




# BridgeStan: Efficient in-memory access to the methods of a Stan model

Edward A. Roualdes<sup>1\*</sup>, Brian Ward<sup>2\*</sup>, Bob Carpenter<sup>2\*</sup>, Adrian Seyboldt<sup>3</sup>, and Seth D. Axen<sup>4</sup>

<sup>1</sup> California State University, Chico <sup>2</sup> Center for Computational Mathematics, Flatiron Institute <sup>3</sup> PyMC Labs <sup>4</sup> Cluster of Excellence Machine Learning: New Perspectives for Science, University of Tübingen ¶ Corresponding author \* These authors contributed equally.

DOI: [10.21105/joss.05236](https://doi.org/10.21105/joss.05236)

## Software

- [Review](#) 
- [Repository](#) 
- [Archive](#) 

Editor: [Nikoleta Glynnatsi](#) 

## Reviewers:

- [@salleuska](#)
- [@saumil-sh](#)

Submitted: 10 January 2023

Published: 22 July 2023

## License

Authors of papers retain copyright and release the work under a Creative Commons Attribution 4.0 International License ([CC BY 4.0](#)).

## Summary

Stan provides a probabilistic programming language in which users can code Bayesian models (Carpenter et al., 2017; Stan Development Team, 2022). A Stan program is transpiled to a C++ class that links to the Stan math library to implement smooth, unconstrained posterior log densities, gradients, and Hessians as well as constraining/unconstraining transforms. Implementation is provided through automatic differentiation in the Stan math library (Carpenter et al., 2015). BridgeStan provides in-memory access to the methods of Stan models through Python, Julia, R, and Rust. This allows algorithm development in these languages with the numerical efficiency and expressiveness of Stan models. Furthermore, these features are exposed through a language-agnostic C API, allowing foreign function interfaces in other languages to utilize BridgeStan with minimal additional development.

## Statement of need

Stan was developed for applied statisticians working on real-world problems and has been used by hundreds of thousands of researchers and practitioners across the social, biological, and physical sciences, as well as engineering, education, sports, and finance. Stan provides several state-of-the-art, gradient-based algorithms: full Bayesian inference with Hamiltonian Monte Carlo, Laplace approximation based on L-BFGS optimization, and automatic differentiation variational inference (ADVI).

In the statistical software environment R, Stan is heavily relied upon for development of applied statistics packages. Using Google's PageRank algorithm on the dependency graph (de Vries, 2014) amongst the 19,159 R packages listed on the Comprehensive R Archive Network (CRAN) as of 2022-12-31, three R packages that exist solely to provide access to Stan rank quite well: rstan ranks at number 70, rstantools 179, and rstanarm 502. Further, two Python interfaces to Stan, pystan and cmdstanpy, both rank in the top 600 packages by downloads on the Python Package Index (PyPI).

C++ can be cumbersome for algorithm prototyping. As such, developers have been requesting ways to access Stan models for algorithm development in Python, R, and Julia. BridgeStan answers this call, making it easy for algorithm developers to incorporate existing Stan models in their evaluation, e.g., the dozens of diverse models with reference posteriors in posteriorsdb (Magnusson et al., 2022). BridgeStan further aids algorithm development by leveraging the functions in the Stan Math library, which are written to appropriately handle numerical issues such as under/overflow and sum-to-one constraints. Algorithm developers using BridgeStan can thus focus more on their algorithms and less on their implementations. These benefits also

ensure more precise and stable implementations of statistical algorithms, allowing for more fair comparisons.

BridgeStan is an interface, written in C-compatible C++, between a Stan program and any higher level language that exposes a C foreign function interface. Julia, Python, R, and Rust each have C foreign function interfaces. Using memory allocated within such higher level languages, BridgeStan provides computations of the log joint density function and corresponding gradient of a Stan model, which is itself implemented using highly templated C++ from the Stan math library. Using a memory-compatible C interface makes this possible even if the host language (e.g., R) was compiled with a different compiler, something no prior interface that exposed Stan's log density calculations could allow.

Other software in the Stan ecosystem provides some overlapping features with BridgeStan. For instance, `rstan` ([Stan Development Team, 2023](#)) provides functions `log_prob` and `grad_log_prob`, which provide access to the log joint density and its gradient. Similarly, `httpstan` ([Riddell et al., 2021](#)) offers `log_prob` and `log_prob_grad`. Such cases of similar functionality are unfortunately limited. As of 2023-05-19, `rstan` via CRAN is still on Stan version 2.21.0 (released 2019-10-18), and the development version of `rstan`, which is not hosted on CRAN, is on Stan version 2.26.1 (released 2021-02-17), while the latest version of Stan is on 2.32.2 (released 2023-05-15). Further, `rstan` is limited to the host language R. On the other hand, `httpstan` is a Python package that offers a REST API, primarily targeting the Stan algorithms, which allows some limited access to the methods of a Stan model. The REST API may be used by languages other than Python but by design cannot take advantage of direct memory access of the host language. Additionally, `httpstan` is not natively supported on Windows operating systems. BridgeStan addresses these issues by providing a portable and easy to maintain shim between any host language with a foreign function interface to C and the core C++ of Stan.

Existing tools with similar automatic differentiation functionality include JAX ([Bradbury et al., 2018](#)) and `Turing.jl` via the JuliaAD ecosystem ([Ge et al., 2018](#)). BridgeStan differs from these tools by providing access to the existing, well-known DSL for modeling and highly efficient CPU computation of the Stan ecosystem. The Stan community predominantly uses CPU hardware, and since Stan has been tuned for CPU performance, BridgeStan is more efficient than its competitors in implementing differentiable log densities on CPUs ([Carpenter et al., 2015](#); [Radul et al., 2020](#); [Tarek et al., 2020](#)). Like the immutable Stan models they interface, BridgeStan functions are thread-safe for parallel applications. They also support all of the internal parallelization of Stan models, such as internal parallel map functions and GPU-enabled matrix operations.

BridgeStan enables memory allocated in the host language (Julia, Python, R, or Rust), to be reused within Stan, though any language with a C foreign function interface could be similarly interfaced to access Stan methods. For instance, the BridgeStan function `log_density_gradient` has as an optional output argument the array into which the gradient will be stored. By avoiding unnecessary copies, BridgeStan is a zero-cost abstraction built upon Stan's math library. If no output argument is passed to `log_density_gradient`, then at most one memory allocation, occurring in the host language, takes place.

## Example

The probabilistic programming language Stan, together with its automatic differentiation tools, enables parameterizations of otherwise numerically challenging distributions. Consider the following Stan program, which encodes an isotropic multivariate Student-t distribution of dimension  $D$  and degrees of freedom  $df$ .

This parameterization<sup>1</sup> of the Student-t distribution enables gradient-based Markov chain Monte

<sup>1</sup>See Wikipedia's page on the [Student's t-distribution](#) for a brief introduction to this parameterization.

Carlo algorithms to capture the heaviness of the tails when  $df$  is less than  $\sim 30$ . Calculating the gradient of the joint log density of this parameterization of the Student-t distribution is not difficult, but it is cumbersome and time-consuming to encode in software. Since BridgeStan uses Stan, users of BridgeStan can trust that their bespoke parameterizations of numerically challenging distributions will be differentiated with thoroughly tested tools from Stan.

```
data {
  int D;
  real df;
}
transformed data {
  vector[D] mu = rep_vector(0.0, D);
  matrix[D, D] Sigma = identity_matrix(D);
  real<lower=0.0> nu = 0.5 * df;
}
parameters {
  vector[D] z;
  vector<lower=0>[D] ig; // ig constrained so ig > 0
}
transformed parameters {
  vector[D] x = z .* sqrt(ig);
}
model {
  z ~ multi_normal(mu, Sigma);
  ig ~ inv_gamma(nu, nu);
}
```

BridgeStan users can access the gradient of this model easily, allowing for simple implementations of sampling algorithms. In the below example, we show an implementation of the Metropolis-adjusted Langevin algorithm (MALA) (Besag, 1994) built on BridgeStan.

```
import bridgestan as bs
import numpy as np

stan_model = "path/to/student-t.stan"
stan_data = "path/to/student-t.json"
model = bs.StanModel.from_stan_file(stan_model, stan_data)
D = model.param_unc_num()
M = 10000

def MALA(model, theta, epsilon=0.45):
    def correction(theta_prime, theta, grad_theta):
        x = theta_prime - theta - epsilon * grad_theta
        return (-0.25 / epsilon) * x.dot(x)

    lp, grad = model.log_density_gradient(theta)
    theta_prop = (
        theta
        + epsilon * grad
        + np.sqrt(2 * epsilon) * np.random.normal(size=model.param_unc_num())
    )

    lp_prop, grad_prop = model.log_density_gradient(theta_prop)
    if np.log(np.random.random()) < lp_prop + correction(
        theta, theta_prop, grad_prop
    ) - lp - correction(theta_prop, theta, grad):
        return theta_prop
    return theta

unc_draws = np.empty(shape=(M, D))
unc_draws[0] = MALA(model, np.random.normal(size=D))
for m in range(1, M):
    unc_draws[m] = MALA(model, unc_draws[m - 1])

# post processing: recover constrained/transformed parameters
draws = np.empty(shape=(M, model.param_num(include_tp=True, include_gq=True)))
for (i, draw) in enumerate(unc_draws):
    draws[i] = model.param_constrain(draw, include_tp=True, include_gq=True)
```

## Conclusion

On the [Stan Discourse forums](#), statistical algorithm developers have long asked for access to the gradients and Hessians that underlie the statistical model of a Stan program. Examples include requests on the Stan Discourse forums related to the phrase [extract gradient](#) or the software from which BridgeStan is derived: [Stan Model Server](#) and [ReddingStan](#). BridgeStan enables access to these methods with a portable and in-memory solution. Further, because statistical models are so easy to write in Stan, algorithm developers can write their model in common statistical notation using the Stan programming language and then rely on the Stan math library and its automatic differentiation toolset to more easily build advanced gradient-based statistical inference algorithms. BridgeStan documentation and example programs are found at <https://roualdes.github.io/bridgestan/index.html>.

## Acknowledgements

Edward A. Roualdes received support from Flatiron Institute during the beginning of this project. Seth D. Axen is funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC number 2064/1 – Project number 390727645.

## References

- Besag, J. (1994). Comments on "Representations of knowledge in complex systems" by U. Grenander and MI Miller. *Journal of the Royal Statistical Society, Series B*, 56(591-592), 4.
- Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., & Zhang, Q. (2018). *JAX: Composable transformations of Python+NumPy programs* (Version 0.3.13). <http://github.com/google/jax>
- Carpenter, B., Gelman, A., Hoffman, M. D., Lee, D., Goodrich, B., Betancourt, M., Brubaker, M., Guo, J., Li, P., & Riddell, A. (2017). Stan: A probabilistic programming language. *Journal of Statistical Software*, 76(1).
- Carpenter, B., Hoffman, M. D., Brubaker, M., Lee, D., Li, P., & Betancourt, M. (2015). The stan math library: Reverse-mode automatic differentiation in c++. *arXiv Preprint arXiv:1509.07164*.
- de Vries, A. (2014). *Finding the essential R packages using the pagerank algorithm*. <https://blog.revolutionanalytics.com/2014/12/a-reproducible-r-example-finding-the-most-popular-packages-1.html>
- Ge, H., Xu, K., & Ghahramani, Z. (2018). Turing: A language for flexible probabilistic inference. *International Conference on Artificial Intelligence and Statistics, AISTATS 2018, 9-11 April 2018, Playa Blanca, Lanzarote, Canary Islands, Spain*, 1682–1690. <http://proceedings.mlr.press/v84/ge18b.html>
- Magnusson, M., Bürkner, P., & Vehtari, A. (2022). *posteriordb: a set of posteriors for Bayesian inference and probabilistic programming* (Version 0.4). <https://github.com/stan-dev/posteriordb>
- Radul, A., Patton, B., Maclaurin, D., Hoffman, M., & A Saurous, R. (2020). Automatically batching control-intensive programs for modern accelerators. *Proceedings of Machine Learning and Systems*, 2, 390–399.
- Riddell, A., Hartikainen, A., & Carter, M. (2021). *Httpstan (4.4.0)*. PyPI.

Stan Development Team. (2022). *About Stan*. <https://mc-stan.org/>

Stan Development Team. (2023). *RStan: The R interface to Stan*. <https://mc-stan.org/>

Tarek, M., Xu, K., Trapp, M., Ge, H., & Ghahramani, Z. (2020). DynamicPPL: Stan-like speed for dynamic probabilistic models. *arXiv Preprint arXiv:2002.02702*.