Interact with external C++ and implement distributions with analytic derivatives for Stan.

Zhi Ling

January 18, 2024

1 Introduction

This case study aims to provide a mature example of implementing analytic derivatives using Stan's external C++ interface. This avoids automatic differentiation to significantly speed up computation. And with the help of template functions, a level of abstraction not provided in Stan is achieved. We will foused on give the process of implementing a probability distribution in Stan, from mathematical details to technical details. This article is generally suitable for statisticians with some C++ experience. Developers familiar with the structure of the Stan Math Library have already done a lot of work in this regard (hats off to the Stan development team), but these efforts are less documented.

During this process, we will intersperse explanations about Stan's gradient interface. The everse automatic differentiation in Stan completely eliminates the implementation burden of derivatives. However, as noted in the Stan's developer wiki, Stan has a number of probability functions that have not implement derivatives. These places can also become performance bottlenecks for potential programs. As long as people with strong mathematical background actively join the development of Stan, the Stan Math Library can make a lot of improvements in this regard.

2 Interacting with external C++

External C++ functions are currently the only way to encode a function with a known analytic gradient outside the Stan Math Library. So let's first introduce how to interact with external C++ code.

Examples of how to use basic external C++ code in Stan can be found in many places. Some official documentation are as follows:

Using external C++ code
Interfacing with External C++ Code
External C++ (experimental)
Using the Stan Math C++ Library

The existing materials exist in scattered documents. Here we provide working examples and point out some noteworthy aspects in practice.

Consider the following Stan model, based on the bernoulli example in the CmdStan documentation. Assume that there are the following Stan code and C++ header files in the same directory.

`bernoulli_example.stan`

```
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);
}
```

external.hpp

```
#include <iostream>
namespace bernoulli_example_model_namespace {
  double make_odds(const double& theta, std::ostream *pstream__) {
    return theta / (1 - theta);
  }
}
```

To use `external.hpp` in Stan, basically we need to achieve the followings:

1. In the Stan model, expose function declarations in the `functions` block.
2. When compile, add `--allow-undefined` to `STANCFLAGS` and specify where the header files are through the `user_header` option.

Below are the code to drive the above model from different interfaces.

2.1 cmdstanpy

```
from cmdstanpy import CmdStanModel
model = CmdStanModel(stan_file='bernoulli_example.stan', compile=False)
model.compile(user_header='external.hpp')
fit = model.sample(data={'N':10, 'y':[0,1,0,0,0,0,0,0,0,1]})
fit.stan_variable('odds')
```

The code is basically from this post. It's just in the lastest version of `cmdstanpy`, we don't have to explicitly add the `--allow-undefined` flag, only specify the `user_header` when compile and `cmdstanpy` will automatically do this.

2.2 cmdstanr

```
library(cmdstanr)
model <- cmdstan_model('bernoulli_example.stan',
    include_paths=getwd(),
    cpp_options=list(USER_HEADER='external.hpp'),
    stanc_options = list("allow-undefined")
)
```

```
fit <- model$sample(data=list(N = 10, y = c(0,1,0,0,0,0,0,0,0,1)))
fit$draws('odds')
```

2.3 pystan

This feature is no longer supported after `pystan` is upgraded to 3.0.

2.4 rstan

For `rstan`, c++ functions do not need to be contained in any specific namespace.

`external_rstan.hpp`

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

`R` code

```
library(rstan)
model <- stan_model('bernoulli_example.stan',
    allow_undefined = TRUE,
    includes = paste0('\n#include "', file.path(getwd(), 'external_rstan.hpp'), '"\n'),
)
fit <- sampling(model, data = list(N = 10, y = c(0,1,0,0,0,0,0,0,0,1)))
extract(fit)$odds
```

2.5 Troubleshooting

If there are errors during compilation or running, try troubleshooting the following issues:

1. If the Stan source code file name is `bernoulli_example.stan`, then the namespace name in all the header files (if there is more than one function to include and they are in different header files) must be `bernoulli_example_model_namespace`. This situation arises if and only if you are using `cmdstan`.
2. If the function prints something to the screen, we must add the additional argument `std::ostream *pstream__` to the function declaration.
3. All function signature in Stan `functions` block must match the function declarations in the header files.

Usually, in a project we will have many a number of model files that all depend on the same (or some) C++ header files. To deal with the first point, one need to manually change the namespace of the C++ code every time compile each of the model, which is tedious and error-prone. The following code can help solve this problem.

In `Python`

```
import re
def change_namespace(stan_name, user_header):
    """Change the namespace name in the user header file.

    Args:
```

```
        stan_name (str): desired Stan file name
        user_header (str): path to user header file
    """
    with open(user_header, 'r') as file:
        content = file.read()
    new_content = re.sub(r'^namespace \S+_model_namespace {',
                         f'namespace {stan_name}_model_namespace {{',
                         content, flags=re.MULTILINE)
    with open(user_header, 'w') as file:
        file.write(new_content)
```

In R

```
#' Change the namespace in the user header file
#'
#' @param stan_name The name of the stan file
#' @param user_header The path to the user header file
#' @return NULL
change_namespace <- function(stan_name, user_header) {
    content <- readLines(user_header, warn = FALSE)
    content <- paste(content, collapse="\n")
    pattern <- "\nnamespace \\S+_model_namespace {"
    replacement <- paste0("\nnamespace ", stan_name, "_model_namespace {")
    new_content <- gsub(pattern, replacement, content, perl = TRUE)
    writeLines(new_content, user_header)
}
```

Now that we know how to call external C++ code from Stan. However, if we simply "translate" what is possible in Stan to C++, there will be almost no difference between them when the compiler translates it. The purpose of using C++ is to achieve functionalities not yet available in Stan, to utilize its vast third-party libraries, and to exploit new language features. Here are some common feature requests

1. C++ has a more mature ecosystem, providing stronger availability for some niche mathematical functions not yet well-supported in Stan. It's unrealistic to expect these functions to be fully integrated into Stan, as it would burden developers.
2. C++ offers advanced abstractions, whereas Stan is more limited linguistically. For example, template function enables the use of the same algorithm on different data types, significantly reducing development time, especially in a production environment.
3. Stan doesn't offer gradient interfaces yet. If one has an analytical gradient and wishes to avoid the overhead of automatic differentiation, external C++ is the only option.

Let's start with an example to see, specifically, how the above three issues arise, are analyzed, and then resolved.

3 Case study

The beta negative binomial distribution is a generalization of negative binomial distribution. It is a compound distribution of negative binomial distribution and beta distribution. Assume

$$\begin{aligned} Y &\sim \mathrm{NB}(r, p) \\ p &\sim \mathrm{Beta}(\alpha, \beta), \end{aligned}$$

where we treat the probability of failure p as a random variable with a beta distribution with parameters α and β. Then the marginal distribution of Y is given by

$$
\begin{aligned}
f(y \mid r, \alpha, \beta) &= \int_0^1 f_{Y|p}(y \mid r, p) \cdot f_p(p \mid \alpha, \beta) \mathrm{d}p \\
&= \int_0^1 \binom{y+r-1}{y} (1-p)^y p^r \cdot \frac{p^{\alpha-1}(1-p)^{\beta-1}}{\mathrm{B}(\alpha, \beta)} \mathrm{d}p \\
&= \frac{\mathrm{B}(r+y, \alpha+\beta)}{\mathrm{B}(r, \alpha)} \frac{\Gamma(y+\beta)}{y!\Gamma(\beta)}.
\end{aligned} \tag{1}
$$

3.1 Implement directly in Stan

The main advantage of using external C++ files is the flexibility to do things that cannot be done directly in the Stan language. But writing a distribution is something Stan can do.

Suppose we have `N` data points and scalar parameters `r, a,` and `b`.

```
data {
  array[N] int<lower=0> y;
}
parameters {
  real<lower=0> r;
  real<lower=0> a;
  real<lower=0> b;
}
```

This distribution can be coded directly in Stan

```
real beta_neg_binomial_lpmf(int y, real r, real a, real b) {
  real lprobs = lgamma(y+1/b) + lbeta(y+r*b/a, 1/a+1/b+1)
              - lgamma(y+1) - lgamma(1/b) - lbeta(r*b/a, 1/a+1);
  return lprobs;
}
...
for (i in 1:N) {
  target += beta_neg_binomial_lpmf(y[i], r, a, b);
}
```

Based on Stan user's guide, the following good practices can speed up running time

```
real beta_neg_binomial_lpmf(array[] int y, real r, real a, real b) {
  int N = size(y);
  vector[N] lprobs;
  for (i in 1:N) {
    lprobs[i] = lgamma(y[i]+1/b) + lbeta(y[i]+r*b/a, 1/a+1/b+1) - lgamma(y[i]+1) - lgamma(1/b) -
        lbeta(r*b/a, 1/a+1);
  }
  return sum(lprobs);
}
...
target += beta_neg_binomial_lpmf(y, r, a, b);
```

If any one of the parameters `r, a,` and `b` can be a vector, we need to repeatedly implement the function so that it has the following signatures

```
real beta_neg_binomial_lpmf(array[] int y, real r, real a, real b)
real beta_neg_binomial_lpmf(array[] int y, vector r, real a, real b)
real beta_neg_binomial_lpmf(array[] int y, real r, vector a, vector b)
real beta_neg_binomial_lpmf(array[] int y, vector r, vector a, real b)
real beta_neg_binomial_lpmf(array[] int y, vector r, real a, vector b)
real beta_neg_binomial_lpmf(array[] int y, real r, vector a, vector b)
real beta_neg_binomial_lpmf(array[] int y, vector r, vector a, vector b)
```

However, this does not cover all cases and errors are likely to occur during the coding process. This is the second issue raised at the end of the previous section.

External C++ allows writing once but automatically adapting to all data structures when faced with a new distribution. We'll see how it works.

3.2 Calculate derivatives

To fully implement a distribution in Stan, we would most like mathematically work out certain derivatives and include them too. Generally speaking, suppose we have a desire distribution $f(y)$, the pdf/pmf, cdf, ccdf of which are denoted by $f(y,\boldsymbol{\theta}), F(y,\boldsymbol{\theta}), C(y,\boldsymbol{\theta})$, respectively, where $\boldsymbol{\theta}$ is the parameter vector. We aim to calculate the derivatives with respect to distribution parameters, after taking the logarithm:

$$\nabla_{\boldsymbol{\theta}} \log f(y,\boldsymbol{\theta}), \nabla_{\boldsymbol{\theta}} \log F(y,\boldsymbol{\theta}), \nabla_{\boldsymbol{\theta}} \log C(y,\boldsymbol{\theta}) \tag{2}$$

where $\nabla_{\boldsymbol{\theta}} \overset{\text{def}}{=} \left[\frac{\partial}{\partial\theta_1}, \frac{\partial}{\partial\theta_2}, \cdots, \frac{\partial}{\partial\theta_n}\right] = \frac{\partial}{\partial\boldsymbol{\theta}}$.

Next, we take BNB distribution as an example to show the calculation process. In which case $\boldsymbol{\theta} = (r, \alpha, \beta)$.

Firstly, we give the conclusions about the first derivatives of the gamma and beta functions. Let $\psi(z)$ denotes the digamma function [Olver, 2010, Ch. 5],

$$\frac{\mathrm{d}}{\mathrm{d}z} \log\Gamma(z) = \frac{\Gamma'(z)}{\Gamma(z)} = \psi(z). \tag{3}$$

The derivative of the logarithmic beta function is

$$\frac{\partial \log B(\alpha,\beta)}{\partial\alpha} = \frac{\partial}{\partial\alpha}\left[\log\Gamma(\alpha) + \log\Gamma(\beta) - \log\Gamma(\alpha+\beta)\right] = \psi(\alpha) - \psi(\alpha+\beta) \tag{4}$$

Similarly,

$$\frac{\partial \log B(\alpha,\beta)}{\partial\beta} = \psi(\beta) - \psi(\alpha+\beta), \tag{5}$$

Derivatives of logarithmic pmf

The BNB logarithmic pmf can be expressed as combination of log gamma and log beta functions:

$$\begin{aligned}\log f(y;r,\alpha,\beta) &= \log\left[\frac{B(r+y,\alpha+\beta)}{B(r,\alpha)}\frac{\Gamma(y+\beta)}{y!\,\Gamma(\beta)}\right] \\ &= \log B(r+y,\alpha+\beta) + \log\Gamma(y+\beta) \\ &\quad - \log B(r,\alpha) - \log\Gamma(\beta) - \log y!\end{aligned} \tag{6}$$

Use the previous result, the partial derivatives with respect to the three parameters r, α, β are

$$\begin{aligned}\frac{\partial \log f}{\partial r} &= \psi(y+r) - \psi(y+r+\alpha+\beta) - \psi(r) + \psi(r+\alpha) \\ \frac{\partial \log f}{\partial \alpha} &= \psi(\alpha+\beta) - \psi(y+r+\alpha+\beta) - \psi(\alpha) + \psi(r+\alpha) \\ \frac{\partial \log f}{\partial \beta} &= \psi(\alpha+\beta) - \psi(y+r+\alpha+\beta) + \psi(y+\beta) - \psi(\beta)\end{aligned} \tag{7}$$

Derivatives of logarithmic ccdf

Then let's take a look at the ccdf. From Wolfrom, the ccdf for $Y \sim \text{BNB}(r, \alpha, \beta)$ is given by

$$\begin{aligned} &P(Y > y) = 1 - F(r, \alpha, \beta) = C(r, \alpha, \beta) \\ &= \frac{\Gamma(r+y+1)B(r+\alpha, \beta+y+1)\,{}_3F_2(\{1, r+y+1, \beta+y+1\}; \{y+2, r+\alpha+\beta+y+1\}; 1)}{\Gamma(r)B(\alpha,\beta)\Gamma(y+2)} \end{aligned} \tag{8}$$

where ${}_3F_2(\{a_1, a_2, a_3\}; \{b_1, b_2\}; z)$ is the generalized hypergeometric function [Olver, 2010, Ch. 16] for $p = 3, q = 2$.

It's too lengthy to explicitly express ${}_3F_2(\{1, r+y+1, \beta+y+1\}; \{y+2, r+\alpha+\beta+y+1\}; 1)$ everytime. In the following we use ellipsis instead of the six parameters. We will denote it as ${}_3F_2(...)$.

Remember now our task is to calculate

$$\nabla_{\boldsymbol{\theta}} \log C(y, \boldsymbol{\theta}) = \left[\frac{\partial \log C(\boldsymbol{\theta})}{\partial r}, \frac{\partial \log C(\boldsymbol{\theta})}{\partial \alpha}, \frac{\partial \log C(\boldsymbol{\theta})}{\partial \beta}\right] \tag{9}$$

Let's first take a close look at the first element in this vector. After simply taking the logarithm and taking the partial derivatives, we have

$$\begin{aligned} \frac{\partial \log C(r, \alpha, \beta)}{\partial r} &= \psi(r+y+1) + \psi(\alpha+r) - \psi(\alpha+\beta+r+y+1) - \psi(r) \\ &+ \frac{\partial \log {}_3F_2(...)}{\partial r}. \end{aligned} \tag{10}$$

The only term in this formula that's hard to express exactly from now is the last term, the partial derivative of the $\log {}_3F_2(...)$ w.r.t. r.

Tackle derivative of $\log {}_3F_2$

Although the notation may look complicated, all we need is the basic chain rule. We have

$$\frac{d \log f}{dt} = \frac{df}{dt}/f \tag{11}$$

Hence

$$\frac{\partial \log {}_3F_2(...)}{\partial r} = \frac{\partial {}_3F_2(...)}{\partial r}/{}_3F_2(...). \tag{12}$$

Now we are curious about $\frac{\partial {}_3F_2(...)}{\partial r}$.

Note that r appears in the second and fifth position of

$${}_3F_2(\{1, r+y+1, \beta+y+1\}; \{y+2, r+\alpha+\beta+y+1\}; 1).$$

Based on the derivative rules for multivariate composite functions. Given a function f that depends on multiple variables (say u and v), where each of these variables is a function of another variable t, then the derivative of f with respect to t is given by:

$$\frac{df}{dt} = \frac{\partial f}{\partial u}\frac{du}{dt} + \frac{\partial f}{\partial v}\frac{dv}{dt}. \tag{13}$$

Therefore

$$\frac{\partial_3 F_2(...)}{\partial r} = {}_3F_2(...)^{(\{0,1,0\},\{0,0\},0)}(...) + {}_3F_2(...)^{(\{0,0,0\},\{0,1\},0)}(...), \tag{14}$$

where the superscript $(\{0,0,1\},\{0,0\},0)$ denotes a specific derivative of the hypergeometric function ${}_3F_2$.

Expression ${}_3F_2^{(\{0,0,1\},\{0,0\},0)}(\{a_1,a_2,a_3\},\{b_1,b_2\},z)$ indicates that we are taking the first derivative with respect to the third parameter a_3. The zeros means that no differentiation is to be taken with respect to the corresponding parameters, i.e.,

$${}_3F_2^{(\{0,0,1\},\{0,0\},0)}(\{a_1,a_2,a_3\},\{b_1,b_2\},z) = \frac{\partial \log {}_3F_2(\{a_1,a_2,a_3\},\{b_1,b_2\},z)}{\partial a_3}. \tag{15}$$

Finally, the partial derivative of the $\log C(r,\alpha,\beta)$ w.r.t. r is

$$\begin{aligned}\frac{\partial \log {}_3F_2(...)}{\partial r} &= \frac{\partial_3 F_2(...)}{\partial r}/{}_3F_2(...) \\ &= \left[{}_3F_2(...)^{(\{0,1,0\},\{0,0\},0)}(...) +_3 F_2(...)^{(\{0,0,0\},\{0,1\},0)}(...)\right]/{}_3F_2(...).\end{aligned} \tag{16}$$

Similarly, the partial derivative of the $\log C(r,\alpha,\beta)$ w.r.t. α,β are

$$\begin{aligned}\frac{\partial \log C(r,\alpha,\beta)}{\partial \alpha} &= \psi(\alpha+r) - \psi(\alpha+\beta+r+y+1) \\ &\quad + \frac{\partial \log {}_3F_2(...)}{\partial \alpha} - \psi(\alpha) + \psi(\alpha+\beta) \\ \frac{\partial \log C(r,\alpha,\beta)}{\partial \beta} &= \psi(\beta+y+1) - \psi(\alpha+\beta+r+y+1) \\ &\quad + \frac{\partial \log {}_3F_2(...)}{\partial \beta} - \psi(\beta) + \psi(\alpha+\beta),\end{aligned} \tag{17}$$

where

$$\begin{aligned}\frac{\partial \log {}_3F_2(...)}{\partial \alpha} &= {}_3F_2(...)^{(\{0,0,0\},\{0,1\},0)}(...)/{}_3F_2(...) \\ \frac{\partial \log {}_3F_2(...)}{\partial \beta} &= \left[{}_3F_2(...)^{(\{0,0,1\},\{0,0\},0)}(...) +_3 F_2(...)^{(\{0,0,0\},\{0,1\},0)}(...)\right]/{}_3F_2(...)\end{aligned} \tag{18}$$

Derivatives of logarithmic cdf

The cdf for $Y \sim \text{BNB}(r,\alpha,\beta)$ is given by

$$P(Y \le y) = F(r,\alpha,\beta) = 1 - C(r,\alpha,\beta) \tag{19}$$

The partial derivative of the $F(r,\alpha,\beta)$ w.r.t. r is

$$\begin{aligned}\frac{\partial \log F(r,\alpha,\beta)}{\partial r} &= \frac{\partial \log[1 - C(r,\alpha,\beta)]}{\partial r} \\ &= -\frac{1}{1 - C(r,\alpha,\beta)} \frac{\partial C(r,\alpha,\beta)}{\partial r} \\ &= -\frac{1}{1 - C(r,\alpha,\beta)} \frac{\partial \log C(r,\alpha,\beta)}{\partial r} C(r,\alpha,\beta).\end{aligned} \tag{20}$$

This is to say, to know $\frac{\partial \log F(r,\alpha,\beta)}{\partial r}$, we only need to know $\frac{\partial \log C(r,\alpha,\beta)}{\partial r}$, which we've already give the the previous subsection. The same for α and β.

It is difficult to implement these functions directly in Stan. First, the ${}_3F_2$ function and its derivatives are not provided in Stan. Second, Stan cannot use user-defined gradients. These are the first and third issues raised at the end of the previous section.

4 Implementation

We've worked out

$$\nabla_{\boldsymbol{\theta}} \log f(\boldsymbol{\theta}), \nabla_{\boldsymbol{\theta}} \log F(\boldsymbol{\theta}), \nabla_{\boldsymbol{\theta}} \log C(\boldsymbol{\theta}). \tag{21}$$

For the implementation, our aim is to compelete four functions: `beta_neg_binomial_lpmf`, `beta_neg_binomial_lcdf`, `beta_neg_binomial_lccdf` and `beta_neg_binomial_rng`. The existing probability functions in the Stan Math Library provide very high-quality examples, which can be found here.

The skeleton of each function is basically as follows.

```
template < {template parameters} >
stan::return_type_t< ... > {distribution}_lpmf( {function parameters} ) {

   // Type Aliases
   using stan::partials_return_t;
   using T_partials_return = partials_return_t< ... >;
   ...

   // Error Handling

   // Check sizes of input parameters
   check_consistent_sizes()
   if (size_zero( {any} )) {
     return 0.0;
   }
   ...

   // Check domain of input parameters
   check_positive_finite()
   ...
   // Add other domain checks as needed

   // Other potential error checks

   // Initialization
   T_partials_return logp = 0.0; // Initialize log probability
   operands_and_partials< {operand types} > ops_partials( {operands} ); // Initialize partial
       derivatives

   // Convert inputs to vector views to handle both scalars and vectors
```

```
    scalar_seq_view< {input types} > {input names}({input variables});
    ...

    // Determine sizes of input data
    size_t size_{input1} = stan::math::size({input1});
    size_t size_{input2} = stan::math::size({input2});
    ...

    // Implementation Details
    for (size_t i = 0; i < max_size( {inputs} ); ++i) {
        // Core logic for calculating the beta-negative binomial log probability
        // This typically involves calls to functions in stan::math, etc.
        // Example:
        // logp += lgamma({args}) - lgamma({args});

        // Gradient calculations for automatic differentiation
        if (!is_constant_all< {input types} >::value) {
            // Compute partial derivatives
            // Example:
            // ops_partials.edge1_.partials_[i] += ...
            // ops_partials.edge2_.partials_[i] += ...
        }
    }

    // Collect results and return
    return ops_partials.build(logp);
}
```

4.1 beta_neg_binomial_lpmf

Let's start with `beta_neg_binomial_lpmf`. We only need to write one function instead of writing it for each input type permutation.

Identity namespace and write code within it.

```
namespace <THE_NAME_OF_STAN_MODEL>_model_namespace {
  ......
}
```

Declaration of the function.

```
template <bool propto, typename T_n, typename T_r, typename T_size1,
          typename T_size2,
          stan::require_all_not_nonscalar_prim_or_rev_kernel_expression_t<
              T_n, T_r, T_size1, T_size2>* = nullptr>
stan::return_type_t<T_r, T_size1, T_size2> beta_neg_binomial_lpmf(const T_n& n,
                                                  const T_r& r,
                                                  const T_size1& alpha,
                                                  const T_size2& beta,
                                                  std::ostream* pstream__) {
  ......
}
```

Specify aliases.

```
  using stan::partials_return_t;
  using stan::ref_type_t;
  using stan::ref_type_if_t;
  using stan::is_constant;
  using stan::is_constant_all;
  using stan::VectorBuilder;
  using stan::scalar_seq_view;
```

```
using stan::math::lgamma;
using stan::math::size;
using stan::math::max_size;
using T_partials_return = partials_return_t<T_r, T_size1, T_size2>;
using T_r_ref = ref_type_t<T_r>;
using T_alpha_ref = ref_type_t<T_size1>;
using T_beta_ref = ref_type_t<T_size2>;
```

Check whether the shape of the incoming data conforms to the specification. It throws a `std::invalid_argument` if the sizes of the input containers don't match.

```
static const char* function = "beta_neg_binomial_lpmf";
check_consistent_sizes(function, "Failures variable", n,
                       "Number of failure parameter", r,
                       "Prior success parameter", alpha,
                       "Prior failure parameter", beta);
if (size_zero(n, r, alpha, beta)) {
  return 0.0;
}
```

Check whether the value of the incoming parameter vectors are within the parameter spaces. throws a `std::domain_error` if any of the parameters are not positive and finite.

```
T_r_ref r_ref = r;
T_alpha_ref alpha_ref = alpha;
T_beta_ref beta_ref = beta;
check_positive_finite(function, "Number of failure parameter", r_ref);
check_positive_finite(function, "Prior success parameter", alpha_ref);
check_positive_finite(function, "Prior failure parameter", beta_ref);
```

If `propto = TRUE` and all other parameters are not autodiff types, return zero.

```
if (!include_summand<propto, T_r, T_size1, T_size2>::value) {
  return 0.0;
}
```

Initialization return value, as well as some quantities that will be reused in subsequent calculations.

```
T_partials_return logp(0.0);
operands_and_partials<T_r_ref, T_alpha_ref, T_beta_ref> ops_partials(r_ref, alpha_ref,
    beta_ref);

scalar_seq_view<T_n> n_vec(n);
scalar_seq_view<T_r_ref> r_vec(r_ref);
scalar_seq_view<T_alpha_ref> alpha_vec(alpha_ref);
scalar_seq_view<T_beta_ref> beta_vec(beta_ref);
size_t size_n = stan::math::size(n);
size_t size_r = stan::math::size(r);
size_t size_alpha = stan::math::size(alpha);
size_t size_beta = stan::math::size(beta);
size_t size_n_r = max_size(n, r);
size_t size_r_alpha = max_size(r, alpha);
size_t size_n_beta = max_size(n, beta);
size_t size_alpha_beta = max_size(alpha, beta);
size_t max_size_seq_view = max_size(n, r, alpha, beta);
```

Determines whether support for incoming observations is valid.

```
for (size_t i = 0; i < max_size_seq_view; i++) {
  if (n_vec[i] < 0) {
    return ops_partials.build(LOG_ZERO);
```

```
  }
}
```

Compute the log pmf

$$\log f(y, r, \alpha, \beta) = \left[\frac{\mathrm{B}(r + y, \alpha + \beta)}{\mathrm{B}(r, \alpha)} \frac{\Gamma(y + \beta)}{y!\, \Gamma(\beta)} \right]. \tag{22}$$

```
// compute gamma(n+1)
VectorBuilder<include_summand<propto>::value, T_partials_return, T_n>
    normalizing_constant(size_n);
for (size_t i = 0; i < size_n; i++)
  if (include_summand<propto>::value)
    normalizing_constant[i] = -lgamma(n_vec[i] + 1);

// compute lbeta denominator with size r and alpha
VectorBuilder<true, T_partials_return, T_r, T_size1> lbeta_denominator(size_r_alpha);
for (size_t i = 0; i < size_r_alpha; i++) {
  lbeta_denominator[i] = lbeta(r_vec.val(i), alpha_vec.val(i));
}

// compute lgamma denominator with size beta
VectorBuilder<true, T_partials_return, T_size2> lgamma_denominator(size_beta);
for (size_t i = 0; i < size_beta; i++) {
  lgamma_denominator[i] = lgamma(beta_vec.val(i));
}

// compute lgamma numerator with size n and beta
VectorBuilder<true, T_partials_return, T_n, T_size2> lgamma_numerator(size_n_beta);
for (size_t i = 0; i < size_n_beta; i++) {
  lgamma_numerator[i] = lgamma(n_vec[i] + beta_vec.val(i));
}

// compute lbeta numerator with size n, r, alpha and beta
VectorBuilder<true, T_partials_return, T_n, T_r, T_size1, T_size2> lbeta_diff(
    max_size_seq_view);
for (size_t i = 0; i < max_size_seq_view; i++) {
  lbeta_diff[i] = lbeta(n_vec[i] + r_vec.val(i),
                        alpha_vec.val(i) + beta_vec.val(i)) + lgamma_numerator[i]
                - lbeta_denominator[i] - lgamma_denominator[i];
}
```

Compute derivative with respect to r, α and β, on the basis of needs.

$$\begin{aligned}
\frac{\partial \log f}{\partial r} &= \psi(y + r) - \psi(y + r + \alpha + \beta) - \psi(r) + \psi(r + \alpha) \\
\frac{\partial \log f}{\partial \alpha} &= \psi(\alpha + \beta) - \psi(y + r + \alpha + \beta) - \psi(\alpha) + \psi(r + \alpha) \\
\frac{\partial \log f}{\partial \beta} &= \psi(\alpha + \beta) - \psi(y + r + \alpha + \beta) + \psi(y + \beta) - \psi(\beta)
\end{aligned} \tag{23}$$

```
// compute digamma(n+r+alpha+beta)
VectorBuilder<!is_constant_all<T_r, T_size1, T_size2>::value, T_partials_return,
              T_n, T_r, T_size1, T_size2>
    digamma_n_r_alpha_beta(max_size_seq_view);
if (!is_constant_all<T_r, T_size1, T_size2>::value) {
  for (size_t i = 0; i < max_size_seq_view; i++) {
    digamma_n_r_alpha_beta[i]
        = digamma(n_vec[i] + r_vec.val(i) + alpha_vec.val(i) + beta_vec.val(i));
  }
```

```
}

// compute digamma(alpha+beta)
VectorBuilder<!is_constant_all<T_size1, T_size2>::value, T_partials_return,
             T_size1, T_size2>
    digamma_alpha_beta(size_alpha_beta);
if (!is_constant_all<T_size1, T_size2>::value) {
  for (size_t i = 0; i < size_alpha_beta; i++) {
    digamma_alpha_beta[i] = digamma(alpha_vec.val(i) + beta_vec.val(i));
  }
}

// compute digamma(n+r)
VectorBuilder<!is_constant_all<T_r>::value, T_partials_return, T_n, T_r>
    digamma_n_r(size_n_r);
if (!is_constant_all<T_r>::value) {
  for (size_t i = 0; i < size_n_r; i++) {
    digamma_n_r[i] = digamma(n_vec[i] + r_vec.val(i));
  }
}

// compute digamma(r+alpha)
VectorBuilder<!is_constant_all<T_r, T_size1>::value, T_partials_return, T_r, T_size1>
    digamma_r_alpha(size_r_alpha);
if (!is_constant_all<T_r, T_size1>::value) {
  for (size_t i = 0; i < size_r_alpha; i++) {
    digamma_r_alpha[i] = digamma(r_vec.val(i) + alpha_vec.val(i));
  }
}

// compute digamma(n+beta)
VectorBuilder<!is_constant_all<T_size2>::value, T_partials_return, T_n, T_size2>
    digamma_n_beta(size_n_beta);
if (!is_constant_all<T_n, T_size2>::value) {
  for (size_t i = 0; i < size_n_beta; i++) {
    digamma_n_beta[i] = digamma(n_vec[i] + beta_vec.val(i));
  }
}

// compute digamma(r)
VectorBuilder<!is_constant_all<T_r>::value, T_partials_return, T_r> digamma_r(size_r);
if (!is_constant_all<T_r>::value) {
  for (size_t i = 0; i < size_r; i++) {
    digamma_r[i] = digamma(r_vec.val(i));
  }
}

// compute digamma(alpha)
VectorBuilder<!is_constant_all<T_size1>::value, T_partials_return, T_size1> digamma_alpha(
    size_alpha);
if (!is_constant_all<T_size1>::value) {
  for (size_t i = 0; i < size_alpha; i++) {
    digamma_alpha[i] = digamma(alpha_vec.val(i));
  }
}

// compute digamma(beta)
VectorBuilder<!is_constant_all<T_size2>::value, T_partials_return, T_size2> digamma_beta(
    size_beta);
if (!is_constant_all<T_size2>::value) {
  for (size_t i = 0; i < size_beta; i++) {
    digamma_beta[i] = digamma(beta_vec.val(i));
  }
```

```
}
```

Build the return value.

```
for (size_t i = 0; i < max_size_seq_view; i++) {
  if (include_summand<propto>::value)
    logp += normalizing_constant[i];
  logp += lbeta_diff[i];

  if (!is_constant_all<T_r>::value)
    ops_partials.edge1_.partials_[i]
        += digamma_n_r[i] - digamma_n_r_alpha_beta[i] - (digamma_r[i] - digamma_r_alpha[i]);
  if (!is_constant_all<T_size1>::value)
    ops_partials.edge2_.partials_[i]
        += digamma_alpha_beta[i] - digamma_n_r_alpha_beta[i] - (digamma_alpha[i] -
            digamma_r_alpha[i]);
  if (!is_constant_all<T_size2>::value)
    ops_partials.edge3_.partials_[i]
        += digamma_alpha_beta[i] - digamma_n_r_alpha_beta[i] + digamma_n_beta[i] -
            digamma_beta[i];
}
return ops_partials.build(logp);
```

For pmf/pdf functions, we have to overload the template function defined above. This version of the function template does not include the `propto` parameter (default to false). It provides a simpler interface, which is used for direct function calls.

```
template <typename T_n, typename T_r, typename T_size1, typename T_size2>
inline stan::return_type_t<T_r, T_size1, T_size2> beta_neg_binomial_lpmf(const T_n& n,
                                                const T_r& r,
                                                const T_size1& alpha,
                                                const T_size2& beta,
                                                std::ostream* pstream__) {
  return beta_neg_binomial_lpmf<false>(n, r, alpha, beta);
}
```

4.2 `beta_neg_binomial_lccdf`

Since `beta_neg_binomial_lccdf` and `beta_neg_binomial_lcdf` are close in terms of formulation:

$$F(r, \alpha, \beta) = 1 - C(r, \alpha, \beta) \tag{24}$$

$$\begin{aligned} \frac{\partial \log F(r, \alpha, \beta)}{\partial r} &= \frac{\partial \log[1 - C(r, \alpha, \beta)]}{\partial r} \\ &= -\frac{1}{1 - C(r, \alpha, \beta)} \frac{\partial C(r, \alpha, \beta)}{\partial r} \\ &= -\frac{1}{1 - C(r, \alpha, \beta)} \frac{\partial \log C(r, \alpha, \beta)}{\partial r} C(r, \alpha, \beta). \end{aligned} \tag{25}$$

Let's explain the structure of `beta_neg_binomial_lccdf` in detail.

Looking further at the specific expressions, we found that there are two difficulties in the implementation of lccdf, namely ${}_3F_2$ and its derivatives $\frac{\partial {}_3F_2(...)}{\partial \boldsymbol{\theta}}$. Luckily, we have `hypergeometric_3F2` and `grad_F32` in Stan Math Library. See docs in `hypergeometric_3F2.hpp` and `grad_F32.hpp` .

Likewise, first declare the function

```
template <typename T_n, typename T_r, typename T_size1, typename T_size2>
stan::return_type_t<T_size1, T_size2> beta_neg_binomial_lccdf(const T_n& n,
                                                const T_r& r,
                                                const T_size1& alpha,
                                                const T_size2& beta,
                                                std::ostream* pstream__) {
  ......
}
```

Make import

```
  using stan::partials_return_t;
  using stan::ref_type_t;
  using stan::ref_type_if_t;
  using stan::is_constant;
  using stan::is_constant_all;
  using stan::VectorBuilder;
  using stan::scalar_seq_view;
  using stan::math::lgamma;
  using stan::math::size;
  using stan::math::max_size;

  using T_partials_return = partials_return_t<T_n, T_r, T_size1, T_size2>;
  using std::exp;
  using std::log;
  using T_r_ref = ref_type_t<T_r>;
  using T_alpha_ref = ref_type_t<T_size1>;
  using T_beta_ref = ref_type_t<T_size2>;
```

Check inputs

```
  static const char* function = "beta_neg_binomial_lccdf";
  check_consistent_sizes(function, "Failure variable", n,
                         "Number of failure parameter", r,
                         "Prior success parameter", alpha,
                         "Prior failure parameter", beta);
  if (size_zero(n, r, alpha, beta)) {
    return 0;
  }

  T_r_ref r_ref = r;
  T_alpha_ref alpha_ref = alpha;
  T_beta_ref beta_ref = beta;
  check_positive_finite(function, "Number of failure parameter", r_ref);
  check_positive_finite(function, "Prior success parameter", alpha_ref);
  check_positive_finite(function, "Prior failure parameter", beta_ref);
```

Initialization

```
  T_partials_return P(0.0);
  operands_and_partials<T_r_ref, T_alpha_ref, T_beta_ref> ops_partials(r_ref, alpha_ref,
      beta_ref);

  scalar_seq_view<T_n> n_vec(n);
  scalar_seq_view<T_r_ref> r_vec(r_ref);
  scalar_seq_view<T_alpha_ref> alpha_vec(alpha_ref);
  scalar_seq_view<T_beta_ref> beta_vec(beta_ref);
  size_t max_size_seq_view = max_size(n, r, alpha, beta);
```

Having previously checked the range of the parameters, don't forget to check the range of the observations

```
// Explicit return for extreme values
// The gradients are technically ill-defined, but treated as neg infinity
for (size_t i = 0; i < stan::math::size(n); i++) {
  if (n_vec.val(i) < 0) {
    return ops_partials.build(negative_infinity());
  }
}
```

Compute the log ccdf

$$\begin{aligned}\log P(Y > y) &= \log C(r, \alpha, \beta) \\ &= \log \Gamma(r + y + 1) + \log B(r + \alpha, \beta + y + 1) \\ &+ {}_3F_2(\{1, r + y + 1, \beta + y + 1\}; \{y + 2, r + \alpha + \beta + y + 1\}; 1) \\ &- \log \Gamma(r) - \log B(\alpha, \beta) - \log \Gamma(y + 2)\end{aligned} \tag{26}$$

```
for (size_t i = 0; i < max_size_seq_view; i++) {
  const T_partials_return n_dbl = n_vec.val(i);
  const T_partials_return r_dbl = r_vec.val(i);
  const T_partials_return alpha_dbl = alpha_vec.val(i);
  const T_partials_return beta_dbl = beta_vec.val(i);
  const T_partials_return b_plus_n = beta_dbl + n_dbl;
  const T_partials_return r_plus_n = r_dbl + n_dbl;
  const T_partials_return a_plus_r = alpha_dbl + r_dbl;
  const T_partials_return one = 1;

  const T_partials_return F = hypergeometric_3F2({one, b_plus_n + 1, r_plus_n + 1},
                                                 {n_dbl + 2, a_plus_r + b_plus_n + 1}, one);
  T_partials_return C = lgamma(r_plus_n + 1) + lbeta(a_plus_r, b_plus_n + 1)
                      - lgamma(r_dbl) - lbeta(alpha_dbl, beta_dbl) - lgamma(n_dbl + 2);
  C = F * exp(C);

  const T_partials_return P_i = C;

  P += log(P_i);
```

And the derivatives

$$\begin{aligned}\frac{\partial \log C(r, \alpha, \beta)}{\partial r} &= \psi(r + y + 1) + \psi(\alpha + r) - \psi(\alpha + \beta + r + y + 1) - \psi(r) \\ &+ \frac{\partial \log {}_3F_2(...)}{\partial r} \\ \frac{\partial \log C(r, \alpha, \beta)}{\partial \alpha} &= \psi(\alpha + r) - \psi(\alpha + \beta + r + y + 1) \\ &+ \frac{\partial \log {}_3F_2(...)}{\partial \alpha} - \psi(\alpha) + \psi(\alpha + \beta) \\ \frac{\partial \log C(r, \alpha, \beta)}{\partial \beta} &= \psi(\beta + y + 1) - \psi(\alpha + \beta + r + y + 1) \\ &+ \frac{\partial \log {}_3F_2(...)}{\partial \beta} - \psi(\beta) + \psi(\alpha + \beta)\end{aligned} \tag{27}$$

where

$$
\begin{aligned}
\frac{\partial \log {}_3F_2(...)}{\partial r} &= \left[{}_3F_2(...)^{(\{0,1,0\},\{0,0\},0)}(...) +_3 F_2(...)^{(\{0,0,0\},\{0,1\},0)}(...) \right] /_3F_2(...) \\
\frac{\partial \log {}_3F_2(...)}{\partial \alpha} &= {}_3F_2(...)^{(\{0,0,0\},\{0,1\},0)}(...) /_3F_2(...) \\
\frac{\partial \log {}_3F_2(...)}{\partial \beta} &= \left[{}_3F_2(...)^{(\{0,0,1\},\{0,0\},0)}(...) +_3 F_2(...)^{(\{0,0,0\},\{0,1\},0)}(...) \right] /_3F_2(...)
\end{aligned} \tag{28}
$$

```
  T_partials_return digamma_abrn
      = is_constant_all<T_r, T_size1, T_size2>::value
          ? 0
          : digamma(a_plus_r + b_plus_n + 1);
  T_partials_return digamma_ab
      = is_constant_all<T_size1, T_size2>::value
          ? 0
          : digamma(alpha_dbl + beta_dbl);

  T_partials_return dF[6];
  if (!is_constant_all<T_r, T_size1, T_size2>::value) {
    grad_F32(dF, one, b_plus_n + 1, r_plus_n + 1, n_dbl + 2, a_plus_r + b_plus_n + 1, one, 1e
        -3);
  }
  if (!is_constant_all<T_r>::value) {
    ops_partials.edge1_.partials_[i]
        += digamma(r_plus_n + 1) + (digamma(a_plus_r) - digamma_abrn) + (dF[2] + dF[4]) / F -
            digamma(r_dbl);
  }
  if (!is_constant_all<T_size1>::value) {
    ops_partials.edge1_.partials_[i]
        += digamma(a_plus_r) - digamma_abrn + dF[4] / F - (digamma(alpha_dbl) - digamma_ab);
  }
  if (!is_constant_all<T_size2>::value) {
    ops_partials.edge2_.partials_[i]
        += digamma(b_plus_n + 1) - digamma_abrn + (dF[1] + dF[4]) / F - (digamma(beta_dbl) -
            digamma_ab);
  }
 }
```

Finally

```
return ops_partials.build(P);
```

4.3 beta_neg_binomial_rng

In the following we describe the random number generator. $Y \sim \text{BNB}(r, \alpha, \beta)$ is the same as

$$
\begin{aligned}
Y &\sim \text{NB}(r, p) \\
p &\sim \text{Beta}(\alpha, \beta),
\end{aligned}
$$

We first sample p from the beta distribution then sample Y from the negative binomial distribution. Stan provides functions to generate random numbers from these two distributions. We just need to pay attention to parameterization.

In Stan, the negative binomial distribution is given by

$$
\text{NegBinomial}(y \mid \alpha, \beta) = \binom{y + \alpha - 1}{y} \left(\frac{\beta}{\beta + 1} \right)^{\alpha} \left(\frac{1}{\beta + 1} \right)^{y}. \tag{29}
$$

In our case

$$f(y \mid r, p) = \binom{y+r-1}{y}(1-p)^y p^r. \tag{30}$$

Through observation, the meaning of r is the same. We have to solve for p.

$$p = \frac{\beta}{\beta+1} \Rightarrow \beta = \frac{p}{1-p} \tag{31}$$

Therefore, provided r, α, β, we generate $p \sim \text{Beta}(\alpha, \beta)$ and then sample $Y \sim \text{NB}(r, \frac{p}{1-p})$.

In the implementation, because this is a random number generating function, we don't need to consider Stan's unique automatic differentiation type of vector, only scalars and `std::vector`. So the code is relatively simple.

Routinely perform shape and range checks.

```
template <typename T_r, typename T_shape1, typename T_shape2, class RNG>
inline typename stan::VectorBuilder<true, int, T_r, T_shape1, T_shape2>::type
beta_neg_binomial_rng(const T_r &r, const T_shape1 &alpha, const T_shape2 &beta,
                RNG &rng, std::ostream* pstream__) {

  using stan::ref_type_t;
  using stan::VectorBuilder;
  using namespace stan::math;

  using T_r_ref = ref_type_t<T_r>;
  using T_alpha_ref = ref_type_t<T_shape1>;
  using T_beta_ref = ref_type_t<T_shape2>;
  static const char *function = "beta_neg_binomial_rng";
  check_consistent_sizes(function, "Number of failure parameter", r,
                       "Prior success parameter", alpha,
                       "Prior failure parameter", beta);

  T_r_ref r_ref = r;
  T_alpha_ref alpha_ref = alpha;
  T_beta_ref beta_ref = beta;
  check_positive_finite(function, "Number of failure parameter", r_ref);
  check_positive_finite(function, "Prior success parameter", alpha_ref);
  check_positive_finite(function, "Prior failure parameter", beta_ref);
```

Use `beta_rng` to generate p. Then compute odds ratio $p/(1-p)$.

```
  using T_p = decltype(beta_rng(alpha_ref, beta_ref, rng));
  T_p p = beta_rng(alpha_ref, beta_ref, rng);

  T_p odds_ratio_p;

  if (std::is_same<T_p, double>::value) {
    // Scalar case
    odds_ratio_p = p / (1 - p);
  } else {
    // Vector case
    odds_ratio_p.resize(p.size());
    for (size_t i = 0; i < p.size(); i++) {
      odds_ratio_p[i] = p[i] / (1 - p[i]);
    }
  }
```

Use `neg_binomial_rng` to sample y.

```
  return neg_binomial_rng(r_ref, odds_ratio_p, rng);
}
```

5 Performance test

Then we compare the performance of precomputing gradients and using automatic differentiation on simulated datasets. We sampled $N = 10000$ points form $\text{BNB}(6, 2, 0.5)$. The theoretical mean is $\frac{6 \cdot 0.5}{2-1} = 3$. The sample mean is 2.99. and the sample variance is 103.85. We run the following model with three global parameters with standard normal priors. To prevent potential identification problems, we constrain β to be less than r.

$$\begin{aligned} Y_i &\sim \text{BNB}(r, \alpha, \beta), i = 1, ..., N \\ r &\sim \mathcal{N}(0, 1) \\ \alpha &\sim \mathcal{N}(0, 1) \\ \beta &\sim \mathcal{N}(0, 1) \quad T[0, r] \end{aligned} \tag{32}$$

We ran 1000 iterations on one chain, the table below shows the parameter estimation and performance comparison. The forward time represents the time spent on the forward pass of the model calculation (statements without automatic differentiation), including evaluating the logarithmic pdf/pmf, etc. The reverse time is the time it takes to compute the reverse pass of the gradient in the context of automatic differentiation. The total time is the sum of these two plus some overhead. The chain stacks represent the number of objects on the stack allocated for chained automatic differentiation. No chain stack represents storage consumed by computations that do not require differentiation. Autodiff calls (No autodiff calls) is the number of executions of the `profile` command in the code block with (without) automatic differentiation[1].

The total time of the pure Stan code with automatic differentiation is about 2 times that of the C++ code. The forward time is 37% more that that of the C++ code, which means that although the Stan code will be transpiled to C++, it is still slower than a typical C++ implementation. After using the analytic derivative, the backward pass time is reduced by 4 orders of magnitude (17000 times faster), already very close to zero. Also, the space occupation of chained automatic differentiation by pure Stan is 4-5 orders of magnitude higher than that of C++, see the Chain stacks entries. The No chain stack implemented by C++ is zero, because analytic derivatives make Stan not need to allocate additional storage space for the intermediate results of forward propagation for reverse mode automatic differentiation.

[1] https://mc-stan.org/docs/cmdstan-guide/stan_csv.html

Metric	**With C++**	**Pure Stan**
Likelihood		
Total time (s)	62.6521	143.977
Forward time (s)	62.6488	85.9782
Reverse time (s)	0.0033	57.9992
Chain stacks	35042	3080921832
No chain stacks	0	32796
Autodiff calls	35042	32796
No autodiff calls	1	1
Priors		
Total time (s)	0.0091	0.0144
Forward time (s)	0.0059	0.0101
Reverse time (s)	0.0031	0.0042
Chain stacks	105126	98388
No chain stacks	0	0
Autodiff calls	35042	32796
No autodiff calls	1	1
Posterior mean		
$\hat{r}$	4.136	4.070
$\hat{\alpha}$	1.725	1.714
$\hat{\beta}$	0.568	0.573

Table 2: Performance analysis comparison between C++ analytic derivative implementation and Stan's built-in automatic differentiation implementation.

6 Unit test

To do.

References

F. W. Olver. *NIST handbook of mathematical functions hardback and CD-ROM*. Cambridge university press, 2010. URL https://dlmf.nist.gov/.