Advanced Stan

Programming, Debugging, Optimizing

Daniel Lee. daniel@stan.fit

StanGroup

Outline

- Morning:
  - - What I'm assuming you know (also sent via email)
    - * Non-centered reparametrization (Matt trick)
    - * parameters on same scale
    - * what Stan program means log p(y | \theta)
  - - Debugging
    - * Basics and philosophy
    - * best practices: git
  - - Constrained vs unconstrained parameters
    - * exercise with Jacobians
    - * exercise with more complex Jacobians
    - * parameters on same scale, in unconstrained space
- Afternoon:
- - Optimization: speed vs time
  - * better to be right and slow than fast and wrong
  - * when someone says "optimization" what that should mean
  - * impact
  - * how to test differences; why one run doesn't cut it
  - * how to debug optimized code
- - How everything is laid out (maybe that's in the morning)
  - * Describe C++ and interfaces
  - * how to build a development version / update versions for RStan, PyStan, CmdStan
  - * how to add a new function
  - * how to debug that function in C++

Outline

Preliminaries

Morning: Stan Language

- typed computation
- understanding what is encoded in the language.
- proportionality constant
- Jacobian

Afternoon: C++

- How everything is laid out
- Digging into C++

am-1

Preliminaries

Why me?

- Who am I?
  - Core Stan dev

- What do I do?
  - Math
  - Stan
  - CmdStan, RStan, PyStan

Getting the most out of today

- I don't know what you don't know
  - I don't know what's difficult to you
  - I live and breathe Stan
  - I speak Andrew-speak
- Ask questions
  - ask me
  - ask your neighbor
  - ask yourself

Getting the most out of today

- Focus on things that are difficult to learn alone
- Goals
  - Read code
  - Work out code
- Exercises
  - Simple exercises focused on one difficulty at a time
  - For more difficulty: think about your statistical problem

You should be comfortable with

- target +=
- parameters must have support over range of the parameters
- Stan types. Arrays vs vectors vs row_vectors
- How to run either:
  - RStan, PyStan, or CmdStan
    (we're not going to focus on this too much)
- non-centered parameterization

Git clone today’s exercises

```
> git clone https://github.com/stangroup/advanced-stan

> cd advanced-stan
```

Exercises

Exercise: am-1

Get 5 programs to translate to C++.

Use the interface of your choice.

- Concepts
  - Compile Stan language to C++
  - Don’t compile C++ program.
    Try to figure this out. If you can’t get this in 5 minutes, ask.

The Stan Compiler

- Compiles Stan language to C++
  - If any valid Stan program does not compile, it’s a bug in the Stan compiler. Please report it.

- One-pass compiler
  - https://en.wikipedia.org/wiki/One-pass_compiler
  - Messages are reported greedily
  - Think back on the compiler messages

Stan Language

- Higher level language for specifying differentiable log density functions
- Turing complete
    - full conditionals and loops
    - functions, recursive functions
    - reassignable local variables and scoping
- Not limited to a graphical model

Stan Language

- Higher level language for specifying differentiable log density functions
- Turing complete
  - full conditionals and loops
  - functions, recursive functions
  - reassignable local variables and scoping
- Not limited to a graphical model
- **Design: inference separate from statistical model**

Specify log joint distribution function

- Data X
- Parameters θ
- Log joint probability distribution function $\log p(\theta, X)$

Usually built as $\log p(\theta, X) = \log p(X) + \log p(X \mid \theta)$

I lied

(it’s more complicated)

Run exercise-3

Run exercise-3

- What is lp__ (in the output)?

Run exercise-3

- What is lp__ (in the output)?
- Why is it not exactly 0?

It's a little more complicated

- Data $$X$$
- Parameters $$\theta$$
- Log joint probability distribution function specified in the Stan language $$\log p(\theta, X)$$

It’s a little more complicated

- Data X
- **Constrained** parameters θ
- Log joint probability distribution function specified in the Stan language is on the **constrained** space $\log p(\theta, X)$

Transforms: Ch 33

- Transforms: one-to-one functions from constrained to unconstrained space

$$q = f(\theta)$$
$$q \in \mathbb{R}^N$$

- The log density we describe is

$$\begin{aligned}
\pi(q) &= \log\left(p(f^{-1}(q), X) \times |\det J_{f^{-1}}(\theta)|\right) \\
&= \log p(f^{-1}(q), X) + \log|\det J_{f^{-1}}(\theta)| \\
&= \log p(\theta, X) + \log|\det J_{f^{-1}}(\theta)|
\end{aligned}$$

Example (sorry, I'm terrible with notation)

$\theta \in \{0, 1\}$

$$f(\theta) = \text{logit}(\theta)$$
$$= \log \frac{\theta}{1-\theta}$$

$$f^{-1}(q) =$$

$\log p(\theta, X) = 0$

$\pi(q) =$

Example

$$\theta \in \{0, 1\}$$

$$\begin{aligned} f(\theta) &= \operatorname{logit}(\theta) \\ &= \log \frac{\theta}{1-\theta} \end{aligned}$$

$$\begin{aligned} f^{-1}(q) &= \operatorname{logit}^{-1}(q) \\ &= \frac{1}{1+\exp(q)} \end{aligned}$$

$$\log p(\theta, X) = 0$$

$$\pi(q) =$$

Example

$$\theta \in \{0, 1\}$$

$$\begin{aligned} f(\theta) &= \operatorname{logit}(\theta) \\ &= \log \frac{\theta}{1-\theta} \end{aligned}$$

$$\begin{aligned} f^{-1}(q) &= \operatorname{logit}^{-1}(q) \\ &= \frac{1}{1+\exp(q)} \end{aligned}$$

$$\log p(\theta, X) = 0$$

$$\begin{aligned} \pi(q) &= \log p(f^{-1}(q), X) + \log |\det J_{f^{-1}}(q)| \\ &= 0 + \log |\det J_{f^{-1}}(q)| \\ &= \end{aligned}$$

Example

$\theta \in \{0, 1\}$

$$\begin{aligned} f(\theta) &= \text{logit}(\theta) \\ &= \log \frac{\theta}{1-\theta} \end{aligned}$$

$$\begin{aligned} f^{-1}(q) &= \text{logit}^{-1}(q) \\ &= \frac{1}{1+\exp(q)} \end{aligned}$$

$$\log p(\theta, X) = 0$$

$$\begin{aligned} \pi(q) &= \log p(f^{-1}(q), X) + \log \, |\det J_{f^{-1}}(q)| \\ &= 0 + \log \, |\det J_{f^{-1}}(q)| \\ &= \log \, |\frac{d}{dq}\text{logit}^{-1}(q)| = \log \, |\text{logit}^{-1}(q) \cdot (1 - \text{logit}^{-1}(q))| \end{aligned}$$

What's wrong with exercise 5?

Can you fix it? (if you're fast, try multivariate)

- What's missing?

Can you fix it?

- What’s missing?
- What’s the unconstrained variable that was transformed?
- What’s the log determinant of the Jacobian of the inverse transform?

Can you fix it?

- What’s missing?
- What’s the unconstrained variable that was transformed?
- What’s the log determinant of the Jacobian of the inverse transform?

```
target += log_beta;
```

Stan works on the unconstrained space

Automatic transforms are important

- Start thinking in the **unconstrained** space
- Hamiltonian system is constructed with $\pi(q)$
- The "nicer" the unconstrained space, the easier it is for Stan's MCMC algorithms

- Difficulties
  - not always intuitive
  - underflow and overflow

Custom transforms may be more efficient

- Stan development team hasn't't experimented too much
- Avoid underflow and overflow where you can

am-2

Setup and Best Practices

Treat Stan Programs as Code

- Version Control
  - git is the new SVN (is the new new CVS)
  - any version control is better than no version control
- When to commit?
  - any time you have something working
  - commits are cheap
- Why?
  - backup
  - collaboration

git commit now!

```
> git add am-1
> git commit -m "Exercises complete"
```

Debugging

- Something is wrong.
- Try to find the last state that it was right.
- Assume everything, even simple things, can be incorrect.

- In Stan programs (won't get too much into this)
  - print()
  - reject()

Let's talk speed

(I'm assuming MCMC)

I'm sensitive to this topic

Terminology is important

- **wall time**
  time it takes from start to finish
- **(code) optimization**
  making something faster w/o changing behavior
- **vectorization**
  generalizing an operation to scalars that apply to vectors, row vectors, or arrays
- **draws** or **iterations**
  number of iterations the MCMC algorithm runs
- **a sample**
  collection of draws
- **effective sample size**
  estimate for the number of independent draws that a sample contains

optimization is worthless if your model doesn't converge

code optimization

What's changes? Does the **behavior** change?

```
for (n in 1:1000)
   target += log(theta);
```

vs.

```
target += 1000 * log(theta);
```

How much do you benefit from something like this?

vectorization

What's the benefit? Is it **faster**?

```
vector[N] theta;
…
vector[N] log_theta;
for (n in 1:N)
    log_theta[n] = log(theta[n]);
```

vs.

```
vector[N] theta;
…
vector[N] log_theta = log(theta);
```

draws / iterations vs effective sample size

What's the difference?

- n_eff <= draws by construction
- what's important for inference?
  - MH is very, very fast for iterations
    (I can construct something faster, but less n_eff)
  - how many draws do you need?
    (trick question)
  - how many n_eff do you need?

What matters?

What matters?

- In general
  - **n_eff / wall time** or **wall time / n_eff**
- In practice: it depends
  - What do you value?
  - What do you do?
  - How much time do you have?
- Most people worry about **wall time**.
  Your first concern should be **n_eff**.

When does wall time matter?

- when it's prohibitive to develop your model
- limited time situations (production)

- When does wall time **not** matter?
  - when model doesn't converge
  - when wall time isn't large enough where it matters
  - heuristic: when n_eff is less than 20% of iterations

when should I optimize code?

- Code optimization often, but not always
  - destroys readability
  - makes the code riskier
  - makes it harder to iterate on the model

- Optimize as a last step.
  Expect gains of 1 - 20% in n_eff / wall time.
  What changes here?

- I value human time, so "losing" 20% isn't bad for my use cases

can I vectorize x?

- Vectorization doesn't speed up anything by itself
- It makes the code more readable

- When is it faster?
  When the devs have done smart things to save repeated computation.
  We'll take a look in the afternoon.

what should I be asking instead?

correct parameterization!

- Optimizing code will improve wall time by a fixed percentage.
  It **will not** affect n_eff.
- A correct parameterization will
  - boost **n_eff**.
    Depending on the model, a lot. 10x is common.
  - decrease wall time.
    2x is common.

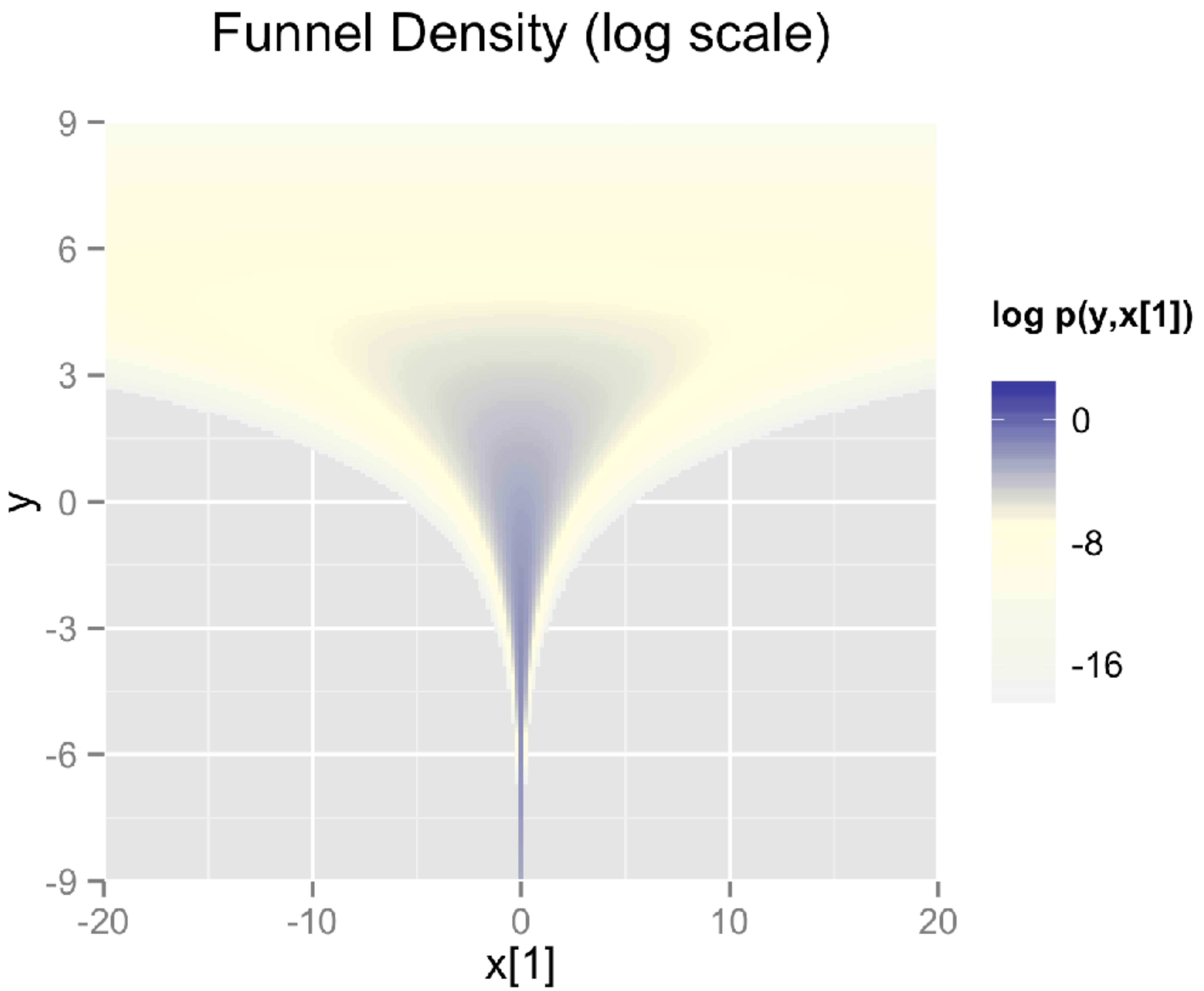
Example: Funnel

- See am-2/funnel.stan

$$y \in \mathbb{R}$$

$$x \in \mathbb{R}^9$$

$$p(y, x) = \text{Normal}(y|0, 3) \times \prod_{n=1}^{9} \text{Normal}(x_n|0, \exp(y/2))$$



Let's code optimize (why shouldn't you do this?)

- Timing is difficult
- What's the variability run-to-run?
  - n_eff: 20 — 100 for y
  - time between 0.1 to 0.6 secs
- Yikes!

- When you code optimize, all behavior should stay exactly the same except the wall time.

Let's code optimize

- We need it to be repeatable. Set seed. Verify it's repeatable.
- Extract draws and set aside.
- Record times. Since there's variability, need more than 1 run.
  (are you surprised at how much variability there is with the same seed?)
- Change the code to be faster.
- Run again.
- Always verify that you're getting the same behavior.

Let's code optimize

- After you're certain you're getting the same values, start timing.
- If you're not, what's going on?
  (I don't get the same)
- If you can't get the same, it's not comparing apples to apples.

Example: non-centered funnel

- This is the correct way to deal with this particular problem.
- Write the model.
- Compare n_eff / time.

- Note: divergences are not something you should ignore.

parameters should be same order of magnitude

What does that mean?

- unconstrained parameters should be the same order of magnitude
  - **not** constrained parameters
- It's better if everything is on unit scale (order 1).
  - it helps with warmup.
    More time is spent in warmup than in the sampling stage

Example: am-2/scales.stan

- Run it
- Reparameterize mu1 so it's the same scale as mu2.
  See what happens.
- Reparameterize both so it's on the unit scale.
  See what happens.

keeping track of leapfrog steps

(if there's time)

pm

How all the pieces are laid out


Interfaces
{Cmd, R, Py}Stan


Interfaces
{Cmd, R, Py}Stan
Stan
C++ API, language, algorithms


Interfaces
{Cmd, R, Py}Stan
Stan
C++ API, language, algorithms
Math
functions, automatic differentiation

Where all the pieces live

all of this is github.com/

- Interfaces
  - CmdStan: stan-dev/cmdstan
  - RStan: stan-dev/rstan
  - PyStan: stan-dev/pystan
- Stan: stan-dev/stan
  - C++ API
  - Language
  - Algorithms
- Math: stan-dev/math
  - Functions
  - Automatic differentiation

Build Interfaces

- Look at different markdown files

Switch to cloned CmdStan directory

Get into the Stan repository; run a unit test

```
> cd stan
> ./runTests.py src/test/unit/version_test.cpp
```

Windows:

```
> cd stan
> .\runTests.py src/test/unit/version_test.cpp
```

Windows and no Python:

```
> cd stan
> make src/test/unit/version_test.exe
> src\test\unit\version_test.exe
```

What's in the test?

```
#include <stan/version.hpp>
#include <gtest/gtest.h>

TEST(Stan, macro) {
  EXPECT_EQ(2, STAN_MAJOR);
  EXPECT_EQ(14, STAN_MINOR);
  EXPECT_EQ(0, STAN_PATCH);
}

TEST(Stan, version) {
  EXPECT_EQ("2", stan::MAJOR_VERSION);
  EXPECT_EQ("14", stan::MINOR_VERSION);
  EXPECT_EQ("0", stan::PATCH_VERSION);
}
```

what's a unit test?

Unit test

- https://en.wikipedia.org/wiki/Unit_testing
- A small piece of code to repeatably test some functionality
- MCMC is a stochastic algorithm
  - computation with a computer is not truly stochastic

Google Test

- Stan (CmdStan, Stan, Math) uses Google Test
  https://github.com/google/googletest
- Expectation framework.
  - EXPECT_FLOAT_EQ(val1, val2);
  - EXPECT_EQ(val1, val2); _NE, _LT, _LE, _GT, _GE

Run the simple test

```
> cp pm/easy_test.cpp <cmdstan-folder>/stan/src/test
> cd <cmdstan-folder>/stan
> ./runTests.py src/test/easy_test.cpp
```

Add a test for the log of the normal distribution

- Add a new TEST(easy, normal) that computes the lpdf of the normal distribution (if you don't remember, find it online)

autodiff

Stan's automatic differentiation

- The Math library
  - provides math and stats functions
  - provides automatic differentiation
- How does reverse-mode automatic differentiation work?
  - Not enough time to do it justice
  - simple: wikipedia
  - complete: https://arxiv.org/abs/1509.07164

Let's just dig into it

```
TEST(easy, rev) {
  stan::math::var x = 2.0;
  stan::math::var y = exp(x);

  y.grad();
  EXPECT_FLOAT_EQ(7.389056, y.val());
  EXPECT_FLOAT_EQ(7.389056, x.adj());

  stan::math::recover_memory();
}
```

Try a gradient more complex

How does this connect to Stan?

- Stan maps data and transformed data to **double**
- Stan maps parameters and transformed parameters to **stan::math::var**

- There's promotion from double to stan::math::var, but no way to go back down

Write a function for the normal lpdf

- Write a function for the computation of normal lpdf.
- What is the function signature?
- Hint: if no autodiff

```
double foo(double y, double mu, double sigma) {
  double lpdf = 0;
  …
  return lpdf;
}
```

- Test it.

Writing your own function

- For your function to make it to the language, you need to have it templated (I can't get into it, but you should be able to read this)

```
template <class T_y, class T_mu, class T_sigma>
typename stan::return_type<T_y, T_mu, T_sigma>::type
foo(T_y y, T_mu mu, T_sigma sigma) {
  stan::return_type<T_y, T_mu, T_sigma>::type lpdf = 0;
  …
  return lpdf;
}
```

Test the function with different types

- This function can take 8 different types of calls. 2^3
- Why is this useful?

- Let's expose your function to the language

Adding the function to the language

- Move the function to something that can be included.
  For now, let's just put it in <cmdstan>/stan/src/stan/foo.hpp
- Open up <cmdstan>/stan/src/stan/model/model_header.hpp and add include to foo.hpp
- Open up <cmdstan>/stan/src/stan/lang/function_signatures.h
  Add this line

```
add("foo", DOUBLE_T, DOUBLE_T, DOUBLE_T, DOUBLE_T);
```

- Rebuild CmdStan
- Write model with "foo"

Custom gradients

- Write a template specialization using operands_and_partials.

What else do you want to know?

Thank you!

Please stay in touch: daniel@stan.fit