4

Introduction to Bayesian Time-Series Analysis using JAGS

In this lab, we'll work through using Bayesian methods to estimate parameters in time series models using JAGS. There's a variety of software tools to do this. R has a number of packages available on the TimeSeries task view,

`http://cran.r-project.org/web/views/TimeSeries.html`

Software to implement more complicated models is also available, and many of you are probably familiar with these options (AD Model Builder and Template Model Builder, WinBUGS, OpenBUGS, JAGS, to name a few). We'll be using JAGS because (1) it's platform independent, (2) most of the glitches have been worked out, and (3) it's easy to use via R. After updating to the latest version of R, install JAGS for your operating platform here:

`http://sourceforge.net/projects/mcmc-jags/files/`

Click on 'JAGS', then the most recent folder, then the platform of your machine. With JAGS installed, open up R (or RStudio) and install the `coda` and `R2jags` packages.

```
require(coda)
require(R2jags)
```

4.1 The dataset

For data for this lab, we'll include a dataset on airquality in New York. We'll load the data and create a couple new variables for future use. For the majority of our models, we're going to treat 'Wind' as the response variable for our time series models.

```
data(airquality)
Wind = airquality$Wind # wind speed
```

```
Temp = airquality$Temp # air temperature
N = dim(airquality)[1] # number of data points
```

4.2 Linear regression with no covariates

We'll start with the simplest time series model possible: linear regression with only an intercept, so that the predicted values of all observations are the same. There are several ways we can write this equation. First, the predicted values can be written as $E[y_t] = u$. Assuming that the residuals are normally distributed, the model linking our predictions to observed data is written as

$$y_t = u + e_t, e_t \sim \mathrm{N}(0, \sigma^2) \tag{4.1}$$

An equivalent way to think about this model is that instead of the residuals as normally distributed with mean zero, we can think of the data y as being normally distributed with a mean of the intercept, and the same residual standard deviation:

$$y \sim \mathrm{N}(E[y_t], \sigma^2) \tag{4.2}$$

Remember that in linear regression models, the residual error is interpreted as independent and identically distributed observation error.

To run the JAGS model, we'll need to start by writing the model in JAGS notation. For our linear regression model, one way to construct the model is

```
###############################################################
# 1. LINEAR REGRESSION with no covariates
# no covariates, so intercept only. The parameters are
# mean 'mu' and precision/variance parameter 'tau.obs'
###############################################################
model.loc="lm_intercept.txt" # name of the txt file
jagsscript = cat("
model {
   # priors on parameters
   mu ~ dnorm(0, 0.01); # mean = 0, sd = 1/sqrt(0.01)
   tau.obs ~ dgamma(0.001,0.001); # This is inverse gamma
   sd.obs <- 1/sqrt(tau.obs); # sd is treated as derived parameter

    for(i in 1:N) {
      Y[i] ~ dnorm(mu, tau.obs);
   }
}
",file=model.loc)
```

A couple things to notice: JAGS is not vectorized so we need to use for loops (instead of matrix multiplication) and the `dnorm` notation means that we assume that value (on the left) is normally distributed around a particular mean with a particular precision (1 over the square root of the variance).

The model can briefly be summarized as follows: there are 2 parameters in the model (the mean and variance of the observation error). JAGS is a bit funny in that instead of giving a normal distribution the standard deviation or variance, you pass in the precision (1/variance), so our prior on 'mu' is pretty vague. The precision receives a gamma prior, which is equivalent to the variance receiving an inverse gamma prior (fairly common for standard Bayesian regression models). We will treat the standard deviation as derived (if we know the variance or precision, which we're estimating, we automatically know the standard deviation). Finally, we write a model for the data 'Y'. Again we use the 'dnorm' distribution to say that the data is normally distributed (equivalent to our likelihood).

The function from the `R2jags` package that we actually use to run the model is 'jags'. There's a parallel version of the function called 'jags.parallel' which is useful for larger, more complex models. The details of both can be found with:

```
?jags
```

or

```
?jags.parallel
```

To actually run the model, we need to create several new objects, representing (1) a list of data that we'll pass to JAGS, (2) a vector of parameters that we want to monitor in JAGS and have returned back to R, and (3) the name of our txt file that contains the JAGS model we wrote above. With those three things, we can call the 'jags' function, and

```
jags.data = list("Y"=Wind,"N"=N) # named list of inputs
jags.params=c("sd.obs","mu") # parameters to be monitored
mod_lm_intercept = jags(jags.data, parameters.to.save=jags.params,
                model.file=model.loc, n.chains = 3, n.burnin=5000,
                n.thin=1, n.iter=10000, DIC=TRUE)
```

Notice that the `jags()` function contains a number of other important arguments. In general, larger is better for all arguments: we want to run multiple MCMC chains (maybe 3 or more), and have a burn-in of at least 5000. The total number of samples after the burn-in period is n.iter-n.burnin, which in this case is 5000 samples. Because we're doing this with 3 MCMC chains, and the thinning rate = 1 (meaning we're saving every sample), we'll retain a total of 1500 posterior samples for each parameter.

The saved object storing our model diagnostics can be accessed directly, and includes some useful summary output,

```
mod_lm_intercept
```

```
Inference for Bugs model at "lm_intercept.txt", fit using jags,
 3 chains, each with 10000 iterations (first 5000 discarded)
 n.sims = 15000 iterations saved
         mu.vect sd.vect    2.5%     25%     50%     75%
mu         9.950    0.29   9.383   9.758   9.947  10.145
sd.obs     3.540    0.23   3.164   3.395   3.529   3.672
deviance 820.617    3.61 818.593 819.138 819.940 821.357
           97.5%  Rhat n.eff
mu        10.513 1.001 15000
sd.obs     3.967 1.001 15000
deviance 826.066 1.003 15000

For each parameter, n.eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor (at convergence, Rhat=1).

DIC info (using the rule, pD = var(deviance)/2)
pD = 6.5 and DIC = 827.1
DIC is an estimate of expected predictive error (lower deviance is better).
```

The last 2 columns in the summary contain `Rhat` (which we want to be close to 1.0), and `neff` (the effective sample size of each set of posterior draws). To examine the output more closely, we can pull all of the results directly into R,

```
attach.jags(mod_lm_intercept)
```

Attaching the R2jags object allows us to work with the named parameters directly in R. For example, we could make a histogram of the posterior distributions of the parameters `mu` and `sd.obs` with the following code,

```
# Now we can make plots of posterior values
par(mfrow = c(2,1))
hist(mu,40,col="grey",xlab="Mean",main="")
hist(sd.obs,40,col="grey",xlab=expression(sigma[obs]),main="")
```

Finally, we can run some useful diagnostics from the coda package on this model output. We've written a small function to make the creation of mcmc lists (an argument required for many of the diagnostics). The function

```
createMcmcList = function(jagsmodel) {
McmcArray = as.array(jagsmodel$BUGSoutput$sims.array)
McmcList = vector("list",length=dim(McmcArray)[2])
for(i in 1:length(McmcList)) McmcList[[i]] = as.mcmc(McmcArray[,i,])
McmcList = mcmc.list(McmcList)
return(McmcList)
}
```

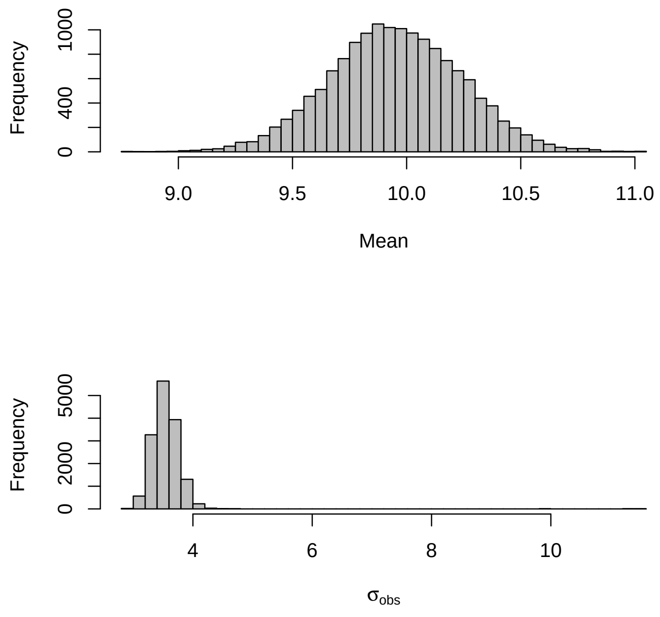

Fig. 4.1.

Creating the MCMC list preserves the random samples generated from each chain and allows you to extract the samples for a given parameter (such as mu) from any chain you want. To extract mu from the first chain, for example, you could use the following code. Because `createMcmcList` returns a list of mcmc objects, we can summarize and plot these directly. Figure 4.2 shows the plot from `plot(myList[[1]])`.

```
 myList = createMcmcList(mod_lm_intercept)
 summary(myList[[1]])
```

```
Iterations = 1:5000
Thinning interval = 1
Number of chains = 1
Sample size per chain = 5000

1. Empirical mean and standard deviation for each variable,
   plus standard error of the mean:

            Mean     SD Naive SE Time-series SE
deviance 820.640 3.8203 0.054027       0.055212
```

```
mu         9.954 0.2942 0.004160       0.004160
sd.obs     3.540 0.2325 0.003288       0.003288

2. Quantiles for each variable:

            2.5%     25%     50%    75%   97.5%
deviance 818.593 819.120 819.903 821.38 826.524
mu         9.374   9.763   9.949  10.15  10.535
sd.obs     3.163   3.397   3.528   3.67   3.958
```

```
 plot(myList[[1]])
```

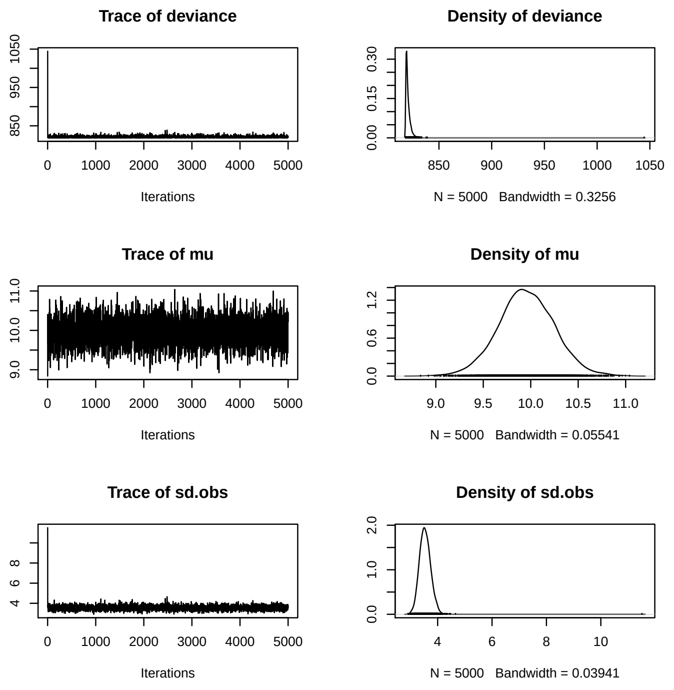

Fig. 4.2.

For more quantitative diagnostics of MCMC convergence, we can rely on the `coda` package in R. There are several useful statistics available, including the Gelman-Rubin diagnostic (for one or several chains), autocorrelation diagnostics (similar to the ACF you calculated above), the Geweke diagnostic, and Heidelberger-Welch test of stationarity.

```
# Run the majority of the diagnostics that CODA() offers
library(coda)
gelmanDiags = gelman.diag(createMcmcList(mod_lm_intercept),multivariate=F)
autocorDiags = autocorr.diag(createMcmcList(mod_lm_intercept))
gewekeDiags = geweke.diag(createMcmcList(mod_lm_intercept))
heidelDiags = heidel.diag(createMcmcList(mod_lm_intercept))
```

4.3 Regression with autocorrelated errors

In our first model, the errors were independent in time. We're going to modify this to model autocorrelated errors. Autocorrelated errors are widely used in ecology and other fields – for a greater discussion, see Morris and Doak (2002) Quantitative Conservation Biology. To make the deviations autocorrelated, we start by defining the deviation in the first time step, $e_1 = Y_1 - u$. The expectation of y_t in each time step is then written as

$$E[y_t] = u + \phi * e_{t-1} \tag{4.3}$$

In addition to affecting the expectation, the correlation parameter ϕ also affects the variance of the errors, so that

$$\sigma^2 = \psi^2 \left(1 - \phi^2\right) \tag{4.4}$$

Like in our first model, we assume that the data follows a normal likelihood (or equivalently that the residuals are normally distributed), $y_t = E[y_t] + e_t$, or $y_t \sim \mathrm{N}(E[y_t], \sigma^2)$. Thus, it is possible to express the subsequent deviations as $e_t = y_t - E[y_t]$, or equivalently as $e_t = y_t - u - \phi * e_{t-1}$. The JAGS script for this model is:

```
###############################################################
# 2. MODIFY THE ERRORS TO BE AUTOCORRELATED
# no covariates, so intercept only.
###############################################################
model.loc=("lmcor_intercept.txt")
jagsscript = cat("
model {
   # priors on parameters
   mu ~ dnorm(0, 0.01);
   tau.obs ~ dgamma(0.001,0.001);
   sd.obs <- 1/sqrt(tau.obs);
   phi ~ dunif(-1,1);
   tau.cor <- tau.obs * (1-phi*phi); # Var = sigma2 * (1-rho^2)

   epsilon[1] <- Y[1] - mu;
   predY[1] <- mu; # initial value
   for(i in 2:N) {
      predY[i] <- mu + phi * epsilon[i-1];
      Y[i] ~ dnorm(predY[i], tau.cor);
      epsilon[i] <- (Y[i] - mu) - phi*epsilon[i-1];
   }
}
",file=model.loc)
```

Notice several subtle changes from the simpler first model: (1) we're estimating the autocorrelation parameter ϕ, which is assigned a Uniform(-1, 1) prior, (2) we model the residual variance as a function of the autocorrelation, and (3) we allow the autocorrelation to affect the predicted values 'predY'. One other change we can make is to add 'predY' to the list of parameters we want returned to R (because this is a simple model, this is also easy to do in R given the other parameters).

```
jags.data = list("Y"=Wind,"N"=N)
jags.params=c("sd.obs","predY","mu","phi")
mod_lmcor_intercept = jags(jags.data, parameters.to.save=jags.params,
        model.file=model.loc, n.chains = 3, n.burnin=5000,
        n.thin=1, n.iter=10000, DIC=TRUE)
```

For some models, we may be interested in examining the posterior fits to data. You can make this plot yourself, but we've also put together a simple function whose arguments are one of our fitted models and the raw data. The function is:

```
plotModelOutput = function(jagsmodel, Y) {
# attach the model
attach.jags(jagsmodel)
x = seq(1,length(Y))
summaryPredictions = cbind(apply(predY,2,quantile,0.025), apply(predY,2,mean),
apply(predY,2,quantile,0.975))
plot(Y, col="white",ylim=c(min(c(Y,summaryPredictions)),max(c(Y,summaryPredictions))),
xlab="",ylab="95% CIs of predictions and data",main=paste("JAGS results:",
jagsmodel$model.file))
polygon(c(x,rev(x)), c(summaryPredictions[,1], rev(summaryPredictions[,3])),
col="grey70",border=NA)
lines(summaryPredictions[,2])
points(Y)
}
```

and we can use the function to plot the predicted posterior mean with 95% CIs, as well as the raw data. For example, try

```
plotModelOutput(mod_lmcor_intercept, Wind)
```

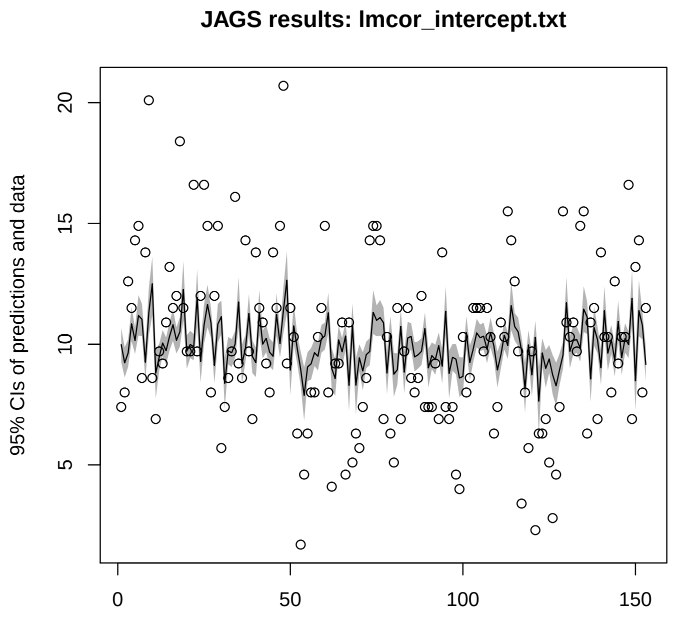

Fig. 4.3. Predicted posterior mean with 95% CIs

4.4 Random walk time series model

All of the previous three models can be interpreted as observation error models. Switching gears, we can alternatively model error in the state of nature, creating process error models. A simple process error model that many of you may have seen before is the random walk model. In this model, the assumption is that the true state of nature (or latent states) are measured perfectly. Thus, all uncertainty is originating from process variation (for ecological problems, this is often interpreted as environmental variation). For this simple model, we'll assume that our process of interest (in this case, daily wind speed) exhibits no daily trend, but behaves as a random walk.

$$E[y_t] = y_{t-1} + e_{t-1} \tag{4.5}$$

And the $e_t \sim \mathrm{N}(0, \sigma^2)$. Remember back to the autocorrelated model (or MA(1) models) that we assumed that the errors e_t followed a random walk. In contrast, the AR(1) model assumes that the errors are independent, but that the state of nature follows a random walk. The JAGS random walk model and R script to run it is below:

```
###############################################################
# 3. AR(1) MODEL WITH NO ESTIMATED AR COEFFICIENT = RANDOM WALK
# no covariates. The model is y[t] ~ Normal(y[n-1], sigma) for
# we'll call the precision tau.pro
# Note too that we have to define predY[1]
###############################################################
model.loc=("rw_intercept.txt")
jagsscript = cat("
model {
   mu ~ dnorm(0, 0.01);
   tau.pro ~ dgamma(0.001,0.001);
   sd.pro <- 1/sqrt(tau.pro);

   predY[1] <- mu; # initial value
   for(i in 2:N) {
      predY[i] <- Y[i-1];
      Y[i] ~ dnorm(predY[i], tau.pro);
   }
}
",file=model.loc)
jags.data = list("Y"=Wind,"N"=N)
jags.params=c("sd.pro","predY","mu")
mod_rw_intercept = jags(jags.data, parameters.to.save=jags.params, model.file=model.loc,
n.chains = 3, n.burnin=5000, n.thin=1, n.iter=10000, DIC=TRUE)
```

4.5 Autoregressive AR(1) time series models

A variation of the random walk model described previously is the autoregressive time series model of order 1, AR(1). This model introduces a coefficient, which we'll call ϕ. The parameter ϕ controls the degree to which the random walk reverts to the mean – when $\phi = 1$, the model is identical to the random walk, but at smaller values, the model will revert back to the mean (which in this case is zero). Also, ϕ can take on negative values, which we'll discuss more in future lectures. The math to describe the AR(1) time series model is:

$$E[y_t] = \phi * y_{t-1} + e_{t-1} \tag{4.6}$$

The JAGS random walk model and R script to run the AR(1) model is below:

```
###############################################################
# 4. AR(1) MODEL WITH AND ESTIMATED AR COEFFICIENT
# We're introducting a new AR coefficient 'phi', so the model is
# y[t] ~ N(mu + phi*y[n-1], sigma^2)
###############################################################
```

```
model.loc=("ar1_intercept.txt")
jagsscript = cat("
model {
   mu ~ dnorm(0, 0.01);
   tau.pro ~ dgamma(0.001,0.001);
   sd.pro <- 1/sqrt(tau.pro);
   phi ~ dnorm(0, 1);

   predY[1] <- Y[1];
   for(i in 2:N) {
      predY[i] <- mu + phi * Y[i-1];
      Y[i] ~ dnorm(predY[i], tau.pro);
   }
}
",file=model.loc)
jags.data = list("Y"=Wind,"N"=N)
jags.params=c("sd.pro","predY","mu","phi")
mod_ar1_intercept = jags(jags.data, parameters.to.save=jags.params,
        model.file=model.loc, n.chains = 3, n.burnin=5000, n.thin=1,
        n.iter=10000, DIC=TRUE)
```

4.6 Univariate state space model

At this point, we've fit models with observation or process error, but we haven't tried to estimate both simultaneously. We will do so here, and introduce some new notation to describe the process model and observation model. We use the notation x_t to denote the latent state or state of nature (which is unobserved) at time t and Y_t to denote the observed data. For introductory purposes, we'll make the process model autoregressive (similar to our AR(1) model),

$$x_t = \phi * x_{t-1} + e_{t-1}; e_{t-1} \sim \mathrm{N}(0, q) \tag{4.7}$$

For the process model, there are a number of ways to parameterize the first 'state', and we'll talk about this more in the class, but for the sake of this model, we'll place a vague weakly informative prior on x_1, $x_1 \sim \mathrm{N}(0, 0.01)$.Second, we need to construct an observation model linking the estimate unseen states of nature x_t to the data y_t. For simplicitly, we'll assume that the observation errors are indepdendent and identically distributed, with no observation component. Mathematically, this model is

$$y_t \sim \mathrm{N}(x_t, r) \tag{4.8}$$

In the two above models, q is the process variance and r is the observation error variance. The JAGS code will use the standard deviation (square root) of these. The code to produce and fit this model is below:

```
###############################################################
# 5. MAKE THE SS MODEL a univariate random walk
# no covariates.
###############################################################
model.loc=("ss_model.txt")
jagsscript = cat("
model {
   # priors on parameters
   mu ~ dnorm(0, 0.01);
   tau.pro ~ dgamma(0.001,0.001);
   sd.q <- 1/sqrt(tau.pro);
   tau.obs ~ dgamma(0.001,0.001);
   sd.r <- 1/sqrt(tau.obs);
   phi ~ dnorm(0,1);

   X[1] <- mu;
   predY[1] <- X[1];
   Y[1] ~ dnorm(X[1], tau.obs);

   for(i in 2:N) {
      predX[i] <- phi*X[i-1];
      X[i] ~ dnorm(predX[i],tau.pro); # Process variation
      predY[i] <- X[i];
      Y[i] ~ dnorm(X[i], tau.obs); # Observation variation
   }
}
",file=model.loc)
jags.data = list("Y"=Wind,"N"=N)
jags.params=c("sd.q","sd.r","predY","mu")
mod_ss = jags(jags.data, parameters.to.save=jags.params, model.file=model.loc, n.chains = 
n.burnin=5000, n.thin=1, n.iter=10000, DIC=TRUE)
```

4.6.1 Including covariates

Returning to the first example of regression with the intercept only, we'll introduce 'Temp' as the covariate explaining our response variable 'Wind'. Note that to include the covariate, we (1) modify the JAGS script to include a new coefficient – in this case 'beta', (2) update the predictive equation to include the effects of the new covariate, and (3) we include the new covariate in our named data list.

```
###############################################################
# 6. Include some covariates in a linear regression
# Use temperature as a predictor of wind
###############################################################
```

```
model.loc=("lm.txt")
jagsscript = cat("
model {
   mu ~ dnorm(0, 0.01);
   beta ~ dnorm(0,0.01);
   tau.obs ~ dgamma(0.001,0.001);
   sd.obs <- 1/sqrt(tau.obs);

   for(i in 1:N) {
      predY[i] <- mu + C[i]*beta;
      Y[i] ~ dnorm(predY[i], tau.obs);
   }
}
",file=model.loc)
jags.data = list("Y"=Wind,"N"=N,"C"=Temp)
jags.params=c("sd.obs","predY","mu","beta")
mod_lm = jags(jags.data, parameters.to.save=jags.params,
        model.file=model.loc, n.chains = 3, n.burnin=5000,
        n.thin=1, n.iter=10000, DIC=TRUE)
```

4.7 Forecasting with JAGS models

There are a number of different approaches to using Bayesian time series models to perform forecasting. One approach might be to fit a model, and use those posterior distributions to forecast as a secondary step (say within R). A more streamlined approach is to do this within the JAGS code itself. We can take advantage of the fact that JAGS allows you to include NAs in the response variable (but never in the predictors). Let's use the same Wind dataset, and the univariate state-space model described above to forecast three time steps into the future. We can do this by including 3 more NAs in the dataset, and incrementing the variable "N" by 3.

```
jags.data = list("Y"=c(Wind,NA,NA,NA),"N"=(N+3))
jags.params=c("sd.q","sd.r","predY","mu")
model.loc=("ss_model.txt")
mod_ss_forecast = jags(jags.data, parameters.to.save=jags.params,
      model.file=model.loc, n.chains = 3, n.burnin=5000, n.thin=1,
      n.iter=10000, DIC=TRUE)
```

We can inspect the fitted model object, and see that 'predY' contains the 3 new predictions for the forecasts from this model.

Problems

4.1 Fit the intercept only model from section 4.2. Set the burn-in to 3, and when the model completes, plot the time series of the parameter ‘mu’ for the first MCMC chain.
  a) Based on your visual inspection, has the MCMC chain convered?
  b) What is the ACF of the first MCMC chain?

4.2 Increase the MCMC burn-in for the model in question 1 to a value that you think is reasonable. After the model has converged, calculate the Gelman-Rubin diagnostic for the fitted model object.

4.3 Compare the results of the `plotModelOutput()` function for the intercept only model from section 4.2. You will to add ”predY” to your JAGS model and to the list of parameters to monitor, and re-run the model.

4.4 Modify the random walk model without drift from section 4.4 to a random walk model with drift. The equation for this model is

$$E[y_t] = y_{t-1} + u + e_{t-1} \tag{4.9}$$

where u is interpreted as the average daily trend in wind speed. What might be a reasonable prior on u?

4.5 Plot the posterior distribution of ϕ for the AR(1) model in section 4.5. Can this parameter be well estimated for this dataset?

4.6 Plot the posteriors for the process and observation variances (not standard deviation) for the univariate state-space model in section 4.6. Which is larger for this dataset?

4.7 Add the effect of temperature to the AR(1) model in section 4.5. Plot the posterior for ’beta’ and compare to the posterior for ’beta’ from the model in section 4.6.1.

4.8 Plot the fitted values from the model in section 4.7, including the forecasts, with the 95% credible intervals for each data point.

4.9* The following is a real life dataset from the Upper Skagit River (Puget Sound, 1952-2005) on salmon spawners and recruits:

```
Spawners = c(2662,1806,1707,1339,1686,2220,3121,5028,9263,4567,
             1850,3353,2836,3961,4624,3262,3898,3039,5966,5931,
             7346,4911,3116,3185,5590,2485,2987,3829,4921,2348,
             1932,3151,2306,1686,4584,2635,2339,1454,3705,1510,
             1331,942,884,666,1521,409,2388,1043,3262,2606,4866,
             1161,3070,3320)
Recruits = c(12741,15618,23675,37710,62260,32725,8659,28101,17054,
                 29885,33047,20059,35192,11006,48154,35829,46231,
                 32405,20782,21340,58392,21553,27528,28246,35163,
                 15419,16276,32946,11075,16909,22359,8022,16445,2912,
                 17642,2929,7554,3047,3488,577,4511,1478,3283,1633,
                 8536,7019,3947,2789,4606,3545,4421,1289,6416,3647)
logRS = log(Recruits/Spawners)
```

Fit the following Ricker model to these data using the following linear form of this model with normally distributed errors:

$$log(R_t/S_t) = a + b * S_t + e_t, \text{ where } e_t \sim \mathrm{N}(0, \sigma^2) \tag{4.10}$$

You will recognize that this form is exactly the same as linear regression, with independent errors (very similar to the intercept only model of Wind we fit in section 4.2).

Within the constraints of the Ricker model, think about how you might want to treat the errors. The basic model described above has independent errors that are not correlated in time. Approaches to analyzing this dataset might involve

- modeling the errors as independent (as described above)
- modeling the errors as autocorrelated
- fitting a state-space model, with independent or correlated process errors

Fit each of these models, and compare their performance (either using their predictive ability, or forecasting ability).