Leveraging Sparsity to Accelerate Automatic Differentiation

Adrian Hill – BIFOLD, TU Berlin

github.com/adrhill
adrianhill.de

JuliaCon Local Paris
October 3rd, 2025

SLIDES

Questions we will answer

- What is Automatic Differentiation (AD)?
- What is Automatic Sparse Differentiation (ASD)?
- Can ASD help you solve your problem?
- How can you use ASD?

SLIDES

Automatic Differentiation

Flavors of Differentiation

1. **Manual:** work out f' by hand
2. **Numeric:** $f'(x) \approx \frac{f(x+\varepsilon)-f(x)}{\varepsilon}$
3. **Symbolic:** code a formula for f, get a formula for f'
4. **Automatic:** code a program for f, get a value for $f'(x)$

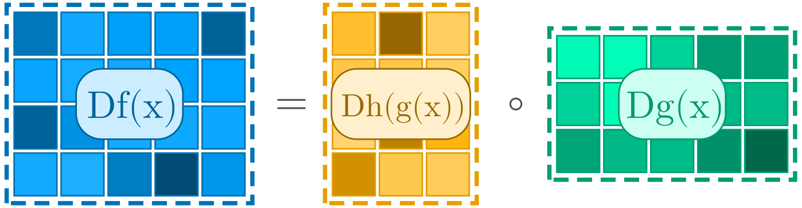
Automatic Differentiation [GW08]

- Programs are **composition** chains (or DAGs) of many functions
- For a composed function $f = h \circ g$, the Jacobian $\boldsymbol{J}_f|_{\boldsymbol{x}}$ at a point of linearization $\boldsymbol{x}$ is given by the **chain rule** as

$$\boldsymbol{J}_f|_{\boldsymbol{x}} = \boldsymbol{J}_h|_{g(\boldsymbol{x})} \cdot \boldsymbol{J}_g|_{\boldsymbol{x}}$$

- Instead of materialized Jacobian matrices, AD uses **matrix-free** Jacobian operators

$$\mathcal{D}f(\boldsymbol{x}) = \mathcal{D}h(g(\boldsymbol{x})) \circ \mathcal{D}g(\boldsymbol{x})$$



We represent matrix-free operators using dashed outlines, matrices and vectors with solid outlines

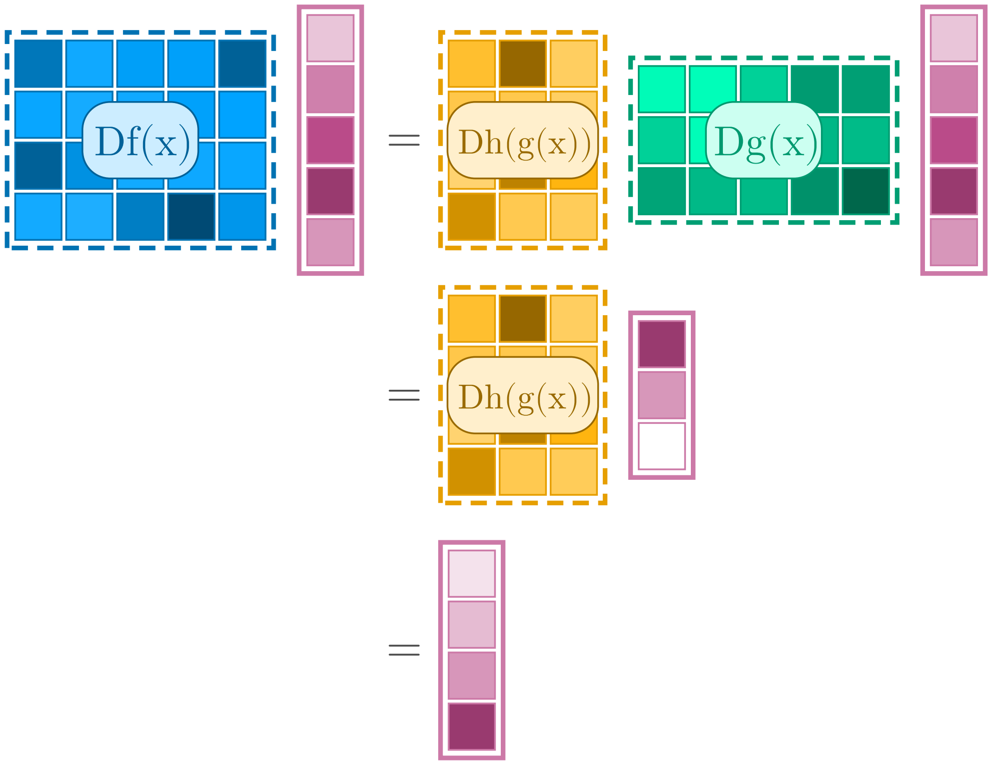
- Primary modes of evaluation of these operators: **forward or reverse**

Forward Mode

- Computes matrix-free **Jacobian-vector products** (JVPs)
- Materializes Jacobians column-by-column

$$\mathcal{D}f(\boldsymbol{x})(\boldsymbol{e}_j) = \boldsymbol{J}_f|_{\boldsymbol{x}} \cdot \boldsymbol{e}_j = \left(\boldsymbol{J}_f|_{\boldsymbol{x}}\right)_{:,j},$$

requiring as many JVPs as the **input dimensionality** of f



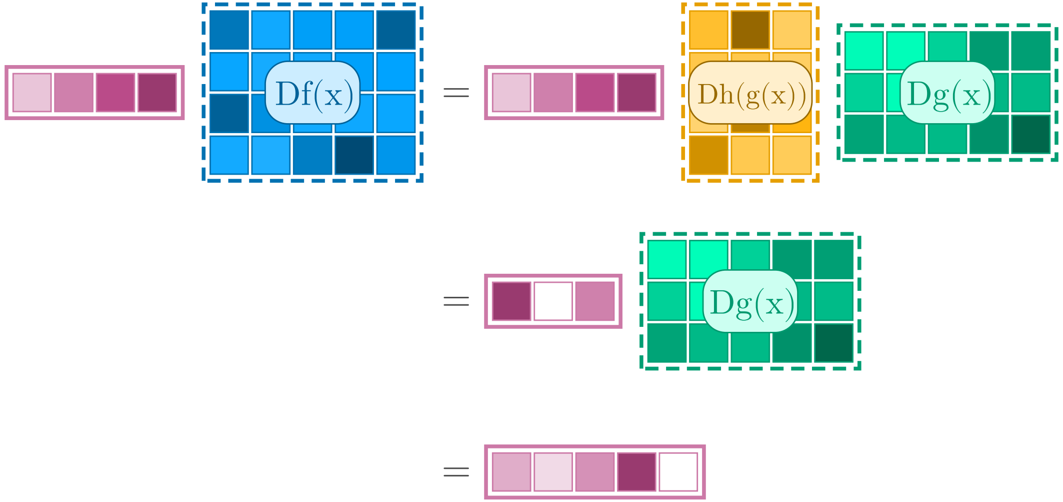
Reverse Mode

- Computes matrix-free **vector-Jacobian products** (VJPs)
- Materializes Jacobians row-by-row

$$\boldsymbol{e}_i^T \cdot \boldsymbol{J}_f|_{\boldsymbol{x}} = \left(\boldsymbol{J}_f|_{\boldsymbol{x}}\right)_{i,:} ,$$

  requiring as many VJPs as the **output dimensionality** of f

- **Special case:** gradient of $f : \mathbb{R}^n \to \mathbb{R}$ requires only a single VJP



Automatic Sparse Differentiation

Automatic Sparse Differentiation

- **Requirement:** sparsity in Jacobian or Hessian
- **Goal:** materialize Jacobian or Hessian matrices from matrix-free operators (JVPs / VJPs/ HVPs)
  - can be more performant
  - more memory efficient
- **Applications**: 2nd-order optimization, root-finding, implicit differentiation
  - direct solvers can be used instead of matrix-free solvers
- **Not useful for gradients**

0.0	1.85	0.0	2.21	0.0
0.0	0.0	0.0	0.97	-2.19
0.0	-0.58	1.47	0.0	0.0
-1.91	0.0	-0.46	0.0	0.0

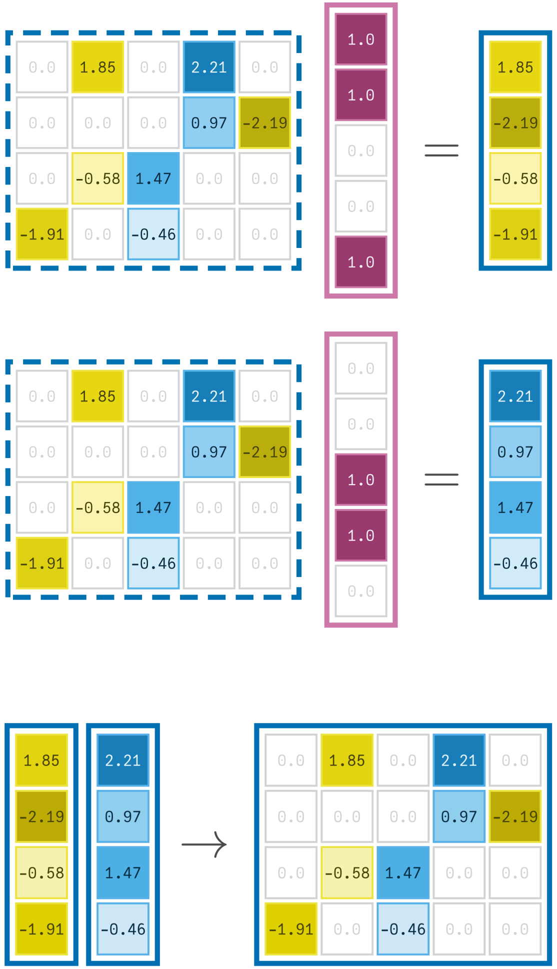
Key Ideas [CPR74]

Assuming we know the structure of the resulting Jacobian matrix:

- Jacobian operators (JVPs, VJPs) are linear maps and therefore additive
- We can simultaneously materialize multiple structurally orthogonal columns (or rows) with a single JVP (or VJP)

$$\mathcal{D}f(\boldsymbol{x})(\boldsymbol{e}_i + \ldots + \boldsymbol{e}_j) = \underbrace{\mathcal{D}f(\boldsymbol{x})(\boldsymbol{e}_i)}_{\left(\boldsymbol{J}_f|_{\boldsymbol{x}}\right)_{:,i}} + \ldots + \underbrace{\mathcal{D}f(\boldsymbol{x})(\boldsymbol{e}_j)}_{\left(\boldsymbol{J}_f|_{\boldsymbol{x}}\right)_{:,j}}$$

- We can then decompress resulting vectors into the Jacobian matrix



Overview of ASD

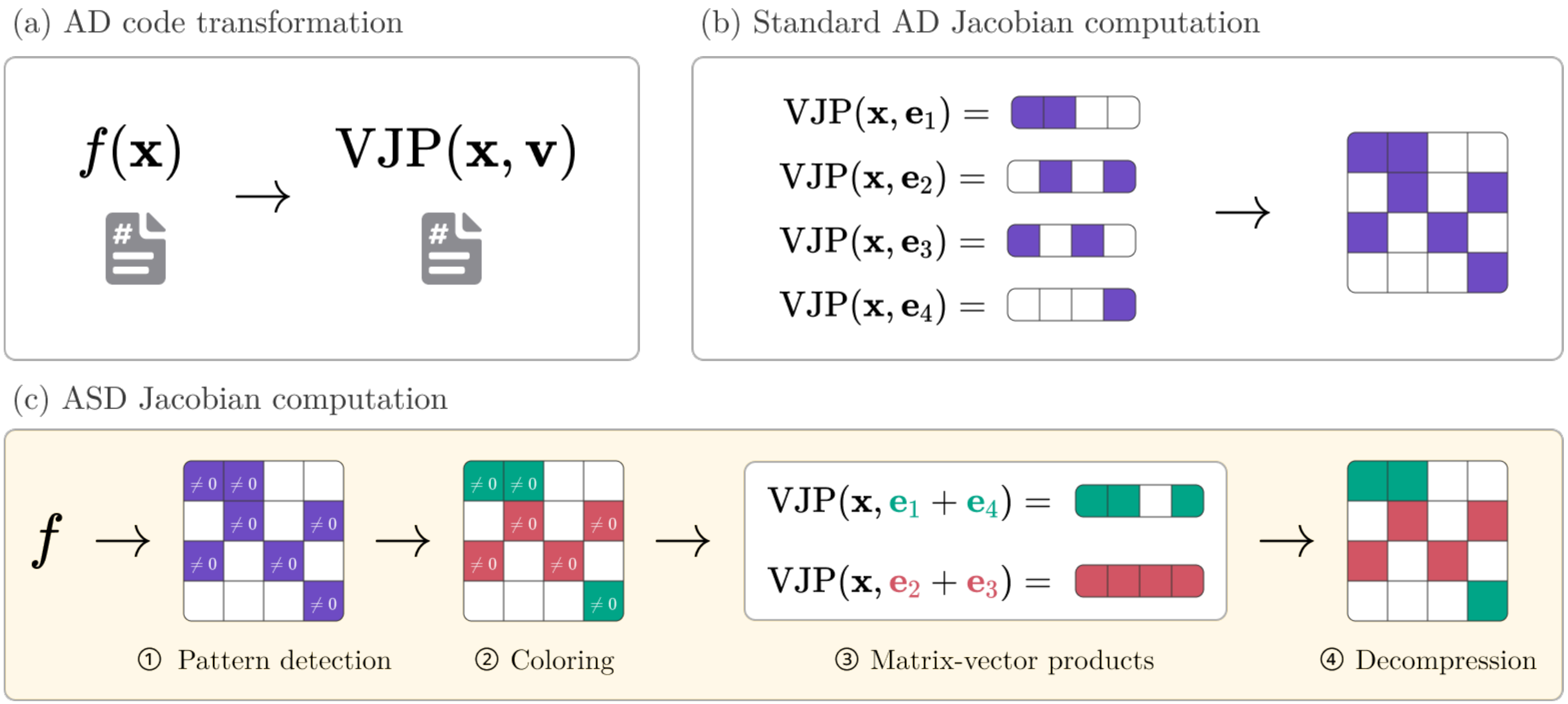


Figure from [HD25]

Step 1:
Sparsity Pattern Detection

Sparsity Pattern Detection: Motivation

Problem: matrix-free Jacobian operators (JVPs, VJPs) are **black-boxes**

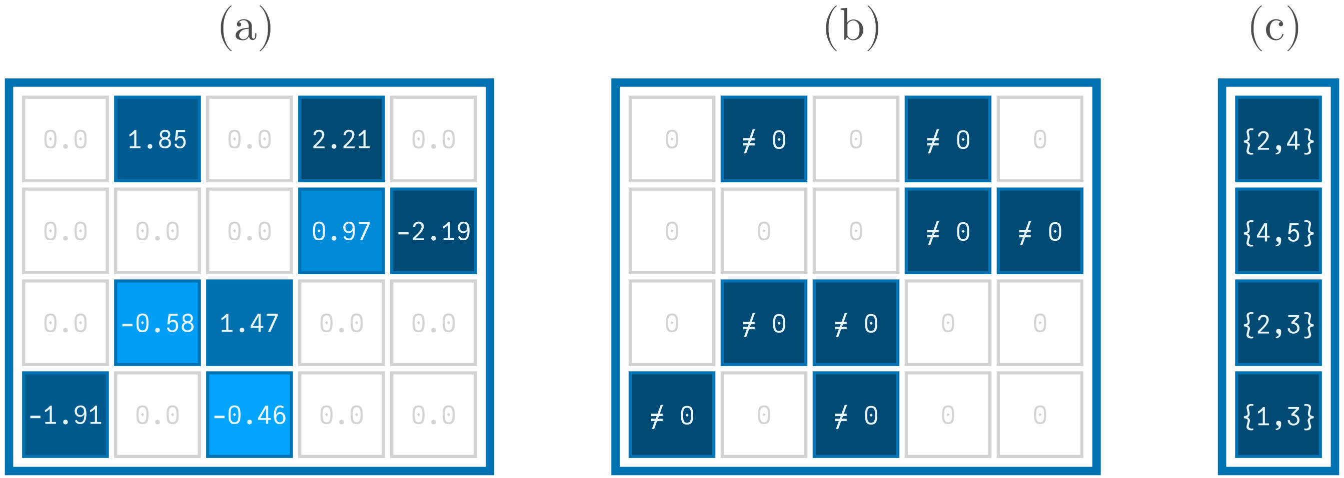
- without materializing Jacobian matrices, their structure is unknown
- if we fully materialize Jacobian matrices via "dense AD", ASD isn't needed

Solution: Implement a fast "boolean"-AD system

- compute sparsity pattern $\left\{(i, j) \mid \frac{\partial f_i}{\partial x_j} \neq 0\right\}$ ("boolean Jacobian")
- has to be faster than the computation of JVPs/VJPs ASD allows us to skip

Propagation of Index Sets [Wal08]

Idea: Represent rows of a sparse matrix by index sets of non-zero values



Sketch of procedure:

1. Seed inputs x_j with index sets $\{j\}$
2. Propagate index sets through compute graph according to chain rule
3. Index set of i-th output corresponds to i-th row of Jacobian $\left\{ j \mid \frac{\partial f_i}{\partial x_j} \neq 0 \right\}$

SparseConnectivityTracer.jl [HD25]

- **Jacobian and Hessian sparsity patterns**
- **Flexible pattern representations**
- **Global tracers**
  - no primal value
  - almost no control flow
  - fast and reusable patterns
- **Local tracers**
  - include primal value
  - support full control flow
  - sparser patterns, not reusable

Published in Transactions on Machine Learning Research (05/2025)

Sparser, Better, Faster, Stronger: Sparsity Detection for Efficient Automatic Differentiation

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

Guillaume Dalle* *guillaume.dalle@enpc.fr*
LVMT, ENPC, Institut Polytechnique de Paris, Univ Gustave Eiffel, Marne-la-Vallée, France

Reviewed on OpenReview: https://openreview.net/forum?id=GtXSN52nIW

Abstract

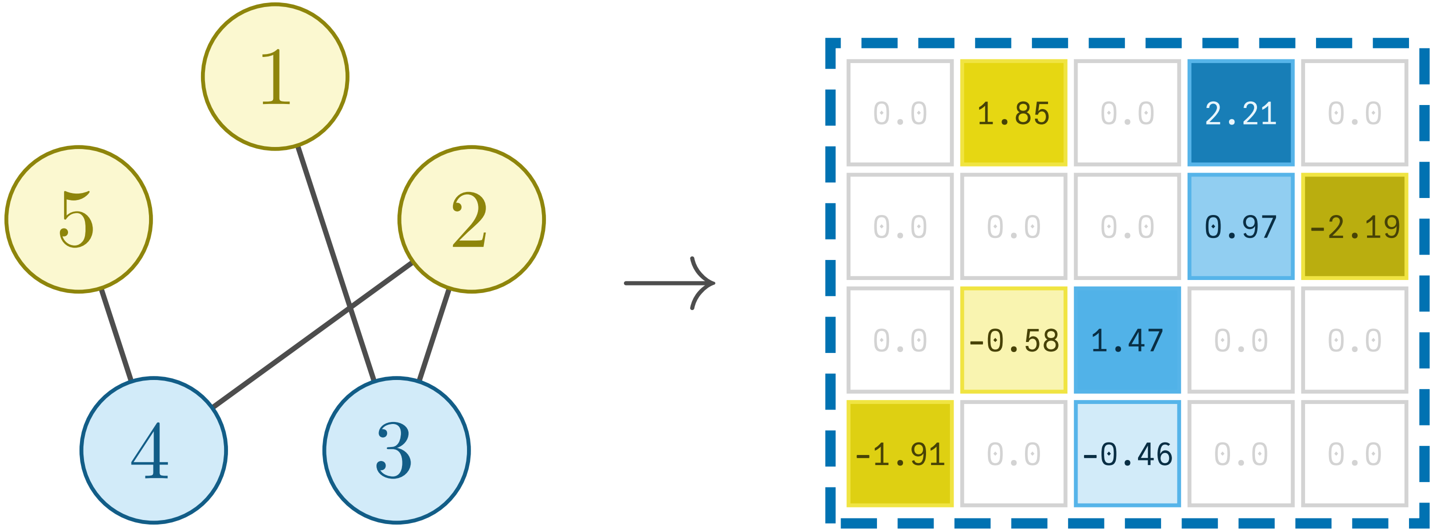
From implicit differentiation to probabilistic modeling, Jacobian and Hessian matrices have many potential use cases in Machine Learning (ML), but they are viewed as computationally prohibitive. Fortunately, these matrices often exhibit sparsity, which can be leveraged to speed up the process of Automatic Differentiation (AD). This paper presents advances in *sparsity detection*, previously the performance bottleneck of Automatic Sparse Differentiation (ASD). Our implementation of sparsity detection is based on operator overloading, able to detect both local and global sparsity patterns, and supports flexible index set representations. It is fully automatic and requires no modification of user code, making it compatible with existing ML codebases. Most importantly, it is highly performant, unlocking Jacobians and Hessians at scales where they were considered too expensive to compute. On real-world problems from scientific ML, graph neural networks and optimization, we show significant speed-ups of up to three orders of magnitude. Notably, using our sparsity detection system, ASD outperforms standard AD for one-off computations, without amortization of either sparsity detection or matrix coloring.

TLDR: Fast boolean ForwardDiff.jl

Step 2:
Matrix Coloring

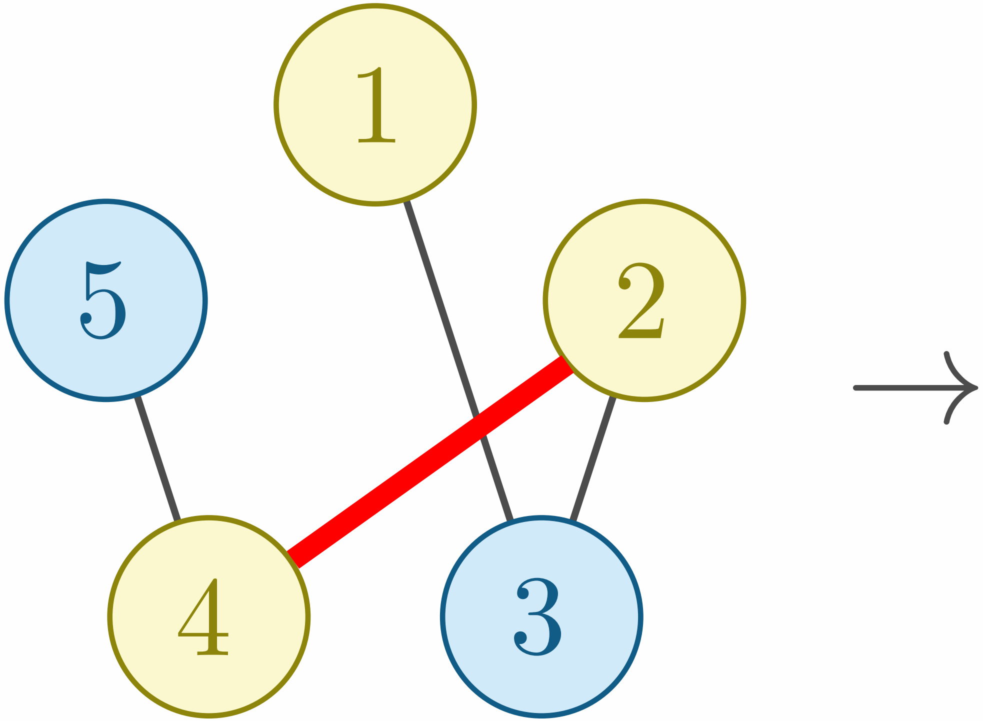
Graph Coloring [GMP05]

Apply graph coloring algorithms to the sparsity pattern
to group together orthogonal (non-overlapping) columns/rows



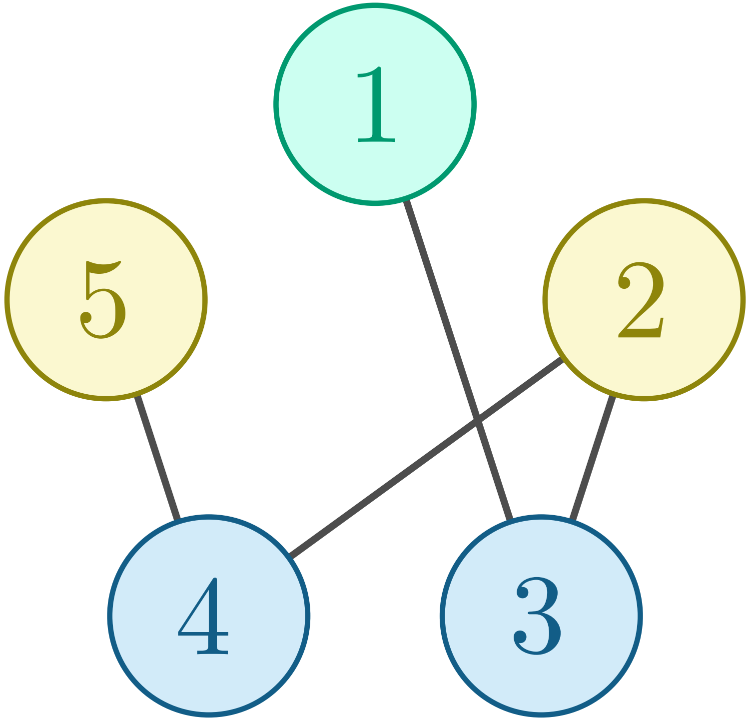
- **Correctness**: guarantee structural orthogonality
- **Efficiency**: try to form the smallest number of groups (NP-hard!)

Coloring example: Infeasible



0.0	1.85	0.0	2.21	0.0
0.0	0.0	0.0	0.97	-2.19
0.0	-0.58	1.47	0.0	0.0
-1.91	0.0	-0.46	0.0	0.0

Coloring example: Suboptimal



→

0.0	1.85	0.0	2.21	0.0
0.0	0.0	0.0	0.97	-2.19
0.0	-0.58	1.47	0.0	0.0
-1.91	0.0	-0.46	0.0	0.0

Finding optimal colorings is NP-hard

SparseMatrixColorings.jl [MDG25]

- SotA methods from `ColPack` in Julia
  - 6x shorter than C++ code
  - similar performance
- Data structure and caching improvements
- New bicoloring algorithms
- Python bindings for non-believers

REVISITING SPARSE MATRIX COLORING AND BICOLORING

ALEXIS MONTOISON*, GUILLAUME DALLE†, AND ASSEFAW GEBREMEDHIN‡

Abstract. Sparse matrix coloring and bicoloring are fundamental building blocks of sparse automatic differentiation. Bicoloring is particularly advantageous for rectangular Jacobian matrices with at least one dense row and column. Indeed, in such cases, unidirectional row or column coloring demands a number of colors equal to the number of rows or columns. We introduce a new strategy for bicoloring that encompasses both direct and substitution-based decompression approaches. Our method reformulates the two variants of bicoloring as star and acyclic colorings of an augmented symmetric matrix. We extend the concept of neutral colors, previously exclusive to bicoloring, to symmetric colorings, and we propose a post-processing routine that neutralizes colors to further reduce the overall color count. We also present the Julia package `SparseMatrixColorings.jl`, which includes these new bicoloring algorithms alongside all standard coloring methods for sparse derivative matrix computation. Compared to `ColPack`, the Julia package also offers enhanced implementations for star and acyclic coloring, vertex ordering, as well as decompression.

Key words. graph coloring, bicoloring, post-processing, sparsity patterns, Jacobian, Hessian, automatic differentiation, Julia

AMS subject classifications. 05C15, 65F50, 65D25, 68R10, 90C06

Further reading

- Plenty of prior work [CPR74, GR91, GW08, PT79], both on sparsity pattern detection [DMM90, GUG95, Wal08, Wal12] and matrix coloring [GMP05]
- Basis for previous slides:

ICLR Blogposts 2025 about call for blogposts submitting reviewing blog past iterations

An Illustrated Guide to Automatic Sparse Differentiation

In numerous applications of machine learning, Hessians and Jacobians exhibit sparsity, a property that can be leveraged to vastly accelerate their computation. While the usage of automatic differentiation in machine learning is ubiquitous, automatic sparse differentiation (ASD) remains largely unknown. This post introduces ASD, explaining its key components and their roles in the computation of both sparse Jacobians and Hessians. We conclude with a practical demonstration showcasing the performance benefits of ASD.

AUTHORS | AFFILIATIONS
Adrian Hill — BIFOLD – Berlin Institute for the Foundations of Learning and Data, Machine Learning Group, Technical University of Berlin, Berlin, Germany
Guillaume Dalle — LVMT, ENPC, Institut Polytechnique de Paris, Univ Gustave Eiffel, Marne-la-Vallée, France
Alexis Montoison — Argonne National Laboratory, Lemont, USA

PUBLISHED
April 28, 2025

Using ASD in Julia

DifferentiationInterface.jl

DifferentiationInterface.jl [DH25]

Common interface for most Julia AD backends:

ForwardDiff.jl

```julia
using DifferentiationInterface
import ForwardDiff

f(x) = diff(x .^ 2) + diff(reverse(x .^ 2))
x = [1.0, 2.0, 3.0, 4.0, 5.0]

jacobian(f, AutoForwardDiff(), x)
```

```
4×5 Matrix{Float64}:
 -2.0   4.0    0.0   8.0  -10.0
  0.0  -4.0   12.0  -8.0    0.0
  0.0   4.0  -12.0   8.0    0.0
  2.0  -4.0    0.0  -8.0   10.0
```

Enzyme.jl

```julia
using DifferentiationInterface
import Enzyme

f(x) = diff(x .^ 2) + diff(reverse(x .^ 2))
x = [1.0, 2.0, 3.0, 4.0, 5.0]

jacobian(f, AutoEnzyme(), x)
```

```
4×5 Matrix{Float64}:
 -2.0   4.0    0.0   8.0  -10.0
  0.0  -4.0   12.0  -8.0    0.0
  0.0   4.0  -12.0   8.0    0.0
  2.0  -4.0    0.0  -8.0   10.0
```

Composable first-order ASD

Compose `AutoSparse` backend, e.g. using **Enzyme.jl**, **SparseConnectivityTracer.jl** and **SparseMatrixColorings.jl**:

```
backend = AutoEnzyme()




jacobian(f, backend, x) # AD
```

```
backend = AutoSparse(
  AutoEnzyme(),
  TracerSparsityDetector(),  # from SCT
  GreedyColoringAlgorithm(), # from SMC
)
jacobian(f, backend, x) # ASD
```

```
4×5 Matrix{Float64}:
 -2.0   4.0    0.0   8.0  -10.0
  0.0  -4.0   12.0  -8.0    0.0
  0.0   4.0  -12.0   8.0    0.0
  2.0  -4.0    0.0  -8.0   10.0
```

```
4×5 SparseMatrixCSC{Float64, Int64}:
 -2.0   4.0     ⋅    8.0  -10.0
   ⋅   -4.0   12.0  -8.0     ⋅
   ⋅    4.0  -12.0   8.0     ⋅
  2.0  -4.0     ⋅   -8.0   10.0
```

Using preparation mechanism, sparsity detection & coloring can be amortized

Composable second-order ASD

Compute sparse Hessians by composing `SecondOrder` and `AutoSparse`:

```julia
using Differentiationterface
using SparseConnectivityTracer
using SparseMatrixColorings
import ForwardDiff
import ReverseDiff

dense_backend = SecondOrder(
  AutoForwardDiff(), # outer backend
  AutoReverseDiff()  # inner backend
)

sparse_backend = AutoSparse(
  dense_backend,
  TracerSparsityDetector(),  # from SCT
  GreedyColoringAlgorithm()  # from SMC
)
```

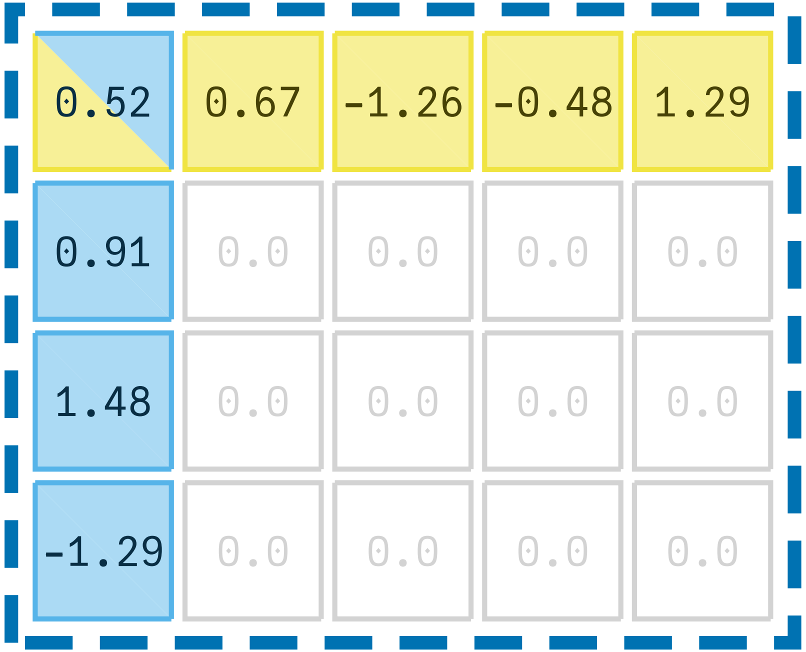
```julia
julia> f(x) = sum(diff(x) .^ 2);
julia> x = [1.0, 2.0, 3.0, 4.0];

julia> hessian(f, dense_backend, x)
4×4 Matrix{Float64}:
  2.0  -2.0   0.0   0.0
 -2.0   4.0  -2.0   0.0
  0.0  -2.0   4.0  -2.0
  0.0   0.0  -2.0   2.0

julia> hessian(f, sparse_backend, x)
4×4 SparseMatrixCSC{Float64, Int64}:
  2.0  -2.0    ⋅     ⋅
 -2.0   4.0  -2.0    ⋅
   ⋅   -2.0   4.0  -2.0
   ⋅     ⋅   -2.0   2.0
```

Composable mixed-mode ASD

ASD can be accelerated further by coloring both rows and columns, combining forward and reverse mode [CV98, HS98, MDG25]



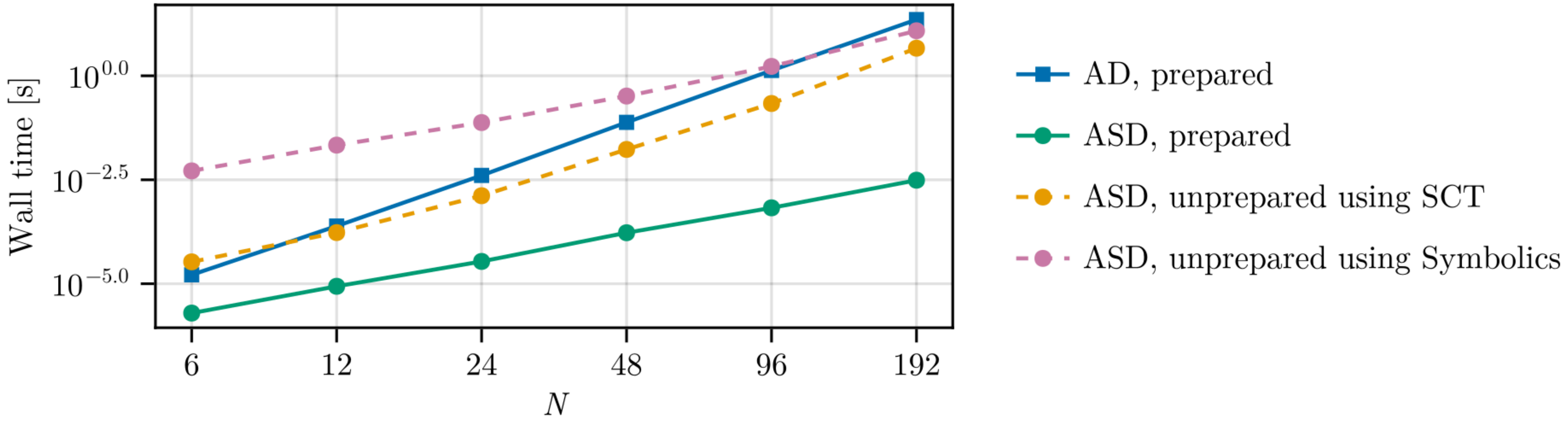
Compose `MixedMode` ASD backend from forward and reverse backends, and use it in `AutoSparse`:

```
backend = AutoSparse(
  MixedMode(fw_backend, rev_backend);
  sparsity_detector,
  bicoloring_algorithm
)

jacobian(f, backend, x)
```

Benchmarks

Jacobian benchmark: Discretized Brusselator PDE



- Sparsity Pattern Detection used to be the Bottleneck
- Benchmark from [Gow+19]

Hessian benchmark: ACOPF

Problem		Sparsity		Hessian computation[1]				
Name	Inputs	Zeros	Colors[2]	AD (prepared)	ASD (prepared)[3]		ASD (unprepared)[3]	
3_lmbd	24	91.15%	6	$1.82 \cdot 10^{-4}$	$\mathbf{8.29 \cdot 10^{-5}}$	**(2.2)**	$1.45 \cdot 10^{-4}$	(1.3)
60_c	518	99.56%	12	$1.15 \cdot 10^{-1}$	$\mathbf{2.36 \cdot 10^{-3}}$	**(48.6)**	$8.61 \cdot 10^{-3}$	(13.3)
240_pserc	2558	99.91%	16	$3.51 \cdot 10^{0}$	$\mathbf{2.50 \cdot 10^{-2}}$	**(140.2)**	$1.04 \cdot 10^{-1}$	(33.6)
1951_rte	15018	99.98%	20	$2.00 \cdot 10^{2}$	$\mathbf{1.54 \cdot 10^{-1}}$	**(1293.4)**	$1.00 \cdot 10^{0}$	(199.1)
2746wp_k	19520	99.99%	14	$3.53 \cdot 10^{2}$	$\mathbf{1.77 \cdot 10^{-1}}$	**(1991.4)**	$1.51 \cdot 10^{0}$	(234.5)
3375wp_k	24350	99.99%	18	$6.25 \cdot 10^{2}$	$\mathbf{2.54 \cdot 10^{-1}}$	**(2463.9)**	$1.71 \cdot 10^{0}$	(365.1)

[1] Wall time in seconds.
[2] Number of colors resulting from greedy symmetric coloring.
[3] In parentheses: Wall time ratio compared to prepared AD (higher is better).

Hessian of Lagrangian of optimization problems from power systems [Bab+21]

Joint work with

Guillaume Dalle
DI, SCT, SMC

Alexis Montoison
SMC

Thank you for your time!

BIFOLD
TU berlin
École nationale des Ponts et Chaussées | Institut Polytechnique de Paris
LVMT Laboratoire Ville Mobilité Transport
Argonne National Laboratory

Bibliography

Babaeinejadsarookolaee, S., Birchfield, A., Christie, R. D., Coffrin, C., DeMarco, C., Diao, R., Ferris, M., Fliscounakis, S., Greene, S., Huang, R., Josz, C., Korab, R., Lesieutre, B., Maeght, J., Mak, T. W. K., Molzahn, D. K., Overbye, T. J., Panciatici, P., Park, B., … Zimmerman, R. (2021, January). *The Power Grid Library for Benchmarking AC Optimal Power Flow Algorithms*. arXiv. https://doi.org/10.48550/arXiv.1908.02788

Blondel, M., Berthet, Q., Cuturi, M., Frostig, R., Hoyer, S., Llinares-Lopez, F., Pedregosa, F., & Vert, J.-P. (2022). Efficient and Modular Implicit Differentiation. *Advances in Neural Information Processing Systems*, *35*, 5230–5242. https://proceedings.neurips.cc/paper_files/paper/2022/hash/228b9279ecf9bbafe582406850c57115-Abstract-Conference.html

Coleman, T. F., & Verma, A. (1998). The Efficient Computation of Sparse Jacobian Matrices Using Automatic Differentiation. *SIAM Journal on Scientific Computing*, *19*(4), 1210–1233. https://doi.org/10.1137/S1064827595295349

Curtis, A. R., Powell, M. J. D., & Reid, J. K. (1974). On the Estimation of Sparse Jacobian Matrices. *IMA Journal of Applied Mathematics*, *13*(1), 117–119. https://doi.org/10.1093/imamat/13.1.117

Dalle, G., & Hill, A. (2025). A Common Interface for Automatic Differentiation. *Arxiv Preprint Arxiv:2505.05542*.

Dixon, L. C. W., Maany, Z., & Mohseninia, M. (1990). Automatic differentiation of large sparse systems. *Journal of Economic Dynamics and Control*, *14*(2), 299–311. https://doi.org/10.1016/0165-1889(90)90023-A

Gebremedhin, A. H., Manne, F., & Pothen, A. (2005). What Color Is Your Jacobian? Graph Coloring for Computing Derivatives. *SIAM Review*, *47*(4), 629–705. https://doi.org/10/cmwds4

Geitner, U., Utke, J., & Griewank, A. (1995,). *Automatic Computation of Sparse Jacobians by Applying the Method of Newsam and Ramsdell*. https://www.semanticscholar.org/paper/Automatic-Computation-of-Sparse-Jacobians-by-the-of-Geitner-Utke/1ed218348fff39e9642d7b7ac38cf0dd66aea47b

Gowda, S., Ma, Y., Churavy, V., Edelman, A., & Rackauckas, C. (2019, September). Sparsity Programming: Automated Sparsity-Aware Optimizations in Differentiable Programming. *Program Transformations for ML Workshop at NeurIPS 2019*. https://openreview.net/forum?id=rJlPdcY38B

Griewank, A., & Reese, S. (1991). *On the calculation of Jacobian matrices by the Markowitz rule* (Issue ANL/CP-75176; CONF-910189-4). https://www.osti.gov/biblio/10118065

Griewank, A., & Walther, A. (2008). *Evaluating derivatives: principles and techniques of algorithmic differentiation* (2nd ed). Society for Industrial and Applied Mathematics. https://epubs.siam.org/doi/book/10.1137/1.9780898717761

Gu, F., Chang, H., Zhu, W., Sojoudi, S., & El Ghaoui, L. (2020). Implicit graph neural networks. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, & H. Lin (Eds.), *Advances in neural information processing systems: Vol. 33. Advances in neural information processing systems*. https://proceedings.neurips.cc/paper/2020/file/8b5c8441a8ff8e151b191c53c1842a38-Paper.pdf

Hill, A., & Dalle, G. (2025, January). *Sparser, Better, Faster, Stronger: Efficient Automatic Differentiation for Sparse Jacobians and Hessians*. arXiv. https://doi.org/10.48550/arXiv.2501.17737

Hossain, A. K. M. S., & Steihaug, T. (1998). Computing a sparse Jacobian matrix by rows and columns. *Optimization Methods and Software*, *10*(1), 33–48. https://doi.org/10.1080/10556789808805700

Montoison, A., Dalle, G., & Gebremedhin, A. (2025). Revisiting Sparse Matrix Coloring and Bicoloring. *Arxiv Preprint Arxiv:2505.07308*.

Powell, M. J. D., & Toint, \. P. L. (1979). On the Estimation of Sparse Hessian Matrices. *SIAM Journal on Numerical Analysis*, *16*(6), 1060–1074. https://doi.org/10.1137/0716078

Walther, A. (2008). Computing sparse Hessians with automatic differentiation. *ACM Transactions on Mathematical Software*, *34*(1), 1–15. https://doi.org/10.1145/1322436.1322439

Walther, A. (2012). On the Efficient Computation of Sparsity Patterns for Hessians. In S. Forth, P. Hovland, E. Phipps, J. Utke, & A. Walther (Eds.), *Recent Advances in Algorithmic Differentiation: Recent Advances in Algorithmic Differentiation*. https://doi.org/10.1007/978-3-642-30023-3_13

Performant
Sparsity Pattern Detection

Revisiting Index Sets

(a)

0.0	1.85	0.0	2.21	0.0
0.0	0.0	0.0	0.97	-2.19
0.0	-0.58	1.47	0.0	0.0
-1.91	0.0	-0.46	0.0	0.0

(b)

0	≠ 0	0	≠ 0	0
0	0	0	≠ 0	≠ 0
0	≠ 0	≠ 0	0	0
≠ 0	0	≠ 0	0	0

(c)

{2,4}
{4,5}
{2,3}
{1,3}

- **First-order:** each scalar contains a set of indices $\left\{i \mid \frac{\partial f}{\partial x_i} \neq 0\right\}$
- **Second-order:** in addition to the first-order set, each scalar contains a set of index tuples $\left\{(i, j) \mid \frac{\partial^2 f}{\partial x_i \partial x_j} \neq 0\right\}$

These can be **local or global**.

First-order Propagation Rules

Let $\alpha(\boldsymbol{x})$ and $\beta(\boldsymbol{x})$ be two intermediate scalar quantities in the computational graph of $f(\boldsymbol{x})$.
We compute a new scalar $\gamma(\boldsymbol{x})$ by applying the two-argument operator φ:

$$\gamma(\boldsymbol{x}) = \varphi(\alpha(\boldsymbol{x}), \beta(\boldsymbol{x}))$$

According to the chain rule,

$$\nabla\gamma = \partial_1\varphi \cdot \nabla\alpha + \partial_2\varphi \cdot \nabla\beta \, .$$

Using the indicator function $\mathbf{1}[x] = \mathbf{1}_{x \neq 0}[x]$ and $\vee$ for element-wise `OR`,
and denoting the sparsity patterns of α and β as $\mathbf{1}[\nabla\alpha]$ and $\mathbf{1}[\nabla\beta]$ respectively,

$$\mathbf{1}[\nabla\gamma] \leq \mathbf{1}[\partial_1\varphi] \cdot \mathbf{1}[\nabla\alpha] \vee \mathbf{1}[\partial_2\varphi] \cdot \mathbf{1}[\nabla\beta] \, .$$

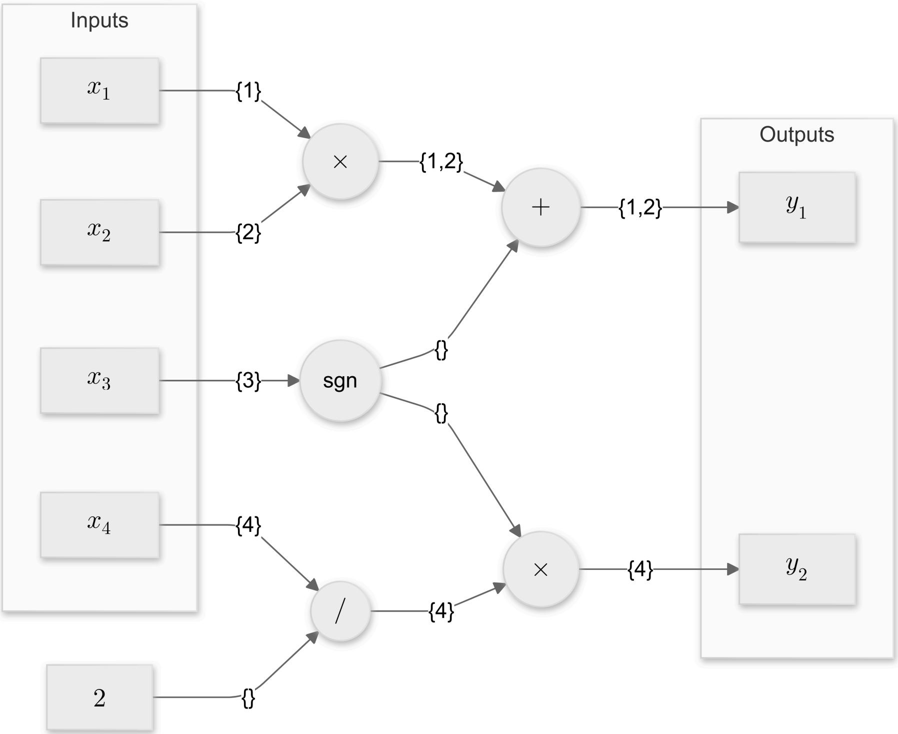
The propagation of first-order sparsity patterns by φ only depends on two values:

$$\mathbf{1}[\partial_1\varphi] \qquad \mathbf{1}[\partial_2\varphi]$$

Based on [Wal08], notation from [HD25]

First-order Propagation: Example

Propagating index sets [Wal08]:



Computational graph for

$$f(\boldsymbol{x}) = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} x_1 x_2 + \operatorname{sign}(x_3) \\ \operatorname{sign}(x_3) \cdot \left(\frac{x_4}{2}\right) \end{pmatrix}.$$

Jacobian has 3 nonzero coefficients:

$$J_f(\boldsymbol{x}) = \begin{pmatrix} x_2 & x_1 & 0 & 0 \\ 0 & 0 & 0 & \frac{\operatorname{sign}(x_3)}{2} \end{pmatrix}$$

Toy Implementation: Global sparsity detection

Operator overloading on new “tracer” number type:

```julia
import Base: +, *, /, sign

struct Tracer
  inds::Set{Int}
end

Tracer() = Tracer(Set{Int}())

+(a::Tracer, b::Tracer) = Tracer(a.inds ∪ b.inds)
*(a::Tracer, b::Tracer) = Tracer(a.inds ∪ b.inds)
/(a::Tracer, b::Int)    = Tracer(a.inds)
sign(a::Tracer)         = Tracer()  # zero derivatives
```

Toy Implementation: Demonstration

Multiple dispatch: no code transformation needed

```
julia> f(x) = [x[1] * x[2] * sign(x[3]), sign(x[3]) * x[4] / 2];

julia> x = Tracer.(Set.([1, 2, 3, 4]))
4-element Vector{Tracer}:
 Tracer(Set([1]))
 Tracer(Set([2]))
 Tracer(Set([3]))
 Tracer(Set([4]))

julia> f(x)
2-element Vector{Tracer}:
 Tracer(Set([2, 1]))
 Tracer(Set([4]))
```

Matches expected pattern of $J_f(x) = \begin{pmatrix} x_2 & x_1 & 0 & 0 \\ 0 & 0 & 0 & \frac{\text{sign}(x_3)}{2} \end{pmatrix}$.

Second-order Propagation Rules

Analogous to the previous slide, for operators $\gamma(\boldsymbol{x}) = \varphi(\alpha(\boldsymbol{x}), \beta(\boldsymbol{x}))$:
using $\vee$ for element-wise `OR`, $\mathbf{1}[x] = \mathbf{1}_{x \neq 0}[x]$

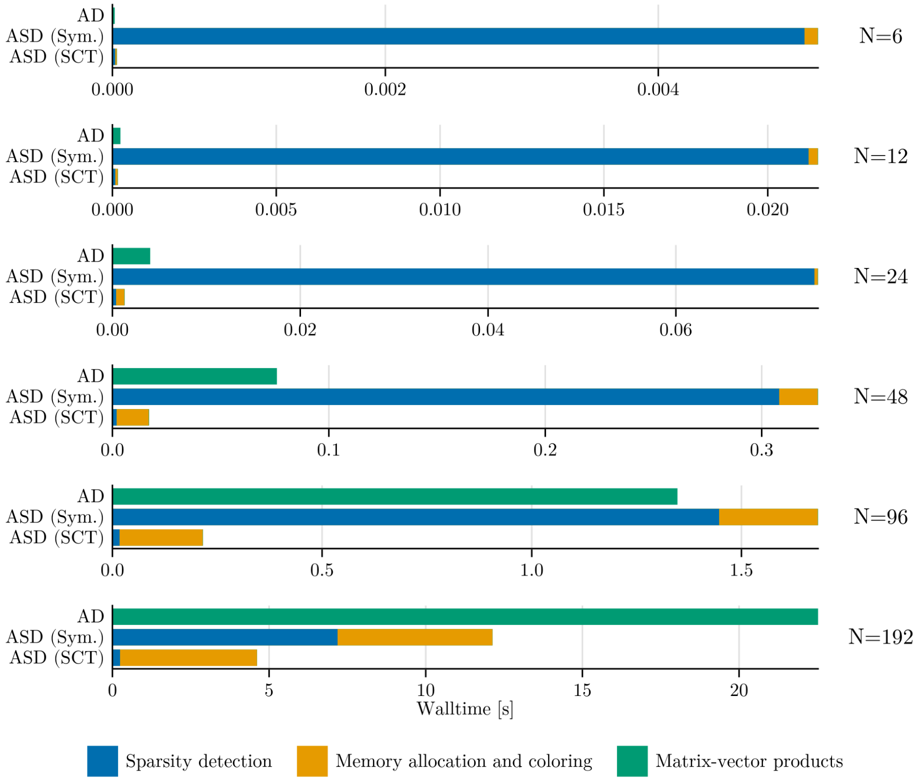
$$\mathbf{1}[\nabla^2\gamma] \leq \left| \begin{array}{llll} & \mathbf{1}[\partial_1\varphi] \cdot \mathbf{1}[\nabla^2\alpha] & \vee & \mathbf{1}[\partial_2\varphi] \cdot \mathbf{1}[\nabla^2\beta] \\ \vee & \mathbf{1}[\partial_1^2\varphi] \cdot (\mathbf{1}[\nabla\alpha] \vee \mathbf{1}[\nabla\alpha]^\top) & \vee & \mathbf{1}[\partial_2^2\varphi] \cdot (\mathbf{1}[\nabla\beta] \vee \mathbf{1}[\nabla\beta]^\top) \\ \vee & \mathbf{1}[\partial_{12}^2\varphi] \cdot (\mathbf{1}[\nabla\alpha] \vee \mathbf{1}[\nabla\beta]^\top) & \vee & \mathbf{1}[\partial_{12}^2\varphi] \cdot (\mathbf{1}[\nabla\beta] \vee \mathbf{1}[\nabla\alpha]^\top) \end{array} \right.$$

Propagation of sparsity patterns up to Hessians only depends on five values:

$\mathbf{1}[\partial_1\varphi] \quad \mathbf{1}[\partial_2\varphi] \quad \mathbf{1}[\partial_1^2\varphi] \quad \mathbf{1}[\partial_2^2\varphi] \quad \mathbf{1}[\partial_{12}^2\varphi]$

Based on [Wal08], notation from [HD25]

Sparsity Pattern Detection used to be the Bottleneck

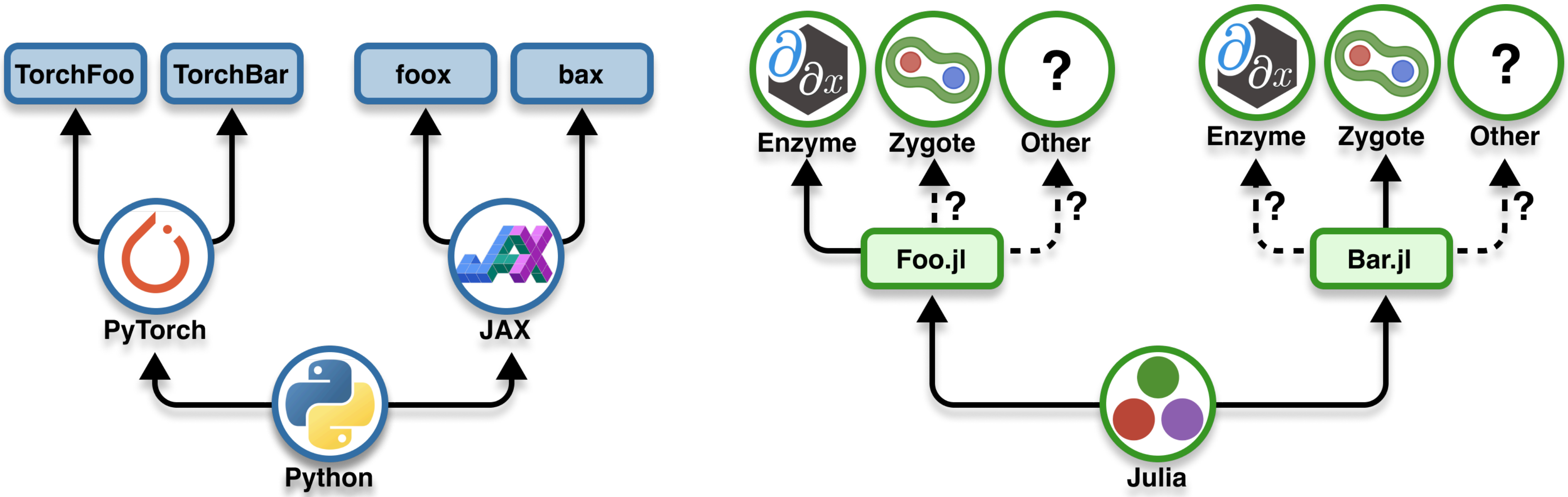


Jacobian bechmark on discretized Brusselator PDE from [Gow+19]

DifferentiationInterface

Automatic Differentiation in Julia

From to our **JuliaCon 2024 talk "Gradients for everyone"**:



Julia has **dozens** of AD backends:

- each with different strengths, caveats ← **DI can't fix those**
- each with their own syntax ← **but DI can fix this**

DifferentiationInterface.jl [DH25]

Common interface for 13 AD backends

- Operators:
  - high-level: `gradient`, `jacobian`, `hessian`, `derivative`, `second_derivative`
  - lower-level: `pushforward` (JVP), `pullback` (VJP), `hvp`
- Variants:
  - out-of-place `y = f(x)` or in-place `f!(y, x)`
  - with or without primal (e.g. `value_and_gradient`)
- “Context arguments”: constants and caches
- Preparation mechanisms

Applications

Newton's method

Root-finding

Solve $F(x) = 0$ by iterating

$$x_{t+1} = x_t - \underbrace{[\partial F(x_t)]}_{\text{Jacobian}}^{-1} F(x_t)$$

Optimization

Solve $\min f(x)$ by iterating

$$x_{t+1} = x_t - \underbrace{[\nabla^2 f(x_t)]}_{\text{Hessian}}^{-1} \nabla f(x_t)$$

Linear systems involving a derivative matrix A.

Implicit differentiation

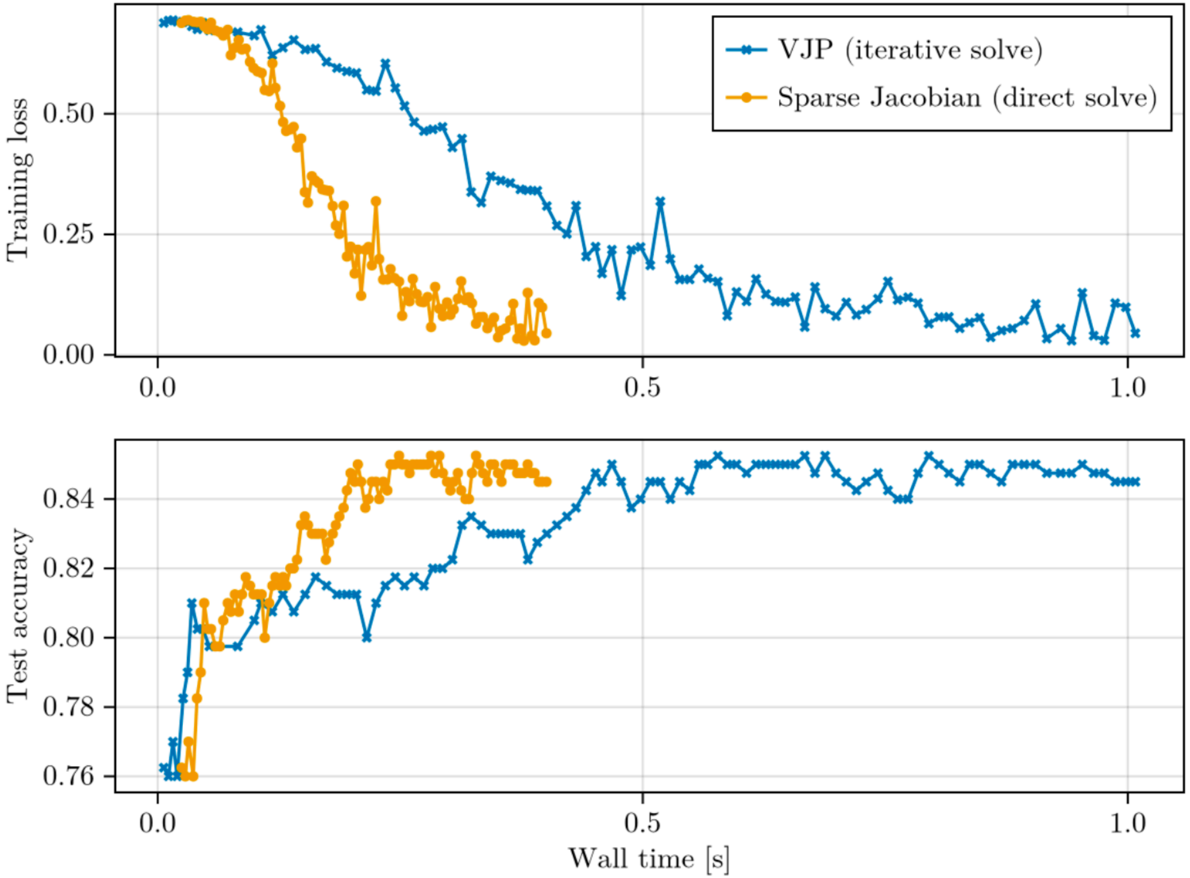
- Differentiate $x \to y(x)$ knowing **optimality conditions** $c(x, y(x)) = 0$.
- **Applications:** fixed-point iterations, optimization problems.
- **Implicit function theorem** [Blo+22]:

$$\partial_1 c(x, y(x)) + \partial_2 c(x, y(x)) \cdot \partial y(x) = 0$$

$$\partial y(x) = -\underbrace{[\partial_2 c(x, y(x))]}_{\text{Jacobian}}^{-1} \partial_1 c(x, y(x))$$

Linear system involving a derivative matrix A.

Jacobian benchmark: Implicit Differentiation



Implitic Graph Neural Networks [Gu+20]