Installation:

- download and install JAGS. **You must have JAGS installed prior to installing the rjags package**.

  If you have admin privileges, you can get JAGS from sourceforge. The download link is at: `https://sourceforge.net/projects/mcmc-jags/files/`. Look for the green download latest version button. Watch out and avoid the big green download now button in the ad at the top of the screen.

  If you do not have admin privileges (or you don't want to mess with that), you can get JAGS from the ISU Software Center. Run the Software Center program and search for JAGS.

- In R, install the rjags package. This is on cran, so install.packages() works as usual. The installation will look for JAGS in your system files and link to it automatically.

Overview:

JAGS is the second of three widely used Bayesian analysis programs and one newcomer. It is always possible to do all the statistical computations to implement an analysis in R (or Python, C, or Fortran). JAGS and its kin allow you to specify a model in terms of computations and distributions. This code is reasonably intuitive once you get the hang of it. The software then determines the most appropriate way to implement that analysis. The first program was BUGS (Unix), with follow-ons WinBUGS (for Windows, with a graphical user interface) and OpenBUGS (open source). Those programs defined the BUGS language, used to specify computations and distributions. The most recent programs are Stan, which provides much faster computations but requires that the model be specified by more complicated code based on the C language, and NIMBLE, which was developed for ecological applications.

Aside: My favorite way to use Stan is by rstanarm (Stan applied regression modeling) that provides an R model interface to Stan. rstanarm provides Bayesian alternatives to many R modeling functions (lm, glm, lmer, glmer, and others). The restriction is that you are restricted to models that can be specified as a formula and can not customize the model.

The implementation of rjags is similar to that of RMark. JAGS is a stand-alone program that does all the computing. The R functions connect R to JAGS, so you can use R for manipulation of input data and results.

R2jags, runjags, and jagsUI are alternatives to the rjags library. All do similar things but function names and output formats are different. I don't know which is easier to use. In particular, jagsUI is "an easier to use" front-end to rjags; rjags then calls JAGS. My understanding is that R2jags and runjags are front-ends directly to JAGS.

Running a Bayesian model using JAGS requires two separate blocks of code.

1. You need to describe your model using the BUGS language. This code specifies the process model, the relationships between the observable data and quantities in the process model, and all prior distributions for unobservable quantities. This goes into a separate text file, usually with a .bug file extension.
2. You need to write R code that provides information to JAGS. JAGS needs to know the file name with your model, a list of all the data, and values of various controls. The jags.model() function assembles all this. The update() and coda.samples() functions run JAGS and returns the output in a form interpretable by R. Other functions in the rjags library summarize, plot, and manipulate the JAGS output.

I illustrate the use of JAGS and rjags by example. The first example provides detailed explanations of the steps and the JAGS code. The other examples focus on JAGS coding of some useful models. We will fit four models to the post-1980 Yellowstone Grizzly bear counts. These models are:

- Model 0: a linear regression of log count on year, observational error only
- Model 0b: a state-space version of that linear regression, observational error only
- Model 0c: the state-space version with a Poisson observation error model
- Model 1: a state-space model with both process error and observational error.

Running JAGS using rjags functions:

The first example fits Model 0, a log-linear regression model: $\log Y_t = \beta_0 + \beta_1(t - 1980) + \varepsilon$, with $\varepsilon \sim \mathrm{N}(0, \sigma^2)$. Y_t is the count in year t. 1980 is subtracted from t so that the intercept, β_0 is the expected log count in 1980. This also avoids severe numerical issues when the intercept is the count in year 0.

The BUGS language version of this model is in lreg.bug. The file grizz0.r creates lreg.bug then fits this model. I show how to fit this model, then discuss the structure of the code.

The steps are:

1. Write the desired model in the BUGS language and store it in a file. I use the .bug extension but .txt is fine. I've already written that model in the lreg.bug file.
    - The BUGS language is a way to describe a statistical model relating the data to parameters and their prior distributions. It was developed for the BUGS/WinBUGS software. JAGS uses the same language with a few differences.

- Easiest to edit the file using a programmer's text editor. Notepad++ is a really good, free, text editor. Notepad++ is approved for use at ISU and is in SoftwareCenter. It's also easy to use the R file editor so long as you remember two things when you save the file. 1) Change the "Save as type" to all files and 2) include the file extension (.bug or .txt) with the filename when you save the file (you don't want the default .r extension for the bug file). You can also use WORD, but you need to make sure to save the file as a text file.

2. In R, store information about the model, the data, and jags control variables in a jags object. This is done by the jags.model() function.

   The required arguments to jags.model() are the name of the file containing the model code and a list containing all the data you want to provide to the model code. The names of the list elements must match the variable names in the model code. You can pass scalars (e.g. n), vectors, or matrices. Optional arguments are the number of MCMC chains (n.chains=) and the number of steps done by jags to tune various internal options (n.adapt =). I find n.adapt=100 usually sufficient; if a model has trouble converging, increasing n.adapt may help.

3. Use update() to iterate the Markov Chain quite a few times. This is the burn-in to move from arbitrary initial values to (we hope) the stationary distribution that we are interested in. You want to repeat this step until the chain has converged to a stationary distribution. I usually use multiple chains (e.g. 3 or more) to check convergence. Convergence is checked in step 6.

4. Use coda.samples() to iterate the Markov Chain quite a few more times, saving samples of all desired variables. I use the coda.samples() function; the jags.samples() does the same thing but returns output in a less usable format (at least for the way I work).

5. My experience is that hierarchical models often don't mix well, so there is a considerable autocorrelation in the chain. I usually sample a long chain that is heavily thinned (e.g. thin=10) to reduce the amount of data that is stored.

6. Check that the chain has converged using trace plots and/or the Gelman-Rubin statistic (or some other diagnostic). plot() draws trace plots, gelman.diag() computes the GR statistic for each variable individually and the multivariate version for all variables simultaneously. Overlapping trace plots and GR statistics close to 1 are evidence of convergence. There is no universal definition of "not converged", but GR > 1.1 is usually considered too bad to use. I have seen > 1.2 suggested.

7. If the MCMC sampler hasn't converged, use update() then coda.samples() again.

8. Extract means, medians, and other summary statistics from the samples.

Exercise: Download and save in an R working directory the grizzly.csv data file, the grizz0.r R code and the lreg.bug BUGS code. Run the grizz0.r code and look at the various bits of output. Ask if you don't understand something or want something additional.

Explanation of the BUGS code in lreg.bug

The BUGS language provides a way to write models that describe the relationships between what is observed and what is to be inferred. Some things to note:

1. Statements can be in any order. You are describing relationships, not the sequence of computations.
2. I almost always organize the code in three sections: the process model, the observation model(s), and the prior distributions().
3. # starts a comment, just as in R.
4. Assignments are done by `<-`; random variables are defined by `~`.
5. The arguments for definitions of random variables must be variables. No computations allowed. I have seen online examples of computations in a random variable definition, but they never seem to work for me (illegal redefinition of a node errors). Safer to define the mean separately. Remember (previous point), this probably needs to be a vector, so is part of the data model loop.
   - `y[i] ~ dpois(exp(b0 + b1*x[i]))` is not allowed.
   - You have to compute the mean (λ) separately from defining the data distribution:
     ```
     my[i] <- exp(b0 + b1*x[i])
     y[i] ~ dpois(my[i])
     ```
6. Variables for intermediate computations can sometimes be reused. I have seen code where a distribution is specified as: `dpois(x[i]*n[i])`. I do know that attempting to specify `dpois(exp(b0 + b1*x[i]))` will fail. My practice is to explicit calculate each intermediate computation then use those computed values in a distribution. This is illustrated by the above code, where `my[i]` is the mean for each observation. This is computed on one line, then used as the parameter for the Poisson distribution on the next line.
   - In R:
     ```
     for (i in 1:n) {
       temp <- exp(l[i] + b[i])
       p[i] <- dpois(temp)  }
     ```
     is just fine because statements are sequentially executed
   - In BUGS: not allowed, because the statements define relationships. You need to define a separate mean for each value of i. A vector is the usual approach:
     ```
     for (i in 1:n) {
       temp[i] <- exp(l[i] + b[i])
       p[i] <- dpois(temp[i])  }
     ```

   The elements of temp are the expected count for each observation.
7. All random variables must either be data or given prior distributions.
8. The parameterization of random variables in BUGS is not always the same as in R. Some things that I need to be careful about:

- In BUGS, Normal distributions are specified as (mean, precision), where precision = 1 / variance. That means a vague prior for a normal like N(0, 1000) is specified as dnorm(0, 0.001)
- Gamma distributions are specified as (shape, rate), so the mean is shape/rate and the variance is shape/rate^2. That means a gamma with mean 1 and variance 1000 is dgamma(0.001, 0.001). A lognormal random variable probably wants the variance prior to be more like mean = 0.1, variance = 1, which is dgamma(0.01, 0.1). But, see below for other choices for variance priors.

9. Choices of prior distribution

- The usual choice of diffuse prior for a mean or regression coefficient is N(0, σ^2) with large σ^2. Because the BUGS language specification of a normal distribution is dnorm(mu, precision), you want a small precision. How small depends on the magnitude of the values. If the parameter is the mean of values around 1000, you expect a large sd (e.g., 100 or more), so use small precision (0.000001). If the parameter is a regression coefficient for which 0.1 is a huge value, use a much larger precision (e.g., 10).
- The conjugate prior for a variance of a normal distribution is the inverse gamma, so the conjugate prior for a precision (1/variance) is the gamma distribution. A common recommendation is dgamma(0.001, 0.001), which gives a mean variance of 1 and a variance of the variance of 1000.
- dgamma(0.001, 0.001) is almost always too variable (for either process or observation variance) when modeling log transformed observations. A variance of 1 corresponds approximately to a 100% cv (sd = mean). A variance of 0.25 (sd = 0.5) or 0.04 (sd = 0.2) is more reasonable.
- When there are multiple sources of random variation, some care is needed with the choice of prior distributions for the "upper-level" variances. A common choice is a uniform prior for the standard deviation. This issue is relevant to the process variance in Model 1 and is discussed in more detail there.
- It is always good practice to evaluate a few different choices of prior distribution. This is called "prior sensitivity analysis".

Model 0b: The linear regression model can be written in state-space form as:

$$\begin{aligned} N_t &= N_{t-1} + r && \text{the process model, no process error, acting on the log scale} \\ \log Y_t &\sim N(N_t, \sigma^2_{obs}) && \text{the observation model} \end{aligned}$$

If you work backwards from N_t to N_0, you find that the state-space model is exactly the same as the linear regression model because $N_t = N_0 + rt$, so β_0 in the linear regression is N_0 in the state-space model and β_1 in the linear regression is r in the state-space model.

This model has 3 parameters: r, σ^2_{obs} and N_0. We need to provide prior distributions for each one.

The bugs and r code to fit this model are in lregb.bug and grizz0b.r. The r code is almost exactly the same as that in grizz0.r Make sure you understand how lregb.bug works.

What is the 95% credible interval for r?

Model 1: The state-space model with process error:

$$\begin{aligned} N_t &= N_{t-1} + r + \tau_t && \text{the process model, with process error} \\ \tau_t &\sim N(0, \sigma^2_{process}) && \\ \log Y_t &\sim N(N_t, \sigma^2_{obs}) && \text{the observation model} \end{aligned}$$

The bugs and r code to fit this model are in exp.bug and grizz1.r. The first part of the r code is almost exactly the same as that in grizz0.r. One difference is that we also ask jags to return the posterior distributions for each N_t. That way we can easily plot the distribution of $\hat{N}(t)$ over time.

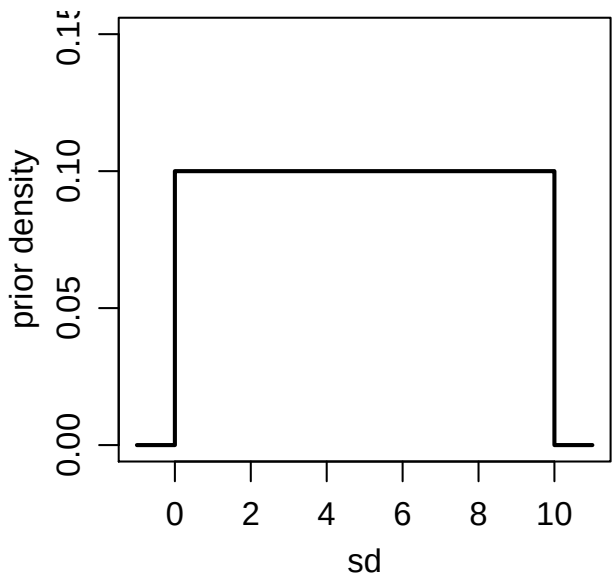
What is the 95% credible interval for r?

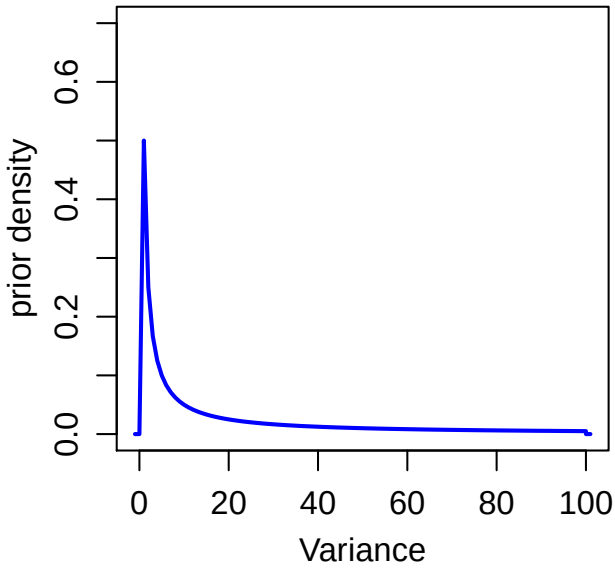
Additional details, relevant to some models:

- Choice of prior distribution for $\sigma^2_{process}$. This in an "upper level" variance because it is "above" the observation-specific variance. There are two issues with upper-level variances: The upper-level variance (or variances) could well be 0 or close to 0. The observation-specific variance is almost never zero. And, there is usually less information about upper-level variances. Think of a mixed model for data with two sources of variability, between forest stands and between plots within a stand. When you sample 5 forest stands and 6 plots per stand, you have 25 df of information for the plot variance but only 4 df for the stand variance.

  The inverse gamma prior distribution for a variance has relatively little probability close to zero, so when the upper level variance really is small, the prior pulls the estimates up more than it should. And, because there is usually less information about upper-level variances, the choice of prior can have considerable influence.

  There are various solutions. Gelman's paper (Gelman 2006, Bayesian Analysis 3:515-533) describes them. The simple recommended solution is $\sigma \sim U(0, c)$, i.e. the sd has a uniform distribution between 0 and c. In terms of the variance, that prior puts a lot more probability on values close to 0. This would be followed with $\tau = \text{pow}(-2)$ to compute $1/sd^2$, and $Y \sim N(\mu, \tau)$. If you prefer a half-Cauchy prior, please use that.





How big should c be? Depends on the likely values of the variance. Note that variances larger than c^2 are forbidden by the prior. My practice is to guess the likely sd and set c to 10 times that likely sd. For quantities like population growth rates, my experience is that c of 1 or 0.1 is often sufficient.

There is a data based check whether c was too small. Look at the posterior density of the sd. If all the probability is piled up at the boundary, i.e., near c, you set c too small.

If a state-space model has trouble converging, c for the process variance is probably too large.

- Specifying an overdispersed distribution for count data

  When you specify a Poisson distribution for an observed count, the variance of that count is forced to equal the expected (mean) count. My experience with ecological counts is that they are usually overdispersed (variance > mean). I know two ways to include overdispersion in a model:

  My favorite way is to include an observation-specific lognormal random effect. The negative binomial distribution is the convolution of a Poisson with a Gamma distribution. The second approach replaces the Gamma with a lognormal. Those are very similar distributions, but the log normal fits better in models with other lognormal terms. One way to do this is, starting with the log-scale mean for observation i in logm[i]:

```
eta[i] ~ dnorm(logm[i], taueta)
mu[i] <- exp(eta[i])
y[i] ~ dpois(mu[i])
```

  The alternative is to use a negative binomial distribution as the observation model. JAGS provides one version of the negative binomial, dnegbin(p,r). Unfortunately, this is # failures until the r'th success parameterization, not the ecological parameterization (in terms of mean, μ, and overdispersion, ϕ). The expected value for JAGS's negbin is $\mu = r * (1 - p)/p$ and the variance is μ/p. Various folks have experienced poor mixing for this distribution, and for ecological models you want to put structure into μ, not p.

- Specifying models with factor variables or lots of regression variables These would be included in the computation of the mean or log mean.

  Fixed effects influencing the log mean of the Poisson are included in the specification of `mu[i]`. The model above has one regression covariate. Multiple regression covariates are handled in the obvious way.

  If you have a factor variable, you need to create and use the indicator variables associated with that factor. Either create them by hand in R before calling jags.model(), or fit a lm or glm to the data and extract the model.matrix(). When you have lots of covariates (either continuous variables or created indicator variables), it is often easier to E.g., for a factor variable called habitat, with 4 levels:

  ```
  dummy <- glm(y ~ habitat, family=poisson, data=something)
  ```

  This will include the indicator variables corresponding to how R expands factors. The default is "set first level to 0" coding, but that can be changed by specifying a different contrast type.

- Specifying a growth rate that varies over time
  Replace r, a constant, with $r[i]$ and allow $r[i]$ to follow a random walk over time. That means that

$$\begin{aligned} N_t &= N_{t-1} + r_t + \tau_t \\ r_t &= r_{t-1} + \nu_t \end{aligned}$$

where τ_t is the process error and ν_t is the random change in the growth rate. When Var $\nu_t = 0$, this model is the exponential growth + process error model with constant growth rate.

For practice, here are some extensions (all optional):

After fitting the exponential growth model with process error and being (at least somewhat) comfortable with the output, feel free to modify the code (grizz1.r, exp.bug, or both) so that:

- Observations conditional on the latent population size have a Poisson distribution
  The BUGS specification for a Poisson distribution is dpois(mean), where mean is the expected value for that observation.
- Observations conditional on the latent population size have an overdispersed Poisson distribution.
- The process model allows the population growth rate to vary smoothly over time, i.e., a local linear trend model.
- The process model allows the population growth rate to depend on current population size, i.e., a Ricker density-dependence model or something like it.