Introduction to Bayesian inference using Rstan: practical 1 solutions

The aim of this practical is to provide an introduction to the syntax and structure of the Stan programming language and to demonstrate some of the efficiency gains that can be achieved by using vectorized operations.

1 Getting started

1.1 Files in Rstudio

- Open `Rstudio`
- Open a new text file:

  ```
  File -> New File -> Text File
  ```

  and type the following Stan programme:

```
data {
  int<lower=1> N;
  real<lower=0,upper=1> p;
}
model {
}
generated quantities {
  int x[N];
  for(n in 1:N) {
    x[n] = bernoulli_rng(p);
  }
}
```

  Save the text file as `simulation.stan`.

- Open a new R script:

  ```
  File -> New File -> R Script
  ```

  and type the following sequence of commands, omitting the comments if you wish:

```
## Load the rstan package:
library(rstan)
## Allow rstan to exploit parallel computation:
rstan_options(auto_write = TRUE)
options(mc.cores = parallel::detectCores())
## Set up some constants to pass to Stan:
constants = list(N=30, p=0.8)
## Compile and run Stan programme:
output = stan(file="simulation.stan", data=constants, iter=1,
              chains=1, algorithm="Fixed_param")
## Extract simulated data:
out = as.matrix(output)
dim(out)
head(out)
```

```
## Format as vector, removing the log posterior (i.e. "__lp"):
x = out[1, -length(out)]
## Plot data:
barplot(table(x))
```

Save the R script. Note that Rstudio will correctly add the file extension `.R`. We can run our R script by sending the commands to the R console. For example, to send one line at a time, press `Ctrl + Enter` at the end of each line in turn.

1.2 *Course R package*

Installing the course R package is straightforward. First install `drat`:

```
install.packages("drat")
```

Then

```
drat::addRepo("jr-packages")
install.packages("jrRstan")
```

This R package contains copies of the practicals, solutions and data sets for the course in addition to some helpful functions that we will use. To load the package, use

```
library("jrRstan")
```

2 *Simulating data*

In the Stan programme above, we are not attempting to carry out any inference. Instead we're using the `generated quantities` block to sample some data; in this case a random sample of size 30 from the Bernoulli distribution with probability 0.8. In other words a sequence of 30 0s and 1s where the probability of getting a 1 is 0.8. When we simply want to use Stan to simulate data in this way, notice that the (required) model block is empty. We will learn about the `stan` function in Chapter 4, but for now, we simply note that the `file` argument should be the name of the text file where the Stan programme is saved and, if the programme contains a `data` block, the `data` argument should be a `list` with a named element for each variable declared. When it is being used to simulate data, we generally set `algorithm="Fixed_param"`, `iter=1` and `chains=1`.

- What happens if you try to pass a value for `N` or `p` that violates the constraints in the `data` block? For example, what happens if you set:

```
faulty_constants = list(N=30, p=10)
stan(file="simulation.stan", data=faulty_constants, iter=1,
             chains=1, algorithm="Fixed_param")

## failed to create the sampler; sampling not done
```

- Add a `transformed data` block and include a print statement to check whether the constants, `N` and `p`, have been correctly passed.

```
// Modified Stan programme, saved in simulation.stan:
data {
  int<lower=1> N;
  real<lower=0,upper=1> p;
}
transformed data {
  print("N = ", N, ", p = ", p);
}
model {
}
generated quantities {
  int x[N];
  for(n in 1:N) {
    x[n] = bernoulli_rng(p);
  }
}
```

```
## Compile and run modified Stan programme:
output = stan(file="simulation.stan", data=constants, iter=1,
              chains=1, algorithm="Fixed_param")

## N = 30, p = 0.8

## trying deprecated constructor; please alert package maintainer

##
## SAMPLING FOR MODEL 'simulation' NOW (CHAIN 1).
##
## Chain 1, Iteration: 1 / 1 [100%]  (Sampling)
##  Elapsed Time: 2e-06 seconds (Warm-up)
##                1.2e-05 seconds (Sampling)
##                1.4e-05 seconds (Total)
```

- Rather than passing the constants `N` and `p` through the `data` block, we can alternatively omit the `data` block and declare and assign values to these variables in the `transformed data` block. We can then omit the `data` argument in the call to the `stan` function in R. Write a new Stan programme of this form called `simulation_v2.stan` then compile and run the programme in R.

```
// New Stan programme, saved in simulation_v2.stan:
transformed data {
  int<lower=1> N = 30;
  real<lower=0,upper=1> p = 0.8;
}
model {
}
generated quantities {
  int x[N];
  for(n in 1:N) {
    x[n] = bernoulli_rng(p);
  }
}
```

```
## Compile and run new Stan programme:
output_v2 = stan(file="simulation_v2.stan", iter=1, chains=1,
                 algorithm="Fixed_param")
```

- In R, we can generate uniform random numbers using the `runif` function. For example:

```
runif(5, min=-3, max=3)

## [1] -0.1353615 -1.2775875  0.5588464  2.2190782  1.4661580
```

Use the `lookup` function to find a corresponding function in Stan.

```
lookup("runif")

##     StanFunction                 Arguments ReturnType Page
## 691  uniform_rng (reals alpha, reals beta)          R  138
```

- Augment your programme `simulation_v2.stan` to additionally simulate 30 uniform random numbers in a vector `z` with lower bound -3 and upper bound 3. Call the new programme `simulation_v3.stan`. Compile and run it in R and check that the output looks reasonable.

```
// Modified Stan programme, saved in simulation_v3.stan:
transformed data {
  int<lower=1> N = 30;
  real<lower=0,upper=1> p = 0.8;
  real a = -3;
  real b = 3;
}
model {
}
generated quantities {
  int x[N];
  vector[N] z;
  for(n in 1:N) {
    x[n] = bernoulli_rng(p);
    z[n] = uniform_rng(a, b);
  }
}
```

```
## Compile and run modified Stan programme:
output_v3 = stan(file="simulation_v3.stan", iter=1, chains=1,
                 algorithm="Fixed_param")
```

```
## Extract simulated data:
out_v3 = as.matrix(output_v3)
colnames(out_v3)

##  [1] "x[1]"  "x[2]"  "x[3]"  "x[4]"  "x[5]"  "x[6]"  "x[7]"  "x[8]"
##  [9] "x[9]"  "x[10]" "x[11]" "x[12]" "x[13]" "x[14]" "x[15]" "x[16]"
## [17] "x[17]" "x[18]" "x[19]" "x[20]" "x[21]" "x[22]" "x[23]" "x[24]"
## [25] "x[25]" "x[26]" "x[27]" "x[28]" "x[29]" "x[30]" "z[1]"  "z[2]"
```

```
## [33] "z[3]"  "z[4]"  "z[5]"  "z[6]"  "z[7]"  "z[8]"  "z[9]"  "z[10]"
## [41] "z[11]" "z[12]" "z[13]" "z[14]" "z[15]" "z[16]" "z[17]" "z[18]"
## [49] "z[19]" "z[20]" "z[21]" "z[22]" "z[23]" "z[24]" "z[25]" "z[26]"
## [57] "z[27]" "z[28]" "z[29]" "z[30]" "lp__"

x = out_v3[1, 1:30]
z = out_v3[1, 31:60]
## Plot data:
par(mfrow=c(1,2))
barplot(table(x), main="Barplot of x")
hist(z)
par(mfrow=c(1,1))
```

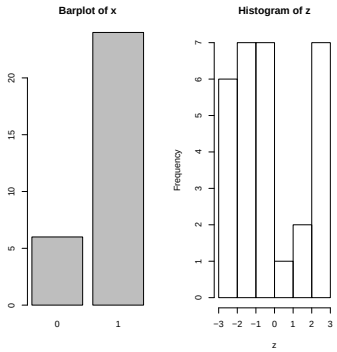

Figure 1: Simulated data.

- Modify `simulation_v3.stan`, saving the file as `simulation_v4.stan`, so that instead of representing `x` and `z` as separate objects, you represent them as a length-2 array of vectors called `xz`.

```
/* The generated quantities block of simulation_v3.stan
   should change as follows, with the result saved in
   simulation_v4.stan: */
generated quantities {
  vector[N] xz[2];
  for(n in 1:N) {
    xz[1][n] = bernoulli_rng(p);
    xz[2][n] = uniform_rng(a, b);
  }
}
```

- Modify `simulation_v4.stan`, saving the file as `simulation_v5.stan`, to additionally simulate 30 normal random variables in a vector `y` where $y[n]$ has mean `2.5 * x[n] + 1.5 * z[n]` and standard deviation 0.1. Compile and run your Stan programme in R and check that the output looks reasonable.

```
/* The generated quantities block of simulation_v4.stan
   should change as follows, with the result saved in
   simulation_v5.stan: */
generated quantities {
  vector[N] y;
  vector[N] xz[2];
  for(n in 1:N) {
    xz[1][n] = bernoulli_rng(p);
    xz[2][n] = uniform_rng(a, b);
    y[n] = normal_rng(2.5 * xz[1][n] + 1.5 * xz[2][n], 0.1);
  }
}
```

```
## Compile and run Stan programme:
output_v5 = stan(file="simulation_v5.stan", iter=1, chains=1,
                 algorithm="Fixed_param")
```

```
## Extract simulated data:
out_v5 = as.matrix(output_v5)
colnames(out_v5)
```

```
##  [1] "y[1]"     "y[2]"     "y[3]"     "y[4]"     "y[5]"     "y[6]"
##  [7] "y[7]"     "y[8]"     "y[9]"     "y[10]"    "y[11]"    "y[12]"
## [13] "y[13]"    "y[14]"    "y[15]"    "y[16]"    "y[17]"    "y[18]"
## [19] "y[19]"    "y[20]"    "y[21]"    "y[22]"    "y[23]"    "y[24]"
## [25] "y[25]"    "y[26]"    "y[27]"    "y[28]"    "y[29]"    "y[30]"
## [31] "xz[1,1]"  "xz[2,1]"  "xz[1,2]"  "xz[2,2]"  "xz[1,3]"  "xz[2,3]"
## [37] "xz[1,4]"  "xz[2,4]"  "xz[1,5]"  "xz[2,5]"  "xz[1,6]"  "xz[2,6]"
## [43] "xz[1,7]"  "xz[2,7]"  "xz[1,8]"  "xz[2,8]"  "xz[1,9]"  "xz[2,9]"
## [49] "xz[1,10]" "xz[2,10]" "xz[1,11]" "xz[2,11]" "xz[1,12]" "xz[2,12]"
## [55] "xz[1,13]" "xz[2,13]" "xz[1,14]" "xz[2,14]" "xz[1,15]" "xz[2,15]"
## [61] "xz[1,16]" "xz[2,16]" "xz[1,17]" "xz[2,17]" "xz[1,18]" "xz[2,18]"
## [67] "xz[1,19]" "xz[2,19]" "xz[1,20]" "xz[2,20]" "xz[1,21]" "xz[2,21]"
## [73] "xz[1,22]" "xz[2,22]" "xz[1,23]" "xz[2,23]" "xz[1,24]" "xz[2,24]"
## [79] "xz[1,25]" "xz[2,25]" "xz[1,26]" "xz[2,26]" "xz[1,27]" "xz[2,27]"
## [85] "xz[1,28]" "xz[2,28]" "xz[1,29]" "xz[2,29]" "xz[1,30]" "xz[2,30]"
## [91] "lp__"
```

```
y = out_v5[1, 1:30]
x = out_v5[1, seq(31, 90, 2)]
z = out_v5[1, seq(32, 90, 2)]
## Plot:
plot(z, y, main="Scatter plot of z vs y")
```

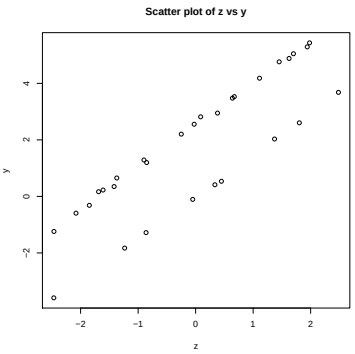

Figure 2: Scatter plot of simulated data.

- If you wanted your Stan programme to return only `y` and not `xz` how could you modify your code?

```
/* The generated quantities block of simulation_v5.stan
   should become: */
generated quantities {
  vector[N] y;
  for(n in 1:N) {
    /* Define xz in a block so that it becomes a
       local variable that will not be printed */
    real xz[2];
    xz[1] = bernoulli_rng(p);
    xz[2] = uniform_rng(a, b);
    y[n] = normal_rng(2.5 * xz[1] + 1.5 * xz[2], 0.1);
  }
}
```

3 *Vectorized operations*

In this part of the practical we will investigate the efficiency gains we can realise through the vectorized operations discussed in Chapter 3.

- Copy the Stan programme from Figure 3.1 into a text file and save it as `figure3_1.stan`.
- Create a revised version of this programme which exploits the vectorization tricks described later in the Chapter. Save it as `figure3_1_v2.stan`. Note that in lectures we considered optimising only the `model` block. It is also possible to make the `functions` and `generated quantities` blocks more efficient.

```
functions {
  real deviance(vector y, matrix X, vector beta, real sigma_sq) {
    vector[num_elements(y)] eta = X * beta; /* Single matrix-vector
                                               calculation */
    real dev = (-2) * normal_lpdf(y | eta, sqrt(sigma_sq)); /*
                                  Vectorized form of the normal
                                  probability function */
    return dev;
  }
}
data {
  int<lower=1> K;           // No. columns in design matrix
  int<lower=1> N;           // Sample size
  matrix[N, K] X;           // Design matrix
  vector[N] y;              // Response variables
  row_vector[K] x_pred;     // Design matrix for prediction
  real m_beta[K];           // Prior means for beta[k]
  real m_sigma_sq;          // Prior mean for log(sigma_sq)
  real<lower=0> s_beta[K];  // Prior std. devs for beta[k]
  real<lower=0> s_sigma_sq; // Prior std. dev for log(sigma_sq)
}
parameters {
  real<lower=0> sigma_sq;
  vector[K] beta;
}
model {
  // Likelihood:
  vector[N] eta = X * beta; // Single matrix-vector calculation
  y ~ normal(eta, sqrt(sigma_sq)); /* Vectorized form of the normal
                                      probability function */
  // Prior:
  beta ~ normal(m_beta, s_beta);  /* Vectorized form of the normal
                                      probability function */
  sigma_sq ~ lognormal(m_sigma_sq, s_sigma_sq);
}
generated quantities {
  real y_pred;  // Predicted response
  real dev; // Deviance
  // Sample from predictive distribution:
  {
    real eta_pred = x_pred * beta; /* Single matrix-vector
                                      calculation */
    y_pred = normal_rng(eta_pred, sqrt(sigma_sq));
  }
  dev = deviance(y, X, beta, sigma_sq);
}
```

- Load the data to pass to the Stan function. For convenience in this practical, the `list` that we will construct in Chapter 4 has been saved in a data set called `practical1_data` which can be loaded through:

```
## Load data to pass to stan function:
data(practical1_data)
```

We can now compile and run each programme by executing the following code:

```
## Run the programme from Figure 3.1:
fig3_1 = stan("figure3_1.stan", data=practical1_data)
## Run the revised programme:
fig3_1_v2 = stan("figure3_1_v2.stan", data=practical1_data)
```

We will consider the `stan` function and its usage in Chapter 4. For now, simply make a note of the time taken to run each programme. (You can access this by passing the object returned by the `stan` function to the `get_elapsed_time` function). How much speed-up do you gain by using the revised programme?

```
## I get a substantial speed up on my machine:
get_elapsed_time(fig3_1)

##           warmup   sample
## chain:1 2.961934 0.716229
## chain:2 3.346058 0.828066
## chain:3 2.763397 0.718156
## chain:4 2.816098 0.669206

get_elapsed_time(fig3_1_v2)

##           warmup   sample
## chain:1 0.462087 0.157590
## chain:2 0.663272 0.153040
## chain:3 0.857258 0.211019
## chain:4 0.715843 0.146785
```

Solutions

Solutions are available as a vignette:

```
library("jrRstan")
vignette("solutions1", package="jrRstan")
```