A Crash Course in Stan

History of BUGS

The BUGS (**B**ayesian inference **U**sing **G**ibbs **S**ampling) project is concerned with flexible software for the Bayesian analysis of complex statistical models using Markov chain Monte Carlo (MCMC) methods.

It was designed to automatically utilize MCMC methods to determine the posterior distribution of Bayesian analysis.

The project began with “Classic BUGS”, which morphed into WinBUGS, and now to OpenBUGS.

- Development of BUGS is essentially dead, though relevant software can still be downloaded and installed.

BUGS is a declarative programming language.

- You tell BUGS what to do but not how to do it!
- You don't need to know how to program a Gibbs sampler or Metropolis-Hastings algorithm to solve a problem.
- You tell BUGS what the model is, and it should take care of the rest, assuming you program the model correctly.

Historically, BUGS doesn't work well for large data.

JAGS (Just Another Gibbs Sampler) is a more recent incarnation of BUGS (http://mcmc-jags.sourceforge.net/).

- Started by Martyn Plummer in 2003.
- Intended to be more like Classic BUGS than either WinBUGS or OpenBUGS.
- Cross platform!
- Written in C++, so (relatively) easy to extend functionality.
- Intended to improve certain aspects of BUGS.
- Can be run through R using the **rjags** package.
- The project is still being developed.

NIMBLE is a system for building and sharing analysis methods for statistical models, especially for hierarchical models and computationally intensive methods. (https://r-nimble.org/)

NIMBLE is built in R but compiles your models and algorithms using C++ for speed. It includes three components:

- A system for using models written in the BUGS model language as programmable objects in R.
- An initial library of algorithms for models written in BUGS, including basic MCMC, which can be used directly or can be customized from R before being compiled and run.
- A language embedded in R for programming algorithms for models, both of which are compiled through C++ code and loaded into R.

This NIMBLE project has many developers and is growing at a rapid rate.

It is still somewhat new (available in October 2014), so its history is yet to be written.

It has a chance to make a big mark in Bayesian statistics.

Stan is a state-of-the-art platform for statistical modeling and high-performance statistical computation (https://mc-stan.org/).

Users specify log density functions in Stan's probabilistic programming language and get:

- Full Bayesian statistical inference with MCMC sampling (NUTS, HMC)
- Approximate Bayesian inference with variational inference (ADVI)
- Penalized maximum likelihood estimation with optimization (L-BFGS)

Stan was officially released in August 2012.

It has quickly gained popularity and support, has a multitude of developers, and active support forum, and more.

- It currently has the most momentum of any of the software systems for Bayesian inference.

Stan is somewhat like BUGS but uses a different language for expressing models and a different sampler for sampling from their posteriors.

The **rstan** package is used to connect to Stan from within R.

- Similar packages exist for Matlab, Python, Julia, etc.

A Stan program defines a statistical model through a conditional probability function $p(\theta|y, x)$, where:

- θ is a sequence of modeled unknown values (e.g., model parameters, latent variables, missing data, future predictions)
- y is a sequence of modeled known values.
- x is a sequence of unmodeled predictors and constants (e.g., sizes, hyperparameters).

Stan programs consist of:

- Variable type declarations
- Statements

Variable types include:

- Constrained and unconstrained integers.
- Scalars, vector, and matrix types.
- (Multidimensional) arrays.
- Special variable types like a covariance or correlation matrix.

Declared variables must have a size and type.

- Bounds should also be specified if relevant.

Variables are declared in blocks corresponding to the variable's use: data, transformed data, parameters, transformed parameters, or generated quantity.

Unconstrained local variables may be declared within statement blocks.

Up to six program blocks are used to define a Stan program:

- **data** (optional)
- **transformed data** (optional)
- **parameters** (required)
- **transformed parameters** (optional)
- **model** (required)
- **generated quantities** (optional)

The **data** block reads external information used in the program:

- The sample size
- The observed responses
- The predictor variables
- Hyperparameters

The **transformed data** block preprocesses the external information for use in the program.

- This can usually be done outside the program also.

The **parameters** block defines parameters (observable and unobservable) used in the model.

- The parameters of the data distribution for which we will define priors.
- New responses we wish to sample from the posterior predictive distribution.

The **transformed parameters** block allows for parameter processing before the posterior is computed.

- We might specify a prior for $\log \sigma$, but the data distribution is parameterized using σ.

The **model** block is where the *data* and *prior* distributions are specified.

The **generated quantities** block allows us to perform processing of samples from the posterior distribution.

Variables can be declared in any model block, but their scope may be different.

- Order matters! You can't work with a variable that hasn't yet been declared.

Stan will save the posterior samples for all variables declared in parameters, transformed parameters, and generated quantities.

- Stan will also always save the iterations for `lp__`, which is the log posterior density (up to a constant).

	data	transformed data	parameters	model	generated quantities
Variable scope	global	global	global	local	global
Variables saved?	No	No	Yes	Yes	Yes

Example: Placenta Previa

Placenta previa is a condition in which the placenta of an unborn child is implanted very low in the uterus, obstructing the child from a normal vaginal delivery. An early study concerning the sex of placenta previa births in Germany found that of a total of 980 births, 437 were female. How strong is the evidence that the proportion of female births in the population of placenta previa births is less than 0.485 (the proportion in the general population)?

Data distribution: $y|\theta \sim \text{Binomial}(980, \theta)$.
Prior distribution: $\theta \sim \text{Beta}(1, 1)$

Preliminaries:

```
# Load libraries rstan (to use stan) and
# coda (to analyze results)
library(rstan)
library(coda)
```

Model:

```
stanmod = "
data {
  int<lower=1> n; // number of trials
  int<lower=0> y; // number of placenta previa
births
  real<lower=0> alpha; // prior parameters
  real<lower=0> beta; // prior parameters
}
parameters {
  real<lower=0, upper=1> theta;
}
model {
  y ~ binomial(n, theta);
  theta ~ beta(alpha, beta);
}
"
```

In the `data` statement, we declared all the constants we would use. In this case:

- `n`, the number of observations.
- `y`, the (integer) of observed counts.
- `alpha` and `beta`, the parameters of the prior distribution.
  - We could have simply done this in the model statement as `theta ~ beta(1, 1)`, if we had wanted.
  - Notice that `alpha` and `beta` can't be negative, so I declared a lower bound of 0 for each variable.

In the `parameter` statement, we declare all the parameters that have priors and/or the variables we want to predict (e.g., $\tilde{y}$). In this case:

- `theta`, the probability that the baby born for a placenta previa pregnancy is female.
- We constrain this value to be between 0 and 1.

In the `model` statement, we describe the data and prior distributions. In this case:

- `y`, the observed data has binomial distribution.
- `theta`, the probability of female placenta previa birth, has a Beta(alpha, beta) distribution.

We must specify the data as a list. The names, type, and dimensions of the objects in the list should match the variables declared in the `data` statement.

Specify the data as a list:

```
# Specify the data in R, using a list format
compatible with STAN:
stan_dat <- list(n = 980, y = 437, alpha = 1, beta
= 1)
```

Next, we compile the model and sample from the model using the `stan` function.

```
# compile the model
fit <- stan(model_code = stanmod, data =
stan_dat, iter = 1000, chains = 4)
```

We must provide:

- The model code
- The data

We might also declare:

- The desired number of samples (half is used as warmup)
- The number of chains to run

There are many other options:

- See `?stan`

Functions available through **rstan**:

- `stan_trace`: traceplot of chains
- `stan_dens`: density plot of parameters
- `stan_ac`: ACF plot of chain
- `summary`: summary of `stanfit` object
- `As.mcmc.list`: convert `stanfit` to `mcmc.list` compatible with **coda** package.

Things you should know about STAN:

1. It does not do things the exact same way as R (though there are many similarities).
   a. e.g., distribution names are slightly different (binomial vs dbinom).
   b. Distribution parameterization is consistent with BDA3 (except perhaps the Negative Binomial distribution?)
2. It can do improper priors over the designated support.
   a. e.g., $p(\log \sigma^2) \propto 1$.
3. It's error messages aren't terribly descriptive.
   a. Start with the simplest model and slowly build your way up.
4. Comments are // (like C++), not ## like R.

There are a number of R packages that work with Stan to making modeling easier or to summarize and visualize the results.

- The **brms** and **rstanarm** packages can be used to easily construct linear and generalized linear models.
- The **bayesplot** package makes it easy to visualize MCMC results and perform posterior checks.
- The **loo** package is used for model comparison.

There are likely others that I don't know about, as development is happening rapidly!