Bayesian Learning TDDE07
Frida Andersson, frian632
Simon Boman, simbo160

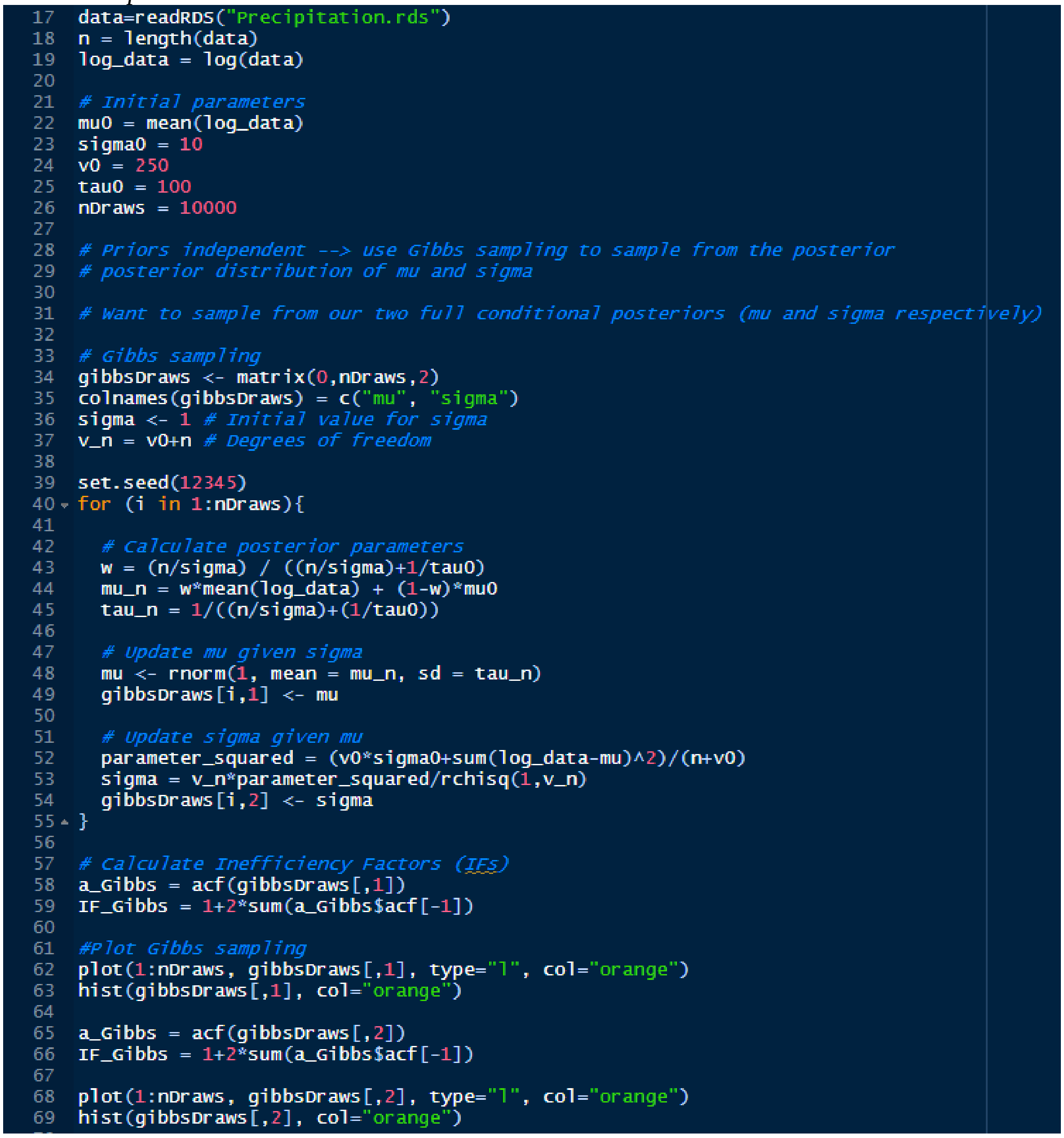
Bayesian Learning Computer Lab 3

- Include all solutions and plots to the stated problems with necessary comments
- Submit report with code attached to the solution of each sub-problem in code PDF document
- Submit a separate file containing all code

1. Gibbs sampler for a normal model

Given: *μ and σ^2 is unknown.*

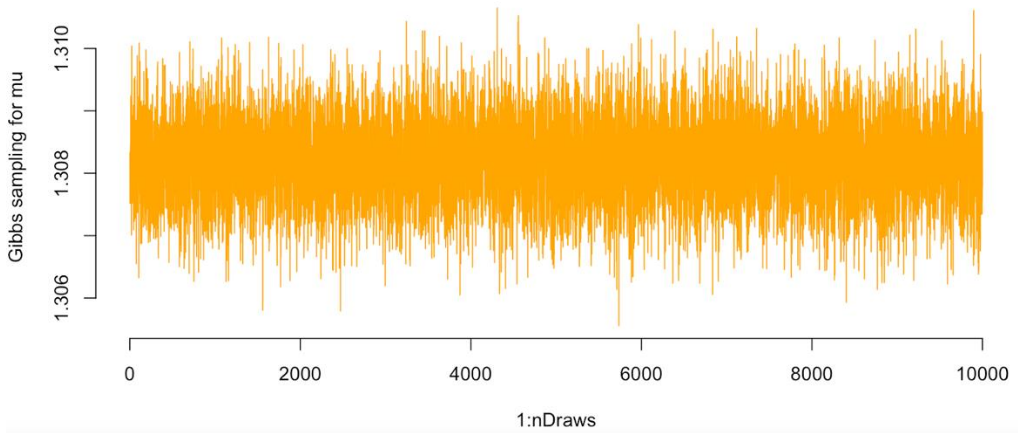
A. *Implement a Gibbs sampler that simulates from the joint posterior and evaluate the convergence by calculating the Inefficiency Factors (IFs) and by plotting the trajectories of the sampled Markov chains*

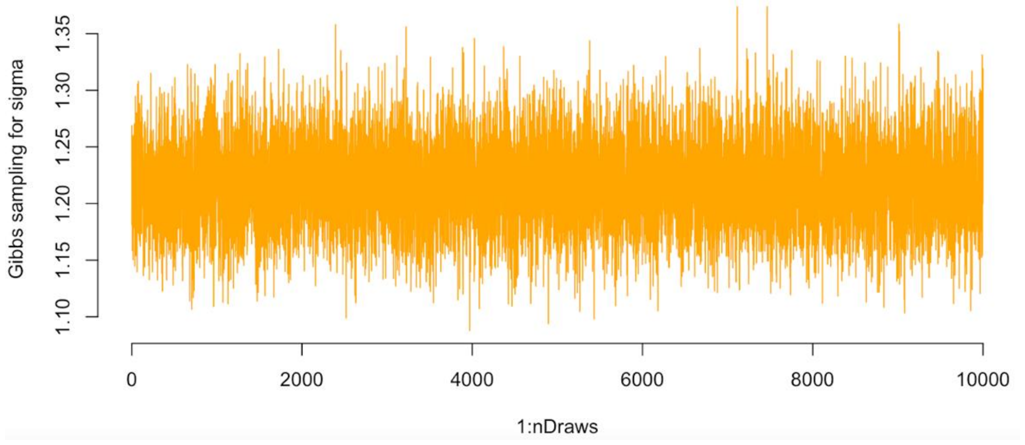


```r
data=readRDS("Precipitation.rds")
n = length(data)
log_data = log(data)

# Initial parameters
mu0 = mean(log_data)
sigma0 = 10
v0 = 250
tau0 = 100
nDraws = 10000

# Priors independent --> use Gibbs sampling to sample from the posterior
# posterior distribution of mu and sigma

# Want to sample from our two full conditional posteriors (mu and sigma respectively)

# Gibbs sampling
gibbsDraws <- matrix(0,nDraws,2)
colnames(gibbsDraws) = c("mu", "sigma")
sigma <- 1 # Initial value for sigma
v_n = v0+n # Degrees of freedom

set.seed(12345)
for (i in 1:nDraws){

  # Calculate posterior parameters
  w = (n/sigma) / ((n/sigma)+1/tau0)
  mu_n = w*mean(log_data) + (1-w)*mu0
  tau_n = 1/((n/sigma)+(1/tau0))

  # Update mu given sigma
  mu <- rnorm(1, mean = mu_n, sd = tau_n)
  gibbsDraws[i,1] <- mu

  # Update sigma given mu
  parameter_squared = (v0*sigma0+sum(log_data-mu)^2)/(n+v0)
  sigma = v_n*parameter_squared/rchisq(1,v_n)
  gibbsDraws[i,2] <- sigma
}

# Calculate Inefficiency Factors (IFs)
a_Gibbs = acf(gibbsDraws[,1])
IF_Gibbs = 1+2*sum(a_Gibbs$acf[-1])

#Plot Gibbs sampling
plot(1:nDraws, gibbsDraws[,1], type="l", col="orange")
hist(gibbsDraws[,1], col="orange")

a_Gibbs = acf(gibbsDraws[,2])
IF_Gibbs = 1+2*sum(a_Gibbs$acf[-1])

plot(1:nDraws, gibbsDraws[,2], type="l", col="orange")
hist(gibbsDraws[,2], col="orange")
```

We start by choosing our initial parameters as μ_0= mean(log_data), $\sigma^2_0 = 10$, $v_0 = 250$, $\tau^2_0 = 100$. Since our priors are independent, we can implement a Gibbs sampler that samples from the posterior distribution of mu and sigma, respectively. We set our initial value for σ^2 to 1 and our degrees of freedom $v_n = v_0 + n$.

In our Gibbs sampler we calculate our posterior parameters and update our parameters μ and σ^2 using the full conditional posteriors from Lecture 7 slide 16. An IF value of 1 indicates that our Gibbs sampling is as efficient as direct draws. Since we obtain IF = 0.914199 for μ and IF = 1.034939 for σ^2, our Gibbs sampler can be considered good. Figures below show the trajectories of the sampled Markov chains for μ and σ^2, respectively. As we can see, both chains converge around the same value.





B. *Plot a histogram or kernel density estimate of the daily precipitation, and the resulting posterior predictive density in one figure. How well does the posterior predictive density agree with this data?*

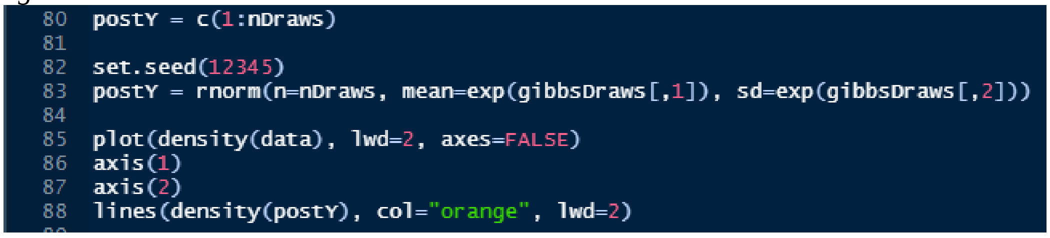
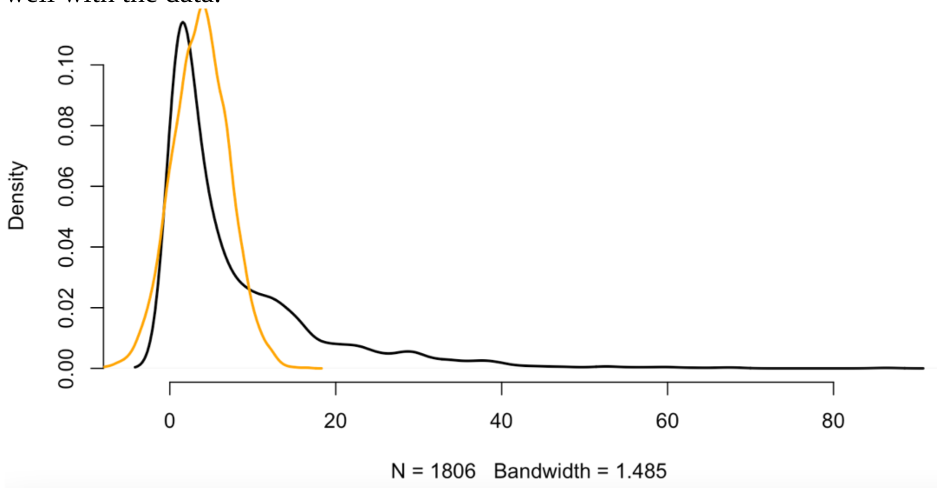

```
postY = c(1:nDraws)

set.seed(12345)
postY = rnorm(n=nDraws, mean=exp(gibbsDraws[,1]), sd=exp(gibbsDraws[,2]))

plot(density(data), lwd=2, axes=FALSE)
axis(1)
axis(2)
lines(density(postY), col="orange", lwd=2)
```

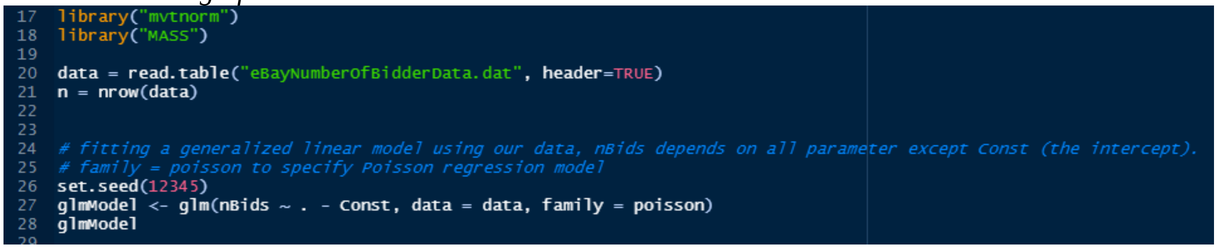
Figure below shows a kernel density estimate of the daily precipitation and posterior predictive density in the same figure. As we can see, the posterior predictive density agrees well with the data.



2. Metropolis Random Walk for Poisson regression

Given: *n =1000.*

A. *Obtain the maximum likelihood estimator of β in the Poisson regression model. Which covariates are significant?*



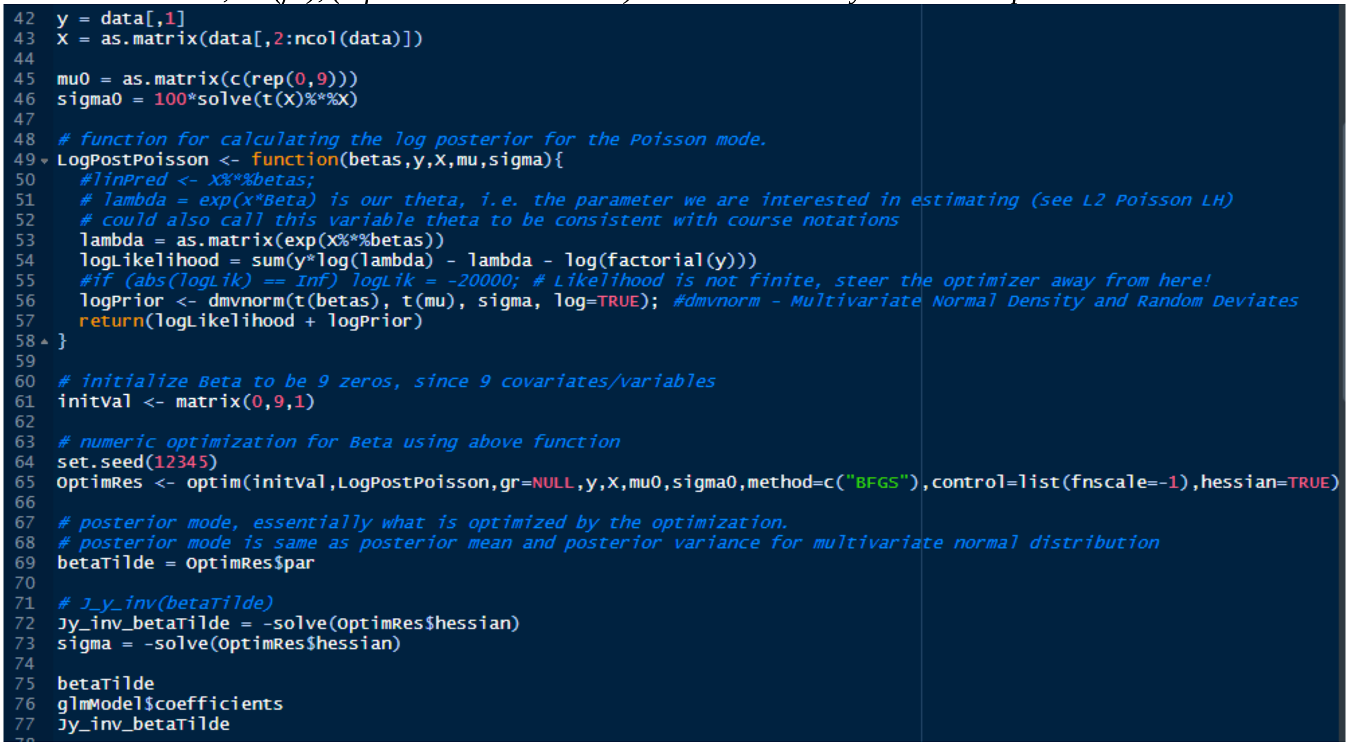
```r
library("mvtnorm")
library("MASS")

data = read.table("eBayNumberOfBidderData.dat", header=TRUE)
n = nrow(data)


# fitting a generalized linear model using our data, nBids depends on all parameter except Const (the intercept).
# family = poisson to specify Poisson regression model
set.seed(12345)
glmModel <- glm(nBids ~ . - Const, data = data, family = poisson)
glmModel
```

The maximum likelihood estimator of β in the Poisson regression model can be obtained using the glm()-function.

```
Call:  glm(formula = nBids ~ . - Const, family = poisson, data = data)

Coefficients:
(Intercept)  PowerSeller     VerifyID       Sealed      Minblem      MajBlem      LargNeg      LogBook  MinBidShare
    1.07244     -0.02054     -0.39452      0.44384     -0.05220     -0.22087      0.07067     -0.12068     -1.89410
```

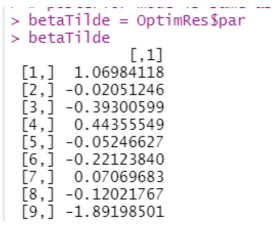
We can then see that out of the covariates, MinBidShare is the most significant as it has the largest coefficient, followed by Sealed and VerifyID. One could also argue that MajBlem is significant, but we consider the rest of the covariates to not be significant.

B. *Do a Bayesian analysis of the Poisson regression. β' (mode) and the inverse of the hessian at the mode, J $^{-1}$(β'), (information at the mode) can be obtained by numerical optimization.*

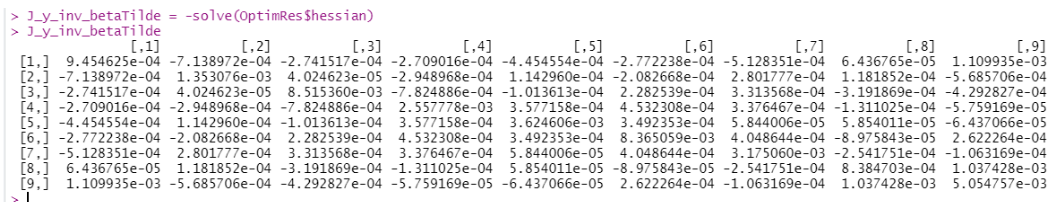


```r
y = data[,1]
X = as.matrix(data[,2:ncol(data)])

mu0 = as.matrix(c(rep(0,9)))
sigma0 = 100*solve(t(X)%*%X)

# function for calculating the log posterior for the Poisson mode.
LogPostPoisson <- function(betas,y,X,mu,sigma){
  #linPred <- X%*%betas;
  # lambda = exp(x*Beta) is our theta, i.e. the parameter we are interested in estimating (see L2 Poisson LH)
  # could also call this variable theta to be consistent with course notations
  lambda = as.matrix(exp(X%*%betas))
  logLikelihood = sum(y*log(lambda) - lambda - log(factorial(y)))
  #if (abs(logLik) == Inf) logLik = -20000; # Likelihood is not finite, steer the optimizer away from here!
  logPrior <- dmvnorm(t(betas), t(mu), sigma, log=TRUE); #dmvnorm - Multivariate Normal Density and Random Deviates
  return(logLikelihood + logPrior)
}

# initialize Beta to be 9 zeros, since 9 covariates/variables
initVal <- matrix(0,9,1)

# numeric optimization for Beta using above function
set.seed(12345)
OptimRes <- optim(initVal,LogPostPoisson,gr=NULL,y,X,mu0,sigma0,method=c("BFGS"),control=list(fnscale=-1),hessian=TRUE)

# posterior mode, essentially what is optimized by the optimization.
# posterior mode is same as posterior mean and posterior variance for multivariate normal distribution
betaTilde = OptimRes$par

# J_y_inv(betaTilde)
Jy_inv_betaTilde = -solve(OptimRes$hessian)
sigma = -solve(OptimRes$hessian)

betaTilde
glmModel$coefficients
Jy_inv_betaTilde
```

Similarly to how we did in lab2, we can define a function that calculates the log posterior for the Poisson model. We can then initialize our β parameter vector (i.e., our theta) to a vector of zeros and then use optim to obtain the parameters by numerical optimization. We then get the following β' and J $^{-1}$ (β').

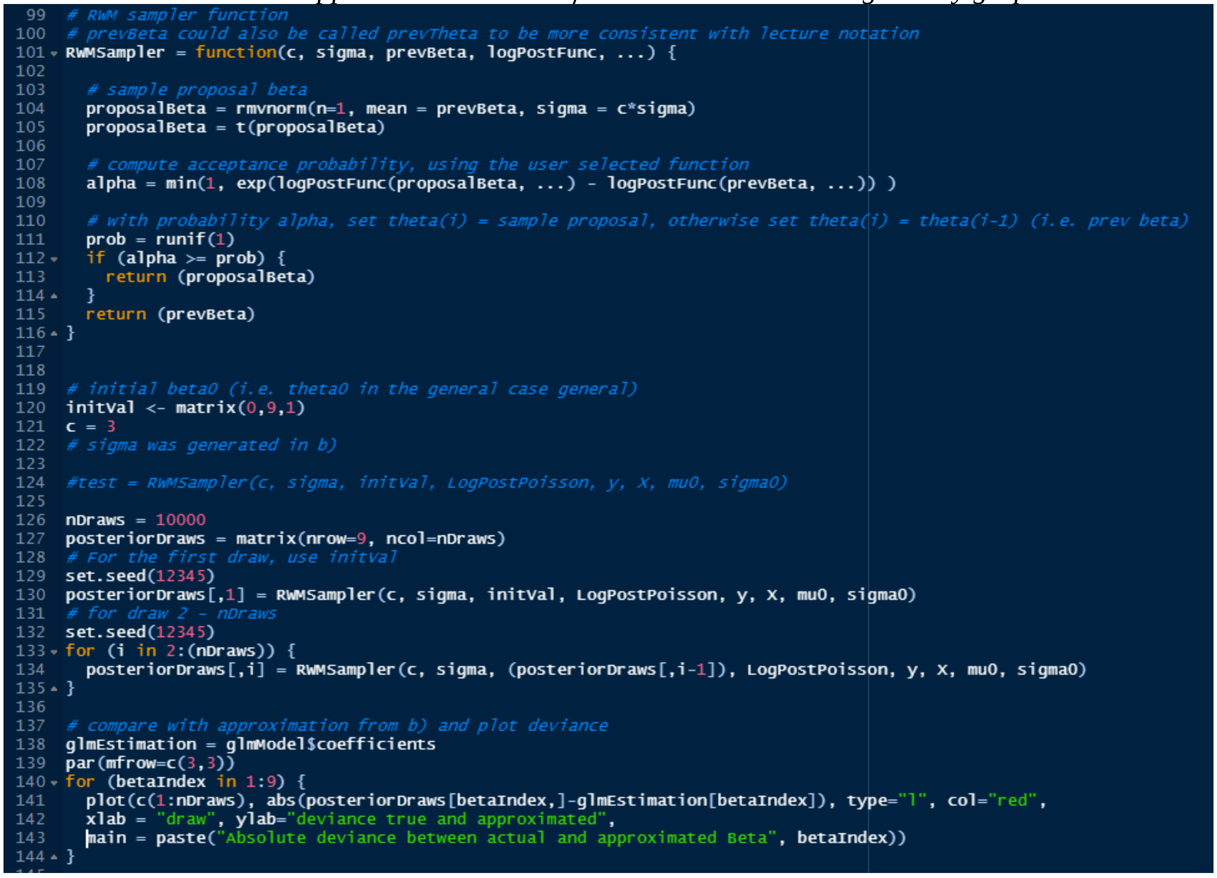


```
> betaTilde = OptimRes$par
> betaTilde
            [,1]
 [1,]  1.06984118
 [2,] -0.02051246
 [3,] -0.39300599
 [4,]  0.44355549
 [5,] -0.05246627
 [6,] -0.22123840
 [7,]  0.07069683
 [8,] -0.12021767
 [9,] -1.89198501
```



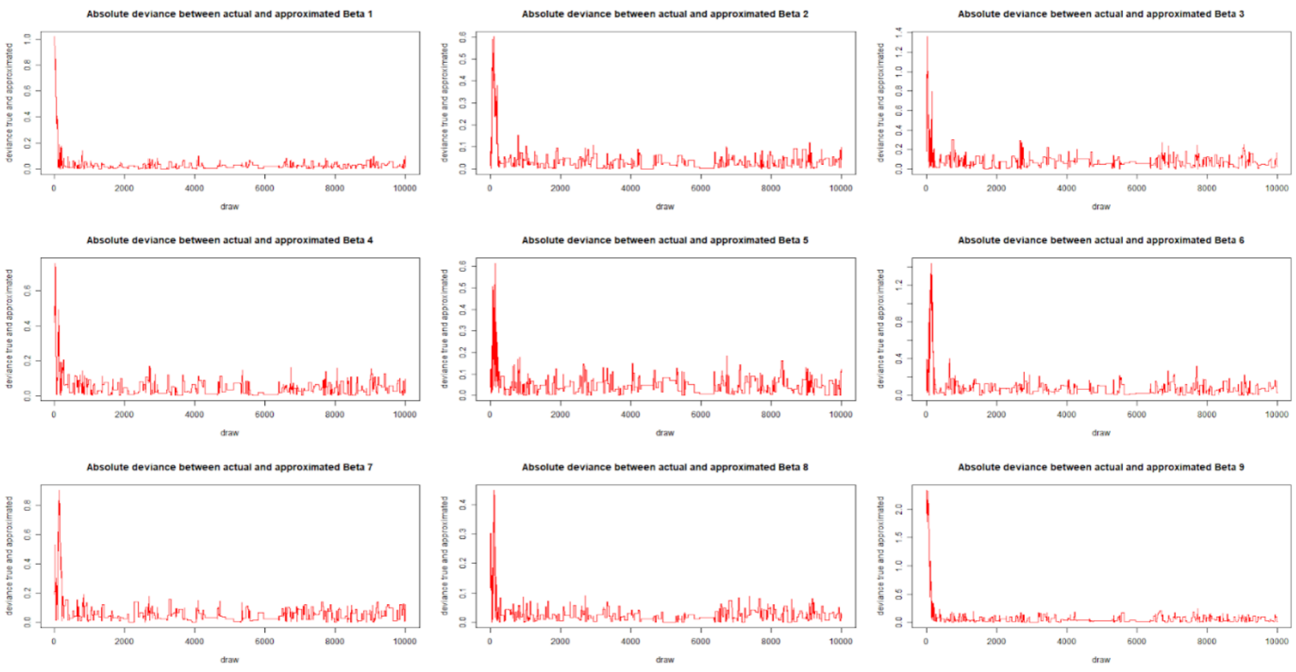
```
> J_y_inv_betaTilde = -solve(OptimRes$hessian)
> J_y_inv_betaTilde
               [,1]          [,2]          [,3]          [,4]          [,5]          [,6]          [,7]          [,8]          [,9]
 [1,]  9.454625e-04 -7.138972e-04 -2.741517e-04 -2.709016e-04 -4.454554e-04 -2.772238e-04 -5.128351e-04  6.436765e-05  1.109935e-03
 [2,] -7.138972e-04  1.353076e-03  4.024623e-05 -2.948968e-04  1.142960e-04 -2.082668e-04  2.801777e-04  1.181852e-04 -5.685706e-04
 [3,] -2.741517e-04  4.024623e-05  8.515360e-03 -7.824886e-04 -1.013613e-04  2.282539e-04  3.313568e-04 -3.191869e-04 -4.292827e-04
 [4,] -2.709016e-04 -2.948968e-04 -7.824886e-04  2.557778e-03  3.577158e-04  4.532308e-04  3.376467e-04 -1.311025e-04 -5.759169e-05
 [5,] -4.454554e-04  1.142960e-04 -1.013613e-04  3.577158e-04  3.624606e-03  3.492353e-04  5.844006e-05  5.854011e-05 -6.437066e-05
 [6,] -2.772238e-04 -2.082668e-04  2.282539e-04  4.532308e-04  3.492353e-04  8.365059e-03  4.048644e-04 -8.975843e-05  2.622264e-04
 [7,] -5.128351e-04  2.801777e-04  3.313568e-04  3.376467e-04  5.844006e-05  4.048644e-04  3.175060e-03 -2.541751e-04 -1.063169e-04
 [8,]  6.436765e-05  1.181852e-04 -3.191869e-04 -1.311025e-04  5.854011e-05 -8.975843e-05 -2.541751e-04  8.384703e-04  1.037428e-03
 [9,]  1.109935e-03 -5.685706e-04 -4.292827e-04 -5.759169e-05 -6.437066e-05  2.622264e-04 -1.063169e-04  1.037428e-03  5.054757e-03
>
```

C. *Simulate from the actual posterior of β using the Metropolis algorithm and compare the results with the approximate results in β. Assess MCMC convergence by graphical methods.*

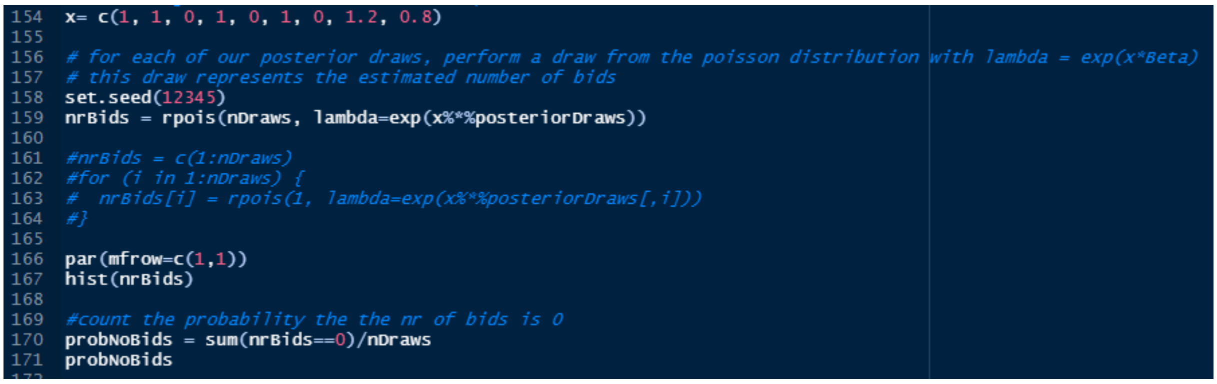


```r
# RWM sampler function
# prevBeta could also be called prevTheta to be more consistent with lecture notation
RWMSampler = function(c, sigma, prevBeta, logPostFunc, ...) {

  # sample proposal beta
  proposalBeta = rmvnorm(n=1, mean = prevBeta, sigma = c*sigma)
  proposalBeta = t(proposalBeta)

  # compute acceptance probability, using the user selected function
  alpha = min(1, exp(logPostFunc(proposalBeta, ...) - logPostFunc(prevBeta, ...)) )

  # with probability alpha, set theta(i) = sample proposal, otherwise set theta(i) = theta(i-1) (i.e. prev beta)
  prob = runif(1)
  if (alpha >= prob) {
    return (proposalBeta)
  }
  return (prevBeta)
}


# initial beta0 (i.e. theta0 in the general case general)
initVal <- matrix(0,9,1)
c = 3
# sigma was generated in b)

#test = RWMSampler(c, sigma, initVal, LogPostPoisson, y, X, mu0, sigma0)

nDraws = 10000
posteriorDraws = matrix(nrow=9, ncol=nDraws)
# For the first draw, use initVal
set.seed(12345)
posteriorDraws[,1] = RWMSampler(c, sigma, initVal, LogPostPoisson, y, X, mu0, sigma0)
# for draw 2 - nDraws
set.seed(12345)
for (i in 2:(nDraws)) {
  posteriorDraws[,i] = RWMSampler(c, sigma, (posteriorDraws[,i-1]), LogPostPoisson, y, X, mu0, sigma0)
}

# compare with approximation from b) and plot deviance
glmEstimation = glmModel$coefficients
par(mfrow=c(3,3))
for (betaIndex in 1:9) {
  plot(c(1:nDraws), abs(posteriorDraws[betaIndex,]-glmEstimation[betaIndex]), type="l", col="red",
  xlab = "draw", ylab="deviance true and approximated",
  main = paste("Absolute deviance between actual and approximated Beta", betaIndex))
}
```

We define a function that uses the Metropolis algorithm which can take any posterior density function, in our case we use the function for the log posterior for the Poisson model we defined in B. We then simulate posterior draws from the actual posterior of β using the Metropolis algorithm and compare it to our approximate results from B. The absolute difference between the draws from the actual posterior and the approximated posterior can be seen in the plot below, and is close to 0 for Beta.

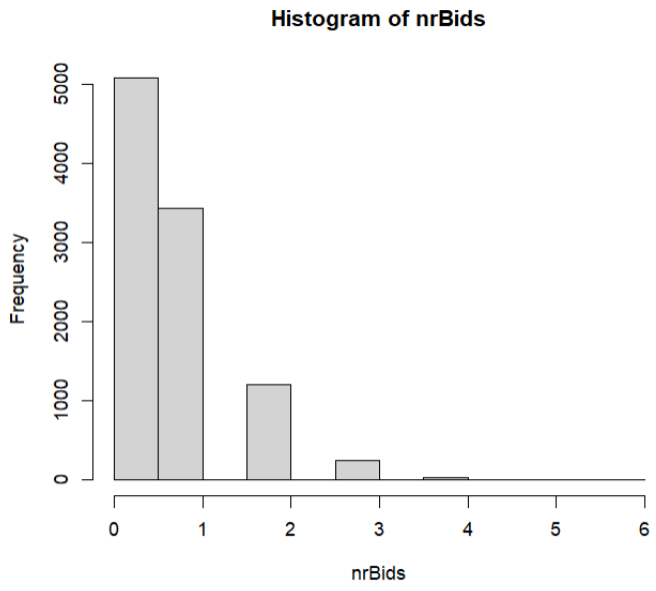


D. *Use the MCMC draws from C. to simulate from the predictive distribution of the number of bidders given x = [1, 1, 0, 1, 0, 1, 0, 1.2, 0.8]. What is the probability of no bidders in this auction?*



```r
x= c(1, 1, 0, 1, 0, 1, 0, 1.2, 0.8)

# for each of our posterior draws, perform a draw from the poisson distribution with lambda = exp(x*Beta)
# this draw represents the estimated number of bids
set.seed(12345)
nrBids = rpois(nDraws, lambda=exp(x%*%posteriorDraws))

#nrBids = c(1:nDraws)
#for (i in 1:nDraws) {
#  nrBids[i] = rpois(1, lambda=exp(x%*%posteriorDraws[,i]))
#}

par(mfrow=c(1,1))
hist(nrBids)

#count the probability the the nr of bids is 0
probNoBids = sum(nrBids==0)/nDraws
probNoBids
```

We can now make draws from the Poisson distribution with the exponential of x * our posterior β-draws as λ. We can then plot a histogram of the number of bids as can be seen in the next plot.



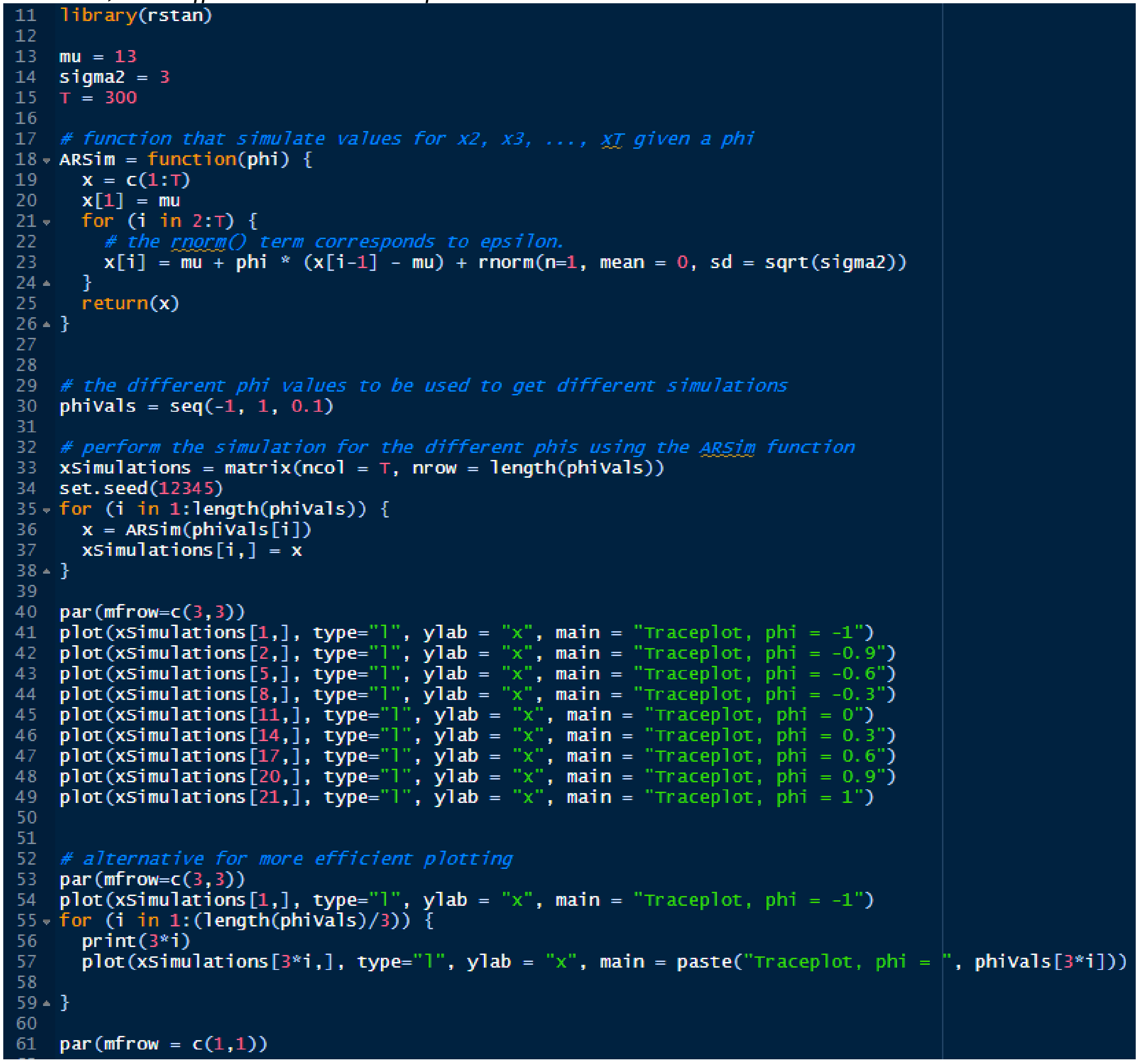
The probability that there are no bids is then the number of draws that are 0 divided by the total number of draws, which is 0.5073 = 50.73%.

3. Time series models in Stan

A. *Write a function in R that simulates data from the AR(1) process*

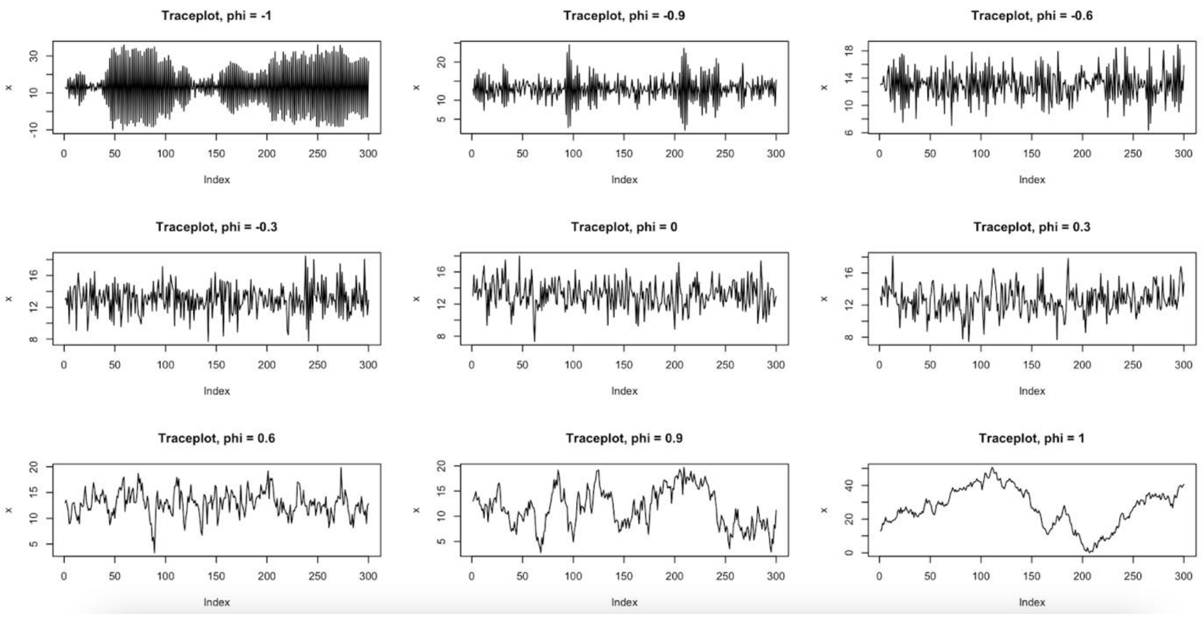
$$x_t = \mu + \phi\left(x_{t-1} - \mu\right) + \varepsilon_t, \quad \varepsilon_t \overset{iid}{\sim} N\left(0, \sigma^2\right),$$

For $\mu = 13$, $\sigma^2 = 3$ and $T = 300$. Look at different realizations for values of Φ between -1 and 1, what effect does the value of Φ have on $x_{1:T}$.

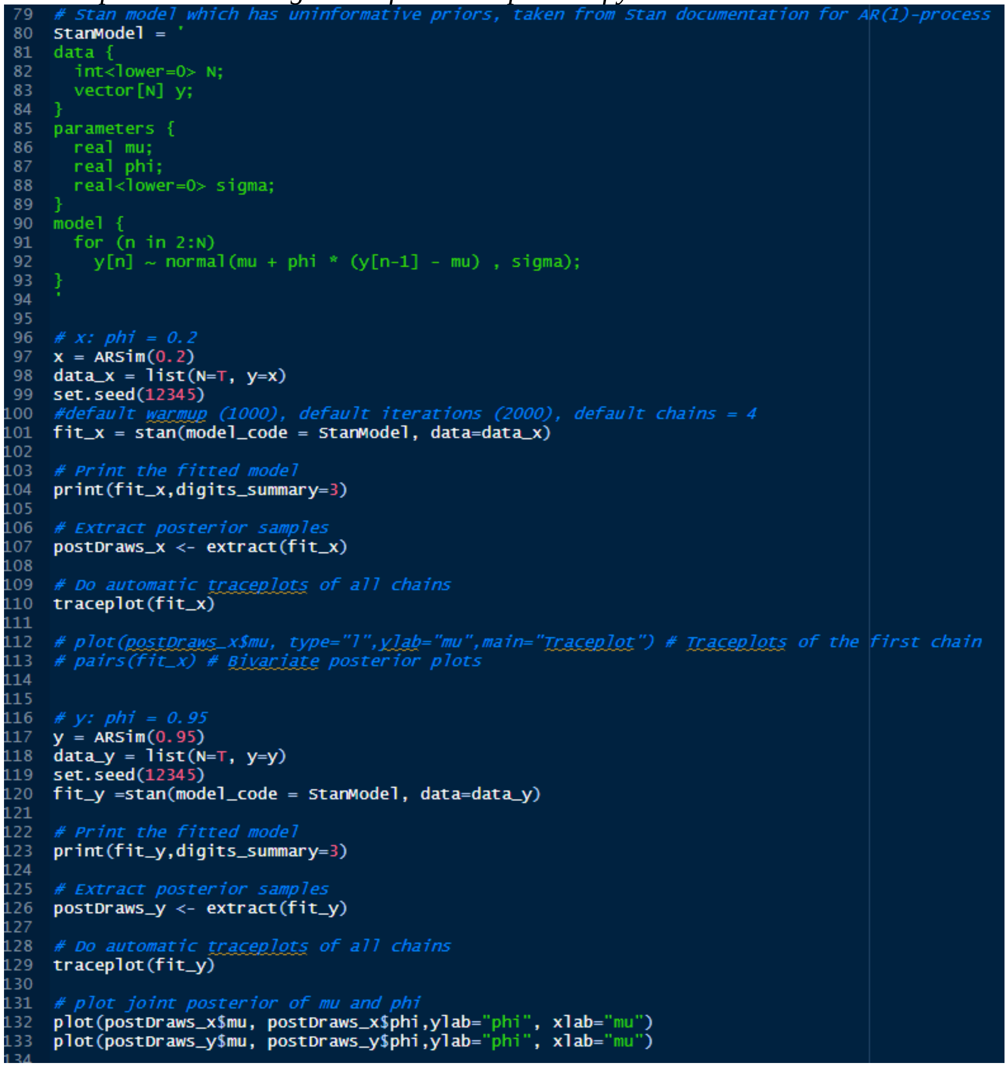


```r
library(rstan)

mu = 13
sigma2 = 3
T = 300

# function that simulate values for x2, x3, ..., xT given a phi
ARSim = function(phi) {
  x = c(1:T)
  x[1] = mu
  for (i in 2:T) {
    # the rnorm() term corresponds to epsilon.
    x[i] = mu + phi * (x[i-1] - mu) + rnorm(n=1, mean = 0, sd = sqrt(sigma2))
  }
  return(x)
}


# the different phi values to be used to get different simulations
phiVals = seq(-1, 1, 0.1)

# perform the simulation for the different phis using the ARSim function
xSimulations = matrix(ncol = T, nrow = length(phiVals))
set.seed(12345)
for (i in 1:length(phiVals)) {
  x = ARSim(phiVals[i])
  xSimulations[i,] = x
}

par(mfrow=c(3,3))
plot(xSimulations[1,], type="l", ylab = "x", main = "Traceplot, phi = -1")
plot(xSimulations[2,], type="l", ylab = "x", main = "Traceplot, phi = -0.9")
plot(xSimulations[5,], type="l", ylab = "x", main = "Traceplot, phi = -0.6")
plot(xSimulations[8,], type="l", ylab = "x", main = "Traceplot, phi = -0.3")
plot(xSimulations[11,], type="l", ylab = "x", main = "Traceplot, phi = 0")
plot(xSimulations[14,], type="l", ylab = "x", main = "Traceplot, phi = 0.3")
plot(xSimulations[17,], type="l", ylab = "x", main = "Traceplot, phi = 0.6")
plot(xSimulations[20,], type="l", ylab = "x", main = "Traceplot, phi = 0.9")
plot(xSimulations[21,], type="l", ylab = "x", main = "Traceplot, phi = 1")


# alternative for more efficient plotting
par(mfrow=c(3,3))
plot(xSimulations[1,], type="l", ylab = "x", main = "Traceplot, phi = -1")
for (i in 1:(length(phiVals)/3)) {
  print(3*i)
  plot(xSimulations[3*i,], type="l", ylab = "x", main = paste("Traceplot, phi = ", phiVals[3*i]))

}

par(mfrow = c(1,1))
```

By creating a function called ARSim with Φ as an in-parameter, we can look at different realizations for the AR(1) process. Using $x_1 = \mu$ as initial value, we can calculate $x_{1:300}$ in a for loop. To evaluate the different values for Φ, we can do a traceplot that shows how x converges. As we can see, a negative value for Φ makes x alternate between increasing and decreasing in value between each time point, while a positive value allows x to increase or decrease several iterations in a row. Thus, a negative phi causes more oscillation. We can also see that a strong negative value, for example Φ = -1 makes x oscillate more than a smaller negative value. The same goes for positive values, where Φ = 1 has no oscillating behavior (at least of we look at T=300), due to the strong dependence between the time points. A value around zero is to be preferred as x converges but within a low range, for example 8-18 when Φ = 0, compared to larger absolute values of phi where the spread grows more wide.



B. *Use the function from A. to simulate two AR(1)-processes using $\Phi = 0.2$ for $x_{1:T}$ and $\Phi = 0.95$ for $y_{1:T}$. Treat your simulated vectors as synthetic data and the values of μ, σ^2 and Φ as unknown parameters. Implement Stan-code that samples from the posterior of the three parameters using non-informative priors of your choice.*

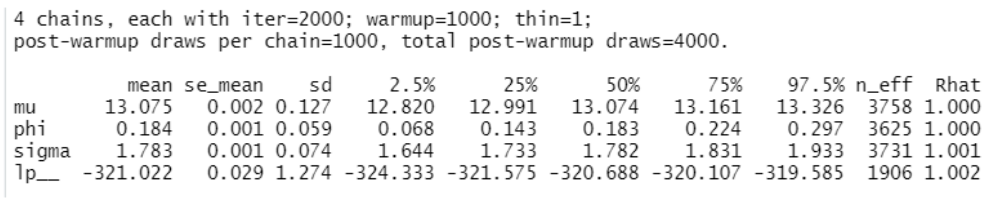


```
# Stan model which has uninformative priors, taken from Stan documentation for AR(1)-process
StanModel = '
data {
  int<lower=0> N;
  vector[N] y;
}
parameters {
  real mu;
  real phi;
  real<lower=0> sigma;
}
model {
  for (n in 2:N)
    y[n] ~ normal(mu + phi * (y[n-1] - mu) , sigma);
}
'

# x: phi = 0.2
x = ARSim(0.2)
data_x = list(N=T, y=x)
set.seed(12345)
#default warmup (1000), default iterations (2000), default chains = 4
fit_x = stan(model_code = StanModel, data=data_x)

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

# Extract posterior samples
postDraws_x <- extract(fit_x)

# Do automatic traceplots of all chains
traceplot(fit_x)

# plot(postDraws_x$mu, type="l",ylab="mu",main="Traceplot") # Traceplots of the first chain
# pairs(fit_x) # Bivariate posterior plots


# y: phi = 0.95
y = ARSim(0.95)
data_y = list(N=T, y=y)
set.seed(12345)
fit_y =stan(model_code = StanModel, data=data_y)

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

# Extract posterior samples
postDraws_y <- extract(fit_y)

# Do automatic traceplots of all chains
traceplot(fit_y)

# plot joint posterior of mu and phi
plot(postDraws_x$mu, postDraws_x$phi,ylab="phi", xlab="mu")
plot(postDraws_y$mu, postDraws_y$phi,ylab="phi", xlab="mu")
```

Using our Stan code and the simulated AR(1)-processes using Φ = 0.2 and Φ = 0.95 together with our function from A. we can sample from the posterior of the three parameters.

i. *Report the posterior mean, 95 % credible intervals and 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?*

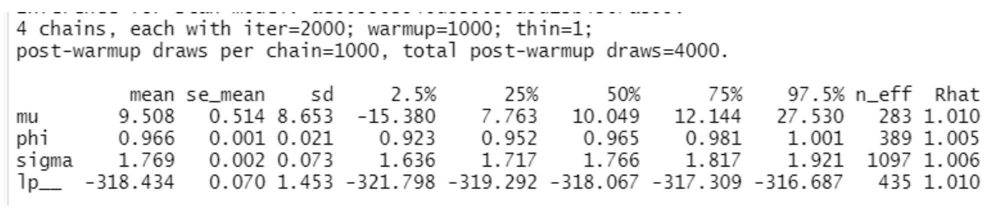
For $x_{1:T}$ (Φ = 0.2), we get the following results:



```
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% n_eff  Rhat
mu      13.075   0.002 0.127   12.820   12.991   13.074   13.161   13.326  3758 1.000
phi      0.184   0.001 0.059    0.068    0.143    0.183    0.224    0.297  3625 1.000
sigma    1.783   0.001 0.074    1.644    1.733    1.782    1.831    1.933  3731 1.001
lp__  -321.022   0.029 1.274 -324.333 -321.575 -320.688 -320.107 -319.585  1906 1.002
```

The posterior means can be seen in the *mean* column, and for the 95% credible intervals the lower bound can be seen in the *2.5%* column and the upper bound can be seen in the *97.5%* column. The number of effective posterior draws can be seen in the *n_eff* column.

Similarly, for $y_{1:T}$ (Φ = 0.95), we get the following results:



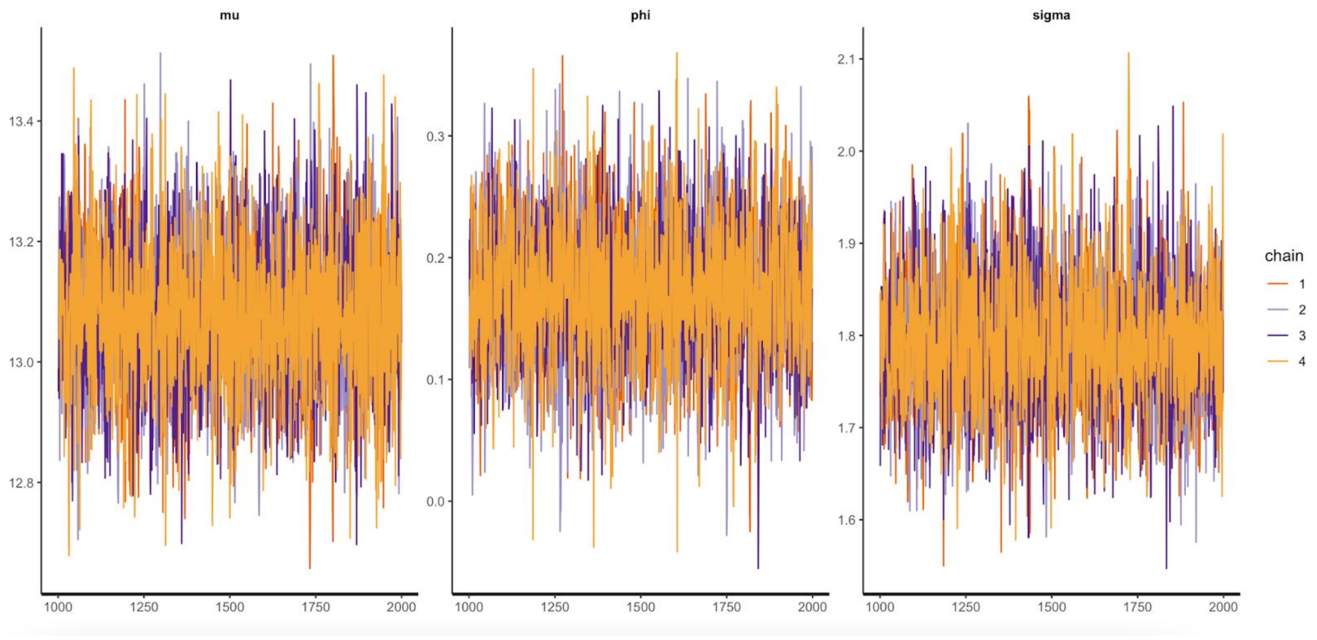
```
4 chains, each with iter=2000; warmup=1000; thin=1;
post-warmup draws per chain=1000, total post-warmup draws=4000.

          mean se_mean    sd     2.5%      25%      50%      75%    97.5% n_eff  Rhat
mu       9.508   0.514 8.653  -15.380    7.763   10.049   12.144   27.530   283 1.010
phi      0.966   0.001 0.021    0.923    0.952    0.965    0.981    1.001   389 1.005
sigma    1.769   0.002 0.073    1.636    1.717    1.766    1.817    1.921  1097 1.006
lp__  -318.434   0.070 1.453 -321.798 -319.292 -318.067 -317.309 -316.687   435 1.010
```

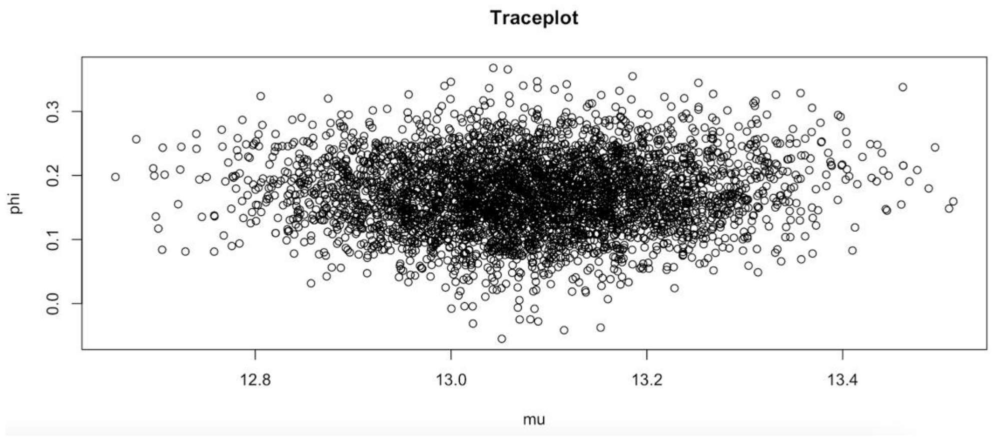
For x, we get μ = 13.018 which is close to the true value of 13, Φ = 0.230 which is also quite close to the true value of 0.2, and a variance of 1.741^2 = 3.031 which is close to the true variance of 3.

For y, μ = 9.508 differs a lot from the true μ of 13. This is in line with our discussion in A., and we can see that the 95% credible interval for mu is very wide when using Φ = 0.95. However, the estimated Φ and σ^2 are close to the true values.

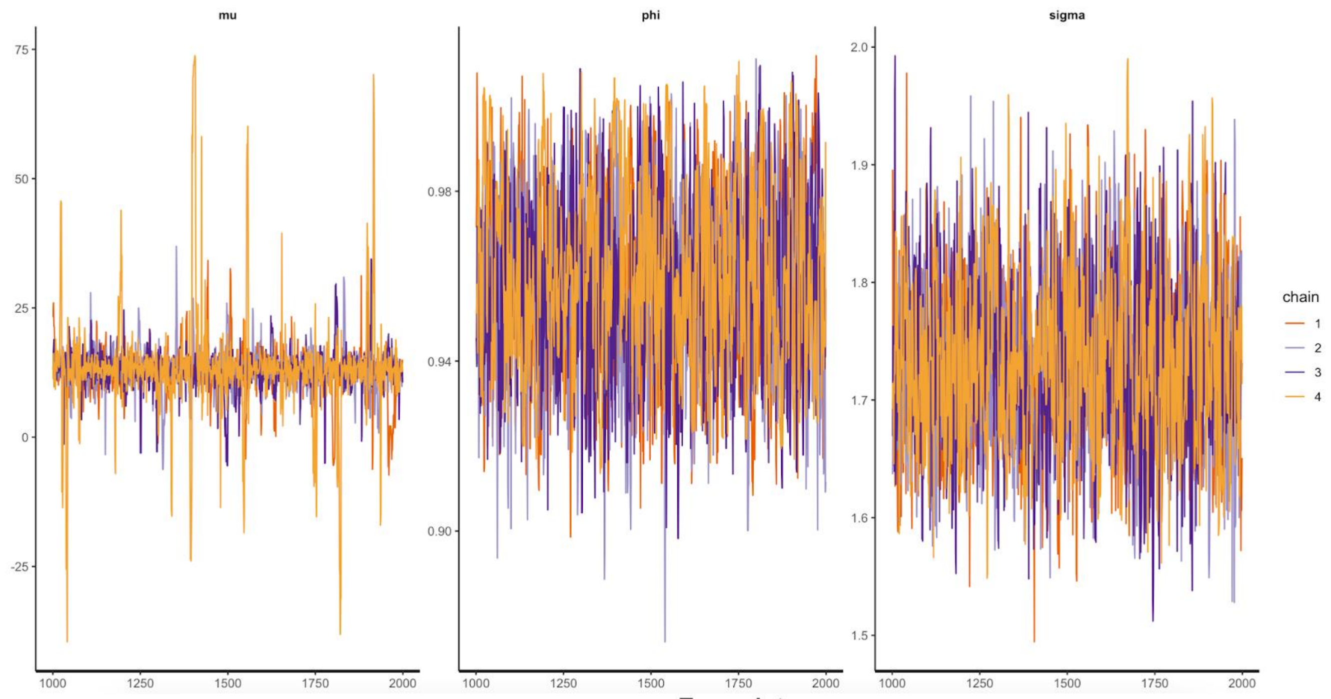
ii. *For each of the two data sets, evaluate the convergence of the samplers and plot the joint posterior of μ and Φ. Comments?*

Figure below shows the convergence of the samplers for X. As we can see, μ, Φ and σ^2 converge around the true values (13, 0.2, sqrt(3)). By plotting the joint posterior of μ and Φ for X, we can see that the distribution is even and centered around the true values for μ and Φ.





The figure below shows the convergence of the sampler for y, and although the sampling converges, we do not converge to the true value of µ. The plot of the joint posterior of µ and Φ also illustrates that distribution is not centered around the true value of µ, especially for values of Φ close to 1. This also illustrates that as Φ increase to be close to 1, the spread of possible values for µ increases, which is in line with what we talked about in A. Consequently, the distribution does not represent the true values for µ and Φ as shown in the figure.



Traceplot