Lambdas, tuples, ragged arrays, and complex numbers in Stan

BOB CARPENTER, Flatiron Institute, New York City

We are introducing four new language features for Stan: complex numbers, ragged arrays, tuples, and lambdas. There will be basic types for complex scalars, vectors, and matrices, backed by arithmetic and special-function support in the Stan math library. Ragged arrays will be homogeneous containers like Stan's existing arrays. Tuples provide a heterogeneous container type; adding names gives us structs. Simple function types enable us to deal with type inference for functions. We will provide lambdas with implicit binding by value, which can be directly implemented via closures.

1 STAN, A LANGUAGE FOR STATISTICAL MODELS

Stan was designed with the goal of making it easy for applied statisticians to express statistical models [Carpenter et al. 2017; Stan Development Team 2021b,c]. From an evolutionary perspective, Stan was derived by generalizing the syntax of the first probabilistic programming language, the Bayesian Inference Using Gibbs Sampling (BUGS) system, to allow for type declarations, local variables, and declarations of variable intent (data vs. parameter vs. derived quantity) [Gilks et al. 1994; Lunn et al. 2012]. In its compilation to a differentiable log density function, it is more like the other first-generation probabilistic programming language, the Automatic Differentiation Model Builder (ADMB) [Fournier et al. 2012].

Another primary design goal is to have programs be readable. In Stan, a simple linear regression can be coded as follows.

```
data { int<lower = 0> N;  int<lower = 1> K;  matrix[N, K] x; vector[N] y; }
parameters { vector[K] beta;  real<lower = 0> sigma; }
model { beta ~ normal(0, 1);  sigma ~ lognormal(0, 1);  y ~ normal(x * beta, sigma); }
```

Like MATLAB [Higham and Higham 2016], Stan is designed around standard mathematical types, operators, and functions, with the addition of sampling notation for concisely expressing log density contributions to the target density. Unlike other PPLs and scripting languages, Stan is strongly statically typed as seen in the declarations for the variables above.

Semantically, a Stan program defines a function from data to a differentiable function from parameters to a log density. The regression program above denotes the curried log density function f defined by

$$f(N, K, x, y)(\beta, \sigma) = \log p(\beta, \sigma \mid y, x, N, K) + \text{const},$$

where the constant does not depend on the parameters β and σ. It does not matter how the log posterior is coded; usually the joint log density is implemented, as in this program, which codes $\log p(\beta, \sigma, y \mid x, N, K)$. Bayes's rule tells us that $p(\beta, \sigma \mid y, x, N, K) \propto p(\beta, \sigma, y \mid x, N, K)$.

The data block defines the signature of the first argument, the parameters block defines the signature of the second argument, and the model block defines the value of the program by incrementing a target log density. The result $f(N, K, x, y)$ represents the posterior log density function up to an additive constant.

To support state-of-the-art inference algorithms such as the no-U-turn form of adaptive Hamiltonian Monte Carlo sampling [Hoffman and Gelman 2014], automatic differentiation variational inference [Kucukelbir et al. 2017], and quasi-Newton optimization [Liu and Nocedal 1989], Stan

Author's address: Bob Carpenter, Flatiron Institute, New York City, bcarpenter@flatironinstitute.org.

implements gradients of the posterior log density function, $\nabla f(N, K, x, y)$ using automatic differentiation [Carpenter et al. 2015].[1] Although the basis of Stan is differentiability, its variables behave like random variables under Bayesian inference. Most importantly, expectations

$$\mathbb{E}[f(\beta) \mid y, x, N, K] \approx \frac{1}{M} \sum_{m=1}^{M} f(\beta^{(m)}, \sigma^{(m)}),$$

have plug-in estimates using Markov chain Monte Carlo draws, where each

$$\beta^{(m)}, \sigma^{(m)} \sim p(\beta, \sigma \mid y, x, N, K)$$

is a (not necessarily independent) draw from the posterior [Roberts and Rosenthal 1998]. Expectations are used for Bayesian parameter and variance estimates, event probability estimates, and posterior predictive inference [Gelman et al. 2013]. Quantiles are similarly preserved.

2 TUPLES

Tuples provide a syntactic representation of product types. They provide heterogeneous containers of fixed size and fixed element type [Pierce 2002]. Tuple type notation and construction are exemplified in

```
(int, array[2] real) y = (3, {1.2, -5.3});
```

where `{1.2, -5.3}` is the constructor for a real array of sized type `array[2] real`. Note that we are introducing a new type language for arrays. The current syntax for an array declaration, `real a[2];` is going to be replaced with `array[2] real a;`. This makes the type syntax for arrays contiguous, rather than split as it is in our current array declarations. The verbose form `array[2] real` was chosen over the more compact `real[2]` in a community vote[2] because of the possible confusion with using `vector[3][2]` for a 2-element array of 3-vectors; the notation `array[2] vector[3]` is much less ambiguous to the casual reader.

Like all Stan type declarations other than function arguments, sizes are required. Sizes are omitted in function type declarations, where a 1-dimensional array of real numbers will be declared as `array[] real`.

Following C++ syntax, we used fixed numerical accessors for tuple elements because we do not want to have to perform run-time bounds checking. After the example statement above is run, we can access elements of a tuple, or even assign them.

```
int a = y.1;  array[2] real b = y.2;  y.1 = 42;  b.2[1] = 0.3;
```

Unlike C++, Stan indexes starting from 1, not 0. Assigning to a tuple type is elementwise.

It is not part of the required specification, but we would like to be able to support Python-style tuple assignment [VanRossum and Drake 2010], such as

```
int a;   array[2] real b;   (a, b) = foo(...);
```

where `foo` is a function returning a tuple of the same type as `y`.

For R dump I/O, we can use the built-in R dump format for lists to store tuples [?]. For JSON I/O, we can code a tuple as a simple array because the reading routines all know the size and type of elements they are reading in Stan. For instance, the value of `y` above would be encoded as `[3, [1.2, -5.3]]`.

Structs of the form used in the C language are nothing more than tuples with named elements [Ritchie et al. 1988]. Although we will not be implementing structs in the first version of tuples,

[1] Stan also supports gradient-vector products, Hessians, Hessian-vector products, and arbitrary higher-order derivatives in the usual way [Griewank 1989].

[2] The discussion and vote took place on the Stan Forums, in the post https://discourse.mc-stan.org/t/new-array-declaration-syntax/16011

they involve nothing more than adding named keys to tuples in both the type declaration and in the constructors and accessors.

```
(a = int, b = array[2] real) y = (a = 3, b = {1.2, -5.3});
int u = y.a;   array[2] real v = y.b;
```

I/O could be used as-is for tuples if the structs maintain their declared order, or they could be handled with dictionaries in I/O.

3 COMPLEX NUMBERS

A complex number $x + yi$ consists of a real component $x \in \mathbb{R}$ and imaginary component $y \in \mathbb{R}$. So we adopt a tuple-like representation that follows the C++ standard library [Josuttis 2012]. Type notation, construction, and access can be understood with a simple example.

```
real x;  real y;  complex z = complex(x, y);  z.real = 2.9;  y.imag = -1.3;
```

The accessors `z.real` and `z.imag` produce lvalues as indicated by the assignment.

Currently, Stan handles assignment and function arguments in the same way—if an expression can be assigned to to a variable of a given type, then it can be passed to a function argument of the that type. The only promotion that is globally supported is of integer types (`int`) to real types (`real`). These can both now be promoted to `complex`, as well. Stan does not fully support covariance of container types, so that `int[]` is not yet assignable to `real[]`, but that feature is also in the works.

In addition to scalars, we introduce types for matrices, `complex_vector`, `complex_row_vector`, and `complex_matrix`. Scalar and matrix arithmetic extends fully to complex numbers, including mixed operations involving complex and real matrices. This is supported through the Eigen C++ template library scalar traits [Guennebaud et al. 2021]. Stan fully supports all functions in the C++ header `<complex>`, which we extend to mixed real/complex types without promotion for efficiency. The C++ complex library includes standard mathematic functions such as power, exponent, log, trigonometric, absolute value (modulus), etc. In addition, Stan supports matrix functions like discrete Fourier transforms, asymmetric eigendecomposition, and Schur decomposition, also based on the underlying Eigen implementations.

The remaining design issue is around I/O, where we can code random numbers using `x + yi` notation in our R dump format (following R) and as an array `[x, y]` in JSON. We can get away without decorating the fact that the pair of values denotes a complex number because Stan pulls data of known type and size based on its declarations. So if the variable `a` is declared complex, then the I/O routines will attempt to read a two-valued array for `a` as data or initialization.

4 RAGGED ARRAYS

Stan currently supports dense arrays of arbitrary dimension. Other than in function arguments, all arrays are declared with their size and that size may not change after declaration. For example, `real a[2, 3]` declares a 2×3 array to which a differently sized array may not be assigned. In many problems, data is ragged. For example, we may have records of student testing, where each student takes a different number of tests, a corpus of documents where each document is of a different length in tokens, or a population ecology study of animals first tagged at different times. The current approach we use is to encode this data in long form, database style. Ragged arrays make more efficient wide data structures straightforward. To declare a variable `a` as a ragged array of size `M`, we use an array `N` of sizes.

```
int<lower = 0> M;  int<lower = 0> N[M];  int a[N];
```

Here, `a[m, n]` is valid if `m` is in `1:M` and `n` is in `1:N[m]`. Similarly, `size(a) = M` and `size(a[m]) = N[m]`.

While this design is relatively simple for two-dimensional ragged arrays, things get hairy quickly in higher dimensions. The obvious reflexive solution is to use ragged arrays to declare the sizes of

ragged arrays. Thus, when declaring a ragged array, the indexes may themselves be a ragged array of integers. Thus we allow ragged arrays of integers to specify the valid indices of a ragged array. For example, a ragged three-dimensional array might be declared as `real[{{2, 3}, {4, 5, 6}}] a;` with type of `a[1]` being `int[{2, 3}]`, and the type of `a[1, 2]` being `int[3]`. This also highlights brace-constructors for standard and ragged arrays.

The specification also supports raggedness in containers that aren't exclusively arrays. For example, we will support a ragged array of vectors, say `vector[{2, 3}]` for the array containing a 2-vector and 3-vector. Ragged arrays may have constraints (implemented as bijections with Jacobian adjustments) in the same way as dense arrays or scalars.

At the level of implementation, Stan arrays are coded as C++ `vector<T>` types, where `T` is the type of the elements of the array (see [Josuttis 2012] for an overview of C++ templating and the `vector` type). Ragged arrays are implemented the same way, which is what allows the same constructors, etc. to be used. Existing functions that assume dense arrays must all be updated to check their arguments if density is required. Assignment is legal when the sizes conform.

For R dump I/O, ragged arrays will be stored as lists. For JSON, they can simply be stored as nested arrays of values [ECMA International 2017].

5 LAMBDAS AND SIMPLE FUNCTIONAL TYPES

Stan's language of types will be closed under functional type construction, allowing a functional type $T_1 \rightarrow T_2$ for any existing types T_1, T_2. Multivariate arguments are coded with tuples. Subtyping of functions respects the integer to real to complex type promotion hierarchy of basic types; if elements of functional type $T \rightarrow U$ can be assigned to variables or function arguments of type $T' \rightarrow U'$ if elements of T' can be assigned to T and elements of U can be assigned to U' (argument is positive polarity, result negative).

We follow the syntax of C++ for functions [Järvi and Freeman 2008, 2010], using, for example, `real(real)` for a univariate function over reals and `real(real, real)` for function from pairs of reals (a tuple type of two real elements) to reals. This notation extends to higher order functions. For example, univariate function composition would have type `real(real)(real(real), real(real))`, which takes two real-valued functions and returns a real-valued function. The type to curry real-valued functions is `real(real)(real)(real(real, real))`, which takes a bivariate real function and returns a function from reals into a univariate function.

Lambda notation also follows C++ for its basic design [Järvi and Freeman 2010]. For example, a simple function to raise a value to the third power might look like

```
real(real) f = (real x) { return x^3; }
```

The notation `(real x)` is the declaration of the variable being abstracted and `x^3` is the body of the function. The idea is that this defines the same function `f` as if we had written

```
real f(real x) { return x^3; }
```

Because function arguments do not require sizes, there is no complication from size declarations for lambdas.

Stan lambdas may capture variables that are in static lexical scope using closures. Closures can capture local variables, block variables in the current block, or block variables in former blocks. For example, we could have written the above function as

```
real a = 3;  real(real) f = (real x) { return x^a; }
```

if `a` is a transformed data or transformed parameter variable. C++ allows variables to be captured by reference or by value, but in Stan, all variables are captured implicitly by value. This decision was made to prevent users from capturing variables that later go out of scope or are modified

after the function definition. We felt both would be confusing to applied statisticians using these techniques, despite being the default behavior in R. With automatic differentiation, copies still propgate derivative information through the chain rule back to their source variables. They just don't track any changes to their source variables. So if we add a statement `a = 2.5` after the definition of `f`, it will not change `f`'s behavior. The upside of pass-by-value is that variables may be captured at the C++ level using constant references if program analysis shows they are not modified later.

Stan goes beyond C++'s syntax and allows R-like expression lambdas as well as program lambdas. This will allow us to write the inverse logit function as

```
real(real) ilogit = (real u) 1 / (1 + exp(-u));
```

instead of requiring the more verbose form `(real u) { return 1 / (1 + exp(-u)); }`. Unlike R, but like Python, Stan uses static lexical scoping for determining which variables can be captured and has standard scoping for conditionals, etc..

With closures, we are also dropping the restriction that all functions are defined before any other code in a special functions block. Functions can now be defined via lambdas anywhere assignment statements may be used.

The main use of lambdas will be in things like differential and algebraic equation solvers or in parallel map functions. Anonymous lambdas that capture data and parameter values eliminate the need for the packing and unpacking of array arguments.

6 CONCLUSION

All four of these features will extend the ease with which users can encode statistical models using Stan. It's well known that automatic differentiation has no problem with complex numbers or closures—it's just a matter of sitting down and doing the engineering to make it work. Ryan Bernstein has coded prototype implementation of tuples, Steve Bronder has a complete language-level (not I/O level) implementation of complex numbers, and Niko Huurre has a complete implementation of closures. Of all of these proposals, ragged arrays are actually the simplest to add. Hopefully, all of these features will see releases in 2021. The full design specifications are available in our design documents repository [Stan Development Team 2021a].

ACKNOWLEDGMENTS

I'd like to thank the Stan development team for extensive feedback on the design documents and for building prototype implementations, specifically Ben Bales, Ryan Bernstein, Steve Bronder, Rok Češnovar, Jonah Gabry, Niko Huurre, Daniel Lee, Mitzi Morris, Sean Talts, Matthijs Vákár, and Sebastian Weber.

REFERENCES

Bob Carpenter, Andrew Gelman, Matthew D Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus A Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2017. Stan: a probabilistic programming language. *Journal of Statistical Software* 76, 1 (2017), 1–32.

Bob Carpenter, Matthew D Hoffman, Marcus Brubaker, Daniel Lee, Peter Li, and Michael Betancourt. 2015. The Stan math library: Reverse-mode automatic differentiation in C++. *arXiv* 1509.07164 (2015).

ECMA International. 2017. The JSON data interchange syntax, 2nd Edition. Standard ECMA-404 (2017).

David A Fournier, Hans J Skaug, Johnoel Ancheta, James Ianelli, Arni Magnusson, Mark N Maunder, Anders Nielsen, and John Sibert. 2012. AD Model Builder: using automatic differentiation for statistical inference of highly parameterized complex nonlinear models. *Optimization Methods and Software* 27, 2 (2012), 233–249.

Andrew Gelman, John B Carlin, Hal S Stern, David B Dunson, Aki Vehtari, and Donald B Rubin. 2013. *Bayesian Data Analysis* (third edition ed.). Chapman & Hall/CRC press.

Wally R Gilks, Andrew Thomas, and David J Spiegelhalter. 1994. A language and program for complex Bayesian modelling. *Journal of the Royal Statistical Society: Series D (The Statistician)* 43, 1 (1994), 169–177.

Andreas Griewank. 1989. *On automatic differentiation.* Technical Report ANL/MCS-P10-1088. Mathematics and Computer Science Division, Argonne National Laboratory.

Gaël Guennebaud, Benoît Jacob, et al. 2021. Eigen. http://eigen.tuxfamily.org.

Desmond J Higham and Nicholas J Higham. 2016. *MATLAB Guide.* SIAM.

Matthew D Hoffman and Andrew Gelman. 2014. The No-U-Turn sampler: adaptively setting path lengths in Hamiltonian Monte Carlo. *J. Mach. Learn. Res.* 15, 1 (2014), 1593–1623.

Jaakko Järvi and John Freeman. 2008. Lambda functions for C++ 0x. In *Proceedings of the 2008 ACM symposium on Applied computing*. 178–183.

Jaakko Järvi and John Freeman. 2010. C++ lambda expressions and closures. *Science of Computer Programming* 75, 9 (2010), 762–772.

Nicolai M Josuttis. 2012. The C++ standard library: a tutorial and reference. (2012).

Alp Kucukelbir, Dustin Tran, Rajesh Ranganath, Andrew Gelman, and David M Blei. 2017. Automatic differentiation variational inference. *The Journal of Machine Learning Research* 18, 1 (2017), 430–474.

Dong C Liu and Jorge Nocedal. 1989. On the limited memory BFGS method for large scale optimization. *Mathematical programming* 45, 1 (1989), 503–528.

David Lunn, Chris Jackson, Nicky Best, Andrew Thomas, and David Spiegelhalter. 2012. *The BUGS book: A practical introduction to Bayesian analysis.* CRC press.

Benjamin C Pierce. 2002. *Types and programming languages.* MIT Press.

Dennis M Ritchie, Brian W Kernighan, and Michael E Lesk. 1988. *The C programming language.* Prentice Hall Englewood Cliffs.

Gareth O Roberts and Jeffrey S Rosenthal. 1998. Markov-chain Monte Carlo: some practical implications of theoretical results. *The Canadian Journal of Statistics/La Revue Canadienne de Statistique* (1998), 5–20.

Stan Development Team. 2021a. Stan Design Documents Repository. https://github.com/stan-dev/design-docs

Stan Development Team. 2021b. Stan Reference Manual. https://mc-stan.org/documentation

Stan Development Team. 2021c. Stan User's Guide. https://mc-stan.org/documentation

Guido VanRossum and Fred L Drake. 2010. *The python language reference.* Python Software Foundation Amsterdam, Netherlands.