A Path Forward for Stan

Sean Talts

ABSTRACT

Stan has proven success as a statistical modeling language for tens of thousands of scientists across the globe. Since its design in 2011, the landscape of statistical computing has shifted such that there are now new opportunities for Stan to achieve better performance and encourage new user-developers. There are a few paths forward, but I'll describe just one particularly enticing one that would take some major investment over the next few years. There are a few areas that need improvement, but the keystone of this vision I call "Stan in Stan" – extending Stan to support writing most of the Math library in the Stan language itself.

1 Introduction

I've worked on the Stan project for the past 3 years as a software engineer, and for the past year or so as its Technical Working Group Director. I'm going back to industry full-time at the start of February and while I still hope to contribute in my spare time, Andrew Gelman asked me to write up what I thought the path forward for the Stan software ecosystem should look like. I don't claim ownership of these ideas, as they've all evolved through countless conversations with others on the Stan project. And please keep in mind that even though supporting evidence exists for many of these and I have worked on other tools in this category, I will be offering few citations and this is just, like, my opinion, man[1].

Stan has proven successful as a statistical modeling language for tens of thousands of practicing scientists, but the landscape for this category of tooling has changed considerably since the time when Stan was first designed. New hardware, larger data sets, and users less willing to hand-tune performance have all appeared in the scene and changed the calculus around some of the original architectural choices. In order for Stan to stay relevant, we need to introduce several major technical changes and adapt some of the revelations from surrounding fields in order to adapt.

Stan has been so successful in large part because of the huge amount of effort that has gone into the pedagogy, methodology research, broader ecosystem packages like those for visualization, and most importantly its amazingly supportive community. But here I will be focusing on a fairly high-level architectural view of the software artifacts and end-user experience around using the core tools, just because that's more in my wheelhouse.

My framework for thinking about this design space is pluralistic, with two separate goals that are difficult to unify. The first could be summarized as a shift towards optimizing the time-to-first-sample of a new user. This includes the time it takes to install Stan and a compatible C++ toolchain, read chunks of the manual, write and compile a model, find bugs or typos, and draw the first sample and display it on screen. I think this metric best captures what we care about because I believe in the common wisdom about software development: much more of your time is spent debugging and iterating on programs than running them. Previously, we often acted as if we cared only about optimizing the time it takes to do a single log probability density (`log_prob`) method evaluation and gradient. This is a very useful first-order metric but ultimately myopic if we want to continue to be the fastest Bayesian modeling framework into the next five years. Eventually, we'll likely want to maximize the time-to-best-fit, which includes time-to-first-sample as well as time to correct the model and fit the correct model fully, but this includes various pieces that are hard to measure. Baby steps.

Separately, I believe we should think more explicitly about a specific aspect of sustainability: what can we do to entice more of Stan's users into contributing back to Stan. Open source seems to work best when a project's developers are also its users for a variety of reasons - passions and needs align; expertise can flow more easily into the project and its code; language, API, and user interface design is done by those who will end up using it. There are some areas of Stan that will require non-statistical expertise, but with good tooling and some architectural changes we can reduce that surface area and create an easy on-ramp for all users to become developers if they so desire.

I'll first talk about Stan's current architecture to give the necessary context while trying to describe problems we currently have in those areas and roughly what we'd like to be able to do. Then I'll step through a few large initiatives that I think would help alleviate many of these issues and seriously boost performance on our two metrics by easing distribution, propagating information where it's needed, and allowing our users to be much more involved in contributing to Stan.

2 Stan's Current Architecture - Modules and Life Cycle

Core Stan can be thought of as four major pieces: a math library that provides numerical functions and their gradients, a compiler that emits a C++ model which uses some of those functions, a set of algorithms (and other services) that operate on

a model, and an interface to the other three. In "core" Stan our interface reference implementation is CmdStan, which is a command-line utility that can go from a Stan model and data file to a set of posterior draws and run various diagnostics on those draws.

The execution of a Stan program has a life cycle as well that corresponds fairly well to those pieces. We must first compile a Stan model into a program, then load any necessary data into the program, then ask the program to perform inference given the model and data, and finally spit out the results in some form. A Stan model has methods that correspond to these phases: the constructor loads data, `log_prob` is called many many times during inference, and `write_array` is called fewer times during inference to write samples out in a streaming fashion. As an example, we might expect inference on a model with HMC to compile once, load data once, run `log_prob` a hundred thousand times, and run `write_array` ten thousand times. This entire life cycle occurs once every time we change the model even slightly, which we might expect to do dozens or even hundreds of times during model development.

These pieces are currently fairly encapsulated from each other in some areas – the compiler takes a Stan program and translates it into a series of atomic Math function calls. No information from the Math function calls is visible to the compiler, and no information about the program or its life cycle is visible to the Math library. This seems like a reasonable separation of concerns prima facie, but we could improve performance if either module knew all of the relevant information from the other. The compiler could, for example, hoist expensive data matrix checks to the constructor if it knew which checks needed to occur on which matrices, but that information is buried opaquely in the Math library. Likewise, if the Math library had a notion of life cycle, we would be able to cache data or results that could be useful in later iterations - examples like GPU data transfer and iterative solver initial guesses come to mind.

Encapsulation is not strong enough in other areas. Interfaces like RStan and PyStan rely on various internal Stan compiler source files and dynamically link a running R or Python interpreter against a compiled model in order to execute it. Allowing external packages to peer into the compiler source code makes it difficult to refactor the compiler code base. Dynamic linking ties Stan's fate with that of R or Python such that a crash brings down the host language as well, and it holds us back a few C++ generations in the 57,000 lines of code we maintain in the Math library. Distribution is brittle in this model; for example, the user needs to have the toolchain used to compile their interface language installed correctly as the default toolchain as a first step. As a mostly outside observer, it seems like RStan has a particularly hard time with CRAN partially because of this choice such that releases usually follow Stan releases by a few months for the past couple of years.

3 New Hardware

CPU core clock speeds haven't gotten much faster in the past 10 years or so - performance gains have come from exploiting parallel hardware in the form of multi-core CPUs and massively SIMD GPUs. Computational frameworks for exploiting these forms of parallelsim have developed mostly separately, though recent efforts in TensorFlow and PyTorch have attempted unifying these resources from the programmer's perspective. I'll start with unrealistically brief computational models here and talk about what I think is the most exciting path forward.

Multi-core CPU parallelism is easier to understand of the two of them - you just have more, equally capable and accessible processors available for general purpose computation. This means that you can hypothetically pick apart your program any way you wish to parallelize it, but in practice there are communication costs and complications between cores that mean you have to be very careful how and when you do this. Surprise is common when someone attempts to perform a workload in parallel and sees a slowdown, but this just comes from an incomplete model of the computational substrate. The most clear wins with the least confusing programming model arise from what is sometimes called Single Instruction, Multiple Data (SIMD) workloads - you have a lot of data and want to do the same thing to all of it with no cross-communication between pieces of the workload. For the statistically minded, think i.i.d. sub-workloads. There are other models for taking advantage of parallelism but I don't see it being very likely that we want to get into any of those in the next five years, so I'll skip them here.

I think there is probably just one main paradigm that we'd want to expose to our users, and that is that of parallel map, fold, and filter (possibly with unordered versions). A library like Clojure's transducers[2] takes these transformations and makes them first class, allowing you to compose computations that, when eventually applied to data, will perform all of these operations in parallel in a smart way. We can get the benefits of that style of computation without making the user aware of it because we have access to the compiler internals and can do our own fusion there.

The Intel TBB is a great recent addition that we should be outsourcing to as much as possible. Its model seems to be setting up one worker per core with lightweight tasks that get queued up to each worker, while allowing each worker to "steal" work from another if it runs out of tasks ("workstealing" for short, I think). This is a reasonably low-level primitive but should be easy to build everything on top of this that we need, at least for workloads running on a single machine.

The addition of multiple cores is just one way in which computing hardware has changed - now we can also make use of

fairly wide vectorized [1] floating point math on CPUs (something we hope Eigen does for us, but it may not[3]). For parallelism across many machines, I think MPI is a pretty decent workhorse for our situation. I think the next generation technology here probably looks something like TensorFlow, which lets you build a graph (with autodiff) and distribute it across machines. Speaking of TensorFlow, one of the biggest hardware advances in the past decade has been in General Purpose Graphics Processing Units, or GPUs. Thanks to Steve, Rok, Tadej, and Erik we've seen a lot of advances and interesting work done in that space in Stan. I think our GPU story is somewhat handicapped by our general Stan architecture that means we don't really know about the Stan lifecycle or language-level constructs when we're e.g. compiling GPU code, choosing which GPU functions to run, or figuring out how to best perform data transfers. These problems are getting addressed somewhat by doing some of those tasks in the compiler, a trend I expect to continue.

4 Stan-in-Stan

"Stan-in-Stan" is just the idea that we should be able to write almost all of our Math library in the Stan language itself with just a few extensions. Just to be clear, it's my goal here to only rewrite those sections that retain nearly 100% of their current C++ Stan Math performance; I think we can do this without much sacrifice with a little careful engineering. I'll detail some of the work that I think is necessary for this and then talk about why I think it's better than sliced bread.

4.1 Necessary work

We could cover something like 80-90% of the Math library if we added just three concepts to the Stan language: user-defined gradients, type traits, and vector views. We might have to come back later for more complicated parts of the Math library like our ODE solvers and certain kinds of iterative algorithms, though that would be mitigated if we also came up with an easy and serious Foreign Function Interface 6.3 story for Stan.

4.2 User-defined gradients

User-defined gradients just means allowing someone writing a Stan function to provide Stan with the analytic gradient of that function. There are myriad proposals for the specific syntax for how to do this in this thread[4], and new ideas should also be posted to that thread). I think I'm in favor of a syntax that works really easily to define most of our Math library in a single function encomposing both value and gradient calculation, perhaps with an escape hatch similar to our current FFI 6.3 system where we can just declare a function to be the gradient of another function. I don't actually think there are currently many objective reasons to prefer any of the various syntax options w.r.t. the capabilities of the compiler to optimize.

One could also imagine at some point in the future using symbolic differentiation[5] to generate a printable first pass at an analytic gradient for many simple Math functions. Symbolic differentiation for functions that do not branch on parameters can be done once at the start of the program during the stanc3 compilation process, so for long-running models even a non-trivial upfront cost there could be worthwhile if you don't have a mathematician handy.

4.3 Type traits

Type traits and vector views are somewhat poorly named, but I think that's how many folks on the project might think of these ideas. The names refer to the C++ ideas that provide similar functionality.

Type traits allow C++ programmers to choose at compile-time which code should be executed based on the compile-time types of particular variables that are passed in. This is of particular interest to us because we write a lot of our Stan Math code such that it is abstract over whether or not a particular function argument can be autodiffed through or not, and we use C++ type traits to do this in a way that doesn't lose performance in either case w.r.t. separate hand-written functions for each scenario. In Stan we have a simpler use-case - we really just want to know what code depends on what arguments being autodiffable or not, and we will also know at Stan compile time which variables are or are not autodiffable. Using dataflow analysis techniques like those showcased by Maria Gorinova[6] and Ryan Bernstein[7], we should be able to take Stan statements and expressions written in their normal style and determine whether or not those particular statements need to be executed or not depending on the type of the variable. For example, if we're computing the normal distribution of an autodiffable mu and static sigma, we know that we don't need to add any terms to the derivative to account for sigma, so we can ellide those computations entirely.

For Stan-in-Stan to match the C++ Math library's performance, we just need to make sure the superfluous computations are not performed. I actually suspect that for most cases the compiler can just perform this analysis automatically, but there may be cases evenetually where we'd like to present the user with an escape hatch. There are no existing proposals for this but it will couple very tightly with the user-defined gradient syntax, since that will be how the compiler figures out which computations are part of the derivative and which are part of the value of the function.

[1] There are at least two different meanings of "vectorization" we care about - one is at the level of the Stan language, when we trade for loops for function calls on arrays or vectors of arguments, and the other is at the CPU level where we can perform some common SIMD operations in parallel if the code is written in a certain way.

4.4 Vector views

Vector views refers to some C++ tools we have in the Stan Math library that allow us to write code as if we were operating over vectors even when we actually are passed scalars without losing performance in either case. This requires a bit of C++ template metaprogramming to get right in C++; in Stan we can simply piggyback on that C++ code when generating C++. There may also be some more elegant way to just generate the correct code based on the type in stanc3, but I think my first attempt would be to just generate code that uses C++ VectorViews under the hood.

4.5 New possibilities

Once users can write performant Math library functions in Stan itself, a new world of possibilities opens up. First, I think this is a huge win from the maintenance perspective - even if we could only rewrite all probability density functions in the Stan language and delete the corresponding Math library code, I would estimate something like a 3x code reduction along with making it much easier for folks to write their own performant distributions and even contribute them back upstream to Stan. We don't need to do a full 100% rewrite of Stan Math (or anything close!) to reap the benefits for whatever parts we do manage to convert, making starting on the conversion process much more attractive and safe to perform iteratively and piecemeal. Standardizing the current boilerplate we write in order to write efficient C++ Math functions such that the compiler can actually write it for you will seriously reduce bugs, complexity, and maintenance costs. We should even be able to write or generate tests much more easily using stanc3 instead of C++ variadic template metaprogramming.

Secondly, this will really empower users of Stan to become the developers of Stan. Right now the hurdles for contribution are mighty high especially to the Math library, and many of those hurdles relate to the difficulties associated with learning, programming, debugging, testing, and distributing C++. Countless times I have had folks ask me how to contribute some Stan function they've written back to Stan only to be told they must rewrite it in C++ using the aforementioned VectorViews and various other template metaprograms.

A major turning point for the iPhone (and several other products) was the introduction of an App Store. The first step is figuring out how to frame your standard tools in that framework and then open the framework up to users to write their own. Once we have a Stan-dard library (hyuk hyuk) of math functions that we distribute, all written in Stan, it should become much easier and more enticing for someone to create a Stan package system that wraps up Stan code into packages that can be imported in a similar way. I think the first Stan package system, even if it's just a quick hack on top of github repositories, will be a change-point of similar magnitude in the Stan story.

One way to leverage all of the exciting work around us in the probabilistic programming field is to simply create a compiler that makes it easy to generate code that runs on the frameworks, architectures, and languages that would otherwise have become competitors. Some systems, like TensorFlow Probability, are working towards coverage of essentially all of the same Math functionality that we have, and all that's required to translate to those systems is a reasonable code-generation system [2]. There are other possible backends, however, that have very little Math library to speak of; once we have our Math library defined in Stan itself, generating code for those backends becomes much much easier and our users are assured that the math will behave the same way across backends.

5 Seamless Distribution

Losing users because they can't actually run our software is stupid [3]. There are a few things we could do to make the process of getting to the point where a user can interactively run Stan code they've written much simpler and robust.

5.1 One statically-linked driver executable per OS

Ideally, users would have the option to install a single executable that contains everything needed to compile and run a model, eliminating the number one install hurdle we face. One way to provide this would be to simply include all of our dependencies; namely archive or link against Clang and libc++. This has two other benefits: we can stop testing other compilers and other versions and target a single known version of clang, and we can use the latest C++17 (and beyond) in our Math library to aid developer productivity. Linking against a clang interpreter would seem like it could add ∼300MB to our 50MB CmdStan release, which is a major increase but doesn't seem like a phase shift.

With one program controlling and executing the entire compile+run process, we can do interesting work around incremental compilation (e.g. we could have `log_prob` and `write_array` compiled in separate translation units and only recompile one of them when it changes) and compile both an optimized and unoptimized Stan model into the same program, allowing us to dynamically switch between the two when an error occurs. This would mean we could turn on the optimizer by default and get all of those performance gains, yet fall back to a more cautious and exhaustively checked version of the model with good

[2] Though there are definitely additional difficulties in attempting to achieve similar performance as their programming paradigms can be fairly dissimilar.

[3] Unless we believe there's some correlation between people who are good at e.g. Linux package management and those who are doing good statistics. I claim there isn't.

error messages only when needed. We could also enable much easier Foreign Function integration (see 6.3) and reduce issues we've had between the various versions of `make` and their interactions with Windows paths, etc.

I believe RStan and PyStan are already built like this; we just need to move CmdStan away from a pure `make`-based workflow into something a little more customizable and batteries-included.

5.2 CmdStan server-mode / ServerStan

We'd like to avoid dynamic linking and bring more of the work of creating an interface into our reference implementation without a loss in performance. To this end it makes sense to extend Stan programs with a server-mode that can listen for commands. I think a simple version that would satisfy all of our needs here would be to add another command-line option to a CmdStan-produced Stan program that reads in the data and then starts a server listening for additional commands over a socket. This would allow us to run `log_prob` and `grad_log_prob` without re-reading the data each time and get us close to the performance of the dynamic linking approach employed by RStan and PyStan, and a crash wouldn't bring down your R or Python session if it's just connected over a socket.

6 Auxiliary Ideas

These ideas are somewhat orthogonal to the main ideas and not quite as imperative in my mind, but they're still really cool so I'll write about a few of them here while they're fresh.

6.1 Pedantic mode

Ryan Bernstein has been working on pedantic mode ([8],[9]), which essentially warns users when we estimate that they're doing something statistically or semantically unsound. This covers simpler things like checking that all parameters contribute to the calculation of the target density as well as more complex heuristic analyses like checking that bounded parameters line up well with distribution function support. See those posts for more details.

6.2 Model composition

We've talked quite a bit about various ways to allow for model composition. Maria Gorinova's first foray into the Stan world was with the SlicStan paper[6] intended to address exactly this topic. I think ultimately this would be a nice goal to achieve to make it easier to share your work and encourage the growth of an ecosystem of code written in Stan. If the current Stan language mostly lets users talk about functions and we all agree we would also like to add derivatives [4], the next logical step would be to let users write about models in Stan. Of course, a model is just a distribution function, so one way to address composition would be to provide semantics for importing one Stan file and accessing its model block from within another Stan file [5].

6.3 Foreign function interface (FFI)

Stan actually already comes with a poorly-advertised FFI capability in that you can declare functions in Stan without defining them and then, during compilation, compile against C or C++ code or libraries that implement those functions. I think for this to be really usable we need two main components. First, we need to be able to drive the entire lifecycle from a single executable (e.g. make the Stan compiler stanc3 responsible for calling the C++ compiler rather than using `make`). A little automation here will go a long way in making this a usable feature from all of our interfaces. Secondly, we really need the ability to specify the user-defined gradient of a foreign function (Section 4.2).

7 Conclusion

I think Stan is in a really exciting place as a project with a great user-base, community, and cutting edge methodological research that could use a few architectural shifts to really adapt the software artifact both to become something users can maintain and improve as well as something that can really take advantage of all of the various underlying backends and hardware that have become really popular in the past few years. These kinds of architectural improvements can often feel a little bit like Henry Ford offering a car to folks who wanted a faster carriage – or like offering Google Glass to a public that wasn't interested. It can be a bit hard to tell in advance. But we've already had a bunch of interesting successes in the stanc3 project. Really exciting work by Rok and Tadej has the Stan compiler automatically computing which data to send to the GPU during model construction rather than sending it every leapfrog iteration, something that we were able to get really great heuristics for up and running very quickly that improved some model times by an order of magnitude. Ryan and Matthijs have coded up a large

[4] Some folks would really like to be able to access the derivatives of functions in a first-class way within Stan as well.

[5] E.g. if we have submodel.stan, in model.stan we could refer to it by writing something like `import submodel; ... y ~ submodel(mu, sigma);`

textbook optimization framework for stanc3, at least one improvement of which is always on now and gave something like 5% improvements to a bunch of models; the rest just need some quality time being debugged before they generate compilable code for all models. I think reifying the Stan program in an AST that is easy to manipulate in OCaml lead to both vanilla optimizations and Stan lifecycle-aware ones, and these would only be amplified if we brought the part of the Stan program that lives opaquely inside the C++ Math library into the stanc3 compiler so we could operate on that. We've shown some proof-of-concept for translating to Tensorflow Probability; that would be made much easier if we had code for much of our math library written in Stan itself.

All told, it's been a great 3 years and I really want to thank everyone for the countless hours of help and hard work folks have contributed. Thank you all!

References

1. Brothers, T. C. The big lebowski (1998).
2. Fränkel, N. Learning clojure transducers. *blog.frankel.ch* (2018). https://blog.frankel.ch/learning-clojure/7/.
3. Team, T. S. D. Stan simd performance. *The Stan Discourse* (2019). https://discourse.mc-stan.org/t/stan-simd-performance/10488.
4. Team, T. S. D. Soliciting syntax ideas for user defined gradients and user defined transformations. *The Stan Discourse* (2019). https://discourse.mc-stan.org/t/soliciting-syntax-ideas-for-user-defined-gradients-and-user-defined-transformations/11125.
5. Team, T. S. D. Ocaml & symbolic differentiation? *The Stan Discourse* (2019). https://discourse.mc-stan.org/t/ocaml-symbolic-differentiation/7511.
6. Gorinova, M. I., Gordon, A. D. & Sutton, C. A. Probabilistic programming with densities in slicstan: Efficient, flexible and deterministic. *CoRR* **abs/1811.00890** (2018). http://arxiv.org/abs/1811.00890.
7. Bernstein, R. Static analysis for probabilistic programs. *ArXiv* **abs/1909.05076** (2019). https://arxiv.org/abs/1909.05076.
8. Team, T. S. D. 'pedantic mode' wishlist. *The Stan Discourse* (2019). https://discourse.mc-stan.org/t/pedantic-mode-wishlist/11737.
9. Bernstein, R. Pedantic mode pull request. *The Stan Github* (2020). https://github.com/stan-dev/stanc3/pull/434.