External C++ Functionality in CmdStan

May Amato-Yarbrough, mba75@nau.edu

January 2023

Contents

Introduction

The aim of this walk-through is to provide **CmdStan** users with the knowledge needed to incorporate external C++ code into their existing **Stan** models. The primary advantage of doing so is the ability to create and use custom C++ functions within your Stan model.

The first example of this functionality will use the Bernoulli example code provided by the Stan developers. The code for this example is found in the "/cmdstan/examples/bernoulli" directory of the **CmdStan installation**. The second example will be a binomial model using data from **Linear Models with R**. The code for this section will be provided below.

There are other Stan interfaces that offer external C++ functionality. A notable example is **RStan**. However, the challenge of using external C++ code is relatively similar in both CmdStan and RStan. The key factors to consider when choosing a Stan interface in this case include:

- Your comfort level with each of the interfaces.
- Data input and hyperparameter setup are more straightforward in RStan.
- Power users may find CmdStan more convenient since data input and hyperparameter configuration are done through the terminal.
- CmdStan is typically more up-to-date with respect to the Stan back-end.

With any Stan interface, the user will be required to **template** their C++ code to match the templating found in **Stan's math library**. Further details regarding the process of templating C++ code to match Stan's math library will be provided below. This requirement constitutes the most time-intensive aspect of this process.

For additional information on using external C++ code within RStan, you can refer to this **vinette**. The use of external C++ code within RStan will not be explored in this walkthrough.

Installation

The first step is download the version of CmdStan appropriate for your system. The available versions of CmdStan are listed under releases in the **stan-dev/cmdstan** repository. For example: The release that was used for this walk-through is **cmdstan-2.29.2.tar.gz**.

Once the tar.gz is downloaded and unzipped into a directory of your choosing (the system's home directory was used for this example), you can open a terminal window and cd into the unzipped folder (e.g. `cd ~/cmdstan-2.29.2/`). Inside your terminal window, run `make build` to complete the CmdStan installation process. If you wanted to delete CmdStan from your system, you can run `make clean-all` from the same directory as you called `make build`.

More recent versions of CmdStan can be used; however, the external C++ functionality may differ in future releases of CmdStan.

The details of the system used for this walkthrough:

- Operating System: Ubuntu 20.04 amd64
- CmdStan Version: v2.29.2
- Compiler/Toolkit: g++ 9.4.0 and make 4.2.1

Repository

For detailed terminal output and what you should expect to see after running the `make`, `sample`, and `stansummary` commands, refer to the "terminal_output" directories within the "bernoulli" and "binomial_glm" directories. These directories can be found in the same github repository associated with this documentation: "/external_cpp_functionality_in_cmdstan". This repository also contains all of the code referred to within this walkthrough, as well as examples of CmdStan makefiles that can be used for each set of code.

Bernoulli example

This section will use the Bernoulli source code provided by the Stan developers. The associated Stan documentation that expands on the Bernoulli source code using external C++ code can be found **here**. A condensed version of this documentation will be found below, but it is recommended that you read the Stan documentation if you want additional detail.

First step: Within the Bernoulli directory of your CmdStan installation, open up the file titled "bernoulli.stan". If you installed CmdStan into your home directory, this file would be found in "~/cmdstan-2.29.2/examples/bernouli".

Above the `data` block, add the following lines of code:

```
functions {
  real make_odds(real theta);
}
```

Below the `model` block, add the additional lines of code:

```
generated quantities {
  real odds;
  odds = make_odds(theta);
}
```

This is the essential first step to editing your Stan code so that it can use external C++ code. You are adding a function called `make_odds` that accepts a `real` data type (`theta`) and outputs a `real` data type. Additional information on data types within Stan can be found **here**. The `make_odds` function will be the function defined within your external C++ file.

Within the same Bernouli directory, create a new file titled "make_odds.hpp". In "make_odds.hpp", add the following lines of code:

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

namespace bernoulli_model_namespace {
template <typename T0__,
          stan::require_all_t<stan::is_stan_scalar<T0__>>* = nullptr>
  stan::promote_args_t<T0__>
  make_odds(const T0__& theta, std::ostream* pstream__) {
    return theta / (1 - theta);
  }
}
```

For now, save and close "make_odds.hpp". A similar file will be revisited later with additional details.

Open up a terminal window within the CmdStan directory ("~/cmdstan-2.29.2/") and execute the following line:

```
make STANCFLAGS=--allow-undefined USER_HEADER=examples/bernoulli/make_odds.hpp \
  examples/bernoulli/bernoulli_makeodds
```

In this `make` call, both the `STANCFLAGS=--allow-undefined` and `USER_HEADER=` flags are necessary for allowing undefined functions within your Stan code and providing a location to an external header file. As noted by the Stan documentation: "you could put STANCFLAGS and USER_HEADER into the make/local file instead of specifying them on the command-line." This provides users with the option of using a simplified `make` call:

```
make examples/bernoulli/bernoulli_makeodds
```

Assuming that your `make` call was successful, you should find a file titled "bernoulli_makeodds" within your Bernoulli directory. This is the compiled executable that is a a result of your previous `make` call. It will contain the original "bernoulli.stan" model code, as well as the custom external C++ code.

Note: Refer to "external_cpp_functionality_in_stan/bernoulli/terminal_output/" on Github for an example of what a successful `make` call would look like.

Once you have the compiled executable the remaining CmdStan steps are the same.

The Stan developers provide two data files for this example: "bernoulli.data.json" and "bernoulli.data.R". Below are the steps one would follow to sampling using the JSON data file.

Sample using the provided example data: `./bernoulli_makeodds sample data file=bernoulli.data.json`

Display using the `stansummary` functionality: `~/cmdstan-2.29.2/bin/stansummary output.csv`

These steps are discussed in detail in this **part of the CmdStan documentation**. Additional arguments will be used in the next section.

Beyond the Bernoulli example

This section will provide a more complicated example of using external C++ code within CmdStan.

Within the "examples" directory of your CmdStan installation, you will need to create a new directory as well as several new files. These files will be similar to the files from the previous section. However, the contents of these files will be different. Navigate to "~/cmdstan-2.29.2/examples/" and create a new directory titled "binomial_glm". Afterwards, open the newly created directory and create the three following files:

1. "binomial_glm.stan"
2. "external_binomial_glm.hpp"
3. "bin_glm.Data.list.dump"

Note: Depending on your preference, it could be easier to use the **touch** command within a terminal window to create these files.

Binomial GLM stan model

Open up "binomial_glm.stan" with a text editor and paste the following contents into the file and then save the file.

```
functions {
    real custom_inv_logit(real value);
}

data {
    int<lower=1> N;                         // #obs
    int<lower=1> p;                         // #params
    int<lower=1> n[N];                      // #trials
    int<lower=0> y[N];                      // #successes
    matrix<lower=0, upper=1> [N,p] X;       // X matrix
    real bkprior[2];                        // bk (k>0) prior mean and sd
}

parameters {
    vector[p] b;
}

model {
    vector[N] output;

    output = X * b;
    for (i in 1:N) {
        // [y | beta]
        y[i] ~ binomial(n[i], custom_inv_logit(output[i]));
    }

    // [beta] = [b0] * prod_k [bk] (k>1)
    b[2:11] ~ normal(bkprior[1], bkprior[2]);

    // [b0 flat improper implied]
}
```

Explanation: The data and parameters block is fairly common for Stan code. The interesting points within the above code are the functions and model blocks.

The functions block: `custom_inv_logit` is the name of the function found in the external C++ file. Similar to the Bernoulii example, the output of the function call is a real number, and the input is a real number called "value".

The model block: The matrix and vector multiplication is done (`output = X * b`) within the Stan code, rather than within the external C++ code. While this adds additional lines of Stan code, it's to avoid the hassle of matrix and vector multiplication within the external C++ code. Refer to the "Matrices and Vectors" subsection below for more information.

External binomial hpp code

Now open up "external_binomial_glm.hpp" and paste the following code. Afterwards, save the file.

```
#include "external_binomial_glm.hpp"
#include "stan/math/prim/fun/exp.hpp"

namespace binomial_glm_model_namespace {
template <typename T0__, stan::require_stan_scalar_t<T0__>* = nullptr>
stan::promote_args_t<T0__>
custom_inv_logit(const T0__& value, std::ostream* pstream__)
{
    stan::math::exp_fun exponent_function;

    if (value > 0) {
        return 1.0 / (1.0 + exponent_function.fun(-value));
    }
    else {
        T0__ e = exponent_function.fun(value);
        return e / (1.0 + e);
    }
}
}
```

Explanation: There are a handful of points worth mentioning here.

Inverse logit: This was modeled after Stan's **inv logit struct** found within the Stan library. While this function could be directly used within your Stan model, it was used as the external function within external_binomial_glm.hpp for demonstrative purposes.

Exponential function: The exponential function comes from Stan's **exp fun struct**. It's important to note that you cannot use the exponential function from the C++ standard library, as it is not designed to accept Stan math types. As such, the best option is to use Stan's `exp_fun` struct. Once the struct option is created (called `exponent_function` above), you can access its exponential function to pass in your `value` parameter.

Creating a new variable: Note that the variable `e` is of type `T0__`; the same type as the input parameter `value`. This is necessary to store the results of the exponential function.

Initial template: When attempting to compile external C++ code with CmdStan, it will create a separate header file from yours that contains the templated header for your external file. This will happen whether compilation is successful or fails.

- For example: When trying to execute a `make` command with your "binomial_glm.stan" file completed but not your external C++ file, you will encounter a compilation error. However, in the same directory as your ".stan" file, you
  should find a file named "binomial_glm.hpp". Open up that file, and you should see the appropriate template for your external C++ file towards the top portion of "binomial_glm.hpp". You will need to include that template (as well as the namespace calls) in your header file.

External binomial RDump file

Now that both the Stan model and external C++ code are ready, all that is left is to run the sampler. However, this will require a data file.

Open up "bin_glm.Data.list.dump", paste the following lines of code, save and then exit.

```
N <- 24
p <- 11
n <-
c(56, 34, 8, 82, 50, 18, 16, 39, 21, 31, 52, 40, 87, 56,
20, 35, 23, 9, 45, 72, 49, 28, 36, 24)
y <-
c(9, 19, 7, 5, 6, 12, 5, 21, 19, 7, 14, 25, 58, 45, 20, 11,
7, 7, 32, 61, 42, 12, 15, 17)
X <-
structure(c(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0,
0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1,
0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0,
1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0,
1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0,
0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0,
0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0,
1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0,
1, 0, 0, 1),
.Dim = c(24, 11))
bkprior <-
c(1.77635683940025e-15, 4.69923960063867)
```

Explanation: This data set was compiled from Julian Faraway's data on head injuries. In order to get the data into a readable format for Stan (".RDump"), Dr. Barber used the `rstan::stan_rdump` function within R.

For more information on creating RDump data files for use in Cmdstan, refer to: **RDump Format for CmdStan**.

Although JSON files were not used in this walkthrough, they should work similarly to RDump files. However, the contents might be structured differently, and the file extensions will be different when entering the data via the command line. For more information on creating JSON data files for use in Cmdstan, refer to: **JSON Format for CmdStan**.

Final steps

Now that all of the necessary files have been provided, the process remains relatively the same as before: `make`, `sample`, `stansummary`.

To make the executable, open up a terminal window within the CmdStan directory ("~/cmdstan-2.29.2/") and execute the following line:

```
make STANCFLAGS=--allow-undefined USER_HEADER=examples/binomial_glm/external_binomial_glm.hpp \
  examples/binomial_glm/binomial_glm
```

If compilation was successful, you should find a file titled "binomial_glm" within your "/binomial_glm" directory.

Sample using the provided example data: `./binomial_glm sample data file=binGLM.Data.list.dump.R`

For a more interesting sampling routine, additional arguments can be provided to CmdStan and used within a for loop to produce multiple chains:

```
for i in {1..3}; do ./binomial_glm sample random seed=110710 \
  data file=binGLM.Data.list.dump.R output refresh=1000 file=output_${i}.csv; done;
```

Display using the `stansummary` functionality: `~/cmdstan-2.29.2/bin/stansummary output_*.csv`

Additional points

Adding more inputs

Going off of the "Beyond the Bernoulli example" section above, you will
likely need to add more than one input to your external C++ code to get the most out of it. Assuming that the data type that you want to add is covered by Stan's data types **here**, you should be able to adapt the previous .stan and .cpp code chunks to include more inputs. To produce the templated code for these updated functions, you should follow the final bullet in the "External binomial hpp code" subsection.

Note: This is slightly more difficult in the case of matrices and vectors, and these types are covered in the "Matrices and Vectors" subsection below. Constrained data types were also not used in this walk-through.

Including more files

Depending on the complexity of your external C++ code, it's likely that you will need to include more than one C++ file in your CmdStan compilation. Fortunately, the external C++ file that is supplied to CmdStan doesn't have to be a header file (".hpp" extension); it can also be a source code file (".cpp"). This means your external C++ file can follow this fairly common file structure:

- Have a "main.cpp" file that is called during CmdStan compilation.
- Within that "main.cpp" file, you can include the headers for additional ".hpp" files.
- Within those ".hpp" files, you can include their associated ".cpp" files.

In this way, when CmdStan attempts to compile, the compiler will first look for the "main.cpp" file, then the ".hpp" files that "main.cpp" file points to, and then the ".cpp" files that are pointed to by the ".hpp" files.

For example, instead of using the following `make` call:

```
make STANCFLAGS=--allow-undefined USER_HEADER=examples/binomial_glm/external_binomial_glm.hpp \
  examples/binomial_glm/binomial_glm
```

... a user can use the following `make` call instead:

```
make STANCFLAGS=--allow-undefined USER_HEADER=examples/binomial_glm/external_binomial_glm.cpp \
  examples/binomial_glm/binomial_glm
```

Within "external_binomial_glm.cpp" you could have an include statement at the top of the file that would link an additional header file. Like so:

```
#include "external_binomial_glm.hpp"
```

For more information on source file inclusion, refer to this **reference**.

Matrices and Vectors

When it comes to matrix and vector multiplication, it becomes more challenging due to the fact that Stan uses the **Eigen library** to represent matrices and the **standard C++ library** to represent vectors. Therefore, your external C++ code will also need to use these libraries.

The difficulty arises with simple operations such as matrix multiplication, which is handled within the Eigen library. The Eigen functions would expect certain data types within the matrices. However, since these Eigen matrices contain Stan data types, you cannot use the functions from the Eigen library to perform these simple operations, as compilation will fail. The same applies to C++ vectors.

The current workaround for this issue involves performing the matrix and vector operations within the Stan model code. Then, the results are stored into a vector within the Stan model code. These values can be iterated over afterwards, passed into an external function, and the result are returned. This approach was used for the binomial GLM example.

The Stan developers most likely have a solution to this issue, which may come in the form of Stan math functions that take the place of Eigen/C++ functions. It would be beneficial to reach out to them via the Stan discourse forum to address and resolve this issue.

Limitations

The "limit" to CmdStan's external C++ functionality is whether the custom function can be auto-differentiated by Stan. If not, then derivatives would need to be computed within the custom function.

Aside from this unique limitation, the typical Stan and C++ limitations still exist. If your Stan model is not syntactically correct, then compilation will fail. If your C++ code is not syntactically correct, then compilation will fail.

It's for these reasons that it might be easier to initially write code for the Stan model and C++ code separate from each other. Once both the Stan model and C++ code have been tested, the process of templating the C++ code can be begin.

Aknowledgements

Dr. Jay Barber, the faculty member that oversaw this section of graduate research.

The Stan team for their **extensive documentation** and help via the **Stan discourse forums**.

Julian Faraway since the head injury data that used was sourced from their book, Linear Models with R.