AbstractDifferentiation.jl: Backend-Agnostic Differentiable Programming in Julia

Anonymous Author(s)
Affiliation
Address
`email`

Abstract

No single Automatic Differentiation (AD) system is the optimal choice for all problems. This means informed selection of an AD system and combinations can be a problem-specific variable that can greatly impact performance. In the Julia programming language, the major AD systems target the same input and thus in theory can compose. Hitherto, switching between AD packages in the Julia Language required end-users to familiarize themselves with the user-facing API of the respective packages. Furthermore, implementing a new, usable AD package required AD package developers to write boilerplate code to define convenience API functions for end-users. As a response to these issues, we present AbstractDifferentiation.jl for the automatized generation of an extensive, unified, user-facing API for any AD package. By splitting the complexity between AD users and AD developers, AD package developers only need to implement one or two primitive definitions to support various utilities for AD users like Jacobians, Hessians and lazy product operators from native primitives such as pullbacks or pushforwards, thus removing tedious – but so far inevitable – boilerplate code, and enabling the easy switching and composing between AD implementations for end-users.

1 Introduction

Differentiable programming (∂P), i.e., the ability to differentiate general computer program structures, has enabled the efficient combination of existing packages for scientific computation and machine learning [Raissi et al., 2019, Rackauckas et al., 2020a, de Avila Belbute-Peres et al., 2018]. Black-box machine learning approaches are flexible but require a large amount of data. Incorporating scientific knowledge about the structure of a problem via ∂P reduces the amount of data needed. It allows the learning task to be simplified, for example, by focusing on learning only the parts of the model that are missing [Rackauckas et al., 2020b, Dandekar et al., 2020]. There are already many examples where such differentiable frameworks have provided performance and accuracy advantages over black-box approaches to machine learning, including but not limited to protein-folding [AlQuraishi, 2018, Ingraham et al., 2018], fluid dynamics [Schenck and Fox, 2018], robotics [Schenck and Fox, 2018], and quantum control [Schäfer et al., 2020, 2021].

∂P is (commonly) realized by automatic differentiation (AD), a family of techniques to efficiently and accurately differentiate numeric functions expressed as computer programs. Generally, besides forward- and reverse-mode AD, the two main AD branches, many software implementations with different pros and cons exist. Some AD software implementations work at a lower level code representation, possibly mixing in LLVM-level compiler passes, to

Submitted to 35th Conference on Neural Information Processing Systems (NeurIPS 2021). Do not distribute.

fully optimize scalar operations [Revels et al., 2016, Moses and Churavy, 2020] while others perform transformations at a higher level to keep linear algebra operations intact for optimal usage of BLAS primitives [Innes et al., 2019, Paszke et al., 2017]. The goal is to make the best choice of AD system in every part of the program without requiring users to extensively contort their code to the differing APIs.

The AD landscape of the Julia programming language is developed in a manner in which composability between the AD systems is possible. While many automatic differentiation systems require specific formulations of the code, for example PyTorch using an alternative implementation of the NumPy API known as torch.numpy [Paszke et al., 2017] with torch.tensor and similarly for Jax with jax.numpy [Bradbury et al., 2018] each differing from the original NumPy [Oliphant, 2006] API in subtle ways with different numerical properties, the Julia AD systems generally act directly on the standard Julia syntax, with its standard library, array implementation, its standard GPU acceleration tools [Besard et al., 2018], and more. This has previously been shown to allow packages in Julia which were developed without knowledge of AD systems to be fully differentiable without modification by multiple different tools [Rackauckas et al., 2020a]. Furthermore, Julia has a common ground on which differentiation rules are defined, ChainRules.jl [White et al., 2021], which is shared amongst the AD packages. This empowers the idea of a “glue AD” system [Rackauckas] where software library authors define ChainRules overloads to add domain insight into the automatic differentiation process without tying to one particular AD system.

However, switching from one backend to another on the user side can still be tedious because the user has to learn and adapt the code towards the user-facing API of the new AD package. Similarly, for the author of the AD package defining an extensive API supporting every possible differentiation use case requires a lot of boilerplate code, e.g. to define the Jacobian function, Jacobian-vector product, Hessian, Hessian-vector product, etc. Defining all of these functions for each AD implementation is tedious and unnecessary since the relationship between these functions is abstract and not implementation-specific. Therefore, while in theory switching between AD systems can be trivially done, in practice the competing APIs of the various AD mechanisms has limited its use throughout the language’s ecosystem.

The Julia Language [Bezanson et al., 2012] has over a dozen automatic differentiation packages [White]. Different packages have different user interfaces and offer different tradeoffs. Popular systems include:

1. ForwardDiff.jl [Revels et al., 2016], an operator-overloading-based, forward-mode AD implementation, with many years of extensive use and thus very high reliability
2. ReverseDiff.jl [Revels, 2018], an operator-overloading-based, reverse-mode AD implementation, featuring several tape-based optimizations
3. Zygote.jl [Innes et al., 2019], a reverse-mode AD implementation that does source code transformation to generate the derivative’s code from the function’s code, operating at the level of Julia’s intermediate representation. Zygote is therefore able to handle arbitrary Julia code but is unable to handle mutation.
4. Enzyme.jl [Moses and Churavy, 2020], a reverse-mode AD implementation that runs by source code transformation at the LLVM level, with excellent performance on scalar operations, but at present lesser performance on large matrix operations.
5. Diffractor.jl [Fischer], a new source-to-source AD package promising high performance on both scalar and vector/tensor code

A more detailed summary of the strengths and limitations of different AD packages is given in Appendix A.

Each of these AD systems (and each of the many others) has its own unique set of advantages and disadvantages. Additionally, all of them only define API functions for a subset of all the possible differentiation use cases, often requiring users to do package-specific implementations of quantities like Jacobian-vector product or Hessian-vector product when needed. Beside the existing stable AD implementations, any new implementation may or may not be mature enough to handle perturbation confusion properly [Siskind and Pearlmutter, 2005, Manzyuk et al., 2019] which prevents one from doing general, higher-order AD correctly. A simple

workaround is to compose various AD packages for each level of differentiation, further giving rise to applications where changing between AD mechanisms is increasingly common.

As AD systems have different pros and cons, a software author will want to change AD systems depending on the problem and available hardware resources, see Appendix B. However, this is more challenging than it might seem. Changing AD systems results in forking the code, even though the nominal value of the software using the AD remains the same. To give some examples: Flux.jl[1] changed from using the Tracker.jl[2] to Zygote.jl [Innes et al., 2019]. This resulted in a fork being created, viz. TrackerFlux.jl[3] for those who want to use the old AD system – even though conceptually Flux is a Neural Network library that should be abstracted away from the AD. PyMC4 was created as an attempt to move from Theano [Al-Rfou et al., 2016], as used in PyMC3 [Salvatier et al., 2016], to using TensorFlow [Abadi et al., 2015]. This attempt was eventually abandoned, in favor of keeping Theano but adding a Jax [Bradbury et al., 2018] backend [The PyMC Development Team]. Not only did the code need to be forked, but the overall attempt was not successful. Admittedly, this was a particularly complex case beyond just AD, with TensorFlow and Theano being more general computational frameworks with AD as just one feature. The work we present here aims to ensure that changing the AD system is accessible by providing consistent abstractions that the author of the ∂P algorithm implementation can use.

A similar but more complex problem was solved by the MathOptInterface.jl [Legat et al., 2020]. MathOptInterface.jl provides common abstractions across constrained mathematical optimizers such as IPOPT [Wachter, 2002], Cbc [Forrest et al., 2018], and Gurobi [Gurobi Optimization, LLC, 2021]. It in turn is used by mathematical optimization frameworks including JuMP [Dunning et al., 2017] and Convex.jl [Udell et al., 2014]. Each of the different mathematical optimizers has their own very unique internal set of abstractions, but MathOptInterface.jl exposes them all in the same way. An additional complication is that each supports different kinds of problems and so this too must be exposed. Further still, for some classes of problems they can be re-expressed as a different kind through a mathematical transformation, MathOptInterface exposes this through an extensible system of so-called "bridges", that will automatically perform these reformulations. This system is considerably more complicated than our setting as every AD system can perform all the operations, to varying degrees of efficiency. The MathOptInterface system has proven very successful, which supports the idea that this kind of abstraction is valuable and can be practically realized.

In light of the above, the authors believe it is necessary to have a backend-agnostic interface to provide objects like the function value, its gradient, Hessian, etc. as well as combining AD implementations together for higher-order AD. Such an interface can help us avoid a combinatorial explosion of code when supporting every differentiation package in Julia in every piece of software requiring gradients and/or Hessians. This is especially important for higher-order derivatives because one can combine any two differentiation backends to create a new higher-order backend. More generally for a kth order derivative, the amount of code required to support n differentiation packages in m ∂P algorithm implementations is $O(m \times n^k)$.

In this paper, we present AbstractDifferentiation.jl [Anonymous], a package that:

- Defines an abstract, extensive API for differentiation in Julia enabling the development of algorithms requiring first and higher-order derivatives in an AD-implementation-agnostic way using a single, unified interface reducing the code complexity from $O(m \times n^k)$ to $O(m + n)$.

- Automatically defines most of the extensive user-facing API for any new AD package from just one or two primitive API function definitions, thus making it easier for the AD package developer to support every possible use case without a great deal of boilerplate code.

[1] `https://github.com/FluxML/Flux.jl`
[2] `https://github.com/FluxML/Tracker.jl`
[3] `https://github.com/AStupidBear/TrackerFlux.jl`

141 ## 2 Levels of abstraction in Julia's AD ecosystem

142 In Figure 1, an overview of the levels of abstraction in Julia's AD ecosystem with Abstract-
143 Differentiation.jl is presented. At the bottom level, we have libraries of differentiation rules
144 (DiffRules.jl and ChainRules.jl) for specific functions. These rules are either defined by AD
145 developers for basic Julia constructs, or by AD users for specific user-defined functions with
146 known analytic derivatives.

147 Sitting on top of the library of rules are all the AD package implementations. At this level,
148 numerous design decisions and optimizations can be made giving a variety of different AD
149 package implementations with different tradeoffs. Each AD package developer will then
150 define a minimal set of primitives and a backend type extending AbstractDifferentiation.jl.
151 These minimal definitions then enable AbstractDifferentiation.jl to automatically define an
152 extensive set of user-facing API functions for AD users to use, e.g. derivative, Jacobian,
153 Hessian, Jacobian-vector product, Hessian-vector product, etc.

154 At the top level, AD users can then use the relevant part of the AbstractDifferentiation.jl API
155 to implement algorithms requiring ∂P. With this abstraction design, the amount of code
156 needed to support all of n AD packages in m algorithms requiring k^{th} order derivatives is
157 only $O(m+n)$, a significant reduction from the $O(m \times n^k)$ without AbstractDifferentiation.jl.
158 Additionally, the AD users and developers do not need to add unnecessary boilerplate code
159 to extend an AD package's API anymore, since AbstractDifferentiation.jl automatically does
160 this for them.

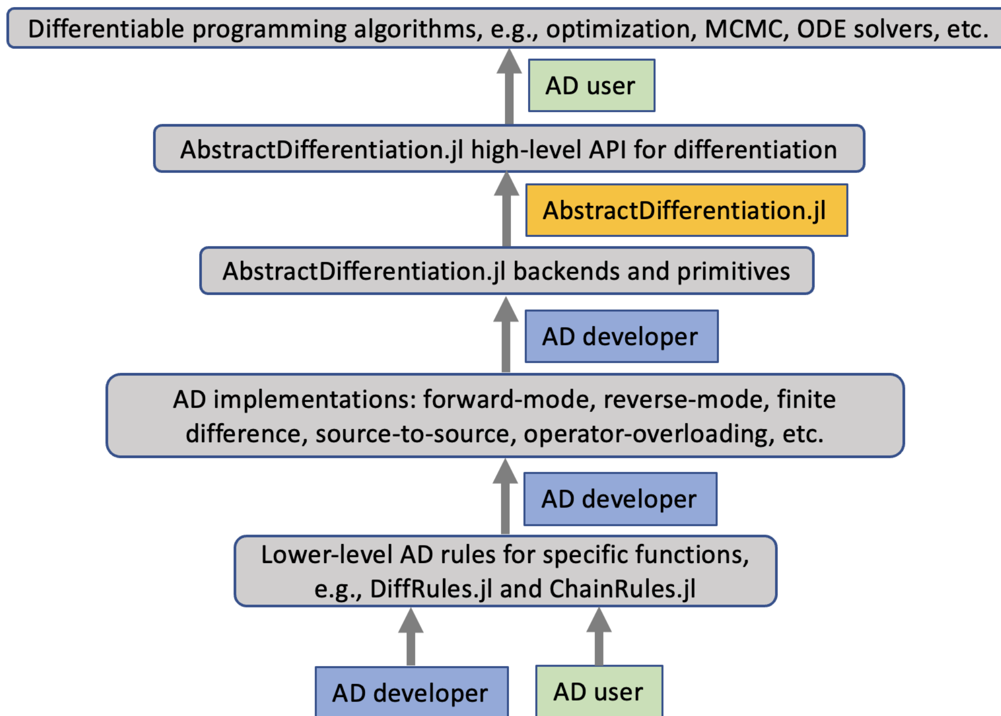

Figure 1: The levels of abstraction in Julia's AD ecosystem.

161 ## 3 API description

162 **Installation and loading** AbstractDifferentiation.jl is a registered Julia package and can
163 be installed by the Julia package manager. The package can be loaded by

```julia
# alternatively: import AbstractDifferentiation as AD
using AbstractDifferentiation
```

Note that AbstractDifferentiation.jl exports "AD" as an alias for the AbstractDifferentiation module. This alias allows us to conveniently access names within AbstractDifferentiation.jl via AD instead of typing the full package name.

3.1 Backends and primitives

Forward-mode, reverse-mode, and finite-difference backends All functionalities in AbstractDifferentiation.jl are implemented based on an `ab::AbstractBackend` type. An AD package developer (or the AD user if necessary) first constructs a backend instance that subtypes `ab::AbstractForwardMode`, `ab::AbstractReverseMode`, or `ab::AbstractFiniteDifference`, which are themselves subtypes of `ab::AbstractBackend`. For example, backends that support ForwardDiff.jl or Zygote.jl are defined as follows:

```julia
## ForwardDiff
struct ForwardDiffBackend <: AD.AbstractForwardMode end
const forwarddiff_backend = ForwardDiffBackend()

## Zygote
struct ZygoteBackend <: AD.AbstractReverseMode end
const zygote_backend = ZygoteBackend()
```

By adding fields to the backend struct, we can control configurations of the differentiation package such as chunk sizes, compilation flags, or method choices. To use a finite differencing method at a central grid of 5 points as implemented in the FiniteDifferences.jl package, we write:

```julia
## FiniteDifferences
struct FDMBackend{A} <: AD.AbstractFiniteDifference
    alg::A
end
# 1 denotes the order of the derivative to estimate.
FDMBackend() = FDMBackend(central_fdm(5, 1))
```

Higher-order backends To compute higher-order derivatives, it may be desirable to combine different backends. We provide `AD.HigherOrderBackend` to implement higher-order backends. Let `ab_f` be a forward-mode automatic differentiation backend and let `ab_r` be a reverse-mode automatic differentiation backend. To construct a higher-order backend for doing forward-over-reverse-mode automatic differentiation, one defines `AD.HigherOrderBackend((ab_f, ab_r))`. Analogously, higher-order backend for doing reverse-over-forward-mode automatic differentiation is constructed via `AD.HigherOrderBackend((ab_r, ab_f))`.

Jacobian, pushforward, and pullback as primitive operation In addition to the definition of a backend, the AD package developer needs to define one of the following primitive operations:

```julia
AD.@primitive function jacobian(ab::backend, f, xs...)
    return ..
end
AD.@primitive function pushforward_function(ab::backend, f, xs...)
    return ..
end
AD.@primitive function pullback_function(ab::backend, f, xs...)
    return ..
end
```

AbstractDifferentiation.jl then generates the other two primitive functions. For instance, a source-to-source reverse-mode AD package developer can specify only `AD.pullback_function` as the native primitive operation.

```
## Zygote is source-to-source reverse-mode
AD.@primitive function pullback_function(ab::ZygoteBackend, f, xs...)
    return function (vs)
        # Supports only single output
        _, back = Zygote.pullback(f, xs...)
        if vs isa AbstractVector
            return back(vs)
        else
            # vs isa Tuple
            @assert length(vs) == 1
            return back(vs[1])
        end
    end
end
```

In the case of operator overloading AD implementations, we require additionally the definition of `AD.primal_value` returning the primal value of the forward pass.

3.2 Automatically provided functions

After these preparatory steps, AbstractDifferentiation.jl automatically defines various functions for AD users making use of the primitives defined. Some of the most important API functions provided are presented in the following. We refer the reader to the package documentation for further details [Anonymous].

Derivative, gradient, jacobian, hessian

```
ds = AD.derivative(ab::AD.AbstractBackend, f, xs::Number...)
gs = AD.gradient(ab::AD.AbstractBackend, f, xs...)
js = AD.jacobian(ab::AD.AbstractBackend, f, xs...)
h = AD.hessian(ab::AD.AbstractBackend, f, x)
```

Value and derivative, gradient, jacobian, hessian

```
v, ds = AD.value_and_derivative(ab::AD.AbstractBackend, f, xs::Number
    ...)
v, gs = AD.value_and_gradient(ab::AD.AbstractBackend, f, xs...)
v, js = AD.value_and_jacobian(ab::AD.AbstractBackend, f, xs...)
v, h = AD.value_and_hessian(ab::AD.AbstractBackend, f, x)
v, g, h = AD.value_gradient_and_hessian(ab::AD.AbstractBackend, f, x)
```

Lazy operators

Finally, we implemented lazy versions of the derivative, gradient, Jacobian, and Hessian,

```
ld = lazy_derivative(ab::AbstractBackend, f, xs::Number...)
lg = lazy_gradient(ab::AbstractBackend, f, xs...)
lj = lazy_jacobian(ab::AbstractBackend, f, xs...)
lh = lazy_hessian(ab::AbstractBackend, f, x)
```

which are of interest in iterative solvers. For example, we compute a vector-Jacobian product by multiplying a single (transposed) vector, or a tuple of an appropriate shape, from the left to the lazy Jacobian operator.

4 ∂P use cases and an example

Many numerical algorithms require the computation of constructs such as the ones described in Section 3.2. Table 1 presents a rough summary linking some of the most widely adopted routines across different domains to the quantities used in the respective iterative update steps. As an example, we present a (simple, non-optimized) backend-agnostic implementation of the Gauss-Newton algorithm to solve non-linear least squares problems in Appendix C.

We also expect AbstractDifferentiation.jl to be specifically handy for (future) AD package like Diffractor.jl or Enzyme.jl where computing constructs like Jacobians or Hessians is technically possible but not yet part of the public API due to abstractions or naming conventions made in the package.

algorithms	required quantities
root finding	
Newton–Raphson	Jacobian
Jacobian-Free Newton Krylov	Jacobian-vector products
optimization	
ADAM	gradient
Newton	gradient, Hessian
Levenberg–Marquardt	Jacobian
Gauss-Newton	Jacobian
differential equations	
stiff ODE solvers	Jacobian
stiff ODE Jacobian-free solvers	Jacobian-vector products
forward sensitivity methods	Jacobian-vector products
adjoint sensitivity methods	vector-Jacobian product

Table 1: Domain-specific algorithms requiring derivatives, gradients, Jacobians, Hessians, vector-Jacobian products, Jacobian-vector products commonly computed by AD packages.

5 Summary & Future work

The ability to straightforwardly combine different packages in one workflow is one of the most versatile and key features of Julia. Switching between different AD packages and combining them for higher-order derivatives is a useful feature to have when selecting the best AD implementation for a specific application. We have presented the AbstractDifferentiation.jl package which makes this switching and combining of AD implementations as painless as possible for end-users while also reducing the amount of necessary boilerplate code per AD package to support all differentiation use cases.

In the future, we aim to support AbstractDifferentiation.jl in all of the AD packages in Julia and remove lots of boilerplate code from popular Julia packages (e.g. in the SciML and TuringLang organizations) that heavily employ AD.

References

M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`. Software available from tensorflow.org.

R. Al-Rfou, G. Alain, A. Almahairi, C. Angermueller, D. Bahdanau, N. Ballas, F. Bastien, J. Bayer, A. Belikov, A. Belopolsky, Y. Bengio, A. Bergeron, J. Bergstra, V. Bisson,

J. Bleecher Snyder, N. Bouchard, N. Boulanger-Lewandowski, X. Bouthillier, A. de Brébisson, O. Breuleux, P.-L. Carrier, K. Cho, J. Chorowski, P. Christiano, T. Cooijmans, M.-A. Côté, M. Côté, A. Courville, Y. N. Dauphin, O. Delalleau, J. Demouth, G. Desjardins, S. Dieleman, L. Dinh, M. Ducoffe, V. Dumoulin, S. Ebrahimi Kahou, D. Erhan, Z. Fan, O. Firat, M. Germain, X. Glorot, I. Goodfellow, M. Graham, C. Gulcehre, P. Hamel, I. Harlouchet, J.-P. Heng, B. Hidasi, S. Honari, A. Jain, S. Jean, K. Jia, M. Korobov, V. Kulkarni, A. Lamb, P. Lamblin, E. Larsen, C. Laurent, S. Lee, S. Lefrancois, S. Lemieux, N. Léonard, Z. Lin, J. A. Livezey, C. Lorenz, J. Lowin, Q. Ma, P.-A. Manzagol, O. Mastropietro, R. T. McGibbon, R. Memisevic, B. van Merriënboer, V. Michalski, M. Mirza, A. Orlandi, C. Pal, R. Pascanu, M. Pezeshki, C. Raffel, D. Renshaw, M. Rocklin, A. Romero, M. Roth, P. Sadowski, J. Salvatier, F. Savard, J. Schlüter, J. Schulman, G. Schwartz, I. V. Serban, D. Serdyuk, S. Shabanian, E. Simon, S. Spieckermann, S. R. Subramanyam, J. Sygnowski, J. Tanguay, G. van Tulder, J. Turian, S. Urban, P. Vincent, F. Visin, H. de Vries, D. Warde-Farley, D. J. Webb, M. Willson, K. Xu, L. Xue, L. Yao, S. Zhang, and Y. Zhang. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016. URL `http://arxiv.org/abs/1605.02688`.

M. AlQuraishi. End-to-end differentiable learning of protein structure. *bioRxiv*, 2018. doi: 10.1101/265231. URL `https://www.biorxiv.org/content/early/2018/02/14/265231`.

Anonymous. AbstractDifferentiation.jl. URL `https://anonymous.4open.science/r/AbstractDifferentiation/README.md`.

T. Besard, C. Foket, and B. De Sutter. Effective extensible programming: Unleashing Julia on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 2018. ISSN 1045-9219. doi: 10.1109/TPDS.2018.2872064.

J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman. Julia: A fast dynamic language for technical computing. *arXiv preprint arXiv:1209.5145*, 2012.

J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL `http://github.com/google/jax`.

R. T. Chen, Y. Rubanova, J. Bettencourt, and D. Duvenaud. Neural ordinary differential equations. *arXiv preprint arXiv:1806.07366*, 2018.

R. Dandekar, C. Rackauckas, and G. Barbastathis. A machine learning-aided global diagnostic and comparative tool to assess effect of quarantine control in covid-19 spread. *Patterns*, 1 (9):100145, 2020.

F. de Avila Belbute-Peres, K. Smith, K. Allen, J. Tenenbaum, and J. Z. Kolter. End-to-end differentiable physics for learning and control. *Advances in neural information processing systems*, 31:7178–7189, 2018.

I. Dunning, J. Huchette, and M. Lubin. Jump: A modeling language for mathematical optimization. *SIAM Review*, 59(2):295–320, 2017.

K. Fischer. Non-local compiler transformations in the presence of dynamic dispatch. URL `https://www.youtube.com/watch?v=mQnSRfseu0c`.

J. Forrest, T. Ralphs, S. Vigerske, LouHafer, B. Kristjansson, jpfasano, EdwinStraver, M. Lubin, H. G. Santos, rlougee, and M. Saltzman. coin-or/cbc: Version 2.9.9, July 2018. URL `https://doi.org/10.5281/zenodo.1317566`.

Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2021. URL `https://www.gurobi.com`.

J. Ingraham, A. Riesselman, C. Sander, and D. Marks. Learning protein structure with a differentiable simulator. In *International Conference on Learning Representations*, 2018.

369 M. Innes, A. Edelman, K. Fischer, C. Rackauckus, E. Saba, V. B. Shah, and W. Tebbutt.
370 Zygote: A differentiable programming system to bridge machine learning and scientific
371 computing. arXiv preprint arXiv:1907.07587, 2019. URL https://arxiv.org/abs/
372 1907.07587.

373 B. Legat, O. Dowson, J. D. Garcia, and M. Lubin. Mathoptinterface: a data structure
374 for mathematical optimization problems, 2020. URL https://arxiv.org/abs/2002.
375 03447.

376 O. Manzyuk, B. A. Pearlmutter, A. A. Radul, D. R. Rush, and J. M. Siskind. Perturbation
377 confusion in forward automatic differentiation of higher-order functions. *Journal of*
378 *Functional Programming*, 29, 2019.

379 W. Moses and V. Churavy. Instead of rewriting foreign code for machine learning, automati-
380 cally synthesize fast gradients. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan,
381 and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages
382 12472–12485. Curran Associates, Inc., 2020. URL https://proceedings.neurips.
383 cc/paper/2020/file/9332c513ef44b682e9347822c2e457ac-Paper.pdf.

384 T. E. Oliphant. *A guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.

385 A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison,
386 L. Antiga, and A. Lerer. Automatic differentiation in pytorch. 2017.

387 C. Rackauckas. Glue AD for full language differentiable pro-
388 gramming. URL http://www.stochasticlifestyle.com/
389 glue-ad-for-full-language-differentiable-programming/.

390 C. Rackauckas, Y. Ma, V. Dixit, X. Guo, M. Innes, J. Revels, J. Nyberg, and V. Ivaturi. A
391 comparison of automatic differentiation and continuous sensitivity analysis for derivatives
392 of differential equation solutions. *arXiv preprint arXiv:1812.01892*, 2018.

393 C. Rackauckas, A. Edelman, K. Fischer, M. Innes, E. Saba, V. B. Shah, and W. Tebbutt.
394 Generalized physics-informed learning through language-wide differentiable programming.
395 In *AAAI Spring Symposium: MLPS*, 2020a.

396 C. Rackauckas, Y. Ma, J. Martensen, C. Warner, K. Zubov, R. Supekar, D. Skinner, and
397 A. Ramadhan. Universal differential equations for scientific machine learning. *arXiv*
398 *preprint arXiv:2001.04385*, 2020b.

399 M. Raissi, P. Perdikaris, and G. E. Karniadakis. Physics-informed neural networks: A deep
400 learning framework for solving forward and inverse problems involving nonlinear partial
401 differential equations. *Journal of Computational Physics*, 378:686–707, 2019.

402 J. Revels. ReverseDiff.jl, 2018. URL http://github.com/JuliaDiff/ReverseDiff.
403 jl.

404 J. Revels, M. Lubin, and T. Papamarkou. Forward-mode automatic differentiation in julia.
405 *arXiv preprint arXiv:1607.07892*, 2016.

406 J. Salvatier, T. V. Wiecki, and C. Fonnesbeck. Probabilistic programming in python using
407 PyMC3. *PeerJ Computer Science*, 2:e55, apr 2016. doi: 10.7717/peerj-cs.55. URL
408 https://doi.org/10.7717/peerj-cs.55.

409 F. Schäfer. Abstractdifferentiation.jl for AD-backend agnostic code. URL https://
410 frankschae.github.io/post/abstract_differentiation/.

411 F. Schäfer, M. Kloc, C. Bruder, and N. Lörch. A differentiable programming method for
412 quantum control. *Machine Learning: Science and Technology*, 1(3):035009, 2020.

413 F. Schäfer, P. Sekatski, M. Koppenhöfer, C. Bruder, and M. Kloc. Control of stochastic
414 quantum dynamics by differentiable programming. *Machine Learning: Science and*
415 *Technology*, 2(3):035004, 2021.

416 C. Schenck and D. Fox. Spnets: Differentiable fluid dynamics for deep neural networks. In
417 *Conference on Robot Learning*, pages 317–335. PMLR, 2018.

418 J. M. Siskind and B. A. Pearlmutter. Perturbation confusion and referential transparency:
419 Correct functional implementation of forward-mode AD. 2005.

420 F. Srajer, Z. Kukelova, and A. Fitzgibbon. A benchmark of selected algorithmic differentiation
421 tools on some problems in computer vision and machine learning. *Optimization Methods*
422 *and Software*, 33(4-6):889–906, 2018.

423 The PyMC Development Team. The future of pymc3, or: Theano is
424 dead, long live theano. URL `https://pymc-devs.medium.com/`
425 `the-future-of-pymc3-or-theano-is-dead-long-live-theano-d8005f8a0e9b`.

426 M. Udell, K. Mohan, D. Zeng, J. Hong, S. Diamond, and S. Boyd. Convex optimization in
427 Julia. *SC14 Workshop on High Performance Technical Computing in Dynamic Languages*,
428 2014.

429 A. Wachter. *An interior point algorithm for large-scale nonlinear optimization with applica-*
430 *tions in process engineering*. PhD thesis, Carnegie Mellon University, 2002.

431 L. White. Juliadiff website. URL `https://juliadiff.org/`.

432 L. White, M. Zgubic, M. Abbott, J. Revels, A. Arslan, S. Axen, S. Schaub, N. Robinson,
433 Y. Ma, G. Dhingra, W. Tebbutt, N. Heim, A. D. W. Rosemberg, N. Schmitz, C. Rack-
434 auckas, D. Widmann, R. Heintzmann, F. Schäfer, K. Fischer, A. Robson, M. Brzezin-
435 ski, A. Zhabinski, M. Besançon, P. Vertechi, S. Gowda, A. Fitzgibbon, C. Lucibello,
436 C. Vogt, D. Gandhi, and F. Chorney. Juliadiff/chainrules.jl: v1.11.5, Sept. 2021. URL
437 `https://doi.org/10.5281/zenodo.5467874`.

438 439

A Summary of the current state of AD packages in Julia as of September 2021

Table 2: This table summarizes the current state of popular Julia AD packages in September 2021. “Scalar” refers to scalar operations support including defining custom rules for scalar-valued functions of scalars. ”Vector/tensor” refers to native vector/tensor support as a construct including the ability to define custom differentiation rules for vector/tensor-valued functions and/or functions of vectors/tensors. Similarly, “First class struct support” refers to the native support of Julia structs as a construct including the ability to define custom differentiation rules for struct-valued functions or functions of structs. “GPU” refers to the ability to differentiate functions of or returning GPU arrays. ”GC” refers to supporting functions that call the Julia garbage collector. “Mutation” refers to the ability to support mutating arrays and structs. “Runtime branches” refers to the ability to support “piece-wise” functions with control flow such that the path that the function takes and ultimately the structure of the mathematical function differentiated depends on the values of the inputs to the function. “Maturity” refers to a subjective measure of how mature each package is in the eyes of the community as well as the feature maturity of the package.

Package	Scalar	Vector / tensor	First class struct support	GPU	GC	Mutation	Runtime branches	Maturity
ForwardDiff	✓	✗	✗	✓	✓	✓	✓	very high
ReverseDiff	slow	✓	✗	✗	✓	limited	✓	high
ReverseDiff with compiled tape	✓	✓	✗	✗	✓	limited	✗	high
Tracker	slow	✓	✗	✓	✓	limited	✓	high
Zygote	slow	✓	✓	✓	✓	✗	✓	high
Enzyme	✓	limited	wip	wip	wip	✓	✓	low
Diffractor	✓	✓	✓	✓	✓	✗	✓	low

440 Table 2 summarizes the current state of the most popular AD packages in the Julia ecosystem
441 as of the time of the writing of this paper.

442 # B AD performance can be problem-specific

443 It is well know that for a function $f : \mathbb{R}^n \to \mathbb{R}^m$ with n independent input variables and m
444 dependent output variables, forward-mode AD is preferred to build the Jacobian when $m \gg n$
445 while reverse-mode AD is preferred when $n \gg m$, i.e. as one increases the number of inputs
446 within the same problem, reverse-mode AD mode will eventually overtake forward-mode
447 AD in performance. This has been investigated in depth for differential equations when
448 applied to models relevant to biopharmacology, alongside various adjoint options [Rackauckas
449 et al., 2018]. This work shows that on sufficiently small ODEs (<100 ODEs + parameters),
450 forward-mode AD via ForwardDiff.jl is the most efficient method comparing against analytical
451 techniques and adjoint techniques using Tracker.jl, Enzyme.jl, and ReverseDiff.jl. When the
452 size of the ODEs+parameters is increased in a stiff partial differential equation, it was shown
453 that Enzyme.jl vector-Jacobian products mixed with a specific adjoint method was the most
454 efficient, outperforming the ForwardDiff.jl techniques long with ReverseDiff.jl and Tracker.jl.

455 In what follows, we demonstrate on two additional examples that the choice of the specific
456 reverse-mode AD package may also significantly impact the performance [Srajer et al., 2018].
457 These examples show ReverseDiff.jl in compiled mode outperforming Enzyme.jl under certain
458 circumstances. However, as ReverseDiff.jl is not compatible with GPUs and was shown to not
459 be performance competitive on other potential equations in scientific computing applications,
460 which allows Zygote.jl and Tracker.jl to be the most efficient. Together this shows in one
461 application that 5 AD systems could potentially be the optimal choice depending on user
462 inputs into the package code. This establishes that the optimal choice of AD mechanism
463 can be rather complex for users and package developers, and thus decreasing the cost of
464 performing such benchmarks is of value to many scientists.

465 **Example 1: Lotka–Volterra model**

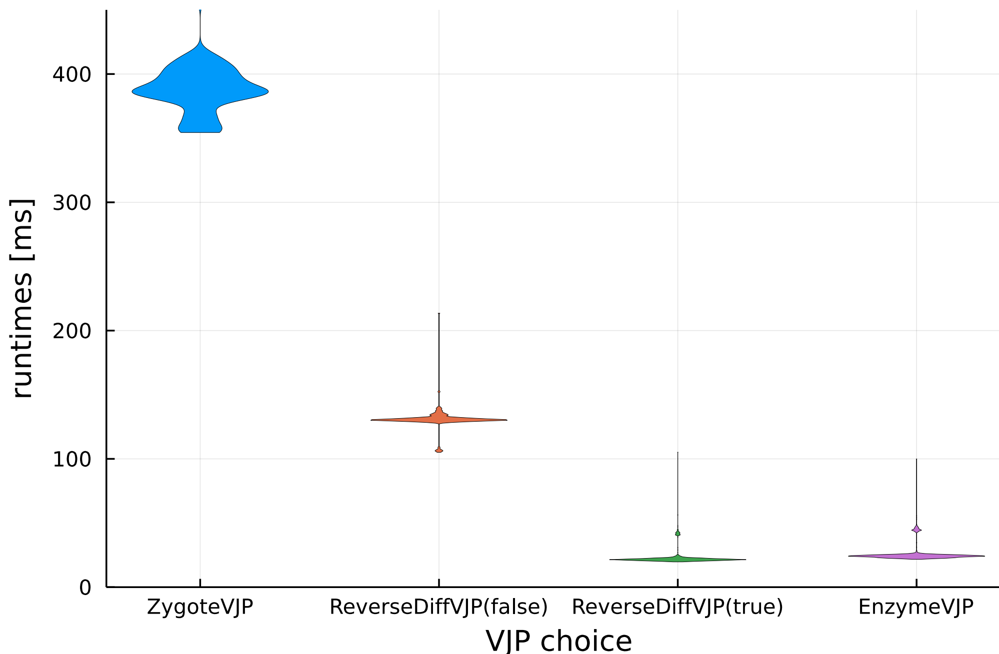

Figure 2: Benchmark 1: Lotka–Volterra model. In all cases, we use a checkpointed interpolating adjoint method [Rackauckas et al., 2020b] to compute the local sensitivities. 'false' and 'true' indicate if the tape in ReverseDiff.jl is precompiled.

466 Suppose that we have an instantaneous objective

$$l(x(t), y(t)) = x(t) + y(t) \tag{1}$$

467 for the Lotka–Volterra model

$$\dot{x} = \alpha x - \beta xy\,, \tag{2}$$
$$\dot{y} = \gamma xy - \delta y\,, \tag{3}$$

468 with initial conditions $x(t=0)=1$ and $y(t=0)=1$. Let ξ denote any of the parameters
469 $\alpha, \beta, \gamma, \delta$. We are interested in the sensitivities $\frac{\partial}{\partial \xi} \sum_i l(x(t_i), y(t_i))$ with respect to an equally
470 spaced time grid between 0 and 10 with a grid spacing of 0.1.

471 Figure 2 shows a violin plot for the runtimes for four choices of the internally used AD
472 system. This demonstrates that the vector-Jacobian products which use static compilation of
473 the ODE function, ReverseDiff.jl with compilation enabled and Enzyme.jl, vastly outperform
474 the other choices for small ODEs with a lot of scalar indexing, which is a common feature
475 in many nonlinear physical and biochemical models. Note that all adjoint techniques were
476 shown to be outperformed by ForwardDiff.jl on this example elsewhere [Rackauckas et al.,
477 2018], but this example still confirms that in many scalar indexing cases the Zygote.jl system
478 can perform rather poorly.

479 **Example 2: Neural ODE**

480 This example is the Spiral Neural ODE chosen from the Neural Ordinary Differential
481 Equations manuscript [Chen et al., 2018]. It is an ODE defined as a neural network applied
482 to the cubed states of the system:

$$\dot{u} = \mathrm{NN}(u^3) \tag{4}$$

483 where $\mathrm{NN}(u)$ is a multilayer perceptron with one hidden layer of size 50 and a tanh activation
484 function, and $u \in \mathbb{R}^2$. Figure 3 shows a violin plot for the runtimes for four choices of
485 the internally used AD system. The results show that for direct differentiation on CPUs,
486 ReverseDiffVJP with a compiled tape is the most efficient method. However, this has many
487 caveats. One caveat is that ReverseDiff.jl's tape-compiled form is only applicable if the code
488 has no branching, and thus would be incompatible with activation functions like relu.

489 Additionally, by testing over various sizes of hidden layers, we established that a RTX
490 2080 Super GPU outperformed a Ryzen 9 5950x CPU when the hidden layer size reached
491 approximately 7,500 (note the crossover point could potentially be a lot smaller in many
492 scenarios if the neural network is deeper since the first and last layer sizes are 2, matching
493 the dimensionality of u). At around this size of neural networks, the Zygote.jl and Tracker.jl
494 strategies on GPUs become more efficient than the one of ReverseDiff.jl which is restricted
495 to CPUs.

496 These two examples, in addition to the prior research, clearly demonstrate that the internal
497 AD system must be carefully chosen based on the problem (and hardware resources) at hand.

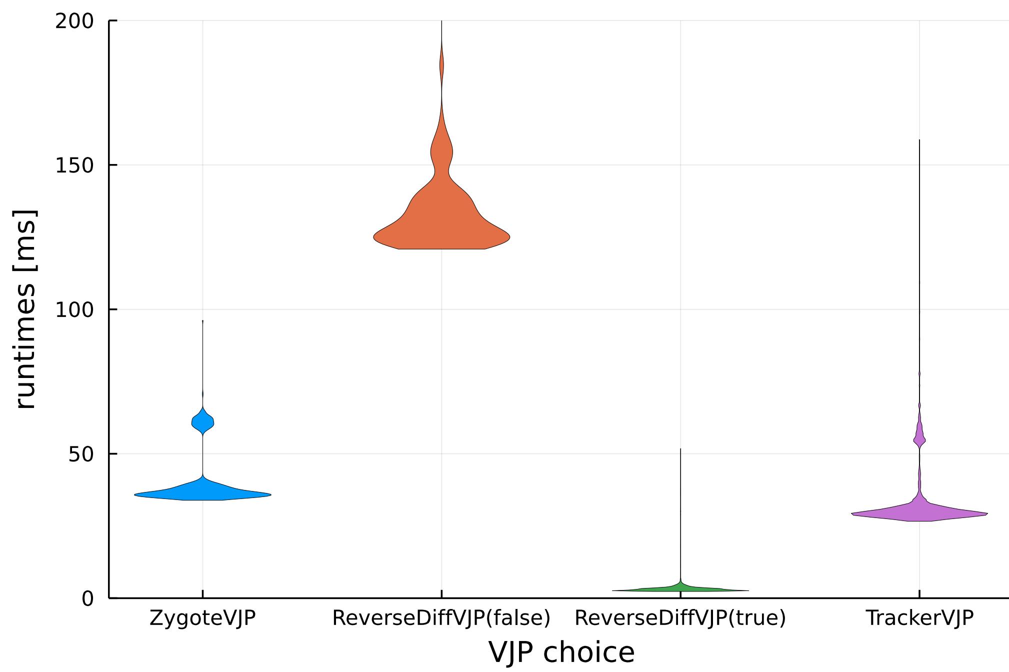

Figure 3: Benchmark 2: Spiral Neural ODE model. In all cases, we use a checkpointed interpolating adjoint method [Rackauckas et al., 2020b] to compute the local sensitivities. 'false' and 'true' indicate if the tape in ReverseDiff.jl is precompiled.

C Implementation of the Gauss-Newton algorithm

In this appendix, we use AbstractDifferentiation.jl for the implementation of the Gauss–Newton algorithm for solving nonlinear least squares problems [Schäfer]. The Gauss–Newton algorithm iteratively finds the value of the N variables $\mathbf{x} = (x_1, \dots, x_N)$ minimizing the sum of squares of M residuals $(f_1, \dots, f_M)$

$$S(\mathbf{x}) = \frac{1}{2} \sum_{i=1}^{M} f_i(\mathbf{x}; \mathbf{p})^2. \tag{5}$$

Starting from an initial guess $\mathbf{x_0}$ for the minimum, the method runs through the iterations

$$\mathbf{x}^{k+1} = \mathbf{x}^k - \alpha_k \left(J^T J\right)^{-1} J^T f(\mathbf{x}^k; p), \tag{6}$$

where the residuals $f(\mathbf{x}^k; p)$ depend on the current step $\mathbf{x}^k$ and parameters p. J is the Jacobian matrix at $\mathbf{x}^k$, and α_k is the step length determined via a line search subroutine.

```
## Gauss-Newton scheme
function GaussNewton!(xs, x, p backend; maxiter=100)
    for i=1:maxiter
        x = step(x, p, backend)
        push!(xs, x)
    end
    return xs, x
end
function step(x, p, backend, α=1//1)
  x2 = deepcopy(x)
  while !done(x,x2,p) # line-search condition
    # first return value of AD.jacobian is ∂f∂x
    # second return value of AD.jacobian is ∂f∂p
    J = AD.jacobian(backend, f, x, p)[1]
    d = -inv(J'*J)*J'*f(x,p)
    copyto!(x2,x + α*d)
    α = α//2
  end
  return x2
end
```

Switching between different AD systems is then easily accomplished by passing different backends (`zygote_backend`, `forwarddiff_backend`, etc.) as input to the `GaussNewton!` function.