An introduction to Stan

A programming language for Bayesian modeling

Trung Dung Tran and Emmanuel Lesaffre

Rijksinstituut voor Volksgezondheid en Milieu
The Netherlands

June 2018

Objectives

- Know Stan and intuitively understand its algorithm(s).
- Learn Stan programming language.
- Write a simple Bayesian model using Stan language and run in R via Rstan interface.
- Do posterior inferences.
- Know some differences between Stan and BUGS.
- Know where to ask for help.

Topic of today

- This presentation focuses on practical aspect of the language in relation with R.
- Theoretical aspect is very briefly mentioned.
- Stan can be used for solving differential equations, but today is only for statistical modeling.

Main reference

- Stan Development Team (2017) Stan Modeling Language: Users Guide and Reference Manual. Version 2.17.3.

Outline

What is Stan

- A high level and imperative probabilistic programming language.
  - Imperative: order is a matter.
  - Probabilistic programming language: C++ behind the scene.
- Users are easily to specify statistical models.
- Stan uses the No-U-Turn sampler as default, an adaptive variant of Hamiltonian Monte Carlo (HMC) algorithm (Hoffman and Gelman, 2014).

Motivation

- Efficiently sample from high-dimensional parameter space (Hoffman and Gelman, 2014).
- A responsible and supporting team: http://mc-stan.org/
- An active discussion group: http://discourse.mc-stan.org/
- An extensive manual (stan-reference-2.17.0.pdf, download from http://mc-stan.org/).

Algorithm(s)

HMC algorithm (Neal, 2012)

- Target density: $p(\theta|y)$ (for short $p(\theta)$) for parameters θ.
- Auxiliary momentum variable: $\rho \sim MultiNormal(0, \Sigma)$.
- Joint density: $p(\rho, \theta) = p(\rho|\theta)p(\theta)$.
- Define a Hamiltonian:

$$\begin{aligned} H(\rho, \theta) &= -\log p(\rho, \theta) = -\log p(\rho|\theta) - \log p(\theta) \\ &= T(\rho|\theta) + V(\theta) = \text{"kinetic energy"} + \text{"potential energy"}. \end{aligned}$$

- Generating transitions from the current θ:
  - $\rho \sim MultiNormal(0, \Sigma)$
  - The pair (ρ, θ) "uniquely" evolves via Hamilton's equations:

$$\begin{aligned} \frac{d\rho}{dt} &= -\frac{\partial V}{\partial \theta} \\ \frac{d\theta}{dt} &= +\frac{\partial T}{\partial \rho} \end{aligned}$$

Algorithm(s)

- Unfortunately, we cannot solve the system analytically.
- Numerical approximation for the Hamiltonian systems: Leapfrog integrator.
- Correcting for numerical errors: Metropolis acceptance step.
- The probability of accepting the proposal $(\rho^*; \theta^*)$ generated by transitioning from (ρ, θ) is

$$\min(1, exp(H(\rho, \theta) - H(\rho^*; \theta^*))).$$

- In summary, HMC algorithm steps:
  - Start at an initial state for θ.
  - Propose a new θ: at a given iteration, sample a new ρ, propose θ using leapfrog integrator with discretization time ε and number of steps L.
  - Acceptance step: updating to the new state or keeping the current one.

Algorithm(s)

No-U-Turn sampler (NUTS)

- Discretization time ε (step size), number of steps L, and Σ are required in HMC algorithm.
- No-U-Turn sampler
  - automatically optimizes ε
  - dynamically adapts L, and
  - estimates Σ.
- Optimizing ε uses dual averaging (Nesterov, 2009), depending on the following.

parameter	*description*	*constraint*	*default*
δ	target Metropolis acceptance rate	$\delta \in [0, 1]$	0.80
γ	adaptation regularization scale	$\gamma > 0$	0.05
κ	adaptation relaxation exponent	$\kappa > 0$	0.75
t_0	adaptation iteration offset	$t_0 > 0$	10

- Sometimes δ needs to be changed (more precisely increased).

What do the user and Stan do

- User:
  - Define $p(\theta|y)$ using Stan language.
  - Choose the algorithm (NUTS is default).
  - (Optional) initialize initial values.
  - Occasionally, adjust default values of the algorithm parameters (when observing pathologies (Stan often highlights)).
  - Select number of chains and iterations (including warm-up).
- Machine:
  - Initialize the initial values (if not done yet).
  - Optimize the algorithm through the warm-up period.
  - Sample the parameters.

An example: A linear regression model

(i) Fit a simple linear regression model with y as the response, x as a covariate, and N as the sample size. (ii) After that check the model using PPC with the skewness discrepancy

$$\frac{\sum_{i=1}^{N}((y_i - \overline{y})/s)^3}{N}$$

where $\overline{y}$ and s are sample mean and standard deviation resp.

Solution (i):

- We are given the data: $y, x,$ and N.
- We need to estimate the parameters, $\boldsymbol{\beta}$ and σ, using the following model:

$$y_i \sim N(\beta_1 + \beta_2 \times x_i, \sigma^2) \quad \forall i = 1, ..., N$$

The linear model

Stan program

```
data {
    int<lower=1> N; // sample size
    vector[N] y; // response
    vector[N] x; // covariate
}

parameters {
    vector[2] beta; // regression coefficients
    real<lower=0> sigma; // SD
}

model {
    beta ~ normal(0, 1000); // prior
    sigma ~ cauchy(0, 5); // half-cauchy implicitly
    for (i in 1:N) {
        y[i] ~ normal(beta[1] + beta[2] * x[i], sigma);
    }
}
```

Blocks

- Blocks

Block	Example
data	Observed responses, covariates
transformed data	Sample mean
parameters	Parameters, missing values
transformed parameters	SD as the squared root of a variance
model	$y \sim N(\mu, \sigma)$
generated quantities	Replicate of the response

- The components are written in that order.
- Sometimes, a functions block defining user's functions is added before the data block.

Working with R

- Call Stan.

```
library(rstan)
options(mc.cores = parallel::detectCores())
rstan_options(auto_write = TRUE)
```

- Prepare data for the data block.

```
data1a = list(N = N, x = x, y=y)
```

- (Optionally) provide initial values.
- Run Stan.

```
model1a = stan(file="model1a.stan", data = data1a,
iter=1000, chains = 4, pars = c('beta', 'sigma'))
```

- Occasionally, adjust algorithm parameters.
- Process output (in R or using Shiny).
- See rstan document (rstan.pdf) for more detail.

Exercise 1: Correct the order

The folder Ex1 contains a Stan program where the code is correct except the order of the blocks.

- Run R and get the meaning of the warning(s).

```
SYNTAX ERROR, MESSAGE(S) FROM PARSER:

variable "beta" does not exist.
  error in 'model247c65b2422e_model_Ex1' at line 4, column 8
  ------------------------------------------------
    2:
    3: model {
    4:         beta ~ normal(0, 1000);          // prior
               ^
    5:         sigma ~ cauchy(0, 5);            // half-cauchy implicitly
  ------------------------------------------------

Error in stanc(file = file, model_code = model_code, model_name = model_name,  :
  failed to parse Stan model 'model_Ex1' due to the above error.
```

- Open model_Ex1.stan. Press Ctrl + G and enter the line number to locate the line with error.
- Why is beta in the code (line 13) but Stan says it does not exist?

Exercise 1: Correct the order

- Even beta does exist in line 13, prior to line 4, no definition or declaration of beta exists.
- Put the parameters block before the model block.
- Now N does not exist before its first use.
- Adjust the code following the message(s) until being correct.

Data block

The model:

$$y_i \sim N(\beta_1 + \beta_2 \times x_i, \sigma^2) \quad \forall i = 1, ..., N$$

- Data block:

```
data {
     int<lower=1> N;              // sample size
     vector[N] y;                 // response
     vector[N] x;                 // covariate
}
```

  - Only declaration, no statement.
  - Values are assigned from the interface, e.g. Rstan.
  - Constraints for checking.

Transformed data block

Computation of the sample mean and standard deviation: $\overline{y}$ and s.

- Transformed data block:

```
transformed data {
    real mean_y;
    real<lower=0> sd_y;
    mean_y = mean(y);
    sd_y = sd(y);
}
```

  - Declaration and definition of 'constants'.
  - Not reading from an external source.
  - Assignments are only allowed to variables declared in this block.
  - Constraints for checking.

Parameters block

The model:

$$y_i \sim N(\beta_1 + \beta_2 \times x_i, \sigma^2) \quad \forall i = 1, ..., N$$

- Parameters block:

```
parameters {
    vector[2] beta;           // regression coefficients
    real<lower=0> sigma;      // SD
}
```

  - The parameters are being sampled by Stan.
  - Only declaration, no statement.
  - Variable constraints are very important.
  - To sample efficiently, any parameter values satisfying the constraints must have support in the model block (i.e., must have non-zero posterior density).
  - Parameter $\leftrightarrow$ unconstrained variable behind the scene.

Transformed parameters block

The model:

$$y_i \sim N(\beta_1 + \beta_2 \times x_i, \sigma^2) \quad \forall i = 1, ..., N$$

If we want to monitor the variance, we could do (but not efficient):

- Transformed parameters block:

```
transformed parameters {
    real<lower=0> sigmasq;      // Variance
    sigmasq = sigma^2;
}
```

  - Declaration followed by statements.
  - Part of the output.
  - Constraints for checking.
  - Not efficient to declare here if you do not use in the model block.

Model block

The model:

$$y_i \sim N(\beta_1 + \beta_2 \times x_i, \sigma^2) \quad \forall i = 1, ..., N$$

- Model block:

```
model {
    beta ~ normal(0, 1000);     // prior
    sigma ~ cauchy(0, 5);       // half-cauchy implicitly
    for (i in 1:N) {
        y[i] ~ normal(beta[1] + beta[2] * x[i], sigma);
    }
}
```

  - Defining $p(\theta|y)$: prior and likelihood.
  - Optional variable declarations followed by statements.
  - The declared variables are local, not part of the output.
  - If we do not specify a prior, flat prior is automatically used.

Generated quantities block

The skewness discrepancy for replicate data $\frac{\sum_{i=1}^{N}((y_i^{rep} - \overline{y^{rep}})/s^{rep})^3}{N}$.

- Generated quantities block:

```
generated quantities {
    vector[N] rep_y;
    real skewness_test;
    for (i in 1:N) {
        rep_y[i] = normal_rng(beta[1] + beta[2] * x[i], sigma);
    }
    skewness_test = step(skewness(rep_y) - skewness(y));
}
```

  - Applications of posterior inference:
    - ★ Generate simulated data for model checking,
    - ★ generate predictions for new data,
    - ★ calculate posterior probabilities,
    - ★ calculate (log) likelihoods, deviances, etc. for model comparison,
    - ★ ...

Generated quantities block

- Generated quantities block:
  - Can use global variables declared in earlier blocks.
  - When possible, define here is more efficient than in the transformed parameters block.
  - Generated quantities are parts of the output.
  - Constraints must succeed or sampling will be halted altogether because it is too late to reject a draw at the point the generated quantities block is evaluated.

Functions block

- Define a repeatedly used function.
- A function computing the skewness of a vector.

```
functions {
    real skewness(vector x){

        int N = num_elements(x);
        vector[N] m3;
        real skewness;

        for (i in 1 : N){
            m3[i] = ((x[i]-mean(x))/sd(x))^3;
        }
        skewness = sum(m3)/N;
        return skewness;
    }
}
```

Syntax overview

- Data types and variable declarations
- Operations
- Indexing
- Statements

Syntax overview

- Declaration before the first use.
- Declarations are placed at the beginning of the corresponding blocks.
- Every quantity has its type.
- Every assignment statement must be followed by a semicolon.
- The order is crucial.
- Be precise about the global and local scope.
  - The variables declared in each block have scope over all subsequent statements but not the other way.
  - Local variables have effect within its container.
- Exceptionally, variables declared in the model block are always local.
- Variables declared as function parameters have scope only within that function definition's body.

Data Types and Variable Declarations

- Every variable must have a declared data type.
- Only values of that type will be assignable to the variable.

Data Types

- Two primitive types:
  - `int` for integer values ($[-2^{31}, 2^{31} - 1]$).
  - `real` for continuous values ($[-2^{1022}, 2^{1022}]$),
- Three vector and matrix types:
  - `vector` for column vectors,
  - `row_vector` for row vectors,
  - `matrix` for matrices.
- Array types, e.g.
  - `real x[10]` declares `x` to be a one-dimensional array of size 10 containing real values.
  - `matrix[3, 3] m[6, 7]` declares `m` to be a two-dimensional array of size 6×7 containing values that are 3×3 matrices.

Data Types

- Arrays are the only way to store sequences of integers.
- Vectors and matrices versus arrays are not assignable to one another, even if their dimensions are identical.
- Vectors, matrices, and arrays are accessed by indexes.
- Their sizes are integer expressions.

Constrained Data Types

- All of the basic data types may be given lower and upper bounds:

```
int<lower = 1> N;
```

```
real<upper = 0> log_p;
```

```
vector<lower = -1, upper = 1>[3] rho;
```

- Bounds for integer or real variables may be arbitrary expressions:

```
real<lower=min(y), upper=max(y)> phi;
```

Constrained Data Types

Special data types for structured vectors and matrices

- Four constrained vector data types:
  - `simplex` for unit simplexes,
  - `unit_vector` for unit-length vectors, i.e. norm is 1,
  - `ordered` for ordered vectors of scalars,
  - `positive_ordered` for vectors of positive ordered scalars.
- Specialized matrix data types:
  - `corr_matrix` for correlation matrices,
  - `cov_matrix` for covariance matrices (symmetric, positive definite),
  - `cholesky_factor_cov` for Cholesky factors of covariance matrices,
  - `cholesky_factor_corr` for Cholesky factors of correlation matrices.

Constrained Data Types

Roles of constrained data types:

- Data checking in the data, transformed data, transformed parameter, and generated quantities block.
- Transformation determination from constrained variables to unconstrained variables.
- Constraints.

Declaration

- Integer and real: type + (optional) <constraint> + name

```
int N;
```

```
int<lower=1> N;
```

```
int<lower=0,upper=1> bin;
```

```
real theta;
```

```
real<lower=0> sigma;
```

```
real<lower=-1,upper=1> rho;
```

- Vectors: type + (optional) <constraint> + length + name

```
vector[3] u;
```

```
vector<lower=0>[3] u;
```

```
ordered[5] c;
```

```
row_vector[2018] u;
```

Declaration

- Matrix: type + (optional) <constraint> + dimension + name

```
matrix[M, N] B;
matrix<upper=0>[3, 4] B;
corr_matrix[3] Sigma;
cov_matrix[K] Omega;
```

- Arrays: type of entries + (optional) <constraint> + name + dimension

```
int n[5];
real a[3, 4];
real<lower=0> z[5, 4, 2];
vector[7] mu[3];
matrix[7, 2] mu[15, 12];
```

Exercise 2: Data types and constraint

- Which data types are used in Ex1?
- Which constraints are used and why?
- What happens if `real sigma;` or `real<upper=0> sigma;`?
- Can we specify `real<lower=1.0> sigma;`?

Exercise 3: Declaration

The folder Ex3 contains a data having two variables, x and y. We fit a simple linear regression model with y the response and x the covariate. This exercise is different from the Ex1 as we put the intercept covariate $(\mathbf{1})$ and x into a matrix of two columns. The Stan program has one missing declaration.

- Run R and adapt the program following the Stan's error message.
- How do we change the program (R and Stan) if we have 2 covariates, $x1$ and $x2$?
- How if we have $p \geq 3$ covariates?

Naming

Rules for naming:

- Case sensitive.
- A declared name cannot be identical with a reserved name (See the manual).
- A name is an ASCII strings containing only the basic lower-case and upper-case letters, digits, and the underscore (_) character.
- Variables must start with a letter (a-z and A-Z) and may not end with two underscores (__).
- Examples of legal names:

```
a, a3, a_3, Sigma, my_cpp_style_variable.
```

- Unlike R and BUGS, full stop (.) is not allowed.

Some notes on vector, matrix, and array

- Intrinsically, vectors and matrices are one-dimensional and two-dimensional arrays respectively but they are not assignable to one another, even if their dimensions are identical.
- Situations where only vectors and matrices may be used:
  - matrix arithmetic operations (e.g., matrix multiplication),
  - linear algebra functions (e.g., eigenvalues and determinants), and
  - multivariate function parameters and outcomes (e.g., multivariate normal distribution arguments).
- Vectors and matrices cannot be typed to return integer values. They are restricted to real values.
- Note: real values are never demoted to integers.

Compound Variable Declaration and Definition

- Stan allows assignable variables to be declared and defined in a single statement.

```
int N = num_elements(x);
```

- Assignable variables are local variables and variables declared in the transformed data, transformed parameters, or generated quantities blocks.

```
int p = 5;
```

- The right-hand side may be any expression which has a type being assignable to the variable being declared.

```
matrix[3, 2] A = 0.5 * (B + C);
```

Vector, matrix, and array expressions

- Use [ ] for vectors and matrices, and { } for arrays.

```
row_vector[2] rv2 = [ 1, 2 ];
```

```
vector[3] v3 = [ a, b, c ]';
```

```
matrix[3,2] m1 = [ [ 1, 2 ], [ 3, 4 ], [5, 6 ] ];
```

```
int b[2, 3] = { { 1, 2, 3 }, { 4, 5, 6 } };
```

Arithmetic and matrix operations

- Integer and real: addition (+), subtraction (-), negation (-), multiplication (*) and division (/) as usual.
- Integer: modulus (%).
- Integer and real: Exponentiation `^`, returning a real value.
- Vector, row vector, matrix: addition, subtraction, negation, and multiplication as usual. Transpose operation using an apostrophe (').
- Vector, row vector, matrix: Element-wise multiplication and division operations, a .* b and a ./b, but not better than for loop.
- Return types for vector and matrix operations are the smallest types that can be statistically guaranteed to contain the result.
- See Chapter 43 for a complete list of operators.

Exercise 4: Arithmetic and matrix operations

Let `k` is of type `int` or `real`, `y` and `mu` are of type `vector` and `Sigma` is of type `matrix`. Which return type of the following expressions:

- `k * y`
- `k * y'`
- `k + Sigma`
- `k * Sigma`
- `(y - mu)'`
- `y'*y`
- `(y - mu)' * Sigma * (y - mu)`

Operator Precedence and Associativity

Op.	*Prec.*	*Assoc.*	*Placement*	*Description*
? :	10	right	ternary infix	conditional
||	9	left	binary infix	logical or
&&	8	left	binary infix	logical and
==	7	left	binary infix	equality
!=	7	left	binary infix	inequality
<	6	left	binary infix	less than
<=	6	left	binary infix	less than or equal
>	6	left	binary infix	greater than
>=	6	left	binary infix	greater than or equal
+	5	left	binary infix	addition
-	5	left	binary infix	subtraction
*	4	left	binary infix	multiplication
/	4	left	binary infix	(right) division
%	4	left	binary infix	modulus
\	3	left	binary infix	left division
.*	2	left	binary infix	elementwise multiplication
./	2	left	binary infix	elementwise division
!	1	n/a	unary prefix	logical negation
-	1	n/a	unary prefix	negation
+	1	n/a	unary prefix	promotion (no-op in Stan)
^	0.5	right	binary infix	exponentiation
'	0	n/a	unary postfix	transposition
()	0	n/a	prefix, wrap	function application
[]	0	left	prefix, wrap	array, matrix indexing

Operator precedence and associativity

- Precedence: smaller precedence has higher priority, e.g.

  `a+b*c` is interpreted as `a+(b*c)`, `u*v'` as `u*(v')`.

- Same precedence: follow associativity.

- Parentheses are recommended. Only () is allowed.

Indexing

- Vectors, matrices, and arrays are accessed by indexes using the same array-like notation.

  If `vector[3] v3 = [ a, b, c ]';` , v3[2] = b;

  If `matrix[3,2] m1 = [ [ 1, 2 ], [ 3, 4 ], [5, 6 ] ];` , m1[2, 1] = 3

  If `int b[2, 3] = { { 1, 2, 3 }, { 4, 5, 6 } };` , b[2, 3] = 6

- Subscripting has higher precedence than any of the arithmetic operations, e.g.

  `alpha*x[1]` is equivalent to `alpha*(x[1])` .

- Indexes can be multiple.

- Expressions are allowed for indexes.

Indexing

- For matrix, `col(matrix x, int n)` returns the n-th column of x and `row(matrix x, int m)` returns the m-th row of x, a row_vector. A shorthand for `row(matrix x, int m)` is `x[m]`.

example	*row index*	*column index*	*result type*
a[i]	single	n/a	row vector
a[is]	multiple	n/a	matrix
a[i, j]	single	single	real
a[i, js]	single	multiple	row vector
a[is, j]	multiple	single	vector
a[is, js]	multiple	multiple	matrix

Figure 4.3: *Special rules for reducing matrices based on whether the argument is a single or multiple index. Examples are for a matrix* a*, with integer single indexes* i *and* j *and integer array multiple indexes* is *and* js*. The same typing rules apply for all multiple indexes.*

- Multiple indexes can be integer arrays of indexes, lower bound 3:, upper bound :5, lower and upper bounds 2:7.

Exercise 5: Indexing

Given $A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$, which (indexing) operator is used to produce the following results?

$$3,\ \begin{bmatrix} 4 & 5 & 6 \end{bmatrix},\ \begin{bmatrix} 2 & 3 \\ 5 & 6 \\ 8 & 9 \end{bmatrix},\ \begin{bmatrix} 7 \\ 8 \\ 9 \end{bmatrix},\ \begin{bmatrix} 5 & 8 \\ 6 & 9 \end{bmatrix}.$$

We will use Stan to check your solution later.

Exercise 6: Ex3 with p covariates

The folder Ex6 contains the same data as in the Ex3. We re-fit the linear regression model in that exercise.

- Re-write `beta[1] * x[i, 1] + beta[2] * x[i, 2]` in the Stan program as a product of a row vector and a vector. Note that x[i, 1] and x[i, 2] are the elements of the i^{th} row of x.
- Run the program.
- Replace 2 by p and adapt the program.
- In R, add one more covariate in x, adapt R code, and re-run the program.

Exercise 7: Ex6 with an array for x

The folder Ex7 contains the same data and programs as in the previous exercise except that we declare x using an array.

- Declare x as an one-dimensional array of size N where its elements are vectors of length p.
- Adapt the Stan program and run it.

Statements

- Blocks = declarations + statements.
- Assignment statement:

```
Sigma = diag_matrix(sigma) * Omega * diag_matrix(sigma);
```

- Compound arithmetic and assignment statement:

Operation	*Compound*	*Long*
addition	`x += y`	`x = x + y`
subtraction	`x -= y`	`x = x - y`
multiplication	`x *= y`	`x = x * y`
division	`x /= y`	`x = x / y`
elementwise multiplication	`x .*= y`	`x = x .* y`
elementwise division	`x ./= y`	`x = x ./ y`

Figure 5.1: *Stan allows compound arithmetic and assignment statements of the forms listed in the table above. The compound form is legal whenever the corresponding long form would be legal and it has the same effect.*

"Sampling" statements

- The most important statement, defining the target.
- Just a notation, no sampling performance.

```
y[i] ~ normal(mu,sigma);
```

- Expressed as an increment on the total log probability

```
target += normal_lpdf(y | mu, sigma);
```

- In general `y ~ dist(theta);` is expressed as

`target += dist_lpdf(y | theta);` if y is `real` and
`target += dist_lpmf(y | theta);` if y is `integer`.

- Log probability increment vs. sampling statement: including vs excluding all constant terms.

Target calculation

- The model:

$$y_i \sim N(\beta_1 + \beta_2 \times x_i, \sigma^2) \quad \forall i = 1, ..., N$$

- Combining with the prior, the posterior:

$$\text{posterior} \propto f_{N(0,1000)}(\beta_1) \times f_{N(0,1000)}(\beta_2) \times f_{half-cauchy(0,5)}(\sigma) \times \prod_{i=1}^{N} f_{N(\beta_1+\beta_2\times x_i,\sigma^2)}(y_i)$$

- Stan works with log scale:

$$\text{log post.} = \log f_{N(0,1000)}(\beta_1) + \log f_{N(0,1000)}(\beta_2) + \log f_{half-cauchy(0,5)}(\sigma) + \sum_{i=1}^{N} \log f_{N(\beta_1+\beta_2\times x_i,\sigma^2)}(y_i) + \text{a constant}$$

Target calculation

- What we see:

```
model {
    beta ~ normal(0, 1000);     // prior
    sigma ~ cauchy(0, 5);       // half-cauchy implicitly
    for (i in 1:N) {
        y[i] ~ normal(beta[1] + beta[2] * x[i], sigma);
    }
}
```

- Behind the scene (C++):

Stan language	Translated
beta ~ normal(0, 1000)	target = 0 (initialization) target += normal_lpdf(beta[1] | 0, 1000) target += normal_lpdf(beta[2] | 0, 1000)
sigma ~ cauchy(0, 5)	target += cauchy_lpdf(sigma | 0, 5)
y[i] ~ normal(beta[1] + beta[2] * x[i], sigma)	target += normal_lpdf(y[i] | beta[1] + beta[2] * x[i], sigma)

- Finally, target = log post. (+ a constant)

Comments

- // This is a comment
- /* This is a comment,
  using two lines */

Other statements

- For loops

```
for (n in 1:N) { // {} is optional if only one line in for loop
    real theta; // theta is local here
    theta = inv_logit(alpha + x[n] * beta);
    y[n] ~ bernoulli(theta);
}
```

- Conditional

```
if (condition1)
    statement1          // group by {} if more than one line
else if (condition2) // (optional)
    statement2
    // ...
else if (conditionN-1)
    statementN-1
else
    statementN
```

- Print statements: Using for debugging.

```
for (n in 1:2)
    print("u[", n, "] = ", u[n]);
```

Other statements

See the Stan manual for other statements:

- While loop.
- Break and continue statements.
- Reject statements.

Exercise 8: Print

The folder Ex8 has a program using the print statement to check your answer in Exercise 5.

- Adapt the program model_Ex8.stan.

Functions

- All built-in mathematical and statistical functions are provided in part VII of the manual.
- Note that binary operators, e.g. real * real are written as operator*(real, real).
- Having higher precedence than any of the others.
- Type signature requirement: input and output type.
- Uniquely determined by its name and its sequence of argument types. Thus, `real mean(real[]);` and `real mean(vector);` are different.
- Constants have no arguments, e.g. `pi()` .

Important built-in functions of a distribution

If `dist()` is a distribution then

- `dist_lpdf` a function computing the log probability density if it is continuous.
- `dist_lpmf` a function computing the log probability mass if it is discrete.
- `dist_rng` a function generating a variate.
- See the manual for more detail.

User-defined functions

- Similar to R but explicitly requires input and output types, e.g.

```
real foo(real a, real b) {
    return a+b;
}
```

- Must have a return as the last statement.

- Last but not last! Be careful when use loop or condition. This is not qualified:

```
real foo(real x) {
    if (x > 2) return 1.0;
    else if (x <= 2) return -1.0;
}
```

since no default else statement.

- The functions block is placed before the data block.

User-defined functions

- Function names follow naming rules.
- Arguments are mandatory, no default values.
- Functions are only declared for base type (integer, real, vector, row_vector, and matrix) and dimensionality.
- No size and no constraints (lower-bound, upper-bound, or specialized types).

```
real skewness(vector x){
    int N = num_elements(x); // local variable,
    ... // others in the function body
}

matrix new_psi_fun(real psi, real t) {
    matrix[2, 2] output; // local variable
    output[1, 1] = psi^t;
    ...
    return output;
}
```

- More details, see Sections 7 and 24 in the manual.

Exercise 9: User-defined functions

The Ex9 contains a Stan program to define the following function: Input is a vector and output is a diagonal matrix where the vector lies along the diagonal.

- Definition: matrix my_func(vector x).
- Declare a local matrix to store the elements of the output.
- Use for loop and if-else condition to define the matrix.

$$A_{ij} = \begin{cases} 0 \text{ if } i \neq j \\ x_i \text{ otherwise} \end{cases}$$

- Define that function in the functions block of the Stan program.
- Use print statement to check your program.
- Check whether the manual having a built-in function doing the same.

Working with R

- Monitor parameters from parameters, transformed parameters, and generated quantities block.

```
stan(..., pars = c('beta', 'sigma'), ...)
```

- Convergence diagnosis using trace plots

```
stan_trace(model1a, pars = 'beta', inc_warmup = T)
stan_trace(model1a, pars = 'sigma', inc_warmup = T)
```

- Saving trace plots

```
windows()
stan_trace(model1a, pars=c('beta'), inc_warmup = F)
savePlot('Rplot_beta', type = 'pdf')
```

- Formal convergence diagnosis by Rhat and effective sample size.

```
summary(model1a)$summary
```

- Final summary.

```
summary(model1a)$summary
```

ShinyStan

A powerful package for visualization and more


```
library(shinystan)
```

```
launch_shinystan(model1a)
```

Exercise 10: Shinystan

The folder Ex10 contains a program to explore shinystan.

- Run the program in R.
- Run shinystan and check convergence using the DIAGNOSE tab.
- Explore more shinystan features.

rstanarm

Applied Regression Modeling via RStan

- Allow regression models to be specified using the customary R modeling syntax.
- Similar syntax as lm and glm in stats package.

```
stan_lm(formula, data, subset, weights, na.action, model = TRUE, x = FALSE,
y = FALSE, singular.ok = TRUE, contrasts = NULL, offset, ...,
prior = R2(stop("'location' must be specified")), prior_intercept = NULL,
prior_PD = FALSE, algorithm = c("sampling", "meanfield", "fullrank"),
adapt_delta = NULL)
```

```
lm(formula, data, subset, weights, na.action,
method = "qr", model = TRUE, x = FALSE, y = FALSE,
qr = TRUE, singular.ok = TRUE, contrasts = NULL, offset, ...)
```

Some important notes

- Be clear about Stan notation, definitions, parameterization, and so on in the manual that are relevant to your model.
- Many examples are given in the manual and the Stan website.
- Keep in mind that Stan also has some limitations:
  - No discrete parameters.
  - No implicit missing data (code as parameters).

Some differences from BUGS

	Stan	BUGS
Sampling	All at a time	One at a time
Statement order	A matter	Does not matter
Gradients	Required	Not required
Naming	Full stop not possible	Full stop not possible
Normal distribution	Standard deviation	Precision
Distribution notation	`normal, poisson`	`dnorm, dpois`
Reassignment of local variables	Allow	Not allow
Discrete parameters	Not allow yet	Allow
Implicit flat prior	Yes	No
Conjugacy	No computational advantage	Yes

More efficient coding

In some situations, the following coding might be more efficient

- Vectorization vs for loop.
  - For loop

```
for (i in 1:N) {
y[i] ~ normal(beta[1] + beta[2] * x[i], sigma);
}
```

  - Vectorization

```
y ~ normal(beta[1] + beta[2] * x, sigma);
```

- Time comparison between for loop and vectorization, measured in second.

Sample size	For loop	Vecterization	Ratio
5000	2.587	1.555	1.664
50000	43.987	24.180	1.819
200000	182.345	131.761	1.384
500000	494.569	372.416	1.328

More efficient coding

- Centered versus non-centered parameterization.
  - Center

```
parameters {
real y;
vector[9] x;
}
model {
y ~ normal(0, 3);
x ~ normal(0, exp(y/2));
}
```

  - Non-centered

```
parameters {
real y_raw;
vector[9] x_raw;
}
transformed parameters {
real y;
vector[9] x;
y = 3.0 * y_raw;
x = exp(y/2) * x_raw;
}
model {
y_raw ~ normal(0, 1); // implies y ~ normal(0, 3)
x_raw ~ normal(0, 1); // implies x ~ normal(0, exp(y/2))
}
```

Some suggestions

- Start writing from the model block.
- Build up with data and parameters blocks, then other blocks.
- Choose priors (BUGS prior might not the best since Stan does not require conjugacy).
- Number of iterations:
  - 10 or some for checking grammatical errors.
  - 100 or 500 or 1000 to see whether model can converge or to check logic error (e.g. beta[2] written as beta[1]).
  - 2000, 5000, or more for final estimates.

Ex11: Posterior predictive check (PPC) for the linear regression model

This exercise, in the folder Ex11, is for checking whether the linear regression model in Exercise 3 is appropriate for that data.

- Use the built-in function `normal_rng(mu, sigma)` to generate the replicate.
- Using Shinystan to visualize the distribution of the response and the replicate.

Ex12: A logistic regression model

In the folder Ex12, the file Ex12.txt contains the data with two variables, x and y where y is binary and x is continuous. The logistic regression model, y is regressing on x, is given in model_Ex12.stan.

- What is the type of y and why?
- Run the program.
- Compute $P(y = 1|x = 1)$ in the generated quantities block.
- In the program, the print statement prints the probability to the console. Adapt the Stan program so that you can monitor that probability in R. Adapt R code to obtain the output.

Ex13: A longitudinal analysis for balanced data. An AR structure

The folder Ex13 contains a balanced longitudinal data collected for $N = 30$ individuals, each has $T = 4$ repeated measurements. Let y_{ij} is the response for the individual i $(i = 1, ..., N)$ at time point j $(j = 1, ..., T)$. Denote $\boldsymbol{y}_i = (y_{i1}, ..., y_{iT})^T$. The covariates are x and $time$. Fit the following model:

$$\boldsymbol{y}_i \sim N(\boldsymbol{\mu}_i, \Sigma)$$

where $\mu_{ij} = \beta_1 + \beta_2 \times x_i + \beta_3 \times time_j + \beta_4 \times x_i \times time_j$ and Σ has an autoregressive structure, i.e. $\Sigma_{ij} = \sigma \times \rho^{|i-j|}$, where $\sigma \geq 0$ and $0 \leq \rho \leq 1$ are two parameters.

- The incomplete Stan program uses two for loops to define Σ in the transformed parameters block. Complete the program.
- Is(Are) there any block(s) that we can define Σ? If yes, adapt your program. Is there any difference between the adapted program(s) and the initial one?

Ex14: A linear mixed model

The folder Ex14 contains the data from the Ex13 one. A new ID is created in R, ranging from 1 to 30. Fit a random intercept model of the following form to the data in Ex13:

$$y_{ij} \sim N(\mu_{ij}, \sigma^2),$$

where $\mu_{ij} = \beta_1 + \beta_2 \times x_i + \beta_3 \times time_j + \beta_4 \times x_i \times time_j + b_i$ and $0 < \sigma$ is a parameter.

- Run the model.
- Extend the model to include a random slope.

Coding with missing values

- Suppose now that y in Ex3 has some missing values:

y	x
0.7008	0.6469
NA	-0.6993
0.3515	-1.3202
...	...

- Unfortunately, Stan does not support NA in the data block.
- Solutions:
  - (i) Separate the observed part and missing part of y into appropriate blocks.
  - Or, (ii) turn y into fully observed.

Coding with missing values

Solution (i): Missing data as parameters.

- Create an index and a missing indicator, then separate y:

Original				Observed part				Missing part			
No	y	x	mis_ind	No	y	x	mis_ind	No	y	x	mis_ind
1	0.7008	0.6469	0	1	0.7008	0.6469	0	2	NA	-0.6993	1
2	NA	-0.6993	1	3	0.3515	-1.3202	0	...	...	...	...
3	0.3515	-1.3202	0	...	...	...	...				
...	...	...	...								

- y is partitioned into y_{obs} and y_{mis}.
  - y_{obs} is declared in the data block.
  - y_{mis} is declared in the parameters block.
- y is merged in transformed parameters block.
- Several index arrays for partitioning and combining.

Coding with missing values

Preparing in R:

- Create an index if not available.
- Create a missing indicator: 0 if observed and 1 if missing.
- Partition the original data frame into two data frames, using the missing indicator.

Coding with missing values

Stan code: Partitioning

```
data {
    int<lower = 0> N_obs;
    int<lower = 0> N_mis;

    int<lower = 1, upper = N_obs + N_mis> id_obs[N_obs];
    int<lower = 1, upper = N_obs + N_mis> id_mis[N_mis];

    vector[N_obs] y_obs;
    vector[N_obs + N_mis] x; // covariate
}

transformed data {
    int<lower = 0> N = N_obs + N_mis; // sample size
}
```

Coding with missing values

Stan code: Combining

```
parameters {
    vector[N_mis] y_mis;
    vector[2] beta; // regression coefficients
    real<lower=0> sigma; // SD
}

transformed parameters {
    vector[N] y;
    y[id_obs] = y_obs;
    y[id_mis] = y_mis;
}
```

Coding with missing values

Stan code: Modeling

```
model {
    beta ~ normal(0, 1000); // prior
    sigma ~ cauchy(0, 5); // half-cauchy implicitly

    for (i in 1:N) {
        y[i] ~ normal(beta[1] + beta[2] * x[i], sigma);
    }
}
```

Coding with missing values

Solution (ii): Using "artificially imputed" data

- Missing values are imputed by "implausible" values in R.

```
y_impute = NULL

for (i in 1 : N){
    if (data$mis_ind[i]==0){y_impute[i] = data$y[i]}
        else y_impute[i] = 999
}
```

Coding with missing values

- Replace missing values by imputed values.

```
data2b = list(N=N, mis_ind = mis_ind,
                   x = x, y=y_impute)
```

- Use if statement in model block

```
for (i in 1:N) {
    if (mis_ind[i] == 0){
        y[i] ~ normal(beta[1] + beta[2] * x[i], sigma);
    }
}
```

Ex15: Dealing with missing values

The folder Ex15 contains the data taken from Ex3 with some missing values on y. Two programs for the two solutions are provided.

- Understand those programs.

Some notes on missing values

- Solution (i) only works with continuous response. Solution (ii) can be used for both continuous and discrete response.
- For missing covariates, the first can be extended and a sequence of imputation models are implemented as long as they are continuous.
- Both solutions may not be used for missing discrete covariates.

Summary

At the end of this presentation:

- You know how to translate a statistical model to a Stan program.
- You are able to run in R.
- You know where to look for help.

References

Hoffman, M. and Gelman, A. (2014). The No-U-Turn sampler: Adaptively setting path lengths in Hamiltonian Monte Carlo, *Journal Of Machine Learning Research* **15**: 1593–1623.

Neal, R. M. (2012). MCMC using Hamiltonian dynamics, *arXiv* .

Nesterov, Y. (2009). Primal-dual subgradient methods for convex problems, *Mathematical Programming* **120**(1): 221–259.

THANK YOU
FOR
YOUR ATTENTION