Intro to BridgeStan

Who, Why, What, Why (again), and How

Brian Ward

February 16th, 2024

FLATIRON INSTITUTE

Who are we?

Myself

Software Engineer at Flatiron Institute

Bob Carpenter

Senior Research Scientist, Group Leader at Flatiron Institute

Edward Roualdes

Associate Professor at Cal State Chico

Why we needed BridgeStan

A *lot* of research is being done on statistical inference

MEADS, ChEES, DRHMC, Pathfinder, …

Edward wanted to actually try them all out, on models he cared about, written in Stan

Aside: Stan models, from the POV of a computer scientist

It's just a C++ class

```
data {
  // constants provided at startup
}
transformed data {
  // constants computable from above
  // only runs once during inference
}

parameters {
  // values provided to model each evaluation
  // can have constraints
}
transformed parameters {
  // values computed from above
  // runs every evaluation (w/ autodiff)
  // and for output
}
model {
  // runs every evaluation
  // computes single value called 'target'
  // and performs autodiff
}

generated quantities {
  // derived variables, not used in inference
  // run once per iteration, no autodiff!
}
```

Aside: Stan models, from the POV of a computer scientist

It’s just a C++ class

```
class model_base {
 public:

  // constructor is specific to each generated model, omitted

  virtual ~model_base() {}

  template <bool propto, bool jacobian>
  inline stan::math::var log_prob(Eigen::VectorX<stan::math::var>&
                                  params_r) const = 0

  virtual void transform_inits(const stan::io::var_context& context,
                               Eigen::VectorXd&
                               params_r) const = 0;

  virtual void write_array(boost::ecuyer1988& base_rng,
                           Eigen::VectorXd& params_r,
                           Eigen::VectorXd&
                           params_constrained_r) const = 0;

  virtual std::string model_name() const = 0;
  virtual std::vector<std::string> model_compile_info() const = 0;
  virtual void get_param_names(std::vector<std::string>&
                               names) const = 0;
  // more helpful stuff, omitted
}
```

```
data {
  // constants provided at startup
}
transformed data {
  // constants computable from above
  // only runs once during inference
}

parameters {
  // values provided to model each evaluation
  // can have constraints
}
transformed parameters {
  // values computed from above
  // runs every evaluation (w/ autodiff)
  // and for output
}
model {
  // runs every evaluation
  // computes single value called 'target'
  // and performs autodiff
}

generated quantities {
  // derived variables, not used in inference
  // run once per iteration, no autodiff!
}
```

```
class model_base {
 public:

  // constructor is specific to each generated model, omitted

  virtual ~model_base() {}

  template <bool propto, bool jacobian>
  inline stan::math::var log_prob(Eigen::VectorX<stan::math::var>&
                                  params_r) const = 0

  virtual void transform_inits(const stan::io::var_context& context,
                               Eigen::VectorXd&
                               params_r) const = 0;

  virtual void write_array(boost::ecuyer1988& base_rng,
                           Eigen::VectorXd& params_r,
                           Eigen::VectorXd&
                           params_constrained_r) const = 0;

  virtual std::string model_name() const = 0;
  virtual std::vector<std::string> model_compile_info() const = 0;
  virtual void get_param_names(std::vector<std::string>&
                               names) const = 0;
  // more helpful stuff, omitted
}
```

Existing tools were lacking

Needed some way to hook into the **Stan model class** itself

- RStan has some support, but Edward did not use R
- PyStan had dropped support for this in its update to 3.0
- Writing all these algorithms natively in C++ was a non-starter

A lot of the Stan team (myself included) had assumed anything that did this needed to be at least as complicated as RStan, or be slow.

Edward decided to just write some code.

The first working version of BridgeStan fit in about 100 lines of C++ and 50 lines of Julia.

It exposed exactly the one function Edward needed to write his algorithms in Julia: a way to calculate the log density of a set of parameters and the gradient with respect to those.

He was able to do this because he combined a few simple tricks the rest of us had overlooked

(More on this in the “How” section)

```cpp
// - snip -

extern "C" {

  struct stanmodel_struct
  {
    void* model_;
    unsigned int seed_;         // TODO don't need this
  };

  stanmodel* _stanmodel_create(char* data_file_path_, unsigned int seed_) {
    std::string data_file_path(data_file_path_);
    // TODO(ear) add catch if data_file_path_ is empty
    // https://github.com/bob-carpenter/stan-model-server/blob/8916ea58cd80da1
/src/main.cpp#L524
    std::ifstream in(data_file_path);
    if (!in.good())
      throw std::runtime_error("Cannot read input file: " + data_file_path);
    cmdstan::json::json_data data(in);
    in.close();

    stanmodel* sm = new stanmodel();
    sm->seed_ = seed_;
    // TODO(ear) try this
    sm->model_ = &new_model(data, seed_, &std::cerr);
    // instead of the below
    // stan::model::model_base* model = &new_model(data, seed_, &std::cerr);
    // sm->model_ = model;
    return sm;
  }

  void _stanmodel_log_density(stanmodel* sm_, double* q_, int D_, double* log_
int jacobian_) {
    const Eigen::Map<Eigen::VectorXd> params_unc(q_, D_);
    Eigen::VectorXd grad;
    std::ostream& err_ = std::cerr; // TODO(ear) maybe std::out

    stan::model::model_base* model = static_cast<stan::model::model_base*>(sm_
    auto model_functor = create_model_functor(model, propto_, jacobian_, err_)

    stan::math::gradient(model_functor, params_unc, *log_density_, grad);

    for (Eigen::VectorXd::Index d = 0; d < D_; ++d) {
      grad_[d] = grad(d);
    }
  }

  int _stanmodel_get_num_unc_params(stanmodel* sm_) {
    stan::model::model_base* model = static_cast<stan::model::model_base*>(sm_
    bool include_generated_quantities = false;
    bool include_transformed_parameters = false;
    std::vector<std::string> names;
    model->unconstrained_param_names(names, include_generated_quantities,
                                     include_transformed_parameters);
    return names.size();
  }

  void _stanmodel_destroy(stanmodel* sm_) {
    if (sm_ == NULL) return;
    delete static_cast<stan::model::model_base*>(sm_->model_);
    delete sm_;
  }
} /* extern "C" */
```

```julia
module JuliaBridgeStan

mutable struct StanModelSymbol
    lib::Ref{Nothing}
    create::Ref{Nothing}
    numparams::Ref{Nothing}
    logdensity::Ref{Nothing}
     free::Ref{Nothing}
    function StanModelSymbol(path::String)
        lib = Libc.Libdl.dlopen(path)
        # TODO probably don't need stanmodel_ prefixing each of these
        crsym = Libc.Libdl.dlsym(lib, :stanmodel_create)
        npsym = Libc.Libdl.dlsym(lib, :stanmodel_get_num_unc_params)
        ldsym = Libc.Libdl.dlsym(lib, :stanmodel_log_density)
        frsym = Libc.Libdl.dlsym(lib, :stanmodel_destroy) # TODO rename free
        return new(lib, crsym, npsym, ldsym, frsym)
        # TODO in my head this should be close when out of scope...
        # but I can't get it to work
        # function f(sms)
        #     Libc.Libdl.dlclose(sms.lib)
        # end
        # finalizer(f, sms)
    end
end

mutable struct StanModelStruct
end

mutable struct StanModel
    smsym::StanModelSymbol
    stanmodel::Ptr{StanModelStruct}
    D::Int
    data::String
    seed::UInt32
    logdensity::Vector{Float64}
    grad::Vector{Float64}
    function StanModel(stanlib_::String, datafile_::String, seed_ = 204)
        sms = StanModelSymbol(stanlib_)
        seed = convert(UInt32, seed_)
        stanmodel = ccall(sms.create, Ptr{StanModelStruct},
                          (Cstring, UInt32),
                          datafile_, seed)
        D = ccall(sms.numparams, Cint, (Ptr{Cvoid},), stanmodel)
        sm = new(sms, stanmodel, D, datafile_, seed, zeros(1), zeros(D))
        function f(sm)
            ccall(sm.smsym.free, Cvoid, (Ptr{Cvoid},), sm.stanmodel)
        end
        finalizer(f, sm)
    end
end

function numparams(sm::StanModel)
    return ccall(sm.smsym.numparams, Cint, (Ptr{Cvoid},), sm.stanmodel)
end

function logdensity_grad!(sm::StanModel, q; propto = 1, jacobian = 1)
    ccall(sm.smsym.logdensity, Cvoid,
          (Ptr{StanModelStruct}, Ref{Cdouble},
           Cint, Ref{Cdouble}, Ref{Cdouble}, Cint, Cint),
          sm.stanmodel, q, sm.D, sm.logdensity, sm.grad, propto, jacobian)
end

export
    StanModel,
    numparams,
    logdensity_grad!

end
```

What is BridgeStan?

BridgeStan is a library exposing the details of any Stan model …

From Edward's original idea grew a package which can give you

- Log density calculations, gradients, Hessians
- Constraining and unconstraining variable transforms
- Access to generated quantities
- Variable names and model metadata

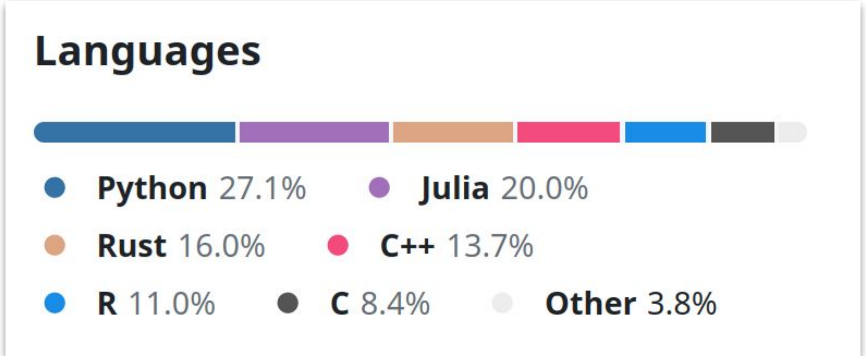
for any Stan model.

FLATIRON INSTITUTE

… in a language you actually use

And it will give you them in Julia, Python, R, Rust, and anything else* that can call C functions.

Language	files	blank	comment	code
C++	6	110	204	732
Python	7	144	301	644
Julia	6	155	231	638
Rust	5	73	162	563
Markdown	8	363	0	500
R	4	34	147	376
Stan	19	2	0	199
make	2	33	23	149
C/C++ Header	2	51	308	119
C	1	5	1	25
JavaScript	1	0	0	25
CSS	1	3	0	15
SUM:	62	973	1377	3985



* Some assembly required

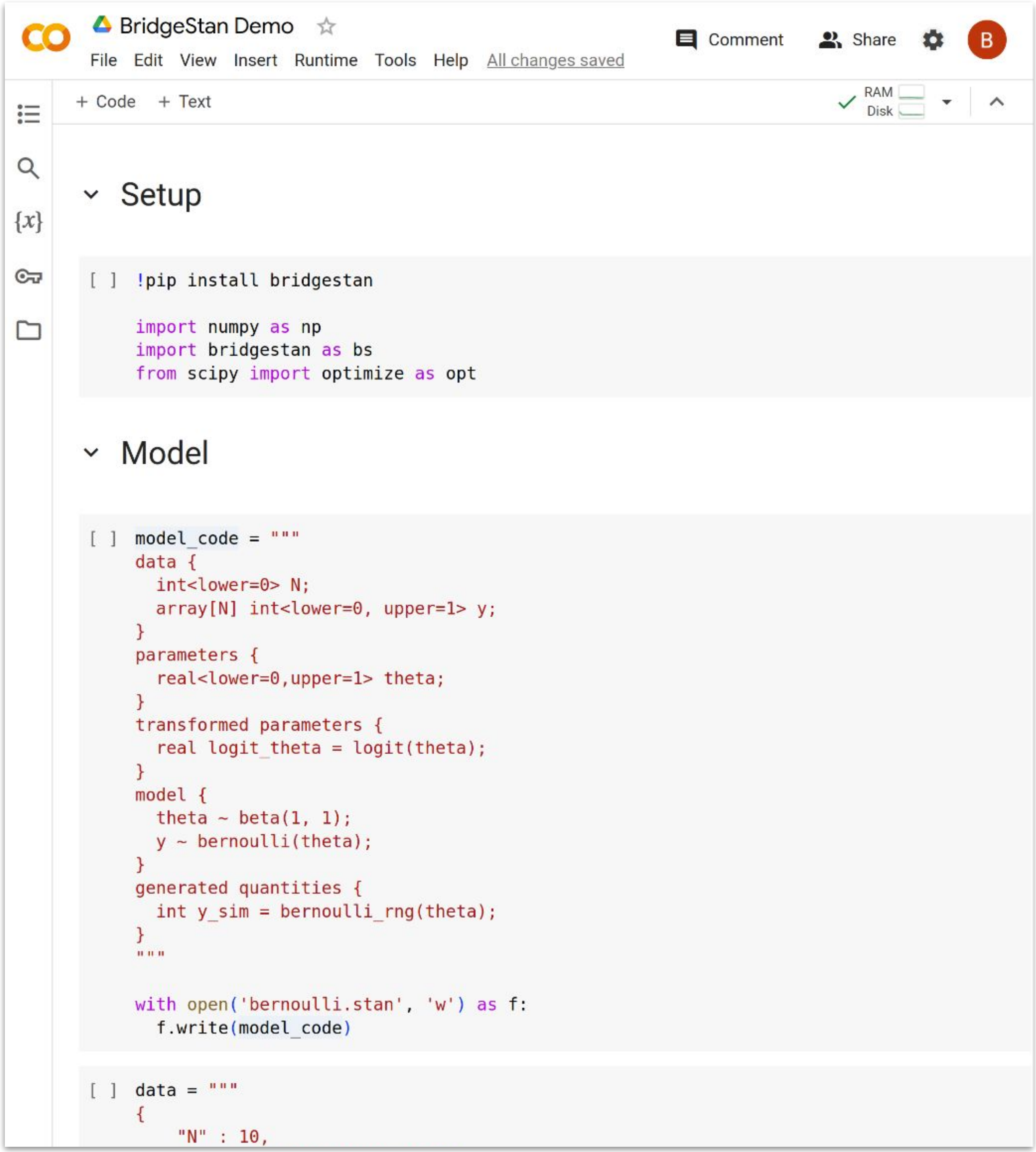
BridgeStan plays nice

- Exceptions in Stan turn into to proper errors in the higher language.
- Print statements in Stan end up where you would expect.
- Opt-in thread safety for multithreaded calls to all functions
- Installation and build automated in each language
- Good documentation, examples, and testing
- **As few copies and as little overhead as possible**

What

Showcase

http://tinyurl.com/5esj5txr



BridgeStan Demo

File Edit View Insert Runtime Tools Help All changes saved

Comment Share

+ Code + Text

RAM Disk

Setup

```
!pip install bridgestan

import numpy as np
import bridgestan as bs
from scipy import optimize as opt
```

Model

```
model_code = """
data {
  int<lower=0> N;
  array[N] int<lower=0, upper=1> y;
}
parameters {
  real<lower=0,upper=1> theta;
}
transformed parameters {
  real logit_theta = logit(theta);
}
model {
  theta ~ beta(1, 1);
  y ~ bernoulli(theta);
}
generated quantities {
  int y_sim = bernoulli_rng(theta);
}
"""

with open('bernoulli.stan', 'w') as f:
  f.write(model_code)
```

```
data = """
{
    "N" : 10,
```

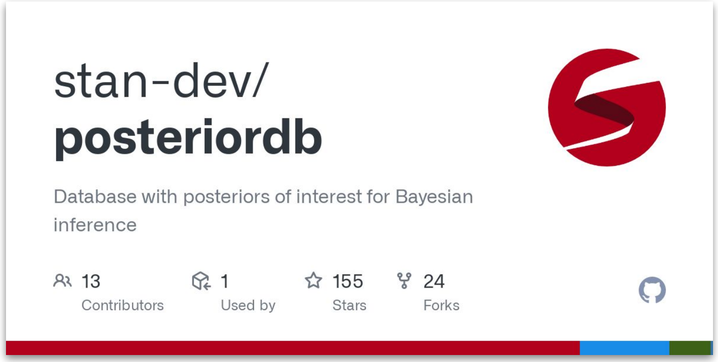
Why you might use BridgeStan

FLATIRON INSTITUTE

Maybe you’re like us

If you are researching algorithms, BridgeStan lets you write them in a language you know, while

- Giving you access to fast, reliable automatic differentiation
- Allowing you to test a variety of existing models quickly
- Enabling comparisons against state-of-the-art algorithms and known posteriors



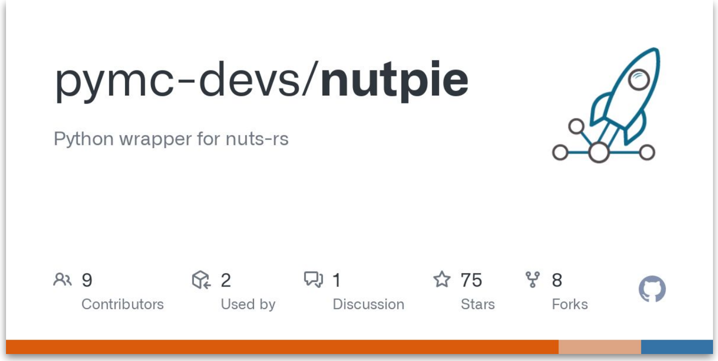
FLATIRON INSTITUTE

Or maybe you’re doing something we never anticipated

BridgeStan also presents a new opportunity for software to use Stan but live outside the Stan C++ bubble.

It has already been used for:

- Plugging Stan into a new NUTS sampler in Rust by the PyMC team
- Using Stan in a distributed sampling package in Julia
- More things showing up in our inboxes monthly



FLATIRON INSTITUTE

How BridgeStan works

A C Interface

The key thing that makes BridgeStan different from tools like RStan is that it *avoids* needing to communicate between Stan and the higher-level language via the C++ binary interface.

Instead, everything is done at a lower level using C's binary interface. This makes a bit more work for the programmer, but gains:

- Portability (Windows *worked the first time we tried it*)
- Language-agnostic code
- Simplicity

```
// - snip -

extern "C" {

  struct stanmodel_struct
  {
    void* model_;
    unsigned int seed_;
  };
```

FLATIRON INSTITUTE

What actually happens under the hood

A Stan model fed into BridgeStan gets wrapped with a simple C API and compiled into a shared library (aka a dynamic link library or DLL).

```
extern "C" {
```

Most languages supply a way to load and call C-like functions in shared libraries. As a result, we avoid needing to write C/C++ that interfaces with the Julia API, or Python API, or …

```
import ctypes
```

```
using Libdl
```

We treat these libraries just like a system library (zlib, etc).

Finally, we provide wrappers around these often low-level tools to open BridgeStan's outputs.

BridgeStan

Thank you.

bward@flatironinstitute.org

https://github.com/roualdes/bridgestan

Roualdes et al., (2023). BridgeStan: Efficient in-memory access to the methods of a Stan model. Journal of Open Source Software, 8(87), 5236, https://doi.org/10.21105/joss.05236

FLATIRON INSTITUTE

BridgeStan

FLATIRON
INSTITUTE