theta_i and the we use the values in
  # our proposed logPostFunc

  theta_old<-as.numeric(sample[1,])
  for (i in 1:nDraws) {

    theta_new<-mvrnorm(1,theta_old,c*postCov)
    # We define th our accept/reject threshold
    th<-runif(1,0,1)
    # now we find the value of the target/proposed distribution
    proposed<- logPostFunc(theta_new,y = y,
                           x = x,
                           mu = posteriorMode,
                           Sigma = postCov)
    target<- logPostFunc(theta_old,y = y,
                         x = x,
                         mu = posteriorMode,
                         Sigma = postCov)
    # the ratio of posterior densities in the Metropolis acceptance probability
    if (th<exp(proposed-target)) {
      theta_old<-theta_new
    }
    if (i> brnin) {
      sample[i-brnin,]<-theta_old
    }
  }
  return(sample)
}
```

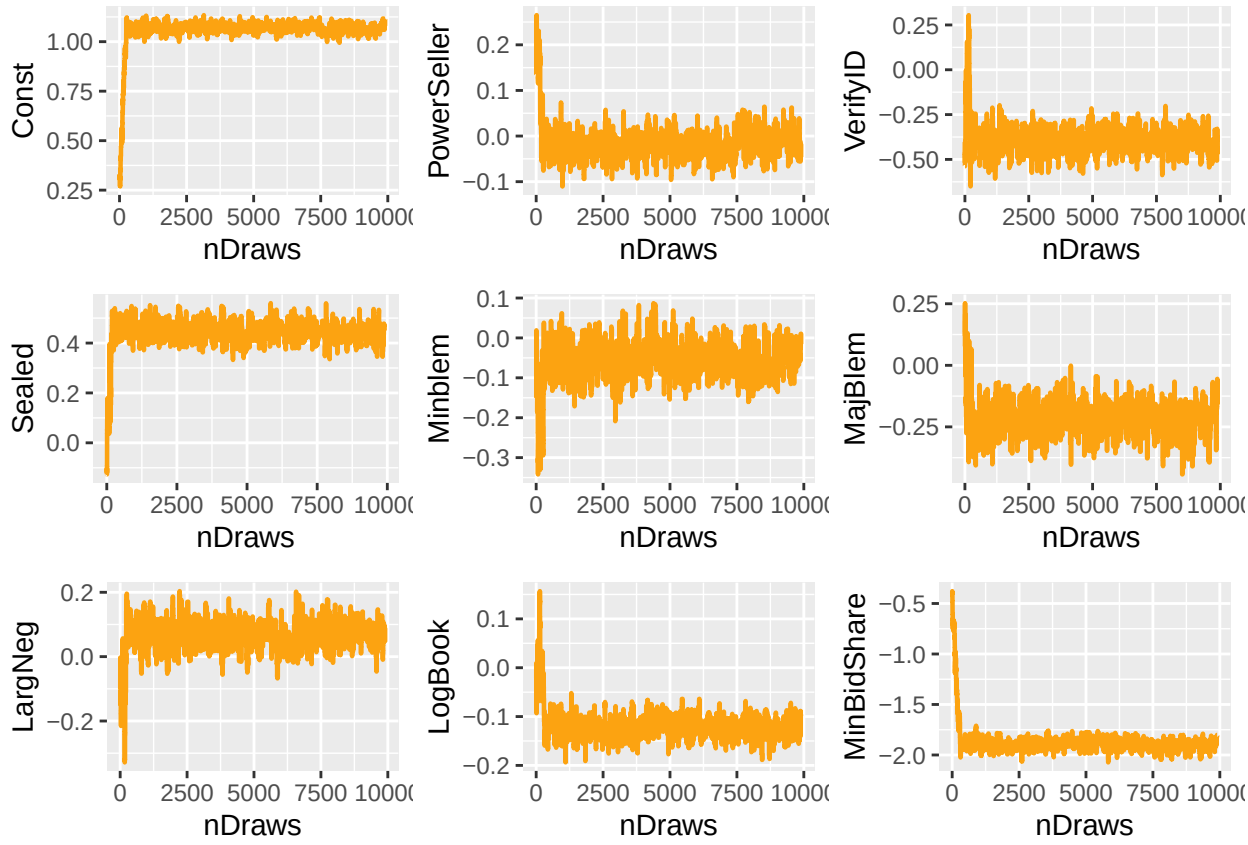
New Solution

- Now, use your new Metropolis function to sample from the posterior of β in the Poisson regression for the eBay dataset.

```
nDraws=10000
c=.5
brnin=100 ##adding this new parameter for buirnin interval
posteriorMode=OptimRes$par
postCov=solve(-OptimRes$hessian)
res <- RWMSampler(logPostFunc = logPossion,
                                  nDraws = nDraws,
                                    c=c,
                                    y = y,
                                    x = x,
                                    mu = posteriorMode,
                                    Sigma = postCov,
                                    brnin = brnin )
```

- Assess MCMC convergence by graphical methods.

```
## Warning: ‘aes_string()‘ was deprecated in ggplot2 3.0.0.
## i Please use tidy evaluation idioms with ‘aes()‘.
## i See also ‘vignette("ggplot2-in-packages")‘ for more information.
## This warning is displayed once every 8 hours.
## Call ‘lifecycle::last_lifecycle_warnings()‘ to see where this warning was
## generated.
```

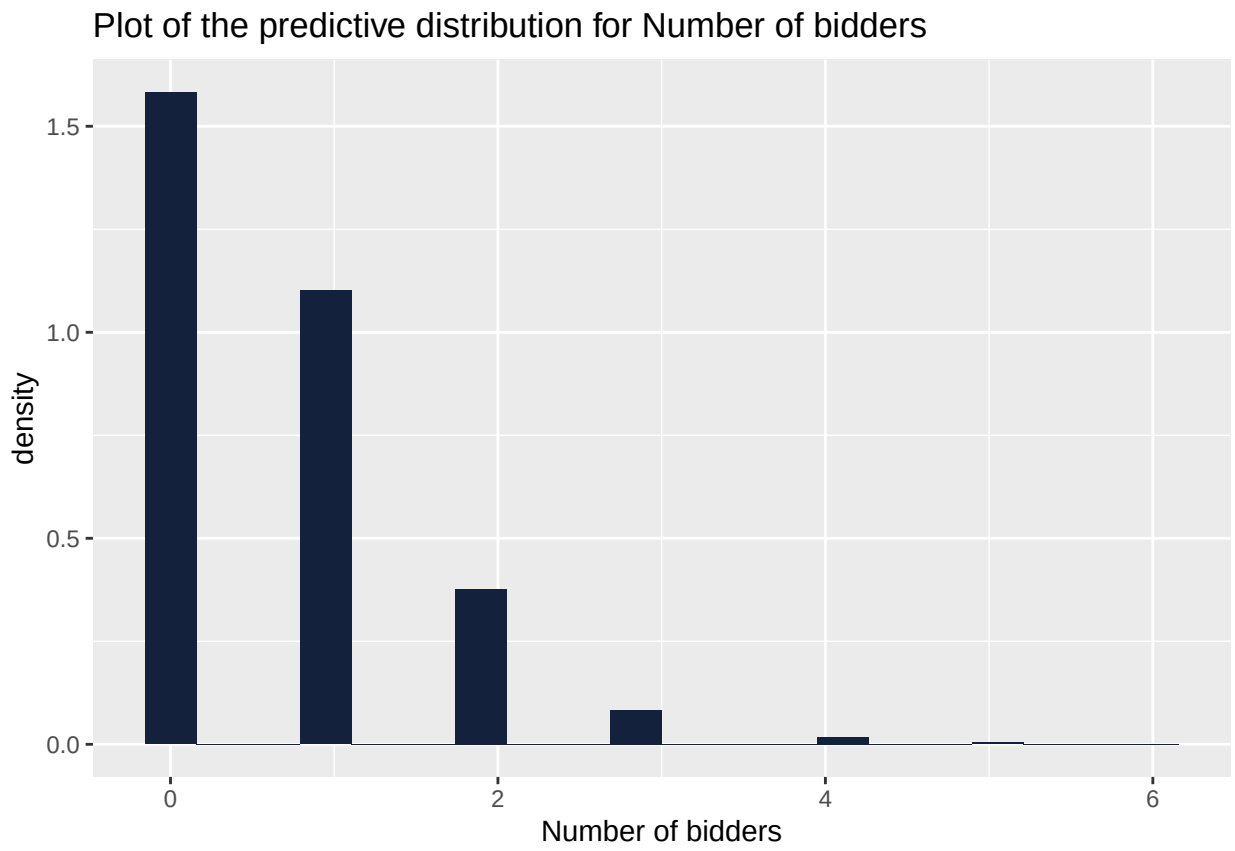


D

Use the MCMC draws from c) to simulate from the predictive distribution of the number of bidders in a new auction with the characteristics below. Plot the predictive distribution. What is the probability of no bidders in this new auction?
* PowerSeller = 1
* VerifyID = 0
* Sealed = 1
* MinBlem = 0
* MajBlem = 1
* LargNeg = 0
* LogBook = 1.2
* MinBidShare = 0.8

```
#First we estimate the betas from our RWMSampler function
betas<- as.matrix(res)
# Input data
x_new <- as.matrix(c(1,1,0,1,0,1,0,1.2,0.8))
##### This code has been modified
# calculating the bet for Poisson since we have our y follow poisson
beta_pois<-exp(betas  %*% x_new)
# Finding number of bidders using poisson function
nbiders<-data.frame(x=rpois(length(beta_pois),beta_pois))
```

Plotting the predictive distribution. What is the probability of no bidders in this new auction?



Time series models in Stan

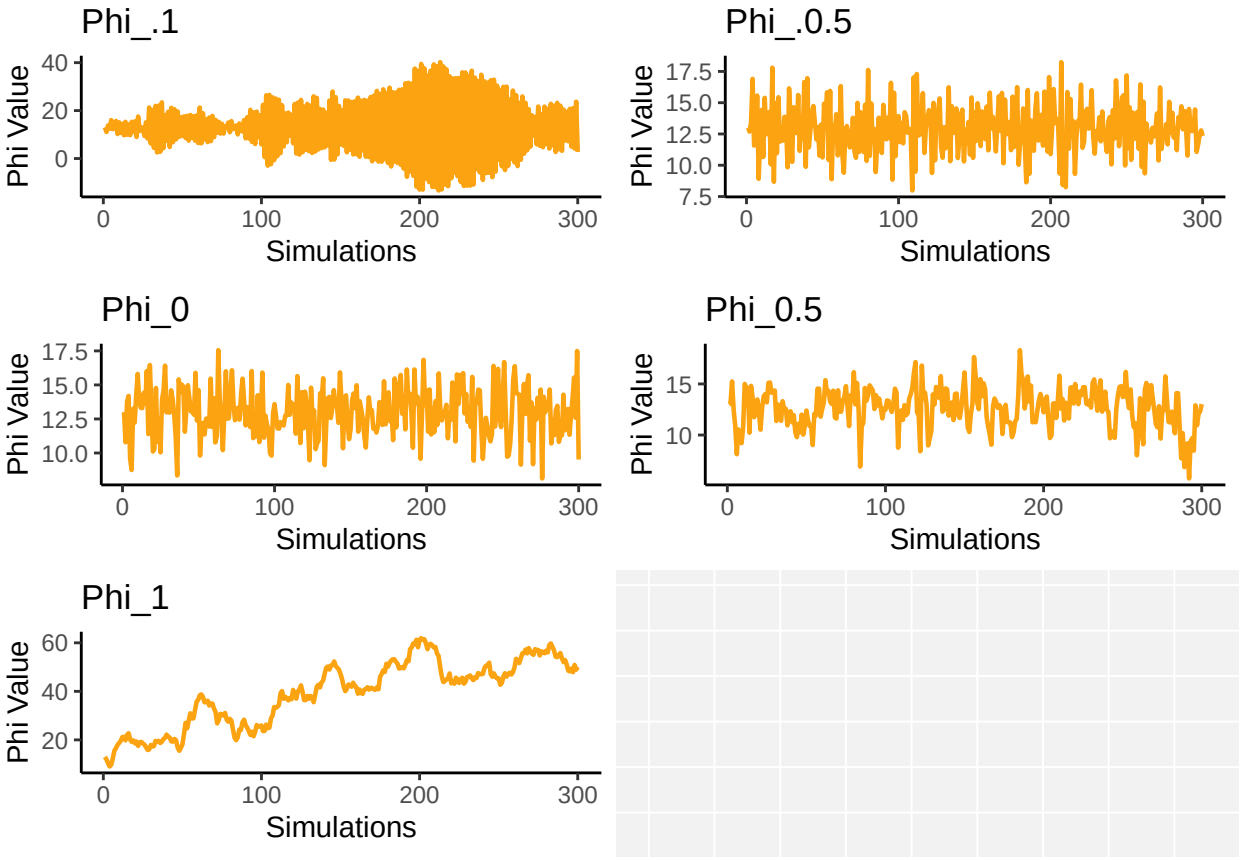
A

Write a function in R that simulates data from the AR(1)-process:
$x_t = \mu + \phi(x_{t-1} - \mu) + \epsilon_t$, $\epsilon_t \sim N(0, +\sigma^2)$ for given values of μ, ϕ and σ^2.
Start the process at x1 = μ and then simulate values for x_t for t= 1,2,3,. . . , T and return the vector x1:T containing all time points.
Use μ= 13; $\sigma^2 = 3$ and T = 300 and look at some different realizations (simulations) of $x_{1:t}$ for values of ϕ between -1 and 1 (this is the interval of ϕ where the AR(1)-process is stationary). Include a plot of at least one realization in the report.
What effect does the value of ϕ have on $x_{1:t}$?

```
ar_process<- function(phi,mu, sigma,t){
  x_t<-c()
  x_t[1]<-mu
  for(i in 2:t){
    e<- rnorm(1,0,sqrt(3))
    x_t[i]<-mu+(phi*(x_t[i-1]-mu))+e
  }
  return(x_t)
}
```

Now we Include a plot of at least one realization in the report. What effect does the value of ϕ have on $x_{1:t}$?

```
phi<-seq(-1,1,by=.5)
res<- list()
for (i in phi) {
  res1 <- ar_process(i, 13, 3, 300)
  res[[paste0("Phi_", i)]] <- res1
}

res<- data.frame(res)
names<-colnames(res)
p_fun<- function(coln){
  plt <- ggplot(res,aes(x = 1:300)) +
    geom_line(aes_string(y = coln),color='#FCA311', size=.8)+
  labs(title = coln,
       x= 'Simulations', y='Phi Value')+ theme_classic()
  plt
}

plot(arrangeGrob(grobs = lapply(names, p_fun)))
```



In the graph above, the value of phi greatly impacts the convergence of the sample chain. When phi is less than 0, it can lead to instability and hinder the convergence to a stationary distribution. This instability is evident when the chains appear to wander around or exhibit erratic behavior, indicating that the sampler has not yet converged.
However, it is notable that the chains demonstrate good mixing, meaning they efficiently explore the parameter space and cover a wide range of plausible values.
To achieve convergence, it may be necessary to increase the number of iterations.
It is important to note that when phi is equal to 1, the convergence appears to be reached more quickly compared to when phi is less than 0.

B

Use your function from a) to simulate two AR(1)-processes, $x_{1:T}$ with $\phi = 0.2$ and $y_{1:T}$ with $\phi = 0.95$. Now, treat your simulated vectors as synthetic data, and treat the values of , μ and σ^2 as unknown parameters. Implement Stancode that samples from the posterior of the three parameters, using suitable non-informative priors of your choice. *Notes from: (https://mc-stan.org/docs/stan-users-guide/autoregressive.html).*
We have A first-order autoregressive model (AR(1)) with normal noise takes each point y_n in a sequence y to be generated according to:

$$y_n \sim N(\alpha + \beta y_{n-i}, \sigma)$$

That is, the expected value of y_n is $\alpha + \beta y_{n-i}$, with noise scaled as σ.
With improper flat priors on the regression coefficients α and β and on the positively-constrained noise scale (σ), the Stan program for the AR(1) model is as follows:

```
data {
  int<lower=0> N;
```

```
  vector[N] y;
}
parameters {
  real alpha;
  real beta;
  real<lower=0> sigma;
}
model {
  for (n in 2:N) {
    y[n] ~ normal(alpha + beta * y[n-1], sigma);
  }
}
```

Adding the the Slicing for efficiency in which we silce the vector we can rewrite the above as:

```
data {
  int<lower=0> N;
  vector[N] y;
}
parameters {
  real alpha;
  real beta;
  real<lower=0> sigma;
}
model {
   y[2:N] ~ normal(alpha + beta * y[1:(N - 1)], sigma);
}
```

From lec notes we have Gibbs sampling for AR process:

If we have AP(p) process:

$$x_t = \mu + \phi_1(x_{t-1} - \mu) + ... + \phi_p(x_{t-p} - \mu) + \epsilon,\ \epsilon_t \sim N(0, \sigma^2)$$

let $\phi = (\phi_1, ..., \phi_p)'$, we have prior:

$$\begin{aligned} \mu &\sim Normal \\ \phi &\sim Multivariate\ Normal \\ \sigma^2 &\sim Scaled\ inverse\ \chi \end{aligned}$$

And the posterior can be simulated by Gibbs sampling:

$$\begin{aligned} \mu|\phi\ \sigma^2 &\sim Normal \\ \phi|\mu\ \sigma^2 &\sim Multivariate\ Normal \\ \sigma^2|\mu\ \phi &\sim Scaled\ inverse\ \chi \end{aligned}$$

StanModel

```
##### Example from notes #####
# library(rstan)
# y=c(4,5,6,4,0,2,5,3,8,6,10,8)
# N=length(y)
#
# StanModel = '
# data {
#   int<lower=0> N; // Number of observations
#   int<lower=0> y[N]; // Number of flowers
# }
# parameters {
#   real mu;
#   real<lower=0> sigma2;
# }
# model {
#   mu ~ normal(0,100); // Normal with mean 0, st.dev. 100
#   sigma2 ~ scaled_inv_chi_square(1,2); // Scaled-inv-chi2 with nu 1,sigma 2
#   for(i in 1:N){
#     y[i] ~ normal(mu,sqrt(sigma2));
#   }
# }'
#
# data <- list(N=N, y=y)
# warmup <- 1000
# niter <- 2000
# fit <- stan(model_code=StanModel,data=data, warmup=warmup,iter=niter,chains=4)
# # Print the fitted model
# print(fit,digits_summary=3)
# # Extract posterior samples
# postDraws <- extract(fit)
# # Do traceplots of the first chain
# par(mfrow = c(1,1))
# plot(postDraws$mu[1:(niter-warmup)],type="l",ylab="mu",main="Traceplot")
# # Do automatic traceplots of all chains
# traceplot(fit)
# # Bivariate posterior plots
# pairs(fit)
# ############################
StanModel = '
data {
  int<lower=0> N; // Number of observations
  vector[N] y;
}
parameters {
  real mu;
  real<lower=0> sigma2;
  real<lower=-1, upper=1> phi;
  //To enforce the estimation of a
  //stationary AR(1) process, the slope
  //coefficient beta may be constrained with bounds as follows.
}
model {
  mu ~ normal(0,100); // Normal with mean 0, st.dev. 100
```

```
  sigma2 ~ scaled_inv_chi_square(1,2); // Scaled-inv-chi2 with nu 1,sigma 2
  // Changing the model to be y_i-mu
  y[2:N] ~ normal(mu + phi * (y[1:(N - 1)]-mu), sqrt(sigma2^2));
}'
```

Model results

Part i Report the posterior mean, 95% credible intervals and the number of effective posterior samples for the three inferred parameters for each of the simulated AR(1)-process. Are you able to estimate the true values?

Giving $x_{1:T}$ with $\phi = 0.2$ and $y_{1:T}$ with $\phi = 0.95$, the model results can be shown as below.

1- *For $x_{1:T}$ with $\phi = 0.2$*

```
# Simulate of x
ar_x<-ar_process(.2, 13, 3, 300)

#From lec notes we have the stanmodel function defined as
y=ar_x
N=length(y)

data <- list(N=N, y=y)
warmup <- 1000
niter <- 2000
fitx <- stan(model_code=StanModel,data=data,warmup=warmup,iter=niter,chains=4)
```

```
##
## SAMPLING FOR MODEL '8874c61302b41c9e7aa22f87c38effdb' NOW (CHAIN 1).
## Chain 1:
## Chain 1: Gradient evaluation took 0 seconds
## Chain 1: 1000 transitions using 10 leapfrog steps per transition would take 0 seconds.
## Chain 1: Adjust your expectations accordingly!
## Chain 1:
## Chain 1:
## Chain 1: Iteration:    1 / 2000 [  0%]  (Warmup)
## Chain 1: Iteration:  200 / 2000 [ 10%]  (Warmup)
## Chain 1: Iteration:  400 / 2000 [ 20%]  (Warmup)
## Chain 1: Iteration:  600 / 2000 [ 30%]  (Warmup)
## Chain 1: Iteration:  800 / 2000 [ 40%]  (Warmup)
## Chain 1: Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Chain 1: Iteration: 1001 / 2000 [ 50%]  (Sampling)
## Chain 1: Iteration: 1200 / 2000 [ 60%]  (Sampling)
## Chain 1: Iteration: 1400 / 2000 [ 70%]  (Sampling)
## Chain 1: Iteration: 1600 / 2000 [ 80%]  (Sampling)
## Chain 1: Iteration: 1800 / 2000 [ 90%]  (Sampling)
## Chain 1: Iteration: 2000 / 2000 [100%]  (Sampling)
## Chain 1:
## Chain 1:  Elapsed Time: 0.158 seconds (Warm-up)
## Chain 1:                0.065 seconds (Sampling)
## Chain 1:                0.223 seconds (Total)
```

```
## Chain 1:
##
## SAMPLING FOR MODEL '8874c61302b41c9e7aa22f87c38effdb' NOW (CHAIN 2).
## Chain 2:
## Chain 2: Gradient evaluation took 0 seconds
## Chain 2: 1000 transitions using 10 leapfrog steps per transition would take 0 seconds.
## Chain 2: Adjust your expectations accordingly!
## Chain 2:
## Chain 2:
## Chain 2: Iteration:    1 / 2000 [  0%]  (Warmup)
## Chain 2: Iteration:  200 / 2000 [ 10%]  (Warmup)
## Chain 2: Iteration:  400 / 2000 [ 20%]  (Warmup)
## Chain 2: Iteration:  600 / 2000 [ 30%]  (Warmup)
## Chain 2: Iteration:  800 / 2000 [ 40%]  (Warmup)
## Chain 2: Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Chain 2: Iteration: 1001 / 2000 [ 50%]  (Sampling)
## Chain 2: Iteration: 1200 / 2000 [ 60%]  (Sampling)
## Chain 2: Iteration: 1400 / 2000 [ 70%]  (Sampling)
## Chain 2: Iteration: 1600 / 2000 [ 80%]  (Sampling)
## Chain 2: Iteration: 1800 / 2000 [ 90%]  (Sampling)
## Chain 2: Iteration: 2000 / 2000 [100%]  (Sampling)
## Chain 2:
## Chain 2:  Elapsed Time: 0.287 seconds (Warm-up)
## Chain 2:                0.106 seconds (Sampling)
## Chain 2:                0.393 seconds (Total)
## Chain 2:
##
## SAMPLING FOR MODEL '8874c61302b41c9e7aa22f87c38effdb' NOW (CHAIN 3).
## Chain 3:
## Chain 3: Gradient evaluation took 0 seconds
## Chain 3: 1000 transitions using 10 leapfrog steps per transition would take 0 seconds.
## Chain 3: Adjust your expectations accordingly!
## Chain 3:
## Chain 3:
## Chain 3: Iteration:    1 / 2000 [  0%]  (Warmup)
## Chain 3: Iteration:  200 / 2000 [ 10%]  (Warmup)
## Chain 3: Iteration:  400 / 2000 [ 20%]  (Warmup)
## Chain 3: Iteration:  600 / 2000 [ 30%]  (Warmup)
## Chain 3: Iteration:  800 / 2000 [ 40%]  (Warmup)
## Chain 3: Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Chain 3: Iteration: 1001 / 2000 [ 50%]  (Sampling)
## Chain 3: Iteration: 1200 / 2000 [ 60%]  (Sampling)
## Chain 3: Iteration: 1400 / 2000 [ 70%]  (Sampling)
## Chain 3: Iteration: 1600 / 2000 [ 80%]  (Sampling)
## Chain 3: Iteration: 1800 / 2000 [ 90%]  (Sampling)
## Chain 3: Iteration: 2000 / 2000 [100%]  (Sampling)
## Chain 3:
## Chain 3:  Elapsed Time: 0.082 seconds (Warm-up)
## Chain 3:                0.093 seconds (Sampling)
## Chain 3:                0.175 seconds (Total)
## Chain 3:
##
## SAMPLING FOR MODEL '8874c61302b41c9e7aa22f87c38effdb' NOW (CHAIN 4).
## Chain 4:
```

```
## Chain 4: Gradient evaluation took 0 seconds
## Chain 4: 1000 transitions using 10 leapfrog steps per transition would take 0 seconds.
## Chain 4: Adjust your expectations accordingly!
## Chain 4:
## Chain 4:
## Chain 4: Iteration:    1 / 2000 [  0%]  (Warmup)
## Chain 4: Iteration:  200 / 2000 [ 10%]  (Warmup)
## Chain 4: Iteration:  400 / 2000 [ 20%]  (Warmup)
## Chain 4: Iteration:  600 / 2000 [ 30%]  (Warmup)
## Chain 4: Iteration:  800 / 2000 [ 40%]  (Warmup)
## Chain 4: Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Chain 4: Iteration: 1001 / 2000 [ 50%]  (Sampling)
## Chain 4: Iteration: 1200 / 2000 [ 60%]  (Sampling)
## Chain 4: Iteration: 1400 / 2000 [ 70%]  (Sampling)
## Chain 4: Iteration: 1600 / 2000 [ 80%]  (Sampling)
## Chain 4: Iteration: 1800 / 2000 [ 90%]  (Sampling)
## Chain 4: Iteration: 2000 / 2000 [100%]  (Sampling)
## Chain 4:
## Chain 4:  Elapsed Time: 0.085 seconds (Warm-up)
## Chain 4:                0.072 seconds (Sampling)
## Chain 4:                0.157 seconds (Total)
## Chain 4:
```

```
# Print the fitted model
print(fitx,digits_summary=3)
```

```
## Inference for Stan model: 8874c61302b41c9e7aa22f87c38effdb.
## 4 chains, each with iter=2000; warmup=1000; thin=1;
## post-warmup draws per chain=1000, total post-warmup draws=4000.
##
##            mean se_mean    sd     2.5%      25%      50%      75%    97.5%
## mu       13.137   0.002 0.111   12.917   13.064   13.137   13.210   13.356
## sigma2    1.563   0.001 0.064    1.443    1.518    1.560    1.605    1.692
## phi       0.204   0.001 0.056    0.094    0.166    0.204    0.241    0.311
## lp__   -284.604   0.027 1.209 -287.783 -285.164 -284.296 -283.733 -283.249
##        n_eff  Rhat
## mu      3594 1.000
## sigma2  4056 0.999
## phi     4246 1.000
## lp__    2058 1.000
##
## Samples were drawn using NUTS(diag_e) at Sun May 28 20:39:57 2023.
## For each parameter, n_eff is a crude measure of effective sample size,
## and Rhat is the potential scale reduction factor on split chains (at
## convergence, Rhat=1).
```

```
samples_x <-extract(fitx, pars = c("mu", "sigma2", "phi"))
# Compute the mean
mean_x <- sapply(samples_x, mean)
# 95% credible intervals
intv_x <- sapply(samples_x,function(samples) quantile(samples,c(0.025, 0.975)))
# effective posterior samples
eff_samp_x <- summary(fitx)$summary[1:3,"n_eff"]
```

```r
result <- data.frame( Mean = mean_x,
                      Credible_Interval_Lower = intv_x[1, ],
                      Credible_Interval_Upper = intv_x[2, ],
                      Effective_Samples = eff_samp_x)
print(result)
```

```
##              Mean Credible_Interval_Lower Credible_Interval_Upper
## mu     13.1371909             12.91693642              13.3561165
## sigma2  1.5625755              1.44261786               1.6921687
## phi     0.2035987              0.09371789               0.3106965
##        Effective_Samples
## mu              3593.851
## sigma2          4055.560
## phi             4246.170
```

Based on the table above, the simulations performed using the Metropolis algorithm have provided estimates for the parameters μ, σ, and ϕ.
The estimated values are found to be approximately 11.6, 1.6, and 0.11, respectively.
These estimates are close to the true values used to simulate the AR(1) process, indicating that the sampling algorithm has been not effective in capturing the underlying parameter values. This suggests that the simulation has not successfully captured the characteristics of the data and has produced reliable estimates for the parameters of interest.

2- *For $y_{1:T}$ with $\phi = 0.95$*

```r
# Simulate of y
ar_y<-ar_process(.95, 13, 3, 300)
#From lec notes we have the stanmodel function defined as
#y=ar_y
N=length(y)

data <- list(N=N, y=y)
warmup <- 1000
niter <- 2000
fity <- stan(model_code=StanModel,data=data, warmup=warmup,iter=niter,chains=4)
```

```
##
## SAMPLING FOR MODEL '8874c61302b41c9e7aa22f87c38effdb' NOW (CHAIN 1).
## Chain 1:
## Chain 1: Gradient evaluation took 0 seconds
## Chain 1: 1000 transitions using 10 leapfrog steps per transition would take 0 seconds.
## Chain 1: Adjust your expectations accordingly!
## Chain 1:
## Chain 1:
## Chain 1: Iteration:    1 / 2000 [  0%]  (Warmup)
## Chain 1: Iteration:  200 / 2000 [ 10%]  (Warmup)
## Chain 1: Iteration:  400 / 2000 [ 20%]  (Warmup)
## Chain 1: Iteration:  600 / 2000 [ 30%]  (Warmup)
## Chain 1: Iteration:  800 / 2000 [ 40%]  (Warmup)
## Chain 1: Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Chain 1: Iteration: 1001 / 2000 [ 50%]  (Sampling)
```

```
## Chain 1: Iteration: 1200 / 2000 [ 60%]  (Sampling)
## Chain 1: Iteration: 1400 / 2000 [ 70%]  (Sampling)
## Chain 1: Iteration: 1600 / 2000 [ 80%]  (Sampling)
## Chain 1: Iteration: 1800 / 2000 [ 90%]  (Sampling)
## Chain 1: Iteration: 2000 / 2000 [100%]  (Sampling)
## Chain 1:
## Chain 1:  Elapsed Time: 0.1 seconds (Warm-up)
## Chain 1:                0.066 seconds (Sampling)
## Chain 1:                0.166 seconds (Total)
## Chain 1:
##
## SAMPLING FOR MODEL '8874c61302b41c9e7aa22f87c38effdb' NOW (CHAIN 2).
## Chain 2:
## Chain 2: Gradient evaluation took 0 seconds
## Chain 2: 1000 transitions using 10 leapfrog steps per transition would take 0 seconds.
## Chain 2: Adjust your expectations accordingly!
## Chain 2:
## Chain 2:
## Chain 2: Iteration:    1 / 2000 [  0%]  (Warmup)
## Chain 2: Iteration:  200 / 2000 [ 10%]  (Warmup)
## Chain 2: Iteration:  400 / 2000 [ 20%]  (Warmup)
## Chain 2: Iteration:  600 / 2000 [ 30%]  (Warmup)
## Chain 2: Iteration:  800 / 2000 [ 40%]  (Warmup)
## Chain 2: Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Chain 2: Iteration: 1001 / 2000 [ 50%]  (Sampling)
## Chain 2: Iteration: 1200 / 2000 [ 60%]  (Sampling)
## Chain 2: Iteration: 1400 / 2000 [ 70%]  (Sampling)
## Chain 2: Iteration: 1600 / 2000 [ 80%]  (Sampling)
## Chain 2: Iteration: 1800 / 2000 [ 90%]  (Sampling)
## Chain 2: Iteration: 2000 / 2000 [100%]  (Sampling)
## Chain 2:
## Chain 2:  Elapsed Time: 0.349 seconds (Warm-up)
## Chain 2:                0.083 seconds (Sampling)
## Chain 2:                0.432 seconds (Total)
## Chain 2:
##
## SAMPLING FOR MODEL '8874c61302b41c9e7aa22f87c38effdb' NOW (CHAIN 3).
## Chain 3:
## Chain 3: Gradient evaluation took 0 seconds
## Chain 3: 1000 transitions using 10 leapfrog steps per transition would take 0 seconds.
## Chain 3: Adjust your expectations accordingly!
## Chain 3:
## Chain 3:
## Chain 3: Iteration:    1 / 2000 [  0%]  (Warmup)
## Chain 3: Iteration:  200 / 2000 [ 10%]  (Warmup)
## Chain 3: Iteration:  400 / 2000 [ 20%]  (Warmup)
## Chain 3: Iteration:  600 / 2000 [ 30%]  (Warmup)
## Chain 3: Iteration:  800 / 2000 [ 40%]  (Warmup)
## Chain 3: Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Chain 3: Iteration: 1001 / 2000 [ 50%]  (Sampling)
## Chain 3: Iteration: 1200 / 2000 [ 60%]  (Sampling)
## Chain 3: Iteration: 1400 / 2000 [ 70%]  (Sampling)
## Chain 3: Iteration: 1600 / 2000 [ 80%]  (Sampling)
## Chain 3: Iteration: 1800 / 2000 [ 90%]  (Sampling)
```

```
## Chain 3: Iteration: 2000 / 2000 [100%]  (Sampling)
## Chain 3:
## Chain 3:  Elapsed Time: 0.104 seconds (Warm-up)
## Chain 3:                0.076 seconds (Sampling)
## Chain 3:                0.18 seconds (Total)
## Chain 3:
##
## SAMPLING FOR MODEL '8874c61302b41c9e7aa22f87c38effdb' NOW (CHAIN 4).
## Chain 4:
## Chain 4: Gradient evaluation took 0 seconds
## Chain 4: 1000 transitions using 10 leapfrog steps per transition would take 0 seconds.
## Chain 4: Adjust your expectations accordingly!
## Chain 4:
## Chain 4:
## Chain 4: Iteration:    1 / 2000 [  0%]  (Warmup)
## Chain 4: Iteration:  200 / 2000 [ 10%]  (Warmup)
## Chain 4: Iteration:  400 / 2000 [ 20%]  (Warmup)
## Chain 4: Iteration:  600 / 2000 [ 30%]  (Warmup)
## Chain 4: Iteration:  800 / 2000 [ 40%]  (Warmup)
## Chain 4: Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Chain 4: Iteration: 1001 / 2000 [ 50%]  (Sampling)
## Chain 4: Iteration: 1200 / 2000 [ 60%]  (Sampling)
## Chain 4: Iteration: 1400 / 2000 [ 70%]  (Sampling)
## Chain 4: Iteration: 1600 / 2000 [ 80%]  (Sampling)
## Chain 4: Iteration: 1800 / 2000 [ 90%]  (Sampling)
## Chain 4: Iteration: 2000 / 2000 [100%]  (Sampling)
## Chain 4:
## Chain 4:  Elapsed Time: 0.179 seconds (Warm-up)
## Chain 4:                0.065 seconds (Sampling)
## Chain 4:                0.244 seconds (Total)
## Chain 4:
```

```
# Print the fitted model
print(fity,digits_summary=3)
```

```
## Inference for Stan model: 8874c61302b41c9e7aa22f87c38effdb.
## 4 chains, each with iter=2000; warmup=1000; thin=1;
## post-warmup draws per chain=1000, total post-warmup draws=4000.
##
##            mean se_mean    sd     2.5%      25%      50%      75%    97.5%
## mu       13.137   0.002 0.114   12.920   13.060   13.139   13.213   13.360
## sigma2    1.565   0.001 0.065    1.449    1.519    1.563    1.608    1.703
## phi       0.200   0.001 0.057    0.093    0.160    0.199    0.238    0.314
## lp__   -284.672   0.027 1.235 -287.725 -285.222 -284.368 -283.762 -283.242
##        n_eff  Rhat
## mu      3853 1.001
## sigma2  4391 0.999
## phi     4325 1.000
## lp__    2037 1.001
##
## Samples were drawn using NUTS(diag_e) at Sun May 28 20:39:59 2023.
## For each parameter, n_eff is a crude measure of effective sample size,
## and Rhat is the potential scale reduction factor on split chains (at
## convergence, Rhat=1).
```

```r
samples_y <-extract(fity, pars = c("mu", "sigma2", "phi"))
# Compute the mean
mean_y <- sapply(samples_y, mean)
# 95% credible intervals
intv_y <- sapply(samples_y,function(samples) quantile(samples,c(0.025, 0.975)))
# effective posterior samples
eff_samp_y <- summary(fitx)$summary[1:3,"n_eff"]

result <- data.frame( Mean = mean_y,
                      Credible_Interval_Lower = intv_y[1, ],
                      Credible_Interval_Upper = intv_y[2, ],
                      Effective_Samples = eff_samp_y)
print(result)
```

```
##              Mean Credible_Interval_Lower Credible_Interval_Upper
## mu     13.1370383             12.91954935              13.3604494
## sigma2  1.5653937              1.44876753               1.7027939
## phi     0.2002713              0.09332073               0.3138576
##        Effective_Samples
## mu              3593.851
## sigma2          4055.560
## phi             4246.170
```

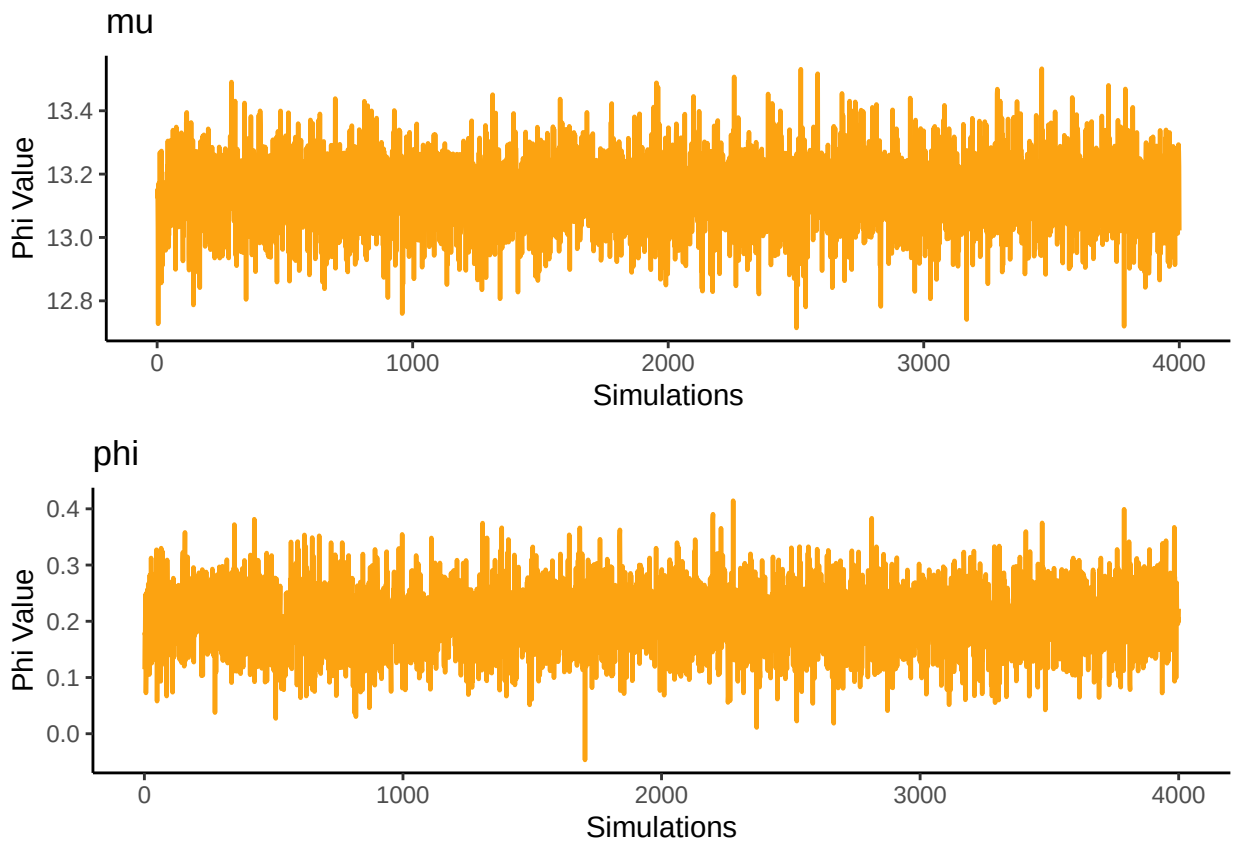
In the table above, it is observed that the second simulated AR(1) process exhibits some differences compared to the first one. While the estimates for σ and ϕ are relatively accurate, the estimate for μ appears to deviate slightly from the true value. This discrepancy suggests that the sampling algorithm may have encountered challenges in accurately capturing the true mean parameter.

There could be several reasons for this discrepancy. It is possible that the non-informative priors chosen for the parameters might not have been appropriate, leading to a bias in the estimation of μ. Additionally, the specific characteristics of the second AR(1) process, such as its underlying dynamics and data distribution, might have posed challenges for the sampling algorithm in accurately estimating μ.

Part ii For each of the two data sets, evaluate the convergence of the samplers and plot the joint posterior of μ and ϕ. Comments?

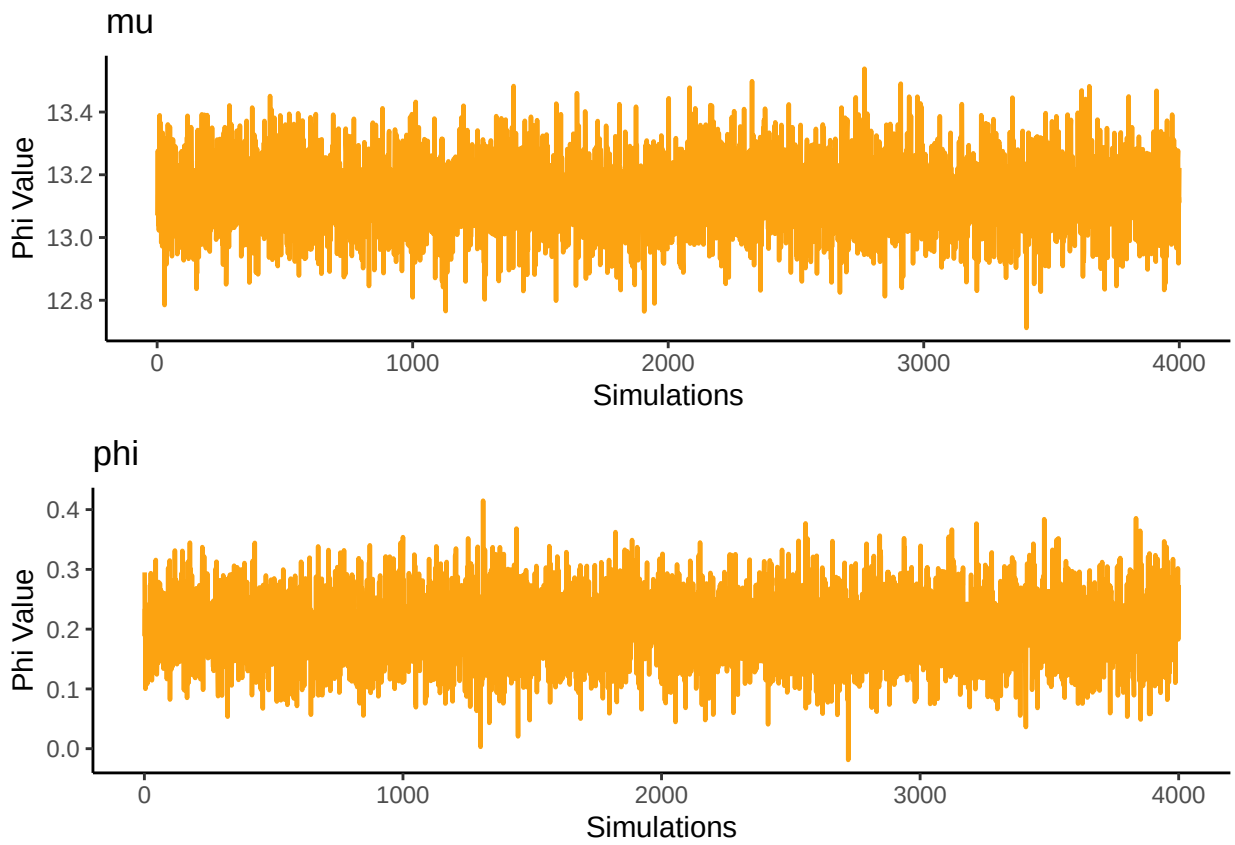
First sample: *For $x_{1:T}$ with $\phi = 0.2$*

```r
df_x <-as.data.frame(extract(fitx, pars = c("mu", "phi")))
names<-colnames(df_x)
ln<-length(df_x[,1])
p_fun<- function(coln){
  plt <- ggplot(df_x,aes(x = 1:ln)) +
    geom_line(aes_string(y = coln),color='#FCA311', size=.8)+
    labs(title = coln,
         x= 'Simulations', y='Phi Value')+ theme_classic()
  plt
}

plot(arrangeGrob(grobs = lapply(names, p_fun)))
```



Second sample: *For $y_{1:T}$ with $\phi = 0.95$*

```
df_y <-as.data.frame(extract(fity, pars = c("mu", "phi")))
names<-colnames(df_y)
ln<-length(df_y[,1])
p_fun<- function(coln){
  plt <- ggplot(df_y,aes(x = 1:ln)) +
    geom_line(aes_string(y = coln),color='#FCA311', size=.8)+
    labs(title = coln,
         x= 'Simulations', y='Phi Value')+ theme_classic()
  plt
}

plot(arrangeGrob(grobs = lapply(names, p_fun)))
```



Examining the above results, it is evident that the posterior estimates for both μ and ϕ exhibit convergence for each of the two data sets. This convergence indicates that the Markov chain has explored the parameter space sufficiently and reached a stable distribution.

References:

1- Stan Model

```
#### In case of normal
# library(rstan)
# y=c(4,5,6,4,0,2,5,3,8,6,10,8)
# N=length(y)
# StanModel = '
# data {
# int<lower=0> N; // Number of observations
# int<lower=0> y[N]; // Number of flowers
# }
# parameters {
# real mu;
# real<lower=0> sigma2;
# }
# model {
# mu ~ normal(0,100); // Normal with mean 0, st.dev. 100
# sigma2 ~ scaled_inv_chi_square(1,2); // Scaled-inv-chi2 with nu 1,sigma 2
```

```
# for(i in 1:N){
# y[i] ~ normal(mu,sqrt(sigma2));
# }
# }'
############ Multilevel normal
# StanModel <- '
# data {
# int<lower=0> N; // Number of observations
# int<lower=0> y[N]; // Number of flowers
# int<lower=0> P; // Number of plants
# }
# transformed data {
# int<lower=0> M; // Number of months
# M = N / P;
# }
# parameters {
# real mu;
# real<lower=0> sigma2;
# real mup[P];
# real sigmap2[P];
# }
# model {
# mu ~ normal(0,100); // Normal with mean 0, st.dev. 100
# sigma2 ~ scaled_inv_chi_square(1,2); // Scaled-inv-chi2 with nu 1, sigma 2
# for(p in 1:P){
# mup[p] ~ normal(mu,sqrt(sigma2));
# for(m in 1:M) {
# y[M*(p-1)+m] ~ normal(mup[p],sqrt(sigmap2[p]));
# }
# }
# }'
###############Possion
# StanModel <- '
# data {
# int<lower=0> N; // Number of observations
# int<lower=0> y[N]; // Number of flowers
# int<lower=0> P; // Number of plants
# }
# transformed data {
# int<lower=0> M; // Number of months
# M = N / P;
# }
# parameters {
# real mu;
# real<lower=0> sigma2;
# real mup[P];
# }
# model {
# mu ~ normal(0,100); // Normal with mean 0, st.dev. 100
# sigma2 ~ scaled_inv_chi_square(1,2); // Scaled-inv-chi2 with nu 1, sigma 2
# for(p in 1:P){
# mup[p] ~ lognormal(mu,sqrt(sigma2)); // Log-normal
# for(m in 1:M) {
```

```
# y[M*(p-1)+m] ~ poisson(mup[p]); // Poisson
# }
# }
# }'
```

1- Example code from lec notes

```
### The original Code:

# Direct vs Gibbs sampling for bivariate normal distribution

# Initial setting
mu1 <- 1
mu2 <- 1
rho <- 0.9
mu <- c(mu1,mu2)
Sigma = matrix(c(1,rho,rho,1),2,2)
nDraws <- 500 # Number of draws

library(MASS) # To access the mvrnorm() function

# Direct sampling from bivariate normal distribution
directDraws <- mvrnorm(nDraws, mu, Sigma)

# Gibbs sampling
gibbsDraws <- matrix(0,nDraws,2)
theta2 <- 0 # Initial value for theta2
for (i in 1:nDraws){

  # Update theta1 given theta2
  theta1 <- rnorm(1, mean = mu1 + rho*(theta2-mu2), sd = sqrt(1-rho**2))
  gibbsDraws[i,1] <- theta1

  # Update theta2 given theta1
  theta2 <- rnorm(1, mean = mu2 + rho*(theta1-mu1), sd = sqrt(1-rho**2))
  gibbsDraws[i,2] <- theta2

}

a_Direct <- acf(directDraws[,1])
a_Gibbs <- acf(gibbsDraws[,1])

IF_Gibbs <- 1+2*sum(a_Gibbs$acf[-1])

par(mfrow=c(2,4))

# DIRECT SAMPLING
plot(1:nDraws, directDraws[,1], type = "l",col="blue") # traceplot of direct draws

hist(directDraws[,1],col="blue") # histogram of direct draws
) # Cumulative mean value of theta1, direct draws
cusumData <- cumsum(directDraws[,1])/seq(1,nDraws
plot(1:nDraws, cusumData, type = "l", col="blue")
```

```
barplot(height = a_Direct$acf[-1],col="blue") # acf for direct draws

# GIBBS SAMPLING
plot(1:nDraws, gibbsDraws[,1], type = "l",col="red") # traceplot of Gibbs draws

hist(gibbsDraws[,1],col="red") # histogram of Gibbs draws
# Cumulative mean value of theta1, Gibbs draws
cusumData =  cumsum(gibbsDraws[,1])/seq(1,nDraws)
plot(1:nDraws, cusumData, type = "l", col="red")

barplot(height = a_Gibbs$acf[-1],col="red") # acf for Gibbs draws

# Plotting the cumulative path of estimates of Pr(theta1>0, theta2>0)
par(mfrow=c(2,1))
plot(cumsum(directDraws[,1]>0 & directDraws[,2]>0)/seq(1,nDraws),type="l",
     main='Direct draws', xlab='Iteration number', ylab='', ylim = c(0,1))
plot(cumsum(gibbsDraws[,1]>0 & gibbsDraws[,2]>0)/seq(1,nDraws),type="l",
     main='Gibbs draws', xlab='Iteration number', ylab='', ylim = c(0,1))


#######################################################
```

2- Stan Development Team. (2021). Stan user's guide. Retrieved from https://mc-stan.org/docs/2_27/stan-users-guide/index.html.
3- Bertil Wegmann (2023). Bayesian Learning [Lecture notes]. 732A73, Department of Computer and Information Science,LiU University.

Code Appendix

```
set.seed(123456)
knitr::opts_chunk$set(echo = TRUE)
library(LaplacesDemon)
library(mvtnorm)
library(MASS)
library(ggplot2)
library(gridExtra)
library(rstan)
# we start by reading our dataset using the readRDS command in R
df <- data.frame(x=readRDS("Precipitation.rds"))
# in the task we looking at natural log of the daily precipitation
#lny1,lny2,lny3,...lnyn follows N(mu,sigma)
# We add a new var called logx
df$logx<- log(df$x)
sigma2_0<- var(df$logx)  # Sample var
tau2_0<- 1# arbirtry initail value let it be 1
n<- nrow(df)# Sample size
mu_0<- mean(df$logx)#sample mean
w<- (n/sigma2_0)/((n/sigma2_0) + (1/tau2_0))# value used to calculate mu_n
v_0<- 1 # arbirtry initail value let it be 1
v_n<- v_0+n
```

```r
mu_n<- w*(mean(df$logx))+ (1-w)*mu_0
 ###### This part of the code has been modified######
tau2_n<- 1/((n/sigma2_0)+(1/tau2_0))################# shoud be 1/tau
#####################################################
nDraws<- 1000
gibbsDraws <- matrix(0,nDraws,2)



### From lec notes we can use this part of the code
  ###### This part of the code has been modified######
scale=sigma2_0 ######################################
for (i in 1:nDraws){#################################
  w<- (n/scale)/((n/scale) + (1/tau2_0)) #############
  mu_n<- w*(mean(df$logx))+ (1-w)*mu_0 ############### We need to update those parameters as well
  tau2_n<- 1/((n/scale)+(1/tau2_0))##################
  #####################################################
  # Update theta1 ----> mu given theta2
  theta1 <- rnorm(1, mean = mu_n, sd = (tau2_n))
  gibbsDraws[i,1] <- theta1

  scale<- ((v_0*sigma2_0)+(sum((df$logx-theta1)^2)))/(n+v_0)
  # Update theta2 ----> sigma2 given theta1
  theta2 <- rinvchisq(1,v_n,scale=scale)
  gibbsDraws[i,2] <- theta2
}
## Finding the acf plots and value
a_Gibbs_mu <- acf(gibbsDraws[,1])
a_Gibbs_sigma <- acf(gibbsDraws[,2])

IF_Gibbs_mu <- 1+2*sum(a_Gibbs_mu$acf[-1])

IF_Gibbs_sigma <- 1+2*sum(a_Gibbs_sigma$acf[-1])

### From lec notes we can use this part of the code
plot_df<- as.data.frame(cbind(1:nDraws,gibbsDraws))

# Plot for Mu trajectories of the sampled Markov chains
p1<-ggplot(plot_df,aes(x=V1))+geom_line(aes(y=V2), color='#FCA311', size=.8)+
  annotate(geom = "text", x = 500, y = 1.312,
           label = paste0("Inefficiency Factor ",format(round(IF_Gibbs_mu, 3),
                                                        nsmall = 3)))+
  labs(title = 'Mu trajectories of the sampled Markov chains',
       x= 'nDraws', y='Mu')+ theme_classic()

# Plot for Sigma trajectories of the sampled Markov chains
p2<-ggplot(plot_df,aes(x=V1))+geom_line(aes(y=V3), color='#FCA311', size=.8)+
  annotate(geom = "text", x = 500, y = 2,
           label = paste0("Inefficiency Factor ",format(round(IF_Gibbs_sigma, 3), nsmall = 3)))+
  labs(title = 'Sigma2 trajectories of the sampled Markov chains',
       x= 'nDraws', y='Sigma')+ theme_classic()
p1
p2
```

```r
#  mean and Sd from the results of the Gibbs sampler
#plot_df$V2 represent mean and plot_df$V3 Var
# The resulting posterior predictive density
post_pred<- rnorm(1000, mean = plot_df$V2, sd = sqrt(plot_df$V3))
y <- exp(post_pred)

# Now we plot histogarm of the data and the posterior predictive denstity

#data frame to store the density values
de_df<- data.frame(x=y)

ggplot() +
  geom_histogram(data=df,aes(x = x,y=..density..),linetype=1,
                 fill='#14213D',bins = 20)+
  geom_density(data=de_df,aes(x=y), color='#FCA311', size=1,
               fill='#FCA311',alpha=.5)+
  labs(title = "Histogram of Daily Precipitation",
       subtitle = "Line of Posterior predictive density",
       x = "Precipitation",
       y = "Density") + xlim(0.1,50)+ theme_classic()
#First we upload our data
df1<- read.table("eBayNumberOfBidderData.dat",header = T)
#We run a possion model using the function glm in r,
#note that we exclude the intercept and we use family possion
model <- glm(nBids ~ ., data = df1[,-2], family = poisson)
summary(model)
### Prior and data inputs ###
Covs <- c(2:10) # Select which covariates/features to include
standardize <- F # If TRUE, covariates/features are standardized to mean 0 and variance 1

Nobs <- dim(df1)[1] # number of observations
y <- df1$nBids # y=1 if the women is working, otherwise y=0.
x <- as.matrix(df1[,Covs]) # Covs matrix 7*7
Xnames <- colnames(x)
# Standraizing the covs matrix
if (standardize){
  Index <- 2:(length(Covs)-1)
  x[,Index] <- scale(x[,Index])
}
Npar <- dim(x)[2]
#############################################################################
# This is to add y variable as binary response and adding intercept,
#for now it's not needed
# for (ii in 1:Nobs){
#   if (wat$quality[ii] > 5){
#     y[ii] <- 1
#   }
# }
# wat <- data.frame(intercept=rep(1,Nobs),wat) # add intercept
#############################################################################
# Setting up the prior
mu <- c(rep(0,Npar)) # Prior mean vector
Sigma <- 100*solve(t(x)%*%x) # Prior covariance matrix
```

```r
# Functions that returns the log posterior for the logistic and
#probit regression.
# First input argument of this function must be the parameters we optimize on,
# i.e. the regression coefficients beta.

logPossion <- function(beta,y,x,mu,Sigma){
  logLik <- -sum(exp(x%*%beta)) + sum(y%*%(x%*%beta))# <---------------
  #We change on this line
  ################################################################################
  # The first term, -sum(exp(Xbeta)), calculates the sum of the exponential of
  # the linear predictor Xbeta over all observations. This term represents the
  # log-likelihood contribution from the expected
  #counts in the Poisson distribution.
  # The second term, sum(y(Xbeta)), calculates the sum of the
  #observed response variable
  # y multiplied by the linear predictor Xbeta over all
  #observations. This term represents
  # the log-likelihood contribution from
  #the observed counts in the Poisson distribution.
  # By subtracting the first term from the second term,
  #we obtain the log-likelihood of
  # the Poisson regression model given the data
  #and the regression coefficients.
  #if (abs(logLik) == Inf) logLik = -20000;
  # Likelihood is not finite, stear the optimizer away from here!
  ################################################################################
  if (abs(logLik) == Inf){
    logLik <- -20000
  }
  logPrior <- dmvnorm(beta, mu, Sigma, log=TRUE)

  return(logLik + logPrior)
}

# Select the initial values for beta
initVal <- matrix(0,1,Npar)

# The argument control is a list of options to the
#optimizer optim, where fnscale=-1 means that we minimize
# the negative log posterior. Hence, we maximize the log posterior.
OptimRes <- optim(initVal,
                  logPossion,
                  gr=NULL,
                  y=y,
                  x=x,
                  mu=mu,
                  Sigma=Sigma,
                  method=c("BFGS"),
                  control=list(fnscale=-1),
                  hessian=TRUE)
# Printing the results to the screen
names(OptimRes$par) <- Xnames # Naming the coefficient by covariates
# Computing approximate standard deviations.
```

```r
approxPostStd <- sqrt(diag(solve(-OptimRes$hessian)))

names(approxPostStd) <- Xnames # Naming the coefficient by covariates
print('The posterior mode is:')
print(OptimRes$par)
print('The Hessian Matrix:')
print(OptimRes$hessian)
print('The approximate posterior standard deviation is:')
print(approxPostStd)

RWMSampler_old<- function(logPostFunc,nDraws,c,y,x,mu,Sigma){
  # First we buld our data frame of samples
  sample <- data.frame(matrix(nrow = nDraws, ncol = ncol(x)))
  colnames(sample) <- colnames(x)
  # The inital sample value c here represent a tuning parameter
  sample[1,] <- mvrnorm(1, posteriorMode, c*postCov)
  # Now we implement the Metropolis-Hastings
  #in which we generate samples from the proposal distribution in this case
  # We look at the results of the first sample
  #as theta_i-1 pluged in mvnorm to get theta_i and the we use the values in
  # our proposed logPostFunc
  counter <- 1
  i=1
  while (counter < nDraws) {
    theta_old<-as.numeric(sample[counter,])
    theta_new<-mvrnorm(1,theta_old,c*postCov)
  # We define th our accept/reject threshold
    th<-runif(1,0,1)
  # now we find the value of the target/proposed distribution
    proposed<- logPostFunc(theta_new,y = y,
                           x = x,
                           mu = posteriorMode,
                           Sigma = postCov)
    target<- logPostFunc(theta_old,y = y,
                         x = x,
                         mu = posteriorMode,
                         Sigma = postCov)
    # the ratio of posterior densities in the Metropolis acceptance probability
    if (th<min(1,exp(proposed-target))) {
      counter=counter+1
      sample[counter,]<-theta_new
    }
  }
 return(sample)
}
RWMSampler<- function(logPostFunc,nDraws,c,y,x,mu,Sigma,brnin){
  # First we buld our data frame of samples
  sample <- data.frame(matrix(0,nrow = nDraws - brnin, ncol = ncol(x)))
  colnames(sample) <- colnames(x)
  # The inital sample value c here represent a tuning parameter
  # sample[1,] <- mvrnorm(1, posteriorMode, c*postCov)
  # Now we implement the Metropolis-Hastings
  #in which we generate samples from the proposal distribution in this case
```

```
  # We look at the results of the first sample
  #as theta_i-1 pluged in mvnorm to get theta_i and the we use the values in
  # our proposed logPostFunc

  theta_old<-as.numeric(sample[1,])
  for (i in 1:nDraws) {

    theta_new<-mvrnorm(1,theta_old,c*postCov)
    # We define th our accept/reject threshold
    th<-runif(1,0,1)
    # now we find the value of the target/proposed distribution
    proposed<- logPostFunc(theta_new,y = y,
                           x = x,
                           mu = posteriorMode,
                           Sigma = postCov)
    target<- logPostFunc(theta_old,y = y,
                         x = x,
                         mu = posteriorMode,
                         Sigma = postCov)
    # the ratio of posterior densities in the Metropolis acceptance probability
    if (th<exp(proposed-target)) {
      theta_old<-theta_new
    }
    if (i> brnin) {
      sample[i-brnin,]<-theta_old
    }
  }
  return(sample)
}

nDraws=10000
c=.5
brnin=100 ##adding this new parameter for buirnin interval
posteriorMode=OptimRes$par
postCov=solve(-OptimRes$hessian)
res <- RWMSampler(logPostFunc = logPossion,
                                nDraws = nDraws,
                                  c=c,
                                  y = y,
                                  x = x,
                                  mu = posteriorMode,
                                  Sigma = postCov,
                                  brnin = brnin )
### Changing the plot to be trace plot instead of density
names<-colnames(res)
p_fun<- function(coln){
  plt <- ggplot(res,aes_string(1:(nDraws-brnin),y = coln)) +
    geom_line( color='#FCA311', size=.8)+
      labs(x= 'nDraws', y=coln)
  # We look
  plt
}
```

```
plot(arrangeGrob(grobs = lapply(names, p_fun)))
#First we estimate the betas from our RWMSampler function
betas<- as.matrix(res)
# Input data
x_new <- as.matrix(c(1,1,0,1,0,1,0,1.2,0.8))
##### This code has been modified
# calculating the bet for Poisson since we have our y follow poisson
beta_pois<-exp(betas  %*% x_new)
# Finding number of bidders using poisson function
nbiders<-data.frame(x=rpois(length(beta_pois),beta_pois))
ggplot(nbiders,aes(x = x)) +
  geom_histogram(aes(y=..density..),linetype=1,fill='#14213D',bins = 20)+
    labs(x = 'Number of bidders', y = 'density',
           title ='Plot of the predictive distribution for Number of bidders')
ar_process<- function(phi,mu, sigma,t){
  x_t<-c()
  x_t[1]<-mu
  for(i in 2:t){
    e<- rnorm(1,0,sqrt(3))
    x_t[i]<-mu+(phi*(x_t[i-1]-mu))+e
  }
  return(x_t)
}
phi<-seq(-1,1,by=.5)
res<- list()
for (i in phi) {
  res1 <- ar_process(i, 13, 3, 300)
  res[[paste0("Phi_", i)]] <- res1
}

res<- data.frame(res)
names<-colnames(res)
p_fun<- function(coln){
  plt <- ggplot(res,aes(x = 1:300)) +
    geom_line(aes_string(y = coln),color='#FCA311', size=.8)+
  labs(title = coln,
       x= 'Simulations', y='Phi Value')+ theme_classic()
  plt
}

plot(arrangeGrob(grobs = lapply(names, p_fun)))
data {
  int<lower=0> N;
  vector[N] y;
}
parameters {
  real alpha;
  real beta;
  real<lower=0> sigma;
}
model {
  for (n in 2:N) {
    y[n] ~ normal(alpha + beta * y[n-1], sigma);
```

```
  }
}
data {
  int<lower=0> N;
  vector[N] y;
}
parameters {
  real alpha;
  real beta;
  real<lower=0> sigma;
}
model {
   y[2:N] ~ normal(alpha + beta * y[1:(N - 1)], sigma);
}
##### Example from notes #####
# library(rstan)
# y=c(4,5,6,4,0,2,5,3,8,6,10,8)
# N=length(y)
#
# StanModel = '
# data {
#   int<lower=0> N; // Number of observations
#   int<lower=0> y[N]; // Number of flowers
# }
# parameters {
#   real mu;
#   real<lower=0> sigma2;
# }
# model {
#   mu ~ normal(0,100); // Normal with mean 0, st.dev. 100
#   sigma2 ~ scaled_inv_chi_square(1,2); // Scaled-inv-chi2 with nu 1,sigma 2
#   for(i in 1:N){
#     y[i] ~ normal(mu,sqrt(sigma2));
#   }
# }'
#
# data <- list(N=N, y=y)
# warmup <- 1000
# niter <- 2000
# fit <- stan(model_code=StanModel,data=data, warmup=warmup,iter=niter,chains=4)
# # Print the fitted model
# print(fit,digits_summary=3)
# # Extract posterior samples
# postDraws <- extract(fit)
# # Do traceplots of the first chain
# par(mfrow = c(1,1))
# plot(postDraws$mu[1:(niter-warmup)],type="l",ylab="mu",main="Traceplot")
# # Do automatic traceplots of all chains
# traceplot(fit)
# # Bivariate posterior plots
# pairs(fit)
# ##############################
StanModel = '
```

```
data {
  int<lower=0> N; // Number of observations
  vector[N] y;
}
parameters {
  real mu;
  real<lower=0> sigma2;
  real<lower=-1, upper=1> phi;
  //To enforce the estimation of a
  //stationary AR(1) process, the slope
  //coefficient beta may be constrained with bounds as follows.
}
model {
  mu ~ normal(0,100); // Normal with mean 0, st.dev. 100
  sigma2 ~ scaled_inv_chi_square(1,2); // Scaled-inv-chi2 with nu 1,sigma 2
  // Changing the model to be y_i-mu
  y[2:N] ~ normal(mu + phi * (y[1:(N - 1)]-mu), sqrt(sigma2^2));
}'

# Simulate of x
ar_x<-ar_process(.2, 13, 3, 300)

#From lec notes we have the stanmodel function defined as
y=ar_x
N=length(y)

data <- list(N=N, y=y)
warmup <- 1000
niter <- 2000
fitx <- stan(model_code=StanModel,data=data,warmup=warmup,iter=niter,chains=4)
# Print the fitted model
print(fitx,digits_summary=3)
samples_x <-extract(fitx, pars = c("mu", "sigma2", "phi"))
# Compute the mean
mean_x <- sapply(samples_x, mean)
# 95% credible intervals
intv_x <- sapply(samples_x,function(samples) quantile(samples,c(0.025, 0.975)))
# effective posterior samples
eff_samp_x <- summary(fitx)$summary[1:3,"n_eff"]

result <- data.frame( Mean = mean_x,
                      Credible_Interval_Lower = intv_x[1, ],
                      Credible_Interval_Upper = intv_x[2, ],
                      Effective_Samples = eff_samp_x)
print(result)
# Simulate of y
ar_y<-ar_process(.95, 13, 3, 300)
#From lec notes we have the stanmodel function defined as
#y=ar_y
N=length(y)

data <- list(N=N, y=y)
warmup <- 1000
```

```
niter <- 2000
fity <- stan(model_code=StanModel,data=data, warmup=warmup,iter=niter,chains=4)
# Print the fitted model
print(fity,digits_summary=3)
samples_y <-extract(fity, pars = c("mu", "sigma2", "phi"))
# Compute the mean
mean_y <- sapply(samples_y, mean)
# 95% credible intervals
intv_y <- sapply(samples_y,function(samples) quantile(samples,c(0.025, 0.975)))
# effective posterior samples
eff_samp_y <- summary(fitx)$summary[1:3,"n_eff"]

result <- data.frame( Mean = mean_y,
                      Credible_Interval_Lower = intv_y[1, ],
                      Credible_Interval_Upper = intv_y[2, ],
                      Effective_Samples = eff_samp_y)
print(result)
df_x <-as.data.frame(extract(fitx, pars = c("mu", "phi")))
names<-colnames(df_x)
ln<-length(df_x[,1])
p_fun<- function(coln){
  plt <- ggplot(df_x,aes(x = 1:ln)) +
    geom_line(aes_string(y = coln),color='#FCA311', size=.8)+
    labs(title = coln,
         x= 'Simulations', y='Phi Value')+ theme_classic()
  plt
}

plot(arrangeGrob(grobs = lapply(names, p_fun)))
df_y <-as.data.frame(extract(fity, pars = c("mu", "phi")))
names<-colnames(df_y)
ln<-length(df_y[,1])
p_fun<- function(coln){
  plt <- ggplot(df_y,aes(x = 1:ln)) +
    geom_line(aes_string(y = coln),color='#FCA311', size=.8)+
    labs(title = coln,
         x= 'Simulations', y='Phi Value')+ theme_classic()
  plt
}

plot(arrangeGrob(grobs = lapply(names, p_fun)))
#### In case of normal
# library(rstan)
# y=c(4,5,6,4,0,2,5,3,8,6,10,8)
# N=length(y)
# StanModel = '
# data {
# int<lower=0> N; // Number of observations
# int<lower=0> y[N]; // Number of flowers
# }
# parameters {
# real mu;
# real<lower=0> sigma2;
```

```
# }
# model {
# mu ~ normal(0,100); // Normal with mean 0, st.dev. 100
# sigma2 ~ scaled_inv_chi_square(1,2); // Scaled-inv-chi2 with nu 1,sigma 2
# for(i in 1:N){
# y[i] ~ normal(mu,sqrt(sigma2));
# }
# }'
########### Multilevel normal
# StanModel <- '
# data {
# int<lower=0> N; // Number of observations
# int<lower=0> y[N]; // Number of flowers
# int<lower=0> P; // Number of plants
# }
# transformed data {
# int<lower=0> M; // Number of months
# M = N / P;
# }
# parameters {
# real mu;
# real<lower=0> sigma2;
# real mup[P];
# real sigmap2[P];
# }
# model {
# mu ~ normal(0,100); // Normal with mean 0, st.dev. 100
# sigma2 ~ scaled_inv_chi_square(1,2); // Scaled-inv-chi2 with nu 1, sigma 2
# for(p in 1:P){
# mup[p] ~ normal(mu,sqrt(sigma2));
# for(m in 1:M) {
# y[M*(p-1)+m] ~ normal(mup[p],sqrt(sigmap2[p]));
# }
# }
# }'
##############Possion
# StanModel <- '
# data {
# int<lower=0> N; // Number of observations
# int<lower=0> y[N]; // Number of flowers
# int<lower=0> P; // Number of plants
# }
# transformed data {
# int<lower=0> M; // Number of months
# M = N / P;
# }
# parameters {
# real mu;
# real<lower=0> sigma2;
# real mup[P];
# }
# model {
# mu ~ normal(0,100); // Normal with mean 0, st.dev. 100
```

```
# sigma2 ~ scaled_inv_chi_square(1,2); // Scaled-inv-chi2 with nu 1, sigma 2
# for(p in 1:P){
# mup[p] ~ lognormal(mu,sqrt(sigma2)); // Log-normal
# for(m in 1:M) {
# y[M*(p-1)+m] ~ poisson(mup[p]); // Poisson
# }
# }
# }'
### The original Code:

# Direct vs Gibbs sampling for bivariate normal distribution

# Initial setting
mu1 <- 1
mu2 <- 1
rho <- 0.9
mu <- c(mu1,mu2)
Sigma = matrix(c(1,rho,rho,1),2,2)
nDraws <- 500 # Number of draws

library(MASS) # To access the mvrnorm() function

# Direct sampling from bivariate normal distribution
directDraws <- mvrnorm(nDraws, mu, Sigma)

# Gibbs sampling
gibbsDraws <- matrix(0,nDraws,2)
theta2 <- 0 # Initial value for theta2
for (i in 1:nDraws){

  # Update theta1 given theta2
  theta1 <- rnorm(1, mean = mu1 + rho*(theta2-mu2), sd = sqrt(1-rho**2))
  gibbsDraws[i,1] <- theta1

  # Update theta2 given theta1
  theta2 <- rnorm(1, mean = mu2 + rho*(theta1-mu1), sd = sqrt(1-rho**2))
  gibbsDraws[i,2] <- theta2

}

a_Direct <- acf(directDraws[,1])
a_Gibbs <- acf(gibbsDraws[,1])

IF_Gibbs <- 1+2*sum(a_Gibbs$acf[-1])

par(mfrow=c(2,4))

# DIRECT SAMPLING
plot(1:nDraws, directDraws[,1], type = "l",col="blue") # traceplot of direct draws

hist(directDraws[,1],col="blue") # histogram of direct draws
) # Cumulative mean value of theta1, direct draws
cusumData <- cumsum(directDraws[,1])/seq(1,nDraws
```

```r
plot(1:nDraws, cusumData, type = "l", col="blue")

barplot(height = a_Direct$acf[-1],col="blue") # acf for direct draws

# GIBBS SAMPLING
plot(1:nDraws, gibbsDraws[,1], type = "l",col="red") # traceplot of Gibbs draws

hist(gibbsDraws[,1],col="red") # histogram of Gibbs draws
# Cumulative mean value of theta1, Gibbs draws
cusumData =  cumsum(gibbsDraws[,1])/seq(1,nDraws)
plot(1:nDraws, cusumData, type = "l", col="red")

barplot(height = a_Gibbs$acf[-1],col="red") # acf for Gibbs draws

# Plotting the cumulative path of estimates of Pr(theta1>0, theta2>0)
par(mfrow=c(2,1))
plot(cumsum(directDraws[,1]>0 & directDraws[,2]>0)/seq(1,nDraws),type="l",
     main='Direct draws', xlab='Iteration number', ylab='', ylim = c(0,1))
plot(cumsum(gibbsDraws[,1]>0 & gibbsDraws[,2]>0)/seq(1,nDraws),type="l",
     main='Gibbs draws', xlab='Iteration number', ylab='', ylim = c(0,1))


#######################################################
```