Occupancy Models with Stan (with help from R)

Richard Erickson

5/7/2018

Contents

Overview

This document provides an introduction to occupancy models in Stan. Note that the terms occupancy models and occurrence models can be used synonymous from a statistical and programming perspective, but can have important ecological differences from a scientific perspective. Both terms are used interchangeably within this document unless specifically indicated that one term is appropriate for a specific location. This document to help collaborators learn how program occupancy models in Stan. Occupancy models can be difficult to program in Stan because they have discrete latent variables (i.e., the site-level occupancy). However, compared to other Bayesian programs, Stan can offer better performance (e.g., more rapidly fitting models, converging when other models could not, fewer iterations required).

This tutorials assumes the reader is familiar with R, occupancy models, and Bayesian statistics. Some base knowledge of Stan would be helpful, even if it is just completing the schools "hello world" example.

The introduction covers a brief overview of occupancy models, logistic regression in Stan and R, and matrix notation in R. Next, two-level occupancy models are introduced. Following this, three-level occupancy models are introduced. More topics may be included (e.g., changing variance structures or correlated estimates) as I have time or need to program these models.

Introduction

This section provides a crash course on useful topics and code for using occupancy models with Stan in R.

A brief overview of occupancy models

Occupancy models allow for imperfect detection. For example, if I look out my back window and do not see a robin, why did I not see it? Maybe the bird was not there or maybe I missed it. If I look out my window on a regular basis, I could use an occupancy model to estimate my probability of detection. Occupancy models can be extended to other situations with imperfect detection as well.

Perfect detection: The logistic regression

In a world with perfect detection (i.e., $p = 1$), we would not need occupancy models because the species we are looking for would always be detected if they were present. In these situations, we could simply use a logistic regression to model the probability that a site is occupied. In base R, we would use the `glm()` function to fit this model using a maximum likelihood approach:

```
glm(y ~ x, family = 'binomial')
```

Note, that if we want, we can fit the model using a probit link function rather than a logit:

```
glm( y ~ x, family = binomial(link = 'probit'))
```

Many good tutorials exist on the differences between probits and logits. Usually, the models produce similar results, although the logit has a slightly fatter tail. Personally, I prefer logits if for no other reason than that is what my PhD advisor used.

The logit model may be also easily coded in Stan as well. I would save this script as a `logisticStan.stan` (**Important note:** Stan models must end with `.stan` with a lower case `s`, otherwise the model will not be complied):

```
data {
  int<lower=0> N;
  vector[N] x;
  int<lower=0,upper=1> y[N];
}
parameters {
  real alpha;
  real beta;
}
model {
  y ~ bernoulli_logit(alpha + beta * x);
}
```

After saving this code as `logisticStan.stan`, we can call the model from R. We'll simulate some data and format it to work with Stan here as well.

```
library(rstan)

### simulate and examine raw data
nObs = 20
xLabel <- factor(rep(c("a", "b"), times = nObs)) # used to Id variables
xProb  <- rep(c(0.25, 0.75), times = nObs) # used to simulate data
y <- rbinom(n = nObs * 2, size = 1, p = xProb)
aggregate(y, by = list(xLabel), FUN = mean)

### First, we fit a GLM using base R
glmOut <- glm(y ~ xLabel, family = 'binomial')
summary(glmOut)
```

```
### Second, we fit the model using Stan:
stanData <- list(N = length(y),
                 x = as.numeric(xLabel) - 1,
                 y = y)

stanOut <- stanOut0 <- stan(file = "logisticStan.stan",
                 data = stanData, chains = 4, iter = 500)

### Useful functions for looking at Stan outputs include:
print(stanOut)
traceplot(stanOut)
traceplot(stanOut, inc_warmup = TRUE)
plot(stanOut)
pairs(stanOut)
```

A few things to point out.

1. Unlike base R, rstan does not have a nice wrapper function. This means we must use integers as the id variable each observation rather than characters or factors.
2. You will want to increase the number of iterations when actually running the model for use. When troubleshooting, I use 100s. When running for publication, I use 1,000s to 10,000s.
3. Notice the Bayesian model diagnostics. If these are unfamiliar to you, I suggest working through BDA3 by Gelman et al. or other introductory Bayesian stats book. The Shinystan interface provides many different Bayesian diagnostics tools that work with Stan.

However, when working with occupancy, we must account for imperfect detection. Because of this, and the details of how Stan works under the hood, we cannot use the default probability functions for Stan. To introduce this topic, we will next look at using the probability `target` in Stan. The `+=` opperator is a programming shortcut. For example `x = x + 2`, can be written as `x += 2`. Our code now looks like this and is saved in the file `logisticTargetStan.stan`:

```
data {
  int<lower=0> N; // Number of samples
  vector[N] x; // predictor vector
  int<lower=0,upper=1> y[N];//response vector
}
parameters {
  real alpha; //intercept
  real beta; ///slope
}
model {
  for( index in 1:N){
    target += binomial_logit_lpmf( y[index] | 1, alpha + beta * x[index]);
      }
}
```

From both of the above example, there are some important features of Stan to note:

First, unlike R or JAGS, Stan requires us to explicitly declare variables like C++. A downside is that the language can be less forgiving. An upside is that the language makes us be precise and makes it harder to "programming by coincidence". Rather than simply estimate parameters using a model, we must understand their data structure within the model. That being said, I've spend many an hour refreshing linear algebra to understand my Stan code and dimensions of objects. But, the end products were code that I now trust and run quickly.

Second, Stan uses code blocks. These are defined in the Stan manual. In a nutshell, the require us to declare

the input `data`, the estimated `parameters`, and our `model`. There are also other types of blocks you can look up in the manual and we might see some of them later.

Third, we can include comments with `//` or blocks of code with `/* comment here */`.

```
stanOutTarget <- stan(file = "logisticTargetStan.stan",
                      data = stanData, chains = 4, iter = 500)

## Useful functions for looking at Stan outputs include:
print(stanOutTarget, pars = c("alpha", "beta", "lp__"))
traceplot(stanOutTarget, pars = c("alpha", "beta", "lp__"))
traceplot(stanOutTarget, inc_warmup = TRUE, pars = c("alpha", "beta", "lp__"))
plot(stanOutTarget, pars = c("alpha", "beta", "lp__"))
```

Exercise: Using the code above, fit a probit regression. You'll need to go to the Stan documentation to figure this out.

Binomial versus Bernoulii and long versus wide data

Another important concept for occupancy modeling is data structure and probability distributions. The Bernoulli distribution produced one sampling event (often denoted as $K = 1$). For example, we might flip a coin once and record this as a data entry. The Bernoulli is a special case of a binomial distribution. The binomial distribution allows us to have multiple sampling events. For example, we might flip a coin 10 times and record the number of heads and tails as their own columns.

For both general data analysis and occurrence modeling, I use both distributions. When fitting a simple binomial GLM in R, my modeling choice depends upon the structure of the data. I use a Bernoulli style input (a vector of 0s and 1s for `y`) if my data has coefficients for each observation or the data was given to me in that format. I use a binomial style input if my data has been aggregated or I want to avoid pseudoreplication (e.g., the tank is the level of replication rather than the individual). R has two methods for inputting binomial style data. First, a matrix of "successes" and "failures" maybe used for `y`. Second, a vector of probabilities may be used for `y` and a `weight =` option specified. Closely relate to these distributions are the data concepts of "wide versus long data in R" and "aggregate versus raw data".

During this code example, you will see how to fit a model using all three methods as well as how to convert code between wide and long formats.

```
## Simulate data in long format
set.seed(1223)
xSim <- rep(c(0.25, 0.75), each = 14)
x <- rep(c("a", "b"), each = 14)
y <- rbinom(n = length(xSim), size = 1, prob = xSim)
dataLong <- data.frame(x = x, y = factor(y, labels = c("fail", "success")))

### cast the data to wide format using reshape2
library(reshape2)
dataWide <- dcast(dataLong, x ~ y)
dataWide$Total <- with(dataWide, success + fail)
dataWide$successProportion <- with(dataWide, success /Total)
dataWide

### Compare the three methods for fitting a logistic regression in R
glm(y ~ x, family = 'binomial', data = dataLong)

glm(cbind(fail, success) ~ x, family = 'binomial', data = dataWide)
```

```
glm(successProportion ~ x, family = 'binomial', data = dataWide,
    weights = Total)
```

For occurrence models in Stan, we must use the Bernoulli distribution (or Binomial with `K = 1`) for the latent variables because we cannot aggregate the data. Specifically, we need details about each replicate at a lower level. For example, We cannot aggregate and say that 3 sites had Robins and 2 sites did not. Instead, we need a vector of of these site-level detentions, for example `c(0, 1, 1, 0, 1, 1, 1)`. For the lowest level of the occurrence model, I often do use a Bernoulli distribution when I do not have coefficients at the observation-level because there are fewer data entries to aggregate over. We will see these

Matrix notation and occupancy models

Models in R such as `lm()` and `glm()` allow users to input formuals. Formulas allow users to input factors and have R convert them to a matrix of dummy variables. `model.matrix()` allows us to create these same type matrices of input variables. There are several benefits to using `model.matrix()` to pre-process inputs for Stan. First, it allows us to easily turn factors into dummy variables. Second, it allows us to easily have matrices of predictors, which in turn allows us to us matrix algebra within Stan. This section introduces `model.matrix()` so that it will be familiar to us later. **Note:** I use shorter matrices than most real applications to save screen space.

model.matrix basis

`model.matrix()` use the `~` (shift-‘ on US keyboards) for its input. In statistical English, this could be read as “predicted by”, for example `y ~ x` could be spoken or read as y predicted by x. The follow example demonstrates how it may be used on a simple factor data.frame:

```
df <- data.frame(city = c("La Crosse", "St. Louis", "Cairo"))
model.matrix( ~ city, data = df)
```

```
##   (Intercept) cityLa Crosse citySt. Louis
## 1           1             1             0
## 2           1             0             1
## 3           1             0             0
## attr(,"assign")
## [1] 0 1 1
## attr(,"contrasts")
## attr(,"contrasts")$city
## [1] "contr.treatment"
```

Several things to notice.

1. `model.matrix()` converted city to an alphabetical order factor.
2. The first factor is the first in alphabetically. This order may be changing the factor order in R.
3. The first factor become a global intercept and the other two levels are compared to this. In the next section, we’ll see how to change this.

Intercept for each level

If we want an intercept for each factor level, we use a `- 1` in the notation.

If we have multiple factors, we can only estimate intercepts for all of one of the factors. For example, if we have months and city, we would need a reference month *or* reference city. Also, notice how order matters. Any decent advanced book on regression analysis explains this in greater detail (e.g., Harrell or Gelman and Hill).

```
df <- expand.grid(city = c("La Crosse", "St. Louis", "Cairo"),
                  month = c("May", "June"))
model.matrix( ~ city + month -1, data = df)
model.matrix( ~ month + city -1, data = df)
```

numeric vs factors

We can also use numeric inputs with `model.matrix()` For example, if we input month as a numeric vector, R creates a matrix with month as a numeric column. If we were using the matrix in a regression, this new column would correspond to a slope estimate.

```
df1 <- expand.grid(month = c( 5, 6, 7))
model.matrix( ~ month, data = df1)
```

```
##   (Intercept) month
## 1           1     5
## 2           1     6
## 3           1     7
## attr(,"assign")
## [1] 0 1
```

Conversely, if we input month as a factor, we get similar results as before.

```
df2 <- expand.grid(month = factor(c( 5, 6, 7)))
model.matrix( ~ month, data = df2)
```

```
##   (Intercept) month6 month7
## 1           1      0      0
## 2           1      1      0
## 3           1      0      1
## attr(,"assign")
## [1] 0 1 1
## attr(,"contrasts")
## attr(,"contrasts")$month
## [1] "contr.treatment"
```

The purpose of this example is to demonstrate how R can sometimes produce unexpected results, especially if we want a measure of time to correspond to an intercept estimate rather than a slope estimate.

Subtract 1 from as.numeric()

Closely related to the above point is a problem I have run into when creating binary response variables in R for use with Stan. For example, let's say we want to model occupancy for a lake and a river:

```
set.seed(12351513)
dfocc <- data.frame(occ = factor(sample(rep(c("yes", "no"), each = 10))),
                    site = factor(rep(c("lake", "river"), each = 10)))
```

Using Base R, we could just run `glm()` on this data:

```
summary(glm(occ ~ site, data = dfocc, family = 'binomial'))
```

But, look at what happens if we try and convert `occ` to a binary response:

```
### base line
as.numeric(factor(dfocc$occ))
```

```
### need -1 to create a vecor of zeros and ones
as.numeric(factor(dfocc$occ)) - 1
```

Now that you've had a crash course on R topics, let's build our first occupancy model with Stan.

Two-level occupancy models

This chapter starts with a simple occupancy model that has one site with multiple observation points (e.g., cameras or traps). The first and simplest model directly estimates probabilities of detection and site occupancy. This model takes a site-by-visit (row of sites, columns of visits or surveys) matrix as the input. The second model then gets changed to estimate parameters on the logit-scale. The third model uses an input vector of the sum of number visits with detentions. The fourth model uses an input vector of binary observations for each visit to a site.

A simple occupancy model in Stan

We will start by building a simple occupancy model that only includes a global intercept at each level. This example is based upon the Stan example provided by Bob Carpenter. The model would be appropriate for a single site with multiple visits or multiple sites with a single visit. This example either uses space or time for replication. The model assumes the same probability of detection for each observation. For simplicity, we will assume that a single site is revisited multiple times.

The probability that the site is occupied is ψ and is estimated on the logit scale: μ_ψ. μ_ψ is predicted by $X\beta$. If a site is occupied during a visit, $Z = 1$, otherwise, it would be zero. The probability of detection at a site is p and is estimated on the logit scale: μ_p. μ_ψ is predicted by $V\alpha$. The raw data for this would Y where the index of Y corresponds to the observation. Y must be binary (i.e., 0s or 1s). Because we are estimating a single intercept, we only need a vector of 1s as our predictor matrix (or, in this case, scalar) X. Note that we could simplify this model several ways (e.g., hard coding a single intercept, estimating on the probability scale rather than logit), but use the current formulation because we need these tools later either to extend the model or for numerical efficiency.

Matrix input occupancy model

The simplest version of this model use The model in Stan contains three code blocks. The first is the `data` block:

```
data {
  int<lower=0> nSites;
  int<lower=0> nSurveys;
  int<lower=0,upper=1> y[nSites, nSurveys];
}
```

which defines the input data into Stan. The second is the `parameters` block:

```
parameters {
  real<lower=0,upper=1> psi;
  real<lower=0,upper=1> p;
}
```

which defines the parameters being estimated. The third block is the `model` block. This block includes local variables to increase the computational efficient of the code. The code also includes priors. If a user does not specify priors with Stan, then Stan has default priors that account for the constraints of parameters. The

third part of the block is the likelihood function. This includes an if-else statement to account for detection (`if`) or non-detection (`else`) at a site.

```
model {
  // local variables to avoid recomputing log(psi) and log(1 - psi)
  real log_psi;
  real log1m_psi;
  log_psi = log(psi);
  log1m_psi = log1m(psi);

  // priors
  psi ~ uniform(0,1);
  p   ~ uniform(0,1);

  // likelihood
  for (r in 1:nSites) {
    if (sum(y[r]) > 0)
      target += log_psi + bernoulli_lpmf(y[r] | p);
    else
      target += log_sum_exp(log_psi + bernoulli_lpmf(y[r] | p),
                log1m_psi);;
  }
}
```

The data simulation process also can provide us with insight into the how the model works and how we assume our data generation process works. For this case, we are specifying the number of sites and visits (or surveys) per site. We also specify our simulated probability of detection p, `p`, and site occupancy probability psi, `psi`. The next simulation step is to simulate if the sites are occupied and then simulation the surveys at each site. Last, the data are wrapped into a list for Stan.

```
## Simulate true occupancy states

set.seed(1234)
nSites <- 250
nSurveys <- 10
p <- 0.6
psi <- 0.4

## simulate site-level occupancy
z <- rbinom( nSites, 1, psi)

## simulate  sample-level detections
y <- matrix( NA, nSites, nSurveys);
for (site in 1:nSites)
  y[site,] <- rbinom( nSurveys, 1, z[site] * p);


simpleModelInput <- list(nSites = nSites,
                         nSurveys = nSurveys,
                         y = y)
```

Modeling building tip: When building a Stan model, I examine the raw data and make sure the Stan model matches the data. Although, this can be an iterative process, because I often discover my data requires wrangling to work with Stan. But, I start the iterative process with the data.

Now that we've simulated the data, we can use it with our model as called by `stan()`.

```
## Fit the model in stan

fitWide <- stan('../simpleModelExamples/occupancy.stan',
                data = simpleModelInput)

fitWide
```

After fitting the model, we can look at the outputs. The default `print()` option for a `stanfit` object shows a summary of the model's fit. In this case, typing `fitWdie` prints these results:

```
Inference for Stan model: occupancy.
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
psi     0.37    0.00 0.03    0.32    0.35    0.37    0.39    0.43  3587    1
p       0.60    0.00 0.02    0.56    0.58    0.60    0.61    0.63  4000    1
lp__ -796.37    0.02 1.01 -799.03 -796.74 -796.08 -795.65 -795.38  1996    1

Samples were drawn using NUTS(diag_e) at Thu Aug 30 13:03:08 2018.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
```

If these results do not make sense, I strongly recommend either reviewing/learning Stan and Bayesian statistics.

Exercise: Simulate different datasets. Using different occupancy probabilities and sample sizes. Figure out when the model works well and when it does not.

Matrix input occupancy model on logit-scale

We can take the model from the previous section and make it more numerically efficient by calculating on the logit scale. This new formulation also allows the inclusion of predictor variables, much like a logistic regression.

This model is different because we are now estimating μ_p and μ_ψ rather than these parameter directly. Note that $\mu_p = \text{logit}^{-1}(p)$ and $\mu_\psi = \text{logit}^{-1}(\psi)$. This means the parameter block for our model is different:

```
parameters {
  real muPsi;
  real muP;
}
```

The model code block is also written differently now. Key differences include

- Slightly different likelihood functions are used; and
- the `mu` parameters are now real number that can range from minus infinity to positive infinity.

However, the model is still very similar to the previously defined model.

```
model {

  real log_muPsi;
  real log1m_muPsi;

  log_muPsi   = log_inv_logit(muPsi);
  log1m_muPsi = log1m_inv_logit(muPsi);
```

```
  muP ~ normal(0, 2);
  muPsi ~ normal(0, 2);

  // likelihood
  for (r in 1:nSites) {
    if (sum(y[r]) > 0)
      target +=
    log_muPsi + bernoulli_logit_lpmf(y[r] | muP);
    else
      target +=
    log_sum_exp(log_muPsi +
            bernoulli_logit_lpmf(y[r] | muP), log1m_muPsi);
  }
}
```

We can transform the parameters to the logit scale to the probability scale in Stan using the `generated quantities` code block. This code block is complied so it is quick, but does not get run through MCMC algorithm.

```
generated quantities{
  real<lower = 0, upper = 1> p;
  real<lower = 0, upper = 1> psi;

  p   = inv_logit(muP);
  psi = inv_logit(muPsi);
}
```

This model is fit the same as previous model.

```
## Fit the model using parameters on the mu scale
fitWideMu <- stan('../simpleModelExamples/occupancyMu.stan',
                  data = simpleModelInput)

fitWideMu
```

The outputs are similar, other than having small numerical differences due to Monte Carlo variability. The output also include the `muP` and `muPsi` parameters.

```
Inference for Stan model: occupancyMu.
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
muPsi   -0.53    0.00 0.13   -0.78   -0.62   -0.53   -0.44   -0.27  3111    1
muP      0.39    0.00 0.07    0.26    0.34    0.39    0.43    0.52  3136    1
p        0.60    0.00 0.02    0.56    0.58    0.60    0.61    0.63  3139    1
psi      0.37    0.00 0.03    0.31    0.35    0.37    0.39    0.43  3125    1
lp__  -793.54    0.02 1.01 -796.18 -793.90 -793.21 -792.83 -792.56  1700    1

Samples were drawn using NUTS(diag_e) at Thu Aug 30 13:26:44 2018.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
```

Binomial input

The previous example assumed that we had the same number of observations per site. One method to side-step this assumption is to use a binomial input rather than a Bernoulli. Now our input `y` is the number of surveys or visits where detections occurred and we need a new input `k`, which corresponds to the number of surveys per sites.

We can summarize our data to be in the needed format and put it into a list for Stan.

```
## Now use summizations of the data
ySum <- apply(y, 1, sum)
k <- rep(dim(y)[2], dim(y)[1])

stanSumationData <- list(
    nSites = nSites,
    y = ySum,
    k = k)
```

This formulation also changes the `data` block of the model:

```
data {
  int<lower=0> nSites;
  int<lower=0> y[nSites];
  int<lower=0> k[nSites];
}
```

As well as the likelihood portion of the `model`:

```
// likelihood
  for (r in 1:nSites) {
    if (y[r] > 0)
      target +=
    log_muPsi + binomial_logit_lpmf(y[r] | k[r],  muP);
    else
      target +=
    log_sum_exp(log_muPsi +
            binomial_logit_lpmf(y[r] | k[r], muP), log1m_muPsi);
  }
```

Before we can fit the model, we need to sum and reformat our data our data:

```
## Now use summizations of the data
ySum <- apply(y, 1, sum)
k <- rep(dim(y)[2], dim(y)[1])

stanSumationData <- list(
    nSites = nSites,
    y = ySum,
    k = k)
```

We can then fit this model using Rstan and look at the results, which are similar to the previous results:

```
fitWideMuBinomial <- stan('../simpleModelExamples/occupancyMuBinomial.stan',
                          data = stanSumationData)

fitWideMuBinomial
```

```
Inference for Stan model: occupancyMuBinomial.
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
muPsi   -0.52    0.00 0.13   -0.78   -0.61   -0.52   -0.43   -0.27  3619    1
muP      0.39    0.00 0.07    0.26    0.35    0.39    0.43    0.52  3589    1
p        0.60    0.00 0.02    0.56    0.59    0.60    0.61    0.63  3589    1
psi      0.37    0.00 0.03    0.31    0.35    0.37    0.39    0.43  3619    1
lp__  -342.89    0.02 1.02 -345.71 -343.27 -342.57 -342.15 -341.90  1709    1

Samples were drawn using NUTS(diag_e) at Thu Aug 30 14:31:42 2018.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
```

Long format occupancy model

What if our data is in a "long format" with one row per survey or visit per site? Then, we can build a model, but it is trickier. I received help from Max Joseph on the Stan user forum who wrote this model. Before diving into the Stan code, the data needs to mutated to be the correct shape. I use the `data.table` package to do this. Most people would probably use the `tidyverse` packages. One key part of this code is creating the index of start and stopping numbers for each survey.

```
## Convert to long format
library(data.table)
yDT <- data.table(y)
yDT[ , site := factor(1:nSites)]
yDT[ , z := z]

yDTlong <- melt(yDT, id.vars = c("site", "z"),
                variable.name = "visit",
                value.name = "y")
yDTlong <- yDTlong[ order(site)]
yDTlong[ , index := 1:nrow(yDTlong)]
yDTlong[ , zObs := ifelse(sum(y) >= 1, 1, 0), by = .(site)]

yDTlongSummary <-
    yDTlong [ , .(any_seen = ifelse(sum(y) >0, 1, 0), .N), by = site]

X_psi <-  model.matrix( ~ 1, data = yDTlongSummary)
n_site <- yDTlong[ , length(unique(site))]
m_psi <- dim(X_psi)[2]

total_surveys <- dim(yDTlong)[1]

X_p <- model.matrix( ~ 1, data = yDTlong)
m_p <- dim(X_p)[2]
startStop <- yDTlong[ , .(start_idx = min(index),
                          end_idx = max(index)), by = site]
site = yDTlong[ , as.numeric(site)]

stan_d <- list(
    n_site = n_site,
    m_psi = m_psi,
```

```
    X_psi = X_psi,
    total_surveys = total_surveys,
    m_p = m_p,
    X_p = X_p,
    site = site,
    y = yDTlong[ , y],
    start_idx = startStop[ , start_idx],
    end_idx = startStop[ , end_idx],
    any_seen = yDTlongSummary[ , any_seen],
    n_survey = yDTlongSummary[ , N])
```

The code for this Stan model become more complicated. First, we cannot simply loop over raw observations. Instead, we need to "cut" out the observations for each survey (site visit). This model also includes the ability to include coefficients at different levels, but these will not be discussed until the next section. These coefficient include matrices of predictors. These are same type that are created by `model.matrix()`.

```
data {
  // site-level occupancy covariates
  int<lower = 1> nSites;
  int<lower = 1> nPsiCoef;
  matrix[nSites, nPsiCoef] Xpsi;

  // survey-level detection covariates
  int<lower = 1> totalSurveys;
  int<lower = 1> nPCoef;
  matrix[totalSurveys, nPCoef] Vp;

  // survey level information
  int<lower = 1, upper = nSites> site[totalSurveys];
  int<lower = 0, upper = 1> y[totalSurveys];
  int<lower = 0, upper = totalSurveys> startIndex[nSites];
  int<lower = 0, upper = totalSurveys> endIndex[nSites];

  // summary of whether species is known to be present at each site
  int<lower = 0, upper = 1> z[nSites];

  // number of surveys at each site
  int<lower = 0> nSurveys[nSites];
}
parameters {
  vector[nPsiCoef] beta_psi;
  vector[nPCoef] beta_p;
}
transformed parameters {
  vector[totalSurveys] logit_p = Vp * beta_p;
  vector[nSites] logit_psi = Xpsi * beta_psi;
}
model {
  vector[nSites] log_psi = log_inv_logit(logit_psi);
  vector[nSites] log1m_psi = log1m_inv_logit(logit_psi);

  beta_psi ~ normal(0, 1);
  beta_p ~ normal(0, 1);
  for (i in 1:nSites) {
```

```
    if (nSurveys[i] > 0) {
      if (z[i]) {
        // site is occupied
        target += log_psi[i]
                  + bernoulli_logit_lpmf(y[startIndex[i]:endIndex[i]] |
                                         logit_p[startIndex[i]:endIndex[i]]);
      } else {
        // site may or may not be occupied
        target += log_sum_exp(
          log_psi[i] + bernoulli_logit_lpmf(y[startIndex[i]:endIndex[i]] |
                                            logit_p[startIndex[i]:endIndex[i]]),
          log1m_psi[i]
        );
      }
    }
  }
}
```

After building this model, it may be fit like any other model. Notice the use of the `pars` option in print to only print the parameters of interest. Also, this model did not calculate probabilities because of the coefficients.

```
## Fit long form model
fitWideMuLong <- stan('../simpleModelExamples/bernoulli-occupancy.stan',
                      data= stan_d,
                      chains = 4, iter = 2000)

print(fitWideMuLong, pars = c("beta_psi", "beta_p"))
```

And the outputs from the model are similar to the other models.

```
Inference for Stan model: bernoulli-occupancy.
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
beta_psi[1] -0.52       0 0.13 -0.78 -0.60 -0.51 -0.43 -0.27  3909    1
beta_p[1]    0.39       0 0.07  0.25  0.34  0.39  0.43  0.51  3393    1

Samples were drawn using NUTS(diag_e) at Thu Aug 30 15:01:05 2018.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
```

Converting these values to probabilities with the `plogis()` function shows the results are similar to the other modeling methods:

```
plogis(-0.52)
```

```
## [1] 0.3728522
```

```
plogis(0.39)
```

```
## [1] 0.5962827
```