Sparse automatic differentiation

From theory to practice

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

Inria, 07.03.2025

Agenda

Introduction

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 system 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[1]

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

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

Linear system involving a derivative matrix A.

[1] Blondel et al. (2022)

Conventional wisdom

- Jacobian and Hessian matrices are too large to compute or store
- We can only access lazy maps $u \longmapsto Au$ (JVPs, VJPs, HVPs[2])
- Linear systems $A^{-1}v$ must be solved with iterative methods
- Downsides: each iteration is expensive, convergence is tricky

[2] Dagréou et al. (2024)

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 even exact solves

Automatic differentiation

Pocket AD

```julia
import Base: +, *  # overload standard operators

struct Dual
    val::Float64
    der::Float64
end

+(x::Dual, y::Dual) = Dual(x.val + y.val, x.der + y.der)
*(x::Dual, y::Dual) = Dual(x.val * y.val, x.der*y.val + x.val*y.der)
+(x, y::Dual) = Dual(x, 0) + y
*(x, y::Dual) = Dual(x, 0) * y
```

Pocket AD

Does it work?

```
julia> f(x) = 1 + 2 * x + 3 * x * x;

julia> f(4)
57

julia> f(Dual(4, 1))  # exact derivative
Dual(57.0, 26.0)

julia> (f(4 + 1e-5) - f(4)) / 1e-5  # approximate derivative
26.000029998840542
```

What is AD?

input	**output**
program to compute the function $$x \longmapsto f(x)$$	program to compute the differential $$x \longmapsto \partial f(x)$$ which is a linear map $u \mapsto \partial f(x)[u]$

Two ingredients only:

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

Composition

For a function $f = g \circ h$, the chain rule gives

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

These linear maps apply as follows:

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

Forward chain rule, illustrated

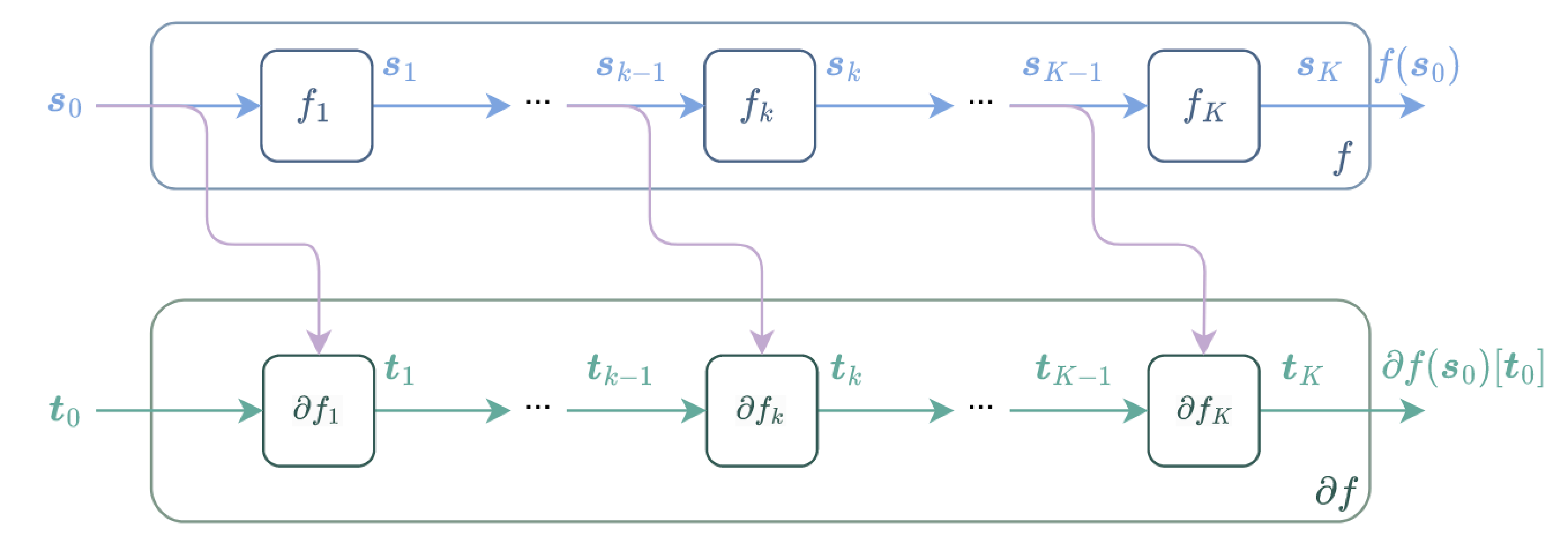


Figure 1: Blondel & Roulet (2024)

Reverse chain rule, illustrated

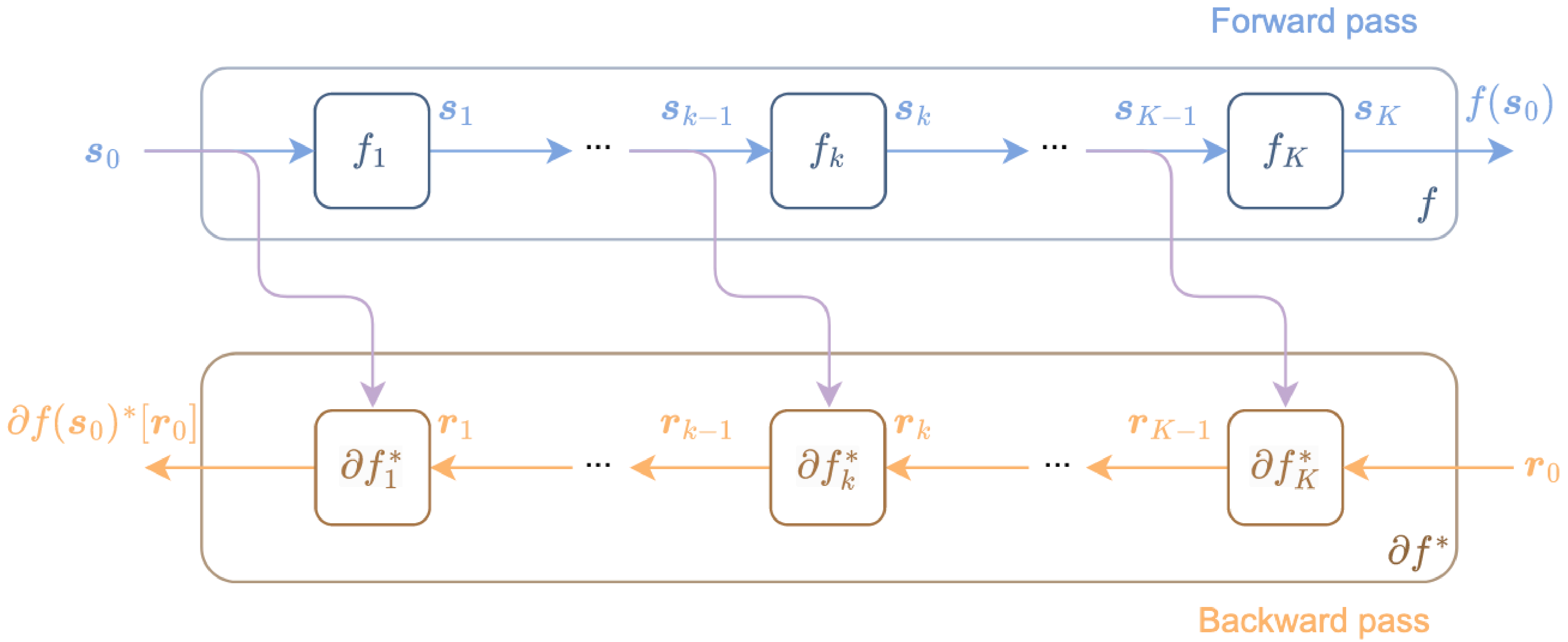


Figure 2: Blondel & Roulet (2024)

Two modes

Forward-mode AD computes Jacobian-Vector Products:

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

Reverse-mode AD computes Vector-Jacobian Products:

$$w \longmapsto \quad \partial f(x)^*[w] = w^* J$$

No need to materialize intermediate Jacobian matrices!

Theorem: cost of 1 JVP or VJP
$\propto$ cost of 1 function evaluation

Interpretations

- Forward mode: “pushforward” of an input perturbation
- Reverse mode: “pullback” of an output sensitivity

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

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

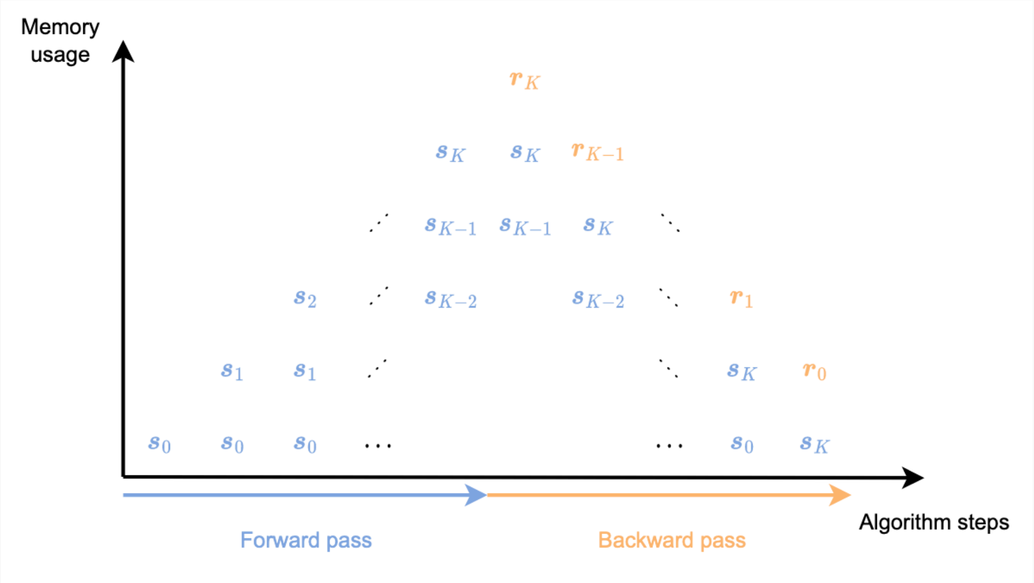


Figure 3: The devil is in the details (Blondel & Roulet, 2024)

Pocket AD, chain rule version

```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, chain rule version

```
# 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, chain rule version

```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
```

Exploiting 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[3].

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

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

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

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

[3] Curtis et al. (1974)

The gist in one slide

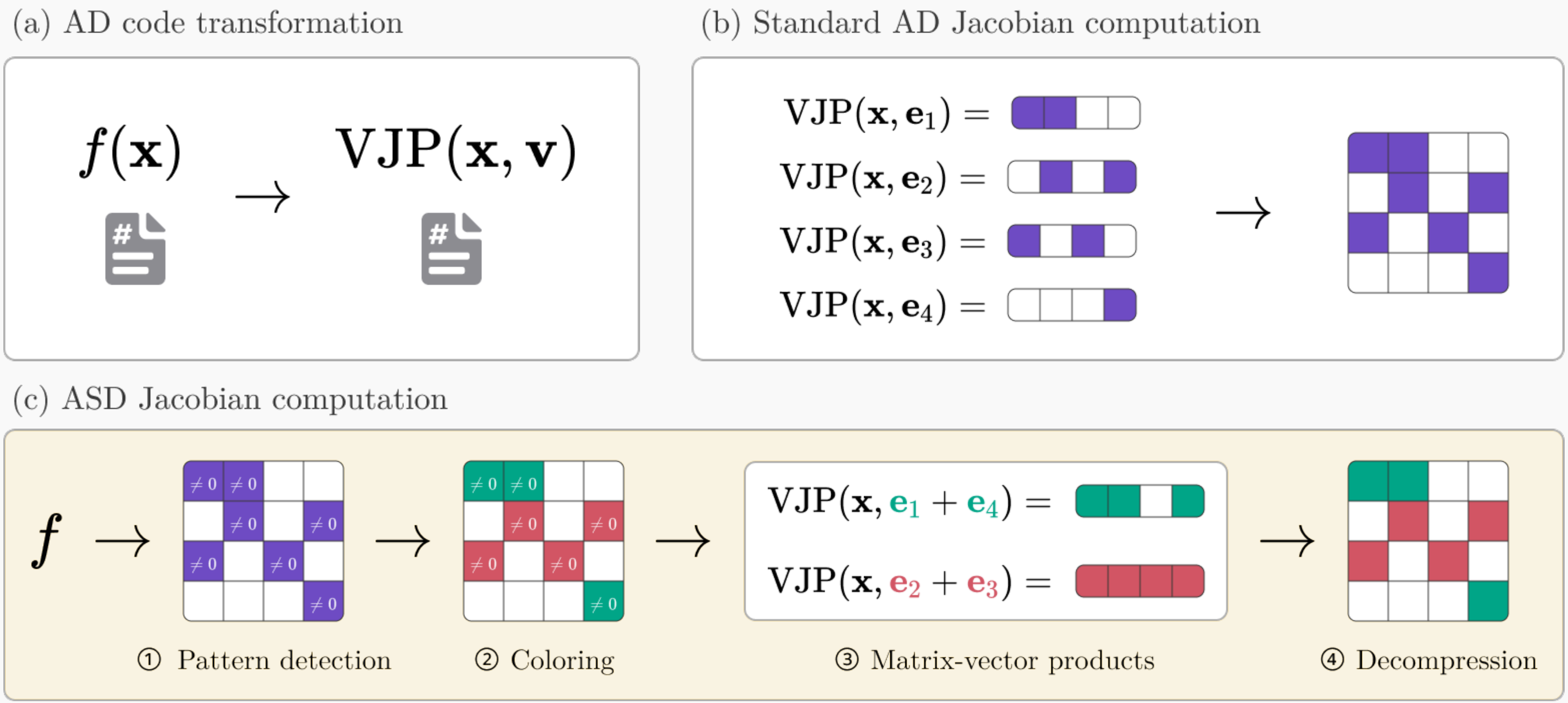


Figure 4: Hill & Dalle (2025)

Two preliminary steps

When grouping columns, we want to

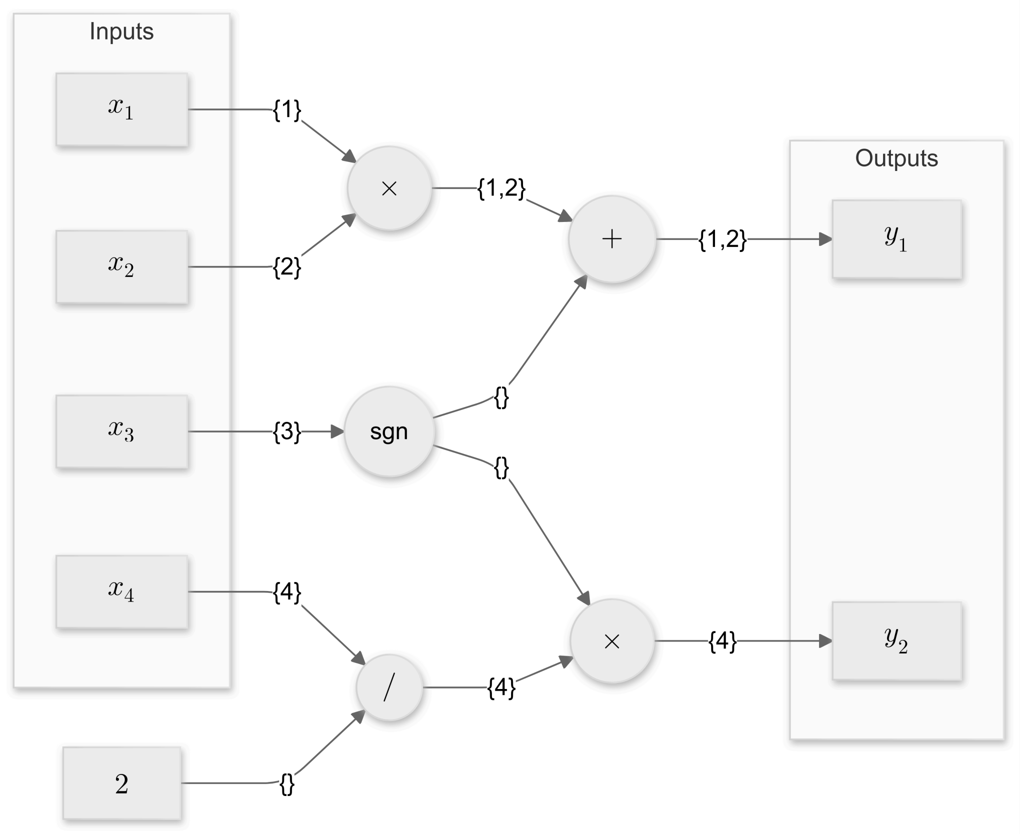
- guarantee structural orthogonality (correctness)
- form the smallest number of groups (efficiency)

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

The preparation phase can be amortized across several inputs.

Pattern detection and coloring

Tracing dependencies in the computation graph



Computation graph for

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

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[4].

[4]Gebremedhin et al. (2005)

Graph representations of a matrix

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

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

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

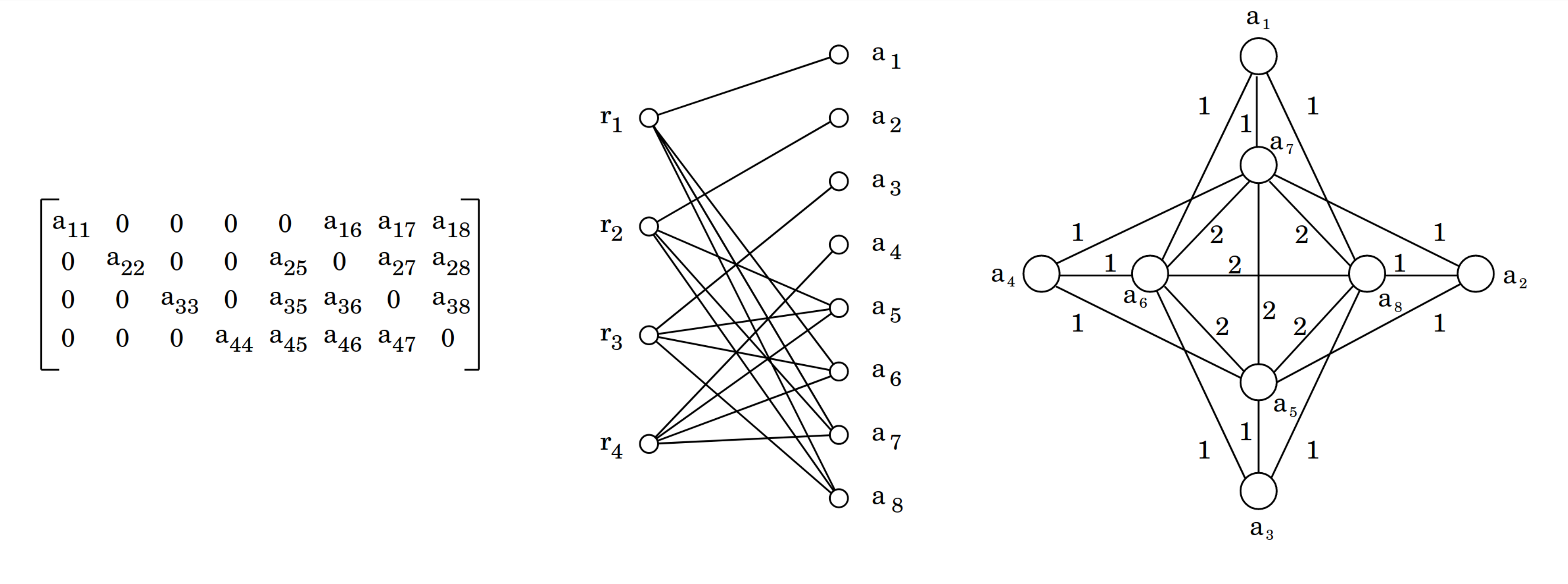


Figure 5: Gebremedhin et al. (2005)

Jacobian coloring

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

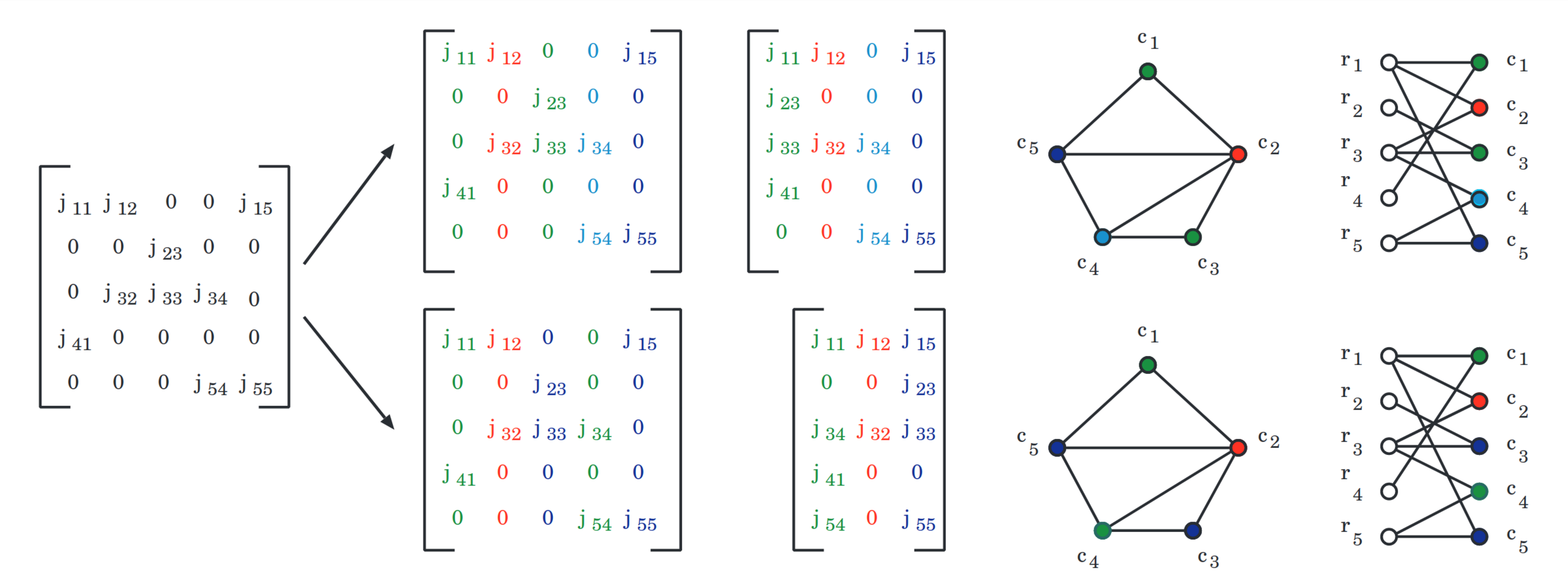


Figure 6: Gebremedhin et al. (2005)

Star coloring of adjacency graph

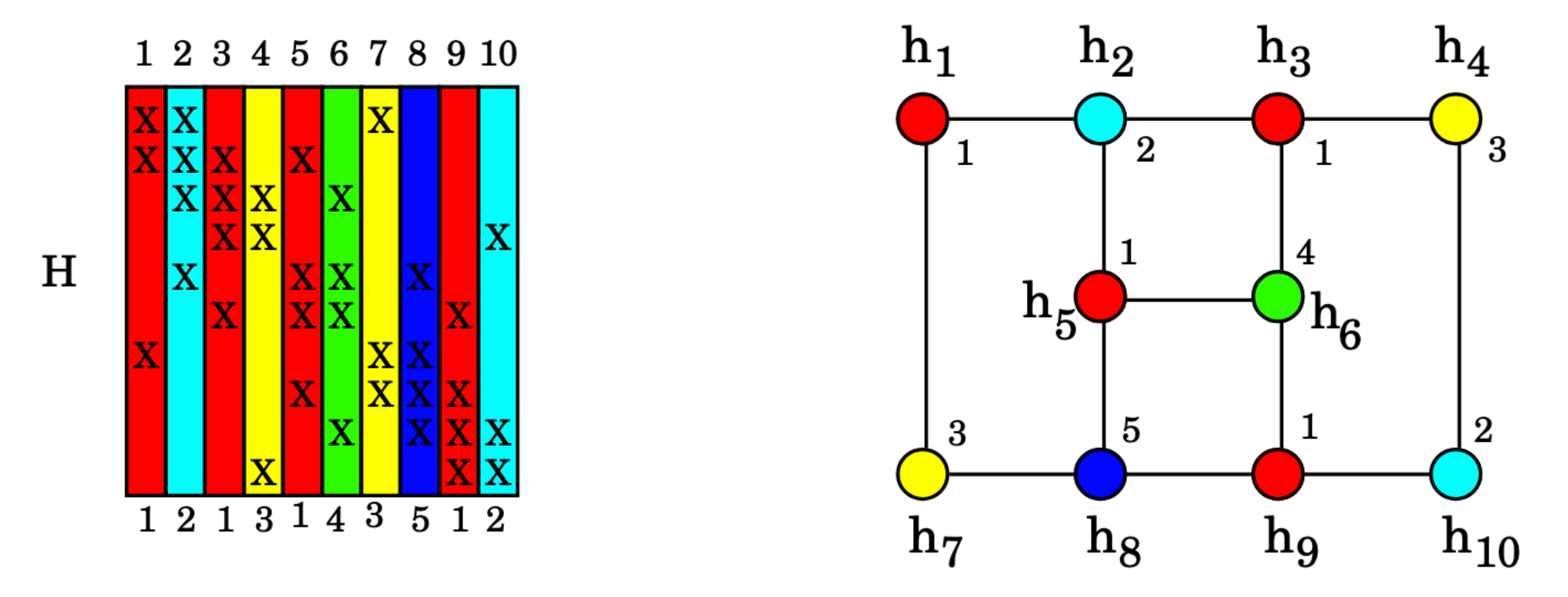


Figure 7: Gebremedhin et al. (2009)

Hessian coloring

Why a "star" coloring[5]? 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 $\Longleftrightarrow$ any 2-colored subgraph is a collection of disjoint stars (it contains no path longer than 3).

[5]Coleman & Moré (1984)

Jacobian bicoloring

Bidirectional coloring of bipartite graph, with neutral color

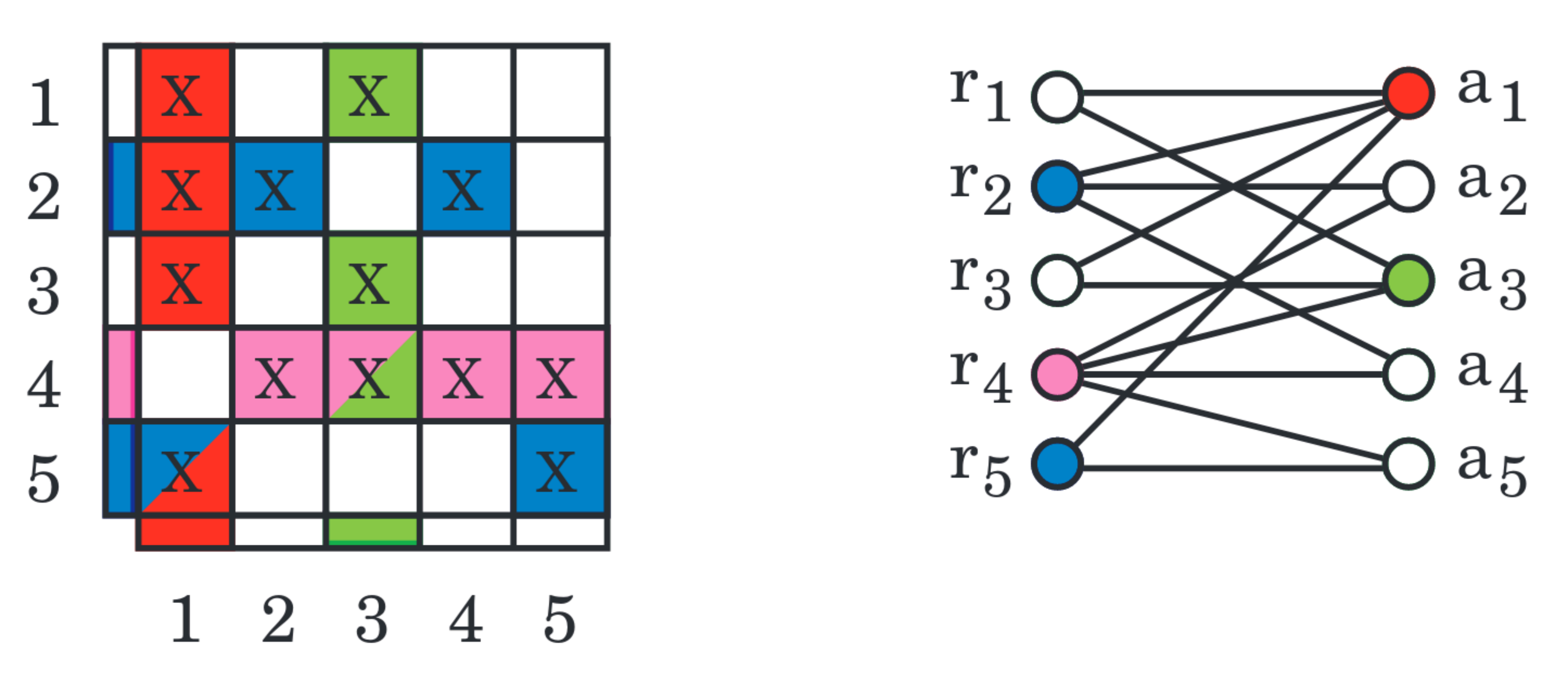


Figure 8: Gebremedhin et al. (2005)

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

The sharp bits

Pattern detection

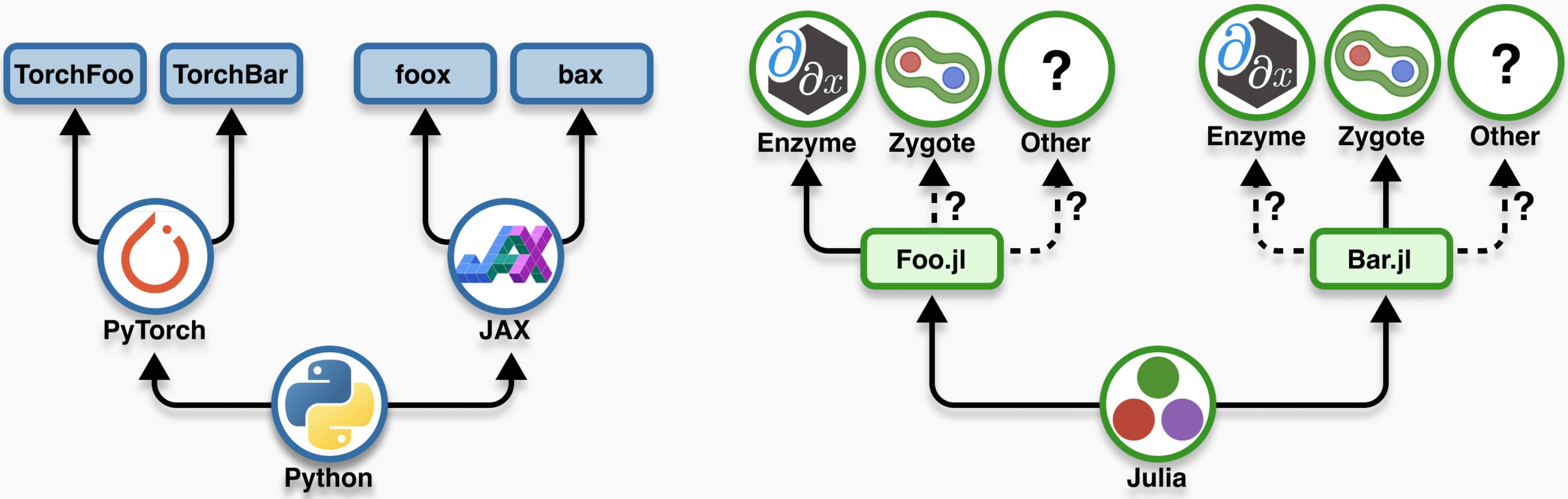
- Linear versus nonlinear interactions
- Local versus global sparsity

Coloring

- Only heuristic algorithms
- Vertex ordering matters a lot

Implementation

AD in Python & Julia



AD in Python & Julia

Step 1: select AD framework

- PyTorch
- JAX
- Tensorflow

Constraints

Step 2: write code

Step 1: write code

Step 2: make code compatible with AD backends

- ForwardDiff
- Zygote
- ...

Constraints

Interfaces for experimenting [new]

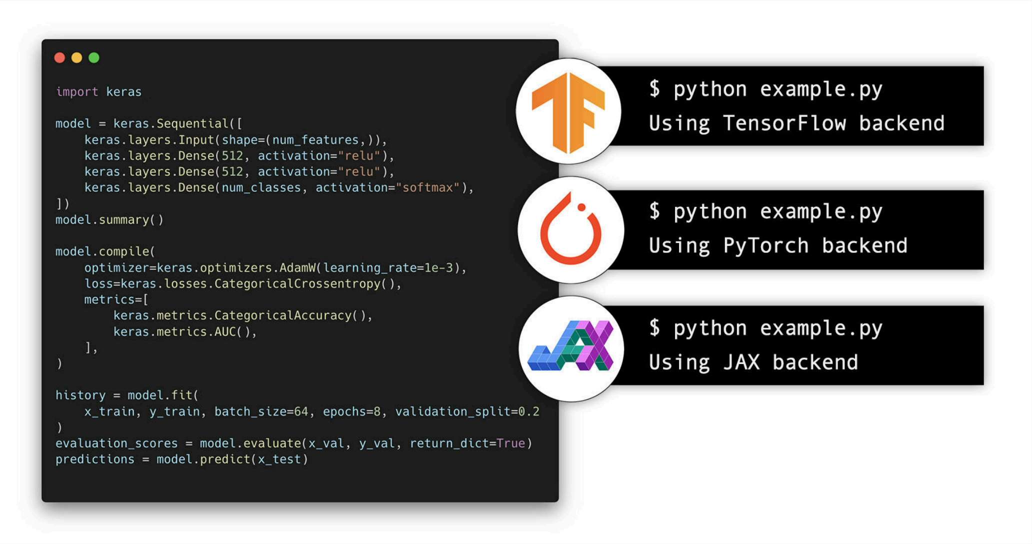


Figure 11: In Python, `Keras` supports Tensorflow, PyTorch and JAX.

Figure 12: In Julia, 14 AD backends inside `Differentiationterface.jl`

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

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`
- Steps 2 & 4: `SparseMatrixColorings.jl`
- Step 3: `Differentiationterface.jl`

	`SCT.jl`	`SMC.jl`	`DI.jl`
lines of code	4861	4242	16971
indirect dependents	420	437	426
downloads / month	4.2k	16k	20k

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.

Conclusion

Perspectives

- GPU-compatible pattern detection and coloring
- Adaptation 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)
- **Hill & Dalle (2025)**

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

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., & 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

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

Dagréou, M., Ablin, P., Vaiter, S., & Moreau, T. (2024, February). How to compute Hessian-vector products?. *The Third Blogpost Track at ICLR 2024*. https://openreview.net/forum?id=rTgjQtGP3O

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, January). *Sparser, Better, Faster, Stronger: Efficient Automatic Differentiation for Sparse Jacobians and Hessians*. arXiv. https://doi.org/10.48550/arXiv.2501.17737

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