Sparser, better, faster, stronger

Automatic differentiation with a lot of zeros

Guillaume Dalle* – LVMT, École des Pont (gdalle.github.io)

Laboratoire Jean Kuntzmann, 19.06.2025

*joint work with Adrian Hill, Alexis Montoison and Assefaw Gebremedhin

Agenda

1. Motivation
2. Automatic differentiation
3. Leveraging sparsity
4. Implementation

Motivation

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 $\min_x f(x)$ by iterating

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

Linear system involving a derivative matrix A.

Implicit differentiation

Differentiate $x \mapsto y(x)$ knowing **conditions** $c(x, y(x)) = 0$.

Applications: fixed-point iterations, optimization problems.

Implicit function theorem

$$\frac{\partial}{\partial x} c(x, y(x)) + \frac{\partial}{\partial y} c(x, y(x)) \cdot \partial y(x) = 0$$

$$\partial y(x) = -\underbrace{\left[\frac{\partial}{\partial y} c(x, y(x))\right]}_{\text{jacobian}}^{-1} \frac{\partial}{\partial x} c(x, y(x))$$

Linear system involving a derivative matrix A.

Linear systems of equations

How to solve $Au = v$?

Direct method (LU, Cholesky)

1. Decompose the matrix A.
2. Get an exact solution by substitution.

Requires storing A explicitly.

Iterative method (CG, GMRES)

1. Rephrase as $\min_u \|Au - v\|^2$.
2. Get an approximate solution.

Only requires matrix-vector products $u \mapsto Au$.

Conventional wisdom

- Jacobian and Hessian matrices are **too large** to compute or store
- We can only access linear maps $u \mapsto Au$ (JVPs, VJPs, HVPs)
- Linear systems $A^{-1}v$ must be solved with **iterative methods**
- Downsides: each iteration is expensive, convergence is tricky

The benefits of sparsity

- Jacobian and Hessian matrices have **mostly zero coefficients**
- We can compute and store A explicitly
- Linear systems $A^{-1}v$ can be solved with iterative **or direct** methods
- Upsides: faster iterations or exact solves, efficient linear algebra

Automatic differentiation

Numeric differentiation

Input	**Output**
program computing the function $x \mapsto f(x)$	approximate value of the directional derivative $\dfrac{f(x+\varepsilon d) - f(x)}{\varepsilon}$

Automatic / algorithmic differentiation

Input	**Output**
program computing the function $x \mapsto f(x)$	program computing the differential $x \mapsto \partial f(x)$ which is a linear map $\partial f(x) : u \mapsto \partial f(x)[u]$

When talking about Jacobian matrices, I will write $\partial_{\mathrm{mat}} f(x)$ instead.

AD under the hood

Two ingredients only:

1. hardcode basic derivatives ($+$, $\times$, exp, log, ...)
2. handle composition $f = g \circ h$

Composition

For a function $f = g \circ h$, the **chain rule** gives its differential:

$$\begin{aligned} \text{standard} \qquad \partial f(x) &= \partial g(h(x)) \circ \partial h(x) \\ \text{adjoint} \qquad \partial f(x)^* &= \partial h(x)^* \circ \partial g(h(x))^* \end{aligned}$$

These linear maps apply as follows:

$$\begin{aligned} \text{forward} \qquad \partial f(x) &: u \xmapsto{\partial h(x)} v \xmapsto{\partial g(h(x))} w \\ \text{reverse} \qquad \partial f(x)^* &: u \xleftarrow[\partial h(x)^*]{} v \xleftarrow[\partial g(h(x))^*]{} w \end{aligned}$$

Why linear maps?

The chain rule has a matrix equivalent:

$$\partial_{\mathrm{mat}}(g \circ h)(x) = \partial_{\mathrm{mat}} g(h(x)) \cdot \partial_{\mathrm{mat}} h(x)$$
$$\partial_{\mathrm{mat}}(g \circ h)(x)^T = \partial_{\mathrm{mat}} h(x)^T \cdot \partial_{\mathrm{mat}} g(h(x))^T$$

Working with linear maps avoids allocation and manipulation of **intermediate Jacobian matrices**.

Essential for neural networks!

Pocket AD

```julia
# Basic rules

using LinearAlgebra

A, b = rand(2, 3), rand(2)
residuals(x) = A * x - b
∂(::typeof(residuals)) = x → (u → A * u)  # ℝ³ → ℝ²
∂ᵀ(::typeof(residuals)) = x → (v → adjoint(A) * v)  # ℝ² → ℝ³

sqnorm(r) = sum(abs2, r)
∂(::typeof(sqnorm)) = r → (v → dot(2r, v))  # ℝ² → ℝ
∂ᵀ(::typeof(sqnorm)) = r → (w → 2r .* w)  # ℝ → ℝ²
```

Pocket AD

```julia
# Composition

function ∂(f::ComposedFunction)
    g, h = f.outer, f.inner
    return x -> ∂(g)(h(x)) ∘ ∂(h)(x)
end

function ∂ᵀ(f::ComposedFunction)
    g, h = f.outer, f.inner
    return x -> ∂ᵀ(h)(x) ∘ ∂ᵀ(g)(h(x))
end
```

Pocket AD

```julia
julia> import ForwardDiff as FD, Zygote

julia> f = sqnorm ∘ residuals;

julia> x, Δx = rand(3), [1, 0, 0];
```

```julia
julia> ∂(f)(x)(Δx)  # partial derivative
0.8691056836969242

julia> ∂ᵀ(f)(x)(1)  # gradient
3-element Vector{Float64}:
 0.8691056836969242
 0.9973491983376236
 0.5768822265195823
```

```julia
julia> FD.derivative(t -> f(x + t * Δx), 0)
0.8691056836969242

julia> Zygote.gradient(f, x)[1]
3-element Vector{Float64}:
 0.8691056836969242
 0.9973491983376236
 0.5768822265195823
```

Two modes

Forward-mode AD computes Jacobian-Vector Products (JVPs) = "pushforward" of an input perturbation:

$$u \mapsto \partial f(x)[u] = Ju$$

Reverse-mode AD computes Vector-Jacobian Products (VJPs) = "pullback" of an output sensitivity:

$$v \mapsto \quad \partial f(x)^*[v] = J^T v = v^T J$$

Theorem (Baur-Strassen): cost of 1 JVP or VJP $\propto$ cost of 1 function evaluation

What about gradients?

Reverse mode computes gradients for roughly the same cost as the function itself:

$$\nabla f(x) = \partial f(x)^*[1]$$

Makes deep learning possible.

The devil is in the details: higher memory footprint.

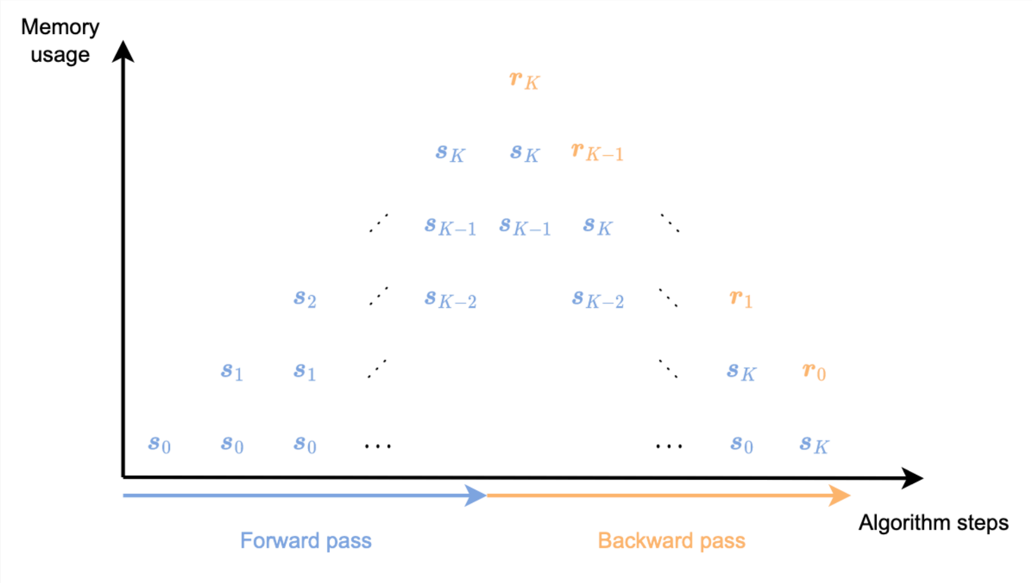


Figure 1: Blondel & Roulet (2024)

What about second-order?

The Hessian matrix is the Jacobian matrix of the gradient function.

A Hessian-Vector Product (HVP) can be computed as the JVP of a VJP, in **forward-over-reverse mode**:

$$\nabla^2 f(x)[v] = \partial(\nabla f)(x)[v] = \partial(\partial^* f(x)[1])[v]$$

Leveraging sparsity

From maps to matrices

To compute the Jacobian matrix J of a composition $f : \mathbb{R}^m \longrightarrow \mathbb{R}^n$:

- ~~product of intermediate Jacobian matrices~~
- reconstruction from several JVPs or VJPs

	forward mode	**reverse mode**
idea	1 JVP gives 1 column	1 VJP gives 1 row
formula	$J_{\cdot,j} = \partial f(x)[e_j]$	$J_{i,\cdot} = \partial f(x)^*[e_i]$
cost	n JVPs (input dimension)	m JVPs (output dimension)

Using fewer products

When the Jacobian is sparse, we can compute it faster.

If columns $j_1, \ldots, j_k$ of J are structurally orthogonal (their nonzeros never overlap), we deduce them all from a single JVP:

$$J_{j_1} + \ldots + J_{j_k} = \partial f(x)\left[e_{j_1} + \ldots + e_{j_k}\right]$$

Once we have grouped columns, sparse AD has two steps:

3. one JVP for each group $c = \{j_1, \ldots, j_k\}$
4. decompression into individual columns $j_1, \ldots, j_k$

The gist in one slide

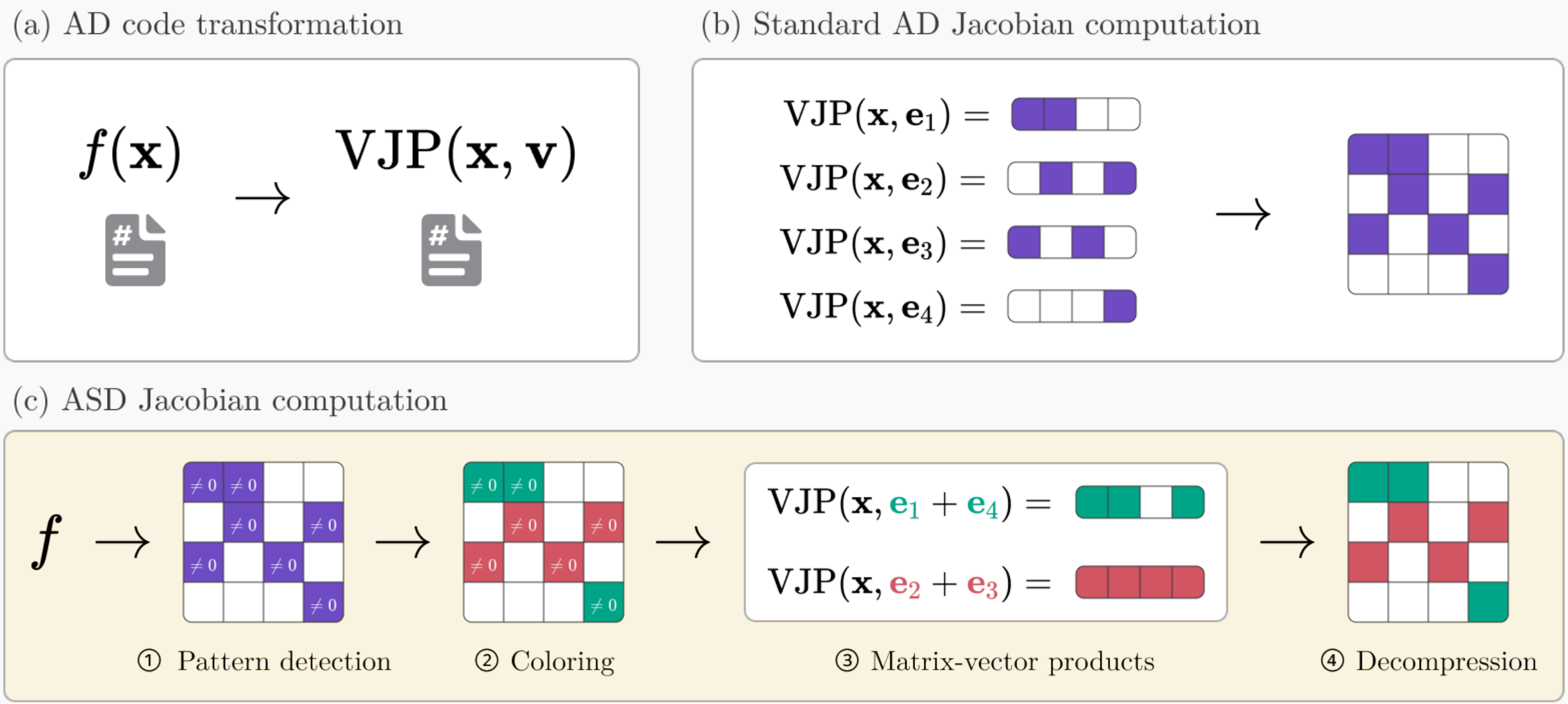


Figure 2: Hill & Dalle (2025)

Two preliminary steps

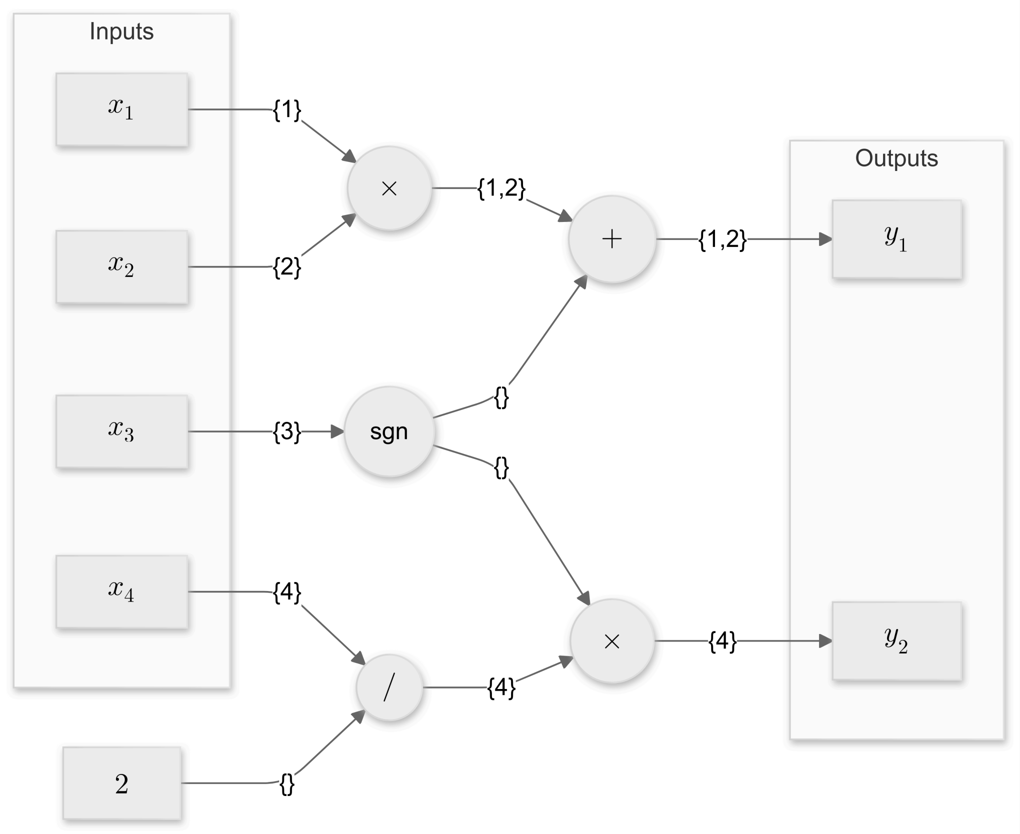
When grouping columns, we want to

- guarantee structural orthogonality (correctness)
- form the smallest number of groups (efficiency)

preparation	**execution**
1. pattern detection 2. coloring	3. matrix-vector products 4. decompression

The preparation phase can be amortized across several inputs.

Tracing dependencies in the computation graph



Computation graph for

$$y_1 = x_1 x_2 + \mathrm{sign}(x_3)$$
$$y_2 = \mathrm{sign}(x_3) \times \left(\frac{x_4}{2}\right)$$

Its Jacobian will have 3 nonzero coefficients.

Pocket pattern detection

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

struct Tracer
  indices::Set{Int}
end

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

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

Pocket pattern detection

Does it work?

```
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]))
```

Partitions of a matrix

Orthogonal for all (i, j) s.t. $A_{ij} \neq 0$,

- column j is alone in group $c(j)$ with a nonzero in row i

Symmetrically orthogonal for all (i, j) s.t. $A_{ij} \neq 0$,

- either column j is alone in group $c(j)$ with a nonzero in row i
- or column i is alone in group $c(i)$ with a nonzero in row j

Each partition can be reformulated as a specific coloring problem[2].

[2] Gebremedhin et al. (2005)

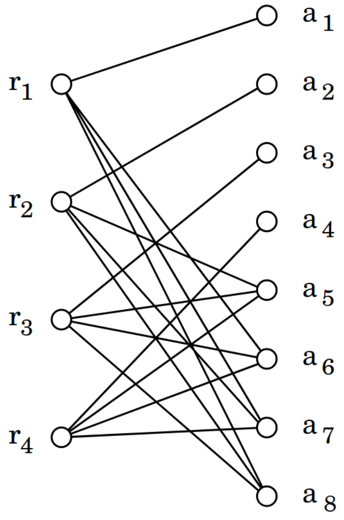
Graph representations of a matrix

Column intersection $(j_1, j_2) \in \mathcal{E} \iff \exists i, A_{ij_1} \neq 0$ and $A_{ij_2} \neq 0$

Bipartite $(i, j) \in \mathcal{E} \iff A_{ij} \neq 0$ (2 vertex sets $\mathcal{I}$ and $\mathcal{J}$)

Adjacency (sym.) $(i, j) \in \mathcal{E} \iff i \neq j$ & $A_{ij} \neq 0$

$$\begin{bmatrix} a_{11} & 0 & 0 & 0 & 0 & a_{16} & a_{17} & a_{18} \\ 0 & a_{22} & 0 & 0 & a_{25} & 0 & a_{27} & a_{28} \\ 0 & 0 & a_{33} & 0 & a_{35} & a_{36} & 0 & a_{38} \\ 0 & 0 & 0 & a_{44} & a_{45} & a_{46} & a_{47} & 0 \end{bmatrix}$$



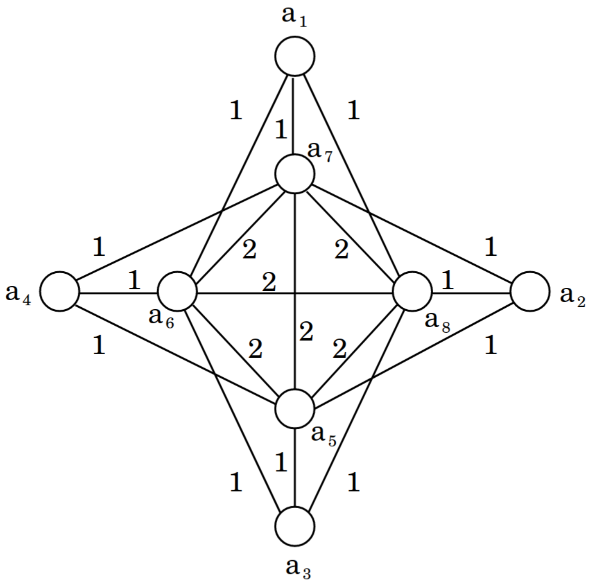


Figure 3: Gebremedhin et al. (2005)

Jacobian coloring

Coloring of intersection graph / distance-2 coloring of bipartite graph

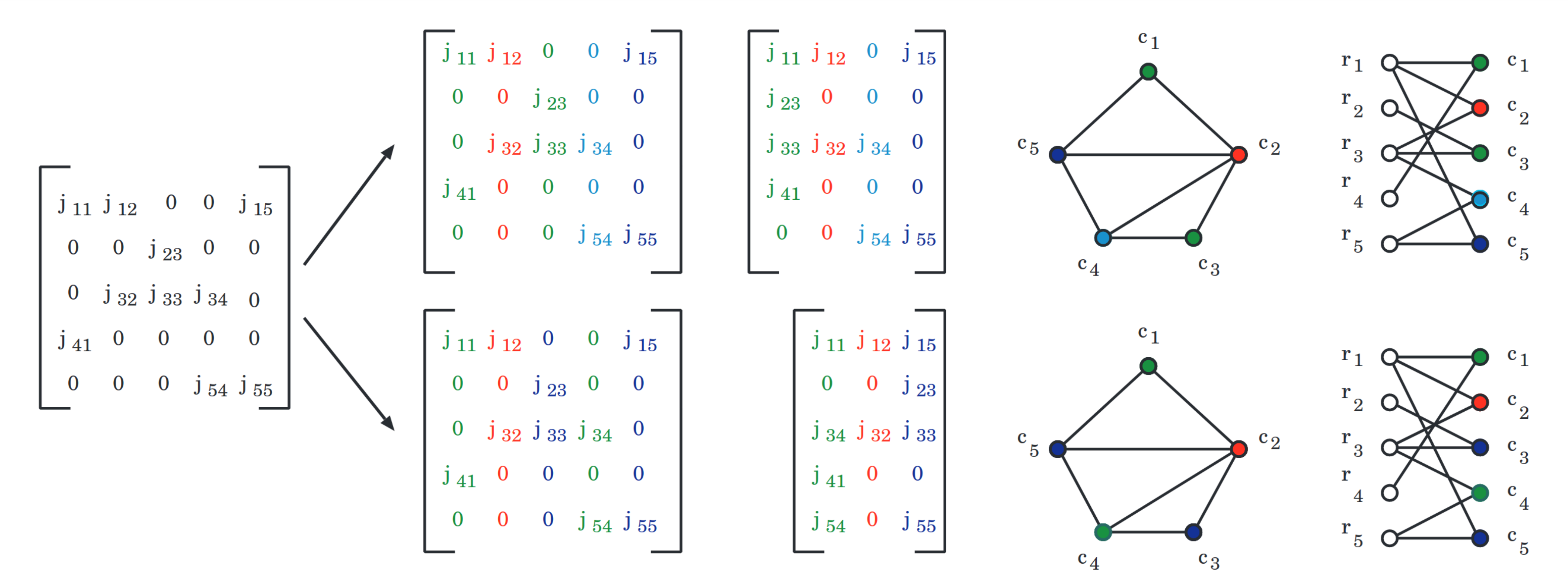


Figure 4: Gebremedhin et al. (2005)

Hessian coloring

Star coloring of adjacency graph

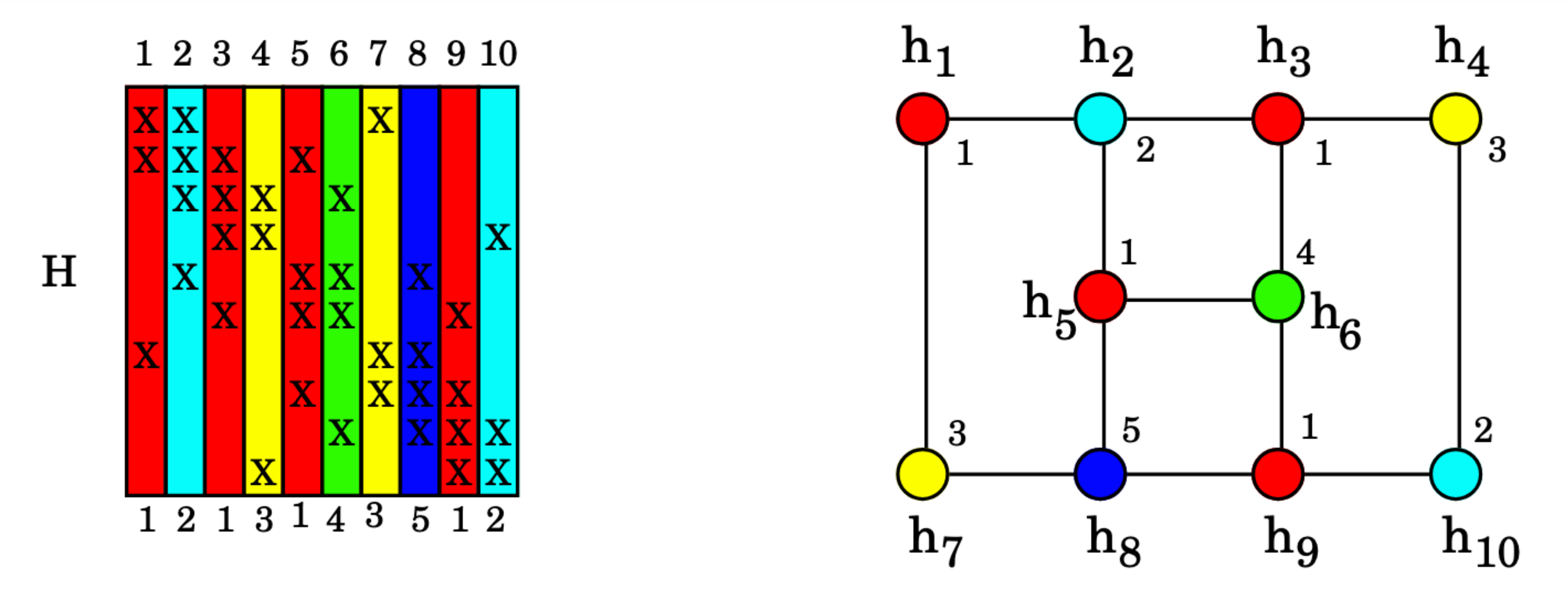


Figure 5: Gebremedhin et al. (2009)

Hessian coloring

Why a "star" coloring[3]? Consider

$$A = \begin{pmatrix} A_{kk} & A_{ki} & \cdot & \cdot \\ A_{ik} & A_{ii} & A_{ij} & \cdot \\ \cdot & A_{ji} & A_{jj} & A_{jl} \\ \cdot & \cdot & A_{lj} & A_{ll} \end{pmatrix}$$

If coloring c yields a symmetrically orthogonal partition:

- $c(i) \neq c(j)$
- $c(i) \neq c(k)$
- $c(j) \neq c(l)$

Any path on 4 vertices (i, j, k, l) must use at least 3 colors $\iff$ any 2-colored subgraph is a collection of disjoint stars (it contains no path longer than 3).

[3]Coleman & Moré (1984)

Jacobian bicoloring

Bidirectional coloring of bipartite graph, with neutral color

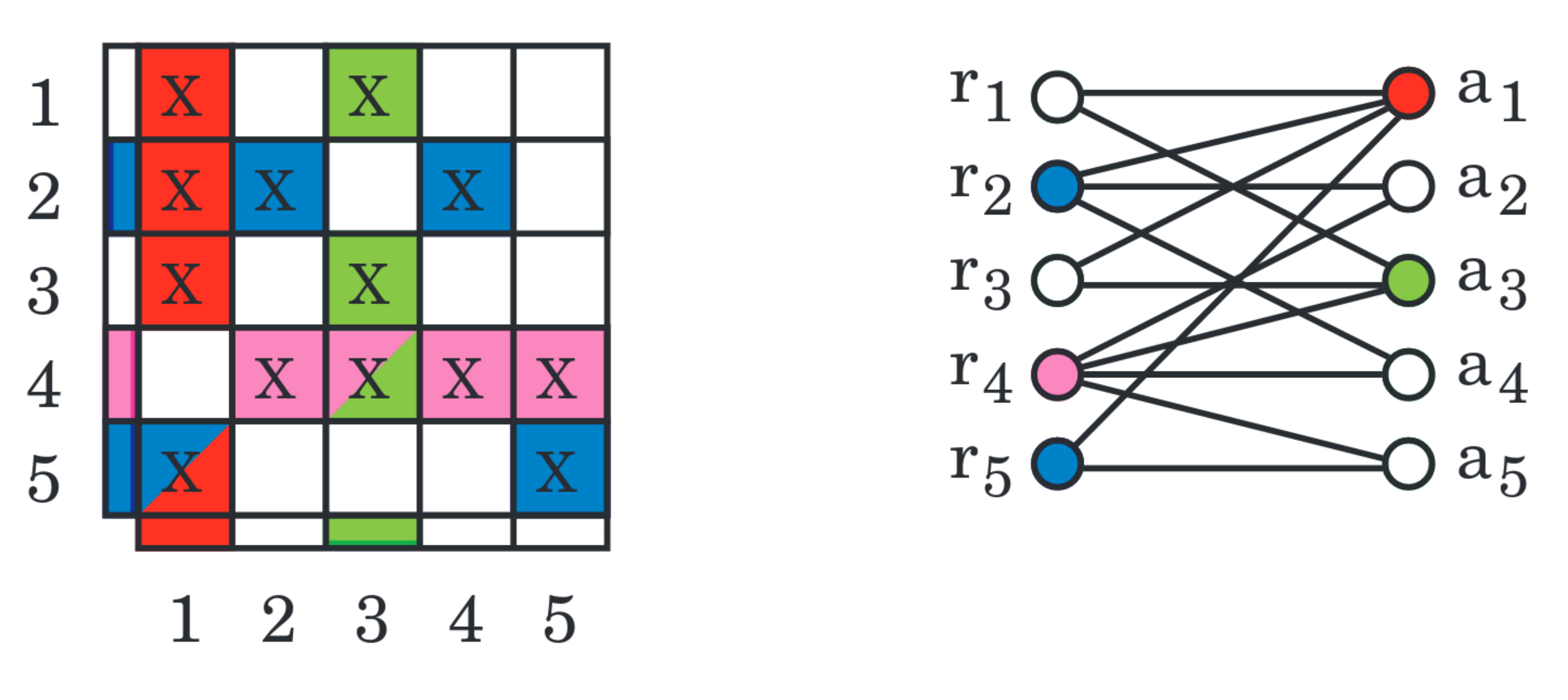


Figure 6: Gebremedhin et al. (2005)

Bicoloring from symmetric coloring [new]

To color the rows and columns of J, color the columns of $H = \begin{pmatrix} 0 & J \\ J & 0 \end{pmatrix}$

It sounds simple, but:

- Some colors may be redundant
- Detecting these is tightly linked to the two-colored structures
- Efficient decompression requires lots of preprocessing

Explanations and benchmarks in Montoison et al. (2025)

The sharp bits

Pattern detection

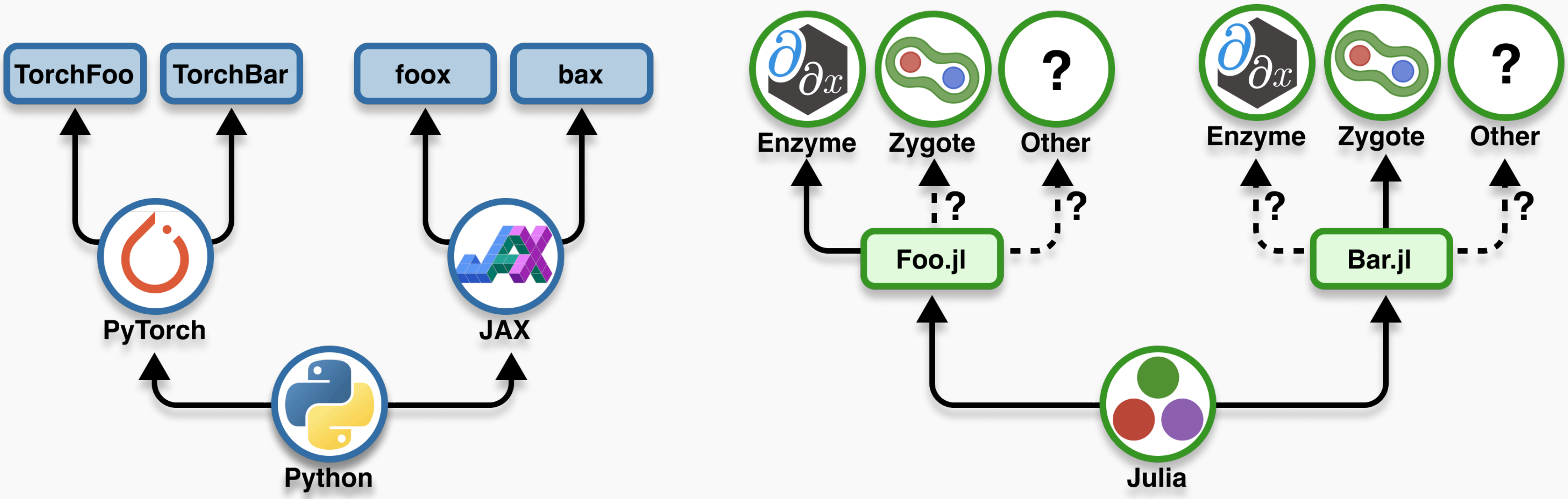
- Local versus global sparsity
- Control flow
- Linear and nonlinear interactions

Coloring

- Only heuristic algorithms
- Vertex ordering matters a lot

Implementation

AD in Python & Julia



Interfaces for experimenting [new]

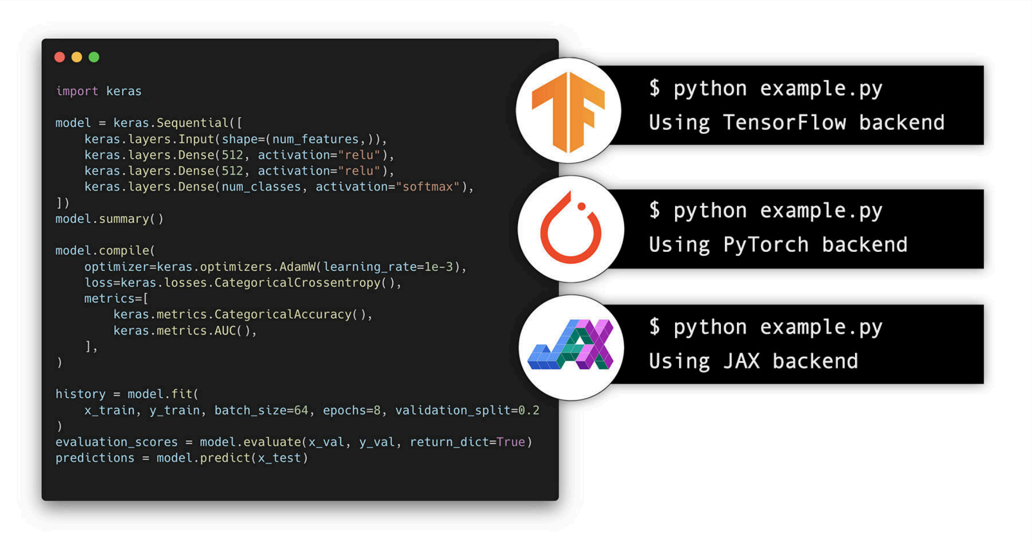
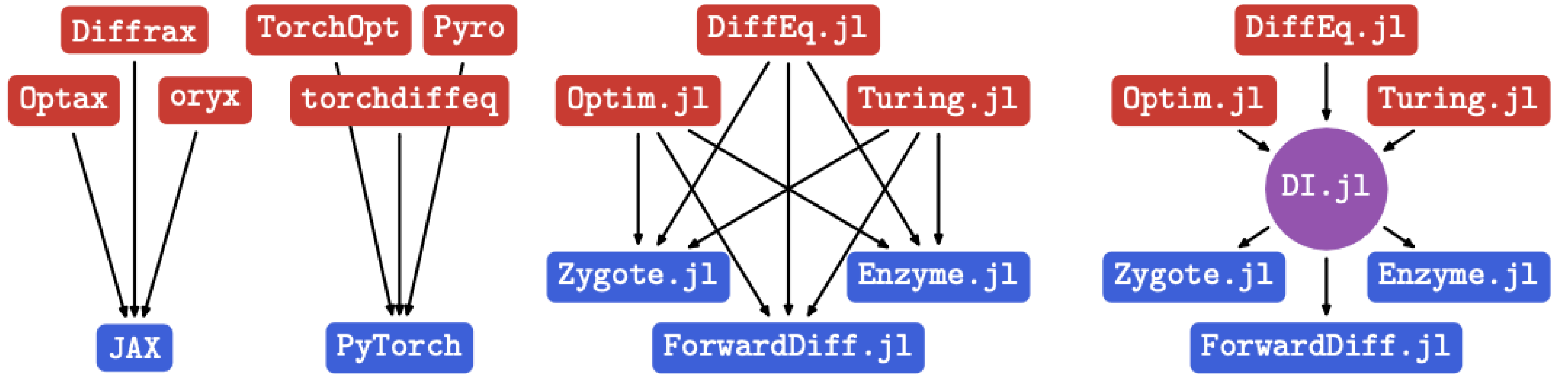


Figure 8: In Python, `Keras` supports Tensorflow, PyTorch and JAX.

Figure 9: In Julia, 14 AD backends inside `Differentiationterface.jl`

Once we have a common syntax, we can do more!

One API to rule them all



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

Previous implementations of sparse AD

- In low-level programming languages (C, Fortran)
- In closed-source languages (Matlab)
- In domain-specific languages (AMPL, CasADi)

Basically nothing in Python (either in JAX or PyTorch).

First drafts in Julia for scientific machine learning, but severely limited: single-backend, slow.

A modern sparse AD ecosystem [new]

Independent packages working together:

- **Step 1:** `SparseConnectivityTracer.jl` (Hill & Dalle, 2025)
- **Steps 2 & 4:** `SparseMatrixColorings.jl` (Montoison et al., 2025)
- **Step 3:** `DifferentiationInterface.jl` (Dalle & Hill, 2025)

	`SCT.jl`	`SMC.jl`	`DI.jl`
lines of code	5202	5184	19980
indirect dependents	461	487	896
downloads / month	7.8k	9.7k	33k

Compatible with generic code!

Impact

Users already include...

- Scientific computing: `SciML` (Julia's `scipy`)
  - Differential equations
  - Nonlinear solvers
  - Optimization
- Probabilistic programming: `Turing.jl`
- Symbolic regression: `PySR`

Live demo

This is the part where things go sideways.

Perspectives

- GPU-compatible pattern detection and coloring
- Pattern detection in JAX with program transformations
- New, unsuspected applications “just because we can”

Going further

On general AD:

- Baydin et al. (2018)
- Margossian (2019)
- Blondel & Roulet (2024)

On sparse AD:

- Gebremedhin et al. (2005)
- Griewank & Walther (2008)

Bibliography

Baydin, A. G., Pearlmutter, B. A., Radul, A. A., & Siskind, J. M. (2018). Automatic Differentiation in Machine Learning: A Survey. *Journal of Machine Learning Research*, *18*(153), 1–43. http://jmlr.org/papers/v18/17-468.html

Blondel, M., & Roulet, V. (2024, July). *The Elements of Differentiable Programming*. arXiv. https://doi.org/10.48550/arXiv.2403.14606

Coleman, T. F., & Moré, J. J. (1984). Estimation of Sparse Hessian Matrices and Graph Coloring Problems. *Mathematical Programming*, *28*(3), 243–270. https://doi.org/10.1007/BF02612334

Dalle, G., & Hill, A. (2025, May). *A Common Interface for Automatic Differentiation*. arXiv. https://doi.org/10.48550/arXiv.2505.05542

Gebremedhin, A. H., Tarafdar, A., Pothen, A., & Walther, A. (2009). Efficient Computation of Sparse Hessians Using Coloring and Automatic Differentiation. *INFORMS Journal on Computing*, *21*(2), 209–223. https://doi.org/10.1287/ijoc.1080.0286

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

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

Hill, A., & Dalle, G. (2025). Sparser, Better, Faster, Stronger: Sparsity Detection for Efficient Automatic Differentiation. *Transactions on Machine Learning Research*. https://openreview.net/forum?id=GtXSN52nIW

Margossian, C. C. (2019). A Review of Automatic Differentiation and Its Efficient Implementation. *Wires Data Mining and Knowledge Discovery*, *9*(4), e1305. https://doi.org/10.1002/widm.1305

Montoison, A., Dalle, G., & Gebremedhin, A. (2025, May). *Revisiting Sparse Matrix Coloring and Bicoloring*. arXiv. https://doi.org/10.48550/arXiv.2505.07308