A simple Stan example

Christopher L. Cahill

16 January 2022

Purpose

The purpose of this tutorial is to work through a simple Stan model. Because we ultimately want to be able to run and use a fairly complex age-structured Stan model, it will pay off in spades if we understand how a simpler model is specified in Stan.

Goals

- Understand the program block sections in a `.stan` file
- Simulate and then estimate a model using fake data where truth is known
- Develop intution surrounding how Stan draws values from the priors and then calculates model predictions
- Debugging with low iterations and `print()` statements

Block sections in a `.stan` file:

Stan files are written in the Stan programming language, which is (at least to me) something of a cross between R, WINBUGS/JAGS, and C++. It is a Turing complete programming language, so anything you can do in R you generally can do in Stan (if you are enough of a masochist and indeed want to do that). It is important to recognize that variable or object declarations in Stan require you to be explicit. Thus, you need to tell Stan whether something is a real, integer, vector, matrix, or whatever.

A `.stan` file consists of program blocks. Blocks you will need to know are:

```
// program block demonstration
data{
  // read in data here -- this section is run one time per Stan run
}
transformed data {
  // transform the data here -- this section is run one time per Stan run
}
parameters {
  // declare the **estimated** parameters here
}
transformed parameters{
  // this section takes parameter estimates and data (or transformed data)
  // and transforms them for use later on
}
```

```
model{
  // this section specifies the prior(s) and likelihood terms,
  // and defines a log probability function (i.e., log posterior) of the model
}
generated quantities{
  // this section creates derived quantities based on parameters,
  // models, data, and (optionally) pseudo-random numbers.
}
```

In Stan, characters occurring after `//` are not run. Thus, these are similar to using `#` in R and can be used to comment code.

As per the comments in the above code, each of the program blocks does a specific thing.

- data{ } reads data into the .stan program.
- transformed data{ } runs calculations on those data (once).
- parameters{ } declares the ***estimated*** parameters in a Stan program.
- transformed parameters{ } takes the parameters, data, and transformed data, and calculates stuff you need for your model.
- model{ } constructs a log probability function as $ln(posterior) = ln(priors) + ln(likelihood)$.
- generated quantities{ } is only executed after you have your sampled posterior. Thus, it is useful for calculating derived quantities given your model, data, and parameters.

A crash course in Bayesian modeling using {Stan}

In an attempt to demonstrate how each of these sections can be used, and how to code something straightforward in Stan we consider the following example. The point here is to get you used to reading data into Stan from R and developing a `.stan` model.

Problem statement

Let's suppose that some poor soul's dissertation counted Gorbachev's Homing Mussels at various locations along a 30 km river. The experimental design of this survey started in the headwaters (km 0) and proceeded downstream (km 30), sampling mussels via point count surveys every 5 km. Surveys were repeated at each location five times (misery).

Your biologist friends tell you there is some industrial development planned for km 18.7. They need to determine what the expected density of critters is in that location so that they can evaluate alternative management options. However, point count surveys only occurred at kms 15 and 20. Can you help them?

Simulate some fake data

First we'll create some fake data for our toy problem. We start in R, and discuss stuff below.

```
# simulate data for Gorbachev's homing mussles
# model form:
# y_i ~ Poisson(lambda_i) # i is survey count
# E(y_i) = lambda_i
# Var(y_i) = lambda_i
# log(lambda_i) = beta_0 + beta_1*x1, where x1 is km
```

```r
# packages
library(dplyr)
```

```
##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

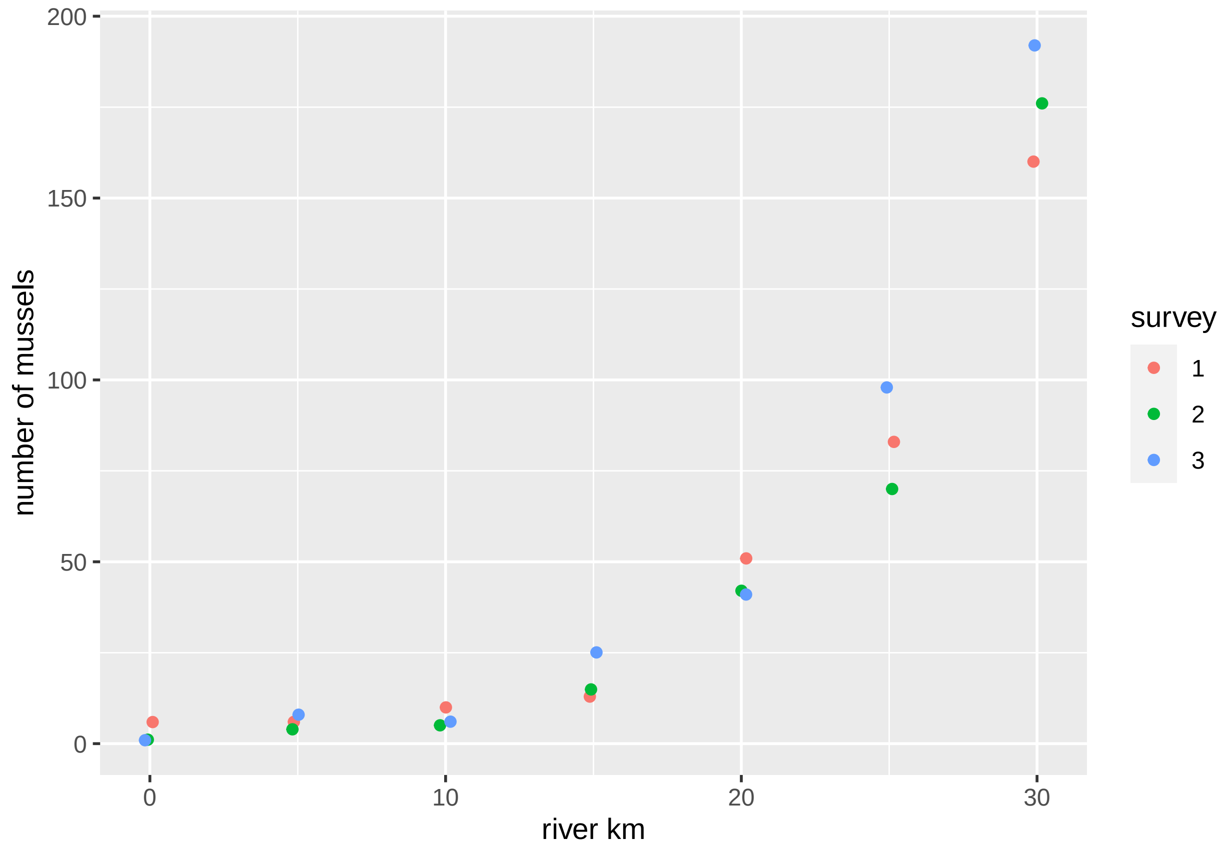
```r
library(ggplot2)
library(rstan)
```

```
## Warning: package 'rstan' was built under R version 4.1.2

## Loading required package: StanHeaders

## rstan (Version 2.21.3, GitRev: 2e1f913d3ca3)

## For execution on a local, multicore CPU with excess RAM we recommend calling
## options(mc.cores = parallel::detectCores()).
## To avoid recompilation of unchanged Stan programs, we recommend calling
## rstan_options(auto_write = TRUE)

## Do not specify '-march=native' in 'LOCAL_CPPFLAGS' or a Makevars file
```

```r
# define some (true) parameters and data for simulation
n_replicates <- 3 # 3 surveys each location along river
survey_locations <- seq(from = 0, to = 30, by = 5)
n_surveys <- length(survey_locations) * n_replicates
lambda <- 2.0
beta_0 <- log(lambda) # mean expected count in log space
beta_1 <- 0.15 # river km effect

# define the linear predictor
mu <- exp(beta_0 + beta_1*survey_locations)

# generate the "observed" data
set.seed(235) # make sure everyone gets same "random" numbers
y_i <- rpois(n_surveys, mu)

data <- data.frame(
  "y_i" = y_i, "km" = rep(survey_locations, n_replicates),
  "survey" = rep(1:n_replicates, each = length(survey_locations))
)
glimpse(data)
```

```
## Rows: 21
## Columns: 3
## $ y_i    <int> 6, 6, 10, 13, 51, 83, 160, 1, 4, 5, 15, 42, 70, 176, 1, 8, 6, 2~
## $ km     <dbl> 0, 5, 10, 15, 20, 25, 30, 0, 5, 10, 15, 20, 25, 30, 0, 5, 10, 1~
## $ survey <int> 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3
```

```
# plot the data
data %>%
  ggplot(aes(x = km, y = y_i, colour = as.factor(survey))) +
  geom_point(position = position_jitter(w = 0.2, h=0.1)) +
  xlab("river km") +
  ylab("number of mussels") +
  labs(colour = "survey")
```



We might now fit a simple generalized linear model (GLM) to those data in R via:

```
# fit a simple model in R
fit_glm = glm(data$y_i ~ data$km, family="poisson")

# examine the output
summary(fit_glm)
```

```
##
## Call:
## glm(formula = data$y_i ~ data$km, family = "poisson")
##
```

```
## Deviance Residuals:
##      Min        1Q    Median        3Q       Max
## -1.66014  -1.11018   0.00593   1.14111   2.09843
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept) 0.793230   0.134944   5.878 4.15e-09 ***
## data$km     0.145894   0.005135  28.412  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for poisson family taken to be 1)
##
##     Null deviance: 1336.245  on 20  degrees of freedom
## Residual deviance:   27.871  on 19  degrees of freedom
## AIC: 132.42
##
## Number of Fisher Scoring iterations: 4
```

Cool, so this simple script simulated some fake data for our toy problem and then estimated it using {glm} in R. Now let's write a .stan file for the exact same model.

Note that for the following code to work, you will have to save a .stan file (possible in R studio by clicking file -> new file -> stan file). I named my script `mussels.stan`.

```
data {
  int<lower=0> n_obs;      // number of observations
  int y_i[n_obs];          // counts
  real<lower=0> km[n_obs]; // covariates
  real<lower=0> my_km;     // which km to predict
}
parameters {
  real beta_0;
  real beta_1;
}
transformed parameters {
  vector[n_obs] lambda;
  for(i in 1:n_obs){
    lambda[i] = exp(beta_0 + beta_1*km[i]);
  }
}
model {
  // specify the priors
  beta_0 ~ normal(0,3);
  beta_1 ~ normal(0,3);

  // specify the likelihood
  y_i ~ poisson(lambda);

  // NOTE: Stan is doing ln(post) = ln(priors) + ln(like) in background
}
generated quantities {
 int<lower=0> y_pred = poisson_rng(exp(beta_0 + beta_1*my_km));
}
```

Now we can compile this model in R:

```r
# detect number of CPUs on current host
options(mc.cores = parallel::detectCores())

# eliminate redundant compilations
rstan::rstan_options(auto_write = TRUE)

# compile the model
m <-
  rstan::stan_model(
    "C:/Users/Chris Cahill/Documents/GitHub/managing-irf-complexity/Rmd/mussels.stan",
    verbose = F
  )
```

We then create a tagged list for stan, an inits() function, and pass this to the sampling function from {rstan}:

```r
# create a tagged list for stan
stan_data <- list("n_obs" = nrow(data),
                  "y_i" = data$y_i,
                  "km" = data$km,
                  "my_km" = 18.7 # where bios want to know density
)

inits <- function() {
  list(
    beta_0 = jitter(0, amount = 1),
    beta_1 = jitter(0, amount = 1)
  )
}

#run the model
fit <-
  rstan::sampling(
    m,
    data = stan_data,
    pars =
      c(
        "beta_0", # intercept
        "beta_1", # slope for km
        "y_pred" # predicted count for my_km
      ),
    iter = 2000,
    warmup = 1000,
    chains = 4,
    init = inits
  )
fit
```

```
## Inference for Stan model: mussels.
## 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_0    0.79    0.01 0.14    0.52    0.69    0.78    0.87    1.06   641 1.01
## beta_1    0.15    0.00 0.01    0.14    0.14    0.15    0.15    0.16   653 1.01
## y_pred   33.87    0.11 5.99   23.00   30.00   34.00   38.00   46.00  3199 1.00
## lp__   3566.68    0.03 1.02 3564.01 3566.28 3566.99 3567.41 3567.66   853 1.00
##
## Samples were drawn using NUTS(diag_e) at Sun Jan 16 21:23:40 2022.
## 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).
```

Note here that we are monitoring the parameters beta_0 and beta_1, and the derived variable y_pred.

Play around with the number of iterations and warmup period and see what happens. If you are playing with the run parameters, change the number of chains to 1 (it is easier to debug and see what is actually happening).

Voila, you've now coded a Bayesian Generalized Linear Model in Stan, and the world is now your oyster because GLMs show up all over the place (see Kery and Schaub 2012). The term y_pred is giving you the posterior predictive distribution (see Gelman et al. 2013) for the number of Gorbachev's Homing Mussels that are likely present at river km 18.7, complete with uncertainty intervals.

Order of Stan operations

A Stan file is perhaps a little counterintuitive if you are not used to Bayesian analysis.

The file reads in data and transforms it in the transformed data { } block. Note the transformed data block was not used in this example. After this, it is easiest to think of things in terms of "draws" from the prior distributions, which are specified in the model{ } block.

Thus, Stan jumps down to the model block and draws a single value for beta_0 and beta_1. Next, it jumps back up into the transformed parameter block, and uses the parameter values it has drawn along with the data and transformed data to create a model prediction. This prediction is then passed back to the model section, where the likelihood terms are calculated (likelihood calculations require data and model predictions).

Stan, in the background, then creates a log probability function as the log of the priors plus the log likelihood(s). This log probability function is evaluated repeatedly for many random draws of the priors and is then passed to an algorithm that uses magic to accept some proportion of the random draws. This algorithm also helps Stan walk toward the most likely parameter values.

Lastly, Stan will then pass these accepted parameter values to the generated quantities block.

Stan does all this magic for you, which is pretty dope (it got me a Ph.D).

If you remind me in person, I will show you a video animation to give you an idea of how the Stan algorithms are working.

Examining chains and debugging

As mentioned in the first tutorial, you can use shinystan to examine chains and a number of diagnostics for your model via:

```
library(shinystan)
launch_shinystan(fit)
```

Debugging is a bit trickier with a .stan model compared to debugging R code. A useful trick is to wrap things that you want to see in a .stan file in a `print()` function, compile that .stan model, and then run it for 10 or fewer iterations with no warm up. You also need to ensure you are only running one chain. If you do these things, whatever was wrapped in `print()` will be spit out in the console.

It is also worth noting that the Stan manual (see link below) is a remarkable collection of statistical computing advice, tips, and tricks. I highly recommend checking it out.

Questions

Dr. Geezer doesn't trust statsitics and thus hates your silly model. Dr. Geezer refuses to trust your model until you demonstrate that it adequately predicts the observed count data (a reasonable request).

1. Can you adapt the mussels.stan code to address this criticism? Note you will have to compile the .stan model each time you change it.
2. How does your model compare to the Frequentist fit using the `glm()` function?
3. Do your answers change if you change the prior sd terms?
4. It turns out the biologists mistakenly told you the industrial development was going in at km 18.7, but it is actually km 28.7. Can you adapt the code to generate a predictive distribution for mussel counts at km 28.7 instead?
5. What was the "true" expected number of mussels at km 28.7?

Resources:

Kery and Schaub. 2012. Bayesian Population Analysis Using Winbugs.

Cahill et al. 2021 CJFAS.

Gelman et al. 2013. Bayesian Data Analsyis.

https://mc-stan.org/users/documentation/

http://mc-stan.org/shinystan/