CmdStan Interface

User's Guide

Stan Development Team

CmdStan Version 2.3.0

Sunday 20th July, 2014

http://mc-stan.org/

Stan Development Team. 2014. *CmdStan: User's Guide*. Version 2.3.0

Contents

Part I

Introduction

1. Overview

This document is a user's guide for the CmdStan interface to the Stan probabilistic modeling language. CmdStan is one of several interfaces to Stan; there are also R and Python interfaces.

1.1. Stan Home Page

For links to up-to-date code, examples, manuals, bug reports, feature requests, and everything else Stan related, see the Stan home page:

http://mc-stan.org/

1.2. Licensing

CmdStan and Stan are both licensed under the new BSD license (3-clause). See the appendix for details, including licensing terms for the dependent packages Boost and Eigen.

1.3. Modeling Language User's Guide and Reference

Stan's modeling language is shared across all of its interfaces. Stan's language, along with a programming guide and many example models, is detailed in the *Stan Modeling Language User's Guide and Reference Manual*, which is available from the Stan home page (see Section 1.1).

1.4. Example Models

There are many example models for Stan, in addition to those in the user's guide and reference. These are all linke dform the Stan home page (see Section 1.1).

2. Getting Started

This chapter is designed to help users get acquainted with the overall design of the Stan language and calling Stan from the command line. Later chapters are devoted to expanding on the material in this chapter with full reference documentation. The content is identical to that found on the getting-started with the command-line documentation on the Stan home page, http://mc-stan.org/.

2.1. For BUGS Users

An appendix in the language user's guide and reference manual describes some similarities and important differences between Stan and BUGS (including WinBUGS, OpenBUGs, and JAGS).

2.2. Installation

For information about supported versions of Windows, Mac and Linux platforms along with step-by-step installation instructions, see Appendix B.

2.3. Building Stan

Building Stan itself works the same way across platforms. To build Stan, first open a command-line terminal application. Then change directories to the directory in which Stan is installed (i.e., the directory containing the file named `makefile`).

```
> cd <stan-home>
```

Then make the library with the following make command

```
> make bin/libstan.a
```

then make the model parser and code generator with the following call, adjusting the `2` in `-j2` to the number of CPU cores available

```
> make -j2 bin/stanc
```

On Windows, that'll be `bin/stanc.exe`.

Warning: The `make` program may take 10+ minutes and consume 2+ GB of memory to build `stanc`. Compiler warnings, such as `uname: not found`, may be safely ignored.

Finally, make the Stan output summary program with the following make command.

```
> make bin/print
```

Building `libstan.a`, `bin/stanc`, and `bin/print` needs to be done only once.

2.4. Compiling and Executing a Model

The rest of this quick-start guide explains how to code and run a very simple Bayesian model.

A Simple Bernoulli Model

The following simple model is available in the source distribution located at `<stan-home>` as

```
src/models/basic_estimators/bernoulli.stan
```

The file contains the following model.

```
data {
  int<lower=0> N;
  int<lower=0,upper=1> y[N];
}
parameters {
  real<lower=0,upper=1> theta;
}
model {
  theta ~ beta(1,1);
  for (n in 1:N)
    y[n] ~ bernoulli(theta);
}
```

The model assumes the binary observed data `y[1],...,y[N]` are i.i.d. with Bernoulli chance-of-success `theta`. The prior on `theta` is `beta(1,1)` (i.e., uniform).

Implicit Uniform Priors

If no prior is specified for a parameter, it is implicitly given a uniform prior on its support. For parameters such as `theta` in the example, which are constrained to fall between 0 and 1, this produces a proper uniform distribution on the support of `theta`. Because `Beta`$(1,1)$ is the uniform distribution, the following sampling statement can be eliminated from the model without changing the log probability calculation.

```
  theta ~ beta(1,1);
```

For parameters with unbounded support, the implicit uniform prior is improper. Stan allows improper priors to be specified in models, but posteriors must be proper in order for sampling to succeed.

Constraints on Parameters

The variable `theta` is defined with lower and upper bounds, which constrain its value. Parameters with constrained support should always specify appropriate constraints in the parameter declaration; if the constraints are absent, sampling will either slow down or stop altogether based on whether the initial values satisfy the constraints.

Vectorizing Sampling Statements

Iterations of the model will be faster if the loop over sampling statements is *vectorized* by replacing

```
for (n in 1:N)
  y[n] ~ bernoulli(theta);
```

with the equivalent vectorized form,

```
y ~ bernoulli(theta);
```

Performance gains from vectorization are not because loops are slow in Stan, but because calls to sampling statements are slow. Vectorization allows multiple calls to a sampling statement to be replaced with a single call that can share common calculations for the log probability function, its gradients, and error checking. For more tips on optimizing the performance of Stan models, see the chapter in the language user's guide and reference manual on optimization.

Data Set

A data set of `N` = 10 observations is available in the file

```
src/models/basic_estimators/bernoulli.data.R
```

The content of the file is as follows.

```
N <- 10
y <- c(0,1,0,0,0,0,0,0,0,1)
```

This defines the contents of two variables, `N` and `y`, using an R-like syntax (see Chapter 6 for more information).

Generating and Compiling the Model

A single call to `make` will generate the C++ code for a model with a name ending in `.stan` and compile it for execution. This call will also compile the library `libstan.a` and the parser/code generator `stanc` if they have not already been compiled.

First, change directories to `<stan-home>`, the directory where Stan was unpacked that contains the file named `makefile` and a subdirectory called `src/`.

```
> cd <stan-home>
```

Then issue the following command:

```
> make src/models/basic_estimators/bernoulli
```

The command for Windows is the same, including the forward slashes.

The `make` command may be applied to files in locations that are not subdirectories issued from another directory as follows. Just replace the relative path `src/models/...` with the actual path.

The C++ generated for the model and its compiled executable form will be placed in the same directory as the model.

Sampling from the Model

The model can be executed from the directory in which it resides.

```
> cd src/models/basic_estimators
```

To execute sampling of the model under Linux or Mac, use

```
> ./bernoulli sample data file=bernoulli.data.R
```

The `./` prefix before the executable is only required under Linux and the Mac when executing a model from the directory in which it resides.

For the Windows DOS terminal, the `./` prefix is not needed, resulting in the following command.

```
> bernoulli sample data file=bernoulli.data.R
```

Whether the command is run in Windows, Linux, or on the Mac, the output is the same. First, the parameters are echoed to the standard output, which shows up on the terminal as follows.

```
method = sample (Default)
  sample
    num_samples = 1000 (Default)
    num_warmup = 1000 (Default)
    save_warmup = 0 (Default)
    thin = 1 (Default)
    adapt
      engaged = 1 (Default)
      gamma = 0.050000000000000003 (Default)
      delta = 0.80000000000000004 (Default)
      kappa = 0.75 (Default)
      t0 = 10 (Default)
      init_buffer = 75 (Default)
      term_buffer = 50 (Default)
      window = 25 (Default)
    algorithm = hmc (Default)
      hmc
        engine = nuts (Default)
          nuts
            max_depth = 10 (Default)
        metric = diag_e (Default)
        stepsize = 1 (Default)
        stepsize_jitter = 0 (Default)
id = 0 (Default)
data
  file = bernoulli.data.R
init = 2 (Default)
random
  seed = 4294967295 (Default)
output
  file = output.csv (Default)
  diagnostic_file =  (Default)
  refresh = 100 (Default)
...
```

The ellipses (`...`) indicate that the output continues (as described below).

After the configuration has been displayed a short timing warning is given.

```
...
Gradient evaluation took 4e-06 seconds
1000 transitions using 10 leapfrog steps per transition would take 0.04 seconds
Adjust your expectations accordingly!
...
```

Next, the sampler counts up the iterations in place, reporting percentage completed, ending as follows.

```
...
Iteration:    1 / 2000 [  0%]  (Warmup)
...
Iteration: 1000 / 2000 [ 50%]  (Warmup)
Iteration: 1001 / 2000 [ 50%]  (Sampling)
...
Iteration: 2000 / 2000 [100%]  (Sampling)
...
```

Sampler Output

Each execution of the model results in the samples from a single Markov chain being written to a file in comma-separated value (CSV) format. The default name of the output file is `output.csv`.

The first part of the output file just repeats the parameters as comments (i.e., lines beginning with the pound sign (#)).

```
# stan_version_major = 2
# stan_version_minor = 1
# stan_version_patch = 0
# model = bernoulli_model
# method = sample (Default)
#   sample
#     num_samples = 1000 (Default)
#     num_warmup = 1000 (Default)
#     save_warmup = 0 (Default)
#     thin = 1 (Default)
#     adapt
#       engaged = 1 (Default)
#       gamma = 0.050000000000000003 (Default)
#       delta = 0.80000000000000004 (Default)
#       kappa = 0.75 (Default)
#       t0 = 10 (Default)
#       init_buffer = 75 (Default)
#       term_buffer = 50 (Default)
#       window = 25 (Default)
#     algorithm = hmc (Default)
#       hmc
#         engine = nuts (Default)
#           nuts
#             max_depth = 10 (Default)
```

```
#          metric = diag_e (Default)
#          stepsize = 1 (Default)
#          stepsize_jitter = 0 (Default)
# id = 0 (Default)
# data
#   file = bernoulli.data.R
# init = 2 (Default)
# random
#   seed = 355899897
# output
#   file = output.csv (Default)
#   diagnostic_file =  (Default)
#   refresh = 100 (Default)
...
```

This is then followed by a header indicating the names of the values sampled.

```
...
lp__,accept_stat__,stepsize__,treedepth__,n_leapfrog__,n_divergent__,theta
...
```

The first column gives the log probability. The next columns, here columns two through five, provide sampler-dependent information. For basic Hamiltonian Monte Carlo (HMC) and its adaptive variant No-U-Turn sampler (NUTS), the sampler-depedent parameters are described in the following table.

Sampler	*Parameter*	*Description*
HMC	`accept_stat__`	Metropolis acceptance probability
HMC	`stepsize__`	Integrator step size
HMC	`int_time__`	Total integration time
NUTS	`accept_stat__`	Metropolis acceptance probability averaged over samples in the slice
NUTS	`stepsize__`	Integrator step size
NUTS	`treedepth__`	Tree depth
NUTS	`n_leapfrog__`	Number of leapfrog calculations
NUTS	`n_divergent__`	Number of divergent iterations

The rest of the columns in the header correspond to model parameters, here just `theta` in the sixth column. The parameter name header is output before warmup begins.

The result of any adaptation taking place during warmup is output next after the parameter names.

```
...
# Adaptation terminated
# Step size = 1.81311
# Diagonal elements of inverse mass matrix:
# 0.415719
...
```

The default sampler is NUTS with an adapted step size and a diagonal inverse mass matrix. For the running example, the step size is 1.81311, and the inverse mass contains the single entry 0.415719 corresponding to the parameter `theta`.

Samples from each iteration are printed out next, one per line in columns corresponding to the headers. [1]

```
...
-6.95293,0.945991,1.09068,2,3,0.335074
-6.92373,0.938744,1.09068,1,1,0.181194
-6.83655,0.934833,1.09068,2,3,0.304882
...
-7.01732,1,1.09068,1,1,0.348244
-8.96652,0.48441,1.09068,1,1,0.549066
-7.22574,1,1.09068,1,1,0.383089
```

The output ends with timing details,

```
...
#  Elapsed Time: 0.006811 seconds (Warm-up)
#                0.011645 seconds (Sampling)
#                0.018456 seconds (Total)
```

Summarizing Sampler Output

The command-line program `bin/print` will display summary information about the run (for more information, see Chapter 5). To run `print` on the output file generated for `bernoulli` on Linux or Mac, use

```
> <stan-home>/bin/print output.csv
```

where `<stan-home>` is the path to where Stan was unpacked. For Windows use backslashes for the executable,

```
> <stan-home>\bin\print output.csv
```

[1] There are repeated entries due to the Metropolis accept step in the No-U-Turn sampling algorithm.

The output of the command will display information about the run followed by information for each parameter and generated quantity. For `bernoulli`, we ran 1 chain and saved 1000 iterations. The information is echoed to the standard output stream. For the running example, the path to `<stan-home>` can be specified from the directory in which the Bernoulli model resides using `../` (with backslashes on Windows) as

```
> ../../../bin/print output.csv
```

For Windows, reverse the slashes. The output is

```
Inference for Stan model: bernoulli_model
1 chains: each with iter=(1000); warmup=(0); thin=(1); 1000 iterations saved.

Warmup took (0.0066) seconds, 0.0066 seconds total
Sampling took (0.011) seconds, 0.011 seconds total

                Mean     MCSE   StdDev        5%   50%   95%  N_Eff  N_Eff/s  R_hat
lp__           -7.3  3.5e-02  6.9e-01  -8.7e+00  -7.0  -6.7    390    34020   1.00
accept_stat__  0.64  1.2e-02  3.6e-01   5.1e-03  0.74   1.0    882    76898   1.00
stepsize__      1.8  7.8e-15  5.6e-15   1.8e+00   1.8   1.8   0.50       44   1.00
treedepth__   0.076  8.6e-03  2.7e-01   0.0e+00  0.00   1.0    942    82167   1.00
n_leapfrog__    2.7  4.9e-02  1.3e+00    1.0    3.0    3.0    716    65090  1.0e+00
n_divergent__  0.00  0.0e+00  0.0e+00   0.0e+00  0.00  0.00   1000    90909   1.00
theta          0.25  4.2e-03  1.2e-01   9.0e-02  0.23  0.47    827    72146   1.00

Samples were drawn using hmc with nuts.
For each parameter, N_Eff is a crude measure of effective sample size,
and R_hat is the potential scale reduction factor on split chains (at
convergence, R_hat=1).
```

In addition to the general information about the runs, `print` displays summary statistics for each parameter and generated quantity.

In the `bernoulli` model, there is a single parameter, `theta`. The mean, standard error of the mean, standard deviation, the 5%, 50%, and 95% quantiles, number of effective samples (total and per second), and $\hat{R}$ value are displayed. These quantities and their uses are described in detail in the introductory Markov chain Monte Carlo (MCMC) chapter of the language user's guide and reference manual.

The command `bin/print` can be called with more than one csv file by separating filenames with spaces. It will also take wildcards in specifying filenames. A typical usage of Stan from the command line would first create one or more Markov chains by calling the model executable, typically in parallel, writing the output CSV file for each into its own directory. Next, after all of the processes are finished, the results would be analyzed using `print` to assess convergence and inspect the means and quantiles of the fitted variables. Additionally, downstream inferences may be performed using the samples (e.g., to make decisions or predictions for unseen data).

Compile-time and Run-time Warnings

Stan tries to report warnings in order to help users correctly formulate and ebug models. There are two warnings in particular that deserve further explanation up front because we have not been able to formulate wording clearly enough to prevent confusion.

Metropolis Rejection Warning

The first problematic warning message involves the Metropolis sampler rejecting a proposal due to an error arising in the evaluation of the log probability function. Such errors are typically due to underflow or overflow in numerical computations that are the unavoidable consequence of floating-point approximations to continuous real values. The following is an example of such a message.

```
Informational Message: The current Metropolis proposal is about
to be rejected because of the following issue:
  Error in function stan::prob::normal_log(N4stan5agrad3varE):
  Scale parameter is 0:0, but must be > 0!
If this warning occurs sporadically, such as for highly constrained
variable types like covariance matrices, then the sampler is fine,
but if this warning occurs often then the model may be either severely
ill-conditioned or misspecified.
```

Despite using the word "Error" in the embedded report of the numerical issue, this is just an *informational message* (i.e., a warning); it is *not* an error. Particularly in early stages of sampler adaptation before adaptation has converged on the high-mass volume of the posterior, the numerical approximation to functions and to the Hamiltonian dynamics followed by the sampler can lead to numerical issues. As the message tries to indicate, if the message only occurs sporadically, then *the sampler is fine* and the user need not worry. In particular, the post-adaptation draws are still valid.

The reason this message is presented at all is that if it occurs repeatedly, the model may have a poorly conditioned constraint (typically with covariance or correlation matrices) or may be misformulated. In a future version of Stan, we plan to rank such messages by severity and turn this one off by default so as not to needlessly worry users.

Jacobian Required Warning

The second problematic warning message appears when a model is compiled and involves the requirement of a Jacobian adjustment to the log probability. If the left-hand side of a sampling statement involves a *non-linear* transform, then a Jacobian

adjustment must be made. A chapter of the language user's guide and reference manual include a full description of the Jacobians that are automatically applied and how to define Jacobians for transforms; in the user-defined distribution chapter there are examples.

In linear transforms or matrix/array slicing, spurious warnings arise. An example is the following model, which involves extraction of a row from a matrix for vectorized sampling.

```
parameters {
  matrix[20,10] y;
}
model {
  for (m in 1:20)
    row(y,m) ~ normal(0,1);
}
```

Compiling the above model leads to the following spurious warning.

```
Warning (non-fatal):  sampling statement (~) contains a transformed
  parameter or local variable. You must increment lp__ with the log
  absolute determinant of the Jacobian of the transform.
     Sampling Statement left-hand-side expression:
          row(y,m) ~ normal_log(...)
```

Such messages may be ignored if the transform involves is linear or only involves pulling out slices or blocks of larger structures.

Optimization

Stan can be used for finding posterior modes as well as sampling from the posterior. The model does not need to be recompiled in order to switch from optimization to sampling, and the data input format is the same. Although many command-line arguments may be provided to configure the optimizer, the following minimal command suffices, using defaults for everything but where to find the data file.

```
./bernoulli optimize data file=bernoulli.data.R
```

which prints out

```
method = optimize
  optimize
    algorithm = bfgs (Default)
      bfgs
        init_alpha = 0.001 (Default)
```

```
        tol_obj = 1e-08 (Default)
        tol_grad = 1e-08 (Default)
        tol_param = 1e-08 (Default)
    iter = 2000 (Default)
    save_iterations = 0 (Default)
id = 0 (Default)
data
  file = bernoulli.data.R
init = 2 (Default)
random
  seed = 2907588507
output
  file = output.csv (Default)
  append_sample = 0 (Default)
  diagnostic_file =  (Default)
  append_diagnostic = 0 (Default)
  refresh = 100 (Default)

initial log joint probability = -10.9308
    Iter      log prob        ||dx||      ||grad||    alpha  # evals  Notes
       7      -5.00402   3.67055e-07   3.06339e-11        1       10
Optimization terminated normally:
  Convergence detected: change in objective function was below
  tolerance
```

The first part of the output reports on the configuration used, here indicating the default BFGS optimizer, with default initial stepsize and tolerances for monitoring convergence. The second part of the output indicates how well the algorithm fared, here converging and terminating normally. The numbers reported indicate that it took 7 iterations and 10 gradient evaluations, resulting in a final state state where the change in parameters was roughly 3.7e-7 and the length of the gradient roughly 3e-11. The `alpha` value is for step size used. This is, not surprisingly, far fewer iterations than required for sampling; even fewer iterations would be used with less stringent user-specified convergence tolerances.

Output from Optimization

The output from optimization is written into the file `output.csv` by default. The output follows the same pattern as the output for sampling, first dumping the entire set of parameters used.

```
# stan_version_major = 2
# stan_version_minor = 1
# stan_version_patch = 0
# model = bernoulli_model
# method = optimize
#   optimize
```

```
#     algorithm = bfgs (Default)
#       bfgs
#         init_alpha = 0.001 (Default)
#         tol_obj = 1e-08 (Default)
#         tol_grad = 1e-08 (Default)
#         tol_param = 1e-08 (Default)
#     iter = 2000 (Default)
#     save_iterations = 0 (Default)
# id = 0 (Default)
# data
#   file = bernoulli.data.R
# init = 2 (Default)
# random
#   seed = 2907588507
# output
#   file = output.csv (Default)
#   append_sample = 0 (Default)
#   diagnostic_file =  (Default)
#   append_diagnostic = 0 (Default)
#   refresh = 100 (Default)
lp__,theta
-5.00402,0.2000000000030634
```

Note that everything is a comment other than a line for the header, and a line for the values. Here, the header indicates the unnormalized log probability with `lp__` and the model parameter `theta`. The maximum log probability is -5.0 and the posterior mode for `theta` is 0.20. The mode exactly matches what we would expect from the data. [2] Because the prior was uniform, the result 0.20 represents the maximum likelihood estimate (MLE) for the very simple Bernoulli model. Note that no uncertainty is reported.

Configuring Command-Line Options

The command-line options for running a model are detailed in Chapter 4. They can also be printed on the command line using Linux or Mac OS with

```
> ./bernoulli help-all
```

and on Windows with

```
> bernoulli help-all
```

[2] The Jacobian adjustment included for the sampler's log probability function is not applied during optimization, because it can change the shape of the posterior and hence the solution.

It may help to glance at the command-line skeletons in Figure 4.4 through Figure 4.8 to get a handle on the options then read the detailed descriptions earlier in Chapter 4.

Testing Stan

To run the Stan unit tests of basic functionality, run the following commands from a shell (where `<stan-home>` is replaced top-level directory into which Stan was unpacked; it should contain a file named `makefile`).[3]

```
> cd <stan-home>
> make -j4 O=0 test-headers
> make -j4 O=0 src/test/unit
> make -j4 O=0 src/test/unit-agrad-rev
> make -j4 O=0 src/test/unit-agrad-fwd
> make -j4 O=0 src/test/unit-distribution
> make -j4 O=3 src/test/CmdStan/models
```

As before, `-j4` indicates that four processes should be run in parallel; adjust the value `4` to correspond to the number of CPU cores available. Code optimization is specified by the letter 'O' followed by an equal sign followed by the digit '0' for no optimization and '3' for more optimization; optimization slows down compilation of the executable but reduces its execution time. Warnings can be safely ignored if the tests complete without a `FAIL` error.

Warning: The unit tests can take 30+ minutes and consume 3+ GB of memory with the default compiler, `g++`. The distribution test and model tests can take even longer. It is faster to run the Clang compiler (option `CC=clang++`), and to run in multiple processes in parallel (e.g., option `-j4` for four threads).

[3] Although command-line Stan runs with earlier versions of `make`, the unit tests require version 3.81 or higher; see Section B.8.2 for installation instructions.

Part II

Commands and Data Formats

3. Compiling Stan Programs

Preparing a Stan program to be run involves two steps,

1. translating the Stan program to C++, and
2. compiling the resulting C++ to an executable.

This chapter discusses both steps, as well as their encapsulation into a single make target.

3.1. Installing Stan

Before Stan can be run, it must be installed; see Appendix B for complete platform-specific installation details.

3.2. Translating and Compiling through `make`

The simplest way to compile a Stan program is through the `make` build tool, which encapsulates the translation and compilation step into a single command. The commands making up the `make` target for compiling a model are described in the following sections, and the following chapter describes how to run a compiled model.

Translating and Compiling Test Models

There are a number of test models distributed with Stan which unpack into the path `src/models`. To build the simple example `src/models/basic_estimators/bernoulli.stan`, the following call to `make` suffices. First the directory is changed to Stan's home directory by replacing `<stan-home>` with the appropriate path.

```
> cd <stan-home>
```

The current directory should now contain the file named `makefile`, which is the default instructions used by `make`. From within the top-level Stan directory, the following call will build an executable form of the Bernoulli estimator.

```
> make src/models/basic_estimators/bernoulli
```

This will translate the model `bernoulli.stan` to a C++ file and compile that C++ file, putting the executable in `src/models/basic_distributions/bernoulli(.exe)`. Although the make command including arguments is itself portable, the target it creates is different under Windows than in Unix-like platforms. Under Linux and the Mac, the executable will be called `bernoulli`, whereas under Windows it will be called `bernoulli.exe`.

Dependencies in `make`

A `make` target can depend on other `make` targets. When executing a `make` target, first all of the targets on which it depends are checked to see if they are up to date, and if they are not, they are rebuilt. This includes the top-level target itself. If the `make` target to build the Bernoulli estimator is invoked a second time, it will see that it is up to date, and not compile anything. But if one of the underlying files has changes since the last invocation `make`, such as the model specification file, it will be retranslated to C++ and recompiled to an executable.

There is a dependency included in the make target that will automatically build the `bin/stanc` compiler and the `bin/libstan.a` library whenever building a model.

Getting Help from the `makefile`

Stan's `makefile`, which contains the top-level instructions to `make`, provides extensive help in terms of targets and options. It is located at the top-level of the distribution, so first change directories to that location.

```
> cd <stan-home>
```

and then invoke make with the target `help`,

```
> make help
```

Options to `make`

Stan's `make` targets allow the user to change compilers, library versions for Eigen and Boost, as well as compilation options such as optimization.

These options should be placed right after the call to `make` itself. For instance, to specify the `clang++` compiler at optimization level 0, use

```
> make CC=clang++ O=0 ...
```

Compiler Option

The option `CC=g++` specifies the `g++` compiler and `CC=clang++` specifies the `clang++` compiler. Other compilers with other names may be specified the same way. A full path may be used, or just the name of the program if it can be found on the system execution path.

Optimization Option

The option `O=0` (that's letter 'O', equal sign, digit '0'), specifies optimization level 0 (no optimization), whereas `O=3` specifies optimization level 3 (effectively full optimization), with levels 1 and 2 in between.

With higher optimization levels, generated executable tends to be bigger (in terms of bytes in memory) and faster. For best results on computationally-intensive models, use optimization level 3 for the Stan library and for compiling models.

Library Options

Alternative versions of Eigen, Boost, and Google Test may be specified using the properties `EIGEN`, `BOOST`, and `GTEST`. Just set them equal to a path that resolves to an appropriate library. See the libraries distributed under `lib` to see which subdirectory of the library distribution should be specified in order for the include paths in the C++ code to resolve properly.

Additional `make` Targets

All of these targets are intended to be invoked from the top-level directory in which Stan was unpacked (i.e., the directory that contains the file named `makefile`).

Clean Targets

A very useful target is `clean-all`, invoked as

```
> make clean-all
```

This removes everything that's created automatically by `make`, including the `stanc` translator, the Stan libraries, and all the automatically generated documentation.

Make Target for `stanc`

To make the `stanc` compiler, use

```
> make bin/stanc
```

As with other executables, the executable `bin/stanc` will be created under Linux and Mac, whereas `bin/stanc.exe` will be created under Windows.

Make Target for Stan Library

To build the Stan library, use the following target,

```
> make bin/libstan.a
```

3.3. Translating Stan to C++ with `stanc`

Building the `stanc` Compiler and the Stan Library

Before the `stanc` compiler can be used, it must be built. Use the following command from the top-level distribution directory containing the file named `makefile`.

```
> make bin/stanc
```

This invocation produces the executable `bin/stanc` under Linux and Mac, and `bin/stanc.exe` under Windows. The invocation of `make`, including the forward slash, is the same on both platforms.

The default compiler option is `CC=g++` and the default optimization level is `O=3` (the letter 'O'); to see how to change these, see the previous section in this chapter on `make`.

The `stanc` Compiler

The `stanc` compiler converts Stan programs to C++ programs. The first stage of compilation involves parsing the text of the Stan program. If the parser is successful, the second stage of compilation generates C++ code. If the parser fails, it will provide a diagnostic error message indicating the location in the input where the failure occurred and reason for the failure.

The following example illustrates a fully qualified call to `stanc` to build the simple Bernoulli model; just replace `<stan-home>` with the top-level directory containing Stan (i.e., the directory containing the file named `makefile`).

For Linux and Mac:

```
> cd <stan-home>
> bin/stanc --name=bernoulli --o=bernoulli.cpp \
  src/models/basic_estimators/bernoulli.stan
```

The backslash (\) indicates a continuation of the same line.

For Windows:

```
> cd <stan-home>
> bin\stanc --name=bernoulli --o=bernoulli.cpp ^
    src/models/basic_estimators/bernoulli.stan
```

The caret (^) indicates continuation on Windows.

This call specifies the name of the model, here `bernoulli`. This will determine the name of the class implementing the model in the C++ code. Because this name is the name of a C++ class, it must start with an alphabetic character (`a-z` or `A-Z`) and

contain only alphanumeric characters (a–z, A–Z, and 0–9) and underscores (_) and should not conflict with any C++ reserved keyword.

The C++ code implementing the class is written to the file bernoulli.cpp in the current directory. The final argument, bernoulli.stan, is the file from which to read the Stan program.

Command-Line Options for stanc

The model translation program stanc is called as follows.

```
> stanc [options] model_file
```

The argument *model_file* is a path to a Stan model file ending in suffix .stan. The options are as follows.

--help
: Displays the manual page for stanc. If this option is selected, nothing else is done.

--version
: Prints the version of stanc. This is useful for bug reporting and asking for help on the mailing lists.

--name=*class_name*
: Specify the name of the class used for the implementation of the Stan model in the generated C++ code.

  Default: *class_name* = *model_file*_model

--o=*cpp_file_name*
: Specify the name of the file into which the generated C++ is written.

  Default: *cpp_file_name* = *class_name*.cpp

--no_main
: Include this flag to prevent the generation of a main function in the output.

  Default: generate a main function

3.4. Compiling C++ Programs

As shown in the previous section (Section 3.3), Stan converts a program in the Stan modeling language to a C++ program. This C++ program must then be compiled using a C++ compiler.

The C++ compilation step described in this chapter, the model translation step described in the last chapter, and the compilation of the dependent binaries bin/stanc and bin/libstan.a may be automated through make; see Section 3.2 for details.

Which Compiler?

Stan has been developed using two portable, open-source C++ compilers, `g++` and `clang++`, both of which run under and generate code for Windows, Macintosh, and Unix/Linux.[1]

The `clang++` compiler is almost twice as fast at low levels of optimization, but the machine code generated by `g++` at high optimization levels is faster.

What the Compiler Does

A C++ compiler like `g++` or `clang++` performs several lower-level operations in sequence,

1. parsing the input C++ source file(s),
2. generating (static or dynamically) relocatable object code, and
3. linking the relocatable object code into executable code.

These stages may be called separately, though the examples in this manual perform them in a single call. The compiler invokes the assembler to convert assembly language code to machine code, and the linker to resolve the location of references in the relocatable object files.

Compiler Optimization

Stan was written with an optimizing compiler in mind, which allows the code to be kept relatively clean and modular. As a result, Stan code runs as much as an order of magnitude or more faster with optimization turned on.

For development of C++ code for Stan, use optimization level 0; for sampling, use optimization level 3. These are controlled through Stan's makefile using `O=0` and directly through `clang++` or `g++` with `-O0`; in both cases, the first character is the letter 'O' and the second the digit '0'.

Building the Stan Library

Before compiling a Stan-generated C++ program, the Stan object library archive must be built using the makefile. This only needs to be done once and then the archive may be reused. The recommended build command for the Stan archive is as follows (replacing `<stan-home>` with the directory into which Stan was unpacked and which contains the file named `makefile`).

[1] As of the current version, Stan cannot be compiled using MSVC, the Windows-specific compiler from Microsoft. MSVC is able to compile the `stanc` compiler, but not the templates required for algorithmic differentiation and the Eigen matrix library.

```
> cd <stan-home>
> make CC=g++ O=3 bin/libstan.a
```

Please be patient and ignore the (unused function) warning messages. Compilation with high optimization on `g++` takes time (as much as 10 minutes or more) and memory (as much as 3GB).

This example uses the `g++` compiler for C++ (makefile option `CC=g++`). The `clang++` compiler may be used by specifying `CC=clang++`.

This example uses compiler optimization level 3 (makefile option `O=3`). Turning the optimization level down to 0 allows the code to built in under a minute in less than 1GB of memory. This will slow down sampling as much as an order of magnitude or more, so it is not recommended for running models. It can be useful for working on Stan's C++ code.

Compiling a Stan Model

Suppose following the instructions in the last chapter (Section 3.3) that a Stan program has been converted to a C++ program that resides in the source file `<stan-home>/my_model.cpp`.

The following commands will produce an executable in the file `my_model` in the current working directory (`<stan-home>`).

```
> cd <stan-home>
> g++ -O3 -Lbin -Isrc -isystem lib/boost_1.54.0                \
    -isystem lib/eigen_3.2.0 my_model.cpp -o my_model -lstan
```

The backslash (\) is used to indicate that the command is continued; it should be entered all one one line. The options used here are as follows.

`-O3` sets optimization level 3,

`-Lbin` specifies that the archive is in the `bin` directory,

`-Isrc` specifies that the directory `src` should be searched for code (it contains the top-level Stan headers),

`-isystem lib/boost_1.54.0` specifies the include directory for the Boost library,

`-isystem lib/eigen_3.2.0` specifies the include directory for the Eigen library,

`my_model.cpp` specifies the name of the source file to compile, and

`-o my_model` is the name of the resulting executable produced by the command (suffixed by `.exe` in Windows).

`-lstan` specifies the name of the archived library (not the name of the file in which it resides),

The library binary and source specifications are required, as is the name of the C++ file to compile. User-supplied directories may be included in header or archive form by specifying additional `-L`, `-l`, and `-I` options.

A lower optimization level may be specified. If there is no executable name specified using the `-o` option, then the model is written into a file named `a.out`.

Library Dependencies

Stan depends on two open-source libraries,

1. the Boost general purpose C++ libraries, and
2. the Eigen matrix and linear algebra C++ libraries.

These are both distributed along with Stan in the directory `<stan-home>/lib`.

The code for Stan itself is located in the directory `<stan-home>/src`. Because not all of Stan is included in the archive `bin/libstan.a`, the `src` directory must also be included for compilation.

4. Running a Stan Program

Once a Stan program is compiled, it can be run in many different ways. It can be used to sample or optimize parameters, or to diagnose a model. Before diving into the detailed configurations, the first section provides some simple examples.

4.1. Getting Started by Example

Once a Stan program defining a model has been converted to a C++ program for that model (see Section 3.3) and the resulting C++ program compiled to a platform-specific executable (see Section 3.4), the model is ready to be run.

All of the Stan functionality is highly configurable from the command line; the options are defined later in this chapter. Each command option also has defaults, which are used in this section.

Sampling

Suppose the executable is in file `my_model` and the data is in file `my_data`, both in the current working directory. To generate samples from a data set using the default settings, use one of the following, depending on platform.

Mac OS and Linux

```
> ./my_model sample data file=my_data
```

Windows

```
> my_model sample data file=my_data
```

On both platforms, this command reads the data from file `my_data`, runs warmup tuning for 1000 iterations (the values of which are discarded), and then runs the fully-adaptive NUTS sampler for 1000 iterations, writing the parameter (and other) values to the file `samples.csv` in the current working directory. When no random number seed is specified, a seed is generated from the system time.

Sampling in Parallel

The previous example executes one chain, which can be repeated to generate multiple chains. However, users may want to execute chains in parallel on a multicore machine.

Mac OS and Linux

To sample four chains using a Bash shell on Mac OS or Linux, execute[1]

```
> for i in {1..4}
  do
    ./my_model sample random seed=12345
       id=$i data file=my_data
       output file=samples$i.csv &
 done
```

The ampersand (&) at the end of the nested command pushes each process into the background, so that the loop can continue without waiting for the current chain to finish. The `id` value makes sure that a non-overlapping set of random numbers are used for each chain. Also note that the output file is explicitly specified, with the variable `$i` being used to ensure the output file name for each chain is unique.

The terminal standard output will be interleaved for all chains running concurrently. To suppress all terminal output, direct the standard output to the "null" device. This is achieved by postfixing `> /dev/null` to a command, which in the above case, means changing the second-to-last line to

```
       output file=samples$i.csv > /dev/null &  \
```

Windows

On Windows, the following is functionally equivalent to the Bash snippet above

```
> for /l %x in (1, 1, 4) do start /b model  sample   ^
    random seed=12345 id=%x data file=my_data         ^
    output file=samples%x.csv
```

The caret (^) indicates a line continuation in DOS.

Combining Parallel Chains

Stan has commands to analyze the output of multiple chains, each stored in their own file; see Chapter 5. RStan also has commands to read in multiple CSV files produced by Stan's command-line sampler.

To compute posterior quantities, it is sometimes easier to have the chains merged into a single CSV file. If the grep and sed programs are installed, then the following will combine the four comma-separated values files into a single comma-separated values file. The command is the same on Windows, Mac OS and Linux.

[1] Complicated multiline commands such as this one are prime candidates for putting into a script file.

```
> grep lp__ samples1.csv > combined.csv
> sed '/^[#l]/d' samples*.csv >> combined.csv
```

Scripting and Batching

The previous examples show how to sample in parallel from the command line. Operations like these can also be scripted, using shell scripts (`.sh`) on Mac OS and Linux and DOS batch (`.bat`) files on Windows. A sequence of several such commands can be executed from a single script file. Such scripts might contain `stanc` commands (see Section 3.3) and `bin/print` commands (see Chapter 5) can be executed from a single script file. At some point, it is worthwhile to move to something with stronger dependency control such as makefiles.

Optimization

Stan can find the posterior mode (assuming there is one). If the posterior is not convex, there is no guarantee Stan will be able to find the global mode as opposed to a local optimum of log probability.

For optimization, the mode is calculated without the Jacobian adjustment for constrained variables, which shifts the mode due to the change of variables. Thus modes correspond to modes of the model as written.

Windows

```
> my_model optimize data file=my_data
```

Mac OS and Linux

```
> ./my_model optimize data file=my_data
```

4.2. Diagnostics

Stan has a basic diagnostic feature that will calculate gradients of the initial state and compare them with those calculated with finite differences. If there are discrepancies, there is a problem with the model or initial states (or a bug in Stan). To run on the different platforms, use one of the following.

Windows

```
> my_model diagnose data file=my_data
```

Mac OS and Linux

```
> ./my_model diagnose data file=my_data
```

4.3. Command-Line Options

Stan executables are highly configurable, allowing the user to specify and customize not only the calculation method but also the data, output, initialization, and random number generation. The arguments are defined hierarchically so that, for example, optimization settings are not necessary when sampling.

The atomic elements of the hierarchy (i.e., those without corresponding values) are *categorical arguments* (sometimes called "flags") which define self-contained categories of arguments.

Stan's commands have more hierarchical structure than is typical of command line executables, which usually have at most two subgroups of commands. Arguments grouped within a category are not ordered with respect to each other. The only ordering is that the global options come before the method argument and subcommand-specific options after the method argument. For example, the following four commands all define the same configuration:[2]

```
> ./model sample output file=samples.csv                \
                       diagnostic_file=diagnostics.csv \
                random seed=1

> ./model sample output diagnostic_file=diagnostics.csv \
                       file=samples.csv                \
                random seed=1

> ./model sample random seed=1                          \
                output file=samples.csv                 \
                       diagnostic_file=diagnostics.csv

> ./model sample random seed=1                          \
                output diagnostic_file=diagnostics.csv \
                       file=samples.csv
```

The categorical arguments `output` and `random` can be in any order provided that the subarguments follow their respective parent, here `diagnostic_file` and `file`

[2] The backslash (\) is used at the end of a line in a command to indicate that it continues on the next line. The indentation to indicate the structure of the command is for pedagogical purposes only; the same result would be obtained writing each command on one line with single spaces separating the elements.

following `output` and `seed` coming after `random`. These four configurations exhaust all valid combinations.

Categorical arguments may appear is isolation, for example when introducing `sample` or `random`, or they may appear as the values for other arguments, such as `hmc` which not only introduces a category of HMC related arguments but also defines the value of the argument `algorithm`. A visual diagram of the available categorical arguments is shown in Figure 4.1, with the mutual exclusivity of these arguments as values shown in Figure 4.2. Specifying conflicting arguments causes the execution to immediately terminate.

Note that any valid argument configuration must either specify a method or a help request.

Method

All commands other than `help` must include at least one method, specified explicitly as `method=`*`method_name`* or implicitly with only *`method_name`*. Currently Stan supports the following methods:

Method	*Description*
`sample`	sample using MCMC
`optimize`	find posterior mode using optimization
`diagnose`	diagnose models

All remaining configurations are option, with default values provided for all arguments not explicitly specified.

Help

Informative output can be retrieved either globally, by requesting help at the top-level, or locally, by requesting help deeper into the hierarchy. Note that after any help has been displayed the execution immediately terminates, even if a method has been specified.

Top-Level Help

If `help` is specified as the only argument then a usage message is displayed. Similarly, specifying `help_all` by itself displays the entire argument hierarchy.

Context-Sensitive Help

Specifying `help` after any argument displays a description and valid options for that argument. For example,

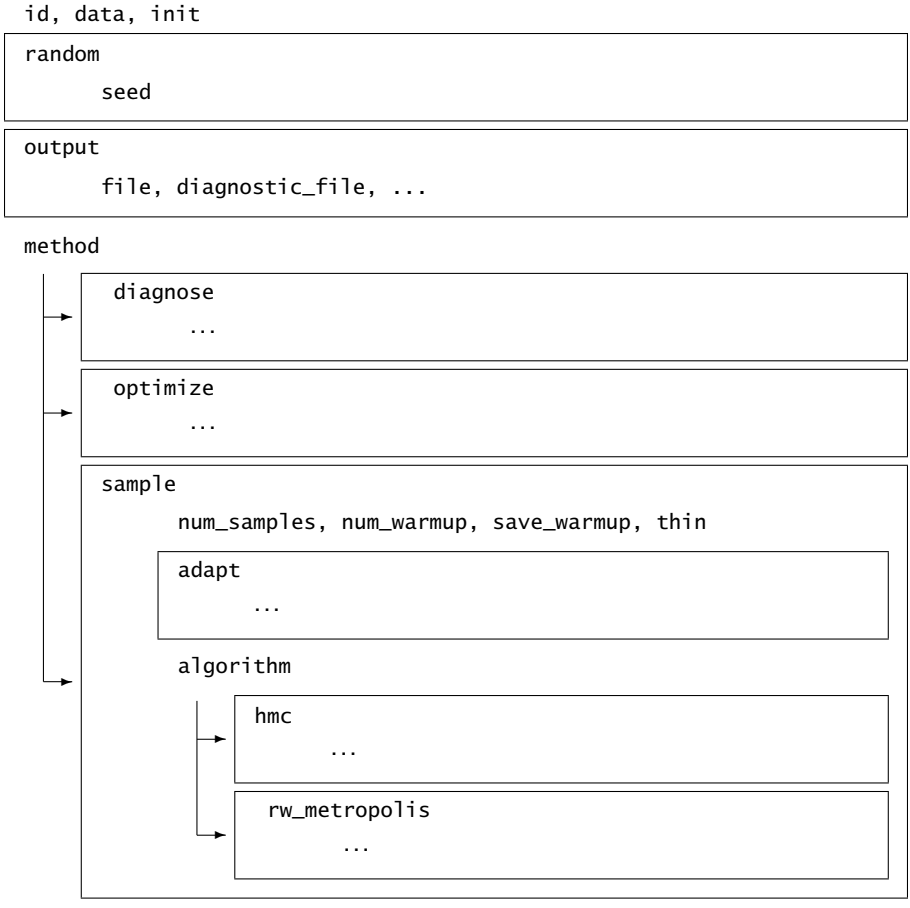

Figure 4.1: *In the hierarchical argument structure, certain arguments, such as* `random` *and* `output`*, introduce new categories of arguments. Categorical arguments may also appear as values of other arguments, such as* `diagnose`*,* `optimize`*, and* `sample`*, which define the mutually exclusive values for the argument* `method`*.*

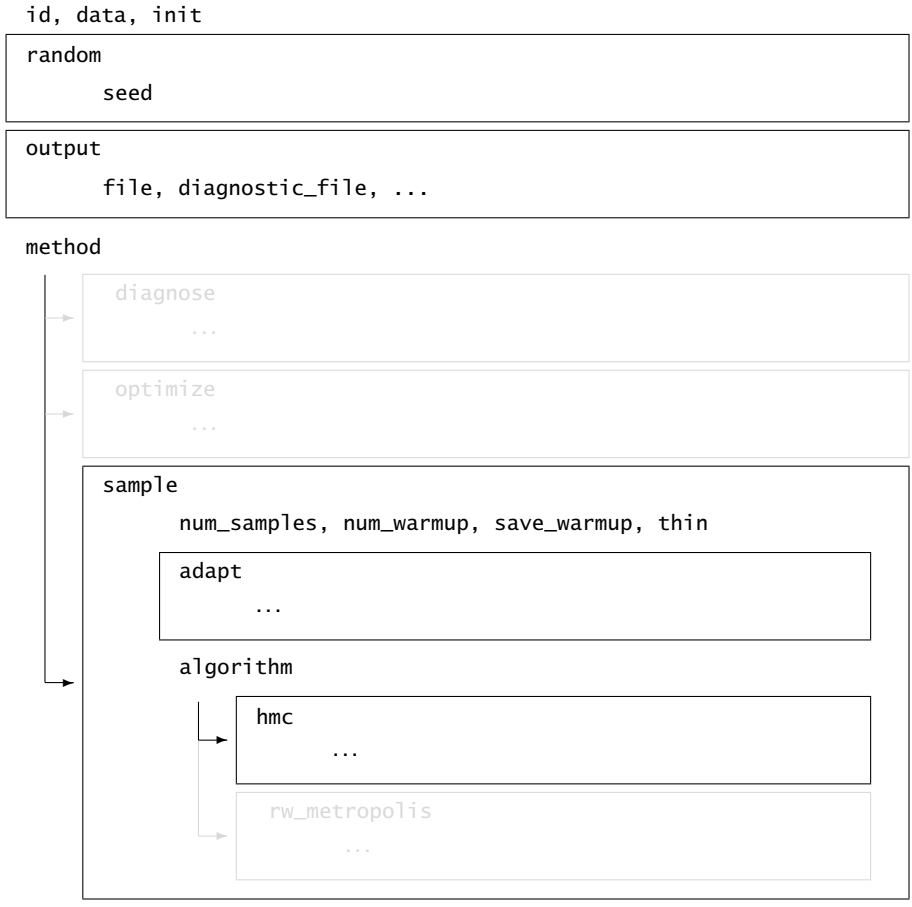

Figure 4.2: *A valid argument configuration defines only one mutually exclusive argument. If conflicting arguments are specified, for example* `method=optimize method=sample`*, then execution immediately terminates with a warning message.*

```
./my_model sample help
```

provides the top-level options for the `sample` method.

Detailed information on the argument, and all arguments deriving from it, can accessed by specifying `help-all` instead,

```
./my_model sample help-all
```

4.4. Full Argument Hierarchy

Here we present the full argument hierarchy, along with relevant details. Some typical use-case examples are provided in the next section.

Typographical Conventions

The following typographical conventions are obeyed in the hierarchy.

- `arg=<value-type>`
  Arguments with values; displays the value type, legal values, and default value
- **`arg`**
  Isolated categorical arguments; displays all valid subarguments
- *`value`*
  Values; describes effect of selecting the value
- ***`avalue`***
  Categorical arguments that appear as values to other arguments; displays all valid subarguments

Top-Level Method Argument

Every command must have exactly one method specified as the very first argument. The value type of `list element` means that the valid values are enumerated as a list.

`method=<list element>`
: Analysis method (Note that `method=` is optional)
  Valid values: `sample, optimize, diagnose`
  (Defaults to `sample`)

Sampling-Specific Arguments

The following arguments are specific to sampling. The method argument `sample` (or `method=sample`) must come first in order to enable the subsequent arguments. The other arguments are optional and may appear in any order.

↳ ***`sample`***
Bayesian inference with Markov Chain Monte Carlo
Valid subarguments: `num_samples, num_warmup, save_warmup, thin, adapt, algorithm`

↳ ↳ `num_samples=`*`<int>`*
Number of sampling iterations
Valid values: $0 \leq$ `num_samples`
(Defaults to 1000)

↳ ↳ `num_warmup=`*`<int>`*
Number of warmup iterations
Valid values: $0 \leq$ `warmup`
(Defaults to 1000)

↳ ↳ `save_warmup=`*`<boolean>`*
Stream warmup samples to output?
Valid values: `0, 1`
(Defaults to 0)

↳ ↳ `thin=`*`<int>`*
Period between saved samples
Valid values: $0 <$ `thin`
(Defaults to 1)

Sampling Adaptation-Specific Parameters

When adaptation is engaged the warmup period is split into three stages (Figure 4.3), with two *fast* intervals surrounding a series of growing *slow* intervals. Here fast and slow refer to parameters that adapt using local and global information, respectively; the Hamiltonian Monte Carlo samplers, for example, define the step size as a fast parameter and the (co)variance as a slow parameter. The size of the the initial and final fast intervals and the initial size of the slow interval are all customizable, although user-specified values may be modified slightly in order to ensure alignment with the warmup period.

Figure 4.3: *Adaptation during warmup occurs in three stages: an initial fast adaptation interval (I), a series of expanding slow adaptation intervals (II), and a final fast adaptation interval (III). For HMC, both the fast and slow intervals are used for adapting the step size, while the slow intervals are used for learning the (co)variance necessitated by the metric. Iteration numbering starts at 1 on the left side of the figure and increases to the right.*

The motivation behind this partitioning of the warmup period is to allow for more robust adaptation. In the initial fast interval the chain is allowed to converge towards the typical set,[3] with only parameters that can learn from local information adapted. After this initial stage parameters that require global information, for example (co)variances, are estimated in a series of expanding, memoryless windows; often fast parameters will be adapted here as well. Lastly the fast parameters are allowed to adapt to the final update of the slow parameters.

Currently all Stan sampling algorithms utilize dual averaging to optimize the step size (this optimization during adaptation of the sampler should not be confused with running Stan's optimization method). This optimization procedure is extremely flexible and for completeness we have exposed each option, using the notation of (Hoffman and Gelman, 2011, 2013). In practice the efficacy of the optimization is sensitive to the value of these parameters, and we do not recommend changing the defaults without experience with the dual averaging algorithm. For more information, see the discussion of dual averaging in (Hoffman and Gelman, 2011, 2013).

Variances or covariances are estimated using Welford accumulators to avoid a loss of precision over many floating point operations.

The following subarguments are introduced by the categorical argument `adapt`. Each subargument must contiguously follow `adapt`, though they may appear in any order.

↳ ↳ **`adapt`**
Warmup Adaptation
Valid subarguments: `engaged, gamma, delta, kappa, t0`

[3] The typical set is a concept borrowed from information theory and refers to the neighborhood (or neighborhoods in multimodal models) of significant posterior probability mass through which the Markov chain will travel in equilibrium.

↳ ↳ ↳ engaged=*<boolean>*
Adaptation engaged?
Valid values: 0, 1
(Defaults to 1)

↳ ↳ ↳ gamma=*<double>*
Adaptation regularization scale
Valid values: 0 < gamma
(Defaults to 0.05)

↳ ↳ ↳ delta=*<double>*
Adaptation target acceptance statistic
Valid values: 0 < delta < 1
(Defaults to 0.8)

↳ ↳ ↳ kappa=*<double>*
Adaptation relaxation exponent
Valid values: 0 < kappa
(Defaults to 0.75)

↳ ↳ ↳ t0=*<double>*
Adaptation iteration offset
Valid values: 0 < t0
(Defaults to 10)

↳ ↳ ↳ init_buffer=*<unsigned int>*
Width of initial fast adaptation interval
Valid values: All
(Defaults to 75)

↳ ↳ ↳ term_buffer=*<unsigned int>*
Width of final fast adaptation interval
Valid values: All
(Defaults to 50)

↳ ↳ ↳ window=*<unsigned int>*
Initial width of slow adaptation interval
Valid values: All
(Defaults to 25)

By setting the acceptance statistic delta to a value closer to 1 (its value must be strictly less than 1 and its default value is 0.8), adaptation will be forced to use smaller step sizes. This can improve sampling efficiency (effective samples per iteration)

at the cost of increased iteration times. Raising the value of `delta` will also allow some models that would otherwise get stuck overcome their blockages; see also the `stepsize_jitter` argument.

Sampling Algorithm- and Engine-Specific Arguments

The following batch of arguments are used to control the sampler used for sampling. The top-level specification is for engine, the only valid value of which is `hmc` (this will change in the future as we add new samplers).

↳ ↳ `algorithm=<list element>`
Sampling algorithm
Valid values: `hmc`, `fixed_param`
(Defaults to `hmc`)

Hamiltonian Monte Carlo is a very general approach to sampling that utilizes techniques of differential geometry and mathematical physics to generate efficient MCMC transitions. This generality manifests in a wealth of implementation choices.

↳ ↳ ↳ ***`hmc`***
Hamiltonian Monte Carlo
Valid subarguments: `engine, metric, stepsize, stepsize_jitter`

All HMC implementations require at least two parameters: an integration step size and a total integration time. We refer to different specifications of the latter as *engines*.

In the `static_hmc` implementation the total integration time must be specified by the user, where as the `nuts` implementation uses the No-U-Turn Sampler to determine an optimal integration time dynamically.

↳ ↳ ↳ ↳ `engine=<list element>`
Engine for Hamiltonian Monte Carlo
Valid values: `static, nuts`
(Defaults to `nuts`)

The following options are activated for static HMC.

↳ ↳ ↳ ↳ ↳ ***`static`***
Static integration time
Valid subarguments: `int_time`

↳ ↳ ↳ ↳ ↳ ↳ `int_time=<double>`
Total integration time for Hamiltonian evolution
Valid values: $0 <$ `int_time`
(Defaults to 2π)

These options are for NUTS, an adaptive version of HMC.

↳ ↳ ↳ ↳ ↳ ***nuts***
The No-U-Turn Sampler
Valid subarguments: `max_depth`

Tree Depth

NUTS generates a proposal by evolving the initial system both forwards and backwards in time to form a balanced binary tree. At each iteration of the NUTS algorithm the tree depth is increased by one, doubling the number of leapfrog steps and effectively doubles the computation time. The algorithm terminates in one of two ways: either the NUTS criterion is satisfied for a new subtree or the completed tree, or the depth of the completed tree hits `max_depth`.

Both the tree depth and the actual number of leapfrog steps computed are reported along with the parameters in the output as `treedepth__` and `n_leapfrog__`, respectively. Because the final subtree may only be partially constructed, these two will always satisfy

$$2^{\text{treedepth}-1} - 1 < N_{\text{leapfrog}} \leq 2^{\text{treedepth}} - 1.$$

`treedepth__` is an important diagnostic tool for NUTS. For example, `treedepth__ = 0` occurs when the first leapfrog step is immediately rejected and the initial state returned, indicating extreme curvature and poorly-chosen step size (at least relative to the current position). On the other hand, if `treedepth__ = max_depth` then NUTS is taking many leapfrog steps and being terminated prematurely to avoid excessively long execution time. For the most efficient sampling `max_depth` should be increased to ensure that the NUTS tree can grow as large as necessary.

For more information on the NUTS algorithm see (Hoffman and Gelman, 2011, 2013).

↳ ↳ ↳ ↳ ↳ ↳ `max_depth=<`*`int`*`>`
Maximum tree depth
Valid values: $0 <$ `max_depth`
(Defaults to 10)

Euclidean Metric

All HMC implementations in Stan utilize quadratic kinetic energy functions which are specified up to the choice of a symmetric, positive-definite matrix known as a *mass matrix* or, more formally, a *metric* (Betancourt and Stein, 2011).

If the metric is constant then the resulting implementation is known as *Euclidean* HMC. Stan allows for three Euclidean HMC implementations: a unit metric, a diagonal metric, and a dense metric. These can be specified with the values `unit_e`, `diag_e`, and `dense_e`, respectively.

Future versions of Stan will also include dynamic metrics associated with *Riemannian* HMC (Girolami and Calderhead, 2011; Betancourt, 2012).

↳ ↳ ↳ ↳ `metric=`*`<list element>`*
Geometry of base manifold
Valid values: `unit_e, diag_e, dense_e`
(Defaults to `diag_e`)

↳ ↳ ↳ ↳ ↳ ↳ *`unit_e`*
Euclidean manifold with unit metric

↳ ↳ ↳ ↳ ↳ ↳ *`diag_e`*
Euclidean manifold with diag metric

↳ ↳ ↳ ↳ ↳ ↳ *`dense_e`*
Euclidean manifold with dense metric

Step Size and Jitter

All implementations of HMC also use numerical integrators requiring a step size. We also allow that step size to be "jittered" randomly during sampling to avoid any poor interactions with a fixed step size and regions of high curvature. The maximum amount of jitter is 1, which will cause step sizes to be selected in the range of 0 to twice the adapted step size. Low step sizes can get HMC samplers unstuck that would otherwise get stuck with higher step sizes. The downside is that jittering below the adapted value will increase the number of leapfrog steps required and thus slow down iterations, whereas jittering above the adapted value can cause premature rejection due to simulation error in the Hamiltonian dynamics calculation. See (Neal, 2011) for further discussion of step-size jittering.

↳ ↳ ↳ ↳ `stepsize=`*`<double>`*
Step size for discrete evolution
Valid values: $0 <$ `stepsize`
(Defaults to 1)

↳ ↳ ↳ ↳ `stepsize_jitter=`*`<double>`*
Uniformly random jitter of the stepsize, in percent
Valid values: $0 \leq$ `stepsize_jitter` ≤ 1
(Defaults to 0)

Fixed Parameter Sampler

The fixed parameter sampler generates a new sample without changing the current state of the Markov chain; only generated quantities may change. This can be useful when, for example, trying to generate pseudo-data using the generated quantities block. If the parameters block is empty (no parameters) then using `algorithm=fixed_param` is mandatory.

↳ ↳ ↳ ***`fixed_param`***
Fixed Parameter Sampler

Optimization-Specific Commands

The following arguments are for the top-level method `optimize`. They allow control of the optimization algorithm, and some of its configuration. The other arguments may appear in any order.

↳ ***`optimize`***
Point estimation
Valid subarguments: `algorithm, iter, save_iterations`

↳ ↳ `algorithm=`*`<list element>`*
Optimization algorithm
Valid values: `bfgs, lbfgs, newton`
(Defaults to `bfgs`)

The following options are for the (L-)BFGS optimizer. BFGS is the default optimizer and also much faster than the other optimizers.

Convergence monitoring in (L-)BFGS is controlled by a number of tolerance values, any one of which being satisified causes the algorithm to terminate with a solution.

- The log probability is considered to have converged if

$$|\log p(\theta_i|y) - \log p(\theta_{i-1}|y)| < \texttt{tol\_obj}$$

or

$$\frac{|\log p(\theta_i|y) - \log p(\theta_{i-1}|y)|}{\max(|\log p(\theta_i|y)|, |\log p(\theta_{i-1}|y)|, 1.0)} < \texttt{tol\_rel\_obj} * \epsilon.$$

- The parameters are considered to have converged if

$$||\theta_i - \theta_{i-1}|| < \texttt{tol\_param}.$$

- The gradient is considered to have converged to 0 if

$$||g_i|| < \texttt{tol\_grad}$$

or

$$\frac{g_i^T \hat{H}_i^{-1} g_i}{\max\left(\left|\log p(\theta_i|y)\right|, 1.0\right)} < \texttt{tol\_rel\_grad} * \epsilon.$$

Here, i is the current iteration, θ_i is the value of the parameters at iteration i, y is the data, $p(\theta_i|y)$ is the posterior probability of θ_i up to a proportion, ∇_θ is the gradient operator with respect to θ, $g_i = \nabla_\theta \log p(\theta_i|y)$ is the gradient at iteration i, $\hat{H}_i$ is the estimate of the Hessian at iteration i, $|u|$ is absolute value (L1 norm) of u, $||u||$ is vector length (L2 norm) of u, and $\epsilon \approx 2e-16$ is machine precision. Any of the convergence tests can be disabled by setting its corresponding tolerance parameter to zero.

The other command-line argument for BFGS is `init_alpha`, which is first step size to try on the initial iteration. If the first iteration takes a long time (and requires a lot of function evaluations) set `init_alpha` to be the roughly equal to the alpha used in that first iteration. `init_alpha` has a tiny default value, which is reasonable for many problems but might be too large or too small depending on the objective function and initialization. Being too big or too small just means that the first iteration will take longer (i.e., require more gradient evaluations) before the line search finds a good step length. It's not a critical parameter, but for optimizing the same model multiple times (as you tweak things or with different data) being able to change it can save some real time.

Finally, L-BFGS has a additional command-line argument, `history_size`, which controls how much memory is used maintaining the approximation of the Hessian. This should be less than the dimensionality of the parameter space and, in general, relatively small values (5 - 10) are sufficient. If L-BFGS performs badly but BFGS is performing well, then consider increasing this. Note that increasing this will increase the memory usage, although this is unlikely to be an issue for typical Stan models.

↳ ↳ ↳ ***(1)bfgs***
(L-)BFGS with linesearch
Valid subarguments: `init_alpha`, `tol_obj`, `tol_rel_obj`, `tol_grad`, `tol_rel_grad`, `tol_param`, `history_size` (lbfgs only)

↳ ↳ ↳ ↳ `init_alpha=`*`<double>`*
Line search step size for first iteration
Valid values: $0 \leq$ `init_alpha`
(Defaults to `0.001`)

↳ ↳ ↳ ↳ `tol_obj=`*`<double>`*
Convergence tolerance on changes in objective function value
Valid values: 0 ≤ `tol_obj`
(Defaults to `1e-12`)

↳ ↳ ↳ ↳ `tol_rel_obj=`*`<double>`*
Convergence tolerance on relative changes in objective function value
Valid values: 0 ≤ `tol_rel_obj`
(Defaults to `1e+4`)

↳ ↳ ↳ ↳ `tol_grad=`*`<double>`*
Convergence tolerance on the norm of the gradient
Valid values: 0 ≤ `tol_grad`
(Defaults to `1e-8`)

↳ ↳ ↳ ↳ `tol_rel_grad=`*`<double>`*
Convergence tolerance on the relative norm of the gradient
Valid values: 0 ≤ `tol_rel_grad`
(Defaults to `1e+7`)

↳ ↳ ↳ ↳ `tol_param=`*`<double>`*
Convergence tolerance on changes in parameter value
Valid values: 0 ≤ `tol_param`
(Defaults to `1e-8`)

↳ ↳ ↳ ↳ `history_size=`*`<int>`*
Number of update vectors to use in Hessian approximations (lbfgs only)
Valid values: 0 < `history_size`
(Defaults to `5`)

The following argument is for Newton's optimization method; there are currently no configuration parameters for Newton's method, and it is not recommended because of the slow Hessian calcuation involving finite differences.

↳ ↳ ↳ ***`newton`***
Newton's method

The remaining arguments apply to all optimizers.

↳ ↳ `iter=`*`<int>`*
Total number of iterations
Valid values: 0 < `iter`
(Defaults to `2000`)

↳ ↳ `save_iterations=<boolean>`
Stream optimization progress to output?
Valid values: `0, 1`
(Defaults to `0`)

Diagnostic-Specific Arguments

The following arguments are specific to diagnostics. As of now, the only diagnostic is gradients of the log probability function.

↳ ***`diagnose`***
Model diagnostics
Valid subarguments: `test`

↳ ↳ `test=<list element>`
Diagnostic test
Valid values: `gradient`
(Defaults to `gradient`)

↳ ↳ ↳ *`gradient`*
Check model gradient against finite differences Valid subarguments: `epsilon`, `error`

↳ ↳ ↳ ↳ `epsilon=<real>`
Finite difference step size
Valid values: 0 < `epsilon`
(Defaults to `1e-6`)

↳ ↳ ↳ ↳ `error=<real>`
Error threshold
Valid values: 0 < `error`
(Defaults to `1e-6`)

General-Purpose Arguments

The following arguments may be used with any of the previous configurations. They may come either before or after the other subarguments of the top-level method.

Process Identifier Argument

`id=<int>`
Unique process identifier, used to advance random number generator so that random numbers do not overlap across chains

Valid values: 0 < `id`
(Defaults to `0`)

Input Data Arguments

`data`
Input data options
Valid subarguments: `file`

↳ `file=<string>`
Input data file
Valid values: Path to existing file
(Defaults to empty path)

Initialization Arguments

Initialization is only applied to parameters defined in the parameters block. Any initial values supplied for transformed parameters or generated quantities are ignored.

`init=<string>`
Initialization method:
- real number $x > 0$ initializes randomly bewteen [-x, x];
- 0 initializes to 0;
- non-number interpreted as a data file

Valid values: All
(Defaults to `2`)

Random Number Generator Arguments

`random`
Random number configuration
Valid subarguments: `seed`

↳ `seed=<unsigned int>`
Random number generator seed
Valid values:
- `seed` ≥ 0 generates seed;
- `seed` < 0 uses seed generated from time

(Defaults to `-1`)

Output Arguments

output
File output options
Valid subarguments: file, diagnostic_file, , refresh

↳ file=<*string*>
Output file
Valid values: Valid path
(Defaults to output.csv)

↳ diagnostic_file=<*string*>
Auxiliary output file for diagnostic information
Valid values: Valid path
(Defaults to empty path)

↳ refresh=<*int*>
Number of interations between screen updates
Valid values: 0 < refresh
(Defaults to 100)

4.5. Command-Line Option Examples

The hierarchical structure of the command-line options can be intimidating, and here we provide an example workflow to help ease the introduction to new users, especially those used to Stan 1.3 or earlier releases. The examples in this section are for Mac OS and Linux; on Windows, just remove the ./ before the executable and change the line-continuation character from Unix's \ to DOS's ^. As in previous sections, the indentation on continued lines is for pedagogical purposes only and does not convey any content to the executable.

Let's say that we've just built our model, model, and are ready to run. We begin by specifying data and init files,

```
> ./model data file=model.data.R init=model.init.R
```

but our model doesn't run. Instead, the above command prints

```
A method must be specified!
Failed to parse arguments, terminating Stan
```

The problem is that we forgot to specify a method.

All Stan arguments have default values, except for the method. This is the only argument that must be specified by the user and a model will not run without it (not to say that the model will run without error, for example a model that requires data will eventually fail unless an input file is specified with `file` under `data`). Assuming that we want to draw MCMC samples from our model, we can either specify a method implicitly,

```
> ./model sample data file=model.data.R init=model.init.R
```

or explicitly,

```
> ./model method=sample data file=model.data.R  \
          init=model.init.R
```

In either case our model now executes without any problem.

Now let's say that we want to customize our execution. In particular we want to set the seed for the random number generator, but we forgot the specific argument syntax. Information for each argument can displayed by calling `help`,

```
> ./model random help
```

which returns

```
random
  Random number configuration
  Valid subarguments: seed
...
```

before printing usage information. For information on the seed argument we just call help one level deeper,

```
> ./model random seed help
```

which returns

```
seed=<unsigned int>
  Random number generator seed
  Valid values: seed > 0, if negative seed is generated from time
  Defaults to -1
  ...
```

Fully informed, we can now run with a given seed,

```
> ./model method=sample data fle=model.data.R  \
          init=model.init.R                    \
          random seed=5
```

The arguments `method`, `data`, `init`, and `random` are all top-level arguments. To really see the power of a hierarchical argument structure let's try to drill down and specify the metric we use for HMC: instead of the default diagonal Euclidean metric, we want to use a dense Euclidean metric. Attempting to specify the metric we try

```
> ./model method=sample data file=model.data.R  \
          init=model.init.R                     \
          random seed=5                         \
          metric=unit
```

only to have the execution fail with the message

```
metric=unit_e is either mistyped or misplaced.
Perhaps you meant one of the following valid configurations?
  method=sample algorithm=hmc metric=<list_element>
Failed to parse arguments, terminating Stan
```

The argument `metric` does exist, but not at the top-level. In order to specify it we have to drill down into sample by first specifying the sampling algorithm, as noted in the suggestion,

```
> ./model method=sample algorithm=hmc metric=unit \
          data file=model.data.R                  \
          init=model.init.R                       \
          random seed=5
```

Unfortunately we still messed up,

```
unit is not a valid value for "metric"
  Valid values: unit_e, diag_e, dense_e
Failed to parse arguments, terminating Stan
```

Tweaking the metric name we make one last attempt,

```
> ./model method=sample algorithm=hmc metric=unit_e \
          data file=model.data.R                    \
          init=model.init.R                         \
          random seed=5
```

which successfully runs.

Finally, let's consider the circumstance where our model runs fine but the NUTS iterations keep saturating the default tree depth limit of 10. We need to change the limit, but how do we specify NUTS let alone the maximum tree depth? To see how

let's take advantage of the `help-all` option which prints all arguments that derive from the given argument. We know that NUTS is somehow related to sampling, so we try

```
> ./model method=sample help-all
```

which returns the verbose output,

```
sample
  Bayesian inference with Markov Chain Monte Carlo
  Valid subarguments: num_samples, num_warmup,
                      save_warmup, thin, adapt, algorithm

  num_samples=<int>
    Number of sampling iterations
    Valid values: 0 <= num_samples
    Defaults to 1000

  num_warmup=<int>
    Number of warmup iterations
    Valid values: 0 <= warmup
    Defaults to 1000

  save_warmup=<boolean>
    Stream warmup samples to output?
    Valid values: [0, 1]
    Defaults to 0

  thin=<int>
    Period between saved samples
    Valid values: 0 < thin
    Defaults to 1

  adapt
    Warmup Adaptation
    Valid subarguments: engaged, gamma, delta, kappa, t0

    engaged=<boolean>
      Adaptation engaged?
      Valid values: [0, 1]
      Defaults to 1

    gamma=<double>
      Adaptation regularization scale
      Valid values: 0 < gamma
```

```
    Defaults to 0.05

  delta=<double>
    Adaptation target acceptance statistic
    Valid values: 0 < delta < 1
    Defaults to 0.65

  kappa=<double>
    Adaptation relaxation exponent
    Valid values: 0 < kappa
    Defaults to 0.75

  t0=<double>
    Adaptation iteration offset
    Valid values: 0 < t0
    Defaults to 10

algorithm=<list element>
  Sampling algorithm
  Valid values: hmc
  Defaults to hmc

  hmc
    Hamiltonian Monte Carlo
    Valid subarguments: engine, metric, stepsize,
                        stepsize_jitter

    engine=<list element>
      Engine for Hamiltonian Monte Carlo
      Valid values: static, nuts
      Defaults to nuts

      static
        Static integration time
        Valid subarguments: int_time

        int_time=<double>
          Total integration time for Hamiltonian evolution
          Valid values: 0 < int_time
          Defaults to 2 * pi

      nuts
        The No-U-Turn Sampler
        Valid subarguments: max_depth
```

```
        max_depth=<int>
          Maximum tree depth
          Valid values: 0 < max_depth
          Defaults to 10

      metric=<list element>
        Geometry of base manifold
        Valid values: unit_e, diag_e, dense_e
        Defaults to diag_e

        unit_e
          Euclidean manifold with unit metric

        diag_e
          Euclidean manifold with diag metric

        dense_e
          Euclidean manifold with dense metric

      stepsize=<double>
        Step size for discrete evolution
        Valid values: 0 < stepsize
        Defaults to 1

      stepsize_jitter=<double>
        Uniformly random jitter of the stepsize, in percent
        Valid values: 0 <= stepsize_jitter <= 1
        Defaults to 0
...
```

Following the hierarchy, the maximum tree depth derives from `nuts`, which itself is a value for the argument `engine` which derives from `hmc`. Adding this to our previous call we attempt

```
> ./model method=sample                          \
              algorithm=hmc                      \
                  metric=unit_e                  \
                  engine=nuts max_depth=-15      \
            data file=model.data.R               \
            init=model.init.R                    \
            random seed=5                        \
```

which yields

```
-1 is not a valid value for "max_depth"
  Valid values: 0 < max_depth
Failed to parse arguments, terminating Stan
```

Where did that negative sign come from? Clumsy fingers are nothing to be embarrassed about, especially with such complex argument configurations. Removing the guilty character, we try

```
> ./model method=sample                    \
              algorithm=hmc                 \
                  metric=unit_e             \
                  engine=nuts max_depth=15  \
          data file=model.data.R            \
          init=model.init.R                 \
          random seed=5
```

which finally runs without issue.

4.6. Command Templates

This section provides templates for all of the arguments deriving from each of the possible methods: `sample`, `optimize`, and `diagnose`. Arguments in square brackets are optional, those not in square brackets are required for the template.

Sampling Templates

The No-U-Turn sampler (NUTS) is the default (and recommended) sampler for Stan. The full set of configuration options is in Figure 4.4.

A standard Hamiltonian Monte Carlo (HMC) sampler with user-specified integration time may also be used. Its set of configuration options are shown in Figure 4.5.

Both NUTS and HMC may be configured with either a unit, diagonal or dense Euclidean metric, with a diagonal metric the default.[4] A unit metric provides no parameter-by-parameter scaling, a diagonal metric scales each parameter independently, and a dense metric also rotates the parameters so that correlated parameters may move together. Although dense metrics offer the hope of superior simulation performance, they require more computation per iteration. Specifically for m samples of a model with n parameters, the dense metric requires $\mathcal{O}(n^3 \log(m) + n^2\, m)$ operations, whereas diagonal metrics require only $\mathcal{O}(n\, m)$. Furthermore, dense metrics are difficult to estimate, given the $\mathcal{O}(n^2)$ components with complex interdependence.

[4] In Euclidean HMC, a diagonal metric emulates different step sizes for each parameter. Explicitly varying step sizes were used in Stan 1.3 and before; Neal (2011) discusses the equivalence.

```
> ./my_model sample                                    \
                algorithm=hmc                          \
                    engine=nuts                        \
                      [max_depth=<int>]                \
                    [metric={unit_e,diag_e,dense_e}]   \
                    [stepsize=<double>]                \
                    [stepsize_jitter=<double>]         \
                [num_samples=<int>]                    \
                [num_warmup=<int>]                     \
                [save_warmup=<boolean>]                \
                [thin=<int>]                           \
                [adapt                                 \
                     [engaged=<boolean>]               \
                     [gamma=<double>]                  \
                     [delta=<double>]                  \
                     [kappa=<double>]                  \
                     [t0=<double>] ]                   \
            [data file=<string>]                       \
            [init=<string>]                            \
            [random seed=<int>]                        \
            [output                                    \
                 [file=<string>]                       \
                 [diagnostic_file=<string>]            \
                 [refresh=<int>] ]
```

Figure 4.4: *Command skeleton for invoking the no-U-turn sampler (NUTS). This is the same skeleton as that for basic HMC in Figure 4.5. Elements in braces are optional. All arguments and their default values are described in detail in Section 4.4.*

```
> ./my_model sample                                    \
                algorithm=hmc                          \
                    engine=static                      \
                      [int_time=<double>]              \
                    [metric={unit_e,diag_e,dense_e}]   \
                    [stepsize=<double>]                \
                    [stepsize_jitter=<double>]         \
                [num_samples=<int>]                    \
                [num_warmup=<int>]                     \
                [save_warmup=<boolean>]                \
                [thin=<int>]                           \
                [adapt                                 \
                     [engaged=<boolean>]               \
                     [gamma=<double>]                  \
                     [delta=<double>]                  \
                     [kappa=<double>]                  \
                     [t0=<double>] ]                   \
            [data file=<string>]                       \
            [init=<string>]                            \
            [random seed=<int>]                        \
            [output                                    \
                 [file=<string>]                       \
                 [diagnostic_file=<string>]            \
                 [refresh=<int>] ]
```

Figure 4.5: *Command skeleton for invoking the basic Hamiltonian Monte Carlo sampler (HMC). This is the same as the NUTS command skeleton shown in Figure 4.4 other than for the engine. Elements in braces are optional. All arguments and their default values are described in detail in Section 4.4.*

```
> ./my_model optimize                                    \
                algorithm=bfgs                           \
                    [init_alpha=<double>]                \
                    [tol_obj=<double>]                   \
                    [tol_rel_obj=<double>]               \
                    [tol_grad=<double>]                  \
                    [tol_rel_grad=<double>]              \
                    [tol_param=<double>]                 \
                [iter=<int>]                             \
                [save_iterations=<boolean>]              \
            [data file=<string>]                         \
            [init=<string>]                              \
            [random seed=<int>]                          \
            [output                                      \
                 [file=<string>]                         \
                 [diagnostic_file=<string>]              \
                 [refresh=<int>] ]
```

Figure 4.6: *Command skeleton for invoking the BFGS optimizer. All arguments and their default values are described in detail in Section 4.4.*

Optimization Templates

Stan supports several optimizers. These share many of their configuration options with the samplers. The default optimizer is the Broyden-Fletcher-Goldfarb-Shanno (BFGS) method and a limited-memory version of this algorithm, L-BFGS, is also implemented (see (Nocedal and Wright, 2006) for more information on BFGS and L-BFGS). The command skeleton for BFGS is in Figure 4.6. Stan also supports Newton's method; see (Nocedal and Wright, 2006) for more information. This method is the least efficient of the three, but has the advantage of setting its own step size. Other than not having a stepsize argument, the skeleton for Newton's method shown in Figure 4.7 is identical to that for BFGS.

Diagnostic Command Skeleton

Stan reports on gradients for the model at a specified or randomly generated initial value. The command-skeleton in this case is very simple, and shown in Figure 4.8.

```
> ./my_model optimize                                    \
                algorithm=newton                         \
                [iter=<int>]                             \
                [save_iterations=<boolean>]              \
            [data file=<string>]                         \
            [init=<string>]                              \
            [random seed=<int>]                          \
            [output                                      \
                 [file=<string>]                         \
                 [diagnostic_file=<string>]              \
                 [refresh=<int>] ]
```

Figure 4.7: *Command skeleton for invoking the Newton optimizer. All arguments and their default values are described in detail in Section 4.4.*

```
> ./my_model diagnose                 \
                [test=gradient]       \
                    [epsilon=<real>]  \
                    [error=<real>]    \
            [data file=<string>]      \
            [init=<string>]           \
            [random seed=<int>]       \
```

Figure 4.8: *Command skeleton for invoking model diagnostics. All arguments and their default values are described in detail in Section 4.4.*

5. Print Command for Output Analysis

Stan is distributed with a print command that is able to read in the output of one or more Markov chains and summarize the posterior fits. This operation mimics the `print(fit)` command in RStan, which itself was modeled on the print functions from R2WinBUGS and R2jags.

5.1. Building the Print Command

Stan's `print` command is built along with `stanc` into the `bin` directory. It can be compiled directly using the makefile as follows from the home directory into which Stan was unpacked (here written as `<stan-home>`).

```
> cd <stan-home>
> make bin/print
```

All the usual compiler options from Stan's makefile apply, such as `O=N` to set optimization level to *N*, and `CC=clang++` to set the compilation to use clang.

5.2. Running the Print Command

The print command is executed on one or more samples.csv files. These files may be provided as command-line arguments separated by spaces. That means that wildcards may be used, as they will be replaced by space-separated file names by the operating system's command-line interpreter.

Suppose there are three samples files in a directory generated by fitting a negative binomial model to a small data set.

```
> ls samples*.csv

samples1.csv        samples2.csv        samples3.csv

> bin/print samples*.csv
```

The result of `bin/print` is displayed in Figure 5.1.[1] The posterior is skewed to the high side, resulting in posterior means ($\alpha = 17$ and $\beta = 10$) that are a long way away from the posterior medians ($\alpha = 9.5$ and $\beta = 6.2$); the posterior median is the value listed under `50%`, which is the 50th percentile of the posterior values.

For Windows, the forward slash in paths need to be converted to backslashes.

[1] RStan's and PyStan's output analysis print may be different than that in the command-line version of Stan.

```
Inference for Stan model: negative_binomial_model
1 chains: each with iter=(1000); warmup=(0); thin=(1); 1000 iterations saved.

Warmup took (0.054) seconds, 0.054 seconds total
Sampling took (0.059) seconds, 0.059 seconds total

                Mean     MCSE   StdDev    5%   50%   95%  N_Eff  N_Eff/s  R_hat
lp__             -14  6.2e-02  1.0e+00   -16   -14   -13    283     4773   1.00
accept_stat__   0.88  5.6e-03  1.8e-01  0.51  0.95   1.0   1000    16881   1.00
stepsize__      0.30  1.3e-15  8.9e-16  0.30  0.30  0.30   0.50      8.5   1.00
treedepth__      1.4  2.6e-02  8.0e-01  0.00   1.0   2.0    946    15978   1.00
n_divergent__    1.4  0.0e+00  0.0e+00  0.00   0.0   0.0   1000    16949   1.00
alpha             17  1.8e+00  2.5e+01   1.9   9.5    50    181     3054   1.00
beta              10  1.1e+00  1.4e+01   1.2   6.2    31    181     3057   1.00

Samples were drawn using hmc with nuts.
For each parameter, N_Eff is a crude measure of effective sample size,
and R_hat is the potential scale reduction factor on split chains (at
convergence, R_hat=1).
```

Figure 5.1: *Example output from* `bin/print`. *The model parameters are* `alpha` *and* `beta`. *The values for each quantity are the posterior means, standard deviations, and quantiles, along with Monte-Carlo standard error, effective sample size estimates (per second), and convergence diagnostic statistic. These values are all estimated from samples. In addition to the parameters, the value* `lp__` *is the total log probability computed by the model (up to an additive constant). The quantity* `accept_stat__` *is the NUTS acceptance statistic used by NUTS for slice and Metropolis rejection,* `stepsize__` *the step size used by NUTS in its Hamiltonian simulation, and* `treedepth__` *is the depth of tree used by NUTS, which is the log (base 2) of the number of leapfrog steps taken during the Hamiltonian simulation.* `n_divergent__` *gives the number of leapfrog iterations with diverging error; because NUTS terminates at the first divergent iteration this should always be either 0 or 1.*

Output of Print Command

`n_divergent`

Stan uses a symplectic integrator to approximate the exact solution of the Hamiltonian dynamics and when the step size is too large relative to the curvature of the log posterior this approximation can diverge and threaten the validity of the sampler. `n_divergent` counts the number of iterations within a given sample that have diverged and any non-zero value suggests that the samples may be biased in which case the step size needs to be decreased. Note that, because sampling is immediately terminated once a divergence is encountered, `n_divergent` should be only 0 or 1.

5.3. Command-line Options

In addition to the filenames, `print` includes three flags to customize the output.

`help`
: Prints usage information
No help output by default

`sig_figs=<int>`
: Sets the number of significant figures displayed in the output
Valid values: 0 <sig_figs
(default = 2)

`autocorr=<int>`
: Calculates and then displays the autocorrelation of the specified chain
Valid values: Any integer matching a chain index
(No autocorrelation output by default)

6. Dump Data Format

For representing structured data in files, Stan uses the dump format introduced in S and used in R and JAGS (and in BUGS, but with a different ordering). A dump file is structured as a sequence of variable definitions. Each variable is defined in terms of its dimensionality and its values. There are three kinds of variable declarations, one for scalars, one for sequences, and one for general arrays.

6.1. Creating Dump Files

Dump files can be created from R using RStan. The function is `stan_rdump` in package `rstan`.

Using R's native `dump()` function can produce dump files which Stan cannot read in. The underlying cause is that R gets creative in the format it uses for output, only being constrained to something that can be executed in R. So it will write the array containing the values 1, 2, 3, 4 as `1:4` rather than as `c(1,2,3,4)`.

6.2. Scalar Variables

A simple scalar value can be thought of as having an empty list of dimensions. Its declaration in the dump format follows the S assignment syntax. For example, the following would constitute a valid dump file defining a single scalar variable `y` with value 17.2.

```
y <-
17.2
```

A scalar value is just a zero-dimensional array value.

6.3. Sequence Variables

One-dimensional arrays may be specified directly using the S sequence notation. The following example defines an integer-value and a real-valued sequence.

```
n <- c(1,2,3)
y <- c(2.0,3.0,9.7)
```

Arrays are provided without a declaration of dimensionality because the reader just counts the number of entries to determine the size of the array.

Sequence variables may alternatively be represented with R's colon-based notation. For instance, the first example above could equivalently be written as

```
n <- 1:3
```

The sequence denoted by `1:3` is of length 3, running from 1 to 3 inclusive. The colon notation allows sequences going from high to low, as in the first of the following examples, which is equivalent to the second.

```
n <- 2:-2
n <- c(2,1,0,-1,-2)
```

6.4. Array Variables

For more than one dimension, the dump format uses a dimensionality specification. For example,

```
y <- structure(c(1,2,3,4,5,6), .Dim = c(2,3))
```

This defines a 2×3 array. Data is stored in column-major order, meaning the values for `y` will be as follows.

```
y[1,1] = 1     y[1,2] = 3     y[1,3] = 5
y[2,1] = 2     y[2,2] = 4     y[2,3] = 6
```

The `structure` keyword just wraps a sequence of values and a dimensionality declaration, which is itself just a sequence of non-negative integer values. The product of the dimensions must equal the length of the array.

If the values happen to form a contiguous sequence of integers, they may be written with colon notation. Thus the example above is equivalent to the following.

```
y <- structure(1:6, .Dim = c(2,3))
```

The same applies to the specification of dimensions, though it is perhaps less likely to be used. In the above example, c(2,3) could be written as `2:3`.

Arrays of more than two dimensions are written in a last-index major form. For example,

```
z <- structure(1:24, .Dim = c(2,3,4))
```

produces a three-dimensional `int` (assignable to `real`) array `z` with values

```
z[1,1,1] =  1   z[1,2,1] =  3   z[1,3,1] =  5
z[2,1,1] =  2   z[2,2,1] =  4   z[2,3,1] =  6

z[1,1,2] =  7   z[1,2,2] =  9   z[1,3,2] = 11
z[2,1,2] =  8   z[2,2,2] = 10   z[2,3,2] = 12
```

```
z[1,1,3] = 13   z[1,2,3] = 15   z[1,3,3] = 17
z[2,1,3] = 14   z[2,2,3] = 16   z[2,3,3] = 18

z[1,1,4] = 19   z[1,2,4] = 21   z[1,3,4] = 23
z[2,1,4] = 20   z[2,2,4] = 22   z[2,3,4] = 24
```

6.5. Matrix- and Vector-Valued Variables

The dump format for matrices and vectors, including arrays of matrices and vectors, is the same as that for arrays of the same shape.

Vector Dump Format

The following three declarations have the same dump format for their data.

```
real a[K];
vector[K] b;
row_vector[K] c;
```

Matrix Dump Format

The following declarations have the same dump format.

```
real a[M,N];
matrix[M,N] b;
```

Arrays of Vectors and Matrices

The key to undertanding arrays is that the array indexing comes before any of the container indexing. That is, an array of vectors is just that — provide an index and get a vector. See the chapter on array and matrix types in the user's guide section of the languag emanual for more information.

For the dump data format, the following declarations have the same arrangement.

```
real a[M,N];
matrix[M,N] b;
vector[N] c[M];
row_vector[N] d[M];
```

Similarly, the following also have the same dump format.

```
real a[P,M,N];
matrix[M,N] b[P];
vector[N] c[P,M];
row_vector[N] d[P,M];
```

6.6. Integer- and Real-Valued Variables

There is no declaration in a dump file that distinguishes integer versus continuous values. If a value in a dump file's definition of a variable contains a decimal point (e.g., `132.3`) or uses scientific notation (e.g., `1.323e2`), Stan assumes that the values are real.

For a single value, if there is no decimal point, it may be assigned to an `int` or `real` variable in Stan. An array value may only be assigned to an `int` array if there is no decimal point or scientific notation in any of the values. This convention is compatible with the way R writes data.

The following dump file declares an integer value for `y`.

```
y <-
2
```

This definition can be used for a Stan variable `y` declared as `real` or as `int`. Assigning an integer value to a real variable automatically promotes the integer value to a real value.

Integer values may optionally be followed by `L` or `l`, denoting long integer values. The following example, where the type is explicit, is equivalent to the above.

```
y <-
2L
```

The following dump file provides a real value for `y`.

```
y <-
2.0
```

Even though this is a round value, the occurrence of the decimal point in the value, `2.0`, causes Stan to infer that `y` is real valued. This dump file may only be used for variables `y` declared as real in Stan.

Scientific Notation

Numbers written in scientific notation may only be used for real values in Stan. R will write out the integer one million as `1e+06`.

Infinite and Not-a-Number Values

Stan's reader supports infinite and not-a-number values for scalar quantities (see the section of the reference manual section of the language manaul for more information on Stan's numerical data types). Both infinite and not-a-number values are supported by Stan's dump-format readers.

Value	*Preferred Form*	*Alternative Forms*
positive infinity	`Inf`	`Infinity`, `infinity`
negative infinity	`-Inf`	`-Infinity`, `-infinity`
not a number	`NaN`	

These strings are not case sensitive, so `inf` may also be used for positive infinity, or `NAN` for not-a-number.

6.7. Quoted Variable Names

In order to support JAGS data files, variables may be double quoted. For instance, the following definition is legal in a dump file.

```
"y" <-
c(1,2,3)
```

6.8. Line Breaks

The line breaks in a dump file are required to be consistent with the way R reads in data. Both of the following declarations are legal.

```
y <- 2
y <-
3
```

Also following R, breaking before the assignment arrow are not allowed, so the following is invalid.

```
y
<- 2  # Syntax Error
```

Lines may also be broken in the middle of sequences declared using the `c(...)` notation., as well as between the comma following a sequence definition and the dimensionality declaration. For example, the following declaration of a $2 \times 2 \times 3$ array is valid.

```
y <-
structure(c(1,2,3,
4,5,6,7,8,9,10,11,
12), .Dim = c(2,2,
3))
```

Because there are no decimal points in the values, the resulting dump file may be used for three-dimensional array variables declared as `int` or `real`.

6.9. BNF Grammar for Dump Data

A more precise definition of the dump data format is provided by the following (mildly templated) Backus-Naur form grammar.

```
definitions ::= definition+

definition ::= name ("<-" | '=') value optional_semicolon

name ::= char*
       | ''' char* '''
       | '"' char* '"'

value ::= value<int> | value<double>

value<T> ::= T
           | seq<T>
           | 'structure' '(' seq<T> ',' ".Dim" '=' seq<int> ')'

seq<int> ::= int ':' int
           | cseq<int>

seq<real> ::= cseq<real>

cseq<T> ::= 'c' '(' vseq<T> ')'

vseq<T> ::= T
          | T ',' vseq<T>
```

The template parameters `T` will be set to either `int` or `real`. Because Stan allows promotion of integer values to real values, an integer sequence specification in the dump data format may be assigned to either an integer- or real-based variable in Stan.

Appendices

A. Licensing

Stan and its two dependent libraries, Boost and Eigen, are distributed under liberal freedom-respecting[1] licenses approved by the Open Source Initiative.[2]

In particular, the licenses for Stan and its dependent libraries have no "copyleft" provisions requiring applications of Stan to be open source if they are redistributed.

This chapter describes the licenses for the tools that are distributed with Stan. The next chapter explains some of the build tools that are not distributed with Stan, but are required to build and run Stan models.

A.1. Stan's License

Stan is distributed under the BSD 3-clause license (BSD New).

http://www.opensource.org/licenses/BSD-3-Clause

A.2. Boost License

Boost is distributed under the Boost Software License version 1.0.

http://www.opensource.org/licenses/BSL-1.0

A.3. Eigen License

Eigen is distributed under the Mozilla Public License, version 2.

http:/http://opensource.org/licenses/mpl-2.0

A.4. Google Test License

Stan uses Google Test for unit testing; it is not required to compile or execute models. Google Test is distributed under the BSD 2-clause license.

http://www.opensource.org/licenses/BSD-License

[1] The link http://www.gnu.org/philosophy/open-source-misses-the-point.html leads to a discussion about terms "open source" and "freedom respecting."

[2] See http://opensource.org.

B. Installation and Compatibility

This appendix describes the hardware and software required to run Stan. The software includes Stan and its libraries, as well as a contemporary C++ compiler. Stan requires hardware powerful enough to build and execute the models. Ideally, that will be a 64-bit computer with at least 4GB of memory and multiple processor cores.

B.1. Operating System

Stan is written in portable C++ without C++11 features, as are the libraries on which it depends. Therefore, Stan should run on any machine for which a suitable C++ compiler is available. In practice, Stan, like the Boost and Eigen libraries on which it depends, is very hard on the compiler and linker.

Stan has been tested on the following operating systems.

- Linux (Debian, Ubuntu, Red Hat),
- Mac OS X (Snow Leopard, Lion, Mountain Lion), and
- Windows (XP, 7, 8).

Stan should work on other versions of these operating systems if compatible C++ compilers can be found. The plan is to keep up with new versions of these operating systems and gradually phase out testing on older versions.

B.2. Step-by-Step Mac Install Instructions

This section provides step-by-step install instructions for the Mac; Linux and Windows sections follow. It repeats the step-by-step install instructions on Stan's home page at `http://mc-stan.org/`.

Stan has been tested on Mac OS X versions Snow Leopard, Lion, and Mountain Lion.

Tips for Mac Users

Finding and Opening Mac Applications and Files

To open an application, use `[Command-Space]` (press both keys at once on the keyboard) to open Spotlight, enter the application's name in the text field, then click on the application in the pop-up menu or `[Return]` if the right file or application is highlighted.

Spotlight can be used in the same way to find files or folders, such as the default `Downloads` folder for web downloads.

Open a Terminal for Shell Commands

To run shell commands, open the built-in Terminal application (see the previous subsection for details on how to find and open applications).

Install Xcode C++ Development Environment

The easiest (but not the only) way to install a C++ development environment on a Mac is to use Apple's Xcode development environment.

From the Xcode home page,

```
https://developer.apple.com/xcode/
```

click `View in Mac App Store`.

From the App Store, click `Install`, enter an Apple ID, and wait for Xcode to finish installing.

Open the Xcode application, click top-level menu `Preferences`, click top-row button `Downloads`, click button for `Components`, click on the `Install` button to the right of the `Command Line Tools` entry, then wait for it to finish installing.

Click the top-level menu item `Xcode`, then click item `Quit Xcode` to quit.

To test, open the Terminal application and enter

```
> make --version
> g++ --version
```

Verify that `make` is at version 3.81 or later and `g++` is at 4.2.1 or later.

Download and Unpack Stan Source

Download the most recent version of `stan-2.m.p.tar.gz` (`m` is the minor version and `p` the patch level) from the Stan downloads list,

```
https://github.com/stan-dev/stan/releases
```

Open the folder containing the download in the Finder (typically, the user's top-level `Downloads` folder).

If the Mac OS has not automatically unpacked the `.tar.gz` file into file `stan-2.m.p.tar`, double-click the `.tar.gz` file to unpack.

Double click on the `.tar` file to unarchive directory `stan-2.m.p`.

Move the resulting directory to a location where it will not be deleted, henceforth called `<stan-home>`.

B.3. Step-by-Step Linux Install Instructions

Stan has been tested on various Linux installations, including Ubuntu, Debian, and Red Hat.

Installing C++ Development Tools

On Linux, C++ compilers and `make` are often installed by default.

To see if the `g++` compiler and `make` build system are already installed, use the commands

```
> g++ --version
```

and

```
> make --version
```

If these are at least at `g++` version 4.2.1 or later and `make` version 3.81 or later, no additional installations are necessary. It may still be desirable to update the C++ compiler `g++`, because later versions are faster.

To install the latest version of these tools (or upgrade an older version), use the commands

```
> sudo apt-get install g++
```

and

```
> sudo apt-get install make
```

A password will likely be required by the superuser command `sudo`.

Downloading and Unpacking Stan Source

Download the most recent stable version of Stan, `stan-2.m.p.tar.gz`, where `m` is the minor version and `p` the patch level), from the Stan downloads page,

`https://github.com/stan-dev/stan/releases`

to the directory where Stan will reside.

In a command shell, change directories to where the tarball was downloaded, say `<download-dir>`, with

```
> cd <download-dir>
```

where `<download-dir>` is replaced with the actual path to the directory.

Then, unpack the distribution into the subdirectory

```
<download-dir>/stan-2.m.p
```

with

```
> tar -xzf stan-2.m.p.tar.gz
```

B.4. Step-by-Step Windows Install Instructions

Stan has been tested on Windows XP, Windows 7, and Windows 8.

Stan also runs under Cygwin, which provides a unix-like shell on top of Windows. Instructions for Cygwin installation are provided below in their own subsection.

Windows Tips

Opening a Command Shell

To open a Windows command shell, first open the `Start Menu` (usually in the lower left of the screen), select option `All Programs`, then option `Accessories`, then program `Command Prompt`.

Alternatively, enter `[Windows+r]` (both keys together on the keyboard), and enter `cmd` into the text field that pops up in the Run window, then press `[Return]` on the keyboard to run.

32-bit Builds

Stan defaults to a 64-bit build. On a 32-bit operating system, set the `BIT` variable to 32. For example, to build the Bernoulli model in Section 2.4, replace the original command with:

```
> make BIT=32 src/models/basic_estimators/bernoulli
```

Rtools C++ Development Environment

The simplest way to install a full C++ build environment that will work for Stan is to use the Rtools package designed for R developers on Windows (even if you don't plan to use R).

First, download the latest *frozen* (i.e., stable) version of Rtools from the Rtools home page, using

`http://cran.r-project.org/bin/windows/Rtools/`

Next, double click on the downloaded file to open the Rtools install wizard, then proceed through its options.

- *Language*: select language, click `Next`,
- *Welcome*: click `Next`,
- *Information*: click `Next`,
- *Setup Location*: accept default (`c:\Rtools`), click `Next`,
- *Select Components*: select default, `Package Authoring`, click `Next`,
- *Select Additional Tasks*: check `Edit Path` and `Save Version in Registry`, click `Next`,
- *System Path Report*: ensure that that the paths to `c:\Rtools\bin` and `c:\Rtools\gcc-4.6.3\bin` are listed at the beginning of the path and click `Next`,
- *Ready to Install*: click `Next`, wait for the install to complete, then
- *Finish*: click `Finish`.
- *Confirm Path*: After the install has completed, open a command prompt and type `PATH` to ensure that the new path is activated and the Rtools folders are in the system path.

Checking the Path

Make sure that `c:\Rtools\bin` has been added to your `PATH` environment variable, and then open another command window. You should be able to follow the last step, *Comfirm Path*, above.

Downloading and Unpacking Stan

The Stan source code distributions are named `stan-2.m.p.tar.gz`, where `m` is the minor version and `p` the patch level.

Download the latest Stan source from the Stan downloads page,

`https://github.com/stan-dev/stan/releases`

to any non-temporary folder. (If in doubt, select `My Documents` on Windows XP or `Documents` on Windows 7.)

Change to the download directory (aka folder) using one of the following commands, replacing `<username>` with a Windows user name.

- *Windows XP*: From the default starting directory, use the following commands (quotes and all):

```
> cd "My Documents"
```

The full path (including quotes) will work from anywhere,

```
> cd "c:\Documents and Settings\<username>\My Documents"
```

- *Windows 7*: From the default starting directory, use

```
> cd Documents
```

or use the full path, including quotes, from anywhere,

```
> cd "c:\Users\<username>\Documents"
```

To verify that the downloaded Stan `.tar.gz` file is there, list the directory contents using:

```
> dir
```

Finally, unpack the distribution using the `tar` command (which is installed as part of Rtools).

```
> tar --no-same-owner -xzf stan-2.m.p.tar.gz
```

The `-no-same-owner` flag is not strictly necessary, but it removes a bunch of irrelevant warnings.

64-bit Cygwin Install Instructions

Stan can be run under Cygwin, the Unix look-and-feel environment for Windows. Cygwin must have recent versions of `make` and `g++` (part of gcc) installed. Within a Cygwin shell, Stan will behave as under other Unixes.

Thanks to Kevin van Horn for mailing the following instructions into the Stan-users mailing list. They only cover 64-bit R and 64-bit Cygwin, but that is what you should be using for Stan anyway.

1. Kill all Cygwin bash shells and shut down R.
2. After installing R and Rtools, make sure that R and Rtools are in the `PATH` environment variable. My R installation directory was `c:\Program Files\R\R-3.0.1` and my Rtools installation directory was `c:\Rtools`, so I added the following to the end of my user `PATH` variable:
   - `C:\Program Files\R\R-3.0.1\bin\x64`
   - `C:\Rtools\bin`

- C:\Rtools\gcc-4.6.3\bin

3. Now there could be a conflict between Cygwin and Rtools when running a bash shell under Cygwin, so I added the following lines to my `.bash_profile` file to remove any `PATH` directory referencing Rtools:

```
> TMP=`echo $PATH | /usr/bin/tr ':' '\n'                      \
                  | /usr/bin/egrep -iv '^/cygdrive/c/Rtools/' \
                  | tr '\n' ':'``
> PATH=${TMP%:}
```

   (Note that the backslash characters signal that the line continues after a return.) This was only necessary to allow me to continue using Cygwin.

4. Apparently there is something in `Rcpp` or `inline` or `rstan` that doesn't like `UNC` paths. My home directory was `\\server\users\kevinv` and apparently this caused my local R library directory to be `\server\users\kevinv\R` as verified by `.libPaths()` from the R command prompt.

   I fixed this by copying the entire directory tree rooted at `\server\users\kevinv\R` over to a local directory on my workstation, `C:\Users\KevinV\R`, then adding the user environment variable `R_LIBS_USER=C:\Users\KevinV\R`. I shut down R and restarted it.

5. At this point the instructions given for installing Rstan finally worked.

MKL Compiler Instructions

Getting the MKL

To purchase a license, see

http://software.intel.com/en-us/intel-mkl

For non-commercial development, see

http://software.intel.com/non-commercial-software-development

Installing and Compiling with MKL

In order to use Intel's math kernel library (MKL) for C++,

- Download and extract a fresh copy of Stan.
- In your makefile change `CC=g++` to `CC=icc`; or you can do this by supplying the argument `CC=icc` to each call to make (aliases are good for this).

- Add the MKL path to the makefile; for example

```
MKLROOT = /apps/intel/2013/mkl)
```

- Add the following to makefile's CFLAGS:

```
-I $(MKLROOT)/include and -DEIGEN_USE_MKL_ALL
```

- Link to your MKL library by adding to your makefile's LDLIBS. The exact implementation will depend on your system. Use the MKL link line advisor for help. For example, you might add

```
-L$(MKLROOT)/lib/intel64 -lmkl_intel_lp64
   -lmkl_core -lmkl_sequential -lpthread -lm
```

- Compile your models as usual, for example

```
make src/models/speed/logistic/logistic
```

Note: Make sure to do the above changes before compiling for the first time - otherwise Stan will be compiled with g++ and you won't see any performance gains.

B.5. Required Software and Tools

The only two absolute requirements for running Stan are the Stan source code (and dependent libraries) and a C++ compiler.

Stan Source

In order to compile Stan models, the Stan source code is required. The latest version of Stan can be downloaded from the following link.

http://mc-stan.org/

The Stan source code distribution includes Stan's source code, documentation, build tools, unit tests, demo models, documentation and source for the required libraries Boost and Eigen, and the source for an optional testing library, Google Test.

Boost C++ Library Source

Stan's parser and some of its mathematical functions and template metaprogramming facilities are implemented with the Boost C++ Library.

- Home: http://www.boost.org/users/license.html

- License: Boost Software License
- Tested Version: 1.54.0

The Boost source code is distributed with Stan.

Eigen Matrix and Linear Algebra Library Source

Stan's matrix algebra depends on the Eigen C++ matrix and linear algebra library.

- Home: http://eigen.tuxfamily.org
- License: Mozilla Public License, version 2.0
- Tested Version: 3.2.0

The Eigen source code is distributed with Stan.

C++ Compiler

Compiling Stan models requires a C++ compiler. Stan has been primarily developed with `clang++` and `g++` and no promises are made for other compilers. The full set of compilers for which Stan has been tested is

- `g++`
  Tested Versions: Mac 4.2.1, 4.6, Linux 4.4–4.7 (plus trunk 4.8, 4.9), Windows 4.6.3
  Home: http://gcc.gnu.org/
  License: GPL3+
- `clang++`, Mac 2.9–3.1, Linux 2.9–3.1
  Home: http://clang.llvm.org/
  License: BSD
- mingw-64, version 2.0 (Windows 7, cross-compiled from Debian Linux)
- Intel C++, Linux version 12.1.3

C++-11 Support

Stan 2.0 does not support C++-11. The remaining incompatibility with the parser will be included soon after Stan 2.0 is released. This will include support for the latest versions of `g++` and `clang++`.

B.6. Optional Components for Developers

Stan is developed using the following set of tools. The various command examples in this manual have assumed they can be found on the command path. The makefile allows precise locations to be plugged in.

GNU Make Build Tool

Stan automates the build, test, documentation, and deployment tasks using scripts in the form of makefiles to run with GNU Make.

- Home: http://www.gnu.org/software/make
- License: GPLv3+
- Tested Versions: 3.81 (Mac OS X), 3.79 (Windows 7)

Doxygen Documentation Generator

Stan's API documentation is generated using the Doxygen Tool.

- Home: http://www.stack.nl/~dimitri/doxygen/index.html
- License: GPL2
- Tested Version(s): Mac OS X 1.8.2, Windows 1.8.2

Git Version Control System

Stan uses the Git version control system for its software, libraries, and documentations. Git is required to interact with the most recent versions of code in the version control repository.

- Home: http://git-scm.com/
- License: GPL2
- Tested Version(s): Mac version 1.7.8.4, Windows version 1.7.9

Google Test C++ Testing Framework

Stan's unit testing is based on the Google's googletest C++ testing framework.

- Home: http://code.google.com/p/googletest/
- License: BSD

- Tested Version(s): 1.6.0

The Google Test framework is distributed with Stan.

B.7. Tips for Mac OS X

Install Xcode

Apple's Xcode contains both the `clang++` and `g++` compilers and make, all of the tools needed to work with Stan as a user. The version of Xcode to install depends on the version of Mac OS X.

Official Apple Xcode Distribution

Xcode 4 may be downloaded for free for Mac OS X 10.7 ("Lion") or later directly from Apple:

Xcode 4: `https://developer.apple.com/xcode/`

Once you've installed Xcode, you need to start it, then open menu option `Xcode`, select `Preferences`, then click on the `Downloads` icon and then click on the `Install` button next to the option labeled "Command Line Tools."

At this point, you should have the make system `make` and the two C++ compilers/linkers, `g++` and `clang++`, installed. This is all you need to run Stan. Xcode will also install the `git` version control system at this point.

Alternative, GCC-Only Installer

A stripped down installer for just the GCC package, including the C++ compilers `g++` and `clang++`, available for Mac OS X 10.6 ("Snow Leopard") or later,

`https://github.com/kennethreitz/osx-gcc-installer/`

The fill list of tools in this distribution is available at:

`http://www.opensource.apple.com/release/developer-tools-41/`

More Recent Compilers

Alternative compilers to those distributed by Apple as part of Xcode are available at the following locations.

Homebrew

One way to get pre-built binaries for Mac OS X is to use Homebrew, which is available from the following link.

> http://mxcl.github.com/homebrew/

MacPorts

MacPorts hosts recent versions of compilers for the Macintosh.

> https://distfiles.macports.org/MacPorts/

After finding the appropriate `.dmg` file, clicking on it, then double clicking on the resulting `.pkg` file, and clicking through some more menus, the following will need to be entered from a terminal window to install it.

```
> sudo port install gccVersion
```

In this command, *gccVersion* is the name of a compiler version, such as `g++-mp-4.6`, for version 4.6. Errors may arise during the install such as the following.

```
Error:  Target org.macports.activate returned:  Image error:
/opt/local/include/gmp.h already exists and does not belong to
a registered port.  Unable to activate port gmp.  Use 'port -f
activate gmp' to force the activation.
```

This issue can be resolved by running the following command.

```
> sudo port -f activate gmp
```

Git Installer

A standalone version of Git for Mac OS X is available from the following site.

> http://code.google.com/p/git-osx-installer/

Although (at the time of this writing) there were only versions listed up to OS X version "Snow Leopard," they work on "Lion."

LaTeX Typesetting Package

Stan uses the LaTeX typesetting package for generating manuals, talks, and other materials (Doxygen is used for API documentation; see below). The first step is to download the MacTeX `.mpkg` file from the following URL [warning: the download is approximately 2GB and the installation approximately 3.5GB].

http://www.tug.org/mactex/2011/

Once it is downloaded, just click on the `.mpkg` file and then follow the installer instructions. The installer will add the command to the `PATH` environment variable so that the `pdflatex` used by Stan is available from the command line.

Lucida Console Font

A free TrueType version of Lucida Console for the Mac is available at the following URL.

http://www.fontpalace.com/font-details/Lucida+Console/

Download the `.ttf` file, then click on it to install. It will then be available as a preference in the Mac terminal application.

Doxygen API Documentation

Stan's API documentation is generated using the Doxygen tool. This tool is available from

http://www.doxygen.org

Select the `Download` link from the second of the right-hand side navigation bars, then select the binary distribution `.dmg` file for Mac OS X. Clicking on the `.dmg` file opens the finder with a view of the unpacked Doxygen executable. Just drag the Doxygen icon into the Applications folder (or wherever you want to keep it). Then add the path to the Doxygen executable,

```
/Applications/Doxygen.app/Contents/Resources/doxygen
```

to the system `PATH` environment variable. You can do add to the `PATH` environment by adding this line to the end of the top-level `~/.profile` file.

```
export PATH=/Applications/Doxygen.app/Contents/Resources:$PATH
```

The next shell started will then be able to find the `doxygen` command.

B.8. Tips for Windows

Install Rtools

The easiest way to get a complete C++ build environment on Windows is to install the most recent version of Rtools.

The latest version verified to work with Stan is Rtools 2.15. Rtools 2.15 includes the g++ 4.6.3 (pre-release) compiler and many other useful command line tools including many Unix commands, such as the following.

```
basename, cat, cmp, comm, cp, cut, date, diff, du, echo,
expr, gzip, ls, make, makeinfo, mkdir, mv, rm, rsync, sed,
sh, sort, tar, texindex, touch, uniq
```

Rtools can be downloaded from the following location.

```
http://cran.r-project.org/bin/windows/Rtools/
```

Install it using the Windows installer. Allow it to edit the PATH environment variable so that commands are available from the command tool.

To verify the installation was successful, open a command window by selecting the following menu items.

Start → Accessories → Command Prompt

To verify that g++ is installed, use the following command.

```
> g++ -v
```

This should report version information for g++. Next, verify that make is installed with the following command.

```
> make -v
```

This should print version information for make.

GNU Make 3.81 or Higher for Tests

Although the version of make distributed with RTools suffices to run Stan, in order to run the unit tests (see Section 2.4.8), a version of GNU make version 3.81 or higher is required. To install such a version of make:

- Install Rtools according to instructions in the previous section.
- Download and install make 3.81 (or higher).
  - we have tested the version installed through the Setup link at http://gnuwin32.sourceforge.net/packages/make.htm
- Edit the PATH environment variable so the installation location of make appears before RTools.
- Remove or move Windows' find.exe from C:\Windows system32. For example, move it to the top-level C: directory.

The reason for the last step is a bug in this version of make. Even though it should pick out find.exe from Rtools, it picks out C:\Windows\system32 first.

Install Git

There are a number of Git clients for Windows that will work. The official Git installer for Windows can be found at the following location.

```
http://code.google.com/p/msysgit/downloads
```

Select the latest full installer and install it.

Bibliography

Betancourt, M. (2012). A general metric for Riemannian manifold Hamiltonian Monte Carlo. *arXiv*, 1212.4693. 39

Betancourt, M. and Stein, L. C. (2011). The geometry of Hamiltonian Monte Carlo. *arXiv*, 1112.4118. 38

Girolami, M. and Calderhead, B. (2011). Riemann manifold Langevin and Hamiltonian Monte Carlo methods. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 73(2):123–214. 39

Hoffman, M. D. and Gelman, A. (2011). The no-U-turn sampler: Adaptively setting path lengths in Hamiltonian Monte Carlo. *arXiv*, 1111.4246. 35, 38

Hoffman, M. D. and Gelman, A. (2013). The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo. *Journal of Machine Learning Research*, in press. 35, 38

Neal, R. (2011). MCMC using Hamiltonian dynamics. In Brooks, S., Gelman, A., Jones, G. L., and Meng, X.-L., editors, *Handbook of Markov Chain Monte Carlo*, pages 116–162. Chapman and Hall/CRC. 39, 51

Nocedal, J. and Wright, S. J. (2006). *Numerical Optimization*. Springer-Verlag, Berlin, second edition. 54