The Journal of Open Source Software

caskade: building Pythonic scientific simulators

Connor Stone [1,2,3¶], **Alexandre Adam** [1,2,3], **Adam Coogan** [1,2,3,a], **Laurence Perreault-Levasseur** [1,2,3,4,5,6], **and Yashar Hezaveh** [1,2,3,4,5,6]

1 Ciela Institute - Montréal Institute for Astrophysical Data Analysis and Machine Learning, Montréal, Québec, Canada **2** Department of Physics, Université de Montréal, Montréal, Québec, Canada **3** Mila - Québec Artificial Intelligence Institute, Montréal, Québec, Canada **4** Center for Computational Astrophysics, Flatiron Institute, 162 5th Avenue, 10010, New York, NY, USA **5** Perimeter Institute for Theoretical Physics, Waterloo, Canada **6** Trottier Space Institute, McGill University, Montréal, Canada **a** Work done while at UdeM, Ciela, and Mila ¶ Corresponding author

DOI: 10.21105/joss.08786

Software

- Review
- Repository
- Archive

Editor: Vincent Knight

Reviewers:

- @avapolzin
- @aslan-ng

Submitted: 13 July 2025
Published: 15 September 2025

Summary

Scientific simulators and pipelines form the core of many research projects. Writing high quality, modular code allows for efficiently scaling a project, but this can be challenging in a research context. Research project goals and solutions to those goals are constantly in flux, requiring many refactoring rounds to meet these changes. The result can be a progressively more unwieldy interconnected code. Here we present a system, caskade, which allows users to focus on modular components of a simulator, these are small and testable to ensure robustness. With caskade one can turn these modular components into abstracted blocks that connect to form a powerful simulator. caskade manages the flow of parameter values through such a simulator.

Statement of Need

Science is an intrinsically iterative process, and so is the development of scientific code. Well written code is flexible and scalable while being performant, this is difficult to achieve in a scientific context where goals often evolve rapidly, requiring code refactoring. A major aspect of this is the parameters of a scientific model, the values that will ultimately be sampled and/or optimized to represent some data. A value may need to alternately be fixed, then allowed to vary (e.g. in Gibbs sampling). Some parameters that were initially separate may need to share a value or some functional relationship. In the extreme, a whole simulator may become a function of a single variable, such as time. Meta-data such as the uncertainty or valid range of a parameter may need to be stored. One may need to represent all parameters as a single 1D vector to interface with external tools, such as emcee (Foreman-Mackey et al., 2013), scipy.optimize (Virtanen et al., 2020), Pyro (Bingham et al., 2019), dynesty (Speagle, 2020), and torch.Optim (Paszke et al., 2019). Large projects and correspondingly large teams require the ability to break projects into manageable subtasks which can later be naturally combined into a complete analysis suite. Most importantly, as all of the above needs change, it is critical to meaningfully re-use older code without "code debt" or "software entropy" growing unsustainably.

Features

The core features of caskade are the `Module` base class, `Param` object, and `forward` decorator. To construct a caskade simulator, one subclasses `Module` then adds some number of `Param` objects as attributes of the class. Any number of class methods may be decorated with `@forward`, meaning caskade will manage the Param arguments of that function. As modules are combined into a larger simulator, caskade builds a directed acyclic graph (DAG) representation. This allows it to automatically manage the flow (cascade) of parameters through the simulator and encode arbitrary relationships between them. This is inspired by the PyTorch framework nn.Module which allows for near-effortless construction of machine learning models. We generalize the object oriented framework to apply to almost any scientific forward model, simulator, analysis pipeline, and so on; caskade manages the flow of parameters through these models.

Thus the primary capability of caskade is the management of Param values as they enter @forward methods of Modules. Any parameter may be transformed between "static" and "dynamic" where static has a fixed value and dynamic is provided at call time. Parameters may be synced with arbitrary functional relationships between them. New parameters may be added dynamically to allow for sophisticated transformations. For example, an entire simulator may be turned into a function of time without modifying the underlying simulator by adding a time parameter and linking appropriately. It is possible to use caskade with NumPy (Harris et al., 2020), JAX (Bradbury et al., 2018), or PyTorch (Paszke et al., 2019) numerical backends.

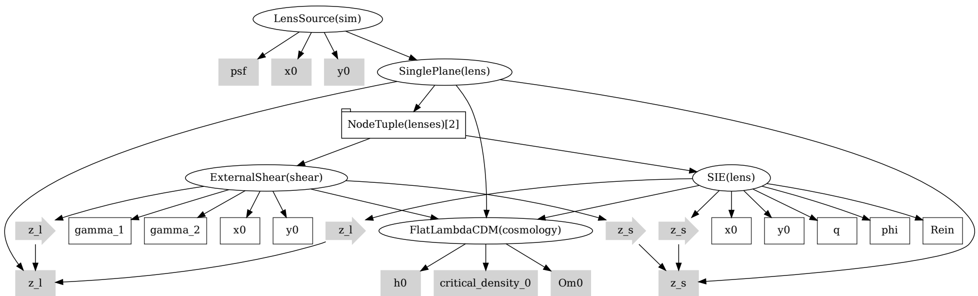

Figure 1: Example caskade DAG representation of a gravitational lensing simulator. Ovals represent Modules, boxes represent dynamic parameters, shaded boxes represent fixed parameters, arrow boxes represent parameters which are functionally dependent on another parameter, and thin arrows show the direction of the graph flow for parameters passed at the top level.

Our suggested design flow is to build out a functional programming base for the package, then use Modules as wrappers for the functional base to design a convenient user interface. This design encourages modular development and is supportive of users who wish to expand functionality at different levels. The caustics package (Stone et al., 2024) implements this code design to great effect. Figure 1 shows an example caskade graph[1] from caustics. In this graph the redshift parameters (z_l and z_s) of each lens are linked to ensure consistent evaluation despite the functional backed having no explicit enforcement of this. See also that all of the lens objects (ExternalShear, SIE, and SinglePlane) point to a single cosmology Module and so share the same cosmological parameters automatically.

State of the Field

In some ways caskade is reminiscent of Hydra (Yadan, 2019), however caskade focuses on numerical parameters and scientific inference, while Hydra focuses on configuration management and large scale process organization. The two may even be used in tandem. Another package, tesseract-core (Häfner & Lavin, 2025) focuses more on containerization and distribution of simulations to interface different ecosystems (PyTorch and JAX as well as Python and C++) and on different compute engines (HPC clusters or in cloud). The SimFrame (Stammler

[1]visual generated by graphviz (Ellson et al., 2004)

& Birnstiel, 2022) package shares caskade's modular and extensible core design, though is focused exclusively on solving differential equations. Encoding the Functional Mockup Interface standard (Blochwitz, 2012) is the Ecos package (Hatledal, 2025) which is also designed for building modular simulators though in the more strict FMI standard which requires auxiliary .xml specification files, caskade focuses on lean and active research development which thrives on minimal overhead. Finally, PathSim also shares the caskade modular simulator building framework, though it focuses exclusively on time-domain dynamical systems. Clearly, many fields of research and development desire such modular simulation-building frameworks; caskade fulfills the role very generally, though not so abstractly as to require overhead schema or meta-data files.

Acknowledgements

This research was enabled by a generous donation by Eric and Wendy Schmidt with the recommendation of the Schmidt Futures Foundation. CS acknowledges the support of a NSERC Postdoctoral Fellowship and a CITA National Fellowship. This research was enabled in part by support provided by Calcul Québec and the Digital Research Alliance of Canada. The work of A.A. was partially funded by NSERC CGS D scholarships. Y.H. and L.P. acknowledge support from the National Sciences and Engineering Council of Canada grants RGPIN-2020-05073 and 05102, the Fonds de recherche du Québec grants 2022-NC-301305 and 300397, and the Canada Research Chairs Program.

Bingham, E., Chen, J. P., Jankowiak, M., Obermeyer, F., Pradhan, N., Karaletsos, T., Singh, R., Szerlip, P., Horsfall, P., & Goodman, N. D. (2019). Pyro: Deep universal probabilistic programming. *The Journal of Machine Learning Research*, *20*(1), 973–978. https://doi.org/10.48550/arXiv.1810.09538

Blochwitz, O., T. (2012). *Functional mockup interface 2.0: The standard for tool independent exchange of simulation models*. 173–184. https://doi.org/10.3384/ecp12076173

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leery, 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

Ellson, J., Gansner, E. R., Koutsofios, E., North, S. C., & Woodhull, G. (2004). Graphviz and dynagraph—static and dynamic graph drawing tools. *Graph Drawing Software*, 127–148. https://doi.org/10.1007/978-3-642-18638-7_6

Foreman-Mackey, D., Hogg, D. W., Lang, D., & Goodman, J. (2013). emcee: The MCMC Hammer. *Publications of the Astronomical Society of the Pacific*, *125*(925), 306. https://doi.org/10.1086/670067

Häfner, D., & Lavin, A. (2025). Tesseract core: Universal, autodiff-native software components for simulation intelligence. *Journal of Open Source Software*, *10*(111), 8385. https://doi.org/10.21105/joss.08385

Harris, C. R., Millman, K. J., Walt, S. J. van der, Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., Kerkwijk, M. H. van, Brett, M., Haldane, A., Río, J. F. del, Wiebe, M., Peterson, P., … Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, *585*(7825), 357–362. https://doi.org/10.1038/s41586-020-2649-2

Hatledal, L. I. (2025). Ecos: An accessible and intuitive co-simulation framework. *Journal of Open Source Software*, *10*(110), 8182. https://doi.org/10.21105/joss.08182

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., … Chintala, S. (2019). PyTorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems 32* (pp. 8024–8035). Curran Associates, Inc. https://doi.org/10.48550/arXiv.1912.01703

Speagle, J. S. (2020). DYNESTY: a dynamic nested sampling package for estimating Bayesian posteriors and evidences. *Monthly Notices of the Royal Astronomical Society*, *493*(3), 3132–3158. https://doi.org/10.1093/mnras/staa278

Stammler, S. M., & Birnstiel, T. (2022). Simframe: A python framework for scientific simulations. *Journal of Open Source Software*, *7*(69), 3882. https://doi.org/10.21105/joss.03882

Stone, C., Adam, A., Coogan, A., Yantovski-Barth, M. J., Filipp, A., Setiawan, L., Core, C., Legin, R., Wilson, C., Barco, G. M., Hezaveh, Y., & Perreault-Levasseur, L. (2024). Caustics: A python package for accelerated strong gravitational lensing simulations. *Journal of Open Source Software*, *9*(103), 7081. https://doi.org/10.21105/joss.07081

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., … SciPy 1.0 Contributors. (2020). SciPy 1.0: Fundamental algorithms for scientific computing in Python. *Nature Methods*, *17*, 261–272. https://doi.org/10.1038/s41592-019-0686-2

Yadan, O. (2019). *Hydra - a framework for elegantly configuring complex applications*. Github. https://github.com/facebookresearch/hydra