CmdStan User's Guide

Version 2.23

Stan Development Team

Contents

Introduction

This document is a user's guide for CmdStan, the command-line interface. to the Stan statistical modeling language. CmdStan is the command-line interface for Stan. CmdStan provides the tools to compile a statistical model written in the Stan probabalistic programming language into a C++ executable program which can then be run to either: do inference on data, producing an estimate of the posterior; generate new quantities of interest from an existing estimate; or generate data from the model according to a given set of parameters

CmdStan provides the programs and tools to compile Stan programs into C++ executables that can be run directly from the command line, together with a few utilities to check and summarize the resulting outputs. CmdStan is one of several interfaces to Stan; there are also R, Python, Matlab, Julia, and Stata interfaces.

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/

Licensing

CmdStan, Stan, and the Stan Math Library are licensed under the new BSD license (3-clause). See the Stan Reference Manual Licenses section for licensing terms for Stan and the dependent packages Boost, Eigen, Sundials, and Intel TBB.

Stan Documentation: User's Guide and Reference Manuals

The Stan user's guide provides example models and programming techniques for coding statistical models in Stan. It also serves as an example-driven introduction to Bayesian modeling and inference:

http://mc-stan.org/docs/stan-users-guide

Stan's modeling language is shared across all of its interfaces. The Stan Language Reference Manual provides a concise definition of the language syntax for all elements in the language.

http://mc-stan.org/docs/reference-manual

The Stan Functions Reference provides definitions and examples for all the functions defined in the Stan math library and available in the Stan programming language, including all probability distributions.

http://mc-stan.org/docs/functions-reference.

Benefits of CmdStan

- With every new Stan release, there is a corresponding CmdStan release, therefore CmdStan provides access to the latest version of Stan, and can be used to run the development version of Stan as well.
- Of the Stan interfaces, CmdStan has the lightest memory footprint, therefore it can fit larger and more complex models. It has has the fewest dependencies, which makes it easier to run in limited environments such as clusters.
- The output generated is in CSV format and can be post-processed using other Stan interfaces or general tools.

QuickStart Guide

This section is designed to help users install CmdStan and get acquainted with the CmdStan interface.

1. CmdStan Installation

To install CmdStan you need:

- A modern C++11 compiler. Supported versions are • Linux: g++ 4.9.3 or clang 6.0 • macOS: the XCode version of clang • Windows: g++ 8.1 (available with RTools 4.0) is recommended; alternatively, g++ 4.9.3 (available with RTools 3.5).
- The GNU-Make utility program or the Windows equivalent mingw32-make. On macOS, this is part of the XCode command line tools installed via command xcode-select --install. On Windows, mingw32-make is installed as part of RTools: https://cran.rstudio.com/bin/windows/Rtools/.
- The CmdStan C++ source code and libraries. The most recent CmdStan release is available as a single compressed tarfile containing all of the CmdStan tools and the Stan and math libraries from GitHub: https://github.com/stan-dev/cmdstan/releases/latest or you can clone the GitHub repo.

The CmdStan release unpacks into a directory called cmdstan-<version> where the version string consists of the major.minor.patch version numbers, e.g. cmdstan-2.23.0. Cloning CmdStan from GitHub creates a directory simply called cmdstan. Throughout this manual, we refer to this top-level CmdStan source directory as **<cmdstan-home>**.

1.1. GNU-Make Utility

CmdStan relies on the GNU-make utility to build both the Stan model executables and the CmdStan tools.

GNU-Make builds executable programs and libraries from source code by reading files called Makefiles which specify how to derive the target program. A Makefile consists of a set of recursive rules where each rule specifies a target, its dependencies, and the specific operations required to build the target. Specifying dependencies for a target provides a way to control the build process so that targets which depend on other files will be updated as needed *only* when there are changes to those other files. Thus Make provides an efficient way to manage complex software.

The CmdStan Makefile is in the <cmdstan-home> directory and is named makefile. This is one of the default GNU Makefile names, which allows you to omit the -f makefile argument to the Make command. Because the CmdStan Makefile includes several other Makefiles, **Make only works properly when invoked from the**

<cmdstan-home> directory; attempts to use this Makefile from another directory by specifying the full path to the file `makefile` won't work. For example, trying to call Make from another directory by specifying the full path the the makefile results in the following set of error messages:

```
make -f ~/github/stan-dev/cmdstan/makefile
/Users/mitzi/github/stan-dev/cmdstan/makefile:58: make/stanc: No such file o
/Users/mitzi/github/stan-dev/cmdstan/makefile:59: make/program: No such file
/Users/mitzi/github/stan-dev/cmdstan/makefile:60: make/tests: No such file o
/Users/mitzi/github/stan-dev/cmdstan/makefile:61: make/command: No such file
make: *** No rule to make target `make/command'.  Stop.
```

Makefile rules can be written as general pattern rules based on file suffixes. The Stan makefile rules specify how to process Stan program files with suffix `.stan` into executable files. For example, to compile the Stan program `my_program.stan` in directory `../my_dir/`, the make target is `../my_dir/my_program` or `../my_dir/my_program.exe` (on Windows).

Make is invoked with a list of target names. Makefile targets can be preceded by zero or more Makefile variable name=value pairs. For example to compile `../my_dir/my_program.stan` for an OpenCL (GPU) machine, the makefile variable `STAN_OPENCL` is set to `TRUE`:

```
> make STAN_OPENCL=TRUE ../my_dir/my_program
```

Makefile variables can also be set by creating a file named `local` in the CmdStan `make` subdirectory which contains a list of `<VARIABLE>=<VALUE>` pairs, one per line. The complete set of Makefile variables can be found in file `cmdstan/stan/lib/stan_math/make/compiler_flags`.

When invoked without any arguments at all, Make prints a help message:

```
> make
------------------------------------------------------------------------------
CmdStan v2.23.0 help

  Build CmdStan utilities:
    > make build

    This target will:
    1. Install the Stan compiler bin/stanc from stanc3 binaries.
    2. Build the print utility bin/print (deprecated; will be removed in v3.
    3. Build the stansummary utility bin/stansummary
```

```
  4. Build the diagnose utility bin/diagnose
  5. Build all libraries and object files compile and link an executable S

  Note: to build using multiple cores, use the -j option to make, e.g.,
  for 4 cores:
  > make build -j4


Build a Stan program:

  Given a Stan program at foo/bar.stan, build an executable by typing:
  > make foo/bar

  This target will:
  1. Install the Stan compiler (bin/stanc or bin/stanc2), as needed.
  2. Use the Stan compiler to generate C++ code, foo/bar.hpp.
  3. Compile the C++ code using cc . to generate foo/bar

Additional make options:
  STANCFLAGS: defaults to "". These are extra options passed to bin/stanc
    when generating C++ code. If you want to allow undefined functions in
    Stan program, either add this to make/local or the command line:
        STANCFLAGS = --allow_undefined
  USER_HEADER: when STANCFLAGS has --allow_undefined, this is the name of
    header file that is included. This defaults to "user_header.hpp" in th
    directory of the Stan program.
  STANC2: When set, use bin/stanc2 to generate C++ code.


Example - bernoulli model: examples/bernoulli/bernoulli.stan

  1. Build the model:
     > make examples/bernoulli/bernoulli
  2. Run the model:
     > examples/bernoulli/bernoulli sample data file=examples/bernoulli/be
  3. Look at the samples:
     > bin/stansummary output.csv


Clean CmdStan:
```

```
    Remove the built CmdStan tools:
    > make clean-all

--------------------------------------------------------------------------------
```

1.2. Building CmdStan

Building CmdStan involves preparing a set of executable programs and compiling the command line interface and supporting libraries. The CmdStan tools are:

- `stanc`: the Stan compiler (translates Stan language to C++).
- `stansummary`: a basic posterior analysis tool. The `stansummary` utility processes one or more output files from a run or set of runs of Stan's HMC sampler. For all parameters and quantities of interest in the Stan program, `stansummary` reports a set of statistics including mean, standard deviation, percentiles, effective number of samples, and $\hat{R}$ values.
- `diagnose`: a basic sampler diagnostic tool which checks for indications that the HMC sampler was unable to sample from the full posterior.

CmdStan releases include pre-built binaries of the Stan language compiler https://github.com/stan-dev/stanc3: `bin/linux-stanc`, `bin/mac-stanc` and `bin/windows-stanc`. The CmdStan makefile `build` task copies the appropriate binary to `bin/stanc`. For CmdStan installations which have been cloned of downloaded from the CmdStan GitHub repository, the makefile task will download the appropriate OS-specific binary from the stanc3 repository's nightly release.

Steps to build CmdStan:

- Download the latest release from https://github.com/stan-dev/cmdstan/releases/latest or clone the GitHub repo.
- Open a command-line terminal window and change directories to the CmdStan home directory.
- Run the makefile target `build` which instantiates the CmdStan utilities and compiles all necessary C++ libraries.

```
> cd <cmdstan-home>
> make build
```

If your computer has multiple cores and sufficient ram, the build process can be parallelized by providing the `-j` option. For example, to build on 4 cores, type:

```
> make -j4 build
```

When `make build` is successful, the directory `<cmdstan-home>/bin/` will contain the executables `stanc`, `stansummary`, and `diagnose` (on Windows, corresponding `.exe` files) and the final lines of console output will show the version of CmdStan that has just been built, e.g.:

```
--- CmdStan v2.23.0 built ---
```

Warning: *The Make program may take 10+ minutes and consume 2+ GB of memory to build CmdStan.*

Windows only: CmdStan requires that the Intel TBB library, which is built by the above command, can be found by the Windows system. This requires that the directory `<cmdstan-home>/stan/lib/stan_math/lib/tbb` is part of the `PATH` environment variable. To permanently make this setting for the current user, you may execute:

```
> mingw32-make install-tbb
```

After changing the `PATH` environment variable, you must open an new shell in order to these setting to take effect. (This is not necessary on Mac and Linux systems because they can use the absolute path to the Intel TBB library when linking into Stan programs.)

1.3. Clone the GitHub CmdStan Repository

The CmdStan release tarfile contains all source files an libraries needed to build CmdStan. The CmdStan GitHub repo contains just the `cmdstan` module; the Stan inference engine algorithms and Stan math library functions are specified as submodules and stored in seperate GitHub repositories. The CmdStan Makefile task `stan-update` assembles these submodules in the proper directory structure.

The following sequence of commands will check out the current CmdStan `develop` branch on GitHub and assemble and build the command line interface and supporting libraries:

```
> git clone https://github.com/stan-dev/cmdstan.git --recursive
> cd cmdstan
> make build
```

The resulting set of directories should have the same structure as the release:

- directory `cmdstan/stan` contains the sub-module `stan` (https://github.com/stan-dev/stan)
- directory `cmdstan/stan/lib/stan_math` contains the sub-module `math` (https://github.com/stan-dev/math)

1.4. Trouble-shooting the installation

To check that the CmdStan installation is complete and in working order, run the following series of commands:

```
# compile the example
> make examples/bernoulli/bernoulli

# fit to provided data (results of 10 trials, 2 out of 10 successes)
> ./examples/bernoulli/bernoulli sample data file=examples/bernoulli/bernou

# default output written to file `output.csv`,
# default num_samples is 1000, output file should have approx 1050 lines
> ls -l output.csv

# run the `bin/stansummary utility to summarize parameter estimates
> bin/stansummary output.csv
```

The sample data in file `bernoulli.json.data` specifies 2 out of 10 successes, therefore the range `mean(theta)`±`sd(theta)` should include 0.2.

Updates to CmdStan or changes in compiler options may result in errors when trying to compile a Stan program. In some cases, these can be resolved by removing the existing CmdStan build and recompiling. The Makefile target `clean-all` should be run before rebuilding CmdStan:

```
> make clean-all
> make build
```

2. Example Model and Data

The following is a simple, complete Stan program for a Bernoulli model of binary data.[1] The model assumes the binary observed data `y[1],...,y[N]` are i.i.d. with Bernoulli chance-of-success `theta`.

```
data {
  int<lower=0> N;
  int<lower=0,upper=1> y[N];
}
parameters {
  real<lower=0,upper=1> theta;
}
model {
  theta ~ beta(1,1);  // uniform prior on interval 0,1
  y ~ bernoulli(theta);
}
```

The input data file contains definitions for the two variables `N` and `y` which are specified in the data block of program `bernoulli.stan` (above).

A data set of `N=10` observations is included in the example Bernoulli model directory in both JSON notation and Rdump data format where 8 out of 10 trials had outcome `0` (failure) and 2 trials had outcome `1` (success). In JSON, this data is:

```
{
    "N" : 10,
    "y" : [0,1,0,0,0,0,0,0,0,1]
}
```

[1]The model is available with the CmdStan distribution at the path `<cmdstan-home>/examples/bernoulli/bernoulli.stan`.

3. Compiling a Stan Program

A Stan program must be in a file with extension `.stan`. The CmdStan makefile rules specify all necessary steps to translate files with suffix `.stan` to a CmdStan executable program. This is a two-stage process:

- first the Stan program is translated to C++ by the `stanc` compiler
- then the C++ compiler compiles all C++ sources and links them together with the CmdStan interface program and the Stan and math libraries.

3.1. Invoking the Make Utility

To compile Stan programs, you must invoke the Make program from the `<cmdstan-home>` directory. The Stan program can be in a different directory, but the directory path names cannot contain spaces - this limitation is imposed by Make.

```
> cd <cmdstan_home>
```

In the call to the Make program, the target is name of the CmdStan executable corresponding to the Stan program file. On Mac and Linux, this is the name of the Stan program with the `.stan` omitted. On Windows, replace `.stan` with `.exe`, and make sure that the path is given with slashes and not backslashes. To build the Bernoulli example, on Mac and Linux:

```
> make examples/bernoulli/bernoulli
```

On Windows, the command is the same with the addition of `.exe` at the end of the target (*note the use of forward slashes*):

```
> make examples/bernoulli/bernoulli.exe
```

The generated C++ code (`bernoulli.hpp`), object file (`bernoulli.o`) and the compiled executable will be placed in the same directory as the Stan program.

The compiled executable consists of the Stan model and the CmdStan command line interface which provides inference algorithms to do MCMC sampling, optimization, and variational inference. The following sections provide examples of doing inference using each method on the example model and data file.

3.2. Dependencies

When executing a `make` target, all its dependencies are checked to see if they are up to date, and if they are not, they are rebuilt. If the you call `make` with target `bernoulli` twice, without any edits to `bernoulli.stan` or other changes to the system, the second call to make willis invoked a second time, it will see that it is up to date, and will not recompile the program:

```
> make examples/bernoulli/bernoulli
make: `examples/bernoulli/bernoulli' is up to date.
```

If the file containing the Stan program is updated, the next call to `make` will rebuild the CmdStan executable.

3.3. Compiler Errors

Stan probabalistic programming language is a programming language with a rich syntax, as such, it is often the case that a carefully written program contains errors, often simple syntax errors such as a misspelled variable name or missing semi-colon (`;`) statement termination.

For example, if in the `bernoulli.stan` program, we introduct a typo on line 9 by writing `thata` instead of `theta`, the Make command fails with the following

```
--- Translating Stan model to C++ code ---
bin/stanc  --o=bernoulli.hpp bernoulli.stan

Semantic error in 'bernoulli.stan', line 9, column 2 to column 7:
   -------------------------------------------------
     7:  }
     8:  model {
     9:    thata ~ beta(1,1);  // uniform prior on interval 0,1
           ^
    10:    y ~ bernoulli(theta);
    11:  }
   -------------------------------------------------

Identifier 'thata' not in scope.

make: *** [bernoulli.hpp] Error 1
```

3.4. Troubleshooting

The stanc compiler is also a program, and while it has been extensively tested, it may still contain errors such that the generated C++ code fails to compile. If this happens, report the error, together with the Stan program on either the Stan Forums or on the Stan compiler GitHub issues tracker.

4. MCMC Sampling

4.1. Running the Sampler

To generate a sample from the posterior distribution of the model conditioned on the data, we run the executable program with the argument `sample` or `method=sample` together with the input data. The executable can be run from any directory. Here, we run it in the directory which contains the Stan program and input data, `<cmdstan-home>/examples/bernoulli`:

```
> cd examples/bernoulli
```

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

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

In Windows, the `./` prefix is not needed:

```
> bernoulli.exe sample data file=bernoulli.data.json
```

The output is the same across all supported platforms. First, the configuration of the program is echoed to the standard output:

```
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)
        metric_file =  (Default)
        stepsize = 1 (Default)
        stepsize_jitter = 0 (Default)
id = 0 (Default)
data
  file = bernoulli.data.json
init = 2 (Default)
random
  seed = 3252652196 (Default)
output
  file = output.csv (Default)
  diagnostic_file =  (Default)
  refresh = 100 (Default)
```

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

```
Gradient evaluation took 1.2e-05 seconds
1000 transitions using 10 leapfrog steps per transition would take 0.12 seco
Adjust your expectations accordingly!
```

Next, the sampler reports the iteration number, reporting the percentage complete.

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

Finally, the sampler reports timing information:

```
 Elapsed Time: 0.007 seconds (Warm-up)
               0.017 seconds (Sampling)
               0.024 seconds (Total)
```

4.2. Running Multiple Chains

A Markov chain generates samples from the target distribution only after it has converged to equilibrium. In theory, convergence is only guaranteed asymptotically as the number of draws grows without bound. In practice, diagnostics must be applied to monitor convergence for the finite number of draws actually available. One way to monitor whether a chain has converged to the equilibrium distribution is to compare its behavior to other randomly initialized chains. For robust diagnostics, we recommend running 4 chains.

To run multiple chains given a model and data, either sequentially or in parallel, we use the Unix or DOS shell `for` loop to set up index variables needed to identify each chain and its outputs.

On MacOS or Linux, the for-loop syntax for both the bash and zsh interpreters is:

```
for NAME [in LIST]; do COMMANDS; done
```

The list can be a simple sequence of numbers, or you can use the shell expansion syntax `{1..N}` which expands to the sequence from 1 to N, e.g. `{1..4}` expands to `1 2 3 4`. Note that the expression `{1..N}` cannot contain spaces.

To run 4 chains for the example bernoulli model on MacOS or Linux:

```
> for i in {1..4}
    do
      ./bernoulli sample data file=bernoulli.data.json \
      output file=output_${i}.csv
    done
```

The backslash (\) indicates a line continuation in Unix. The expression `${i}` substitutes in the value of loop index variable `i`. To run chains in parallel, put an ampersand (&) at the end of the nested sampler command:

```
> for i in {1..4}
    do
      ./bernoulli sample data file=bernoulli.data.json \
      output file=output_${i}.csv &
    done
```

This pushes each process into the background which allows the loop to continue without waiting for the current chain to finish.

On Windows, the DOS for-loop syntax is one of:

```
for %i in (SET) do COMMAND COMMAND-ARGUMENTS
for /l %i in (START, STEP, END) do COMMAND COMMAND-ARGUMENTS
```

To run 4 chains in parallel on Windows:

```
>for /l %i in (1, 1, 4) do start /b bernoulli.exe sample ^
                                    data file=bernoulli.data.json my_data ^
                                    output file=output_%i.csv
```

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

4.3. Stan CSV Output File

Each execution of the model results in draws 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 records the version of the underlying Stan library and the configuration as comments (i.e., lines beginning with the pound sign (#)).

```
# stan_version_major = 2
# stan_version_minor = 23
# stan_version_patch = 0
# model = bernoulli_model
# method = sample (Default)
#   sample
#     num_samples = 1000 (Default)
#     num_warmup = 1000 (Default)
...
# output
#   file = output.csv (Default)
#   diagnostic_file =  (Default)
#   refresh = 100 (Default)
```

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

```
lp__,accept_stat__,stepsize__,treedepth__,n_leapfrog__,divergent__,energy__,
```

The first output columns report the HMC sampler information:

- `lp__` - the total log probability density (up to an additive constant) at each sample
- `accept_stat__` - the average Metropolis acceptance probability over each simulated Hamiltonian trajectory
- `stepsize__` - integrator step size
- `treedepth__` - depth of tree used by NUTS (NUTS sampler)
- `n_leapfrog__` - number of leapfrog calculations (NUTS sampler)
- `divergent__` - has value `1` if trajectory diverged, otherwise `0`. (NUTS sampler)
- `energy__` - value of the Hamiltonian
- `int_time__` - total integration time (HMC sampler)

The remaining columns correspond to model parameters. For the Bernoulli model, it is just the final column, `theta`.

The header line is written to the output file before warmup begins. If option `save_warmup` is set to `1`, the warmup draws are output directly after the header.

The total number of warmup draws saved is `num_warmup` divided by `thin`, rounded up (i.e., `ceiling`).

Following the warmup draws (if any), are comments which record the results of adaptation: the stepsize, and inverse mass metric used during sampling:

```
# Adaptation terminated
# Step size = 0.884484
# Diagonal elements of inverse mass matrix:
# 0.535006
```

The default sampler is NUTS with an adapted step size and a diagonal inverse mass matrix. For this example, the step size is 0.884484, and the inverse mass contains the single entry 0.535006 corresponding to the parameter `theta`.

Draws from the posterior distribution are printed out next, each line containing a single draw with the columns corresponding to the header.

```
-6.84097,0.974135,0.884484,1,3,0,6.89299,0.198853
-6.91767,0.985167,0.884484,1,1,0,6.92236,0.182295
-7.04879,0.976609,0.884484,1,1,0,7.05641,0.162299
-6.88712,1,0.884484,1,1,0,7.02101,0.188229
-7.22917,0.899446,0.884484,1,3,0,7.73663,0.383596
...
```

The output ends with timing details:

```
#  Elapsed Time: 0.007 seconds (Warm-up)
#                0.017 seconds (Sampling)
#                0.024 seconds (Total)
```

4.4. Summarizing Sampler Output(s) with `stansummary`

The `stansummary` utility processes one or more output files from a run or set of runs of Stan's HMC sampler given a model and data. For all columns in the Stan csv output file `stansummary` reports a set of statistics including mean, standard deviation, percentiles, effective number of samples, and $\hat{R}$ values.

To run `stansummary` on the output files generated by the for loop above, by the above run of the `bernoulli` model on Mac or Linux:

```
<cmdstan-home>/bin/stansummary output_*.csv
```

On Windows, use backslashes to call the `stansummary.exe`.

```
<cmdstan-home>\bin\stansummary.exe output_*.csv
```

The stansummary output consists of one row of statistics per column in the Stan csv output file. Therefore, the first rows in the stansummary report statistics over the sampler state. The final row of output summarizes the estimates of the model variable `theta`:

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

Warmup took (0.0070, 0.0070, 0.0070, 0.0070) seconds, 0.028 seconds total
Sampling took (0.020, 0.017, 0.021, 0.019) seconds, 0.077 seconds total

                Mean     MCSE  StdDev     5%   50%   95%    N_Eff  N_Eff/s
lp__            -7.3  1.8e-02    0.75   -8.8  -7.0  -6.8  1.8e+03  2.4e+04
accept_stat__   0.89  2.7e-03    0.17   0.52  0.96   1.0  3.9e+03  5.1e+04
stepsize__       1.1  7.5e-02    0.11   0.93   1.2   1.2  2.0e+00  2.6e+01
treedepth__      1.4  8.1e-03    0.49    1.0   1.0   2.0  3.6e+03  4.7e+04
n_leapfrog__     2.3  1.7e-02    0.98    1.0   3.0   3.0  3.3e+03  4.3e+04
divergent__     0.00      nan    0.00   0.00  0.00  0.00      nan      nan
energy__         7.8  2.6e-02     1.0    6.8   7.5   9.9  1.7e+03  2.2e+04
theta           0.25  2.9e-03    0.12  0.079  0.23  0.46  1.7e+03  2.1e+04

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 this example, we conditioned the model on a dataset consisting of the outcomes of 10 bernoulli trials, where only 2 trials reported success. The 5%, 50%, and 95% percentile values for `theta` reflect the uncertainty in our estimate, due to the small amount of data, given the prior of `beta(1, 1)`

5. Optimization

The CmdStan executable can run Stan's optimization algorithms for penalized maximum likelihood estimation which provide a deterministic method to find the posterior mode. 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.

The executable does not need to be recompiled in order to switch from sampling to optimization, and the data input format is the same. The following is a minimal call to Stan's optimizer using defaults for everything but the location of the data file.

```
> ./bernoulli optimize data file=bernoulli.data.json
```

Executing this command prints both output to the console and to a csv file.

The first part of the console output reports on the configuration used. The above command uses all default configurations, therefore the optimizer used is the L-BFGS optimizer and its default initial stepsize and tolerances for monitoring convergence:

```
 ./bernoulli optimize data file=bernoulli.data.json
method = optimize
  optimize
    algorithm = lbfgs (Default)
      lbfgs
        init_alpha = 0.001 (Default)
        tol_obj = 9.9999999999999998e-13 (Default)
        tol_rel_obj = 10000 (Default)
        tol_grad = 1e-08 (Default)
        tol_rel_grad = 10000000 (Default)
        tol_param = 1e-08 (Default)
        history_size = 5 (Default)
    iter = 2000 (Default)
    save_iterations = 0 (Default)
id = 0 (Default)
data
  file = bernoulli.data.json
init = 2 (Default)
random
  seed = 3316231346 (Default)
```

```
output
  file = output.csv (Default)
  diagnostic_file =  (Default)
  refresh = 100 (Default)
```

The second part of the output indicates how well the algorithm fared, here converging and terminating normally. The numbers reported indicate that it took 5 iterations and 8 gradient evaluations. This is, not surprisingly, far fewer iterations than required for sampling; even fewer iterations would be used with less stringent user-specified convergence tolerances. The `alpha` value is for step size used. In the final state the change in parameters was roughly 0.0002 and the length of the gradient roughly 9e-8.

```
Initial log joint probability = -5.26908
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0
       5      -5.00402   0.000172451   9.39034e-08           1           1
Optimization terminated normally:
  Convergence detected: relative gradient magnitude is below tolerance
```

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 as comment lines:

```
# stan_version_major = 2
# stan_version_minor = 23
# stan_version_patch = 0
# model = bernoulli_model
# method = optimize
#   optimize
#     algorithm = lbfgs (Default)
...
```

Following the config information, are two lines of output: the CSV headers and the recorded values:

```
lp__,theta
-5.00402,0.2
```

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.[1]

[1] 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.

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.

6. Variational Inference

CmdStan can approximate the posterior distribution using variational inference. The approximation is a Gaussian in the unconstrained variable space. Stan implements two variational algorithms. The `algorithm=meanfield` option uses a fully factorized Gaussian for the approximation. The `algorithm=fullrank` option uses a Gaussian with a full-rank covariance matrix for the approximation.

The executable does not need to be recompiled in order to switch to variational inference, and the data input format is the same. The following is a minimal call to Stan's variational inference algorithm using defaults for everything but the location of the data file.

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

Executing this command prints both output to the console and to a csv file.

The first part of the console output reports on the configuration used. Here it indicates the default mean-field setting of the variational inference algorithm. It also indicates the default parameter sizes and tolerances for monitoring the algorithm's convergence.

```
method = variational
  variational
    algorithm = meanfield (Default)
      meanfield
    iter = 10000 (Default)
    grad_samples = 1 (Default)
    elbo_samples = 100 (Default)
    eta = 1 (Default)
    adapt
      engaged = 1 (Default)
      iter = 50 (Default)
    tol_rel_obj = 0.01 (Default)
    eval_elbo = 100 (Default)
    output_samples = 1000 (Default)
id = 0 (Default)
data
  file = bernoulli.data.json
init = 2 (Default)
```

```
random
  seed = 3323783840 (Default)
output
  file = output.csv (Default)
  diagnostic_file =  (Default)
  refresh = 100 (Default)
```

After the configuration has been displayed, informational and timing messages are output:

```
------------------------------------------------------------
EXPERIMENTAL ALGORITHM:
  This procedure has not been thoroughly tested and may be unstable
  or buggy. The interface is subject to change.
------------------------------------------------------------


Gradient evaluation took 2.1e-05 seconds
1000 transitions using 10 leapfrog steps per transition would take 0.21 seco
Adjust your expectations accordingly!
```

The rest of the output describes the progression of the algorithm. An adaptation phase finds a good value for the step size scaling parameter `eta`. The evidence lower bound (ELBO) is the variational objective function and is evaluated based on a Monte Carlo estimate. The variational inference algorithm in Stan is stochastic, which makes it challenging to assess convergence. That is, while the algorithm appears to have converged in ~ 250 iterations, the algorithm runs for another few thousand iterations until mean change in ELBO drops below the default tolerance of 0.01.

```
Begin eta adaptation.
Iteration:   1 / 250 [  0%]  (Adaptation)
Iteration:  50 / 250 [ 20%]  (Adaptation)
Iteration: 100 / 250 [ 40%]  (Adaptation)
Iteration: 150 / 250 [ 60%]  (Adaptation)
Iteration: 200 / 250 [ 80%]  (Adaptation)
Success! Found best value [eta = 1] earlier than expected.

Begin stochastic gradient ascent.
  iter             ELBO   delta_ELBO_mean   delta_ELBO_med   notes
   100           -6.131             1.000            1.000
   200           -6.458             0.525            1.000
   300           -6.300             0.359            0.051
   400           -6.137             0.276            0.051
```

```
   500            -6.243            0.224            0.027
   600            -6.305            0.188            0.027
   700            -6.289            0.162            0.025
   800            -6.402            0.144            0.025
   900            -6.103            0.133            0.025
  1000            -6.314            0.123            0.027
  1100            -6.348            0.024            0.025
  1200            -6.244            0.020            0.018
  1300            -6.293            0.019            0.017
  1400            -6.250            0.017            0.017
  1500            -6.241            0.015            0.010   MEDIAN ELBO CON
```

```
Drawing a sample of size 1000 from the approximate posterior...
COMPLETED.
```

The output from variational 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 as CSV comments:

```
# stan_version_major = 2
# stan_version_minor = 23
# stan_version_patch = 0
# model = bernoulli_model
# method = variational
#   variational
#     algorithm = meanfield (Default)
#       meanfield
#     iter = 10000 (Default)
#     grad_samples = 1 (Default)
#     elbo_samples = 100 (Default)
#     eta = 1 (Default)
#     adapt
#       engaged = 1 (Default)
#       iter = 50 (Default)
#     tol_rel_obj = 0.01 (Default)
#     eval_elbo = 100 (Default)
#     output_samples = 1000 (Default)
...
```

Next is the column header line, followed more CSV comments reporting the adapted value for the stepsize, followed by the values. The first line is special: it is the mean of the variational approximation. The rest of the output contains `output_samples` number of samples drawn from the variational approximation.

```
lp__,log_p__,log_g__,theta
# Stepsize adaptation complete.
# eta = 1
0,0,0,0.236261
0,-6.82318,-0.0929121,0.300415
0,-6.89701,-0.158687,0.321982
0,-6.99391,-0.23916,0.343643
0,-7.35801,-0.51787,0.401554
0,-7.4668,-0.539473,0.123081
...
```

The header indicates the unnormalized log probability with `lp__`. This is a legacy feature that we do not use for variational inference. The ELBO is not stored unless a diagnostic option is given.

For further details, see Kucukelbir, Alp, Rajesh Ranganath, Andrew Gelman, and David M. Blei. 2015. *Automatic Variational Inference in Stan.* arXiv 1506.03431. http://arxiv.org/abs/1506.03431.

7. Generating Quantities of Interest from a Fitted Model

The generated quantities block computes *quantities of interest* (QOIs) based on the data, transformed data, parameters, and transformed parameters. It can be used to:

- generate simulated data for model testing by forward sampling
- generate predictions for new data
- calculate posterior event probabilities, including multiple comparisons, sign tests, etc.
- calculating posterior expectations
- transform parameters for reporting
- apply full Bayesian decision theory
- calculate log likelihoods, deviances, etc. for model comparison

The `generate_quantities` method allows you to generate additional quantities of interest from a fitted model without re-running the sampler. Instead, you write a modified version of the original Stan program and add a generated quantities block or modify the existing one which specifies how to compute the new quantities of interest. Running the `generate_quantities` method on the new program together with sampler outputs from the fitted model runs the generated quantities block of the new program using the estimated parameter values from the existing sample.

To illustrate how this works, we add poterior predictive checks to the example model `bernoulli.stan`. We create a new model, `bernoulli_yrep.stan` which contains the following generated quantities block:

```
generated quantities {
  int y_sim[N];
  real<lower=0,upper=1> theta_rep;
  for (n in 1:N)
    y_sim[n] = bernoulli_rng(theta);
  theta_rep = sum(y) / N;
}
```

We compile this model, and then run it with the fit from a previous run of the original model and the same input data file:

```
> ./bernoulli_yrep generate_quantities ....
```

CmdStan Reference

This section provides a complete reference for all CmdStan methods:

- sample
- optimize
- variational
- generate_quantities
- diagnose
- help

8. Command-Line Interface Overview

A CmdStan executable is built from the Stan model concept and the CmdStan command line parser. The command line argument syntax consists of sets of keywords and keyword-value pairs. Arguments are grouped by the following keywords:

- `method` - specifies the kind of inference done on the model. Each kind of inference requires further configuration via sub-arguments. The `method` argument is required. It can be specified overtly as the a keyword-value pair `method=<inference>` or implicitly as one of the following:
  - `sample` - obtain a sample from the posterior using HMC
  - `optimize` - penalized maximum likelihood estimation
  - `variational` - automatic variational inference
  - `generate_quantities` - run model's `generated quantities` block on existing sample to obtain new quantities of interest.
  - `diagnose` - compute and compare sampler gradient calculations to finite differences.
- `data` - specifies the input data file, if any.
- `output` - specifies program outputs, both disk files and terminal window outputs.
- `init` - specifies initial values for the model parameters, if any.
- `random` - specifies the seed for the psuedo-random number.

The remainder of this chapter covers the general configuration options used for all processing. The following chapters cover the per-inference configuration options.

8.1. Input Data Argument

The values for all variables declared in the data block of the model are read in from an input data file in either JSON or Rdump format. The syntax for the input data argument is:

```
data file=<filepath>
```

The keyword `data` must be followed directly by the keyword-value pair `file=<filepath>`. If the model doesn't declare any data variables, this argument is ignored.

The input data file must contain definitions for all data variables declared in the data block. If one or more data block variables are missing from the input data file, the program will print and error message to the terminal. For example, the model `bernoulli.stan` defines two data variables `N` and `y`. If the input data file doesn't include both variables, or if the data variable doesn't match the declared type and dimensions, the program will exit with an error message at the point where it first encounters missing data.

For example if the input data file doesn't include the definition for variable `y`, the executable exits with the following message:

```
Exception: variable does not exist; processing stage=data initialization; va
```

8.2. Output Control Arguments

The `output` keyword is used to specify non-default options for output files and messages written to the terminal window. The `output` keyword takes several keyword-value pair sub-arguments.

The keyword value pair file=<filepath> specifies the location of the Stan csv output file. If unspecified, the output file is written to a file named `output.csv` in the current working directory.

The keyword value pair diagnostic_file=<filepath> specifies the location of the auxiliary output file. By default, no auxiliary output file is produced.

The keyword value pair refresh=<int> specifies the number of iterations between progress messages written to the terminal window. The default value is 100 iterations.

8.3. Initialize Model Parameters Argument

Initialization is only applied to parameters defined in the parameters block. By default, all parameters are initialized to random draws from a uniform distribution over the range $[-2, 2]$. These values are on the unconstrained scale, so must be inverse transformed back to satisfy the constraints declared for parameters. Because zero is chosen to be a reasonable default initial value for most parameters, the interval around zero provides a fairly diffuse starting point. For instance, unconstrained variables are initialized randomly in $(-2, 2)$, variables constrained to be positive are initialized roughly in $(0.14, 7.4)$, variables constrained to fall between 0 and 1 are initialized with values roughly in $(0.12, 0.88)$.

The initialization argument is specified as keyword-value pair with keyword `init`. The value can be one of the following:

- positive real number x. All parameters will be initialized to random draws from a uniform distribution over the range $[-x, x]$.

- 0 - All parameters will be initialized to zero values on the unconstrained scale. The transforms are arranged in such a way that zero initialization provides reasonable variable initializations: 0 for unconstrained parameters; 1 for parameters constrained to be positive; 0.5 for variables to constrained to lie between 0 and 1; a symmetric (uniform) vector for simplexes; unit matrices for both correlation and covariance matrices; and so on.
- filepath - A data file in JSON or Rdump format containing initial parameters values for some or all of the model parameters. User specified initial values must satisfy the constraints declared in the model (i.e., they are on the constrained scale). Parameters which aren't explicitly initializied will be initialized randomly over the range $[-2, 2]$.

8.4. Random Number Generator Arguments

The random-number generator's behavior is determined by the unsigned seed (positive integer) it is started with. If a seed is not specified, or a seed of 0 or less is specified, the system time is used to generate a seed. The seed is recorded and included with Stan's output regardless of whether it was specified or generated randomly from the system time.

The syntax for the random seed argument is:

```
random seed=<int>
```

The keyword `random` must be followed directly by the keyword-value pair `seed=<int>`.

8.5. Chain Identifier Argument: `id`

The chain identifier argument is used in conjunction with the `random seed` argument when running multiple Markov chains for sampling. The chain identifier is used to advance the random number generator a very large number of random variates so that two chains with the same seed and different identifiers draw from non-overlapping subsequences of the random-number sequence determined by the seed. Together, the seed and chain identifier determine the behavior of the random number generator.

The syntax for the random seed argument is:

```
id=<int>
```

The default value is 0.

When running a set of chains from the command line with a specified seed, this argument should be set to the chain index. E.g., when running 4 chains, the value should be 1,..,4, successively. When running multiple chains from a single command, Stan's interfaces manage the chain identifier arguments automatically.

For complete reproducibility, every aspect of the environment needs to be locked down from the OS and version to the C++ compiler and version to the version of Stan and all dependent libraries. See the Stan Reference Manual Reproducibility chapter for further details.

8.6. Command Line Help

CmdStan provides a `help` and `help-all` mechanism that displays either the available top-level or keyword-specific key-value argument pairs. To display top-level help, call the CmdStan executable with keyword `help`:

```
> ./bernoulli help
Usage: ./bernoulli <arg1> <subarg1_1> ... <subarg1_m> ... <arg_n> <subarg_n_

Begin by selecting amongst the following inference methods and diagnostics,
  sample      Bayesian inference with Markov Chain Monte Carlo
  optimize    Point estimation
  variational  Variational inference
  diagnose    Model diagnostics
  generate_quantities  Generate quantities of interest

Or see help information with
  help        Prints help
  help-all    Prints entire argument tree

Additional configuration available by specifying
  id          Unique process identifier
  data        Input data options
  init        Initialization method: "x" initializes randomly between [-x, x
  random      Random number configuration
  output      File output options

See ./bernoulli <arg1> [ help | help-all ] for details on individual argumen
```

9. MCMC Sampling using Hamiltonian Monte Carlo

The `sample` method provides Bayesian inference over the model conditioned on data using Hamiltonian Monte Carlo (HMC) sampling. By default, the inference engine used is the No-U-Turn sampler (NUTS), an adaptive form of Hamiltonian Monte Carlo sampling. For details on HMC and NUTS, see the Stan Reference Manual chapter on MCMC Sampling.

The full set of configuration options available for the `sample` method is reported at the beginning of the sampler output file as csv comments. When the example model `bernoulli.stan` is run via the command line with all default arguments, the resulting Stan csv file header comments show the complete set of default configuration options:

```
# 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 = 0.05 (Default)
#       delta = 0.8 (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)
#         metric_file =  (Default)
```

```
#         stepsize = 1 (Default)
#         stepsize_jitter = 0 (Default)
```

9.1. Iterations

At every sampler iteration, the sampler returns a set of estimates for all parameters and quantities of interest in the model. During warmup, the NUTS algorithm adjusts the HMC algorithm parameters `metric` and `stepsize` in order to efficiently sample from *typical set*, the neighborhood substantial posterior probability mass through which the Markov chain will travel in equilibrium. After warmup, the fixed metric and stepsize are used to produce a set of draws.

The following keyword-value arguments control the total number of iterations:

- `num_samples`
- `num_warmup`
- `save_warmup`
- `thin`

The values for arguments `num_samples` and `num_warmup` must be a non-negative integer. The default value for both is 1000.

For well-specified models and data, the sampler may converge faster and this many warmup iterations may be overkill. Conversely, complex models which have difficult posterior geometries may require more warmup iterations in order to arrive at good values for the step size and metric.

The number of sampling iterations to runs depends on the effective sample size (EFF) reported for each parameter and the desired precision of your estimates. An EFF of at least 100 is required to make a viable estimate. The precision of your estimate is $\sqrt{N}$; therefore every additional decimal place of accuracy increases this by a factor of 10.

Argument `save_warmup` takes the value of either 0 or 1, which correspond to `False` and `True` respectively. The default value is 0, i.e., warmup draws are not saved to the output file. When the value is 1, the warmup draws are written to the csv output file directly after the csv header line.

Argument `thin` controls the number of draws from the posterior written to the output file. The value of argument `thin` must be a positive integer. When thin is set to value N, every N^{th} iteration is written to the output file.

Should the value of `thin` exceed the specified number of iterations, the first iteration is saved to the output. This is because the iteration counter starts from zero and whenever the counter modulo the value of `thin` equals zero, the iteration is saved to the output file. Since zero modulo any positive integer is zero, the first iteration is always saved. When `num_sampling=M` and `thin=N`, the number of iterations written to the output csv file will be `ceiling(M/N)`. If `save_warmup=1`, thinning is applied to the warmup iterations as well.

9.2. Adaptation

The `adapt` keyword is used to specify non-default options for the sampler adaptation schedule and settings.

Adaptation can be turned off by setting sub-argument `engaged` to value 0. If `engaged=0`, no adaptation will be done, and all other adaptation sub-arguments will be ignored. Since the default argument is `engaged=1`, this keyword-value pair can be omitted from the command.

There are two sets of adaptation sub-arguments: step size optimization parameters and the warmup schedule. These are described in detail in the Reference Manual section Automatic Parameter Tuning.

Step size optimization configuration

The following keyword-value arguments control the settings used to optimize the step size:

- `delta` - The target Metropolis acceptance rate. The default value is 0.8. Its value must be strictly between 0 and 1. Increasing the default value forces the algorithm to use smaller step sizes. This can improve sampling efficiency (effective sample size per iteration) at the cost of increased iteration times. Raising the value of `delta` will also allow some models that would otherwise get stuck to overcome their blockages.

- `gamma` - Adaptation regularization scale. Must be a positive real number, default value is 0.05. This is a parameter of the Nesterov dual-averaging algorithm. We recommend always using the default value.

- `gamma` - Adaptation relaxation exponent. Must be a positive real number, default value is 0.75. This is a parameter of the Nesterov dual-averaging algorithm. We recommend always using the default value.

- `t_0` - Adaptation iteration offset. Must be a positive real number, default value is 10. This is a parameter of the Nesterov dual-averaging algorithm. We recommend always using the default value.

Warmup schedule configuration

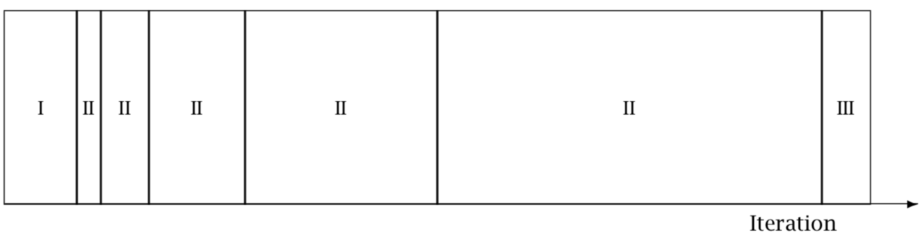
When adaptation is engaged, the warmup schedule is specified by sub-arguments, all of which take positive integers as values:

- `init_buffer` - The number of iterations spent tuning the step size at the outset of adaptation.
- `window` - The initial number of iterations devoted to tune the metric, will be doubled successively.
- `term_buffer` - The number of iterations used to re-tune the step size once the metric has been tuned.

The specified values may be modified slightly in order to ensure alignment between the warmup schedule and total number of warmup iterations.

The following figure is taken from the Stan Reference Manual, where label "I" correspond to `init_buffer`, the initial "II" corresponds to `window`, and the final "III" corresponds to `term_buffer`:

Warmup Epochs Figure. *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.*



9.3. Algorithm

The `algorithm` keyword-value pair specifies the algorithm used to generate the sample. There are two possible values: `hmc`, which generates from an HMC-driven Markov chain; and `fixed_param` which generates a new sample without changing the state of the Markov chain. The default argument is `algorithm=hmc`.

Samples from a set of fixed parameters

If a model doesn't specify any parameters, then argument `algorithm=fixed_param` is mandatory.

The fixed parameter sampler generates a new sample without changing the current state of the Markov chain. This can be used to write models which generate pseudo-data via calls to RNG functions in the transformed data and generated quantities blocks.

HMC samplers

All HMC algorithms have three parameters:

- step size
- metric
- integration time - the number of steps taken along the Hamiltonian trajectory

See the Stan Reference Manual section on HMC algorithm parameters for further details.

Step size

The HMC algorithm simulates the evolution of a Hamiltonian system. The step size parameter controls the resolution of the sampler. Low step sizes can get HMC samplers unstuck that would otherwise get stuck with higher step sizes.

The following keyword-value arguments control the step size:

- `stepsize` - How far to move each time the Hamiltonian system evolves forward. Must be a positive real number, default value is 1.

- `stepsize_jitter` - Allows step size to be "jittered" randomly during sampling to avoid any poor interactions with a fixed step size and regions of high curvature. Must be a real value between 0 and 1. The default value is 0. Setting `stepsize_jitter` to 1 causes step sizes to be selected in the range of 0 to twice the adapted step size. Jittering below the adapted value will increase the number of steps required and will slow down sampling, while jittering above the adapted value can cause premature rejection due to simulation error in the Hamiltonian dynamics calculation. We strongly recommend always using the default value.

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 (2017).

The `metric` argument specifies the choice of Euclidean HMC implementations:

- `metric=unit` specifies unit metric (diagonal matrix of ones).
- `metric=diag_e` specifies a diagonal metric (diagonal matrix with positive diagonal entries). This is the default value.

- metric=dense_e‘ specifies a dense metric (a dense, symmetric positive definite matrix).

By default, the metric is estimated during warmup. However, when `metric=diag_e` or `metric=dense_e`, an initial guess for the metric can be specified with the `metric_file` argument whose value is the filepath to a JSON or Rdump file which contains a single variable `inv_metric`. For a `diag_e` metric the `inv_metric` value must be a vector of positive values, one for each parameter in the system. For a `dense_e` metric, `inv_metric` value must be a positive-definite square matrix with number of rows and columns equal to the number of parameters in the model.

The `metric_file` option can be used with and without adaptation enabled. If adaptation is enabled, the provided metric will be used as the initial guess in the adaptation process. If the initial guess is good, then adaptation should not change it much. If the metric is no good, then the adaptation will override the initial guess.

If adaptation is disabled, both the `metric_file` and `stepsize` arguments should be specified.

Integration Time

The total integration time is determined by the argument `engine` which take possible values:

- `nuts` - the No-U-Turn Sampler which dynamically determines the optimal integration time.
- `static` - an HMC sampler which uses a user-specified integration time.

The default argument is `engine=nuts`.

The NUTS sampler generates a proposal by starting at an initial position determined by the parameters drawn in the last iteration. It then evolves the initial system both forwards and backwards in time to form a balanced binary tree. The algorithm is iterative; at each iteration the tree depth is increased by one, doubling the number of leapfrog steps thus effectively doubling the computation time. The algorithm terminates in one of two ways: either the NUTS criterion (i.e., a U-turn in Euclidean space on a subtree) is satisfied for a new subtree or the completed tree; or the depth of the completed tree hits the maximum depth allowed.

When `engine=nuts`, the subargument `max_depth` can be used to control the depth of the tree. The default argument is `max_depth=10`. In the case where a model has a difficult posterior from which to sample, `max_depth` should be increased to ensure that that the NUTS tree can grow as large as necessary.

When the argument `engine=static` is specified, the user must specify the integration time via keyword `int_time` which takes as a value a positive number. The default value is 2π.

9.4. Sampler Diagnostic File

The `output` keyword sub-argument `diagnostic_file=<filepath>` specifies the location of the auxiliary output file which contains sampler information for each draw, including the gradients on the unconstrained scale and log probabilities. By default, no auxiliary output file is produced.

9.5. Examples

The Quickstart Guide MCMC Sampling chapter section on multiple chains showed how to run multiple chains given a model and data, using the minimal required command line options: the method, the name of the data file, and a chain-specific name for the output file.

To run 4 chains in parallel on Mac OS and Linux, the syntax in both bash and zsh is the same:

```
> for i in {1..4}
    do
      ./bernoulli sample data file=my_model.data.json \
                  output file=output_${i}.csv &
    done
```

The backslash (\) indicates a line continuation in Unix. The expression `${i}` substitutes in the value of loop index variable `i`. The ampersand (&) pushes each process into the background which allows the loop to continue without waiting for the current chain to finish.

On Windows the corresponding loop is:

```
>for /l %i in (1, 1, 4) do start /b bernoulli.exe sample ^
                                    data file=my_model.data.json my_data ^
                                    output file=output_%i.csv
```

The caret (^) indicates a line continuation in DOS. The expression `%i` is the loop index.

In the following examples, we focus on just the nested sampler command for Unix.

Running multiple chains with a specified RNG seed

For reproducibility, we specify the same RNG seed across all chains and use the chain id argument to specify the RNG offset.

The RNG seed is specified by `random seed=<int>` and the offset is specified by id=<loop index>, so the call to the sampler is:

```
./my_model sample data file=my_model.data.json \
           output file=output_${i}.csv \
           random seed=12345 id=${i}
```

Changing the default warmup and sampling iterations

The warmup and sampling iteration keyword-value arguments must follow the sample keyword. The call to the sampler which overrides the default warmup and sampling iterations is:

```
./my_model sample num_warmup=500 num_sampling=500 \
           data file=my_model.data.json \
           output file=output_${i}.csv
```

Saving warmup draws

To save warmup draws as part of the Stan csv output file, use the keyword-value argument save_warmup=1. This must be grouped with the other sample keyword sub-arguments.

```
./my_model sample num_warmup=500 num_sampling=500 save_warmup=1 \
           data file=my_model.data.json \
           output file=output_${i}.csv
```

Initializing parameters

By default, all parameters are initialized to random draws from a uniform distribution over the range $[-2, 2]$. To initialize some or all parameters to good starting points on the constrained scale from a data file in JSON or Rdump format, use the keyword-value argument init=<filepath>:

```
./my_model sample init=my_param_inits.json data file=my_model.data.json \
           output file=output_${i}.csv
```

Specifying the metric and stepsize

An initial guess for the metric can be specified with the metric_file argument whose value is the filepath to a JSON or Rdump file which contains a single variable inv_metric. The metric_file option can be used with and without adaptation enabled.

By default, the metric is estimated during warmup adaptation. If the initial guess is good, then adaptation should not change it much. If the metric is no good, then the adaptation will override the initial guess. For example, the JSON file bernoulli.diag_e.json, contents

```
{ "inv_metric" : [0.296291] }
```

can be used as the initial metric as follows:

```
../my_model sample algorithm=hmc metric_file=bernoulli.diag_e.json \
            data file=my_model.data.json \
            output file=output_${i}.csv
```

If adaptation is disabled, both the `metric_file` and `stepsize` arguments should be specified.

```
../my_model sample adapt engaged=0 \
            algorithm=hmc stepsize=0.9 \
            metric_file=bernoulli.diag_e.json \
            data file=my_model.data.json \
            output file=output_${i}.csv
```

The resulting output csv file will contain the following set of comment lines:

```
# Adaptation terminated
# Step size = 0.9
# Diagonal elements of inverse mass matrix:
# 0.296291
```

Changing the NUTS-HMC adaptation parameters

The Stan User's Guide section on model conditioning and curvature provides a discussion of adaptation and stepsize issues. The Stan Reference Manual section on HMC algorithm parameters explains the NUTS-HMC adaptation schedule and the tuning parameters for setting the step size. The keyword-value arguments for these settings are grouped together under the `adapt` keyword which itself is a sub-argument of the `sample` keyword.

Models with difficult posterior geometries may required increasing the `delta` argument closer to 1.

```
./my_model sample adapt delta=0.95 \
            data file=my_model.data.json \
            output file=output_${i}.csv
```

To skip adaptation altogether, use the keyword-value argument `engaged=0`. Disabling adaptation disables both metric and stepsize adaptation, so a stepsize should be provided along with a metric to enable efficient sampling.

```
../my_model sample adapt engaged=0 \
            algorithm=hmc stepsize=0.9 \
            metric_file=bernoulli.diag_e.json \
            data file=my_model.data.json \
```

```
            output file=output_${i}.csv
```

Even with adaptation disabled, it is still advisable to run warmup iterations in order to allow the initial parameter values to be adjusted to estimates which fall within the typical set.

To skip warmup altogether requires specifying both `num_warmup=0` and `adapt engaged=0`.

```
../my_model sample num_warmup=0 adapt engaged=0 \
            algorithm=hmc stepsize=0.9 \
            metric_file=bernoulli.diag_e.json \
            data file=my_model.data.json \
            output file=output_${i}.csv
```

Increasing the tree-depth

Models with difficult posterior geometries may required increasing the `max_depth` argument from its default value 10. This requires specifying a series of keyword-argument pairs:

```
./my_model sample adapt delta=0.95 \
            algorithm=hmc engine=nuts max_depth=15 \
            data file=my_model.data.json \
            output file=output_${i}.csv
```

Capturing Hamiltonian diagnostics and gradients

The `output` keyword sub-argument `diagnostic_file=<filepath>` write the sampler parameters and gradients of all model parameters for each draw to a csv file:

```
./my_model sample data file=my_model.data.json \
            output file=output_${i}.csv \
            diagnostic_file=diagnostics_${i}.csv
```

Suppressing progress updates to the console

The `output` keyword sub-argument `refresh=<int>` specifies the number of iterations between progress messages written to the terminal window. The default value is 100 iterations. The progress updates look like:

```
Iteration:    1 / 2000 [  0%]  (Warmup)
Iteration:  100 / 2000 [  5%]  (Warmup)
Iteration:  200 / 2000 [ 10%]  (Warmup)
Iteration:  300 / 2000 [ 15%]  (Warmup)
```

For simple models which fit quickly, such updates can be annoying; to suppress them altogether, set `refresh=0`. This only turns off the `Iteration:` messages; the configuration and timing information are still written to the terminal.

```
./my_model sample data file=my_model.data.json \
            output file=output_${i}.csv \
            refresh=0
```

For complicated models which take a long time to fit, setting the refresh rate to a low number, e.g. 10 or even 1, provides a way to more closely monitor the sampler.

Everything Example

The CmdStan argument parser requires keeping sampler config sub-arguments together; interleaving sampler config with the inputs, outputs, inits, RNG seed and chain id config results in an error message such as the following:

```
./bernoulli sample data file=bernoulli.data.json adapt delta=0.95
adapt is either mistyped or misplaced.
Perhaps you meant one of the following valid configurations?
  method=sample sample adapt
  method=variational variational adapt
Failed to parse arguments, terminating Stan
```

The following example provides a template for a call to the sampler which specifies input data, initial parameters, initial step-size and metric, adaptation, output, and RNG initialization.

```
./my_model sample num_warmup=2000 \
           init=my_param_inits.json \
           adapt delta=0.95 init_buffer=100 \
           window=50 term_buffer=100 \
           algorithm=hmc engine=nuts max_depth=15 \
           metric=dense_e metric_file=my_metric.json \
           stepsize=0.6555 \
           data file=my_model.data.json \
           output file=output_${i}.csv refresh=10 \
           random seed=12345 id=${i}
```

The keywords `sample`, `data`, `output`, and `random` are the top-level argument groups. Within the `sample` config arguments, the keyword `adapt` groups the adaptation algorithm parameters and the keyword-value `algorithm=hmc` groups the NUTS-HMC parameters.

The top-level groups can be freely ordered with respect to one another. The following is also a valid command:

```
./my_model random seed=12345 id=${i} \
           data file=my_model.data.json \
           output file=output_${i}.csv refresh=10 \
           sample num_warmup=2000 \
           init=my_param_inits.json \
           algorithm=hmc engine=nuts max_depth=15 \
           metric=dense_e metric_file=my_metric.json \
           stepsize=0.6555 \
           adapt delta=0.95 init_buffer=100 \
           window=50 term_buffer=100
```

10. Maximum Likelihood Estimation

The `optimize` method finds the mode of the posterior distribution, assuming that 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.

The full set of configuration options available for the `optimize` method is reported at the beginning of the sampler output file as csv comments. When the example model `bernoulli.stan` is run with `method=optimize` via the command line with all default arguments, the resulting Stan csv file header comments show the complete set of default configuration options:

```
# model = bernoulli_model
# method = optimize
#   optimize
#     algorithm = lbfgs (Default)
#       lbfgs
#         init_alpha = 0.001 (Default)
#         tol_obj = 9.9999999999999998e-13 (Default)
#         tol_rel_obj = 10000 (Default)
#         tol_grad = 1e-08 (Default)
#         tol_rel_grad = 10000000 (Default)
#         tol_param = 1e-08 (Default)
#         history_size = 5 (Default)
#     iter = 2000 (Default)
#     save_iterations = 0 (Default)
```

10.1. Optimization Algorithms

The `algorithm` argument specifies the optimization algorithm. This argument takes one of the following three values:

- `lbfgs` A quasi-Newton optimizer. This is the default optimizer and also much faster than the other optimizers.
- `bfgs` A quasi-Newton optimizer.

- `newton` A Newton optimizer. This is the least efficient optimization algorithm, but has the advantage of setting its own stepsize.

See the Stan Reference Manual's Optimization chapter for a description of these algorithms.

10.2. The quasi-Newton optimizers

For both BFGS and L-BFGS optimizers, convergence monitoring is controlled by a number of tolerance values, any one of which being satisfied causes the algorithm to terminate with a solution. See the BFGS and L-BFGS configuration chapter for details on the convergence tests.

Both BFGS and L-BFGS have the following configuration arguments:

- `init_alpha` - The initial step size parameter. Must be a positive real number. Default value is 0.001
- `tol_obj` - Convergence tolerance on changes in objective function value. Must be a positive real number. Default value is 1^{-12}.
- `tol_rel_obj` - Convergence tolerance on relative changes in objective function value. Must be a positive real number. Default value is 1^{4}.
- `tol_grad` - Convergence tolerance on the norm of the gradient. Must be a positive real number. Default value is 1^{-8}.
- `tol_rel_grad` - Convergence tolerance on the relative norm of the gradient. Must be a positive real number. Default value is 1^{7}.
- `tol_param` - Convergence tolerance on changes in parameter value. Must be a positive real number. Default value is 1^{-8}.

The `init_alpha` argument specifies the 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. The default value is very small, 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.

In addition to the above, the L-BFGS algorithm has argument `history` which controls the size of the history it uses to approximate the Hessian. The value should be less than the dimensionality of the parameter space and, in general, relatively small values (5-10) are sufficient; the default value is 5.

If L-BFGS performs poorly but BFGS performs well, consider increasing the history size. Increasing history size will increase the memory usage, although this is unlikely to be an issue for typical Stan models.

10.3. The Newton optimizer

There are no configuration parameters for the Newton optimizer. It is not recommended because of the slow Hessian calculation involving finite differences.

11. Variational Inference Algorithm: ADVI

Stan implements an automatic variational inference algorithm, called Automatic Differentiation Variational Inference (ADVI) Kucukelbir et al. (2015). ADVI uses Monte Carlo integration to approximate the variational objective function, the ELBO (evidence lower bound). ADVI optimizes the ELBO in the real-coordinate space using stochastic gradient ascent. The measures of convergence are similar to the relative tolerance scheme of Stan's optimization algorithms.

The full set of configuration options available for the `variational` method is reported at the beginning of the sampler output file as csv comments. When the example model `bernoulli.stan` is run with `method=variational` via the command line with all default arguments, the resulting Stan csv file header comments show the complete set of default configuration options:

```
# method = variational
#   variational
#     algorithm = meanfield (Default)
#       meanfield
#     iter = 10000 (Default)
#     grad_samples = 1 (Default)
#     elbo_samples = 100 (Default)
#     eta = 1 (Default)
#     adapt
#       engaged = 1 (Default)
#       iter = 50 (Default)
#     tol_rel_obj = 0.01 (Default)
#     eval_elbo = 100 (Default)
#     output_samples = 1000 (Default)
```

The console output includes a notice that this algorithm is considered to be experimental:

```
EXPERIMENTAL ALGORITHM:
  This procedure has not been thoroughly tested and may be unstable
  or buggy. The interface is subject to change.
```

12. Standalone Generate Quantities

The `generate_quantities` method allows you to generate additional quantities of interest from a fitted model without re-running the sampler.

This method requires sub-argument `fitted_params` which takes as its value an existing Stan csv file that contains a sample from an equivalent model, i.e., a model with the same parameters, transformed parameters, and model blocks, conditioned on the same data.

13. Diagnosing HMC by Comparison of Gradients

CmdStan 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.

Mac OS and Linux

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

Windows

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

CmdStan Tools

This section provides a reference for the CmdStan tools:

- stanc
- stansummary
- diagnose
- print (deprecated)

14. `stanc`: Translating Stan to C++

CmdStan translates Stan programs to C++ using the Stan compiler program which is included in the CmdStan release `bin` directory as program `stanc`.

As of release 2.22, the CmdStan Stan to C++ compiler is written in OCaml. This compiler is called "stanc3" and has has its own repository https://github.com/stan-dev/stanc3, from which pre-built binaries for Linux, Mac, and Windows can be downloaded.

Prior to release 2.22, the Stan compiler program was compiled from C++ source code that was part of the core Stan library. This C++ compiler is still available as program `bin/stanc2`. This compiler is no longer being maintained, i.e., existing bugs will not be fixed and new functions and features are only available in the stanc3 compiler. Its intended use is as a diagnostic tool and backup for the new stanc3 compiler. For some future version, it will be dropped from the release altogether.

14.1. Instantiating the `stanc` Binary

Before the Stan compiler can be used, the binary stanc must be created. This can be done using the makefile as follows. For Mac and Linux:

```
make bin/stanc
```

For Windows:

```
make bin/stanc.exe
```

To build the `bin/stanc2` program, specify:

```
make bin/stanc2
```

14.2. The Stan Compiler Program

The Stan compiler program `stanc` converts Stan programs to C++ concepts. If the compiler encounters syntax errors in the program, it will provide an 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 generate the C++ translation of the example model `bernoulli.stan`. For Linux and Mac:

```
> cd <cmdstan-home>
> bin/stanc --o=bernoulli.hpp examples/bernoulli/bernoulli.stan
```

For Windows:

```
> cd <cmdstan-home>
> bin/stanc.exe --o=bernoulli.hpp examples/bernoulli/bernoulli.stan
```

The base name of the Stan program file determines the name of the C++ model class. 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.hpp` in the current directory. The final argument, `bernoulli.stan`, is the file from which to read the Stan program.

In practice, stanc is invoked indirectly, via the GNU Make utility, which contains rules that compile a Stan program to its corresponding executable. To build the simple Bernoulli model via `make`, we specify the name of the target executable file. On Mac and Linux, this is the name of the Stan program with the `.stan` omitted. On Windows, replace `.stan` with `.exe`, and make sure that the path is given with slashes and not backslashes. For Linux and Mac:

```
> make examples/bernoulli/bernoulli
```

For Windows:

```
> make examples/bernoulli/bernoulli.exe
```

The makefile rules first invoke the stanc compiler to translate the Stan model to C++ , then compiles and links the C++ code to a binary executable. The makefile variable `STANCFLAGS` can be used to to override the default arguments to stanc, e.g.,

```
> make STANCFLAGS="--include-paths=~/foo" examples/bernoulli/bernoulli
```

To use the stanc2 compiler instead of the stanc3 compiler, set the make option `STANC2`:

```
> make STANC2=TRUE examples/bernoulli/bernoulli
```

14.3. Command-Line Options for stanc3

The stanc3 compiler has the following command-line syntax:

```
> stanc (options) <model_file>
```

where `<model_file>` is a path to a Stan model file ending in suffix `.stan`.

The stanc3 options are:

- `--help` - Displays the complete list of stanc3 options, then exits.
- `--version` - Display stanc version number

- `--name=<model_name>` - Specify the name of the class used for the implementation of the Stan model in the generated C++ code.
- `--o=<file_name>` - Specify the name of the file into which the generated C++ is written.
- `--allow-undefined` - Do not throw a parser error if there is a function in the Stan program that is declared but not defined in the functions block.
- `--include_paths=<dir1,...dirN>` - Takes a comma-separated list of directories that may contain a file in an `#include` directive.
- `--use-opencl` - If set, will use additional Stan OpenCL features enabled in the Stan-to-C++ compiler.
- `--auto-format` - Pretty prints the program to the console.
- `--print-canonical` - Prints the canonicalized program to the console.
- `--print-cpp` - If set, output the generated C++ Stan model class to stdout.
- `--O` - Allow the compiler to apply all optimizations to the Stan code. **WARNING:** *This is currently an experimental feature!*
- `--warn-uninitialized` - Emit warnings about uninitialized variables to stderr. Currently an experimental feature.

The compiler also provides a number of debug options which are primarily of interest to stanc3 developers; use the `--help` option to see the full set.

14.4. Command-Line Options for stanc2

The stanc2 compiler has the same command-line syntax as the stanc3 compiler, but has fewer options:

- `--help` - Displays the complete list of stanc3 options, then exits.
- `--version` - Display stanc version number
- `--name=<model_name>` - Specify the name of the class used for the implementation of the Stan model in the generated C++ code.
- `--o=<file_name>` - Specify the name of the file into which the generated C++ is written.
- `--allow_undefined` - Do not throw a parser error if there is a function in the Stan program that is declared but not defined in the functions block.

- `--include_paths=<dir1,...dirN>` - Takes a comma-separated list of directories that may contain a file in an `#include` directive.

14.5. Using External C++ Code

The `--allow_undefined` flag can be passed to the call to stanc, which will allow undefined functions in the Stan language to be parsed without an error. We can then include a definition of the function in a C++ header file. This requires specifying two makefile variables: - `STANCFLAGS=--allow_undedefined` - `USER_HEADER=<header_file.hpp>`, where `<header_file.hpp>` is the name of a header file that defines a function with the same name and signature in a namespace that is formed by concatenating the `class_name` argument to stanc documented above to the string `_namespace`

As an example, consider the following variant of the Bernoulli example

```
functions {
       real make_odds(real theta);
}
data {
       int<lower=0> N;
       int<lower=0,upper=1> y[N];
     }
     parameters {
       real<lower=0,upper=1> theta;
}
model {
       theta ~ beta(1,1);  // uniform prior on interval 0,1
       y ~ bernoulli(theta);
     }
     generated quantities {
       real odds;
       odds = make_odds(theta);
}
```

Here the `make_odds` function is declared but not defined, which would ordinarily result in a parser error. However, if you put `STANCFLAGS = --allow_undefined` into the `make/local` file or into the `stanc` call, then the stanc compiler will translate this program to C++, but the generated C++ code will not compile unless you write a file such as `examples/bernoulli/make_odds.hpp` with the following lines

```
namespace bernoulli_model_namespace {
          template <typename T0__>  inline  typename
```

```
          boost::math::tools::promote_args<T0__>::type  make_odds(const T0__&
theta, std::ostream* pstream__) {
       return theta / (1 - theta);  }
       }
```

Given the above, the following `make` invocation should work

```
> make STANCFLAGS=--allow_undefined USER_HEADER=examples/bernoulli/make_odds
```

Alternatively, you could put `STANCFLAGS` and `USER_HEADER` into the `make/local` file instead of specifying them on the command-line.

If the function were more complicated and involved functions in the Stan Math Library, then you would need to prefix the function calls with `stan::math::` The `pstream__` argument is mandatory in the signature but need not be used if your function does not print any output. To see the necessary boilerplate look at the corresponding lines in the generated C++ file.

For more details about how to write C++ code using the Stan Math Library, see https://arxiv.org/abs/1509.07164.

15. `stansummary`: MCMC Output Analysis

CmdStan's `stansummary` utility processes one or more output files from a run or set of runs of Stan's HMC sampler. For all parameters and quantities of interest in the Stan program, `stansummary` reports a set of statistics including mean, standard deviation, percentiles, effective number of samples, and $\hat{R}$ values.

15.1. Building the stansummary Command

The CmdStan makefile task `build` compiles the `stansummary` utility into the `bin` directory. It can be compiled directly using the makefile as follows:

```
> cd <cmdstan-home>
> make bin/stansummary
```

15.2. Running the stansummary Command

The `stansummary` utility processes one or more output files from a run or set of runs of Stan's HMC sampler given a model and data. For all columns in the Stan csv output file `stansummary` reports a set of statistics including mean, standard deviation, percentiles, effective number of samples, and $\hat{R}$ values.

To run `stansummary` on the output file or files generated by a run of the sampler, on Mac or Linux:

```
<cmdstan-home>/bin/stansummary <file_1.csv> ... <file_N.csv>
```

On Windows, use backslashes to call the `stansummary.exe`.

```
<cmdstan-home>\bin\stansummary.exe <file_1.csv> ... <file_N.csv>
```

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.

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

15.3. Command-line Options

The `stansummary` command syntax provides a set of flags to customize the output which must precede the list of filenames:

- `--sig_figs` - The number of significant figures displayed in the output. Must be an integer value > 0. The default value is 2.

- `--csv_file` - Writes output to the specified filename in csv format using `#` for comment lines. Appends output to the file if it exists.
- `--percentiles` - An string containing an ordered list of integers in the range (0,100) specifying the percentiles to use in the set of output columns. Defaults to `"5, 50, 95"`.
- `--autocorr` - 0-based index into the list of filenames used to display the autocorrelation between all draws in that chain. For example, for a set of 4 csv files `--autocorr 0` will output the autocorrelation for chain 1 and `--autocorr 3` will examine the autocorrelation for chain 4. No autocorrelation output by default.

16. `diagnose`: Diagnosing Biased Hamiltonian Monte Carlo Inferences

CmdStan is distributed with a utility that is able to read in and analyze the output of one or more Markov chains to check for the following potential problems:

- Divergent transitions
- Transitions that hit the maximum treedepth
- Low E-BFMI values
- Low effective sample sizes
- High $\hat{R}$ values

The meanings of several of these problems are discussed in https://arxiv.org/abs/1701.02434.

16.1. Building the diagnose Command

The CmdStan makefile task `build` compiles the `diagnose` utility into the `bin` directory. It can be compiled directly using the makefile as follows:

```
> cd <cmdstan-home>
> make bin/diagnose
```

16.2. Running the diagnose Command

The `diagnose` command is executed on one or more output files, which are provided as command-line arguments separated by spaces. If there are no apparent problems with the output files passed to `diagnose`, it outputs a message that all transitions are within treedepth limit and that no divergent transitions were found. It problems are detected, it outputs a summary of the problem along with possible ways to mitigate it.

To fully exercise the `diagnose` command, we run 4 chains to sample from the Neal's funnel distribution, discussed in the Stan User's Guide reparameterization section https://mc-stan.org/docs/stan-users-guide/reparameterization-section.html. This program defines a distribution which exemplifies the difficulties of sampling from some hierarchical models:

```
parameters {
  real y;
  vector[9] x;
}
```

```
model {
  y ~ normal(0, 3);
  x ~ normal(0, exp(y/2));
}
```

This program is available on GitHub: https://github.com/stan-dev/example-models/blob/master/misc/funnel/funnel.stan

Stan has trouble sampling from the region where y is small and thus x is constrained to be near 0. This is due to the fact that the density's scale changes with y, so that a step size that works well when y is large is inefficient when y is small and vice-versa.

Running 4 chains produces output files `output_1.csv`, ..., `output_4.csv`. We run `diagnose` command on this fileset:

```
> bin/diagnose output_*.csv
```

The output is printed to the terminal window:

```
Processing csv files: output_1.csv, output_2.csv, output_3.csv, output_4.csv

Checking sampler transitions treedepth.
9 of 4000 (0.23%) transitions hit the maximum treedepth limit of 10, or 2^10
Trajectories that are prematurely terminated due to this limit will result i
For optimal performance, increase this limit.

Checking sampler transitions for divergences.
9 of 4000 (0.23%) transitions ended with a divergence.
These divergent transitions indicate that HMC is not fully able to explore t
Try increasing adapt delta closer to 1.
If this doesn't remove all divergences, try to reparameterize the model.

Checking E-BFMI - sampler transitions HMC potential energy.
The E-BFMI, 0.078, is below the nominal threshold of 0.3 which suggests that
If possible, try to reparameterize the model.

Effective sample size satisfactory.

The following parameters had split R-hat greater than 1.1:
  y
Such high values indicate incomplete mixing and biased estimation.
You should consider regularizing your model with additional prior informatio
```

```
Processing complete.
```

In this example, changing the model to use a non-centered parameterization is the only way to correct these problems. In this second model, the parameters `x_raw` and `y_raw` are sampled as independent standard normals, which is easy for Stan.

```
parameters {
  real y_raw;
  vector[9] x_raw;
}
transformed parameters {
  real y;
  vector[9] x;

  y = 3.0 * y_raw;
  x = exp(y/2) * x_raw;
}
model {
  y_raw ~ std_normal(); // implies y ~ normal(0, 3)
  x_raw ~ std_normal(); // implies x ~ normal(0, exp(y/2))
}
```

This program is available on GitHub: https://github.com/stan-dev/example-models/blob/master/misc/funnel/funnel_reparam.stan

We compile the program and run 4 chains, as before. Now the `diagnose` command doesn't detect any problems:

```
Processing csv files: output_1.csv, output_2.csv, output_3.csv, output_4.csv

Checking sampler transitions treedepth.
Treedepth satisfactory for all transitions.

Checking sampler transitions for divergences.
No divergent transitions found.

Checking E-BFMI - sampler transitions HMC potential energy.
E-BFMI satisfactory for all transitions.

Effective sample size satisfactory.

Split R-hat values satisfactory all parameters.
```

```
Processing complete, no problems detected.
```

16.3. `diagnose` Warnings and Recommendations

Divergent transitions after warmup

Stan uses Hamiltonian Monte Carlo (HMC) to explore the target distribution — the posterior defined by a Stan program + data — by simulating the evolution of a Hamiltonian system. In order to approximate the exact solution of the Hamiltonian dynamics we need to choose a step size governing how far we move each time we evolve the system forward. That is, the *step size controls the resolution of the sampler*.

Unfortunately, for particularly hard problems there are features of the target distribution that are too small for this resolution. Consequently the sampler misses those features and returns biased estimates. Fortunately, this mismatch of scales manifests as *divergences* which provide a practical diagnostic. If there are any divergences after warmup, then the samples may be biased.

If the divergent transitions cannot be eliminated by increasing the `adapt_delta` parameter, we have to find a different way to write the model that is logically equivalent but simplifies the geometry of the posterior distribution. This problem occurs frequently with hierarchical models and one of the simplest examples is Neal's Funnel, which is discussed in the reparameterization section of the Stan User's Guide.

Maximum treedepth exceeded

Warnings about hitting the maximum treedepth are not as serious as warnings about divergent transitions. While divergent transitions are a *validity* concern, hitting the maximum treedepth is an *efficiency* concern. Configuring the No-U-Turn-Sampler (the variant of HMC used by Stan) requires putting a cap on the depth of the trees that it evaluates during each iteration (for details on this see the *Hamiltonian Monte Carlo Sampling* chapter in the Stan Reference Manual. When the maximum allowed tree depth is reached it indicates that NUTS is terminating prematurely to avoid excessively long execution time.

This is controlled through the `max_depth` argument. If the number of transitions which exceed maximum treedepth is low, increasing `max_depth` may correct this problem.

Low E-BFMI values - sampler transitions HMC potential energy.

The sampler csv output column `energy__` is used to diagnose the accuracy of any Hamiltonian Monte Carlo sampler. If the standard deviation of `energy` is much larger than $\sqrt{D/2}$, where D is the number of *unconstrained* parameters, then the sampler is unlikely to be able to explore the posterior adequately. This is usually due to heavy-tailed posteriors and can sometime be remedied by reparameterizing the model.

The warning that some number of chains had an estimated Bayesian Fraction of Missing Information (BFMI) that was too low implies that the adaptation phase of the Markov Chains did not turn out well and those chains likely did not explore the posterior distribution efficiently. For more details on this diagnostic, see https://arxiv.org/abs/1604.00695. Should this occur, you can either run the sampler for more iterations, or consider reparameterizing your model.

Low effective sample sizes
Roughly speaking, the effective sample size (ESS) of a quantity of interest captures how many independent draws contain the same amount of information as the dependent sample obtained by the MCMC algorithm. Clearly, the higher the ESS the better. Stan uses $\hat{R}$ adjustment to use the between-chain information in computing the ESS. For example, in case of multimodal distributions with well-separated modes, this leads to an ESS estimate that is close to the number of distinct modes that are found.

Bulk-ESS refers to the effective sample size based on the rank normalized draws. This does not directly compute the ESS relevant for computing the mean of the parameter, but instead computes a quantity that is well defined even if the chains do not have finite mean or variance. Overall bulk-ESS estimates the sampling efficiency for the location of the distribution (e.g. mean and median).

Often quite smaller ESS would be sufficient for the desired estimation accuracy, but the estimation of ESS and convergence diagnostics themselves require higher ESS. We recommend requiring that the bulk-ESS is greater than 100 times the number of chains. For example, when running four chains, this corresponds to having a rank-normalized effective sample size of at least 400.

High $\hat{R}$
$\hat{R}$ (R-hat) convergence diagnostic compares the between- and within-chain estimates for model parameters and other univariate quantities of interest. If chains have not mixed well (ie, the between- and within-chain estimates don't agree), $\hat{R}$ is larger than 1. We recommend running at least four chains by default and only using the sample if $\hat{R}$ is less than 1.01. Stan reports $\hat{R}$ which is the maximum of rank normalized split-R-hat and rank normalized folded-split-R-hat, which works for thick tailed distributions and is sensitive also to differences in scale. For more details on this diagnostic, see https://arxiv.org/abs/1903.08008.

There is further discussion in https://arxiv.org/abs/1701.02434; however the correct resolution is necessarily model specific, hence all suggestions general guidelines only.

17. `print` (deprecated): MCMC Output Analysis

The `print` utility is deprecated, but is still available until CmdStan v3.0. Use the `stansummary` utility instead.

Appendices

This section contains the following appendices:

- JSON format
- RDump data format
- Bibliography

Betancourt, Michael. 2017. "A Conceptual Introduction to Hamiltonian Monte Carlo." *arXiv* 1701.02434. https://arxiv.org/abs/1701.02434.

Kucukelbir, Alp, Rajesh Ranganath, Andrew Gelman, and David M. Blei. 2015. "Automatic Variational Inference in Stan." *arXiv* 1506.03431. http://arxiv.org/abs/1506.03431.