Journal of Machine Learning Research 27 (2026) 1-13 Submitted 5/25; Revised 1/26; Published 1/26

A Common Interface for Automatic Differentiation

Guillaume Dalle GUILLAUME.DALLE@ENPC.FR
LVMT, ENPC, Institut Polytechnique de Paris, Univ Gustave Eiffel, Marne-la-Vallée, France

Adrian Hill HILL@TU-BERLIN.DE
BIFOLD – Berlin Institute for the Foundations of Learning and Data, Berlin, Germany
Machine Learning Group, Technical University of Berlin, Berlin, Germany

Editor: Joaquin Vanschoren

Abstract

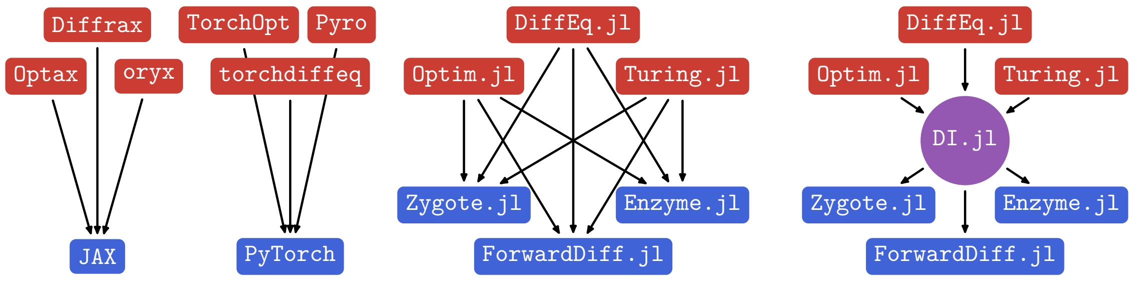
For scientific machine learning tasks with a lot of custom code, picking the right Automatic Differentiation (AD) system matters. Our Julia package `DifferentiationInterface.jl` provides a common frontend to a dozen AD backends, unlocking easy comparison and modular development. In particular, its built-in preparation mechanism leverages the strengths of each backend by amortizing one-time computations. This is key to enabling sophisticated features like sparsity handling without putting additional burdens on the user.

Keywords: automatic differentiation, differentiable programming, scientific computing, Julia programming language

1. Motivation

Automatic Differentiation (AD) is a cornerstone of modern machine learning (Baydin et al., 2018). By generating derivatives directly from computer code, AD obviates the need for manual differentiation of complex algorithms. While this separation of concerns enables quick prototyping, it requires compatibility between each application and one or more AD systems. For standardized tasks like deep learning, AD is often used as part of an integrated framework, which makes compatibility straightforward. Python programmers may pick `TensorFlow` (Abadi et al., 2015), `PyTorch` (Paszke et al., 2019), or `JAX` (Bradbury et al., 2018), then write all of their code within the boundaries of that framework, from neural layers to optimization routines. Yet if the task is not standardized, and if the software stack is not already set in stone (e.g., by technical or collaboration constraints), "shopping around" for the best AD solution can be hugely beneficial.

Indeed, not all AD libraries offer the same mutation support, looping abilities, branching behavior, or sparsity handling. These variations impact performance when the function to differentiate is not a typical neural network. A crucial example is scientific machine learning, which seeks to build differentiable models of physical processes. The modeling code might involve non-vectorizable procedures, in-place memory updates, custom iterations, nested control flow, and scalar indexing, which are a struggle for frameworks revolving around deep learning. When one cannot pick the best AD system *a priori*, perhaps one should write the code first, and compare options later? In the present paper, we show that this is possible.



(a) Python (b) Julia (before DI) (c) Julia (now)

Figure 1: Comparison of the AD ecosystems in Python and Julia for applications to optimization, differential equations, and probabilistic programming

Since its inception a decade ago, the Julia language (Bezanson et al., 2017) has emerged as a worthy contender for numerical computing and scientific machine learning. In that community, the freedom to choose between AD systems has been a longstanding goal (Sapienza et al., 2024). This flexibility is a social necessity, because Julia's AD landscape is less centralized and well-funded than Python's, leaving key packages in the hands of students or academics with temporary positions. Luckily, it is also within reach, because Julia's built-in arrays and operations are sped up by just-in-time compilation. As a result, AD does not need to target a very specific subset of accelerated primitives (like `jax.numpy`): in theory, the whole language lends itself to differentiation, which facilitates separation between AD and its uses. In practice, of course, each AD library imposes a different set of tradeoffs.

To facilitate comparison, the missing ingredient was a common API compatible with nearly every AD package in Julia. Hence we developed `DifferentiationInterface.jl`,[1] or `DI` for short, a unified frontend to a dozen AD backends (see Appendix A for a list). It acts as a single entry point for AD users, abstracting away implementation details to focus on the desired output. Such a common interface offers flexibility and modularity, without compromising on speed. The benefits of this approach are synthesized in Figure 1: for common machine learning tasks, the number of necessary bindings decreases drastically.

2. Design principles

The main concepts underpinning `DI` are backends, operators, and preparation. While the first two were inspired by a previous proof of concept (Schäfer et al., 2022), we argue that preparation is the key novelty unlocking peak performance and widespread adoption.

Backends. A backend is a Julia object which represents a choice of AD package, combined with package-specific parameters (e.g., the differentiation mode, or the taping behavior). Backend types are already defined in a widely-used third-party package called `ADTypes.jl`,[2] which `DI` leverages instead of introducing a new standard. Thanks to Julia's multiple dispatch mechanism, user code is specialized on the provided backend to generate efficient

1. `https://github.com/JuliaDiff/DifferentiationInterface.jl`
2. `https://github.com/SciML/ADTypes.jl`

```julia
using DifferentiationInterface
# change 2 lines to switch backends
import ForwardDiff
back = AutoForwardDiff()

f(x) = sum(abs2, x)
x = float.(1:5)
g = gradient(f, back, x)
```

```julia
gradient!(f, g, back, x)  # in-place
y, g = value_and_gradient(f, back, x)  # primal
```

```julia
prep = prepare_gradient(f, back, zero(x))
for iteration in 1:1000
    x -= 0.1 * gradient(f, prep, back, x)  # fast
end
```

Listing 1: Example uses of `DI`

assembly. A minimal example is displayed in Listing 1 (left), where we compute the gradient of the squared Euclidean norm with `ForwardDiff.jl` (Revels et al., 2016). In this code sample, only two lines need to change for a different AD package to be used instead (see Appendix C.1 for a more involved example).

Operators. `DI` provides a set of 8 differentiation operators. First-order operators are **`pushforward`** (Jacobian-vector product), **`pullback`** (vector-Jacobian product), **`derivative`**, **`gradient`**, and **`jacobian`**. Second-order operators are **`second_derivative`**, **`hvp`** (Hessian-vector product), and **`hessian`**. Each operator possesses 4 variants: in-place and out-of-place, with or without primal output, as demonstrated in Listing 1 (top right). If an AD package does not natively provide a given operator, `DI` takes over with a default chain of fallbacks until reaching the lowest-level operators `pushforward` and `pullback`. For instance, (1) `gradient` relies on `pullback`, (2) `hvp` combines `pushforward` with `gradient`, and (3) `hessian` calls `hvp` once per input dimension.

Preparation. AD is very useful inside iterative procedures like gradient descent, where the same function is differentiated many times on different inputs. In such settings, it is worth preparing for repeated differentiation, paying a one-time cost to speed up each subsequent execution (sometimes by orders of magnitude, see Appendix C.3). This initial step takes different forms depending on the AD package. For some, preparation will record an execution tape or perform the source transformation. For others, it will precompute useful data like basis vectors, or preallocate caches, or even perform symbolic simplification.

`DI`'s most valuable contribution consists in hiding this complexity from users, who are only asked to provide a typical input with the correct type and size (e.g., "I want to prepare the gradient of my function for double-precision vectors of length 10"). Then, whatever information or memory the AD package needs is encapsulated in the result of preparation, and can be reused as many times as necessary. The corresponding syntax is demonstrated in Listing 1 (bottom right). By encouraging `DI` users to adopt this preparation mechanism across their computationally intensive tasks, we set the stage for arbitrarily complex AD methods to be used transparently. In particular, this is crucial for the efficient application of sparse AD techniques (see below).

3. Supported features

Not only is it easier to call each package with `DI`, the interface also delivers additional abilities. Here we list a few features which are natively supported by a subset of AD packages, but which `DI` makes easily accessible for all (or most) of them.

Contexts. When the function to differentiate takes several arguments, usually not all derivatives are needed. `DI` assumes that only the first argument is actually differentiated (a common limitation of AD packages in Julia), but it supports additional non-differentiated arguments of two types: `Constants` and `Caches`. The first kind is dedicated to fixed parameters, whereas the second kind is for buffer storage which gets overwritten by the function to avoid new allocations.

Sparsity. `DI` allows the efficient computation of sparse Jacobian and Hessian matrices, using techniques reviewed in Gebremedhin et al. (2005). This functionality is made possible by two additional packages: `SparseConnectivityTracer.jl`[3] for sparsity pattern detection (Hill and Dalle, 2025), and `SparseMatrixColorings.jl`[4] for coloring problems (Montoison et al., 2025). Both of these preliminary steps happen during the preparation phase, so that their high cost is amortized by subsequent computations.

Backend combination. A single differentiation mode is not always enough. For instance, HVPs and Hessian matrices are most efficiently computed in forward-over-reverse mode (Pearlmutter, 1994; Dagréou et al., 2024). When individual AD packages do not include both modes, `DI` supports the creation of a `SecondOrder` object to stack different backends and achieve the required behavior. Similarly, the `MixedMode` wrapper is used for sparse Jacobians that combine forward and reverse passes.

Backend translation. Some functions are differentiable with one AD backend, but not the other. In such cases, writing custom rules for every package is tedious. Instead, `DI` includes utilities for translating between packages, essentially saying "when package A tries to differentiate this function, use package B under the hood". This is implemented as a wrapper `DifferentiateWith(f, other_backend)`.

Testing. The main appeal of having access to several AD packages is to figure out which one is best for a given application. To make this easier, `DI` comes with a sibling package `DifferentiationInterfaceTest.jl` (or `DIT`), which contains testing and benchmarking functionality. The user only has to define a testing scenario, and they can quickly compare correctness and speed between the candidate backends, with standardized result reporting (see Appendix C.2 for a demonstration).

4. Perspectives

So far, `DI` has mostly been developed for use cases in scientific machine learning, where problems have a moderate dimension and parallelism is hard to achieve due to numerous scalar operations. A natural prospect would be to improve GPU support and testing, which is crucial for large-scale workloads. Another avenue for extension is backend-agnostic

3. `https://github.com/adrhill/SparseConnectivityTracer.jl`
4. `https://github.com/gdalle/SparseMatrixColorings.jl`

definition of custom rules. `DI` is a unified way to call AD, but it does not yet include rule-building utilities, which vary much more between backends. A first common interface was already introduced by `ChainRules.jl` (White et al., 2025), but the advent of mutation-friendly AD systems like `Enzyme.jl` (Moses and Churavy, 2020; Moses et al., 2021) and `Mooncake.jl` (Tebbutt and Ge, 2026) brings new challenges for standardization.

Acknowledgments

Adrian Hill acknowledges funding from the German Federal Ministry of Education and Research under the grant BIFOLD25B.

The authors want to thank the people who have inspired `DI`, contributed to it, or tried it out themselves. In alphabetical order, those include: Fredrik Bagge Carlson, Valentin Churavy, Jadon Clugston, Vaibhav Dixit, Francis Gagnon, Hong Ge, Patrick Mogensen, Alexis Montoison, William S. Moses, Avik Pal, Qingyu Qu, Chris Rackauckas, Frank Schäfer, Niklas Schmitz, Mohamed Tarek, Will Tebbutt, Frames Catherine White, Penelope Yong, and many others.

Appendix A. Supported AD packages

Table 1 lists every AD package that `DI` provides an interface to. Together, these cover a large majority of AD use cases in Julia (see Sapienza et al. (2024) for a recent review of the ecosystem). The taxonomy of paradigms is taken from Margossian (2019).

Package	**Paradigm**	**Modes**	**Reference**
`ChainRules.jl`	Source transformation	Both	White et al. (2025)
`Enzyme.jl`	Source transformation	Both	Moses and Churavy (2020) Moses et al. (2021)
`FastDifferentiation.jl`	Symbolic differentiation	-	
`FiniteDiff.jl`	Finite differences	-	
`FiniteDifferences.jl`	Finite differences	-	
`ForwardDiff.jl`	Operator overloading	Forward	Revels et al. (2016)
`GTPSA.jl`	Operator overloading	Forward	
`Mooncake.jl`	Source transformation	Reverse	Tebbutt and Ge (2026)
`PolyesterForwardDiff.jl`	Operator overloading	Forward	Mester et al. (2022)
`ReverseDiff.jl`	Operator overloading	Reverse	
`Symbolics.jl`	Symbolic differentiation	-	Gowda et al. (2022)
`Tracker.jl`	Operator overloading	Reverse	
`Zygote.jl`	Source transformation	Reverse	Innes (2019) Innes et al. (2019)

Table 1: List of AD packages supported by `DI`

Appendix B. Limitations

To an observer familiar with other AD systems, `DI`'s API might seem rather unexpected. Some of its limitations are listed below:

1. Applying the (optional) preparation mechanism and propagating its result through user code is often necessary for good performance, which goes against functional programming paradigms and complicates nested AD.

2. Only one of the function's arguments can be differentiated, the other ones being reduced to contexts.

3. Official support and testing is limited to number and array inputs/outputs, as opposed to more involved structs.

4. Backend objects live in a different package (namely `ADTypes.jl`), whose development can sometimes precede `DI`'s concrete implementation.

To understand these pain points, it is helpful to remember that `DI` emerged *a posteriori*, on top of an existing language and package ecosystem on which we have limited agency. For example:

1. Julia is not a functional programming language. On the contrary, idiomatic Julia code places strong emphasis on preallocation and reuse of memory to increase performance, which is why many backends already offer such an option (`DI` only wraps it). For some backends, preparation is even used to generate the differentiated code at runtime, which makes it absolutely essential (although other backends like `Enzyme.jl` perform this step at compile-time).

2. Julia's historical AD systems (`ForwardDiff.jl`, `ReverseDiff.jl`, `FiniteDiff.jl`) only provide functionality for a single active argument, which is why `DI` keeps that common denominator. Such a simple API already caters to a significant portion of user needs, as evidenced by package adoption.

3. Since `DI` needs to ensure proper testing across all backends and operators, additional iteration over a wide variety of types would make continuous integration impractical (each testing run already takes over an hour, using multiple runners). In addition, forward-mode backends usually do not support arbitrary types, since they need a way to iterate over input dimensions when computing gradients. Note that reverse-mode backends are usually able to handle more complicated objects, but it is not `DI`'s stated role to define and test the boundaries of support for each one.

4. The `ADTypes.jl` package and backend types existed before `DI` came along: they were heavily used inside the SciML ecosystem[5] in particular. To facilitate the transition for end users, we chose to adopt and extend that framework (for example with new backends and sparse functionality) instead of starting from scratch.

Now that `DI` has been developed and integrated, future AD libraries might take the reverse approach, treating `DI` as an *a priori* standard to implement. Such a decision was already made by the recent `Mooncake.jl`. In parallel, `DI` itself will hopefully keep evolving to cover an increasing diversity of use cases.

Appendix C. More examples

The code examples displayed below were run on a 2023 MacBook Pro M3 using Julia v1.11.5. The API corresponds to the versions `DI` v0.6.52 and `DIT` v0.9.6, it may change in future releases (following semantic versioning).

C.1 Composability

In Listing 2, we demonstrate the ease of switching the backend object inside a Hessian computation. First, we use a simple forward-mode backend, which is slow in high dimension (top frame). Second, we improve performance by switching to forward-over-reverse mode with `SecondOrder` (middle frame). Third, we further speed things up by leveraging sparsity with `AutoSparse` (bottom frame). This modularity allows the user to experiment and find the best AD option in a given scenario, without diving into the documentation of each package.

5. `https://sciml.ai/`

```julia
using DifferentiationInterface, Chairmarks
import ForwardDiff, ReverseDiff
using SparseArrays, SparseConnectivityTracer, SparseMatrixColorings

forward = AutoForwardDiff()
reverse = AutoReverseDiff(; compile=true)
sparsity_detector = TracerSparsityDetector()
coloring_algorithm = GreedyColoringAlgorithm()

f(x) = sum(diff(x) .^ 2)
xsmall, xbig, x0big = float.(1:5), float.(1:1000), zeros(1000)
```

```julia
back1 = forward

p1 = prepare_hessian(f, back1, x0big)
```

```
julia> hessian(f, back1, xsmall)
5×5 Matrix{Float64}:
  2.0  -2.0   0.0   0.0   0.0
 -2.0   4.0  -2.0   0.0   0.0
  0.0  -2.0   4.0  -2.0   0.0
  0.0   0.0  -2.0   4.0  -2.0
  0.0   0.0   0.0  -2.0   2.0

julia> @b hessian($f, $p1, $back1, $xbig)
5.463 s (42620 allocs: 18.106 GiB, ...)
```

```julia
back2 = SecondOrder(forward, reverse)

p2 = prepare_hessian(f, back2, x0big)
```

```
julia> hessian(f, back2, xsmall)
5×5 Matrix{Float64}:
  2.0  -2.0   0.0   0.0   0.0
 -2.0   4.0  -2.0   0.0   0.0
  0.0  -2.0   4.0  -2.0   0.0
  0.0   0.0  -2.0   4.0  -2.0
  0.0   0.0   0.0  -2.0   2.0

julia> @b hessian($f, $p2, $back2, $xbig)
91.667 ms (3780 allocs: 354.360 MiB, ...)
```

```julia
back3 = AutoSparse(
    SecondOrder(forward, reverse);
    sparsity_detector,
    coloring_algorithm
)

p3 = prepare_hessian(f, back3, x0big)
```

```
julia> hessian(f, back3, xsmall)
5×5 SparseMatrixCSC{Float64, Int64}
with 13 stored entries:
  2.0  -2.0    ⋅     ⋅     ⋅
 -2.0   4.0  -2.0    ⋅     ⋅
   ⋅   -2.0   4.0  -2.0    ⋅
   ⋅     ⋅   -2.0   4.0  -2.0
   ⋅     ⋅     ⋅   -2.0   2.0

julia> @b hessian($f, $p3, $back3, $xbig)
116.417 μs (9 allocs: 55.188 KiB)
```

Listing 2: Switching backends within `DI`

```julia
using DifferentiationInterface, DifferentiationInterfaceTest
import Enzyme, ForwardDiff, Mooncake, ReverseDiff, Zygote

f(x) = sum(abs2, x)
x_candidates = [rand(10), rand(100), rand(1_000), rand(10_000)]

scenarios = [Scenario{:gradient,:out}(f, x; res1=2x) for x in x_candidates]

backends = [
    AutoEnzyme(; mode=Enzyme.Reverse),
    AutoForwardDiff(),
    AutoMooncake(; config=nothing),
    AutoReverseDiff(; compile=true),
    AutoZygote(),
]

df = benchmark_differentiation(backends, scenarios; benchmark=:full)
```

Listing 3: Comparing backends with `DIT`

C.2 Benchmarking

In Listing 3, we showcase the functionalities of `DIT` for comparing performance across various backends. Users are free to define arbitrary scenarios on which to test differentiation operators. Here, we measure the time it takes to compute the gradient of the squared Euclidean norm, for inputs of increasing dimension. In addition to benchmarking, `DIT` provides similar utilities to check derivative correctness against a reference value.

C.3 Preparation impact

The runtime measurements generated by Listing 3 are displayed in Figure 2, and they show that the influence of preparation is very backend-dependent. Of course, it also depends on the test function and the differentiation operator. In this specific case, `Enzyme.jl` and `Zygote.jl` do not benefit from preparation at all. Meanwhile, the other three backends are greatly sped up by preparation, due to the way they work internally. `ForwardDiff.jl` uses preparation to pre-allocate the necessary memory for repeated forward passes, while `Mooncake.jl` performs source transformation and `ReverseDiff.jl` compiles an execution tape. Note that the user does not need to know any of these implementation details to exploit the full abilities of these packages, or to compare them.

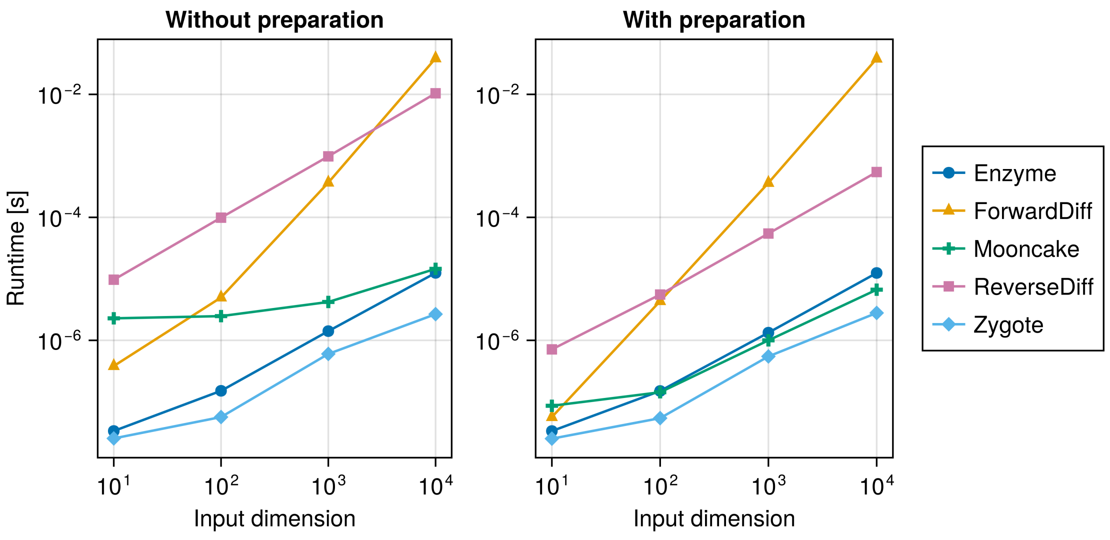


Figure 2: Impact of preparation on gradient performance of $f : x \mapsto \|x\|^2$

References

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL https://www.tensorflow.org/.

Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic Differentiation in Machine Learning: A Survey. *Journal of Machine Learning Research*, 18(153):1–43, 2018. ISSN 1533-7928. URL http://jmlr.org/papers/v18/17-468.html.

Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A Fresh Approach to Numerical Computing. *SIAM Review*, 59(1):65–98, January 2017. ISSN 0036-1445, 1095-7200. doi: 10.1137/141000671. URL https://epubs.siam.org/doi/10.1137/141000671.

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. URL http://github.com/google/jax.

Mathieu Dagréou, Pierre Ablin, Samuel Vaiter, and Thomas Moreau. How to compute Hessian-vector products? In *The Third Blogpost Track at ICLR 2024*, February 2024. URL https://openreview.net/forum?id=rTgjQtGP3O.

Assefaw Hadish Gebremedhin, Fredrik Manne, and Alex Pothen. What Color Is Your Jacobian? Graph Coloring for Computing Derivatives. *SIAM Review*, 47(4):629–705, January 2005. ISSN 0036-1445. doi: 10/cmwds4. URL https://epubs.siam.org/doi/abs/10.1137/S0036144504444711.

Shashi Gowda, Yingbo Ma, Alessandro Cheli, Maja Gwóźzdź, Viral B. Shah, Alan Edelman, and Christopher Rackauckas. High-performance symbolic-numerics via multiple dispatch. *ACM Commun. Comput. Algebra*, 55(3):92–96, January 2022. ISSN 1932-2232. doi: 10.1145/3511528.3511535. URL https://dl.acm.org/doi/10.1145/3511528.3511535.

Adrian Hill and Guillaume Dalle. Sparser, better, faster, stronger: Sparsity detection for efficient automatic differentiation. *Transactions on Machine Learning Research*, 2025. ISSN 2835-8856. URL https://openreview.net/forum?id=GtXSN52nIW.

Michael Innes. Don't Unroll Adjoint: Differentiating SSA-Form Programs, March 2019. URL http://arxiv.org/abs/1810.07951.

Mike Innes, Alan Edelman, Keno Fischer, Chris Rackauckas, Elliot Saba, Viral B. Shah, and Will Tebbutt. A Differentiable Programming System to Bridge Machine Learning and Scientific Computing, July 2019. URL http://arxiv.org/abs/1907.07587.

Charles C. Margossian. A review of automatic differentiation and its efficient implementation. *WIREs Data Mining and Knowledge Discovery*, 9(4):e1305, 2019. ISSN 1942-4795. doi: 10.1002/widm.1305. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/widm.1305.

Rachel Mester, Alfonso Landeros, Chris Rackauckas, and Kenneth Lange. Differential methods for assessing sensitivity in biological models. *PLOS Computational Biology*, 18(6): e1009598, June 2022. ISSN 1553-7358. doi: 10.1371/journal.pcbi.1009598. URL https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1009598.

Alexis Montoison, Guillaume Dalle, and Assefaw Gebremedhin. Revisiting Sparse Matrix Coloring and Bicoloring, May 2025. URL http://arxiv.org/abs/2505.07308.

William Moses and Valentin Churavy. Instead of Rewriting Foreign Code for Machine Learning, Automatically Synthesize Fast Gradients. In *Advances in Neural Information Processing Systems*, volume 33, pages 12472–12485. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper/2020/hash/9332c513ef44b682e9347822c2e457ac-Abstract.html.

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, pages 1–16, New York, NY, USA, November 2021. Association for Computing Machinery. ISBN 978-1-4503-8442-1. doi: 10.1145/3458817.3476165. URL https://doi.org/10.1145/3458817.3476165.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

Barak A. Pearlmutter. Fast Exact Multiplication by the Hessian. *Neural Computation*, 6(1):147–160, January 1994. ISSN 0899-7667. doi: 10.1162/neco.1994.6.1.147. URL https://ieeexplore.ieee.org/abstract/document/6796137.

Jarrett Revels, Miles Lubin, and Theodore Papamarkou. Forward-Mode Automatic Differentiation in Julia, July 2016. URL http://arxiv.org/abs/1607.07892.

Facundo Sapienza, Jordi Bolibar, Frank Schäfer, Brian Groenke, Avik Pal, Victor Boussange, Patrick Heimbach, Giles Hooker, Fernando Pérez, Per-Olof Persson, and Christopher Rackauckas. Differentiable Programming for Differential Equations: A Review, June 2024. URL http://arxiv.org/abs/2406.09699.

Frank Schäfer, Mohamed Tarek, Lyndon White, and Chris Rackauckas. AbstractDifferentiation.jl: Backend-Agnostic Differentiable Programming in Julia, February 2022. URL http://arxiv.org/abs/2109.12449.

Will Tebbutt and Hong Ge. Mooncake: Towards a Differentiable General-Purpose Language, January 2026. URL https://github.com/chalk-lab/Mooncake.jl.

Frames White, Michael Abbott, Jarrett Revels, Miha Zgubic, Seth Axen, Alex Arslan, Simeon David Schaub, Nick Robinson, Yingbo Ma, Sam, Christopher Rackauckas, Niklas Heim, David Widmann, Gaurav Dhingra, Will Tebbutt, Niklas Schmitz, Mason Protter, Carlo Lucibello, Keno Fischer, Neven Sajko, Rainer Heintzmann, frankschae, Andreas Noack, Anton Smirnov, Andrei Zhabinski, mattBrzezinski, Rory Finnegan, and Jerry Ling. JuliaDiff/ChainRules.jl: V1.72.3. Zenodo, February 2025. URL https://zenodo.org/records/14926720.