The State of Julia for Scientific Machine Learning

Edward Berman *
Northeastern University
Boston, MA 02115
berman.ed@northeastern.edu

Jacob Ginesin *
Northeastern University
Boston, MA 02115
ginesin.j@northeastern.edu

Abstract

Julia has been heralded as a potential successor to Python for scientific machine learning and numerical computing, boasting ergonomic and performance improvements. Since Julia's inception in 2012 and declaration of language goals in 2017, its ecosystem and language-level features have grown tremendously. In this paper, we take a modern look at Julia's features and ecosystem, assess the current state of the language, and discuss its viability and pitfalls as a replacement for Python as the de-facto scientific machine learning language. We call for the community to address Julia's language-level issues that are preventing further adoption.

1 Introduction — A Tale of Two Languages

Historically, Python has been the dominant language for machine learning research, including in the sciences. Python is intuitive and has a rich ecosystem across all of the natural sciences. Yet, the rise of Python as the go-to language for machine learning has been in many ways unnatural. Python is a scripting language, extremely slow, and challenging to maintain. The advent of Julia was, in many ways, designed to address the limitations of Python. Julia boasts a suite of language-level features that are designed for intuitive and fast numerical computations.

Since Julia's introduction in 2012, the language has experienced steady growth among practitioners [11, 1, 64]. Given Julia's specific intent to solve problems in scientific machine learning, it's natural to ask why Julia hasn't challenged Python in popularity within the natural sciences. Some of this can be attributed to the *momentum* Python has accumulated over the years within the community [79]. However, with the rapid growth of scientific machine learning in recent years, it is to be seen whether Julia will see broader adoption.

In this work, we explore the readiness of Julia as the de-facto tool for scientific machine learning. We focus not only on the wealth of libraries at Julia's disposal but also its performance, design philosophy, and overall ergonomics. A key theme in this work is that different programming languages provide different abstractions that significantly change the way a user interacts with a scientific machine learning problem (SMLP), and Julia provides a very different set of abstractions than what is seen in other ecosystems. In contrast to previous perspectives [58, 46], we argue that while the Julia ecosystem provides a number of useful abstractions for SMLPs, the limitations are severe enough to prevent it from further adoption. We conclude by calling for the community to discuss and address Julia's language-level limitations.

2 The Scientific Computing Ecosystem

In this Section, we compare Julia's ecosystems with its contemporaries. We compare primarily with Python as it is the go-to choice for most researchers tasked with a SMLP. However, we recognize

---

*Equal Contribution.

that languages such as C, C++, Fortran, and Cuda see similar usage at smaller scales. We point out examples of SMLPs that Julia is especially useful for throughout.

Linear Algebra   Arguably the most important part of the Julia ecosystem is the standard library for linear algebra. Fast numerical computations motivated Julia's inception. To achieve this, Julia employs a Just in Time (JIT) compiler to speed up its computation. The utility of the JIT compiler is seen across the language, but especially within its linear algebra library. For example, Julia is significantly faster than Python and about as fast as C in terms of random matrix multiplication and random matrix statistics [11]. Python matrix operations can be JIT compiled with Jax [14], however, the exact performance gain compared to C or Julia has yet to be properly assessed. Julia also has a plethora of easy to use functions for matrix factorization and libraries for working with sparse graphs that can make matrix operations even faster [106]. This makes Julia the most compelling choice for SMLPs like SLAM [97, 98], where there is natural graph sparsity to exploit. There are also some widely adopted libraries for sparsity in C and C++ [73, 36, 45, 65, 37, 24, 38, 26, 28, 29, 23, 22, 31, 30, 5, 6, 94, 32, 33, 34, 35, 16, 111, 25, 21, 27, 110, 99, 43].

Constrained Optimization   In a similar flavor to numerical linear algebra, constrained optimization algorithms mesh well with Julia's strengths. Julia's constrained optimization algorithms are fast and mathematically sound, and in comparison to Python, much more abundant.

Julia has the most sophisticated support for specifying optimization problems on manifolds. Naive approaches to SMLPs do not account for inductive biases and physical constraints (e.g. [10, 9, 98, 97, 4, 12]). One approach to endowing physical constraints into a problem is by specifying a manifold that the data lives on. This is implemented in Julia with Manopt [7]. While there are Python and Matlab ports of Manopt [13, 107], they offer only a strict subset of what the Julia library offers [7]. The Python port in particular is very limited. There exists similar packages in R and C++ [52, 76] but they are not actively maintained.

Manopt is designed to constrain iterative updates to stay on a specified manifold. However, often times constraints are specified softly via loss functions such as those in Physics Informed Neural Networks (PINNs). PINNs are able to take boundary and initial conditions into account when solving for a physical law specified as a differential equation. Julia handles this wonderfully, with Flux.jl [57, 56] and Lux.jl [88] both providing neural network backends that are compatible with the DiffEqFlux.jl [92] and NeuralPDE.jl [113] packages. Even beyond neural methods, Julia has an extremely rich ecosystem for solving differential equations [93], however this is out of the scope of this paper. Lux.jl in particular supports arbitrary types, making it composable with arbitrary ODE/PDE solvers and even other languages (more on this in § 3). In contrast, there are no de-facto solutions for physics informed machine learning in Python. That is, no existing solution accommodates neural networks specified in either TensorFlow [3], Haiku [51], or PyTorch [89] simultaneously. As a result, we see standard architectures spread across different sub-ecosystems without any libraries available to bridge them together. For example, we see the canonical implementations for lagrangian neural networks [19] and normalizing flows [63, 48] implemented in Jax (possibly with Diffrax [63]), and the canonical implemenations of Kolmogorov Arnold Networks [72] and Deep Operator Networks [70, 74] implemented in PyTorch.

Julia rounds out its suite of constrained optimization libraries with GeometricFlux.jl [108]. Geometric-Flux.jl again builds on the Flux.jl framework with specific layers and functionality for the $5(+1)G$ fields, which includes **G**raphs and sets, **G**rids and Euclidean Spaces, **G**roups and Homogenous Spaces, **G**eodesics and Manifolds, **G**auges and Bundles, and **G**eometric Algebra [15]. As with PINNs, these same functionalities appear scattered throughout different Jax and Torch libraries that are often incompatible with one another. This support makes Julia an attractive choice for working on SMLPs such as excited state molecular dynamics [112, 104]. There is also the GraphNeuralNetworks.jl library [75], which is more actively maintained than GeometricFlux.jl, and has similar features.

Automatic Differentiation   Julia maintains several different packages for both forward mode and reverse mode automatic differentiation (AD) with ForwardDiff.jl [96] and Zygote.jl [56] respectively (see also [82, 39, 100, 42]). In other ecosystems, forward mode and reverse mode AD are typically handled within the same library, as is the case with PyTorch [89] and Jax [14]. In general, Julia's AD libraries are much more ambitious in scope, allowing for more overhead from the user at the potential cost of ease of use. We identify this as one of the few areas where there is a lot of friction

when writing Julia code. One way in which this is realized is with custom gradient interventions and definitions. In Julia, macros for overriding and specifying differentiation rules are housed in many separate libraries, including Zygote.jl itself and the general purpose DiffRules.jl library [95]. Navigating the maze of different tools is often counter intuitive and involves much more domain knowledge than with other tools. In contrast, custom differentiation rules are very easy in Jax with the *@custom_JVP* and *@custom_VJP* macros.

Zygote.jl is most prominently used as the backend of the Flux.jl machine learning library. Additionally, the aforementioned automatic differentiation tools also enable Optim.jl [81] for use with SymbolicRegressions.jl. In the space of symbolic regression, Julia has a clear advantage in terms of support. The most widely adopted library for symbolic regression is SymbolicRegressions.jl [18]. The Python alternative PySR is simply a wrapper for this library, which is in turn a dependency for PyTorch [69].

Probabilistic Programming   Julia boasts many Probabilistic Programming Languages (PPLs) built on top of it. These PPLs make SMLPs involving Bayesian statistics extremely simple to express, and include packages like SOSS.jl [103] and Turing.jl [47]. SOSS.jl uniquely provides a set of tools for working with measure-theoretic objects not seen in other languages [61, 102, 103]. Save for the measure theoretic support, a similar library also exists in Python with PyAutoFit [84]. Additionally, Jax [14] with NumPyro [91, 44] makes it exceptionally easy to specify probabilistic problems.

3   Design Philosophy and Ergonomic Machine Learning

Julia was founded on strong design ideals; in this section we discuss Julia's features and how they enable fluid development and solving of SMLPs.

Multiple Dispatch   As evidenced by [11], Julia makes extensive use of multiple dispatch [83], a method to automatically choose function behavior based on input types. In practice, this can make writing code much more ergonomic for the end user. As an example, the Python/C++ library for computing correlation functions, TreeCorr [59], has 18 different subclasses that the user must remember in order to correlate different kinds of objects. All of these subclasses are prefixed with single letters, such as KVCORRELATION for scalar-vector correlations and VVCORRELATION for vector-vector correlations. Needless to say, it can be difficult to keep track of all 18 combinations, but with Julia's multiple dispatch, we can safely define the same function 18 times for the different types of inputs without having to worry about specifying which one to call in practice. This is the approach also taken by CosmoCorr.jl [8]. Other libraries that make extensive use of multiple dispatch include JuMP.jl [41], ForwardDiff.jl [96], and Julia's standard library.

Composition vs. Inheritance   Julia's design philosophy and style guides revolve around the idea of highly composable interfaces. This naturally plays off of the use of multiple dispatch: functions are not attached to a single struct and instead are declared globally, potentially multiple times for different types. An example of this can be seen with Flux.jl [57, 56]. A neural network in Flux.jl is just a chain of functions that can act on different types. This makes Flux.jl much more ergonomic as it abstracts away the different input types you may provide to a neural network, eliminating the need to extend a module and format the input data in a certain way (i.e. as one would do with PyTorch [89]). It is for this reason that Flux.jl boasts that it is the library that *doesn't make you tensor*. The emphasis on composition is also useful for package management. Since there is very little shared state between different user defined structs due to multiple dispatch, packages are less likely to conflict. Even when there are conflicts, Julia's package manager Pkg.jl [66, 49] is much easier to use. Python's package management on the contrary, which depends on Python virtual environment management with Pip or Conda, is notoriously error prone.

Two Language Problem   The two language problem states that a programming language is either fast or easy to use, but not both. Thus, we often write performance critical code in C or C++ and call it from Python. This makes projects difficult to maintain as there are now multiple environments the user needs to keep track off. This also makes reproducibility and contribution more difficult as it requires everyone in the scientific community to know a second, low level technical language[2].

---

[2]or, at least how to compile it with make or cmake, given the project doesn't provide a dockerfile or similar.

The most glaring example of this is PyTorch [89], which is a Python interface to a C++ backend. However, even libraries that provide much simpler functionality than a neural network have this issue. For example, TreeCorr also has a Python API for functions written in C++ [59]. Julia claims to solve the two language problem, allowing both flexible prototyping and deep performance optimization within the same language [11]. For example, the Flux.jl [57, 56] and CosmoCorr.jl [8] libraries are able to efficiently train neural networks and estimate correlation functions all with high-level Julia code. In general, the interplay between prototyping and optimizing is a core design pattern when implementing solutions for SMLPs, and Julia's design uniquely facilitates this interplay.

4 Limitations of Julia for Scientific Machine Learning

In this section, we outline the limitations we find most severe in Julia for solving SMLPs.

Lack of Software Engineering Features In stark comparison with Python, Julia lacks many software engineering-oriented language and ecosystem features. Primarily, Julia has significantly less mature testing infrastructure than Python; the Unittest and Pytest Python libraries are much more robust than Julia's built in testing library. Additionally, Julia has virtually no support for more advanced testing methods such as property-based testing, symbolic execution, and contract-based testing, of which are universally employed by Python's large-scale numerical methods and machine learning libraries such as Pandas [77], NumPy [50], SciPy [109] , SymPy [78], Scikit-Learn [90], Jax [14], Tinygrad [20], and Cupy [85] through the Hypothesis [55], Crosshair [101], and Deal [71] libraries. Julia also lacks an actively maintained static type checker [80, 105, 40]. More rigorous testing methods are especially important in scientific computation, where precision and correctness of implementations of algorithms is paramount. In being able to deeply test properties of an algorithm, we can better reconcile epistemic (model) uncertainties [87] from aleatoric (data) uncertainties.

Complex Debugging Messages On a language level, Julia's error messages have continually been an unaddressed pain point in the community. The complexity of multiple dispatch, as well as the notoriously long stack traces, make digesting Julia's error messages a tedious process. Additionally, a recent Julia Hub survey showed that Julia programmers cited complex debugging messages as one of the biggest technical hurdles for the language [53, 54].

Limited Industry Adoption While Julia may be gaining popularity in academic circles, machine learning research has a heavy industry presence unseen in many other disciplines. As such, open source contributions from industry is a huge factor in the overall adoption of a language or library. A key limitation of Julia is its lack of support from industry giants. Julia's biggest competitor is arguably Jax [14], which is maintained by Google Deepmind — we see Jax as something that treats the symptoms of Python's problems, the most significant of which is its performance. Moreover, companies like HuggingFace have designed their API's for sharing model weights and datasets exclusively around the Python machine learning libraries, making it much more difficult for Julia users to share their scientific models with one another and necessitating the use of Python wrappers (e.g. [17]). While Julia has a growing number of industry backers [1], there is a clear resource advantage for the Python ecosystems. Supporting this, the 2024 Julia Hub survey [53, 54] found 64% of users thought one of the biggest non-technical problems with the language was "there are not enough Julia users in my field or industry," a jump of $\approx 6\%$ from 2023. The 2024 survey also found that while 71% of respondents use Julia for research, only 16% reported "I use Julia in production for a business critical task" [53, 54]. Industrial applications require robust testing and interoperability with existing tools, supporting the idea that the limitations laid out in this section are not at all unrelated.

Poor Interoperability Despite the annoyances of the two language problem laid out in § 3, users will inevitably need to use multiple languages until scientists start writing their domain specific applications in Julia. While calling Python, R, C, and C++ functions in Julia is relatively easy with the PythonCall.jl [62], PyCall.jl [60], RCall.jl [67], and the Julia standard library function *ccall* [68], the converse is often extremely difficult and unintuitive as the user will need to explicitly manage Julia's context and the passing of complex datatypes between languages. While [62] handles the Julia to Python conversion most graciously, there is still much to be desired in calling Julia from other languages.

5 Conclusion and Call to Action

Not only does Julia match and in many ways exceed Python with its suite of general purpose scientific computing tools, but it also provides many more specific tools for solving SMLPs unseen in any other language. This is particularly true for constrained optimization problems that naturally arise in the natural sciences. Unfortunately, this has proven to be not enough for Julia to gain widespread adoption. While Julia has been slowly gaining traction, in comparison to the growth of scientific machine learning as a whole, Julia's growth has been quite slow.

Julia's lack of software engineering-centric features, lackluster debugging infrastructure, subpar industry adoption, its "rivalry" with Jax, and insufficient interop all bottleneck further adoption. However, none of these limitations in isolation seem like dealbreakers. On paper, Julia *should* compete with Python. So again we ask, why doesn't it?

In the seminal paper, "Julia: A Fresh Approach to Numerical Computing" [11], the initial vision for Julia is summarized in three points. 1) A programming language can be high-level, dynamic, and fast. 2) A programming language can be used for both scripting and deployment. 3) A programming language should supply a mechanism to easily abstract away unnecessary detail typically left to "the experts." Arguably, Julia today has addressed all three of these points and matured with respect to them. Yet, Julia remains relatively niche despite the growing potential user base.

In present year, Julia users have mostly moved away from improving the Julia language and are now more focused on developing libraries for specific projects. With love, we argue the Julia community has moved on too quickly. The state of machine learning is rapidly changing, and Julia has the potential to address many of the pain points in the community. However, if we do not address problems at the language level, we are in danger of repeating many of the mistakes of Python.

We believe the current state of Julia lacks vision. Julia needs a new constitution: a set of concrete goals for improvement, adoption, and outreach. While Python has a clear list of future goals [2], and individual ecosystems *within* Julia such as SciML have roadmaps [86], the Julia language itself has nothing but surface-level GitHub issues. It is only once we map out and solve these language-level issues that we can then ask if Julia is capable of succeeding Python as the de-facto language for scientific machine learning.

We ask the Julia and scientific machine learning communities: what is the future for Julia?

Acknowledgments and Disclosure of Funding

References

[1] Juliahub case studies. `https://info.juliahub.com/case-studies`. Accessed: 2024-08-31.

[2] Python enhancement proposals. `https://peps.python.org/`. Accessed: 2024-09-03.

[3] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for Large-Scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, Savannah, GA, November 2016. USENIX Association.

[4] P-A Absil, Robert Mahony, and Rodolphe Sepulchre. *Optimization algorithms on matrix manifolds*. Princeton University Press, 2008.

[5] Patrick R. Amestoy, Timothy A. Davis, and Iain S. Duff. An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications*, 17(4):886–905, 1996.

[6] Patrick R. Amestoy, Timothy A. Davis, and Iain S. Duff. Algorithm 837: Amd, an approximate minimum degree ordering algorithm. *ACM Trans. Math. Softw.*, 30(3):381–388, sep 2004.

[7] Ronny Bergmann. Manopt.jl: Optimization on manifolds in julia. *Journal of Open Source Software*, 7(70):3866, 2022.

[8] Edward Berman. Cosmocorr. `https://github.com/EdwardBerman/CosmoCorr`, 2024. GitHub repository.

[9] Edward Berman and Jacqueline McCleary. Shopt.jl: A julia package for empirical point spread function characterization of jwst nircam data. *Journal of Open Source Software*, 9(100):6144, 2024.

[10] Edward M. Berman, Jacqueline E. McCleary, Anton M. Koekemoer, Maximilien Franco, Nicole E. Drakos, Daizhong Liu, James W. Nightingale, Marko Shuntov, Diana Scognamiglio, Richard Massey, Guillaume Mahler, Henry Joy McCracken, Brant E. Robertson, Andreas L. Faisst, Caitlin M. Casey, Jeyhan S. Kartaltepe, and COSMOS-Web: The JWST Cosmic Origins Survey. Efficient point-spread function modeling with shopt.jl: A point-spread function benchmarking study with jwst nircam imaging. *The Astronomical Journal*, 168(4):174, sep 2024.

[11] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

[12] Nicolas Boumal. *An introduction to optimization on smooth manifolds*. Cambridge University Press, 2023.

[13] Nicolas Boumal, Bamdev Mishra, P-A Absil, and Rodolphe Sepulchre. Manopt, a matlab toolbox for optimization on manifolds. *The Journal of Machine Learning Research*, 15(1):1455–1459, 2014.

[14] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[15] Michael M Bronstein, Joan Bruna, Taco Cohen, and Petar Veličković. Geometric deep learning: Grids, groups, graphs, geodesics, and gauges. *arXiv preprint arXiv:2104.13478*, 2021.

[16] Yanqing Chen, Timothy A. Davis, William W. Hager, and Sivasankaran Rajamanickam. Algorithm 887: Cholmod, supernodal sparse cholesky factorization and update/downdate. *ACM Trans. Math. Softw.*, 35(3), oct 2008.

[17] Ching Wen Cheng. Transformers.jl: Hugging face models, 2024. Accessed: 2024-09-06.

[18] Miles Cranmer. Interpretable machine learning for science with pysr and symbolicregression.jl. *arXiv preprint arXiv:2305.01582*, 2023.

[19] Miles Cranmer, Sam Greydanus, Stephan Hoyer, Peter Battaglia, David Spergel, and Shirley Ho. Lagrangian neural networks. *arXiv preprint arXiv:2003.04630*, 2020.

[20] George Hotz Daniel and Contributors. tinygrad: Simple and small neural network library in python. `https://github.com/tinygrad/tinygrad`, 2024. GitHub repository.

[21] T. A. Davis. *Direct Methods for Sparse Linear Systems*. SIAM, Philadelphia, PA, 2006.

[22] Timothy A. Davis. Algorithm 832: Umfpack v4.3—an unsymmetric-pattern multifrontal method. *ACM Trans. Math. Softw.*, 30(2):196–199, jun 2004.

[23] Timothy A. Davis. A column pre-ordering strategy for the unsymmetric-pattern multifrontal method. *ACM Trans. Math. Softw.*, 30(2):165–195, jun 2004.

[24] Timothy A. Davis. Algorithm 849: A concise sparse cholesky factorization package. *ACM Trans. Math. Softw.*, 31(4):587–591, dec 2005.

[25] Timothy A. Davis. Algorithm 915, suitesparseqr: Multifrontal multithreaded rank-revealing sparse qr factorization. *ACM Trans. Math. Softw.*, 38(1), dec 2011.

[26] Timothy A. Davis. Algorithm 930: Factorize: An object-oriented linear system solver for matlab. *ACM Trans. Math. Softw.*, 39(4), jul 2013.

[27] Timothy A. Davis. Algorithm 1000: Suitesparse:graphblas: Graph algorithms in the language of sparse linear algebra. *ACM Trans. Math. Softw.*, 45(4), dec 2019.

[28] Timothy A. Davis and Iain S. Duff. An unsymmetric-pattern multifrontal method for sparse lu factorization. *SIAM Journal on Matrix Analysis and Applications*, 18(1):140–158, 1997.

[29] Timothy A. Davis and Iain S. Duff. A combined unifrontal/multifrontal method for unsymmetric sparse matrices. *ACM Trans. Math. Softw.*, 25(1):1–20, mar 1999.

[30] Timothy A. Davis, John R. Gilbert, Stefan I. Larimore, and Esmond G. Ng. Algorithm 836: Colamd, a column approximate minimum degree ordering algorithm. *ACM Trans. Math. Softw.*, 30(3):377–380, sep 2004.

[31] Timothy A. Davis, John R. Gilbert, Stefan I. Larimore, and Esmond G. Ng. A column approximate minimum degree ordering algorithm. *ACM Trans. Math. Softw.*, 30(3):353–376, sep 2004.

[32] Timothy A. Davis and William W. Hager. Modifying a sparse cholesky factorization. *SIAM Journal on Matrix Analysis and Applications*, 20(3):606–627, 1999.

[33] Timothy A. Davis and William W. Hager. Multiple-rank modifications of a sparse cholesky factorization. *SIAM Journal on Matrix Analysis and Applications*, 22(4):997–1013, 2001.

[34] Timothy A. Davis and William W. Hager. Row modifications of a sparse cholesky factorization. *SIAM Journal on Matrix Analysis and Applications*, 26(3):621–639, 2005.

[35] Timothy A. Davis and William W. Hager. Dynamic supernodes in sparse cholesky update/downdate and triangular solves. *ACM Trans. Math. Softw.*, 35(4), feb 2009.

[36] Timothy A. Davis, William W. Hager, Scott P. Kolodziej, and S. Nuri Yeralan. Algorithm 1003: Mongoose, a graph coarsening and partitioning library. *ACM Trans. Math. Softw.*, 46(1), mar 2020.

[37] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1), dec 2011.

[38] Timothy A. Davis and Ekanathan Palamadai Natarajan. Algorithm 907: Klu, a direct sparse solver for circuit simulation problems. *ACM Trans. Math. Softw.*, 37(3), sep 2010.

[39] ChainRules Core Developers. Chainrules.jl. `https://github.com/JuliaDiff/ChainRules.jl`, 2024. GitHub repository.

[40] Mypy Developers. Mypy: Optional static typing for python. `https://github.com/python/mypy`, 2024. Accessed: 2024-09-07.

[41] Iain Dunning, Joey Huchette, and Miles Lubin. Jump: A modeling language for mathematical optimization. *SIAM review*, 59(2):295–320, 2017.

[42] EnzymeAD. Reactant.jl, 2024. Accessed: 2024-11-05.

[43] R Falgout et al. Hypre: Scalable linear solvers and multigrid methods, 2020.

[44] Daniel Foreman-Mackey. An introduction to numpyro. `https://dfm.io/posts/intro-to-numpyro/`, 2020. Accessed: 2024-08-30.

[45] Leslie V. Foster and Timothy A. Davis. Algorithm 933: Reliable calculation of numerical rank, null space bases, pseudoinverse solutions, and basic solutions using suitesparseqr. *ACM Trans. Math. Softw.*, 40(1), oct 2013.

[46] Kaifeng Gao, Gang Mei, Francesco Piccialli, Salvatore Cuomo, Jingzhi Tu, and Zenan Huo. Julia language in machine learning: Algorithms, applications, and open issues. *Computer Science Review*, 37:100254, 2020.

[47] Hong Ge, Kai Xu, and Zoubin Ghahramani. Turing: A language for flexible probabilistic inference. In Amos Storkey and Fernando Perez-Cruz, editors, *Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics*, volume 84 of *Proceedings of Machine Learning Research*, pages 1682–1690. PMLR, 09–11 Apr 2018.

[48] James Halverson and Sneh Pandya. On the generality and persistence of cosmological stasis, 2024.

[49] Eric P. Hanson. Julia's package registration tooling, 2024. Accessed: 2024-10-28. https://ericphanson.com/blog/2024/julias-package-registration-tooling/.

[50] Charles R Harris, K Jarrod Millman, Stéfan J Van Der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. Array programming with numpy. *Nature*, 585(7825):357–362, 2020.

[51] Tom Hennigan, Trevor Cai, Tamara Norman, Lena Martens, and Igor Babuschkin. Haiku: Sonnet for JAX, 2020.

[52] Wen Huang, P-A Absil, Kyle A Gallivan, and Paul Hand. Roptlib: an object-oriented c++ library for optimization on riemannian manifolds. *ACM Transactions on Mathematical Software (TOMS)*, 44(4):1–21, 2018.

[53] Julia Hub. Juliacon 2024 | function room | day 3, 2024. Accessed: 2024-10-24, timestamp 3:13:40.

[54] Julia Hub. Sample form, 2024. Accessed: 2024-10-24. form.jotform.com/241373057274355.

[55] HypothesisWorks. Hypothesis: A python library for property-based testing. `https://github.com/HypothesisWorks/hypothesis`, 2024. Accessed: 2024-09-07.

[56] Michael Innes, Elliot Saba, Keno Fischer, Dhairya Gandhi, Marco Concetto Rudilosso, Neethu Mariya Joy, Tejan Karmali, Avik Pal, and Viral Shah. Fashionable modelling with flux. *arXiv preprint arXiv:1811.01457*, 2018.

[57] Mike Innes. Flux: Elegant machine learning with julia. *Journal of Open Source Software*, 3(25):602, 2018.

[58] Mike Innes, Stefan Karpinski, Viral Shah, David Barber, PLEPS Saito Stenetorp, Tim Besard, James Bradbury, Valentin Churavy, Simon Danisch, Alan Edelman, et al. On machine learning and programming languages. Association for Computing Machinery (ACM), 2018.

[59] Mike Jarvis, Gary Bernstein, and Bhuvnesh Jain. The skewness of the aperture mass statistic. *Monthly Notices of the Royal Astronomical Society*, 352(1):338–352, 2004.

[60] Steven G. Johnson and contributors. Pycall.jl: Calling python functions from the julia language. `https://github.com/JuliaPy/PyCall.jl`, 2024. Accessed: 2024-08-30.

[61] JuliaMath and contributors. Measuretheory.jl, 2024. GitHub repository.

[62] JuliaPy. Pythoncall.jl. `https://github.com/JuliaPy/PythonCall.jl`, 2024. Accessed: 2024-10-22.

[63] Patrick Kidger. *On Neural Differential Equations*. PhD thesis, University of Oxford, 2021.

[64] Logan Kilpatrick. Some julia growth & usage stats, 2024. Accessed: 2024-09-02.

[65] Scott P. Kolodziej, Mohsen Aznaveh, Matthew Bullock, Jarrett David, Timothy A. Davis, Matthew Henderson, Yifan Hu, and Read Sandstrom. The suitesparse matrix collection website interface. *Journal of Open Source Software*, 4(35):1244, 2019.

[66] Tom Kwong, Kristoffer Carlsson, Stefan Karpinski, Elliot Saba, et al. Pkg.jl: Julia's package manager. `https://github.com/JuliaLang/Pkg.jl`, 2023. Accessed: 2024-08-30.

[67] Randy Lai, Simon Byrne, Douglas Bates, Phillip Alday, Viral B. Shah, Milan Bouchet-Valat, Dave Kleinschmidt, Dilum Aluthge, Changcheng Li, Ranjan Anantharaman, Antoine Baldassari, Steven G. Johnson, lbilli, Robert Feldt, Páll Haraldsson, Pietro Monticone, Michael Hatherly, Lukas Elmiger, Iain Dunning, and Rik Huijzer. JuliaInterop/RCall.jl: v0.14.4, August 2024.

[68] The Julia Language. Calling c and fortran code, 2024. Accessed: 2024-09-06.

[69] The Julia Programming Language. Interpretable machine learning with symbolicregression.jl | miles cranmer | juliacon 2023, 2023. Accessed: 2024-10-13.

[70] Wei Li, Ruqing Fang, Junning Jiao, Georgios N Vassilakis, and Juner Zhu. Tutorials: Physics-informed machine learning methods of computing 1d phase-field models. *APL Machine Learning*, 2(3), 2024.

[71] life4. Deal: A python library for design by contract programming. `https://github.com/life4/deal`, 2024. Accessed: 2024-09-07.

[72] Ziming Liu, Yixuan Wang, Sachin Vaidya, Fabian Ruehle, James Halverson, Marin Soljačić, Thomas Y Hou, and Max Tegmark. Kan: Kolmogorov-arnold networks. *arXiv preprint arXiv:2404.19756*, 2024.

[73] Christopher Lourenco, Jinhao Chen, Erick Moreno-Centeno, and Timothy A. Davis. Algorithm 1021: Spex left lu, exactly solving sparse linear systems via a sparse left-looking integer-preserving lu factorization. *ACM Trans. Math. Softw.*, jun 2022.

[74] Lu Lu, Pengzhan Jin, Guofei Pang, Zhongqiang Zhang, and George Em Karniadakis. Learning nonlinear operators via deeponet based on the universal approximation theorem of operators. *Nature machine intelligence*, 3(3):218–229, 2021.

[75] Carlo Lucibello and other contributors. Graphneuralnetworks.jl: a geometric deep learning library for the julia programming language, 2021.

[76] Sean Martin, Andrew M Raim, Wen Huang, and Kofi P Adragni. Manifoldoptim: an r interface to the roptlib library for riemannian manifold optimization. *Journal of Statistical Software*, 93:1–32, 2020.

[77] Wes McKinney et al. pandas: a foundational python library for data analysis and statistics. *Python for high performance and scientific computing*, 14(9):1–9, 2011.

[78] Aaron Meurer, Christopher P Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K Moore, Sartaj Singh, et al. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, 2017.

[79] Leo A Meyerovich and Ariel S Rabkin. Empirical analysis of programming language adoption. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pages 1–18, 2013.

[80] Microsoft. Pyright: Static type checker for python, 2023. Accessed: 2024-09-07.

[81] Patrick K. Mogensen and Asbjørn N. Riseth. Optim: A mathematical optimization package for julia. *Journal of Open Source Software*, 3(24):615, 2018.

[82] William S. Moses, Valentin Churavy, Ludger Paehler, Jan Hückelheim, Sri Hari Krishna Narayanan, Michel Schanen, and Johannes Doerfert. Reverse-mode automatic differentiation and optimization of gpu kernels via enzyme. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[83] Radu Muschevici, Alex Potanin, Ewan Tempero, and James Noble. Multiple dispatch in practice. *Acm sigplan notices*, 43(10):563–582, 2008.

[84] James. W. Nightingale, Richard G. Hayes, and Matthew Griffiths. 'pyautofit': A classy probabilistic programming language for model composition and fitting. *Journal of Open Source Software*, 6(58):2550, 2021.

[85] ROYUD Nishino and Shohei Hido Crissman Loomis. Cupy: A numpy-compatible library for nvidia gpu calculations. *31st confernce on neural information processing systems*, 151(7), 2017.

[86] SciML Organization. Sciml roadmap, 2024. Accessed: 2024-09-03.

[87] Ian Osband, Zheng Wen, Seyed Mohammad Asghari, Vikranth Dwaracherla, Morteza Ibrahimi, Xiuyuan Lu, and Benjamin Van Roy. Epistemic neural networks. *Advances in Neural Information Processing Systems*, 36:2795–2823, 2023.

[88] Aayush Pal. Lux: Explicit parameterization of deep neural networks in julia (v0.4.50), 2023.

[89] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[90] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

[91] Du Phan, Neeraj Pradhan, and Martin Jankowiak. Composable effects for flexible and accelerated probabilistic programming in numpyro. *arXiv preprint arXiv:1912.11554*, 2019.

[92] Christopher Rackauckas, Yingbo Ma, Julius Martensen, Collin Warner, Kirill Zubov, Rohit Supekar, Dominic Skinner, and Ali Ramadhan. Universal differential equations for scientific machine learning. *arXiv preprint arXiv:2001.04385*, 2020.

[93] Christopher Rackauckas and Qing Nie. Differentialequations. jl–a performant and feature-rich ecosystem for solving differential equations in julia. *Journal of open research software*, 5(1):15–15, 2017.

[94] Steven C. Rennich, Darko Stosic, and Timothy A. Davis. Accelerating sparse cholesky factorization on gpus. *Parallel Computing*, 59:140–150, 2016. Theory and Practice of Irregular Applications.

[95] Jarrett Revels, Mike Innes, Lyndon White, et al. Diffrules.jl. `https://github.com/JuliaDiff/DiffRules.jl`, 2018. Accessed: 2024-10-13.

[96] Jarrett Revels, Miles Lubin, and Theodore Papamarkou. Forward-mode automatic differentiation in julia. *arXiv preprint arXiv:1607.07892*, 2016.

[97] David M Rosen, Luca Carlone, Afonso S Bandeira, and John J Leonard. Se-sync: A certifiably correct algorithm for synchronization over the special euclidean group. *The International Journal of Robotics Research*, 38(2-3):95–125, 2019.

[98] David M Rosen, Luca Carlone, Afonso S Bandeira, and John J Leonard. A certifiably correct algorithm for synchronization over the special euclidean group. In *Algorithmic Foundations of Robotics XII: Proceedings of the Twelfth Workshop on the Algorithmic Foundations of Robotics*, pages 64–79. Springer, 2020.

[99] Yousef Saad. Iterative methods for sparse linear systems.

[100] Frank Schäfer, Mohamed Tarek, Lyndon White, and Chris Rackauckas. Abstractdifferentiation. jl: Backend-agnostic differentiable programming in julia. *NeurIPS 2021 Differentiable Programming Workshop*, 2021.

[101] Philip Schanely. Crosshair: An analysis tool for python that uses symbolic execution. `https://github.com/pschanely/CrossHair`, 2024. Accessed: 2024-09-07.

[102] Chad Scherrer and Moritz Schauer. Applied measure theory for probabilistic modeling. *Proceedings of the JuliaCon Conferences*, 1(1):92, 2022.

[103] Conner Scherrer and contributors. Soss.jl, 2024. GitHub repository.

[104] Noah Shinn. Solvent. `https://github.com/noahshinn/solvent`, 2024. Accessed: 2024-09-04.

[105] Ashwin Prasad Shivarpatna Venkatesh, Samkutty Sabu, Jiawei Wang, Amir M. Mir, Li Li, and Eric Bodden. Typeevalpy: A micro-benchmarking framework for python type inference tools. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, pages 49–53, 2024.

[106] The Julia Language. Sparsearrays module. `https://docs.julialang.org/en/v1/stdlib/SparseArrays/`, 2024. Accessed: 2024-09-01.

[107] James Townsend, Niklas Koep, and Sebastian Weichwald. Pymanopt: A python toolbox for optimization on manifolds using automatic differentiation. *Journal of Machine Learning Research*, 17(137):1–5, 2016.

[108] Yueh-Hua Tu. Geometricflux. jl: a geometric deep learning library in julia. *Proceedings of JuliaCon*, 1:1, 2020.

[109] Pauli Virtanen, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, et al. Scipy 1.0: fundamental algorithms for scientific computing in python. *Nature methods*, 17(3):261–272, 2020.

[110] Tianshi Xu, Vassilis Kalantzis, Ruipeng Li, Yuanzhe Xi, Geoffrey Dillon, and Yousef Saad. pargemslr: A parallel multilevel schur complement low-rank preconditioning and solution package for general sparse matrices. *Parallel Computing*, 113:102956, 2022.

[111] Sencer Nuri Yeralan, Timothy A. Davis, Wissam M. Sid-Lakhdar, and Sanjay Ranka. Algorithm 980: Sparse qr factorization on the gpu. *ACM Trans. Math. Softw.*, 44(2), aug 2017.

[112] Zihan Zou, Yujin Zhang, Lijun Liang, Mingzhi Wei, Jiancai Leng, Jun Jiang, Yi Luo, and Wei Hu. A deep learning model for predicting selected organic molecular spectra. *Nature Computational Science*, 3:1–8, 11 2023.

[113] Kirill Zubov, Zoe McCarthy, Yingbo Ma, Francesco Calisto, Valerio Pagliarino, Simone Azeglio, Luca Bottero, Emmanuel Luján, Valentin Sulzer, Ashutosh Bharambe, Nand Vinchhi, Kaushik Balakrishnan, Devesh Upadhyay, and Chris Rackauckas. Neuralpde: Automating physics-informed neural networks (pinns) with error approximations, 2021.