Intermediate Bayesian Analyses with JAGS

For Fish and Wildlife Professionals

Lead Instructor
Ben Staton, Quantitative Fisheries Scientist
Columbia River Inter-Tribal Fish Commission

Co-Instructor
Henry Hershey, Graduate Research Assistant
Auburn University School of Fisheries, Aquaculture, and Aquatic Sciences

Meeting
2019 Joint Meeting of the American Fisheries Society and The Wildlife Society

Workshop Date/Time
September 29, 2019; 8:00am – 5:00pm

Workshop Location
Reno-Sparks Convention Center (4590 S Virginia St, Reno, NV), Room A17

Table of Contents

Part I

BACKGROUND MATERIAL

Workshop Overview

Content

This workshop will extend the material covered in day one ("Introductory to Bayesian Analyses with JAGS") to equip participants with the necessary knowledge to implement the JAGS software for fitting more complex models used in fisheries and wildlife analysis problems[1]. As in day one, content will focus on applied aspects, but some review of the theory will also be required. Topics to be covered include:

- Review of the generalized linear modeling framework
- Use and interpretation of random effects
- Evaluation of model adequacy
- Introduction to state-space models

These topics will be explored using the JAGS software to investigate basic fisheries and wildlife analysis problems including:

- Logistic, Poisson, negative binomial, and zero-inflated Poisson regression
- Posterior predictive checks for model adequacy
- Logistic regression with random slopes and intercepts
- State-space expression of the Cormack-Jolly-Seber model

Instructional Methods

Workshop instruction will be split between lecture material and hands-on activities, though far more time will be spent working with JAGS than in day one (see the schedule). Participants will have the opportunity to ask questions and explore each of the analyses on their own before moving on to the next analysis. Analyses will begin simple and become progressively more complex throughout the afternoon. The workshop will conclude with the participants adapting the Cormack-Jolly-Seber model to investigate a question of their own choosing. Interested participants may stay late to see more state-space models, including a biomass dynamics model and a Pacific salmon spawner-recruit model.

Materials

Participants will need to provide their own laptop (either PC or Mac) to fully participate. **All necessary software should be installed before coming to the workshop** (see the installation instructions; all required software is identical to day one except for a handful of optional R packages). This will ensure valuable workshop time is not spent on topics other than

[1] Although the example data come from the fields of fisheries science and management, the subject matter is highly relevant to practitioners of wildlife science and management as well.

course material. Participants will receive a zipped file containing all workshop materials including the slides, code templates, and data for each exercise. Upon receiving this zipped file, please create a folder called `AFS19_2` in a location analogous to `C:/Users/usr/Documents` and unzip all files to that folder. This will ensure that all participants and instructors know where to find the files used in the workshop.

Prerequisites

Given the intermediate nature of this workshop, we will assume participants have a working knowledge of the BUGS language and the workflow of calling JAGS through R (commensurate with the day one material). It is assumed participants have statistical understanding commensurate with an introductory undergraduate-level theoretical or applied statistics course. We expect that participants are at least vaguely familiar with probability distributions and their usage in likelihood functions. Experience with the generalized linear model would also be useful, though we will review it prior to fitting any models.

This workshop will be taught by calling JAGS through Program R. We will assume no prior participant knowledge of the R language and all participants will be provided with completed code (with the exception of the BUGS code for the model definition). However, some prior knowledge of R would be helpful. Participants with no R experience may wish to familiarize themselves with some basic material by completing sections 1-4 in this free tutorial.

About the Instructors

Ben Staton is a Quantitative Fisheries Scientist with the Columbia River Inter-Tribal Fisheries Commission, and is based out of Portland, OR. He received a BSc in 2013 from Michigan State University, and MSc and PhD degrees from Auburn University in 2015 and 2019, respectively. Ben's graduate research focused on the development and evaluation of quantitative tools for informing harvest management strategies and policies for Pacific salmon fisheries in western Alaska, where he spent much of his time between 2014 and 2019. Bayesian inferential methods are a cornerstone of Ben's research, primarily because of their flexibility and applicability to state-space population dynamics models. A subset of Ben's work can be found on GitHub and Google Scholar.

Henry Hershey is a Graduate Research Assistant pursuing a PhD in Fisheries Science at Auburn University. He received a BA in biology from Case Western Reserve University in 2016, and completed a MSc in Fisheries Science at Auburn University in 2019. His thesis described the behaviors, movements, and physiology of large migratory fishes as they relate to dam passage. His PhD work is focused on using new biotelemetry techniques to study the energetics of migrating fish. Henry is also an avid fly fisherman and a published illustrator with artwork featured in blogs, magazines, books, and peer reviewed journal articles. Henry can be found on Twitter (@spoonbill_hank) and Instagram (@hanks_fish_art).

Workshop Timeline

Time[1]	Topic
8:00	Begin
8:00 – 8:20	Welcome and Introductions
8:20 – 9:00	Review of the generalized linear model
9:00 – 9:45	*Exercise 2.1 (Logistic regression)*
9:45 – 10:00	**Break**
10:00 – 10:45	*Exercise 2.2 (Poisson regression)*
10:45 – 12:00	*Exercise 2.3 (PP checks and negative binomial regression)*
12:00 – 1:00	**Lunch**
1:00 – 1:45	*Exercise 2.4 (Zero-inflated Poisson regression)*
1:45 – 2:00	Review of random/mixed-effects models
2:00 – 2:45	*Exercise 2.5 (Mixed-effect logistic regression)*
2:45 – 3:00	**Break**
3:00 – 3:30	Intro to state-space models and CJS
3:30 – 5:00	*Exercise 2.6 (CJS models) and 2.7 (independent)*
5:00	Adjourn
5:00 – 6:00	Optional: more state-space models

[1] Times listed are approximate. Some topics may require more or less time depending on participant interest.

Installation Instructions

NOTE: all software requirements are identical to those of day one, with the exception of several optional R packages, which we will use to illustrate how to fit some of the models in R without JAGS.

R

R is a statistical programming language and environment commonly used by academic and applied researchers in our fields. In the workshop, R will be used to prepare the data, call JAGS, and process the output.

If you do not already have it, please install R on your laptop. Any R versions later than 3.0.0 should be adequate, but you can install the latest version if you wish.

R can be downloaded here. Select the appropriate link for your operating system and download and run the executable file (all default settings are adequate).

RStudio

RStudio is an integrated development environment for R, and has several nice features that will be useful in this workshop. If you are a seasoned R user and prefer to use another IDE or the standard R interface, feel free. The instructor will use RStudio when demonstrating code.

RStudio can be downloaded here. Select the appropriate link for your operating system and download and run the executable file (all default settings are adequate).

JAGS

JAGS is the MCMC engine we will use in the workshop (i.e., it is the program that does the actual Bayesian heavy-lifting).

JAGS can be downloaded here. The instructor will be using version 4.3.0, and you are recommended to use this version as well.

Mac users will have some extra steps to install JAGS (see the info at the bottom of that webpage) – if you run into problems, contact Henry Hershey, as he is a Mac user and has installed JAGS successfully.

Necessary R Packages

You will need to actively install at least two R packages. Perform the steps below only **after** you have installed at least R and JAGS. The first (`{jagsUI}`) is hosted on CRAN and can be installed via:

```
install.packages("jagsUI")
```

It is the personal opinion of one of the instructors that the built-in capabilities for dealing with output from these models are less than fantastic. So he wrote a package (`{postpack}`; source code and information here; see here for a brief tutorial) to facilitate post-processing. It can be installed via:

```
install.packages("devtools") # on CRAN, if you don't already have it
devtools::install_github("bstaton1/postpack")
```

This will install another of the author's packages (`{StatonMisc}`) as well, which we will use for the `ext_device()` function (which creates a new plotting device window, regardless of the operating system of the user). To verify everything has worked correctly, try loading these two packages:

```
library(jagsUI)
library(postpack)
```

If any errors pop up, then something didn't work correctly and you should uninstall R/RStudio and re-install the most current versions before trying again. You may get some warnings or messages, but so long as you don't see `Error in library(jagsUI) : there is no package called jagsUI` (or similar for the other packages), you're all set.

Optional R Packages

We will use several additional R packages to illustrate how to fit some of the models in R without JAGS. We will do this primarily to generate reasonable initial values for MCMC sampling, but in some cases we will compare estimates and predictions from Bayesian and non-Bayesian methods. The instructor will use these packages:

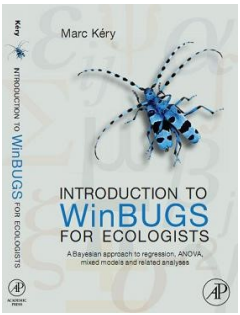
- `{MASS}` for the `glm.nb()` function, which fits a negative binomial regression
- `{pscl}` for the `zeroinfl()` function, which fits zero-inflated count models
- `{lme4}` for the `glmer()` function, which fits mixed-effect GLMs

All three packages may be found on CRAN, and can be installed via `install.packages("pkgName")`.

Supplemental Resources/Reading

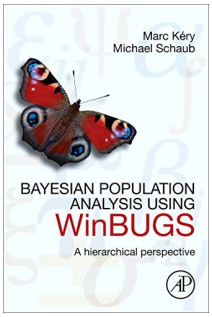
Books

Kéry (2010): Introduction to WinBUGS for Ecologists



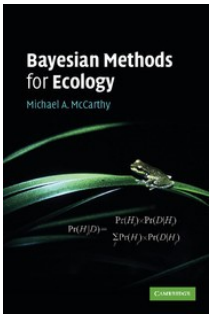
This is an excellent introductory text on Bayesian inference with R and BUGS. It provides concise and thorough descriptions of not only Bayesian inference, but the generalized linear model as well. It covers a wide variety of models and topics including both fixed and mixed effects models. In each of the chapters, readers simulate the data under the model to be used for inference, further ensuring a full understanding of the model they are fitting. Much of the instructors' understanding of the BUGS language came directly from this book.

Kéry and Schuab (2012): Bayesian Population Analysis using WinBUGS



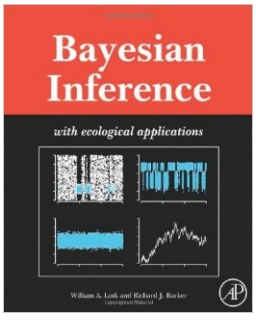
This is a follow-up to Kéry's first text, and it covers more advanced models in population analysis. Many of the models presented require more specialized data: namely capture-recapture data, oftentimes with multiple recapture periods. It covers methods for estimating detection probability and survival probability, population abundance, recruitment, and site-occupancy. It emphasizes the use of hierarchical models for population inference, based on data being produced by a both true (latent) underlying biological process and an observation process. It is more advanced than the first book, but still very readable.

McCarthy (2007): Bayesian Methods for Ecology



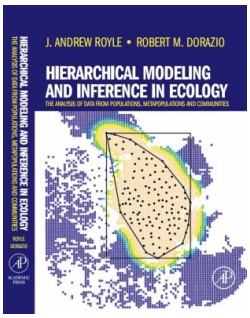
This is another great introductory text with relatively simple but diverse examples covering generalized linear models, mark-recapture, and analyses with subjective priors. Although found elsewhere, this text provides an excellent summary of primary probability distributions used in ecological modeling, with descriptions of conjugacy.

Link and Barker (2010): Bayesian Inference with Ecological Applications



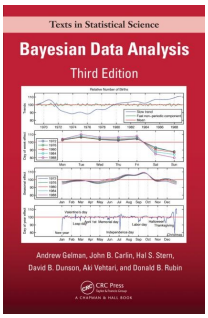
This is a somewhat more theoretical text than the books by Kéry and McCarthy, though still very readable. It gives thorough technical descriptions of Bayesian interpretations of probability and statistical inference and contrasts them with Frequentist interpretations. It also uses BUGS to implement Bayesian inference and includes an excellent chapter on Bayesian multimodel inference, which is not covered in either of the Kéry books listed above.

Royle and Dorazio (2008): Hierarchical Modeling and Inference in Ecology



This is a more complex text for readers interested in contemporary population modeling approaches including occupancy models, N-mixture models, individual-level random effects, and metapopulation models. The book includes many examples of complex hierarchical models that are beyond the scope of what will be covered in the workshop. These types of models are becoming increasingly applied to ecological problems so we thought it important to include in this list. We would recommend having a firm grasp on the material of the two Kéry books listed above before moving on to this text.

Gelman et al. (2014): Bayesian Data Analysis



This is another more complex and theoretical text, and has no ecological focus nor worked examples in the BUGS language (which is why we list it last). However, it is widely acknowledged as one of the foremost and comprehensive texts on Bayesian statistical inference, so we would be remiss exclude it from this list. In particular, if readers are interested in the details of the various MCMC algorithms, this is perhaps the best reference as the authors devote nearly 100 pages to the topic of computation in the Bayesian framework.

Journal Articles

Here is a (highly incomplete) list of articles that discuss the use of Bayesian methods in ecology or fisheries/wildlife management at a broad level rather than focusing on one particular application (click for full citation information):

Ellison (1996), Ludwig (1996), Punt and Hilborn (1997), Meyer and Millar (1999), Wade (2000), and King (2012).

Other Resources

In addition to these texts, there are a wide variety of other materials out there:

- Worked BUGS examples on the OpenBUGS website.
- The JAGS user manual has a ton of great information about using JAGS, but less about completed examples or implementing it through R
- There are many blogs, forums, and similar with information about JAGS and Bayesian inference in general available by web search.

Tips/Tricks Regarding BUGS Modeling

General Recommendations

- Always start with a simple model and make sure everything is working properly before building in complexity.
- If there is a missing value (`NA`) in a response variable, a value will be imputed automatically that represents the model prediction. If there is a missing value in a predictor variable, JAGS will crash. If this is the case, either exclude that observation from your model, or figure out a way to insert a value for that predictor.
- The more loops you have, the slower JAGS will run. It is generally advisable to minimize the number of loops you have in your model code.
- The BUGS language prohibits the same variable being on the left hand side of an operator twice in the same model. For example, a model containing both `mu ~ dnorm(0, 0.001)` and `mu <- a + b * x[i]` will crash.
- Always evaluate the fit of your model to the data. Examine residuals, plot the fitted curve over your data, and compare random predictions from the model to your data. MCMC sampling generally works well, but there are cases where it can give unexpected behavior.
- If possible and practicable, fit a maximum likelihood version of the same analysis and compare the estimates. If you are using vague priors, the results should be nearly identical. If the estimates largely differ, you may have specified the model incorrectly or perhaps a prior you thought was vague was actually not.
- The JAGS error messages are generally more informative than those generated by WinBUGS or OpenBUGS, but they are not always very helpful. If you get an error you don't recognize, check your model specification first, then data, then initial values.
- If at any point the model has to calculate `log(x)` and `x` is negative, JAGS will crash. Whenever taking logs, make sure it is impossible to have a negative value there.

Initial Values

- If you don't specify initial values for some nodes in your model, JAGS will generate random initial values from the prior you specify for that node. The sampler may crash if the data are highly inconsistent with the sampled initial value (this is a common cause of the `Invalid parent values` error).
- Simple models (i.e., simple regression models) will typically do fine without specifying initial values.
- More complex models may need initial values close to the posterior to get off the ground without crashing.
- If using a gamma prior on a precision parameter (e.g., `tau ~ dgamma(0.001, 0.001)`), it is highly recommended to use an initial value.
- Be sure to run at least two chains with fairly different initial values to help assess convergence.

- Initial values must not be outside the range of a prior (e.g., if `sig ~ dunif(0,1)`, then an initial value of 1.2 will cause a crash with an `Invalid parent value` error).

Priors

- Whenever possible, choose conjugate priors (e.g., a beta prior for the success probability parameter of a binomial distribution). This can help convergence.
- If you wish to be "ignorant" about a parameter *a priori*, pick diffuse but not excessively diffuse priors. If the data are weakly informative, this can slow convergence.
- Prior sensitivity should be investigated for key parameters. Try a couple different priors. If the results are sensitive to slight changes in the prior, this may not be a huge problem, but it is important to know. Further, it should be reported on.
- If you don't know what your prior looks like, visualize it. R is very helpful for this. For example, you can compare two priors for a probability parameter:

```
# different values of the random variable
x = seq(0,1,length = 100)

# density functions
prior1 = function(x) dbeta(x, 4, 2)
prior2 = function(x) dbeta(x, 2, 1)

# plot them
plot(prior1(x) ~ x, type = "l", col = "red")
lines(prior2(x) ~ x, col = "blue")
```

A Brief Tutorial for {postpack}

The ability to interface with JAGS from R is very well developed, and multiple packages exist for doing so. However, once JAGS finishes running, the user is left with an object of class `mcmc.list`, which are not the most intuitive to work with. In particular, subsetting, summarizing, plotting, and reporting posterior information from **desired nodes** is highly cumbersome with the built-in workflow (to our knowledge, and in our opinion). Thus, one of the instructors developed `{postpack}` to aid in performing these tasks.

For this tutorial, we will use the output from Exercise 1.2, which seeks to illustrate the comparison of the mean length between age 6 male and female Chinook salmon. Suppose we have already fitted this model and wish to summarize the output.

First, load the package into R:

```
library(postpack)
```

If you do not have it already, follow the installation instructions to get it. The output of our model is stored in an object called `post`, and is an object of class `mcmc.list`, see:

```
class(post)
```

```
## [1] "mcmc.list"
```

If we have forgotten, we can find out the names of the parameters (i.e., monitored nodes) in our model:

```
get_p(post)
```

```
## [1] "deviance" "diff_mu"  "diff_sig" "mu_f"     "mu_m"     "sig_f"
## [7] "sig_m"    "test_mu"  "test_sig"
```

The `"deviance"` node is monitored automatically, the rest of these quantities were ones we chose to monitor. If we are curious about the dimensions of our MCMC algorithm and output, we can run:

```
post_dim(post)
```

```
##      burn post_burn      thin    chains     saved     nodes
##      6000      5000         1         2     10000        13
```

Extracting Summaries

With the default methods, you can summarize an `mcmc.list` object as follows:

```
summary(post)
```

```
##
## Iterations = 6001:11000
## Thinning interval = 1
## Number of chains = 2
## 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   1070.5146  3.07395 0.0307395      0.0443962
## diff_mu      44.6257 16.93636 0.1693636      0.1693515
## diff_sig    -20.4531 12.69261 0.1269261      0.1840236
## mu_f        851.5863  6.75986 0.0675986      0.0675957
## mu_m        806.9605 15.41456 0.1541456      0.1541437
## sig_f        56.4416  4.93516 0.0493516      0.0701175
## sig_m        76.8947 11.73015 0.1173015      0.1719324
## test_mu[1]    0.9947  0.07261 0.0007261      0.0007819
## test_mu[2]    0.9761  0.15275 0.0015275      0.0015275
## test_mu[3]    0.9274  0.25949 0.0025949      0.0026217
## test_mu[4]    0.8123  0.39049 0.0039049      0.0038985
## test_mu[5]    0.6164  0.48629 0.0048629      0.0048630
## test_sig      0.0349  0.18354 0.0018354      0.0020868
##
## 2. Quantiles for each variable:
##
##               2.5%     25%     50%     75%    97.5%
## deviance   1066.72 1068.25 1069.81 1072.05 1078.145
## diff_mu      10.24   33.69   44.78   55.68   78.078
## diff_sig    -48.03  -27.97  -19.34  -11.67    1.499
## mu_f        838.19  847.20  851.64  855.94  865.058
## mu_m        776.63  796.93  806.78  816.72  838.160
## sig_f        47.70   53.00   56.08   59.46   67.115
## sig_m        58.27   68.53   75.49   83.62  103.510
## test_mu[1]    1.00    1.00    1.00    1.00    1.000
## test_mu[2]    1.00    1.00    1.00    1.00    1.000
## test_mu[3]    0.00    1.00    1.00    1.00    1.000
## test_mu[4]    0.00    1.00    1.00    1.00    1.000
```

```
## test_mu[5]    0.00    0.00    1.00    1.00    1.000
## test_sig      0.00    0.00    0.00    0.00    1.000
```

However, note that **all** nodes are summarized (which many take a long time if the model has many nodes) and that much of the important information is stored in more than one location in the summary (e.g., mean and standard deviations are stored separately from the quantiles, making them difficult to access).

With `post_summ()`, we can summarize only nodes we wish to, and correct the problem of storing different outputs separately:

```
post_summ(post, p = c("sig_f", "sig_m"))
```

```
##           sig_f     sig_m
## mean  56.441550  76.89468
## sd     4.935165  11.73015
## 50%   56.079576  75.48791
## 2.5%  47.704079  58.27181
## 97.5% 67.115161 103.50981
```

We can calculate convergence (`Rhat`) and sampling (`ess`) diagnostics for desired nodes as well:

```
post_summ(post, p = c("sig_f", "sig_m"), ess = T, Rhat = T)
```

You can report other quantiles if desired and round them:

```
post_summ(post, p = c("sig_f", "sig_m"),
          p_summ = c(0.025, 0.25, 0.5, 0.75, 0.975), rnd = 1)
```

Extracting Samples

Often you will want to take a subset out of your output while retaining each saved posterior sample, for example, to calculate the posterior of some derived quantity external to the JAGS model. This is particularly true for larger models with more monitored nodes, but we can illustrate the usage of `post_subset()` here:

```
post_sub = post_subset(post, p = c("mu_f", "mu_m"))
class(post_sub)
```

```
## [1] "mcmc.list"
```

```
get_p(post_sub)
```

```
## [1] "mu_f" "mu_m"
```

```
post_dim(post_sub)
```

```
##      burn post_burn      thin    chains     saved     nodes
##      6000      5000         1         2     10000         2
```

Notice the characteristics of our reduced output. If we wanted our output as a `matrix` object, while also knowing which chain and iteration each sample was, we could run:

```
post_sub = post_subset(post, p = c("mu_f", "mu_m"),
                       matrix = T, chains = T, iters = T)
class(post_sub)
```

```
## [1] "matrix"
```

```
head(post_sub); tail(post_sub)
```

```
##      CHAIN ITER     mu_f     mu_m
## [1,]     1 6001 857.5109 844.1988
## [2,]     1 6002 850.7043 809.4654
## [3,]     1 6003 863.9967 779.0806
## [4,]     1 6004 855.8765 810.4594
## [5,]     1 6005 844.6550 798.6125
## [6,]     1 6006 850.6184 787.4258
```

```
##          CHAIN  ITER     mu_f     mu_m
##  [9995,]     2 10995 850.8820 817.0188
##  [9996,]     2 10996 862.1750 789.2286
##  [9997,]     2 10997 854.3016 798.6603
##  [9998,]     2 10998 840.9702 821.6403
##  [9999,]     2 10999 859.7436 803.0222
## [10000,]     2 11000 847.5422 795.9223
```

The `chains` and `iters` arguments are both optional and are `FALSE` by default.

Subsetting with Regular Expressions

Previous versions of these functions required that if you wanted to subset both `mu_m` and `mu_f`, you had to write them explicitly as above. However, with the magic of pattern matching with regular expressions, and particularly the `stringr::str_detect()` function, far more

general subsetting is now available. However, you will need to learn some very basic regular expression syntax to get it to work properly.

Pattern matching is driven by the `match_p()` function, so if you're ever confused about why you're getting some output but not exactly what you expected, you can use this function to help you out. We will use it here to illustrate the basic concepts for our needs.

Suppose we want output from all nodes containing the string `"mu"`:

```
match_p(post, p = "mu")
```

```
## [1] "diff_mu"    "mu_f"       "mu_m"       "test_mu[1]" "test_mu[2]"
## [6] "test_mu[3]" "test_mu[4]" "test_mu[5]"
```

Now, if we only want nodes that start with `"mu"`, we can write:

```
match_p(post, p = "^mu")
```

```
## [1] "mu_f" "mu_m"
```

The `^` filters out any nodes that do not start with `"m"` and are immediately followed by `"u"`. Similarly, we can force a match to end with `"f"` with the `$`:

```
match_p(post, p = "f$")
```

```
## [1] "mu_f"  "sig_f"
```

Remember, `match_p()` was only used here as an example to illustrate the pattern matching – this works in *every* `{postpack}` function that takes a `p` argument:

```
post_summ(post, p = "f$")
```

```
##            mu_f     sig_f
## mean  851.58627 56.441550
## sd      6.75986  4.935165
## 50%   851.64001 56.079576
## 2.5%  838.18798 47.704079
## 97.5% 865.05831 67.115161
```

Once you get used to how this works, you might start being strategic about how you name your nodes in your models to facilitate easy subsetting.

Nodes with multiple elements

Note that there is one node (`"test_mu"`) in our output that has multiple elements (note the usage of the `type = "all"` argument, which says you want to see **all** monitored nodes, not only the unique base names):

```
get_p(post, type = "all")
```

```
##  [1] "deviance"   "diff_mu"    "diff_sig"   "mu_f"       "mu_m"
##  [6] "sig_f"      "sig_m"      "test_mu[1]" "test_mu[2]" "test_mu[3]"
## [11] "test_mu[4]" "test_mu[5]" "test_sig"
```

If we try to match this node without any other information specified, we will get all elements:

```
match_p(post, "test_mu")
```

```
## [1] "test_mu[1]" "test_mu[2]" "test_mu[3]" "test_mu[4]" "test_mu[5]"
```

If we wish to only see the third element, we would need to specify its name **exactly**:

```
match_p(post, "test_mu[3]")
```

```
## [1] "test_mu[3]"
```

(Note that in this case, `match_p(post, "3")` would also work, as it is the only node with a `"3"` in the node name). When using regular expressions, you would normally need to escape the `"["` and `"]"`, because they have special meanings in regular expressions (i.e., `"test_mu\\[3\\]"`), however, the author of `{postpack}` included functionality to insert these escapes if they are not present (or ignore them if they are).

If you would like to extract the elements 2, 3, and 5, you could use the `|` operator:

```
match_p(post, "test_mu[2|3|5]")
```

```
## [1] "test_mu[2]" "test_mu[3]" "test_mu[5]"
```

More regular expressions

As final example, suppose you have a model with nodes`"z0"`, `"z1"`, `"z[1]"`, and`"z[2]"` (the `post` object from Exercise 1.2 does not have these nodes - this is hypothetical). If you run:

```
post_summ(post, "z")
```

You will get a summary of all nodes containing the character `"z"`, which is all nodes. If you want only the nodes `"z0"` and `"z1"`, you can run:

```
post_summ(post, "z.$")
```

The `"."` acts as a wildcard that will match anything. So this says "match all nodes that have `"z"` and are followed by only one character". If you want to extract `"z[1]"` and `"z[2]"` *only*, you could run:

```
post_summ(post, "z[")
```

A far more exhaustive description of how regular expressions work, especially in the context of the `{stringr}` package (on which `{postpack}` strongly depends), is provided here – though note that the `"["` and `"]"` symbols have different behavior in the `{postpack}` functions as they are always escaped. Future versions of the package may make the automatic escape optional.

Other useful features

- Create diagnostic plots (density and trace plots for each chain) for requested nodes (see `?diag_plots`)
- Convert a `matrix` object to a `mcmc.list` object (see `?matrix2mcmclist`)
- Convert summary statistics (e.g., posterior mean) into the array format used by the model (see `?array_format`)
- Combine multiple `mcmc.list` objects into one (see `?post_bind`)
- Thin the retained posterior samples at evenly spaced intervals, perhaps for the development of long-running post-processing code (see `?post_thin`)

Part II

R/JAGS CODE EXAMPLES

Exercise 2.1: Logistic Regression

```
jags_model = function() {
  # PRIORS
  b0 ~ dnorm(0, 1e-6)
  b1 ~ dnorm(0, 1e-6)

  # LIKELIHOOD
  for (i in 1:n_obs) {
    y[i] ~ dbern(p[i])
    logit(p[i]) <- b0 + b1 * x[i]
  }

  # DERIVED QUANTITIES
  for (t in 1:n_pred) {
    logit(pred_p[t]) <- b0 + b1 * pred_x[t]
  }
}
```

Exercise 2.2: Poisson Regression

```
jags_model = function() {
  # PRIORS
  b0 ~ dnorm(0, 1e-6)
  b1 ~ dnorm(0, 1e-6)

  # LIKELIHOOD
  for (i in 1:n_obs) {
    y[i] ~ dpois(lambda[i])
    log(lambda[i]) <- b0 + b1 * x[i]
  }

  # DERIVED QUANTITIES
  for (i in 1:n_pred) {
    log(pred_y[i]) <- b0 + b1 * pred_x[i]
  }
}
```

Exercise 2.3: Posterior Predictive Checks and Neg. Bin. Regression

Analysis assuming Poisson

First, use `y = dat$redds` in `jags_data` then `y = dat$redds_od`

```
jags_model = function() {
  # PRIORS
  b0 ~ dnorm(0, 1e-6)
  b1 ~ dnorm(0, 1e-6)

  # LIKELIHOOD
  for (i in 1:n_obs) {
    y[i] ~ dpois(lambda[i])
    log(lambda[i]) <- b0 + b1 * x[i]
  }

  # DERIVED QUANTITIES
  for (i in 1:n_pred) {
    log(pred_y[i]) <- b0 + b1 * pred_x[i]
  }

  # POSTERIOR PREDICTIVE CHECK
  for (i in 1:n_obs) {
    # draw random data under model assumptions
    y_new[i] ~ dpois(lambda[i])

    # calculate pearson residuals
    obs_resid[i] <- (y[i] - lambda[i])/sqrt(lambda[i])
    new_resid[i] <- (y_new[i] - lambda[i])/sqrt(lambda[i])

    # calculate squared residuals
    D_obs[i] <- obs_resid[i]^2
    D_new[i] <- new_resid[i]^2
  }

  # sum for total fit criterion
  fit_obs <- sum(D_obs[])
  fit_new <- sum(D_new[])

  # test for if fit_obs > fit_new
  bp <- step(fit_obs - fit_new)
}
```

Analysis assuming negative binomial

Start with `y = dat$redds_od` then move back to `y = dat$redds` (`r` should run up against the prior)

```
jags_model = function() {
  # PRIORS
  b0 ~ dnorm(0, 1e-6)
  b1 ~ dnorm(0, 1e-6)
  r ~ dunif(0, 10)

  # LIKELIHOOD
  for (i in 1:n_obs) {
    y[i] ~ dnegbin(p[i], r)
    p[i] <- r/(lambda[i] + r)
    log(lambda[i]) <- b0 + b1 * x[i]
  }

  # DERIVED QUANTITIES
  for (i in 1:n_pred) {
    log(pred_y[i]) <- b0 + b1 * pred_x[i]
  }

  # POSTERIOR PREDICTIVE CHECK
  for (i in 1:n_obs) {
    # draw random data under model assumptions
    y_new[i] ~ dnegbin(p[i], r)

    # calculate pearson residuals
    obs_resid[i] <- (y[i] - lambda[i])/sqrt(lambda[i])
    new_resid[i] <- (y_new[i] - lambda[i])/sqrt(lambda[i])

    # calculate squared residuals
    D_obs[i] <- obs_resid[i]^2
    D_new[i] <- new_resid[i]^2
  }

  # sum for total fit criterion
  fit_obs <- sum(D_obs[])
  fit_new <- sum(D_new[])

  # test for if fit_obs > fit_new
  bp <- step(fit_obs - fit_new)
}
```

Negative Binomial Parameterization?

The negative binomial distribution (in its strict interpretation) models the number of failures resulting from a series of Bernoulli trials with common success parameter p before r successes occur. However, it is commonly used to model over-dispersed counts. We use it parameterized in terms of the mean count (λ) and extra-Poisson variability (r).

The JAGS user manual 4.3.0 (page 51) lists that:

$$\lambda = \frac{r(1-p)}{p}$$

In regression, we must predict $\log(\lambda)$ from covariates, so we must make p a function of λ and r. This requires some basic algebra:

Move p to left-hand side:

$$\lambda p = r(1-p)$$

Distribute r on right-hand side:

$$\lambda p = r - rp$$

Move rp to left-hand side:

$$\lambda p + rp = r$$

Factor out the common p term:

$$p(\lambda + r) = r$$

Divide to get p alone:

$$p = \frac{r}{\lambda + r}$$

Just as is used in our JAGS code above. More information on the negative binomial distribution can be found in the R documentation: `?rnbinom` - they even allow the parameterization we use here (and note that it is commonly used in ecology):

```
x = rnbinom(n = 1e6, mu = 30, size = 2)
mean(x)
```

```
## [1] 29.97953
```

Where `mu` is λ and `size` is r. The negative binomial can also be expressed as a gamma-Poisson mixture, but we won't show that in the workshop as JAGS has the `dnegbin()` function.

Exercise 2.4: Zero-Inflated Poisson Regression

```
jags_model = function() {
  # PRIORS
  b0 ~ dnorm(0, 1e-6)
  b1 ~ dnorm(0, 1e-6)
  z0 ~ dnorm(0, 1e-3)
  z1 ~ dnorm(0, 1e-3)

  # LIKELIHOOD
  for (i in 1:n_obs) {
    # count model
    y[i] ~ dpois(z[i] * lambda[i])
    log(lambda[i]) <- b0 + b1 * x1[i]

    # zero model
    logit(psi[i]) <- z0 + z1 * x2[i]
    z[i] ~ dbern(1 - psi[i]) # psi is pr(unsuitable)
  }

  # DERIVED QUANTITIES
  logit(psi_lw) <- z0
  logit(psi_hi) <- z0 + z1

  for (i in 1:n_pred) {
    log(pred_y[i]) <- b0 + b1 * pred_x[i]
  }
}
```

Exercise 2.5: Mixed-Effect Logistic Regression

Fixed-Effects Formulation

```
jags_model = function() {
  # PRIORS
  for (j in 1:n_grp) {
    b0[j] ~ dnorm(0, 1e-3)
    b1[j] ~ dnorm(0, 1e-3)
  }

  # LIKELIHOOD
  for (i in 1:n_obs) {
    y[i] ~ dbern(p[i])
    logit(p[i]) <- b0[grp[i]] + b1[grp[i]] * x[i]
  }

  # DERIVED QUANTITIES
  for (j in 1:n_grp) {
    for (i in 1:n_pred) {
      logit(pred_p[i,j]) <- b0[j] + b1[j] * pred_x[i]
    }
  }
}
```

Mixed-Effect Formulation

```
jags_model = function() {
  # HYPER-PRIORS (PRIORS ON THE HYPERDIST)
  B0 ~ dnorm(0, 1e-3)
  B1 ~ dnorm(0, 1e-3)
  sig_B0 ~ dunif(0, 10)
  sig_B1 ~ dunif(0, 10)
  tau_B0 <- 1/sig_B0^2
  tau_B1 <- 1/sig_B1^2

  # YEAR EFFECTS
  for (j in 1:n_grp) {
    b0[j] ~ dnorm(B0, tau_B0)
    b1[j] ~ dnorm(B1, tau_B1)
  }

  # LIKELIHOOD
  for (i in 1:n_obs) {
    y[i] ~ dbern(p[i])
    logit(p[i]) <- b0[grp[i]] + b1[grp[i]] * x[i]
  }

  # DERIVED QUANTITIES
  for (j in 1:n_grp) {
    for (i in 1:n_pred) {
      logit(pred_p[i,j]) <- b0[j] + b1[j] * pred_x[i]
    }
  }
}
```

Exercise 2.6: Cormack-Jolly-Seber Models

Exercise 2.6.1: Simplest Model

```
jags_model = function() {
  # PRIORS
  phi ~ dbeta(1,1)  # survival probability between arrays
  p ~ dbeta(1,1)    # detection probability at each array

  # LIKELIHOOD
  for (i in 1:n) {
    # j = 1 is the release occasion - known alive
    for (j in 2:J) {
      # survival process: must have been alive in j-1 to have non-zero pr(alive at j)
      z[i,j] ~ dbern(phi * z[i,j-1])

      # detection process: must have been alive in j to observe in j
      y[i,j] ~ dbern(p * z[i,j])
    }
  }

  # DERIVED QUANTITIES
  # survivorship is probability of surviving from release to a detection occasion
  survship[1] <- 1
  for (j in 2:J) {
    survship[j] <- survship[j-1] * phi
  }
}
```

Exercise 2.6.2: Occasion-Level Covariates

```
jags_model = function() {
  # PRIORS
  a0 ~ dunif(-5, 5)  # for detection
  a1 ~ dunif(-5, 5)  # for detection
  b0 ~ dunif(-5, 5)  # for survival
  b1 ~ dunif(-5, 5)  # for survival

  # APPLY COEFFICIENTS/COVARIATES
  for (j in 2:J) {
    logit(p[j]) <- a0 + a1 * width[j]
    logit(phi[j]) <- b0 + b1 * birds[j]
  }

  # LIKELIHOOD
  for (i in 1:n) {
    # j = 1 is the release occasion - known alive
    for (j in 2:J) {
      # survival process
      z[i,j] ~ dbern(phi[j] * z[i,j-1])

      # detection process
      y[i,j] ~ dbern(p[j] * z[i,j])
    }
  }

  # DERIVED QUANTITIES
  survship[1] <- 1
  for (j in 2:J) {
    survship[j] <- survship[j-1] * phi[j]
  }
}
```

Exercise 2.6.3: Individual-Level Covariate

```
jags_model = function() {
  # PRIORS
  a0 ~ dunif(-5, 5)  # for detection
  a1 ~ dunif(-5, 5)  # for detection
  b0 ~ dunif(-5, 5)  # for survival
  b1 ~ dunif(-5, 5)  # for survival
  b2 ~ dunif(-0.1, 0.1) # for survival

  for (j in 2:J) {
    logit(psi[j]) <- a0 + a1 * width[j]
  }

  # LIKELIHOOD
  for (i in 1:n) {
    for (j in 2:J) {
      # apply site and individual-based survival
      logit(phi[i,j]) <- b0 + b1 * birds[j] + b2 * length[i]

      # survival process
      z[i,j] ~ dbern(phi[i,j] * z[i,j-1])

      # detection process
      y[i,j] ~ dbern(psi[j] * z[i,j])
    }
  }

  # DERIVED QUANTITIES
  # survival between occasions and at a given length
  for (i in 1:n_pred) {
    for (j in 1:2) {
      logit(pred_phi[i,j]) <- b0 + b1 * pred_birds[j] + b2 * pred_length[i]
    }
  }

  # survivorship for fish of different sizes
  survship_small[1] <- 1
  survship_large[1] <- 1
  for (j in 2:J) {
    survship_small[j] <- survship_small[j-1] * pred_phi[1,birds[j] + 1]
    survship_large[j] <- survship_large[j-1] * pred_phi[30,birds[j] + 1]
  }
}
```

Bonus Exercises: More State-Space Models

Example 1: Pink Salmon Spawner-Recruit Model

```
jags_model = function() {
  # priors
  alpha ~ dunif(1, 20)
  beta ~ dunif(0,1)
  sigmaR ~ dunif(0,3)
  tauR <- 1/sigmaR^2

  ## BIOLOGICAL PROCESS MODEL ##
  # estimate true recruitment in first brood year (no spawner link)
  Rmean[1] <- log(alpha)/beta  # unfished equilibrium recruits
  R[1] ~ dlnorm(log(Rmean[1]), tauR)

  # use ricker dynamics w/white noise process noise for remaining years
  for (y in 2:nyrs) {
    Rmean[y] <- alpha * S[y-1] * exp(-beta * S[y-1])  # deterministic recruitment
    R[y] ~ dlnorm(log(Rmean[y]), tauR)                # latent (realized) recruitment
  }

  ## HARVEST PROCESS MODEL ##
  for (y in 1:nyrs) {
    U[y] ~ dbeta(1,1)          # annual exploitation rate
    H[y] <- R[y] * U[y]        # harvest each year
    S[y] <- R[y] * (1 - U[y])  # escapement each year
  }

  ## OBSERVATION MODEL ##
  for (y in 1:nyrs) {
    Sobs[y] ~ dlnorm(log(S[y]), tauS)
    Hobs[y] ~ dlnorm(log(H[y]), tauH)
  }

  ## MANAGEMENT QUANTITIES ##
  # approximations
  Smsy <- log(alpha)/beta * (0.5 - 0.07 * log(alpha))
  MSY <- alpha * Smsy * exp(-beta * Smsy) - Smsy

  # for a optimum yield probability profile
  for (i in 1:nSpred) {
    # predict recruitment and sustained yield at each level of escapement
    Rpred[i] <- alpha * Spred[i] * exp(-beta * Spred[i])
```

```
    SY[i] <- Rpred[i] - Spred[i]

    # test for whether yield would be at least 70%, 80%, or 90% of MSY
    p_meet[i,1] <- step(SY[i] - (0.7 * MSY))
    p_meet[i,2] <- step(SY[i] - (0.8 * MSY))
    p_meet[i,3] <- step(SY[i] - (0.9 * MSY))
  }
}
```

Example 2: Tuna Biomass-Dynamics Model

```
jags_model = function() {
  # PRIORS
  r ~ dunif(0.1, 1.2)            # population growth rate
  k ~ dunif(100, 400)            # carrying capacity
  q ~ dunif(0.001, 0.5)          # index catchability
  sig_proc ~ dunif(0, 1)         # process noise sd
  tau_proc <- 1/sig_proc^2       # process noise precision

  ## BIOLOGICAL PROCESS MODEL ##
  B_mean[1] ~ dunif(0, 300)                     # expected initial biomass
  B_true[1] ~ dlnorm(log(B_mean[1]), tau_proc) # realized latent initial biomass
  for (t in 2:nyrs) {
    # expected states
    B_mean[t] <- max(B_true[t-1] + r * B_true[t-1] * (1 - (B_true[t-1]/k)) - C[t-1], 1)

    # realized latent states
    B_true[t] ~ dlnorm(log(B_mean[t]), tau_proc)
  }

  ## OBSERVATION MODEL ##
  tau_obs <- 1/sig_obs^2         # obsevation error on index catches
  for (t in 1:nyrs) {
    index[t] ~ dlnorm(log(index_true[t]), tau_obs)
    index_true[t] <- q * B_true[t]
  }

  ## MANAGEMENT QUANTITIES ##
  Bmsy <- k/2
  MSY <- (r * k)/4
  # montior the probability that B_true is greater than 90% of Bmsy through time
  for (t in 1:nyrs) {
    p90_Bmsy[t] <- step(B_true[t] - (0.9 * Bmsy))
  }

  # biomass projection
  B_1990 <- B_true[nyrs] + r * B_true[nyrs] * (1 - (B_true[nyrs]/k)) - C[nyrs]
}
```

References

Ellison, A. M. (1996). An introduction to Bayesian inference for ecological research and environmental decision-making. *Ecological Applications*, 6(4):1036–1046.

Gelman, A., Carlin, J. B., Stern, H. S., Dunson, D. B., Vehtari, A., and Rubin, D. B. (2014). *Bayesian Data Analysis*. CRC Press, Boca Raton, 3 edition.

Kéry, M. (2010). *Introduction to WinBUGS for ecologists: a Bayesian approach to regression, ANOVA, mixed models, and related analyses*. Elsevier/Academic Press, Burlington, Massachusetts.

Kéry, M. and Schuab, M. (2012). *Bayesian population analysis using WinBUGS: a hierarchical perspective*. Academic Press, Waltham, MA.

King, R. (2012). A review of Bayesian state-space modelling of capture–recapture–recovery data. *Interface Focus*, 2(2):190–204.

Link, W. A. and Barker, R. J. (2010). *Bayesian inference: with ecological applications*. Elsevier/Academic Press, Amsterdam Boston London.

Ludwig, D. (1996). Uncertainty and the assessment of extinction probabilities. *Ecological Applications*, 6(4):1067–1076.

McCarthy, M. (2007). *Bayesian methods for ecology*. Cambridge University Press, Cambridge, UK New York.

Meyer, R. and Millar, R. B. (1999). BUGS in Bayesian stock assessments. *Canadian Journal of Fisheries and Aquatic Sciences*, 56(6):1078–1087.

Punt, A. and Hilborn, R. (1997). Fisheries stock assessment and decision analysis: the Bayesian approach. *Reviews in Fish Biology and Fisheries*, 7(1):35–63.

Royle, J. A. and Dorazio, R. M. (2008). *Hierarchical modeling and inference in ecology: the analysis of data from populations, metapopulations and communities*. Academic Press, Amsterdam Burlington, MA.

Wade, P. R. (2000). Bayesian methods in conservation biology. *Conservation Biology*, 14(5):1308–1316.