Writing Models in Stan

Why write models in Stan

- Clarity - you know exactly what the model is doing and what each parameter represents (simplifies interpretation and model assessment).
- Flexibility - you can build a truly *bespoke* model that fits your data structure perfectly and estimates exactly what you want it to (especially useful for complex models and/or assessing comparisons or summaries of parameters, e.g., differences between group means).

Why not write models in Stan

- Speed - with some small changes to the formula, `brms` can probably fit the collection of models that you are interested in, even quite complex models.
- Safety - with great power, comes great responsibilty
  - Extra important to follow a careful workflow, including prior predictive checks, fitting the model to simulated data and confirming it recovers the true parameter values, convergence diagnostics, and posterior predictive checks.

Structure of a Stan model

A stan program is divided into "blocks". Three blocks are almost always included (not technically necessary, but you're on thin ice if you don't have at least these three blocks).

1. **"data"** block: where you declare the data types, their dimensions, any restrictions (i.e. upper= or lower= , which act as checks for Stan), and their names. Each named data object represents a named element of the `list` that you will create in R.
2. **"parameters"** block: This is where you declare the parameters you want to model, their dimensions, restrictions, and name. For example, in a simple linear regression, you'll need three parameters, the intercept, a slope, and the standard deviation of the errors around the regression line.
3. **"model"** block: This is where you include any sampling statements, including the "likelihood" (the log probability calculations). This is where the distributions are defined, including the priors. You can restrict priors using upper or lower when declaring the parameters (i.e. <lower=0> to make sure a parameter is positive - think standard deviation terms)

Four blocks are optional:

- **"functions"**
  - You can define a custom function for use in the model. You probably won't use this at first, but it's very flexible. It can be a function to sample from a custom distribution, a complex calculation that needs to be run multiple times, etc.
- **"transformed data"**
  - Can be very simple calculations (e.g., defining an n-1 value) or re-ordering or grouping of data. This also may not be necessary at first.
- **"transformed parameters"**
  - Summaries of parameters and data that are necessary to calculate the likelihood, or that you want to use later in the model (in the model or generated quantities blocks), or that you want to save for accessing after sampling.
- **"generated quantities"**
  - Derived quantities based on parameters, data, and random numbers (e.g., posterior predictions or some random variate from a normal distribution with a mean and sd based on parameters).

Comments are indicated by "//" and are ignored by Stan, and vital to future-you.

The order is critical: model blocks must occur in this order:

```
functions {
}

data {
}

transformed data {
}

parameters {
}

transformed parameters {
}

model {
}

generated quantities {
}
```

Declaring the type, name, and dimensions of EVERYTHING

Every variable (including data and parameters) in a Stan model must be explicitly declared: the data type, dimension, and name. Each block begins (almost always) with declaration statements that tell the program what to expect. They are generally global (if declared in one block, they can be used in following blocks, and the names cannot be used twice), except for two conditions.

1. in the model block, they are always local. If a variable is declared in the model block, it will not be accessible outside of the model block, but it can be re-declared in a subsequent block (see variable `p` in model 11.4 below).
2. Within a loop, variables are always local. They get overwritten at each stage of the loop and so are not accessible anywhere else.

Example cmdstanr model

Here I've taken some of the code from Chapter 11 to set up model `m11.4`: the logistic regression model that estimates the treatment effects and separate intercepts for each chimpanzee (`actor`) on the `pulled_left` binary response.

Data setup, including defining a `list` in R (`dat_list`), that has three components, named to match the data variable names.

```
## R code 11.1 - taken directly from the book
library(rethinking)
data(chimpanzees)
d <- chimpanzees

## R code 11.2
d$treatment <- 1 + d$prosoc_left + 2*d$condition

## R code 11.10
# trimmed data list
dat_list <- list(
  pulled_left = d$pulled_left,
  actor = d$actor,
  treatment = as.integer(d$treatment) )
```

Then we define and fit the model with `ulam`.

```
## R code 11.11
m11.4 <- ulam(
  alist(
    pulled_left ~ dbinom( 1 , p ) ,
    logit(p) <- a[actor] + b[treatment] ,
    a[actor] ~ dnorm( 0 , 1.5 ),
    b[treatment] ~ dnorm( 0 , 0.5 )
  ) , data=dat_list , chains=4 , log_lik=TRUE ,
  messages = FALSE)
```

Stan code for the model

Then we'll use the `rethinking` function `stancode` to extract the model as it was written in Stan. The output of the function is a character vector, that we can write to a text file with a *.stan* extension.

```r
m11.4_stan_code <- stancode(m11.4)
```

```
data{
    array[504] int pulled_left;
    array[504] int treatment;
    array[504] int actor;
}
parameters{
     vector[7] a;
     vector[4] b;
}
model{
     vector[504] p;
    b ~ normal( 0 , 0.5 );
    a ~ normal( 0 , 1.5 );
    for ( i in 1:504 ) {
        p[i] = a[actor[i]] + b[treatment[i]];
        p[i] = inv_logit(p[i]);
    }
    pulled_left ~ binomial( 1 , p );
}
generated quantities{
    vector[504] log_lik;
     vector[504] p;
    for ( i in 1:504 ) {
        p[i] = a[actor[i]] + b[treatment[i]];
        p[i] = inv_logit(p[i]);
    }
    for ( i in 1:504 ) log_lik[i] = binomial_lpmf( pulled_left[i] | 1 , p[i] );
}
```

```r
cat(m11.4_stan_code,
    file = "m11.4_Stan_code.stan")
```

The model includes four blocks, each defined by the curly braces. Here's the data block with some added annotation, where the three data variables are declared (everything between the curly braces that follow the key word data.

```
data{
    array[504] int pulled_left; // the response variable 0 or 1 (1 = pulled left)
    array[504] int treatment; // the indicator for each treatment 1 through 4.
    array[504] int actor; // the indicator for each chimpanzee (1 through 7 )
}
```

Notice that these are the same names and dimensions of the `dat_list`

```
str(dat_list)
```

```
List of 3
 $ pulled_left: int [1:504] 0 1 0 0 1 1 0 0 0 0 ...
 $ actor      : int [1:504] 1 1 1 1 1 1 1 1 1 1 ...
 $ treatment  : int [1:504] 1 1 2 1 2 2 2 2 1 1 ...
```

The overthinking box on page 334 in Section 11.1.1 explains more of the components of the Stan model.

An elaboration of the Stan model.

If I was writing this Stan model for my own use, I would generalise some of the components so that I could run it on different datasets. In this case, that means defining some of the dimensions (number of observations, number of actors, number of treatments) as data in the model. First, add three components to the data list.

```
dat_list[["n_observations"]] <- length(dat_list[["pulled_left"]])
dat_list[["n_actors"]] <- max(dat_list[["actor"]]) #number of actors
dat_list[["n_treatments"]] <- max(dat_list[["treatment"]]) #number of treatments
```

Then we can re-write the Stan code so that it will work with any size of dataset and any collection of actors and treatments. We can use these dimensions in the variable declaration lines in the data, parameters, and models blocks, as well as the loops in the model and generated quantities blocks.

```
data{
    int<lower=1> n_observations; // number of observations
    int<lower=1> n_actors;// number of actors
    int<lower=1> n_treatments;// number of treatments
    array[n_observations] int pulled_left; // the response variable 0 or 1 (1 = pulled lef
    array[n_observations] int treatment; // the indicator for each treatment 1 through 4.
    array[n_observations] int actor; // the indicator for each chimpanzee (1 through 7 )
}

parameters{
     vector[n_actors] a;
     vector[n_treatments] b;
}
model{
     vector[n_observations] p;
    b ~ normal( 0 , 0.5 );
    a ~ normal( 0 , 1.5 );
    for ( i in 1:n_observations ) {
        p[i] = a[actor[i]] + b[treatment[i]];
        p[i] = inv_logit(p[i]);
    }
    pulled_left ~ binomial( 1 , p );
}
generated quantities{
    vector[n_observations] log_lik;
     vector[n_observations] p;
    for ( i in 1:n_observations ) {
        p[i] = a[actor[i]] + b[treatment[i]];
        p[i] = inv_logit(p[i]);
    }
    for ( i in 1:n_observations ) log_lik[i] = binomial_lpmf( pulled_left[i] | 1 , p[i] );
}
```

I'll write that model to another *.stan* text file to save it.

```
write("
data{
    int<lower=1> n_observations; // number of observations
    int<lower=1> n_actors;// number of actors
    int<lower=1> n_treatments;// number of treatments
    array[n_observations] int pulled_left; // the response variable 0 or 1 (1 = pulled lef
    array[n_observations] int treatment; // the indicator for each treatment 1 through 4.
    array[n_observations] int actor; // the indicator for each chimpanzee (1 through 7 )
}

parameters{
     vector[n_actors] a;
     vector[n_treatments] b;
}
model{
     vector[n_observations] p;
    b ~ normal( 0 , 0.5 );
    a ~ normal( 0 , 1.5 );
    for ( i in 1:n_observations ) {
        p[i] = a[actor[i]] + b[treatment[i]];
        p[i] = inv_logit(p[i]);
    }
    pulled_left ~ binomial( 1 , p );
}
generated quantities{
    vector[n_observations] log_lik;
     vector[n_observations] p;
    for ( i in 1:n_observations ) {
        p[i] = a[actor[i]] + b[treatment[i]];
        p[i] = inv_logit(p[i]);
    }
    for ( i in 1:n_observations ) log_lik[i] = binomial_lpmf( pulled_left[i] | 1 , p[i] );
}"
,

"m11.4_modified_Stan_code.stan")
```

HMC sampling using cmdstanr

The R package `cmdstanr` is the go-to package for using Stan from within R. It's used in the backend of the packages `rethinking` and `brms`. The package `rstan` has much of the same functionality, but because it is housed on CRAN, it runs an older version of Stan.

If you haven't already installed cmndstanr, the package has great set-up documentation and tools built into the package to make sure your set-up works.

Special note for Windows users

If you're running Stan on a Windows system, you should take advantage of the `cmdstanr` functions that run `Stan` models in Linux. This will likely cut the MCMC run-times by 30-50%.

Installing Windows Subsystem for Linux (WSL) is a small hassle, but only needs to be done once. Follow the directions at the above link.

Once the WSL installation is complete, re-install `cmdstan` using `cmdstanr::install_cmdstan(overwrite = TRUE, wsl = TRUE)`. Now, everytime you run a model using `cmdstan` (and therefore anytime you run a model using `rethinking` or `brms`), it will use the Linux installation to run Stan. It's seamless and you'll be very thankful you did it, if you ever want to fit a large model and/or model large datasets.

There are three basic steps to fitting a model in `cmdstanr`.

1 - Compile the model

```r
library(cmdstanr)
m11.4_stan <- cmdstanr::cmdstan_model("m11.4_modified_Stan_code.stan")
```

2 - Sample

```r
stan_fit <- m11.4_stan$sample(
data = dat_list,
seed = 1999, # not necessary
chains = 4, # default
parallel_chains = 4, # default
refresh = 1000, # reduces the number of messages
iter_warmup = 1000, # default
iter_sampling = 1000 # default
)
```

```
Running MCMC with 4 parallel chains...

Chain 1 Iteration:    1 / 2000 [  0%]  (Warmup)
Chain 2 Iteration:    1 / 2000 [  0%]  (Warmup)
Chain 1 Iteration: 1000 / 2000 [ 50%]  (Warmup)
Chain 1 Iteration: 1001 / 2000 [ 50%]  (Sampling)
Chain 2 Iteration: 1000 / 2000 [ 50%]  (Warmup)
Chain 2 Iteration: 1001 / 2000 [ 50%]  (Sampling)
Chain 3 Iteration:    1 / 2000 [  0%]  (Warmup)
Chain 3 Iteration: 1000 / 2000 [ 50%]  (Warmup)
Chain 3 Iteration: 1001 / 2000 [ 50%]  (Sampling)
Chain 4 Iteration:    1 / 2000 [  0%]  (Warmup)
Chain 4 Iteration: 1000 / 2000 [ 50%]  (Warmup)
Chain 4 Iteration: 1001 / 2000 [ 50%]  (Sampling)
Chain 2 Iteration: 2000 / 2000 [100%]  (Sampling)
Chain 2 finished in 1.4 seconds.
Chain 1 Iteration: 2000 / 2000 [100%]  (Sampling)
Chain 1 finished in 1.8 seconds.
Chain 3 Iteration: 2000 / 2000 [100%]  (Sampling)
Chain 3 finished in 1.5 seconds.
Chain 4 Iteration: 2000 / 2000 [100%]  (Sampling)
Chain 4 finished in 1.7 seconds.

All 4 chains finished successfully.
Mean chain execution time: 1.6 seconds.
Total execution time: 6.8 seconds.
```

3 - Summarise

The `$summary()` function built into the fitted model object provides parameter summaries as well as convergence diagnostics

```r
fit_sum <- stan_fit$summary()
kableExtra::kable(fit_sum[1:10,],
                  digits = 3)
```

variable	mean	median	sd	mad	q5	q95	rhat	ess_bulk	ess_tail
lp__	-268.381	-268.052	2.360	2.254	-272.704	-265.115	1.003	1804.284	2576.485
a[1]	-0.455	-0.453	0.329	0.331	-1.001	0.071	1.005	1446.445	2153.997
a[2]	3.872	3.812	0.757	0.732	2.738	5.217	1.002	3702.480	2458.456
a[3]	-0.752	-0.748	0.335	0.332	-1.299	-0.202	1.003	1626.247	2519.966
a[4]	-0.752	-0.743	0.340	0.336	-1.316	-0.203	1.004	1543.525	2438.514
a[5]	-0.445	-0.449	0.325	0.331	-0.973	0.079	1.003	1402.022	2506.169
a[6]	0.475	0.473	0.341	0.342	-0.086	1.048	1.003	1589.370	2060.532
a[7]	1.959	1.944	0.414	0.403	1.300	2.660	1.002	2316.522	2666.182
b[1]	-0.039	-0.035	0.283	0.282	-0.507	0.417	1.007	1307.736	1979.435
b[2]	0.481	0.483	0.285	0.279	0.024	0.953	1.005	1327.208	2088.818

With these built-in functions that use the `$` method on a cmdstanr object, you'll find the help documentation easier to access if you use the `::` syntax to explicitly define what package's documentation to search. For example `?cmdstanr::summary`, gets you right to the documentation for `summary` from the `cmdstanr` package.

Save the output from `cmdstanr`. Stan saves each iteration in a series of csv files by default stored in a temporary directory, but it doesn't load everything into the R-session. So, to ensure that everything is saved (all posterior draws and diagnostics) use the `$save_object` function with the fitted model.

```
stan_fit$save_object("saved_stan_fit.rds") #must add the .rds
```

Assessing model fit (WAIC and PSISloo)

The `cmdstanr` includes a function to calculate the psis_loo for any model that parameter name `log_lik` in the generated quantities block.

```
loo_psis <- stan_fit$loo()
loo_psis
```

```
Computed from 4000 by 504 log-likelihood matrix.

         Estimate   SE
elpd_loo   -266.2  9.5
p_loo         8.5  0.4
looic       532.4 19.0
```

```
------
MCSE of elpd_loo is 0.0.
MCSE and ESS estimates assume MCMC draws (r_eff in [1.3, 2.3]).

All Pareto k estimates are good (k < 0.7).
See help('pareto-k-diagnostic') for details.
```

But if you want to calculate waic, you'll have to extract the posterior draws and use the `loo` package functions.

```r
library(loo)
log_lik_draws <- stan_fit$draws("log_lik")
loo_waic <- waic(log_lik_draws)
loo_waic
```

```
Computed from 4000 by 504 log-likelihood matrix.

          Estimate   SE
elpd_waic   -266.2  9.5
p_waic         8.5  0.4
waic         532.4 19.0
```

Visualising the posterior

To examine trace plots, etc. there are a two options that I tend to use.

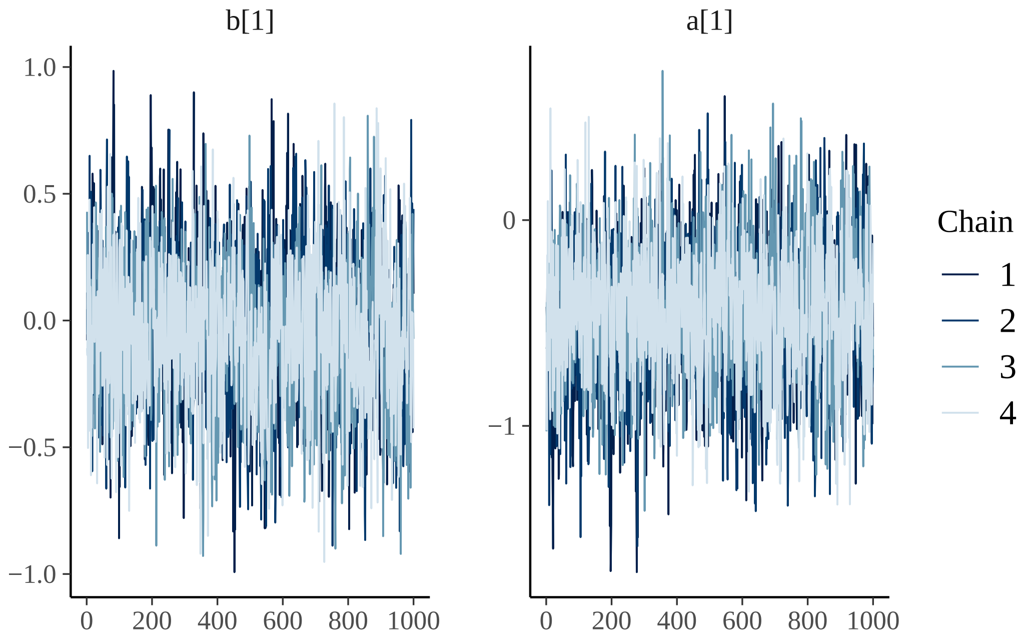
package bayesplot to visualise particular parameters

This package has many other useful plotting options, in addition to the trace plots. Note, that again, you need to extract the posterior samples using the `$draws` function on the `cmdstanr` object.

```r
#library(bayesplot)

trace <- bayesplot::mcmc_trace(stan_fit$draws(),
                               pars = c("b[1]",
                                        "a[1]"))

trace
```



To explore multiple parameters and relationships among those parameters, you can use the package `shinystan` to interactively explore the convergence, correlations among parameters, etc.

```
library(shinystan)
shinystan::launch_shinystan(stan_fit)
```

Hierarchical version example

We can modify model 11.4 to match the hierarchical version of the model in Chapter 13.

Here is the modified data object that includes the block_id grouping factor.

```
dat_list <- list(
pulled_left = d$pulled_left,
actor = d$actor,
block_id = d$block, # the new blocking id variable
treatment = as.integer(d$treatment),
n_actors = max(d$actor),
n_treatments = max(as.integer(d$treatment)),
n_blocks = max(d$block),
n_observations = nrow(d))
```

Hierarchical model code

```
write("
data{
    int<lower=1> n_observations; // number of observations
    int<lower=1> n_actors;// number of actors
    int<lower=1> n_treatments;// number of treatments
    int<lower=1> n_blocks;// number of blocks

    array[n_observations] int pulled_left; // the response variable 0 or 1 (1 = pulled lef
    array[n_observations] int treatment; // the indicator for each treatment 1 through 4.
    array[n_observations] int actor; // the indicator for each chimpanzee (1 through 7 )
    array[n_observations] int block_id; // the indicator for block
}

parameters{
     vector[n_actors] a_raw; //uncentered parameterisation
     vector[n_treatments] b;
     vector[n_blocks] g;
     real<lower=0> sigma_a; // sd of actor intercepts
     real<lower=0> sigma_g; // sd of block effects
     real a_bar;
}

// including a transformed parameters block to account for
// the uncentered parameterisation
transformed parameters {
  vector[n_actors] a; //uncentered parameterisation

  a = a_bar + sigma_a * a_raw;
  //vectorized re-scaling and centering of a_raw
  // equivalent to a ~ normal(a_bar,sigma_a);
}


model{
     vector[n_observations] p;
    b ~ normal( 0 , 0.5 );
    a_bar ~ normal( 0 , 1.5 );
    a_raw ~ std_normal(); // same as writing normal(0,1)
    sigma_a ~ exponential(1);
    sigma_g ~ exponential(1);
    g ~ normal(0,sigma_g); // centered parameterisation

    for ( i in 1:n_observations ) {
        p[i] = a[actor[i]] + g[block_id[i]] + b[treatment[i]];
        p[i] = inv_logit(p[i]);
    }
    pulled_left ~ binomial( 1 , p );
}
generated quantities{
    vector[n_observations] log_lik;
     vector[n_observations] p;
    for ( i in 1:n_observations ) {
        p[i] = a[actor[i]] + b[treatment[i]];
        p[i] = inv_logit(p[i]);
    }
    for ( i in 1:n_observations ) log_lik[i] = binomial_lpmf( pulled_left[i] | 1 , p[i] );
}"
,

"m11.4_hierarchical_modified_Stan_code.stan")
```

Compiling the hierarchical model

```
m11.4_hierarchical_stan <- cmdstanr::cmdstan_model("m11.4_hierarchical_modified_Stan_code.
```

Sampling from the hierarchical model

```r
stan_fit_hierarchical <- m11.4_hierarchical_stan$sample(
data = dat_list,
seed = 1999, # not necessary
chains = 4, # default
parallel_chains = 4, # default
refresh = 1000, # reduces the number of messages
iter_warmup = 1000, # default
iter_sampling = 1000 # default
)
```

```
Running MCMC with 4 parallel chains...

Chain 1 Iteration:    1 / 2000 [  0%]  (Warmup)
Chain 2 Iteration:    1 / 2000 [  0%]  (Warmup)
Chain 3 Iteration:    1 / 2000 [  0%]  (Warmup)
Chain 4 Iteration:    1 / 2000 [  0%]  (Warmup)
Chain 2 Iteration: 1000 / 2000 [ 50%]  (Warmup)
Chain 2 Iteration: 1001 / 2000 [ 50%]  (Sampling)
Chain 3 Iteration: 1000 / 2000 [ 50%]  (Warmup)
Chain 4 Iteration: 1000 / 2000 [ 50%]  (Warmup)
Chain 1 Iteration: 1000 / 2000 [ 50%]  (Warmup)
Chain 3 Iteration: 1001 / 2000 [ 50%]  (Sampling)
Chain 4 Iteration: 1001 / 2000 [ 50%]  (Sampling)
Chain 1 Iteration: 1001 / 2000 [ 50%]  (Sampling)
Chain 1 Iteration: 2000 / 2000 [100%]  (Sampling)
Chain 1 finished in 6.0 seconds.
Chain 4 Iteration: 2000 / 2000 [100%]  (Sampling)
Chain 4 finished in 5.9 seconds.
Chain 3 Iteration: 2000 / 2000 [100%]  (Sampling)
Chain 3 finished in 6.2 seconds.
Chain 2 Iteration: 2000 / 2000 [100%]  (Sampling)
Chain 2 finished in 6.6 seconds.

All 4 chains finished successfully.
Mean chain execution time: 6.2 seconds.
Total execution time: 10.3 seconds.


Warning: 1 of 4000 (0.0%) transitions ended with a divergence.
See https://mc-stan.org/misc/warnings for details.
```

There is a warning about a single divergent transition. it's only one out of 4000, so maybe not a big deal. We could probably avoid this completely if we used a non-centered parameterisation for the group effects as well as the actor effects. If we wanted to better understand why this divergent transition appeared, exploring the model output in `shinystan` would almost certainly help `shinystan::launch_shinystan(stan_fit_hierarchical)`

Additional resources

The main Stan website has lots of useful information.

The Stan user guide has a huge collection of example models. For example, here's the section that covers regression models.

The Stan reference manual has the technical bits (e.g., how to declare variables, conditional statements, expressions)

The Stan Forums, including the questions tagged with #Ecology.