CmdStan User's Guide

Version 2.34

Stan Development Team

Table of Contents

Overview

This document is a user's guide for CmdStan, the command-line interface to the Stan statistical modeling language. 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.

In CmdStan, statistical models written in the Stan probabilistic programming language are translated into a C++ program which is then compiled together with the CmdStan routines that provide the logic needed to manage all user inputs and program outputs and the Stan inference algorithms and math library. The resulting command line executable program can be used to

- do inference on data, producing an exact or approximate estimate of the posterior;
- generate new quantities of interest from an existing estimate;
- generate data from the model according to a given set of parameters.

The packages CmdStanR and CmdStanPy provide interfaces to CmdStan from R and Python, respectively, similarly, JuliaStan also interfaces with CmdStan.

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.

Stan documentation

- **Stan User's Guide** 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:

- **Stan Reference Manual** 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 together with an overview of the inference algorithms and posterior inference tools.
- **Stan Functions Reference** 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.

Part I

Quickstart Guide

1. CmdStan Installation

There are a few ways that you can install CmdStan. Depending on your operating system and your level of expertise, you can either:

- Use the conda package management system to install a pre-built version of CmdStan along with the required dependencies. *Recommended for Windows users*.
- Install the source code from GitHub CmdStan repository. This requires a modern C++ compiler and toolchain. See the C++ Toolchain section for further details.

1.1. Installation via `conda`

With conda, you can install CmdStan from the conda-forge channel. This will install a pre-built version of CmdStan along with the required dependencies (i.e. a C++ compiler, a version of Make, and required libraries) detailed below under [Source installation]. The conda installation is designed so one can use the R or Python bindings to CmdStan seamlessly. Additionally, it provides the command `cmdstan_model` to activate the CmdStan makefile from anywhere.

Note: This requires that conda has been installed already on your machine. You can either install miniconda, a free, minimal installer for conda or you can get the full Anaconda system which provides graphical installer wizards for MacOS and Windows users.

We recommend installing CmdStan in a new conda environment:

```
conda create -n stan -c conda-forge cmdstan
```

This command creates a new conda environment named `stan` and downloads and installs the `cmdstan` package as well as CmdStan and the required C++ toolchain.

To install into an existing conda environment, use the conda `install` command instead of `create`:

```
conda install -c conda-forge cmdstan
```

Whichever installation method you use, afterwards you must activate the new environment or deactivate/activate the existing one. For example, if you installed cmdstan into a new environment `stan`, run the command

```
conda activate stan
```

By default, the latest release of CmdStan is installed. If you require a specific release of CmdStan, CmdStan versions 2.26.1 and *newer* can be installed by specifying `cmdstan==VERSION` in the install command. For example to install an earlier version of CmdStan into your current conda environment, run the following command, then re-activate the environment

```
conda install -c conda-forge cmdstan=2.27.0
```

CmdStan install location under conda

A Conda environment is a directory that contains a specific collection of Conda packages. To see the locations of your conda environments, use the command

```
conda info -e
```

The shell environment variable `CONDA_PREFIX` points to the active conda environment (if any). Both CmdStan and the C++ toolchain are installed into the `bin` subdirectory of the conda environment directory, i.e., `$CONDA_PREFIX/bin/cmdstan` (Linux, MacOS), `%CONDA_PREFIX%\bin\cmdstan` (Windows).

Please report conda-specific install problems directly to the conda-forge issue tracker, here.

1.2. Installation from GitHub

Installation from GitHub consists of the following steps:

- Verify that you have a modern C++ toolchain. See the C++ Toolchain section for details.
- Download the CmdStan source code from GitHub
- Build the CmdStan libraries and executables
- Check the installation by compiling and running the CmdStan example model `bernoulli.stan`.

Downloading the source code

The GitHub source code is divided into sub-modules, each in its own repository. The CmdStan repo contains just the `cmdstan` module; the Stan inference engine algorithms and Stan math library functions are specified as submodules and stored in the GitHub repositories stan and math, respectively.

A CmdStan release is compressed tarfile which contains CmdStan and the Stan and math library submodules. The most recent CmdStan release is always available

as https://github.com/stan-dev/cmdstan/releases/latest. A CmdStan release is versioned by major, minor, patch numbers, e.g., "2.29.2". Please ensure you download a tarfile which is named "cmdstan-<version-number" rather than using the "Source Code" links at the bottom of the release. These are automatically generated by GitHub and do **not** contain the required submodules. The release tarfile unpacks into a directory named "cmdstan-", e.g. "cmdstan-2.29.2".

By cloning the CmdStan repository with argument `--recursive`, Git automatically initializes and updates each submodule in the repository, including nested submodules if any of the submodules in the repository have submodules themselves. The following command will download the source code from the current development branch of CmdStan into a directory named `cmdstan`:

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

Throughout this manual, we refer to this top-level CmdStan source directory as **`<cmdstan-home>`**. This directory contains the following subdirectories:

- 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)

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:

- 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    # on Windows use mingw32-make
```

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    # on Windows use mingw32-make
```

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 v2.29.2 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. See these instructions for details on changing the `PATH`. 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 for the new environment variable settings 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. Checking the Stan compiler

To check that the CmdStan installation is complete and in working order, run the following series of commands from the folder which CmdStan was installed.

On Linux and macOS:

```
# 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/bernoulli.data.json

# 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
```

On Windows:

```
# compile the example
> mingw32-make examples/bernoulli/bernoulli.exe

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

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

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

1.4. Troubleshooting the installation

Updates to CmdStan, changes in compiler options, or updates to the C++ toolchain may result in errors when trying to compile a Stan program. Often, these problems can be resolved by removing the existing CmdStan binaries and recompiling. To do this, you must run the makefile commands from the `<cmdstan-home>` directory:

```
> cd <cmdstan-home>
> make clean-all    # on Windows use mingw32-make
> make build
```

Common problems

This section contains solutions to problems reported on https://discourse.mc-stan.org

Compiler error message about PCH file

To speed up compilation, the Stan makefile pre-compiles parts of the core Stan

library. If these pre-compiled files are out of sync with the compiled model, the compiler will complain, e.g.:

```
error: PCH file uses an older PCH format that is no longer supported
```

In this case, clean and rebuild CmdStan, as shown in the previous section.

Windows: 'mingw32-make' is not recognised

If the C++ toolchain has been installed but not properly registered, then the call to `mingw32-make` will result in error message:

```
'mingw32-make' is not recognised as an internal or external command
```

To fix this, ensure you have followed the steps for adding the toolchain to your `PATH` and installing the additional utilities covered in the configuration instructions

Windows: 'g++' or 'cut' is not recognized

The CmdStan makefile uses a few shell utilities which might not be present in Windows, resulting in the error message:

```
'cut' is not recognized as an internal or external command,
operable program or batch file.
```

To fix this, ensure you have followed the steps for adding the toolchain to your `PATH` and installing the additional utilities covered in the configuration instructions

Spaces in paths to CmdStan or model

Both `make` and `mingw32-make` can fail when dealing with files in folders with a space somewhere in their file path. Particularly on Windows, this can be an issue when CmdStan, or the models you are trying to build, are placed in the `One Drive` folder.

Unfortunately, the errors created by this situation are not alwas informative. Some errors you may see are:

```
mingw32-make: *** INTERNAL: readdir: Invalid argument
```

```
make: *** [make/program:50: x.hpp] Error 2
```

If the (fully-expanded) folder path to CmdStan or the model you are trying to build contains a space, we recommend trying a different location if you encounter any issues during building.

1.5. C++ Toolchain

Compiling a Stan program requires a modern C++ compiler and the GNU Make build utility (a.k.a. "gmake"). These vary by operating system.

Linux

The required C++ compiler is `g++ 4.9 3`. On most systems the GNU Make utility is pre-installed and is the default `make` utility. There is usually a pre-installed C++ compiler as well, however, it may not be new enough. To check, run commands:

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

If these are at least at `g++` version 4.9.3 or later and `make` version 3.81 or later, no additional installations are necessary. It may still be desirable to update the C++ compiler `g++`, because later versions are faster.

To install the latest version of these tools (or upgrade an older version), use the following commands or their equivalent for your distribution, install via the commands:

```
sudo apt install g++
sudo apt install make
```

If you can't run `sudo`, you will need to ask your sysadmin or cluster administrator to install these tools for you.

MacOS

To install a C++ development environment on a Mac, use Apple's Xcode development environment https://developer.apple.com/xcode/.

From the Xcode home page `View in Mac App Store`.

- From the App Store, click `Install`, enter an Apple ID, and wait for Xcode to finish installing.
- Open the Xcode application, click top-level menu `Preferences`, click top-row button `Downloads`, click button for `Components`, click on the `Install` button to the right of the `Command Line Tools` entry, then wait for it to finish installing.
- Click the top-level menu item `Xcode`, then click item `Quit Xcode` to quit.

To test, open the Terminal application and enter:

```
clang++ --version
make --version
```

If you have installed XCode, but don't have `make`, you can install the XCode command-line tools via command:

```
xcode-select --install
```

Windows

The Windows toolchain consists of programs `g++`, the C++ compiler, and `mingw32-make`, the GNU Make utility. To check if these are present, open a command shell [1] and type:

```
g++ --version
mingw32-make --version
```

CmdStan is known compatible with the RTools40, RTools42, and RTools43 toolchains. These require slightly different steps to configure, so please follow the appropriate steps below. All toolchains will require updating your `PATH` variable, See these instructions for details on changing the `PATH` if you are unfamiliar. The following instructions will assume that the default installation directory was used, so be sure to update the paths accordingly if you have chosen a different directory.

1.5.0.3.1 RTools40 RTools40 provides both a standard `g++-8` toolchain and a `g++-10` Universal C Runtime (UCRT) toolchain. Note the that newer `g++-10` UCRT toolchain is only available for 64-bit systems, whereas the `g++-8` toolchain is available for both. Additionally the UCRT is only natively supported on Windows 10 and newer, older systems will require a Microsoft update

1.5.0.3.1.1 Installation Download the installer and complete the prompts to install.

Next, you need to add the location of the toolchain to your `PATH` environment variable so that it can be called from the command line. Add the following lines to your `PATH`:

```
C:\rtools40\usr\bin

# Add only one of the below
C:\rtools40\mingw32\bin   # 32-bit g++-8
C:\rtools40\mingw64\bin   # 64-bit g++-8
```

[1] To open a Windows command shell, first open the **Start Menu**, (usually in the lower left of the screen), select option **All Programs**, then option **Accessories**, then program **Command Prompt**. Alternatively, enter `[Windows+r]` (both keys together on the keyboard), and enter `cmd` into the text field that pops up in the Run window, then press `[Return]` on the keyboard to run.

```
C:\rtools40\ucrt64\bin    # 64-bit g++-10 (UCRT)
```

CmdStan additionally needs the `mingw32-make` utility, which you can install using RTools. Navigate to the installation directory (e.g., `C:\rtools40`) and launch the `msys2.exe` file. Execute the appropriate command below to install `mingw32-make` for your selected toolchain:

```
pacman -Sy mingw-w64-i686-make            # 32-bit g++-8
pacman -Sy mingw-w64-x86_64-make          # 64-bit g++-8
pacman -Sy mingw-w64-ucrt-x86_64-make     # 64-bit g++-10 (UCRT)
```

1.5.0.3.2 RTools42 & RTools43 Both RTools42 & RTools43 provide 64-bit UCRT toolchains, where RTools42 provides g++-10 and RTools43 provides g++-12. The installation/configuration is identical for both toolchains.

1.5.0.3.2.1 Installation Download the installer for your preferred toolchain and complete the prompts for installation:

- RTools42
- RTools43

Next, you need to add the toolchain directory to your `PATH` variable. Add the appropriate lines from below:

```
# RTools42
C:\rtools42\usr\bin
C:\rtools42\ucrt64\bin

# RTools43
C:\rtools43\usr\bin
C:\rtools43\ucrt64\bin
```

Next, you need to install the `mingw32-make` utility and some additional compiler dependencies. Navigate to the installation directory of the toolchain and launch the `msys2.exe` file. Execute the below commands to install the needed dependencies:

```
pacman -Sy mingw-w64-ucrt-x86_64-make mingw-w64-ucrt-x86_64-gcc
```

32-bit Builds

CmdStan defaults to a 64-bit build. On a 32-bit operating system, you must specify the make variable `BIT=32` as part of the `make` command, described in the next section.

1.6. Using GNU Make

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 or dire
/Users/mitzi/github/stan-dev/cmdstan/makefile:59: make/program: No such file or di
/Users/mitzi/github/stan-dev/cmdstan/makefile:60: make/tests: No such file or dire
/Users/mitzi/github/stan-dev/cmdstan/makefile:61: make/command: No such file or di
make: *** No rule to make target `make/command'.  Stop.
```

The conda-forge `cmdstan` package provides a solution to this problem via `cmdstan_model` command which lets you run the CmdStan makefile from anywhere to compile a Stan model.

Makefile syntax allows general pattern rules based on file suffixes. Stan programs must be stored in files with suffix `.stan`; the CmdStan makefile rules specify how to transform the Stan source code into a binary executable. 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).

To call Make, you invoke the utility name, either `make` or `mingw32-make`, followed by, in order:

- zero or more Make program options, then specify any Make variables as a series of

- zero of more Make variables, described below
- zero or more *target* names; the set of names is determined by the Makefile rules.

```
make <flags> <variables> <targets>
```

Makefile Variables

Make targets can be preceded by any number of Makefile variable name=value pairs. For example, to compile `../my_dir/my_program.stan` for an OpenCL (GPU) machine, set the makefile variable `STAN_OPENCL` to `TRUE`:

```
> make STAN_OPENCL=TRUE ../my_dir/my_program    # on Windows use mingw32-
make
```

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. For example, if you are working on a 32-bit machine, you would put the line `BIT=32` into the file `<cmdstan-home>/make/local` so that all CmdStan programs and Stan models compile properly.

The complete set of Makefile variables can be found in file `<cmdstan-home>/cmdstan/stan/lib/stan_math/make/compiler_flags`.

Make Targets

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

```
> make    # on Windows use mingw32-make
------------------------------------------------------------------------
------
CmdStan v2.33.1 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 v3.0)
    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 Stan pr

    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), 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 the
      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 the
      header file that is included. This defaults to "user_header.hpp" in the
      directory of the Stan program.
    STANC3_VERSION: When set, uses that tagged version specified; otherwise, downl
      the nightly version.
    STAN_CPP_OPTIMS: Turns on additonal compiler flags for performance.
    STAN_NO_RANGE_CHECKS: Removes the range checks from the model for performance.


  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/bernoull
    3. Look at the samples:
       > bin/stansummary output.csv


  Clean CmdStan:

    Remove the built CmdStan tools:
    > make clean-all

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

---

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;
  array[N] int<lower=0, upper=1> y;
}
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 in a row, without any editing `bernoulli.stan` or otherwise changing the system, on the second invocation, Make will determine that the executable is already newer than the Stan source file 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

The Stan probabilistic programming language is a programming language with a rich syntax, as such, it is often the case that a carefully written program contains errors.

The simplest class of errors are simple syntax errors such as forgetting the semi-colon statement termination marker at the end of a line, or typos such as a misspelled variable name. For example, if in the `bernoulli.stan` program, we introduce 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
```

Stan is a strongly-typed language; and the compiler will throw an error if statements

or expressions violate the type rules. The following trivial program `foo.stan` contains an illegal assignment statement:

```
data {
  real x;
}
transformed data {
  int y = x;
}
```

The Make command fails with the following:

```
Semantic error in 'foo.stan', line 5, column 2 to column 12:
   -------------------------------------------------
     3:  }
     4:  transformed data {
     5:    int y = x;
           ^
     6:  }
   -------------------------------------------------

Ill-typed arguments supplied to assignment operator =:
 lhs has type int and rhs has type real
```

The Stan Reference Manual provides a complete specification of the Stan programming language. The Stan User's Guide also contains a full description of the errors and warnings stanc can emit.

3.4. Troubleshooting C++ compiler or linker errors

If the stanc compiler successfully translates a Stan program to C++, the resulting C++ code should be valid C++ which can be compiled into an executable. 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.

The Make command prints the following message to the terminal at the point when it compiles and links the C++ file:

```
--- Compiling, linking C++ code ---
```

If the program fails to compile for any reason, the C++ compiler and linker will most likely print a long series of error messages to the console.

If this happens, please report the error, together with the Stan program on either the Stan Forums or on the Stan compiler GitHub issues tracker.

3.5. C++ compilation and linking flags

Users can set flags for the C++ compiler and linker and compiler to optimize their executables. We advise users to only do this once they are sure their basic setup of Cmdstan without flags works.

The `CXXFLAGS` and `LDFLAGS` makefile variables can be used to set compiler and linker flags respectively. We recommend setting these in the `make/local` file.

For example:

```
CXXFLAGS = -O2
```

A recommend a set of `CXXFLAGS` and `LDFLAGS` flags can be turned on by setting `STAN_CPP_OPTIMS=true` in the `make/local` file. These are tested compiler and link-time optimizations that can speed up execution of certain models. We have observed speedups up to 15 percent, but this depends on the model, operating system and hardware used. The use of these flags does considerably slow down compilation, so they are not used by default.

Optimizing by ignoring range checks

When assigning or reading from with vectors, row_vectors, matrices or arrays using indexing, Stan checks that a supplied index is valid (not out of range), which avoids segmentation faults and other difficult-to-debug runtime errors.

For some models these checks can represent a significant part of the models execution time. By setting the `STAN_NO_RANGE_CHECKS=true` makefile flag in the `make/local` file the range checks can be removed. Use this flag with caution (only once the indexing has been validated). In case of any unexpected behavior remove the flag for easier debugging.

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)
      save_metric = 0 (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)
    num_chains = 1 (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 seconds.
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.

There are two different ways of running multiple chains, with the `num_chains` argument using a single executable and by using the Unix and DOS shell to run multiple executables.

Using the num_chains argument to run multiple chains

The `num_chains` argument can be used for all of Stan's samplers with the exception of the `static` HMC engine.

Example that will run 4 chains:

```
./bernoulli sample num_chains=4 data file=bernoulli.data.json output file=output.c
```

If the model was not compiled with `STAN_THREADS=true`, the above command will run 4 chains sequentially and will produce the sample in `output_1.csv`, `output_2.csv`, `output_3.csv`, `output_4.csv`. A suffix with the chain id is appended to the provided output filename (`output.csv` in the above command).

If the model was compiled with `STAN_THREADS=true`, the chains can run in parallel, with the `num_threads` argument defining the maximum number of threads used to run the chains. If the model uses no within-chain parallelization (`map_rect` or `reduce_sum` calls), the below command will run 4 chains in parallel, provided there are cores available:

```
./bernoulli sample num_chains=4 data file=bernoulli.data.json output file=output.c
```

If the model uses within-chain parallelization (`map_rect` or `reduce_sum` calls), the threads are automatically scheduled to run the parallel parts of a single chain or run the sequential parts of another chains. The below call starts 4 chains that can use 16 threads. At a given moment a single chain may use all 16 threads, 1 thread, anything in between, or can wait for a thread to be available. The scheduling is left to the Threading Building Blocks scheduler.

```
./bernoulli_par sample num_chains=4 data file=bernoulli.data.json output file=outp
```

Using shell for running multiple chains

To run multiple chains given a model and data, either sequentially or in parallel, we can also 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__,theta
```

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 (static HMC sampler)

Because the above header is from the NUTS sampler, it has columns `treedepth__`, `n_leapfrog__`, and `divergent__` and doesn't have column `int_time__`. 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=(1,1,1,1);
```

```
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    R_ha
lp__            -7.3  1.8e-02    0.75   -8.8  -7.0  -
6.8  1.8e+03  2.4e+04  1.0e+00
accept_stat__   0.89  2.7e-03    0.17   0.52  0.96   1.0  3.9e+03  5.1e+04  1.0e+0
stepsize__       1.1  7.5e-02    0.11   0.93   1.2   1.2  2.0e+00  2.6e+01  2.5e+1
treedepth__      1.4  8.1e-03    0.49    1.0   1.0   2.0  3.6e+03  4.7e+04  1.0e+0
n_leapfrog__     2.3  1.7e-02    0.98    1.0   3.0   3.0  3.3e+03  4.3e+04  1.0e+0
divergent__     0.00      nan    0.00   0.00  0.00  0.00      nan      nan      na
energy__         7.8  2.6e-02     1.0    6.8   7.5   9.9  1.7e+03  2.2e+04  1.0e+0
theta           0.25  2.9e-03    0.12  0.079  0.23  0.46  1.7e+03  2.1e+04  1.0e+0

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 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 = 1e-12 (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 = 87122538 (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 0.002 and the length of the gradient roughly 3e-05 (0.00003).

```
Initial log joint probability = -6.85653
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # eval
       5      -5.00402    0.00184936   3.35074e-
05           1           1        8
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
-5.00402,0.200003
```

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. 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.

All of the optimizers stream per-iteration intermediate approximations to the command line console. The sub-argument `save_iterations` specifies whether or not to save the intermediate iterations to the output file. Allowed values are 0 or 1, corresponding to `False` and `True` respectively. The default value is 0, i.e., intermediate iterations are not saved to the output file. Running the optimizer with `save_iterations=1` writes both the initial log joint probability and values for all iterations to the output CSV file.

Running the example model with option `save_iterations=1`, i.e., the command

```
> ./bernoulli optimize save_iterations=1 data file=bernoulli.data.json
```

produces CSV file output rows:

```
lp__,theta
-6.85653,0.493689
-6.10128,0.420936
-5.02953,0.22956
-5.00517,0.206107
-5.00403,0.200299
-5.00402,0.200003
```

6. Variational Inference using Pathfinder

The CmdStan method `pathfinder` uses the Pathfinder algorithm of Zhang et al. (2022). Pathfinder is a variational method for approximately sampling from differentiable log densities. Starting from a random initialization, Pathfinder locates normal approximations to the target density along a quasi-Newton optimization path, with local covariance estimated using the negative inverse Hessian estimates produced by the L-BFGS optimizer. Pathfinder returns draws from the Gaussian approximation with the lowest estimated Kullback-Leibler (KL) divergence to the true posterior.

Pathfinder differs from the ADVI method in that it uses quasi-Newton optimization on the log posterior instead of stochastic gradient descent (SGD) on the Monte Carlo computation of the evidence lower bound (ELBO). Pathfinder's approach is both faster and more stable than that of ADVI. Compared to ADVI and short dynamic HMC runs, Pathfinder requires one to two orders of magnitude fewer log density and gradient evaluations, with greater reductions for more challenging posteriors.

A single run of the Pathfinder algorithm generates a set of approximate draws. Inference is improved by running multiple Pathfinder instances and using Pareto-smoothed importance resampling (PSIS) of the resulting sets of draws. This better matches non-normal target densities and also eliminates minor modes. By default, the `pathfinder` method uses 4 independent Pathfinder runs, each of which produces 1000 approximate draws, which are then importance resampled down to 1000 final draws.

The following is a minimal call the Pathfinder algorithm using defaults for everything but the location of the data file.

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

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

The first part of the console output reports on the configuration used.

```
method = pathfinder
  pathfinder
    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)
    num_psis_draws = 1000 (Default)
    num_paths = 4 (Default)
    psis_resample = 1 (Default)
    calculate_lp = 1 (Default)
    save_single_paths = 0 (Default)
    max_lbfgs_iters = 1000 (Default)
    num_draws = 1000 (Default)
    num_elbo_draws = 25 (Default)
id = 1 (Default)
data
  file = examples/bernoulli/bernoulli.data.json
init = 2 (Default)
random
  seed = 1995513073 (Default)
output
  file = output.csv (Default)
  diagnostic_file =  (Default)
  refresh = 100 (Default)
  sig_figs = -1 (Default)
  profile_file = profile.csv (Default)
num_threads = 1 (Default)
```

The rest of the output describes the progression of the algorithm.

By default, the Pathfinder algorithm runs 4 single-path Pathfinders in parallel, the uses importance resampling on the set of returned draws to produce the specified number of draws.

```
Path [1] :Initial log joint density = -11.543343
Path [1] : Iter      log prob        ||dx||      ||grad||     alpha      alpha0
              5      -6.748e+00      1.070e-03   1.707e-
05    1.000e+00  1.000e+00       126 -6.220e+00 -6.220e+00
Path [1] :Best Iter: [5] ELBO (-6.219833) evaluations: (126)
Path [2] :Initial log joint density = -7.443345
Path [2] : Iter      log prob        ||dx||      ||grad||     alpha      alpha0
              5      -6.748e+00      9.936e-05   3.738e-
07    1.000e+00  1.000e+00       126 -6.164e+00 -6.164e+00
Path [2] :Best Iter: [5] ELBO (-6.164015) evaluations: (126)
Path [3] :Initial log joint density = -18.986308
Path [3] : Iter      log prob        ||dx||      ||grad||     alpha      alpha0
```

```
              5      -6.748e+00      2.996e-04   4.018e-
06    1.000e+00  1.000e+00       126 -6.201e+00 -6.201e+00
Path [3] :Best Iter: [5] ELBO (-6.200559) evaluations: (126)
Path [4] :Initial log joint density = -8.304453
Path [4] : Iter      log prob        ||dx||      ||grad||     alpha      alpha0
              5      -6.748e+00      2.814e-04   2.034e-
06    1.000e+00  1.000e+00       126 -6.221e+00 -6.221e+00
Path [4] :Best Iter: [3] ELBO (-6.161276) evaluations: (126)
Total log probability function evaluations:8404
```

Pathfinder outputs a StanCSV file file which contains the importance resampled draws from multi-path Pathfinder. The initial CSV comment rows contain the complete set of CmdStan configuration options. Next is the column header line, followed the set of approximate draws. The Pathfinder algorithm first outputs `lp_approx__`, the log density in the approximating distribution, and `lp__`, the log density in the target distribution, followed by estimates of the model parameters, transformed parameters, and generated quantities.

```
lp_approx__,lp__,theta
-2.4973, -8.2951, 0.0811852
-0.87445, -7.06526, 0.160207
-0.812285, -7.07124, 0.35819
...
```

The final lines are comment lines which give timing information.

```
# Elapsed Time: 0.016000 seconds (Pathfinders)
#               0.003000 seconds (PSIS)
#               0.019000 seconds (Total)
```

Pathfinder provides option `save_single_paths` which will save output from the single-path Pathfinder runs. See section Pathfinder Method for details.

7. Variational Inference using ADVI

The CmdStan method `variational` uses the Automatic Differentiation Variational Inference (ADVI) algorithm of Kucukelbir et al. (2017) to provide an approximate posterior distribution of the model conditioned on the data. The approximating distribution it uses is a Gaussian in the unconstrained variable space, either a fully factorized Gaussian approximation, specified by argument `algorithm=meanfield` option, or a Gaussian approximation using a full-rank covariance matrix, specified by argument `algorithm=fullrank`. By default, ADVI uses option `algorithm=meanfield`.

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: the default option `algorithm=meanfield` and the default 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 seconds.
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 CONVERGED

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.

8. 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 (i.e., a set of draws) from the fitted model runs the generated quantities block of the new program using the the existing sample by plugging in the per-draw parameter estimates for the computations in the generated quantities block. See the Stan User's Guide section Stand-alone generated quantities and ongoing prediction.

To illustrate how this works we use the `generate_quantities` method to do posterior predictive checks using the estimate of `theta` given the example bernoulli model and data, following the posterior predictive simulation procedure in the Stan User's Guide.

We write a program `bernoulli_ppc.stan` which contains the following generated quantities block, with comments to explain the procedure:

```
generated quantities {
  real<lower=0, upper=1> theta_rep;
  array[N] int y_sim;
  // use current estimate of theta to generate new sample
```

```
  for (n in 1:N) {
    y_sim[n] = bernoulli_rng(theta);
  }
  // estimate theta_rep from new sample
  theta_rep = sum(y_sim) * 1.0 / N;
}
```

The rest of the program is the same as in `bernoulli.stan`.

The `generate_method` requires the sub-argument `fitted_params` which takes as its value the name of a Stan CSV file. The per-draw parameter estimates from the `fitted_params` file will be used to run the generated quantities block.

If we run the `bernoulli.stan` program for a single chain to generate a sample in file `bernoulli_fit.csv`:

```
> ./bernoulli sample data file=bernoulli.data.json output file=bernoulli_fit.csv
```

Then we can run the `bernoulli_ppc.stan` to carry out the posterior predictive checks:

```
> ./bernoulli_ppc generate_quantities fitted_params=bernoulli_fit.csv \
                  data file=bernoulli.data.json \
                  output file=bernoulli_ppc.csv
```

The output file `bernoulli_ppc.csv` consists of just the values for the variables declared in the generated quantities block, i.e., `theta_rep` and the elements of `y_sim`:

```
# model = bernoulli_ppc_model
# method = generate_quantities
#   generate_quantities
#     fitted_params = bernoulli_fit.csv
# id = 0 (Default)
# data
#   file = bernoulli.data.json
# init = 2 (Default)
# random
#   seed = 2135140492 (Default)
# output
#   file = bernoulli_ppc.csv
#   diagnostic_file =  (Default)
#   refresh = 100 (Default)
theta_rep,y_sim.1,y_sim.2,y_sim.3,y_sim.4,y_sim.5,y_sim.6,y_sim.7,y_sim.8,y_sim.9,
```

```
0.2,0,0,1,0,0,0,0,0,1,0
0.3,1,0,0,1,0,1,0,0,0,0
0.8,1,0,1,1,1,1,1,1,1,0
0.1,0,0,0,0,0,1,0,0,0,0
0.3,0,0,0,0,0,0,1,1,1,0
```

Note: the only relevant analysis of the resulting CSV output is computing per-column statistics; this can easily be done in Python, R, Excel or similar, or you can use the CmdStanPy and CmdStanR interfaces which provide a better user experience for this workflow.

Given the current implementation, to see the fitted parameter values for each draw, create a copy variable in the generated quantities block, e.g.:

```
generated quantities {
  real<lower=0, upper=1> theta_cp = theta;
  real<lower=0, upper=1> theta_rep;
  array[N] int y_sim;
  // use current estimate of theta to generate new sample
  for (n in 1:N) {
    y_sim[n] = bernoulli_rng(theta);
  }
  // estimate theta_rep from new sample
  theta_rep = sum(y_sim) * 1.0 / N;
}
```

Now the output is slightly more interpretable: `theta_cp` is the same as the `theta` used to generate the values `y_sim[1]` through `y_sim[1]`. Comparing columns `theta_cp` and `theta_rep` allows us to see how the uncertainty in our estimate of `theta` is carried forward into our predictions:

```
theta_cp,theta_rep,y_sim.1,y_sim.2,y_sim.3,y_sim.4,y_sim.5,y_sim.6,y_sim.7,y_sim.8
0.102391,0,0,0,0,0,0,0,0,0,0
0.519567,0.2,0,1,0,0,1,0,0,0,0
0.544634,0.6,1,0,0,0,0,1,1,1,1
0.167651,0,0,0,0,0,0,0,0,0,0
0.167651,0.1,1,0,0,0,0,0,0,0,0
```

Part II

Reference Manual

9. 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.
  - `log_prob` - compute the log probability and gradient of the model for one set of parameters.
  - `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 pseudo-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.

9.1. 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 prints an error message to stderr and returns a non-zero return code. 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; variable
```

9.2. 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. This option is only valid for the iterative algorithms `sample` and `variational`.

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.

The keyword value pair `sig_figs=<int>` specifies the number of significant digits for all numerical values in the output files. Allowable values are between 1 and 18, which is the maximum amount of precision available for 64-bit floating point arithmetic. The default value is 6. ***Note:*** increasing `sig_figs` above the default will increase the size of the output CSV files accordingly.

The keyword value pair `profile_file=<filepath>` specifies the location of the output file for profiling data. If the model uses no profiling, the output profile file is not produced. If the model uses profiling and `profile_file` is unspecified,

the profiling data is written to a file named `profile.csv` in the current working directory.

The keyword value pair `save_cmdstan_config=<int>` specifies whether to save the configuration options used to run the program to a file named `<output file>_config.json` alongside the other output files. The default value is 0, which means the configuration file is not saved. The contents of this file are similar to the comments in the Stan CSV file, but should be more portable across versions and easier to parse.

9.3. 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 initialized will be initialized randomly over the range $[-2, 2]$.

9.4. 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>`.

9.5. 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 1.

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.

9.6. 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
```

9.7. Error messages and return codes

CmdStan executables and utility programs use streams standard output (stdout) and standard error (stderr) to report information and error messages, respectively. Some methods also generate warning messages when the algorithm detects potential problems with the inference. Depending on the method, these messages are sent to either standard out or standard error.

All program executables provide a return code between 0 and 255:

- 0 - Program ran to termination as expected.
- value in range [1 : 125] - Method invoked could not run due to problems with model or data.
- value > 128 - Fatal error during execution, process terminated by signal. To determine the signal number, subtract 128 from the return value, e.g. return code 139 results from termination signal 11 (segmentation violation).

A non-zero return code or outputs sent to stderr indicate problems with the inference. However, a return code of zero and absence of error messages doesn't necessarily mean that the inference is valid, it is still necessary to validate the inferences using all available summary and diagnostic techniques.

10. 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 sample method configuration options:

```
# method = sample (Default)
#   sample
#     num_samples = 1000 (Default)
#     num_warmup = 1000 (Default)
#     save_warmup = 0 (Default)
#     thin = 1 (Default)
#     adapt
#       engaged = 1 (Default)
#       gamma = 0.050000 (Default)
#       delta = 0.800000 (Default)
#       kappa = 0.750000 (Default)
#       t0 = 10.000000 (Default)
#       init_buffer = 75 (Default)
#       term_buffer = 50 (Default)
#       window = 25 (Default)
#       save_metric = 0 (Default)
#     algorithm = hmc (Default)
#       hmc
#         engine = nuts (Default)
#           nuts
#             max_depth = 10 (Default)
#         metric = diag_e (Default)
#         metric_file =  (Default)
```

```
#         stepsize = 1.000000 (Default)
#         stepsize_jitter = 0.000000 (Default)
#     num_chains = 1 (Default)
```

10.1. 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 values 0 or 1, corresponding 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. Some users familiar with older approaches to MCMC sampling might be used to thinning to eliminate an expected autocorrelation in the samples. HMC is not nearly as susceptible to this autocorrelation problem and thus thinning is generally not required nor advised, as HMC can produce *anticorrelated* draws, which

increase the effective sample size beyond the number of draws from the posterior. Thinning should only be used in circumstances where storage of the samples is limited and/or RAM for later processing the samples is limited.

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.

10.2. 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.

The boolean sub-argument `save_metric` was added in Stan version 2.34. When `save_metric=1` (true), the adapted stepsize and metric are output as JSON at the end of adaptation. The saved metric file name is the output file basename with the suffix `_metric.json`, e.g., if using the default output filename `output.csv`, the saved metric file will be `output_metric.json`. This metric file can be reused in subsequent sampler runs as the initial metric, via sampler argument `metric_file`.

Step size optimization configuration

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 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. Models with difficult posterior geometries may required increasing the `delta` argument closer to 1; we recommend first trying to raise it to 0.9 or at most 0.95. Values about 0.95 are strong indication of bad geometry; the better solution is to change the model geometry through reparameterization which could yield both more efficient and faster sampling.
- `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.
- `kappa` - 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:

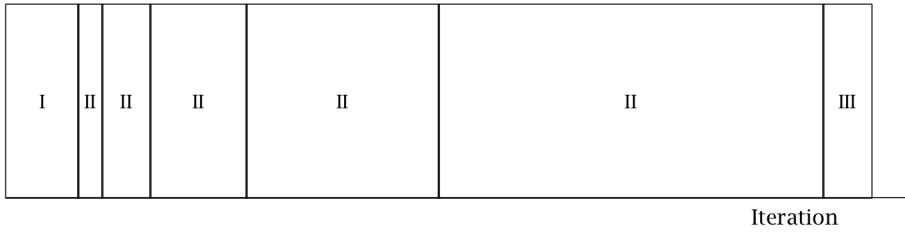
- `init_buffer` - The number of iterations spent tuning the step size at the outset of adaptation.
- `window` - The initial number of iterations devoted to tune the metric, will be doubled successively.
- `term_buffer` - The number of iterations used to re-tune the step size once the metric has been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



10.3. 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.

10.4. 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, and the gradients on the unconstrained scale and log probabilities for all parameters in the model. By default, no auxiliary output file is produced.

10.5. Multiple chains in one executable

As described in the quickstart section on parallelism, the preferred way to run multiple chains is to use the `num_chains` argument.

This will run multiple chains of MCMC from the same executable, which can save on memory usage due to only needing one copy of the model and data. As noted in the quickstart guide, this will be done in parallel if the model was compiled with `STAN_THREADS=true`.

The `num_chains` argument changes the meanings of several other arguments when it is greater than `1` (the default). Many arguments are now interpreted as a "template" which is used for each chain.

For example, when `num_chains=2`, the argument `output file=foo.csv` *no longer produces a file* `foo.csv`, but instead produces two files, `foo_1.csv` and `foo_2.csv`. If you also supply `id=5`, the files produced will be `foo_5.csv` and `foo_6.csv` – `id=5` gives the id of the first chain, and the remaining chains are sequential from there.

This also applies to input files, like those used for initialization. For example, if `num_chains=3` and `init=bar.json` will first look for `bar_1.json`. If it exists, it will use `bar_1.json` for the first chain, `bar_2.json` for the second, and so on. If `bar_1.json` does not exist, it falls back to looking for `bar.json`, and if it exists, uses the same initial values for each chain. The numbers in these filenames are also based on the `id` argument, which defaults to `1`.

10.6. Examples - older parallelism

Note: Many of these examples can be simplified by using the `num_chains` argument.

The Quickstart Guide MCMC Sampling chapter section on multiple chains also 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.

This creates multiple copies of the model process which will all load the data.

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 on an unconstrained scale 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
```

To verify that the specified values will be used by the sampler, you can run the sampler with option `algorithm=fixed_param`, so that the initial values are used to generate the sample. Since this generates a set of identical draws, setting `num_warmp=0` and `num_samples=1` saves unnecessary iterations. As the output values are also on the constrained scale, the set of reported values will match the set of specified initial values.

For example, if we run the example Bernoulli model with specified initial value for parameter "theta":

```
{ "theta" : 0.5 }
```

via command:

```
./bernoulli sample algorithm=fixed_param num_warmup=0 num_samples=1 \
            init=bernoulli.init.json data file=bernoulli.data.json
```

The resulting output CSV file contains a single draw:

```
lp__,accept_stat__,theta
0,0,0.5
#
#  Elapsed Time: 0 seconds (Warm-up)
#                0 seconds (Sampling)
#                0 seconds (Total)
#
```

Specifying the metric and stepsize

An initial estimate 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 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
```

As of Stan versione 2.34, the adapted metric can be saved in JSON format, via sub-argument `save_metric`, described above. This allows for no or minimal adaptation starting from this file. It is still necessary to specify the `stepsize` argument as well as the `metric_file` arguments; the former is the value of the `stepsize` element in the saved metric file, and the later is the metric file path.

Changing the NUTS-HMC adaptation parameters

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
```

11. 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)
#     jacobian = 0 (Default)
#     iter = 2000 (Default)
#     save_iterations = 0 (Default)
```

11.1. Jacobian adjustments

The `jacobian` argument specifies whether or not the call to the model's log probability function should include the log absolute Jacobian determinant of inverse parameter transforms. Without the Jacobian adjustment, optimization returns the (regularized) maximum likelihood estimate (MLE), $\text{argmax}_\theta \, p(y|\theta)$, the value which maximizes the likelihood of the data given the parameters, (including prior terms). Applying the Jacobian adjustment produces the maximum a posteriori estimate

(MAP), the maximum value of the posterior distribution, $\text{argmax}_\theta \; p(y|\theta) \; p(\theta)$. By default this value is `0` (false), do not include the Jacobian adjustment.

11.2. 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.

All of the optimizers stream per-iteration intermediate approximations to the command line console. The sub-argument `save_iterations` specifies whether or not to save the intermediate iterations to the output file. Allowed values are 0 or 1, corresponding to `False` and `True` respectively. The default value is 0, i.e., intermediate iterations are not saved to the output file.

11.3. 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 section 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_size` 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.

11.4. 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.

12. Pathfinder Method for Approximate Bayesian Inference

The Pathfinder algorithm is described in section Pathfinder overview.

The `pathfinder` method runs multi-path Pathfinder by default, which returns a PSIS sample over the draws from several individual ("single-path") Pathfinder runs. Argument `num_paths` specifies the number of single-path Pathfinders, the default is 4. If `num_paths` is set to 1, then only one individual Pathfinder is run without the PSIS reweighting of the sample.

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

```
method = pathfinder
  pathfinder
    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)
    num_psis_draws = 1000 (Default)
    num_paths = 4 (Default)
    psis_resample = 1 (Default)
    calculate_lp = 1 (Default)
    save_single_paths = 0 (Default)
    max_lbfgs_iters = 1000 (Default)
    num_draws = 1000 (Default)
    num_elbo_draws = 25 (Default)
```

12.1. Pathfinder Configuration

- `num_psis_draws` - Final number of draws from multi-path pathfinder. Must be a positive integer. Default value is 1000.

- `num_paths` - Number of single pathfinders. Must be a positive integer. Default value is 4.
- `save_single_paths` - When True (1), save outputs from single pathfinders. Valid values: $\{0, 1\}$. Default is 0 (False).
- `max_lbfgs_iters` - Maximum number of L-BFGS iterations. Must be a positive integer. Default value is 1000.
- `num_draws` - Number of approximate posterior draws for each single pathfinder. Must be a positive integer. Default value is 1000. Can differ from `num_psis_draws`.
- `num_elbo_draws` - Number of Monte Carlo draws to evaluate ELBO. Must be a positive integer. Default value is 25.
- `psis_resample` - If True (1), perform psis resampling on samples returned from individual pathfinders. If False (0), returns all `num_paths * num_draws samples` draws from the individual pathfinders. Valid values: $\{0, 1\}$. Default is 1 (True).
- `calculate_lp` - If True (1), log probabilities of the approximate draws are calculated and returned with the output. If False (0), each pathfinder will only calculate the lp values needed for the ELBO calculation. If False, psis resampling cannot be performed and the algorithm returns `num_paths * num_draws samples`. The output will still contain any lp values used when calculating ELBO scores within L-BFGS iterations. Valid values: $\{0, 1\}$. Default is 1 (True).

12.2. L-BFGS Configuration

Arguments `init_alpha` through `history_size` are the full set of arguments to the L-BFGS optimizer and have the same defaults for optimization.

12.3. Multi-path Pathfinder CSV files

By default, the `pathfinder` method uses 4 independent Pathfinder runs, each of which produces 1000 approximate draws, which are then importance resampled down to 1000 final draws. The importance resampled draws are output as a StanCSV file.

The CSV files have the following structure:

- The full set of configuration options available for the `pathfinder` method is reported at the beginning of the sampler output file as CSV comments.

- The CSV header row consists of columns `lp_approx__`, `lp__`, and the Stan model parameters, transformed parameters, and generated quantities in the order in which they are declared in the Stan program.
- The data rows contain the draws from the single- or multi-path run.
- Final comments containing timing information.

12.4. Single-path Pathfinder Outputs.

The boolean option `save_single_paths` is used to save both the draws and the ELBO iterations from the individual Pathfinder runs. When `save_single_paths` is 1, the draws from each are saved to StanCSV files with the same format as the PSIS sample and the ELBO evaluations along the L-BFGS trajectory for each are saved as JSON. Given an output file name, CmdStan adds suffixes to the base filename to distinguish between the output files. For the default output file name `output.csv` and default number of runs (4), the resulting CSV files are

```
output.csv
output_path_1.csv
output_path_1.json
output_path_2.csv
output_path_2.json
output_path_3.csv
output_path_3.json
output_path_4.csv
output_path_4.json
```

The individual sample CSV files have the same structure as the PSIS sample CSV file. The JSON files contain information from each ELBO iteration.

To see how this works, we run Pathfinder on the centered-parameterization of the `eight-schools` model, where the posterior distribution has a funnel shape:

```
> eight_schools pathfinder save_single_paths=1 data file=eight_schools.data.json
```

Each JSON file records the approximations to the target density at each point along the trajectory of the L-BFGS optimization algorithms.

```
{
  "0": {
    "iter": 0,
    "unconstrained_parameters": [1.00595, -0.503687, 1.79367, 0.99083, 0.498077, -
0.65816, 1.49176, -1.22647, 1.62911, 0.767445],
```

```
    "grads": [-0.868919, 0.45198, -0.107675, -0.0123304, 0.163172, 0.354362, -
0.108746, 0.673306, -0.102268, -4.51445]
  },
  "1": {
    "iter": 1,
    "unconstrained_parameters": [1.00595, -0.503687, 1.79367, 0.99083, 0.498077, -
0.65816, 1.49176, -1.22647, 1.62911, 0.767445],
    "grads": [-0.868919, 0.45198, -0.107675, -0.0123304, 0.163172, 0.354362, -
0.108746, 0.673306, -0.102268, -4.51445],
    "history_size": 1,
    "lbfgs_success": true,
    "pathfinder_success": true,
    "x_center": [0.126047, -0.065048, 1.55708, 0.958509, 0.628075, -
0.217041, 1.32032, -0.561338, 1.42988, 1.23213],
    "logDetCholHk": -2.6839,
    "L_approx": [[-0.0630456, -0.0187959], [0, 1.08328]],
    "Qk": [[-0.361073, 0.5624], [0.183922, -0.279474], [-
0.0708175, 0.15715], [-0.00917823, 0.0215802], [0.0606019, -
0.0814513], [0.164071, -0.285769], [-0.057723, 0.112428], [0.276376, -
0.424348], [-0.0620524, 0.131786], [-0.846488, -0.531094]],
    "alpha": [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    "full": false,
    "lbfgs_note": ""
  },
  ...,
  "171": {
    "iter": 171,
    "unconstrained_parameters": [1.60479, 1.60479, 1.60479, 1.60479, 1.60479, 1.60
35.7821],
    "grads": [2.66927e+15, -0.117312, -0.0639521, -2.66927e+15, -
0.0445885, 0.0321579, 0.00499827, -0.163952, -0.032084, 6.4073],
    "history_size": 5,
    "lbfgs_success": true,
    "pathfinder_success": true,
    "x_center": [5.58876e+15, 5.58876e+15, 5.58876e+15, 5.58876e+15, 5.58876e+15,
2.02979e+17],
    "logDetCholHk": 299.023,
    "L_approx": [[4.6852e+06, 4.6852e+06, 4.6852e+06, 4.6852e+06, 4.6852e+06, 4.68
1.70162e+08], [0, 2.19511e+13, 2.19511e+13, 2.19511e+13, 2.19511e+13, 2.19511e+13,
7.97244e+14], [0, 0, 2.19511e+13, 2.19511e+13, 2.19511e+13, 2.19511e+13, 2.19511e+
7.97244e+14], [0, 0, 0, 2.19511e+13, 2.19511e+13, 2.19511e+13, 2.19511e+13, 2.1951
7.97244e+14], [0, 0, 0, 0, 2.19511e+13, 2.19511e+13, 2.19511e+13, 2.19511e+13, 2.1
7.97244e+14], [0, 0, 0, 0, 0, 2.19511e+13, 2.19511e+13, 2.19511e+13, 2.19511e+13,
```

```
7.97244e+14], [0, 0, 0, 0, 0, 0, 2.19511e+13, 2.19511e+13, 2.19511e+13, -
7.97244e+14], [0, 0, 0, 0, 0, 0, 0, 2.19511e+13, 2.19511e+13, -
7.97244e+14], [0, 0, 0, 0, 0, 0, 0, 0, 2.19511e+13, -
7.97244e+14], [0, 0, 0, 0, 0, 0, 0, 0, 0, 2.89552e+16]],
    "Qk": [],
    "alpha": [1.11027e-12, 2.24669e-12, 2.05603e-12, 3.71177e-12, 5.7855e-
12, 1.80169e-12, 3.40291e-12, 2.29699e-12, 3.43423e-12, 1.25815e-08],
    "full": true,
    "lbfgs_note": ""
  },
  "172": {
    "iter": 172,
    "unconstrained_parameters": [1.60531, 1.60531, 1.60531, 1.60531, 1.60531, 1.60
35.801],
    "grads": [-0, -0.11731, -0.0639469, 0.0179895, -
0.0445842, 0.0321643, 0.00500256, -0.163947, -0.0320824, 7],
    "history_size": 5,
    "lbfgs_success": false,
    "pathfinder_success": false,
    "lbfgs_note": ""
  }
}
```


Option `num_paths=1` runs one single-path Pathfinder and output CSV file contains the draws from that run without PSIS reweighting. The combination of arguments `num_paths=1 save_single_paths=1` creates just two output files, the CSV sample and the set of ELBO iterations. In this case, the default output file name is "output.csv" and the default diagnostic file name is "output.json".

13. Variational Inference Algorithm: ADVI

CmdStan can approximate the posterior distribution using variational inference. The approximation is a Gaussian in the unconstrained variable space.

Stan implements an automatic variational inference algorithm, called Automatic Differentiation Variational Inference (ADVI) Kucukelbir et al. (2017). 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 algorithm progression consists of an adaptation phase followed by a sampling phase. The 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. The algorithm runs until the mean change in ELBO drops below the specified tolerance.

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.
```

13.1. Variational algorithms

Stan implements two variational algorithms. The `algorithm` argument specifies the variational algorithm.

- `algorithm=meanfield` - Use a fully factorized Gaussian for the approximation. This is the default algorithm.
- `algorithm=fullrank` Use a Gaussian with a full-rank covariance matrix for the approximation.

13.2. Configuration

- `iter=<int>` Maximum number of iterations. Must be > 0. Default is 10000.
- `grad_samples=<int>` Number of samples for Monte Carlo estimate of gradients. Must be > 0. Default is 1.
- `elbo_samples=<int>` Number of samples for Monte Carlo estimate of ELBO (objective function). Must be > 0. Default is 100.
- `eta=<double>` Stepsize weighting parameter for adaptive stepsize sequence. Must be > 0. Default is 1.0.
- `adapt` Warmup Adaptation keyword, takes sub-arguments:
  - `engaged=<boolean>` Adaptation engaged? Valid values: $(0, 1)$. Default is 1.
  - `iter=<int>` Maximum number of adaptation iterations. Must be > 0. Default is 50.
- `tol_rel_obj=<double>` Convergence tolerance on the relative norm of the objective. Must be > 0. Default is 0.01.
- `eval_elbo=<int>` Evaluate ELBO every Nth iteration. Must be > 0. Default is 100.
- `output_samples=<int>` Number of posterior samples to draw and save. Must be > 0. Default is 1000.

13.3. CSV output

The output file consists of the following pieces of information:

- The full set of configuration options available for the `variational` method is reported at the beginning of the sampler output file as CSV comments.
- The first three output columns are labelled `lp__`, `log_p__`, `log_g__`, the rest are the model parameters.
- The stepsize adaptation information is output as CSV comments following column header row.
- The following line contains the mean of the variational approximation.
- The rest of the output contains `output_samples` number of samples drawn from the variational approximation.

To illustrate, we call Stan's variational inference on the example model and data:

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

By default, the output file is `output.csv`. Lines 1 - 28 contain configuration information:

```
# 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)
...
```

The column header row is:

```
lp__,log_p__,log_g__,theta
```

The stepsize adaptation information is:

```
# Stepsize adaptation complete.
# eta = 1
```

The reported mean variational approximations information is:

```
0,0,0,0.214911
```

That is, the estimate for `theta` given the data is `0.2`.

The following is a sample based on this approximation:

```
0,-14.0252,-5.21718,0.770397
0,-7.05063,-0.10025,0.162061
0,-6.75031,-0.0191099,0.241606
...
```

14. Standalone Generate Quantities

The `generate_quantities` method allows you to generate additional quantities of interest from a fitted model without re-running the sampler. For an overview of the uses of this feature, see the QuickStart Guide section and the Stan User's Guide section on Stand-alone generated quantities and ongoing prediction.

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.

If we run the `bernoulli.stan` program for a single chain to generate a sample in file `bernoulli_fit.csv`:

```
> ./bernoulli sample data file=bernoulli.data.json output file=bernoulli_fit.csv
```

Then we can run the `bernoulli_ppc.stan` to carry out the posterior predictive checks:

```
> ./bernoulli_ppc generate_quantities fitted_params=bernoulli_fit.csv \
                  data file=bernoulli.data.json \
                  output file=bernoulli_ppc.csv
```

The `fitted_params` file must be a Stan CSV file; attempts to use a regular CSV file will result an error message of the form:

```
Error reading fitted param names from sample csv file <filename.csv>
```

The `fitted_params` file must contain columns corresponding to legal values for all parameters defined in the model. If any parameters are missing, the program will exit with an error message of the form:

```
Error reading fitted param names from sample csv file <filename.csv>
```

The parameter values of the `fitted_params` are on the constrained scale and must obey all constraints. For example, if we modify the contencts of the first reported draw in `bernoulli_fit.csv` so that the value of `theta` is outside the declared bounds `real<lower=0, upper=1>`, the program will return the following error message:

```
Exception: lub_free: Bounded variable is 1.21397, but must be in the interval [0,
```

15. Laplace sampling

The `laplace` method produces a sample from a normal approximation centered at the mode of a distribution in the unconstrained space. If the mode is a maximum a posteriori (MAP) estimate, the samples provide an estimate of the mean and standard deviation of the posterior distribution. If the mode is a maximum likelihood estimate (MLE), the sample provides an estimate of the standard error of the likelihood. In general, the posterior mode in the unconstrained space doesn't correspond to the mean (nor mode) in the constrained space, and thus the sample is needed to infer the mean as well as the standard deviation. (See this case study for a visual illustration.)

This is computationally inexpensive compared to exact Bayesian inference with MCMC. The goodness of this estimate depends on both the estimate of the mode and how much the true posterior in the unconstrained space resembles a Gaussian.

15.1. Configuration

This method takes 2 arguments:

- `jacobian` - Whether or not the Jacobian adjustment should be included in the gradient. The default value is 1 (include adjustment). (Note: in optimization, the default value is `0`, for historical reasons.)
- `mode` - Input file of parameters values on the constrained scale. When Stan's `optimize` method is used to estimate the modal values, the value of boolean argument `jacobian` should be 0 if `optimize` was run with default settings, i.e., the input is the MLE estimate; if `optimize` was run with argument `jacobian=1`, then the `laplace` method default setting, `jacobian=1`, should be used.

15.2. CSV output

The output file consists of the following pieces of information:

- The full set of configuration options available for the `laplace` method is reported at the beginning of the output file as CSV comments.
- Output columns `log_p__` and `log_q__`, the unnormalized log density and the unnormalized density of the Laplace approximation, respectively. These

can be used for diagnostics and importance sampling.

- Output columns for all model parameters on the constrained scale.

15.3. Example

To get an approximate estimate of the mode and standard deviation of the example Bernoulli model given the example dataset:

- find the MAP estimate by running optimization with argument `jacobian=1`
- run the Laplace estimator using the MAP estimate as the `mode` argument.

Because the default output file name from all methods is `output.csv`, a more informative name is used for the output of optimization. We run the commands from the CmdStan home directory. This results in a sample with mean 2.7 and standard deviation 0.12. In comparison, running the NUTS-HMC sampler results in mean 2.6 and standard deviation 0.12.

```
./examples/bernoulli/bernoulli optimize jacobian=1 \
  data file=examples/bernoulli/bernoulli.data.json \
  output file=bernoulli_optimize_lbfgs.csv random seed=1234


./examples/bernoulli/bernoulli laplace mode=bernoulli_optimize_lbfgs.csv \
 data file=examples/bernoulli/bernoulli.data.json random seed=1234
```

The header and first few data rows of the output sample are shown below.

```
# stan_version_major = 2
# stan_version_minor = 31
# stan_version_patch = 0
# model = bernoulli_model
# start_datetime = 2022-12-20 01:01:14 UTC
# method = laplace
#   laplace
#     mode = bernoulli_lbfgs.csv
#     jacobian = 1 (Default)
#     draws = 1000 (Default)
# id = 1 (Default)
# data
#   file = examples/bernoulli/bernoulli.data.json
# init = 2 (Default)
# random
#   seed = 875960551 (Default)
# output
```

```
#   file = output.csv (Default)
#   diagnostic_file =  (Default)
#   refresh = 100 (Default)
#   sig_figs = -1 (Default)
#   profile_file = profile.csv (Default)
# num_threads = 1 (Default)
# stanc_version = stanc3 v2.31.0-7-g20444266
# stancflags =
log_p__,log_q__,theta
-9.4562,-2.33997,0.0498545
-6.9144,-0.0117349,0.182898
-7.18171,-0.746034,0.376428
...
```

16. Extracting log probabilities and gradients for diagnostics

CmdStan can return the computed log probability and the gradient with respect to a set of parameters.

This is similar to the `diagnose` subcommand, but the output format differs and the results here are not compared with those from finite differences.

Note: Startup and data initialization costs mean that this method is not an efficient way to calculate these quantities. It is provided only for convenience and should not be used for serious computation.

16.1. Configuration

This method takes 3 arguments:

- `jacobian` - Whether or not the Jacobian adjustment for constrained parameters should be included in the gradient. Default value is 1 (include adjustment).
- `constrained_params` - Input file of parameters values on the constrained scale. A single set of constrained parameters can be specified using JSON format. Alternatively, the input file can be set of draws in StanCSV format.
- `unconstrained_params` - Input file (JSON or R dump) of parameter values on unconstrained scale. These files should contain a single variable, called `params_r`, which is a flattened vector of all unconstrained parameters. If this object is two dimensional, each entry should be a vector of the same form and the output will feature multiple rows.

Only one of `constrained_params` and `unconstrained_params` can be specified.

For more on the differences between constrained and unconstrained parameters, see the Stan reference manual section on variable transforms.

16.2. CSV output

The output file consists of the following pieces of information:

- The full set of configuration options available for the `log_prob` method is reported at the beginning of the output file as CSV comments.
- Column headers, the first column is labelled `lp__`, and the rest are named after parameters. These will be the unconstrained parameters, regardless of whether constrained or unconstrained parameters were supplied as input.
- Values which correspond to the value of the log density (column 1) and the gradient with respect to each parameter (remaining columns).

For example, if we have a file called params.json:

```
{
    "theta" : 0.1
}
```

We can run the example model:

```
/bernoulli log_prob constrained_params=params.json data file=bernoulli.data.json
```

This yields

```
# stan_version_major = 2
# stan_version_minor = 31
# stan_version_patch = 0
# model = bernoulli_model
# start_datetime = 2022-11-17 20:46:06 UTC
# method = log_prob
#   log_prob
#     unconstrained_params =  (Default)
#     constrained_params = params.json
#     jacobian = 1 (Default)
# id = 1 (Default)
# data
#   file = bernoulli.data.json
# init = 2 (Default)
# random
#   seed = 2390820139 (Default)
# output
#   file = output.csv (Default)
#   diagnostic_file =  (Default)
#   refresh = 100 (Default)
#   sig_figs = -1 (Default)
#   profile_file = profile.csv (Default)
# num_threads = 1 (Default)
```

```
# stanc_version = stanc3 2.31.0 (Linux)
# stancflags =
lp_,theta
-7.856,1.8
```

17. Diagnosing HMC by Comparison of Gradients

CmdStan has a basic diagnostic feature that will calculate the gradients of the initial state and compare them with gradients calculated by finite differences. Discrepancies between the two indicate that there is a problem with the model or initial states or else there is a bug in Stan.

To allow for the possibility of adding other kinds of diagnostic tests, the `diagnose` method argument configuration has subargument `test` which currently only takes value `gradient`. There are two available gradient test configuration arguments:

- `epsilon` - The finite difference step size. Must be a positive real number. Default value is 1^{-6}
- `error` - The error threshold. Must be a positive real number. Default value is 1^{-6}

To run on the different platforms with the default configuration, use one of the following.

Mac OS and Linux

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

Windows

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

To relax the test threshold, specify the `error` argument as follows:

```
> ./my_model diagnose test=gradient error=0.0001 data file=my_data
```

To see how this works, we run diagnostics on the example bernoulli model:

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

Executing this command prints output to the console and as a series of comment lines to the output csv file. The console output is:

```
method = diagnose
  diagnose
    test = gradient (Default)
```

```
      gradient
        epsilon = 9.9999999999999995e-07 (Default)
        error = 9.9999999999999995e-07 (Default)
id = 0 (Default)
data
  file = bernoulli.data.json
init = 2 (Default)
random
  seed = 2152196153 (Default)
output
  file = output.csv (Default)
  diagnostic_file =  (Default)
  refresh = 100 (Default)

TEST GRADIENT MODE

 Log probability=-8.42814

 param idx           value           model     finite diff           error
         0       0.0361376         -3.1084         -3.1084    -2.37554e-10
```

The same information is printed to the output file as csv comments, i.e., each line is prefixed with a pound sign #.

18. Parallelization

Stan provides three ways of parallelizing execution of a Stan model:

- multi-threading with Intel Threading Building Blocks (TBB),
- multi-processing with Message Passing Interface (MPI) and
- manycore processing with OpenCL.

18.1. Multi-threading with TBB

In order to exploit multi-threading in a Stan model, the models must be rewritten to use the `reduce_sum` and `map_rect` functions. For instructions on how to rewrite Stan models to use these functions see Stan's User guide chapter on parallelization, the reduce_sum case study or the Multithreading and Map-Reduce tutorial.

Compiling

Once a model is rewritten to use the above-mentioned functions, the model must be compiled with the `STAN_THREADS` makefile flag. The flag can be supplied in the `make` call but we recommend writing the flag to the `make/local` file.

An example of the contents of `make/local` to enable threading with TBB:

```
STAN_THREADS=true
```

The model is then compiled as normal:

```
make path/to/model
```

Running

Before running a multi-threaded model, we need to specify the maximum number of threads the program can run (total threads for all chains). This is done by setting the `num_threads` argument. Valid values for `num_threads` are positive integers and -1. If `num_threads` is set to -1, all available cores will be used.

Generally, this number should not exceed the number of available cores for best performance.

Example:

```
./model sample data file=data.json num_threads=4 ...
```

When the model is compiled with `STAN_THREADS` we can sample with multiple chains with a single executable (see section running multiple chains for cases when

this is available). When running multiple chains `num_threads` is the maximum number of threads that can be used by all the chains combined. The exact number of threads that will be used for each chain at a given point in time is determined by the TBB scheduler. The following example start 2 chains with 8 total threads available:

```
./model sample num_chains=2 data file=data.json num_threads=8 ...
```

18.2. Multi-processing with MPI

In order to use multi-processing with MPI in a Stan model, the models must be rewritten to use the `map_rect` function. By using MPI, the model can be parallelized across multiple cores or a cluster. MPI with Stan is supported on MacOS and Linux.

Dependencies

Compiling and running Stan models with MPI requires that the system has an MPI implementation installed. For Unix systems the most commonly used implementations are MPICH and OpenMPI.

Compiling

Once a model is rewritten to use `map_rect`, additional makefile flags must be written to the `make/local`. These are:

- `STAN_MPI`: Enables the use of MPI with Stan if `true`.
- `CXX`: The name of the MPI C++ compiler wrapper. Typically `mpicxx`.
- `TBB_CXX_TYPE`: The C++ compiler the MPI wrapper wraps. Typically `gcc` on Linux and `clang` on macOS.

An example of `make/local` on Linux:

```
STAN_MPI=true
CXX=mpicxx
TBB_CXX_TYPE=gcc
```

The model is then compiled as normal:

```
make path/to/model
```

Running

The Stan model compiled with `STAN_MPI` is run using an MPI launcher. The MPI standard suggests using `mpiexec`, but a vendor wrapper for the launcher like `mpirun` can also be used. The launcher is supplied the path to the built executable and the number of processes to start: `-n X` for `mpiexec` or `-np X` for `mpirun` where `X` is replaced by the integer representing the number of processes.

Example for running a model with six processes:

```
mpiexec -n 6 path/to/model sample data file=data.json ...
```

18.3. OpenCL

Dependencies

OpenCL is supported on most modern CPUs and GPUs. In order to run OpenCL-enabled Stan models, an OpenCL runtime for the target device must be installed. This subsection lists installation instructions for OpenCL runtimes of the commonly-found devices.

In order to check if any OpenCL-enabled device and its runtime is already present use the `clinfo` tool. On Linux, `clinfo` can typically be installed with the default package manager (for example `sudo apt-get install clinfo` on Ubuntu). For Windows, pre-built `clinfo` binary can be found here.

Also use `clinfo` to verify successful installation of OpenCL runtimes.

NVIDIA GPU

- Linux:

  Install the NVIDIA GPU driver and the NVIDIA CUDA Toolkit. On Ubuntu the commands to install both is:

  ```
  sudo apt update
  sudo apt install nvidia-driver-460 nvidia-cuda-toolkit
  ```

  Replace the driver version (`460` in the above case) with the lastest number at the time of installation.

- Windows:

  Install the NVIDIA GPU Driver and CUDA Toolkit.

AMD GPU

- Linux:

  Install `Radeon Software for Linux` available here.

- Windows:

  We recommend installing the open source OCL-SDK.

AMD CPU

Install the open source PoCL.

Intel CPU/GPU

Follow Intel's install instructions given here (requires registration).

Compiling

In order to enable the OpenCL backend the model must be compiled with the `STAN_OPENCL` makefile flag. The flag can be supplied in the `make` call but we recommend writing the flag to the `make/local` file.

An example of the contents of `make/local` to enable parallelization with OpenCL:

```
STAN_OPENCL=true
```

If you are using OpenCL with an integrated GPU you also need to add the `INTEGRATED_OPENCL` flag, as the sharing of memory between CPU and GPU is slightly different with integrated graphics:

```
INTEGRATED_OPENCL=true
```

The model is then compiled as normal:

```
make path/to/model
```

Running

The Stan model compiled with `STAN_OPENCL` can also be supplied the OpenCL platform and device IDs of the target device. These IDs determine the device on which to run the OpenCL-supported functions on. You can list the devices on your system using the `clinfo` program. If the system has one GPU and no OpenCL CPU runtime, the platform and device IDs of the GPU are typically `0`. In that case you can also omit the OpenCL IDs as the default `0` IDs are used in that case.

We supply these IDs when starting the executable as shown below:

```
path/to/model sample data file=data.json opencl platform=0 device=1
```

Part III

CmdStan Utilities

19. 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`. One can view the complete stanc documentation in the Stan User's Guide.

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.

19.1. 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
```

This is also done as part of the `make build` command.

19.2. 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
```

20. `stansummary`: MCMC Output Analysis

The CmdStan `stansummary` program reports statistics for one or more sampler chains over all sampler and model parameters and quantities of interest. The statistics reported include both summary statistics of the estimates and diagnostic statistics on the sampler chains, reported in the following order:

- Mean - sample mean
- MCSE - Monte Carlo Standard Error, a measure of the amount of noise in the sample
- StdDev - sample standard deviation
- Quantiles - default 5%, 50%, 95%
- N_eff - effective sample size - the number of independent draws in the sample
- N_eff/S - the number of independent draws per second
- R_hat - $\hat{R}$ statistic, a measure of chain equilibrium, must be within 0.05 of 1.0.

When reviewing the `stansummary` output, it is important to check the final three output columns first - these are the diagnostic statistics on chain convergence and number of independent draws in the sample. A $\hat{R}$ statistic of greater than 1.05 indicates that the chain has not converged and therefore the sample is not drawn from the posterior, thus the estimates of the mean and all other summary statistics are invalid.

Estimation by sampling produces an approximate value for the model parameters; the MCSE statistic indicates the amount of noise in the estimate. Therefore MCSE column is placed next to the sample mean column, in order to make it easy to compare this sample with others.

For more information, see the Posterior Analysis chapter of the Stan Reference Manual which describes both the theory and practice of MCMC estimation techniques. The summary statistics and the algorithms used to compute them are described in sections Notation for samples and Effective Sample Size.

20.1. 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
```

20.2. Running the `stansummary` program

The `stansummary` utility processes one or more output files from a set of chains from one run of the HMC sampler. 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>
```

For example, after running 4 chains to fit the example model `eight_schools.stan` to the supplied example data file, we run `stansummary` on the resulting Stan CSV output files to get the following report:

```
> bin/stansummary eight_*.csv
Input files: eight_1.csv, eight_2.csv, eight_3.csv, eight_4.csv
Inference for Stan model: eight_schools_model
4 chains: each with iter=(1000,1000,1000,1000); warmup=(0,0,0,0); thin=(1,1,1,1);

Warmup took (0.048, 0.060, 0.047, 0.045) seconds, 0.20 seconds total
Sampling took (0.057, 0.058, 0.061, 0.066) seconds, 0.24 seconds total

                Mean     MCSE  StdDev     5%   50%   95%  N_Eff  N_Eff/s    R_hat

lp__             -18     0.33     5.1    -26   -19  -
9.1    233      963      1.0
accept_stat__   0.88  1.6e-02    0.23   0.21  0.98  1.00    203      838  1.0e+00
stepsize__      0.18  2.2e-02   0.031   0.14  0.20  0.22    2.0      8.3  3.9e+13
treedepth__      3.8  5.9e-02    0.78    2.0   4.0   5.0    175      724  1.0e+00
n_leapfrog__      18  1.3e+00     9.4    7.0    15    31     51      212  1.0e+00
divergent__    0.015  4.1e-03    0.12   0.00  0.00  0.00    865     3576  1.0e+00
energy__          23  3.4e-01     5.5     13    23    32    258     1066  1.0e+00

mu               7.9     0.16     5.1  -0.23   7.9    16   1021     4221      1.0
theta[1]          12     0.30     8.6  -0.48    11    28    837     3459      1.0
theta[2]         7.8     0.15     6.4   -2.7   7.7    18   1717     7096     1.00
theta[3]         6.1     0.19     7.7   -6.5   6.5    18   1684     6958      1.0
theta[4]         7.5     0.15     6.7   -3.1   7.4    18   2026     8373      1.0
theta[5]         4.7     0.17     6.4   -6.7   5.3    15   1391     5747     1.00
theta[6]         5.9     0.16     6.7   -5.8   6.2    16   1673     6915     1.00
theta[7]          11     0.22     7.0  0.057    10    23   1069     4419      1.0
theta[8]         8.3     0.20     7.9   -4.2   8.0    22   1503     6209     1.00
tau              7.2     0.26     5.2    1.5   5.9    17    401     1657      1.0
```

```
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).
```

The console output information consists of

- Model, chains, and timing summaries
- Sampler parameter statistics
- Model parameter statistics
- Sampling algorithm - either nuts (shown here) or static HMC.

There is one row per parameter and the row order in the summary report corresponds to the column order in the Stan CSV output file.

Sampler parameters

The initial Stan CSV columns provide information on the sampler state for each draw:

- `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 (static HMC sampler)

Because we ran the NUTS sampler, the above summary reports sampler parameters `treedepth__`, `n_leapfrog__`, and `divergent__`; the static HMC sampler would report `int_time__` instead.

Model parameters and quantities of interest

The remaining Stan CSV columns report the values of all parameters, transformed parameters, and generated quantities in the order in which these variables are declared in the Stan program. For container variables, i.e., vector, row_vector, matrix, and array variables, the statistics for each element are reported separately, in row-major order. The `eight_schools.stan` program parameters block contains the following parameter variable declarations:

```
  real mu;
  array[J] real theta;
  real<lower=0> tau;
```

In the example data, `J` is 8; therefore the stansummary listing reports on `theta[1]` through `theta[8]`.

20.3. Command-line options

The `stansummary` command syntax provides a set of flags to customize the output which must precede the list of filenames. When invoked with no arguments or with the `-h` or `--help` option, the program prints the usage message to the console and exits.

```
Report statistics for one or more Stan CSV files from a HMC sampler run.
Example:  stansummary model_chain_1.csv model_chain_2.csv
Options:
  -a, --autocorr [n]          Display the chain autocorrelation for the n-
th
                              input file, in addition to statistics.
  -c, --csv_filename [file]   Write statistics to a CSV file.
  -h, --help                  Produce help message, then exit.
  -p, --percentiles [values]  Percentiles to report as ordered set of
                              comma-separated numbers from (0.1,99.9), inclusive.
                              Default is 5,50,95.
  -s, --sig_figs [n]          Significant figures reported. Default is 2.
                              Must be an integer from (1, 18), inclusive.
  -i, --include_param [name]  Include the named parameter in the summary output.
                              By default, all parameters in the file are summarize
                              passing this argument one or more times will filter
                              the output down to just the requested arguments.
```

Both short an long option names are allowed. Short names are specified as `-<o> <value>`; long option names can be specified either as `--<option>=<value>` or `--<option> <value>`.

The `--percentiles` argument can also be passed an empty string `""`, which results in no percentiles being displayed in the output of the command.

The amount of precision in the sampler output limits the amount of real precision in the summary report. CmdStan's command line interface also has output argument `sig_figs`. The default sampler output precision is 6. The `--sig_figs` argument to the stansummary program should not exceed the `sig_figs` argument to the sampler.

21. `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.

21.1. 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
```

21.2. 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. 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 leapf
Trajectories that are prematurely terminated due to this limit will result in slow
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 the pos
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 HMC m
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 information or a
```

```
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.
```

21.3. `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 sometimes 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.

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

The `print` utility is deprecated, but is still available until CmdStan v3.0. It has been replaced by the `stansummary` utility.

Part IV

Appendices

23. Stan CSV File Format

The output from all CmdStan methods is in CSV format. A Stan CSV file is a data table where the columns are the method and model parameters and quantities of interest. Each row contains one record's worth of data in plain-text format using the comma character (','') as the field delimiter (hence the name).

For the Stan CSV files, data is strictly numerical, however, possible values include both positive and negative infinity and "Not-a-Number" which are represented as the strings `NaN`, `inf`, `+inf`, `-inf`. All other values are written in decimal notation with at most 6 digits of precision.

Stan CSV files have a header row containing the column names. They also make extensive use of CSV comments, i.e., lines which begin with the `#` character. In addition to initial and final comment rows, some methods also put comment rows in the middle of the data table, which makes it difficult to use many of the commonly used CSV parser packages.

23.1. CSV column names and order

The data table is laid out with zero or more method-specific columns followed by the Stan program variables declared in the parameter block, then the variables in the transformed parameters block, finally variables declared in the generated quantities, in declaration order.

Stan provides three types of container objects: arrays, vectors, and matrices. In order to output all elements of a container object, it is necessary to choose an indexing notation and a serialization order. The Stan CSV file indexing notation is

- The column name consists of the variable name followed by the element indices.
- Indices are delimited by periods ('.').
- Indexing is 1-based, i.e., given a dimension of size N, the first element index is 1 and the last element index is N.
- Tuples are laid out element-by-element, with each tuple slot being delimited by a colon (':').

Container variables are serialized in column major order, a.k.a. "Fortran" order. In column major-order, all elements of column 1 are listed in ascending order, followed

by all elements of column 2, thus the first index changes the slowest and the last index changes the fastest.

To see how this works, consider a 3-dimensional variable with dimension sizes 2, 3, and 4, e.g., an array of matrices, a 2-D array of vectors or row_vectors, or a 3-D array of scalars. Given a Stan program with model parameter variable:

```
array[2, 3, 4] real foo;
```

The Stan CSV file will require 24 columns to output the elements of `foo`. The first 6 columns will be labeled:

```
foo.1.1.1, foo.1.1.2, foo.1.1.3, foo.1.1.4, foo.1.2.1, foo.1.2.2
```

The final 6 columns will be labeled:

```
foo.2.2.3, foo.2.2.4, foo.2.3.1, foo.2.3.2, foo.2.3.3, foo.2.3.4
```

To see how a tuple would be laid out, consider the following variable:

```
tuple(real, array[3] real) bar;
```

This will correspond to 4 columns in the CSV file, which are labeled

```
bar:1,bar:2.1,bar:2.2,bar:2.3
```

23.2. MCMC sampler CSV output

The sample method produces both a Stan CSV output file and a diagnostic file which contains the sampler parameters together with the gradients on the unconstrained scale and log probabilities for all parameters in the model.

To see how this works, we show snippets of the output file resulting from the following command:

```
./bernoulli sample save_warmup=1 num_warmup=200 num_samples=100 \
            data file=bernoulli.data.json \
            output file=bernoulli_samples.csv
```

Sampler Stan CSV output file

The sampler output file contains the following:

- Initial comment rows listing full CmdStan argument configuration.
- Header row
- Data rows containing warmup draws, if run with option `save_warmup=1`
- Comment rows for adaptation listing step size and metric used for sampling
- Sampling draws
- Comment rows giving timing information

Initial comments rows: argument configuration

All configuration arguments are listed, one per line, indented according to CmdStan's hierarchy of arguments and sub-arguments. Arguments not overtly specified on the command line are annotated as `(Default)`.

In the above example the `num_samples`, `num_warmup`, and `save_warmup` arguments were specified, whereas subargument `thin` is left at its default value, as seen in the initial comment rows:

```
# stan_version_major = 2
# stan_version_minor = 24
# stan_version_patch = 0
# model = bernoulli_model
# method = sample (Default)
#   sample
#     num_samples = 100
#     num_warmup = 200
#     save_warmup = 1
#     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 = 2991989946 (Default)
```

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

Note that when running multi-threaded programs which use `reduce_sum` for high-level parallelization, the number of threads used will also be included in this initial comment header.

Column headers

The CSV header row lists all sampler parameters, model parameters, transformed parameters, and quantities of interest. The sampler parameters are described in detail in the output file section of the Quickstart Guide chapter on MCMC Sampling. The example model `bernoulli.stan` only contains one parameter `theta`, therefore the CSV file data table consists of 7 sampler parameter columns and one column for the model parameter:

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

As a second example, we show the output of the `eight_schools.stan` model on run on example dataset. This model has 3 parameters: `mu`, `theta` a vector whose length is dependent on the input data, here `N = 8`, and `tau`. The initial columns are for the 7 sampler parameters, as before. The column headers for the model parameters are:

```
mu,theta.1,theta.2,theta.3,theta.4,theta.5,theta.6,theta.7,theta.8,tau
```

Data rows containing warmup draws

When run with option `save_warmup=1`, the thinned warmup draws are written to the CSV output file directly after the CSV header line. Since the default option is `save_warmup=0`, this section is usually not present in the output file.

Here we specified `num_warmup=200` and left `thin` at the default value 1, therefore the next 200 lines are data rows containing the sampler and model parameter values for each warmup draw.

```
-6.74827,1,1,1,1,0,6.75348,0.247195
-6.74827,4.1311e-103,14.3855,1,1,0,6.95087,0.247195
-6.74827,1.74545e-21,2.43117,1,1,0,7.67546,0.247195
-6.77655,0.99873,0.239791,2,7,0,6.81982,0.280619
-6.7552,0.999392,0.323158,1,3,0,6.79175,0.26517
```

Comment rows for adaptation

During warmup, the sampler adjusts the stepsize and the metric. At the end warmup, the sampler outputs this information as comments.

```
# Adaptation terminated
# Step size = 0.813694
# Diagonal elements of inverse mass matrix:
# 0.592879
```

As the example bernoulli model only contains a single parameter, and as the default metric is `diag_e`, the inverse mass matrix is a 1×1 matrix, and the length of the diagonal vector is also 1.

In contrast, if we run the eight schools example model with metric `dense_e`, the adaptation comments section lists both the stepsize and the full 10×10 inverse mass matrix:

```
# Adaptation terminated
# Step size = 0.211252
# Elements of inverse mass matrix:
# 25.6389, 17.3379, 13.9455, 15.9036, 15.1953, 8.73729, 16.9486, 14.4231, 17.4969,
# 17.3379, 79.8719, 12.2989, -1.28006, 9.92895, -
3.51622, 10.073, 22.0196, 19.8151, 4.71028
# 13.9455, 12.2989, 36.1572, 12.8734, 11.9446, 9.09582, 9.74519, 10.9539, 12.1204,
# 15.9036, -1.28006, 12.8734, 59.9998, 10.245, 8.03461, 16.9754, 3.13443, 9.68292,
1.36097
# 15.1953, 9.92895, 11.9446, 10.245, 43.548, 15.3403, 13.0537, 7.69818, 10.1093, 0
# 8.73729, -3.51622, 9.09582, 8.03461, 15.3403, 39.981, 12.7695, 1.16248, 6.13749,
2.08507
# 16.9486, 10.073, 9.74519, 16.9754, 13.0537, 12.7695, 45.8884, 11.6074, 8.96413,
1.15946
# 14.4231, 22.0196, 10.9539, 3.13443, 7.69818, 1.16248, 11.6074, 49.4083, 18.9169,
# 17.4969, 19.8151, 12.1204, 9.68292, 10.1093, 6.13749, 8.96413, 18.9169, 68.0228,
# 0.518757, 4.71028, 0.211353, -1.36097, 0.155245, -2.08507, -
1.15946, 3.15661, 1.74104, 1.50433
```

Note that when the sampler is run with arguments `algorithm=fixed_param`, this section will be missing.

Data rows containing sampling draws

The output file contains the values for the thinned set draws during sampling. Here we specified `num_sampling=100` and left `thin` at the default value 1, therefore the next 100 lines are data rows containing the sampler and model parameter values for each sampling iteration.

```
-8.76921,0.796814,0.813694,1,1,0,9.75854,0.535093
-6.79143,0.979604,0.813694,1,3,0,9.13092,0.214431
-6.79451,0.955359,0.813694,2,3,0,7.19149,0.289341
```

Timing information

Upon successful completion, the sampler writes timing information to the output CSV file as a series of final comment lines:

```
#
#  Elapsed Time: 0.005 seconds (Warm-up)
#                0.002 seconds (Sampling)
#                0.007 seconds (Total)
#
```

Diagnostic CSV output file

The diagnostic file contains the following:

- Initial comment rows listing full CmdStan argument configuration.
- Header row
- Data rows containing warmup draws, if run with option `save_warmup=1`
- Sampling draws
- Comment rows giving timing information

The columns in this file contain, in order:

- all sampler parameters
- all model parameter estimates (on the unconstrained scale)
- the latent Hamiltonian for each parameter
- the gradient for each parameters

The labels for the latent Hamiltonian columns are the parameter column label with prefix `p_` and the labels for the gradient columns are the parameter column label with prefix `g_`.

These are the column labels from the file `bernoulli_diagnostic.csv`:

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

Profiling CSV output file

The profiling information is stored in a plain CSV format with no meta information in the comments.

Each row represents timing information collected in a `profile` statement for a given thread. It is possible that some `profile` statements have only one entry (if

they were only executed by one thread) and others have multiple entries (if they were executed by multiple threads).

The columns are as follows:

- `name`, The name of the `profile` statement that is being timed
- `thread_id`, The thread that executed the `profile` statement
- `total_time`, The combined time spent executing statements inside the `profile` which includes calculation with and without automatic differentiation
- `forward_time`, The time spent in the `profile` statement during the forward pass of a reverse mode automatic differentiation calculation or during a calculation without automatic differentiation
- `reverse_time`, The time spent in the `profile` statement during the reverse (backward) pass of reverse mode automatic differentiation
- `chain_stack`, The number of objects allocated on the chaining automatic differentiation stack. There is a function call for each of these objects in the reverse pass
- `no_chain_stack`, The number of objects allocated on the non-chaining automatic differentiation stack
- `autodiff_calls`, The total number of times the `profile` statement was executed with automatic differentiation
- `no_autodiff_calls` - The total number of times the `profile` statement was executed without automatic differentiation

23.3. Optimization output

- Config as comments
- Header row
- Penalized maximum likelihood estimate

23.4. Variational inference output

- Config as comments
- Header row
- Adaptation as comments
- Variational estimate
- Sample draws from estimate of the posterior

23.5. Generate quantities outputs

- Header row
- Quantities of interest

23.6. Diagnose method outputs

- Header row
- Gradients

24. JSON Format for CmdStan

CmdStan can use JSON format for input data for both model data and parameters. Model data is read in by the model constructor. Model parameters are used to initialize the sampler and optimizer.

24.1. Creating JSON files

You can create the JSON file yourself using the guidelines below, but a more convenient way to create a JSON file for use with CmdStan is to use the `write_stan_json()` function provided by the CmdStanR interface.

24.2. JSON syntax summary

JSON is a data interchange notation, defined by an EMCA standard. JSON data files must in Unicode. JSON data is a series of structural tokens, literal tokens, and values:

- Structural tokens are the left and right curly bracket `{}`, left and right square bracket `[]`, the semicolon `:`, and the comma `,`.
- Literal tokens must always be in lowercase. There are three literal tokens: `true`, `false`, `null`.
- A primitive value is a single token which is either a literal, a string, or a number.
- A string consists of zero or more Unicode characters enclosed in double quotes, e.g. `"foo"`. A backslash is used to escape the double quote character as well as the backslash itself. JSON allows the use of Unicode character escapes, e.g. `"\\uHHHH"` where `HHHH` is the Unicode code point in hex.
- Numbers are represented using either decimal notation or scientific notation. The following are examples of numbers: `17`, `17.2`, `-17.2`, `-17.2e8`, `17.2e-8`. There is no distinction between integer and real numbers in the JSON format other than whether they have periods or scientific notation.
- The special floating point values for positive infinity, negative infinity, and not-a-number can be represented in multiple ways. Positive infinity can be represented as the string `"Inf"`, the string `"Infinity"`, or the atom `Infinity`. Negative infinity can be represented as the string `"-Inf"`, the string

`"-Infinity"`, or the atom `-Infinity`. Not-a-number can be represented as the string `"NaN"` or the atom `NaN`. These values may be mixed with other numerical types.

- A complex scalar is represented as a two-element array consisting of its real component followed by its imaginary component. For example, the complex number $2.3 - 1.83i$ would be represented in JSON as the two-element array `[2.3, -1.83]`.
- A JSON array is an ordered, comma-separated list of zero or more JSON values enclosed in square brackets. The elements of an array can be of any type. The following are examples of arrays: `[]`, `[1]`, `[0.2, "-inf", true]`.
- Vectors and row vectors in JSON are representing as arrays of their elements. For example, both the vector $\begin{bmatrix}1 & 2\end{bmatrix}^\top$ and the row vector $\begin{bmatrix}1 & 2\end{bmatrix}$ are represented by the JSON array `[1, 2]`.
- Complex vectors are represented as arrays of two-element arrays. For example, the complex vector $\begin{bmatrix}2.3 - 1.83i & -4.8 + 2i\end{bmatrix}^\top$ is represented as `[[2.3, -1.83], [-4.8, 2]]` in JSON. A complex row vector has the same representation as its transpose (the vector with the same elements).
- Matrices are represented as arrays of their row vectors. For example, the 2×3 matrix
$$\begin{bmatrix}1 & 2.7 & -9.8\\ 4.2 & 1.8 & -7.3\end{bmatrix}$$
is represented in JSON as `[[1, 2.7, -9.8], [4.2, 1.8, -7.3]]`.
- Complex matrices are also represented as arrays of their row vectors. For example, the 2×3 complex matrix
$$\begin{bmatrix}1 + 2i & 3 - 4.2i & 13.1 + 2.7i\\ 3.1 & -5i & 0\end{bmatrix}$$
would be represented in JSON as `[[[1, 2], [3, -4.2], [13.1, 2.7]], [[3.1, 0], [0, -5], [0, 0]]]`.
- Tuples are written as nested JSON objects where the keys are strings for the numbered slots in the tuple. For example, the tuple `(1.5, 3.4)` is represented in JSON as `{"1": 1.5, "2": 3.4}`.
- A name-value pair consists of a string followed by a colon followed by a value, either primitive or compound.

- A JSON object is a comma-separated series of zero or more name-value pairs enclosed in curly brackets. Each name-value pair is a member of the object. Membership is unordered. Member names are not required to be unique. The following are examples of objects: `{ }`, `{"foo": null}`, `{"bar" : 17, "baz" : [14,15,16.6] }`.

24.3. Stan data types in JSON notation

Stan follows the JSON standard. A Stan input file in JSON notation consists of single JSON object which contains zero or more name-value pairs. This structure corresponds to a Python data dictionary object. The following is an example of JSON data for the simple Bernoulli example model:

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

Matrix data and multi-dimensional arrays are indexed in row-major order. For a Stan program which has data block:

```
data {
  int d1;
  int d2;
  int d3;
  array[d1, d2, d3] int ar;
}
```

the following JSON input would be valid:

```
{ "d1" : 2,
  "d2" : 3,
  "d3" : 4,
  "ar" : [[[0,1,2,3], [4,5,6,7], [8,9,10,11]],
          [[12,13,14,15], [16,17,18,19], [20,21,22,23]]]
}
```

JSON ignores whitespace. In the above examples, the spaces and newlines are only used to improve readability and can be omitted.

All data inputs are encoded as name-value pairs. The following table provides more examples of JSON data. The left column contains a Stan data variable declaration and the right column contains valid JSON data inputs.

Stan declaration	JSON encoding
`int i`	`"i": 17`

Stan declaration	JSON encoding

`real a`	`"a" : 17`
	`"a" : 17.2`
	`"a" : "NaN"`
	`"a" : "+inf"`
	`"a" : "-inf"`

`complex z`	`"z": [1, -2.3]`
`array[5] int`	`"a" : [1, 2, 3, 4, 5]`
`array[5] real a`	`"a" : [ 1, 2, 3.3, "NaN", 5 ]`
`array[2] complex b`	`"b" : [[1, -2.3], [4.9, 0]]`
`vector[5] a`	`"a" : [1, 2, 3.3, "NaN", 5]`
`row_vector[5] a`	`"a" : [1, 2, 3.3, "NaN", 5]`
`matrix[2, 3] a`	`"a" : [[ 1, 2, 3 ], [ 4, 5, 6]]`
`complex_vector[2] c`	`"c" : [[-1.2, 3.3], [4.8, 1.9], [2.3, 0]]`
`complex_row_vector[2] c`	`"c" : [[-1.2, 3.3], [4.8, 1.9], [2.3, 0]]`
`complex_matrix[2, 3] d`	`"d" : [[[1, 1], [2, 2], [3, 3]], [4, 4], [5, 5], [6, 6]]]`
`tuple(real, array[2] int) t`	`"t" : { "1": 1.4, "2": [1, 2]}`

Empty arrays in JSON

JSON notation is not able to distinguish between multi-dimensional arrays where any dimension is 0, e.g., a 2-D array with dimensions $(1, 0)$, i.e., an array which contains a single array which is empty, has JSON representation `[ ]`. To see how this works, consider the following Stan program data block:

```
data {
  int d;
  array[d] int ar_1d;
  array[d, d] int ar_2d;
  array[d, d, d] int ar_3d;
}
```

In the case where variable `d` is `1`, all arrays will contain a single value. If array variable `ar_d1` contains value `7`, 2-D array variable `ar_d2` contains (an array which

contains) value `8`, and 3-D array variable `ar_d3` contains (an array which contains an array which contains) value `9`, the JSON representation is:

```
{ "ar_d1" : [7],
  "ar_d2" : [[8]],
  "ar_d3" : [[[9]]]
}
```

However, in the case where variable `d` is `0`, `ar_d1` is empty, i.e., it contains no values, as is `ar_d2`, `ar_d3`, and the JSON representation is

```
{ "d" : 0,
  "ar_d1" : [ ],
  "ar_d2" : [ ],
  "ar_d3" : [ ]
}
```

25. RDump Format for CmdStan

NOTE: Although the RDump format is still supported, I/O with JSON is faster and recommended. See the chapter on JSON for more details.

RDump format can be used to represent values for Stan variables. This format was introduced in SPLUS and is used in R, JAGS, and in BUGS (but with a different ordering).

A dump file is structured as a sequence of variable definitions. Each variable is defined in terms of its dimensionality and its values. There are three kinds of variable declarations: - scalars - sequences - general arrays

25.1. Creating dump files

Dump files can be created from R using RStan, via the `rstan` package function `stan_rdump`. Stan RDump files must be created via `stan_rdump` and not by R's native `dump` function because R's `dump` function uses a richer syntax than is supported by the underlying Stan i/o libraries.

25.2. Scalar variables

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

```
y <- 17.2
```

25.3. Sequence variables

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

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

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

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

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

The sequence denoted by `1:3` is of length 3, running from 1 to 3 inclusive. The colon notation allows sequences going from high to low. The following are equivalent:

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

As a special case, a sequence of zeros can also be represented in the dump format by `integer(x)` and `double(x)`, for type int and double, respectively. Here `x` is a non-negative integer to specify the length. If `x` is 0, it can be omitted. The following are some examples.

```
x1 <- integer()
x2 <- integer(0)
x3 <- integer(2)
y1 <- double()
y2 <- double(0)
y3 <- double(2)
```

25.4. Array variables

For more than one dimension, the dump format uses a dimensionality specification. For example, the following defines a 2×3 array:

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

Data is stored column-major, thus the values for `y` will be:

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

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

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

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

Sequence notation can be used within any call to the generic `c()` function in R. In the above example, c(2,3) could be written as `c(2:3)`.

The generalization of column-major indexing is last-index major indexing. Arrays of more than two dimensions are written in a last-index major form. For example,

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

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

```
z[1, 1, 1] = 1
z[2, 1, 1] = 2
z[1, 2, 1] = 3
z[2, 2, 1] = 4
z[1, 3, 1] = 5
z[2, 3, 1] = 6
z[1, 1, 2] = 7
z[2, 1, 2] = 8
z[1, 2, 2] = 9
z[2, 2, 2] = 10
z[1, 3, 2] = 11
z[2, 3, 2] = 12
z[1, 1, 3] = 13
z[2, 1, 3] = 14
z[1, 2, 3] = 15
z[2, 2, 3] = 16
z[1, 3, 3] = 17
z[2, 3, 3] = 18
z[1, 1, 4] = 19
z[2, 1, 4] = 20
z[1, 2, 4] = 21
z[2, 2, 4] = 22
z[1, 3, 4] = 23
z[2, 3, 4] = 24
```

If the underlying 3-D array is stored as a 1-D array in last-index major format, the innermost array elements will be contiguous.

The sequence of values inside `structure` can also be `integer(x)` or `double(x)`. In particular, if one or more dimensions is zero, `integer()` can be put inside `structure`. For instance, the following example is supported by the dump format.

```
y <- structure(integer(), .Dim = c(2,0))
```

25.5. Matrix- and vector-valued variables

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

Vector dump format

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

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

Matrix dump format

The following declarations have the same dump format.

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

Arrays of vectors and matrices

The key to understanding arrays is that the array indexing comes before any of the container indexing. That is, an array of vectors is just that: each array element is a vector. See the chapter on array and matrix types in the user's guide section of the language manual for more information.

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

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

Similarly, the following also have the same dump format.

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

25.6. Complex-valued variables

At this time, there is no support for complex number input through the R dump format. As an alternative, the JSON input format supports complex numbers.

25.7. Integer- and real-valued variables

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

For a single value, if there is no decimal point, it may be assigned to an `int` or

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

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

```
y <- 2
```

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

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

```
y <- 2L
```

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

```
y <- 2.0
```

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

Scientific notation

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

Infinite and not-a-number values

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

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

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

25.8. Quoted variable names

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

```
"y" <- c(1,2,3) \end{Verbatim}
```

25.9. Line breaks

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

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

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

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

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

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

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

25.10. BNF grammar for dump data

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

```
definition ::= name <- value optional_semicolon

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

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

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

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

zero_array<int> ::= "integer" '(' <non-negative int>? ')'

zero_array<real> ::= "double" '(' <non-negative int>? ')'

seq<real> ::= cseq<real>

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

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

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

26. 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-undefined`
- `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 a compatible signature. This function can appear in the global namespace or in the model namespace, which is defined as the name of the model (either the file name, or the `--name` argument to stanc) followed by `_namespace`.

This is an advanced feature which is only recommended to users familiar with the internals of Stan's Math library. Most existing C++ code will need to be modified to work with Stan, to varying degrees.

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

```
functions {
  real make_odds(data real theta);
}
data {
  int<lower=0> N;
  array[N] int<lower=0, upper=1> y;
}
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

```
#include <ostream>

double make_odds(const double& theta, std::ostream *pstream__) {
  return theta / (1 - theta);
}
```

The signature for this function needs to fulfill all the usages in the C++ class emitted by stanc. The `pstream__` argument is mandatory in the signature but need not be used if your function does not print any output. Because `make_odds` was declared with a `data` argument and only used in `generated quantites`, a signature which accepts and returns `double` is acceptable. Functions which will have parameters passed as input in the `transformed parameters` or `model` blocks will require the ability to accept Stan's autodiff types. If you wish to autodiff through this function, the simplest option is to make it a template, like

```
template <typename T>
T make_odds(const T &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.hpp e
```

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 add `#include <stan/model/model_header.hpp>` and prefix the function calls with `stan::math::`.

26.1. Derivative specializations

External C++ functions are currently the only way to encode a function with a known analytic gradient outside the Stan Math Library. This is done very similarly to how a function would be added to the Math library with a reverse-mode

specialization. The following code is adapted from the Stan Math documentation.

Suppose you have the following (nonsensical) model which relies on a function called `my_dot_self`. We will implement this as a copy of the built-in `dot_self` function.

```
functions {
  // both overloads end up using the same C++ template
  real my_dot_self(vector theta);
  real my_dot_self(row_vector theta);
}
data {
  int<lower=0> N;
  vector[N] input_data;
}
transformed data {
  // no autodiff for data - will call using doubles
  real ds = my_dot_self(input_data);
}
parameters {
  row_vector[N] thetas;
}
model {
  thetas ~ normal(0,1);
  // autodiff - will call using stan::math::var types
  input_data ~ normal(thetas, my_dot_self(thetas));
}
```

If you wanted to autodiff through this function, the following header would suffice[1]:

```
#include <stan/model/model_header.hpp>
#include <ostream>

template <typename EigVec, stan::require_eigen_vector_t<EigVec> * = nullpt
inline stan::value_type_t<EigVec> my_dot_self(const EigVec &x, std::ostrea
{
    const auto &x_ref = stan::math::to_ref(x);
    stan::value_type_t<EigVec> sum_x = 0.0;
    for (int i = 0; i < x.size(); ++i)
```

[1] Details of programming in the Stan Math style are omitted from this section, it is presented only as an example

```
    {
        sum_x += x_ref.coeff(i) * x_ref.coeff(i);
    }
    return sum_x;
}
```

However, we know the derivative of this function directly. To leverage this, we could use a more complicated form which has two function templates that differentiate themselves based on whether or not derivatives are required:

```
#include <stan/model/model_header.hpp>
#include <ostream>

template <typename EigVec, stan::require_eigen_vector_t<EigVec> * = nullpt
          stan::require_not_st_var<EigVec> * = nullptr>
inline double my_dot_self(const EigVec &x, std::ostream *pstream__)
{
    auto x_ref = stan::math::to_ref(x);
    double sum = 0.0;
    for (int i = 0; i < x.size(); ++i)
    {
        sum += x_ref.coeff(i) * x_ref.coeff(i);
    }
    return sum;
}

template <typename EigVec, stan::require_eigen_vt<stan::is_var, EigVec> *
inline stan::math::var my_dot_self(const EigVec &v, std::ostream *pstream_
{
    // (1) put v into our memory arena
    stan::arena_t<EigVec> arena_v(v);
    // (2) calculate forward pass using
    // (3) the .val() method for matrices of var types
    stan::math::var res = my_dot_self(arena_v.val(), pstream__);
    // (4) Place a callback for the reverse pass on the callback stack.
    stan::math::reverse_pass_callback(
        [res, arena_v]() mutable
        { arena_v.adj() += 2.0 * res.adj() * arena_v.val(); });
    return res;
}
```

For more details about how to write C++ code using the Stan Math Library, see the Math library documentation at https://mc-stan.org/math/ or the paper at https://arxiv.org/abs/1509.07164.

26.2. Special functions: RNGs, distributions, editing `target`

Some functions have special meanings in Stan and place additional requirements on their signatures if used in external C++.

- RNGs must end with `_rng`. They will be passed a "base RNG object" as the second to last argument, before the pointer to the ostream. We recommend making this a template, since it may change, but at the moment it is always a `boost::random::ecuyer1988` object.
- Functions which edit the `target` directly must end with `_lp` and will be passed a reference to `lp__` and a reference to a `stan::math::accumulator` object as the final parameters before the ostream pointer. They are also expected to have a boolean template parameter `propto__` which controls whether or not constant terms can be dropped.
- Probability distributions must end with `_lpdf` or `_lpmf` and will be passed a boolean template parameter `propto__` which controls whether or not constant terms can be dropped.

References

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

Kucukelbir, Alp, Dustin Tran, Rajesh Ranganath, Andrew Gelman, and David M Blei. 2017. "Automatic Differentiation Variational Inference." *Journal of Machine Learning Research*.

Zhang, Lu, Bob Carpenter, Andrew Gelman, and Aki Vehtari. 2022. "Pathfinder: Parallel Quasi-Newton Variational Inference." *Journal of Machine Learning Research* 23 (306): 1–49. http://jmlr.org/papers/v23/21-0889.html.