01-05: Software for applied Bayesian inference

1 - Purpose

- Learn the basic syntax of the STAN language
- Write code to elicit simple models and implement Bayesian Inference in STAN
- Use the R interface R-Stan
- Use packages for MCMC analysis

2 - Before you start

Make sure to have access to a computer with a working version of R and Rstudio and download the class material. The R script that is generated during this lesson is available here: first stan program.R. Save the R file with whatever name you want. You will also need a Stan program, available here: reg.stan. Make sure to save this file with the name "reg.stan"

Note: You can also type the code yourself (or copy and paste it) by following the instructions in this lesson .

3 - Stan

3.1 - The Basics

Stan is a probabilistic modeling language freely available at https://mc-stan.org/. We will use the Stan language as a means to implement Bayesian Inference. In particular we will perform MCMC-based Bayesian model fitting. But Stan is capable of much more than that.
There are several interfaces to the language Stan. We will use an R interface (see section 3). It is important to note that the R interface to Stan still requires that the user write a Stan program (that is not R code) and that then R code is used to deploy the Stan program.

For reference, we recommend to look into the documentation available at the Stan website. In particular, we recommend reviewing the *Stan User's Guide* and *Stan Language Reference Manual*. More advanced references can be found in the *Stan Language Functions Reference*.

A typical Stan program defines a statistical model through a conditional probability function of unknown values conditional on observed data. Unknown values include parameters, missing data, latent variables, etc. Data includes modeled observed values and unmodeled predictors. A program (see sample below) usually include variable type declarations and statements.

```
functions {
// ... function declarations and definitions ...
}
data {
// ... declarations ...
}
transformed data {
```

```
// ... declarations ... statements ...
}
parameters {
// ... declarations ...
}
transformed parameters {
// ... declarations ... statements ...
}
model {
// ... declarations ... statements ...
}
generated quantities {
// ... declarations ... statements ...
}
```

3.1.1 - Variable types

Variables can be constrained and unconstrained of several types: integer, scalar, vector, matrix and multi-dimensional arrays of other types.

Variables are declared in code blocks corresponding to the variable's use (see blow). A representation of a Stan program is shown below.

3.2 - Code Blocks in Stan

There are several code blocks in Stan. The way in which Stan builds a model depends on where (which block) a variables are defined. For instance: variables defined in the data block will be considered observed quantities and variables defined in the parameters block will be considered unknown modeled values.

3.2.1 - data

Variables declared in this block are read from an external input source such a designated R data structure.

3.2.2 - transformed data

This block is for declaring variables that don't need to be changed when running the sampling cycle of the program. The difference with the data block is that the transformed data accepts statement definitions.

3.2.3 - parameter

The variables declared in the `parameters` program block correspond directly to the variables being sampled by Stan. These are the unknowns of the model that will be sampled and for which a posterior distribution will be obtained. There are no assignment statements in this block as the parameters will be sampled depending on the model definition.

3.2.4 - transformed parameter

The `transformed parameters` program block consists of variable declarations followed by assignment statements. Variable declared as a transformed parameter will be included in the sampling output. It is legal to define in this block variables which are only a function of data or transformed data, but it may become computationally very inefficient compared to defining those quantities in the transformed data block, because transformed data quantities only need to be updated once, while transformed parameters will be updated with every iteration of the MCMC. This block is better used to define variables and quantities that are a function of variables defined in the parameters block and that will be used in the model block. For functions of parameters that are not used in the model and that will only be used for reporting, see generated quantities.

3.2.5 - model

The `model` program block consists of optional variable declarations followed by statements. The statements in the model block typically define the model. This is the block in which probability (sampling notation) statements are allowed. The statements in this block will define: 1) the prior distribution of the parameters, b) the distribution of the data (likelihood function). If the prior specification is skipped, then a flat prior is assumed by Stan. To define prior distributions and likelihood, Stan uses functions that represent probability distributions.

For instance, suppose we need to specify a Gaussian distribution with expected value mu and standard deviation sigma, for a set of N values (they could be observations or other quantities depending on their defibition in the previous blocks) stored in vector y. The following model block would be used.

```
model{
   for (i in 1:N){
       y[i]~normal(mu,sigma);
   }
}
```

Note here a few things: 1) this is NOT R code. This is Stan code, but it has an R flavor to it. 2) the function "normal" is precisely the way in which a univariate Gaussian distribution is specified in Stan. 3) the Gaussian distribution in Stan is parameterized as a function of its standard deviation. 3) other definitions could be included in this model statement as needed.

3.2.6 - generated quantity

The `generated quantities` program block is different from other blocks. The generated quantities block don't affect the sampled parameter values. The block is executed after a sample has been generated. This block is useful to define quantities such as new observations/predictions, (back) transformation of parameters for the purposes of reporting them, etc. The common characteristic of the quantities defined in this block is that they are not necessary for sampling model parameters. Thus, they can be obtained after sampling parameters from the posterior distribution to make the model fitting process more efficient.

3.3 - A first program

Let's look into a simple Stan Program to fit a simple linear regression. Where y is the response variable and x is the predictor (N observations each).

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

Observe that there are three variables defined in the data block. N is a scalar (integer) which defines the number of observations, and it is read from the R environment. x and y are vectors of length N, by default the elements of a vector are real values.

The <lower=0> notation indicates the lower bound of the parameter (or observation) space. For instance, the number of observations, N, can't be negative. And having the specification int<lower=0> N; will force the Stan interpreter to check that the parsed N from the R environment is in fact a non-negative integer. When <lower=> is used with parameters, the effect is different as explained below.

The parameters block include the definition of three variables: the y-intercept (alpha), the slope (beta) and the standard deviation (sigma). These are all scalars, real values, but sigma is constrained to be non-negative. Moreover, the specification real<lower=0> sigma; in the parameter block defines a non-negative real-valued parameter space for the standard deviation of a Gaussian distribution. This has consequences for the prior definition of such parameter (see below)

The model block includes the definition of the likelihood. Notice that the distribution of all elements in y are defined with a single line. This is short-hand and convenient. But it actually means:

```
model{
   for (i in 1:N){
       y[i]~normal(alpha+beta*x[i],sigma);
   }
}
```

It is important to note that in addition to being more concise, the vectorized notation results in a compiled model that runs faster.

An implicit assumption in this model definition is that all parameters receive a flat, improper prior over their defined domain. For the case of alpha and beta, the domain is the full set of real values, while for sigma, the domain is the set of all non-negative real values (given the use of <lower=> in its definition). If we prefer to include a proper prior for any of the parameters, we can do so in the model statement. For instance, to include a proper prior for beta, with mean 0 and standard deviation 0.001, we can specify:

```
model{
   beta~normal(0,0.001);
   for (i in 1:N){
       y[i]~normal(alpha+beta*x[i],sigma);
   }
}
```

4 - Rstan

The R package **rstan** provides an R interface to Stan. The **rstan** package allows the user to: 1) parse data to the Stan engine to fit Stan models using data stored in R and 2) access the output from the Stant engine directly from the R workspace, to perform posterior inferences.

4.1 - Install Rstan

To install Rstan, follow the instructions in this page:
https://github.com/stan-dev/rstan/wiki/RStan-Getting-Started

But before starting, make sure you have the following.
1. A computer that can run R and R studio
2. A current version of R and Rstudio

Follow the instruction in the link up to (and including) the section "Checking the C++ Toolchain".

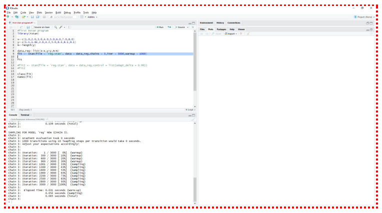
4.2 - The R stan Basic function and its use

The STAN model (for instance the code defined in section 3) has to be saved as a flat file (text only), preferably with extension .stan. For instance, suppose we save the model presented above in a file called reg.stan, which is available in the current working directory of R. Then, we only need to define the input quantities and use the stan function in R.

```
x<-c(1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0)
y<-c(1.1,1.84,2.6,4.2,5.0,6.4,8.1,9.1)
N<-length(y)
data_reg<-list(x=x,y=y,N=N)
fit <- stan(file = 'reg.stan', data = data_reg)
```

Shortcut Menu
Go to Top of Page
Print/ Save as PDF
Maximize All Images
Minimize All Images

If this code is run, R will produce a large amount of comment output (screen log) showing the pro MCMC sampling. If no warnings or errors are produced, the object fit will be created.



```
#first Rstan program
library(rstan)

x<-c(1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0)
y<-c(1.1,1.84,2.6,4.2,5.0,6.4,8.1,9.1)
N<-length(y)

data_reg<-list(x=x,y=y,N=N)
fit <- stan(file = 'reg.stan', data = data_reg,chains = 3,iter = 3000,warmup = 1000)

fit

#fit2 <- stan(file = 'reg.stan', data = data_reg,control = list(adapt_delta = 0.99))
#fit2

class(fit)
names(fit)
```

```
Chain 2:                0.109 seconds (Total)
Chain 2:

SAMPLING FOR MODEL 'reg' NOW (CHAIN 3).
Chain 3:
Chain 3: Gradient evaluation took 0 seconds
Chain 3: 1000 transitions using 10 leapfrog steps per transition would take 0 seconds.
Chain 3: Adjust your expectations accordingly!
Chain 3:
Chain 3:
Chain 3: Iteration:    1 / 3000 [  0%]  (Warmup)
Chain 3: Iteration:  300 / 3000 [ 10%]  (Warmup)
Chain 3: Iteration:  600 / 3000 [ 20%]  (Warmup)
Chain 3: Iteration:  900 / 3000 [ 30%]  (Warmup)
Chain 3: Iteration: 1001 / 3000 [ 33%]  (Sampling)
Chain 3: Iteration: 1300 / 3000 [ 43%]  (Sampling)
Chain 3: Iteration: 1600 / 3000 [ 53%]  (Sampling)
Chain 3: Iteration: 1900 / 3000 [ 63%]  (Sampling)
Chain 3: Iteration: 2200 / 3000 [ 73%]  (Sampling)
Chain 3: Iteration: 2500 / 3000 [ 83%]  (Sampling)
Chain 3: Iteration: 2800 / 3000 [ 93%]  (Sampling)
Chain 3: Iteration: 3000 / 3000 [100%]  (Sampling)
Chain 3:
Chain 3:  Elapsed Time: 0.031 seconds (Warm-up)
Chain 3:                0.054 seconds (Sampling)
Chain 3:                0.085 seconds (Total)
Chain 3:
>
```

Fig 1: Output after running the Stan function in R

A first step in every sampling-based Bayesian inference work would be to check convergence. But for this time only we will postpone that check for a later section and proceed with looking into the posterior inferences assuming that convergence is attained and that the retained sample size is adequate.

4.3 - First Bayesian Inference with Rstan

we will first check the class of the object fit. We can do that using the class() function in R:

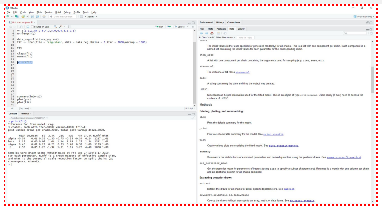
```
class(fit)
```

The class "stanfit" is actually a list that has many components inside it (those of you who want to look under the hood can try to dissect the content using the str() function in R. We will proceed and learn how to use this class. It is also recommended that you look into the associated help, to find which methods can be used with this class of objects.

```
help(stanfit)
```

The quickest way to look into a stanfit object is just to print it

```
print(fit)
```



```
y<-c(1.1,1.84,2.6,4.2,5.0,6.4,8.1,9.1)
N<-length(y)

data_reg<-list(x=x,y=y,N=N)
fit <- stan(file = 'reg.stan', data = data_reg,chains = 3,iter = 3000,warmup = 1000)

fit

class(fit)
names(fit)

print(fit)

summary(lm(y~x))
plot(y~x)
plot(fit)
```

```
> print(fit)
Inference for Stan model: reg.
3 chains, each with iter=3000; warmup=1000; thin=1;
post-warmup draws per chain=2000, total post-warmup draws=6000.

       mean se_mean   sd  2.5%   25%   50%   75% 97.5% n_eff Rhat
alpha -0.54    0.01 0.39 -1.30 -0.75 -0.55 -0.34  0.24  1217 1.01
beta   1.19    0.00 0.08  1.04  1.14  1.19  1.23  1.34  1311 1.01
sigma  0.46    0.01 0.22  0.23  0.33  0.40  0.52  1.00  1129 1.00
lp__   2.38    0.05 1.70 -1.94  1.91  3.05  3.77  4.40  1038 1.00

Samples were drawn using NUTS(diag_e) at Fri Sep 27 10:43:17 2019.
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).
>
```

R: Class 'stanfit': fitted Stan model

inits:
The initial values (either user-specified or generated randomly) for all chains. This is a list with one component per chain. Each component is a named list containing the initial values for each parameter for the corresponding chain.

stan_args:
A list with one component per chain containing the arguments used for sampling (e.g. iter, seed, etc.).

stanmodel:
The instance of S4 class stanmodel.

date:
A string containing the date and time the object was created.

.MISC:
Miscellaneous helper information used for the fitted model. This is an object of type environment. Users rarely (if ever) need to access the contents of .MISC.

Methods

Printing, plotting, and summarizing:

show
Print the default summary for the model.

print
Print a customizable summary for the model. See print.stanfit.

plot
Create various plots summarizing the fitted model. See plot,stanfit-method.

summary
Summarize the distributions of estimated parameters and derived quantities using the posterior draws. See summary,stanfit-method.

get_posterior_mean
Get the posterior mean for parameters of interest (using pars to specify a subset of parameters). Returned is a matrix with one column per chain and an additional column for all chains combined.

Extracting posterior draws:

extract
Extract the draws for all chains for all (or specified) parameters. See extract.

as.array, as.matrix, as.data.frame
Coerce the draws (without warmup) to an array, matrix or data frame. See as.array.stanfit.

Fig 2: Print the content of the stanfit object to obtain a summary of the posterior distribution of the model parameters.

A table is printed with a row per parameter. Notice that lp__ is added. This term is proportional to the log-likelihood and we will revisit this in the model comparison class.

Also, pay attention to the columns of the output table. These represent a summary of the posterior distribution, for instance; mean and sd are the arithmetic mean and standard deviation of the posterior. Five percentiles are also included. An important column is n_eff, this is the efective sample size. When this number is close to the actual sample size (total post-warmup draws) it indicates that there is a very low autocorrelation in the values sampled for the posterior distribution. In our case, the total sample size is 6000 and the efective sample size is around 1200, indicating that there will be some autocorrelation in the sampled parameters. We will come back to this in the next section.

5 - Convergence diagnostics and posterior inference

5.1 - Coda and related packages

We will use several packages for convergence diagnostics with the stan output. In particular we will use the packages coda and mcmcplots.

```
library(mcmcplots) #automatically loads coda package
as_list<-As.mcmc.list(fit)
mcmcplot(as_list)
```

In line 2 we extract the samples from the posterior distribution into an mcmc.list object, which is a standard object used by the coda package to store samples from a posterior distribution. In line 3, a quick web-report is created including graphics summaries for all the parameters.

Fig 3: Example of mcmcplot graphic summary.

The included graphical summary in this case includes a density plot (top left), a trace plot (bottom), autocorrelation plot (top right) and a running mean plot (center left).

The most important plots here are the trace plot, where we see that all chains seem to be sampling from similar sample spaces and the autocorrelation plot (which in this case indicates a mild autocorrelation, which virtually vanishes for samples obtained farther than five iterations apart. The autocorrelation, however explains the differences between total sample size and effective sample size mentioned before.

5.1.1 - First convergence diagnostics

For convergence diagnostics in this case we will look back into the report obtained in the print summary (*Figure 2*), in particular to the column Rhat. When this statistic is close to 1.00, it is an indication that the three mcmc chains are sampling from the same posterior distribution (see convergence diagnostics lesson).

Another convergence diagnostic is Geweke's criteria:

```
geweke.diag(as_list)
```

This criteria resembles a z-test between the first part of the chain and the second part of the chain. Absolute values of the Geweke Statistic that are below 2.0 indicates no differences between the first and the second part of the chain (which is a hint for within-chain convergence). In this case, all the statistics are well below 2.0.

6 - Recap

In this lesson we learned:
1. Basics of the Stan modeling language
2. How to perform model fit using rstan
3. How to extract posterior inferences and perform basic convergence criteria

We accomplished using a simple example (linear regression)

7 - Practice activity - I need to finish the dropbox for this one.

Please complete the following activity and submit the material in here. You should submit two files: 1) r script necessary to reproduce all results. 2) a report (word or pdf accepted) with the answers to the questions below.

1. Modify the Stan program and obtain the posterior distribution of the error variance (not just standard deviation). Hint: use the generated quantities block.
2. look into the help of the rstan package and perform model fit with a single mcmc chain, burn in of 2000 and retain 6000 iterations without any thinning.
3. Perform convergence diagnostics and compare results (including effective sample size) to the ones obtained in our first run.
4. Include in the report the graphical and numerical diagnostics used and interpret the results. I list a few questions below for you to use as guidance in crafting the report
   A. Is the burn-in appropriate? Could we've done less?
   B. Is the number of retained iterations appropriate to obtain posterior inferences?
   C. Use the results form our first run as a basis for comparison.