

Sparser, better, faster, stronger

Automatic differentiation with a lot of zeros

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

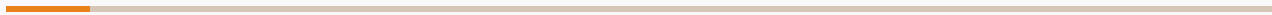
Laboratoire Jean Kuntzmann, 19.06.2025

*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

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

Optimization

Solve $\min_x f(x)$ by iterating

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

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:

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

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

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

Differentiation

Given $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$, its **differential** $\partial f(x)$ is the **linear map** which best approximates f around x :

$$f(x + u) = f(x) + \partial f(x)[u] + o(u)$$

It can be represented by the **Jacobian matrix**, which I will denote by $J = \partial_{\text{mat}} f(x)$ instead.

Numeric differentiation

input	output
program computing the function $x \mapsto f(x)$	approximation of the differential with the same program $\partial f(x)[u] \approx \frac{f(x + \varepsilon u) - f(x)}{\varepsilon}$

Automatic / algorithmic differentiation

input	output
program computing the function $x \mapsto f(x)$	new program computing the exact differential $(x, u) \mapsto \partial f(x)[u] \text{ or } \partial f(x)^*[u]$

AD under the hood

Two ingredients only:

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

Composition

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

$$\text{standard} \quad \partial f(x) = \partial g(h(x)) \circ \partial h(x)$$

$$\text{adjoint} \quad \partial f(x)^* = \partial h(x)^* \circ \partial g(h(x))^*$$

These linear maps apply as follows:

$$\text{forward} \quad \partial f(x) : U \xrightarrow{\partial h(x)} V \xrightarrow{\partial g(h(x))} W$$

$$\text{reverse} \quad \partial f(x)^* : U \xleftarrow{\partial h(x)^*} V \xleftarrow{\partial g(h(x))^*} W$$

Why linear maps?

The chain rule has a matrix equivalent:

$$\partial_{\text{mat}}(g \circ h)(x) = \partial_{\text{mat}}g(h(x)) \cdot \partial_{\text{mat}}h(x)$$

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

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

Essential for neural networks (scalar output but vector encodings)!

Two modes

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

$$u \mapsto \partial f(x)[u] = Ju \quad \text{with} \quad J = \partial_{\text{mat}} f(x)$$

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

$$v \mapsto \partial f(x)^*[v] = J^T v = v^T J \quad \text{with} \quad J = \partial_{\text{mat}} f(x)$$

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

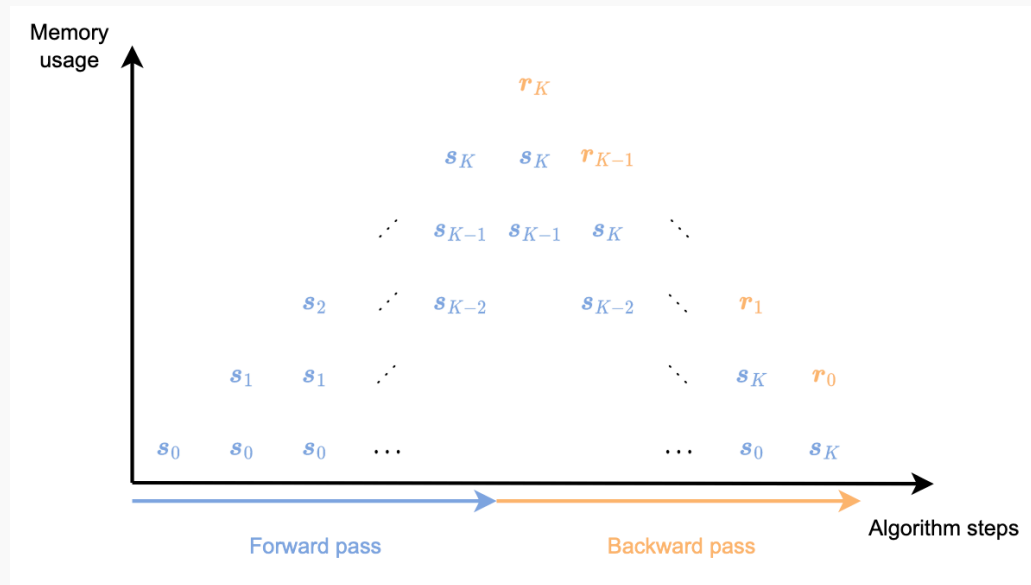
What about gradients?

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.



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**²:

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

²Pearlmutter (1994)

Pocket AD

```
# Basic rules
```

```
using LinearAlgebra
```

```
A, b = rand(2, 3), rand(2)
```

```
residuals(x) = A * x - b
```

```
∂(::typeof(residuals)) = x → (u → A * u) #  $\mathbb{R}^3 \rightarrow \mathbb{R}^2$ 
```

```
∂T(::typeof(residuals)) = x → (v → adjoint(A) * v) #  $\mathbb{R}^2 \rightarrow \mathbb{R}^3$ 
```

```
sqnorm(r) = sum(abs2, r)
```

```
∂(::typeof(sqnorm)) = r → (v → dot(2r, v)) #  $\mathbb{R}^2 \rightarrow \mathbb{R}$ 
```

```
∂T(::typeof(sqnorm)) = r → (w → 2r .* w) #  $\mathbb{R} \rightarrow \mathbb{R}^2$ 
```

Pocket AD

```
# Composition
```

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

```
function ∂⊤(f::ComposedFunction)
    g, h = f.outer, f.inner
    return x → ∂⊤(h)(x) ∘ ∂⊤(g)(h(x))
end
```

Pocket AD

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

```
julia> f = sqnorm ∘ residuals;
```

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

```
julia> ∂(f)(x)(u) # directional derivative  
0.8691056836969242
```

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

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

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

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 VJPs (output dimension)

Using fewer products

When the Jacobian is sparse, we can compute it faster³.

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

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

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

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

³Curtis et al. (1974)

Two preliminary steps

When grouping columns, we want to

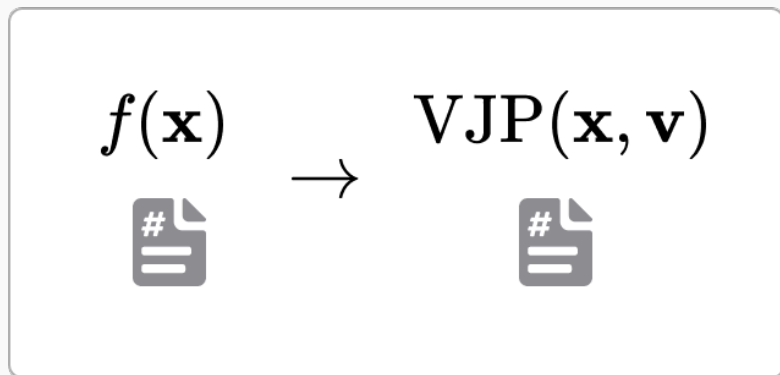
- guarantee orthogonality (correctness) \Rightarrow pattern detection
- form the smallest number of groups (efficiency) \Rightarrow coloring

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

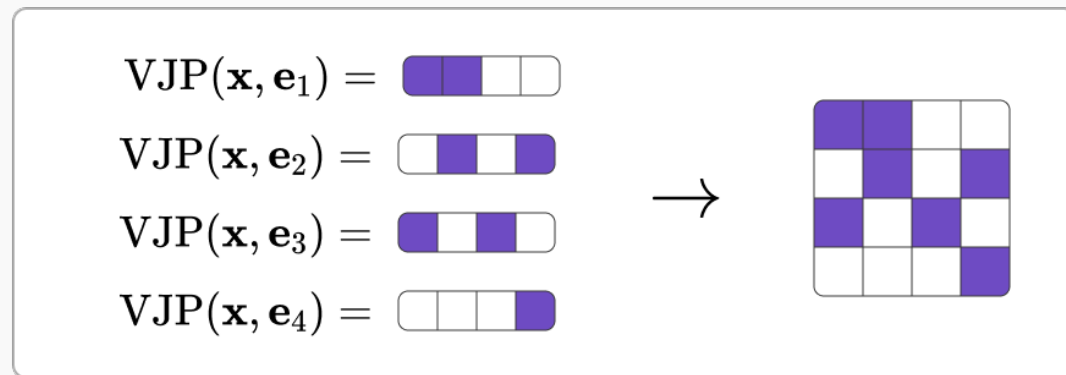
The preparation phase can be **amortized** across several inputs.

The gist in one slide

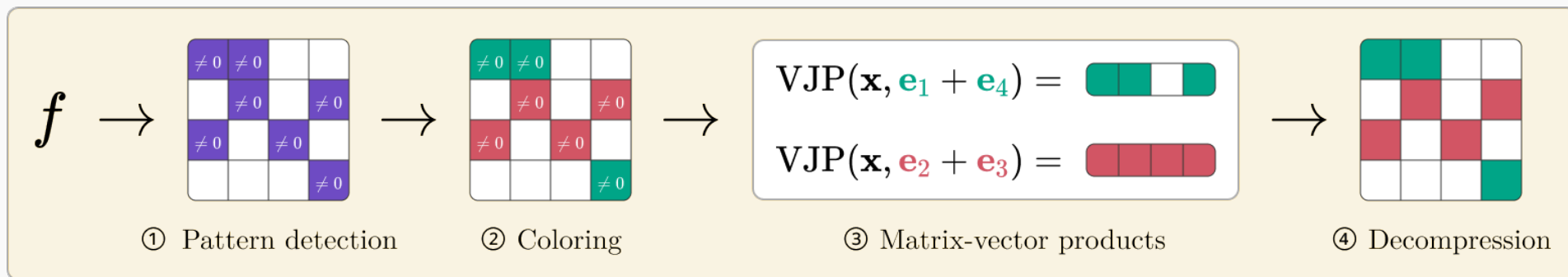
(a) AD code transformation



(b) Standard AD Jacobian computation

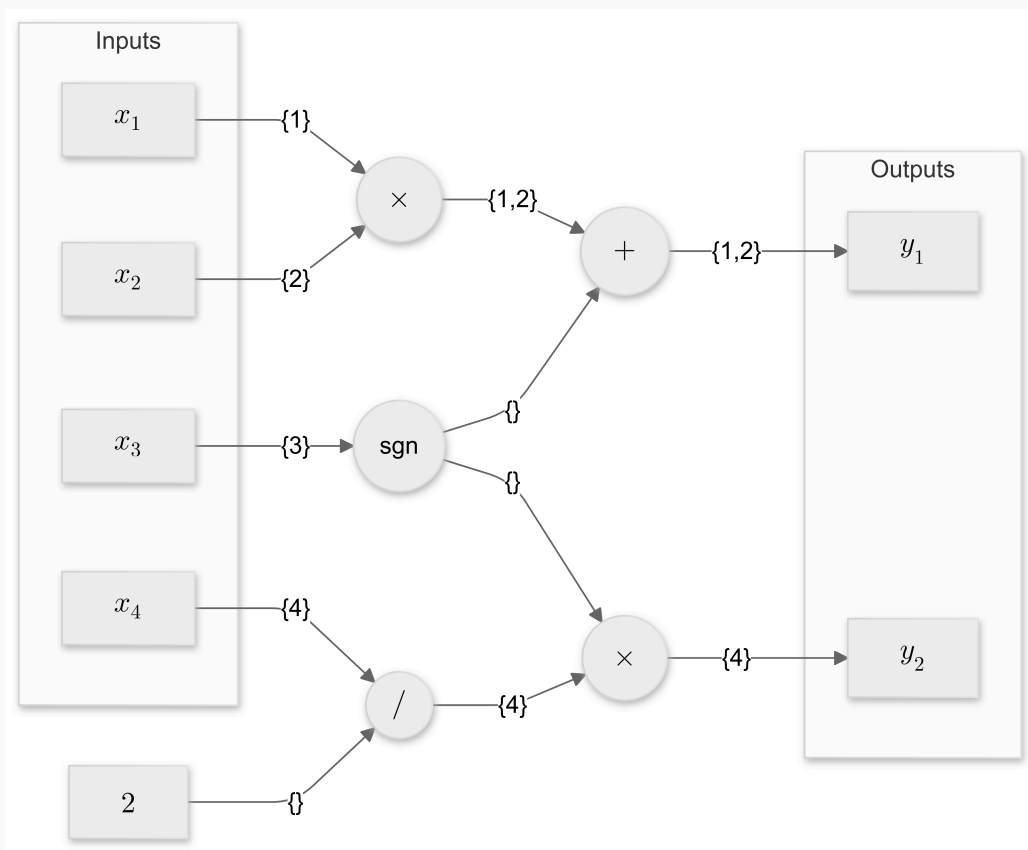


(c) ASD Jacobian computation



Hill & Dalle (2025)

Tracing dependencies in the computation graph



Hill et al. (2025)

Computation graph for

$$y_1 = x_1 x_2 + \text{sign}(x_3)$$

$$y_2 = \text{sign}(x_3) \times \left(\frac{x_4}{2} \right)$$

Its Jacobian matrix will have 3 nonzero coefficients:

$$\begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Pocket pattern detection

```
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 :: Real) = Tracer(a.indices)
```

```
sign(a :: Tracer) = Tracer() # zero derivatives
```

Pocket pattern detection

```
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]))
```

Coloring for Jacobians

Matrix problem

Orthogonal partition of the columns of A .

If $A_{ij_1} \neq 0$ and $A_{i,j_2} \neq 0$, then columns j_1 and j_2 are in different groups $c(j_1) \neq c(j_2)$

These are equivalent⁴ if we define the graph representation

$$\mathcal{E} = \{(i, j) \in \mathcal{I} \times \mathcal{J} : A_{i,j} \neq 0\}$$

Graph problem

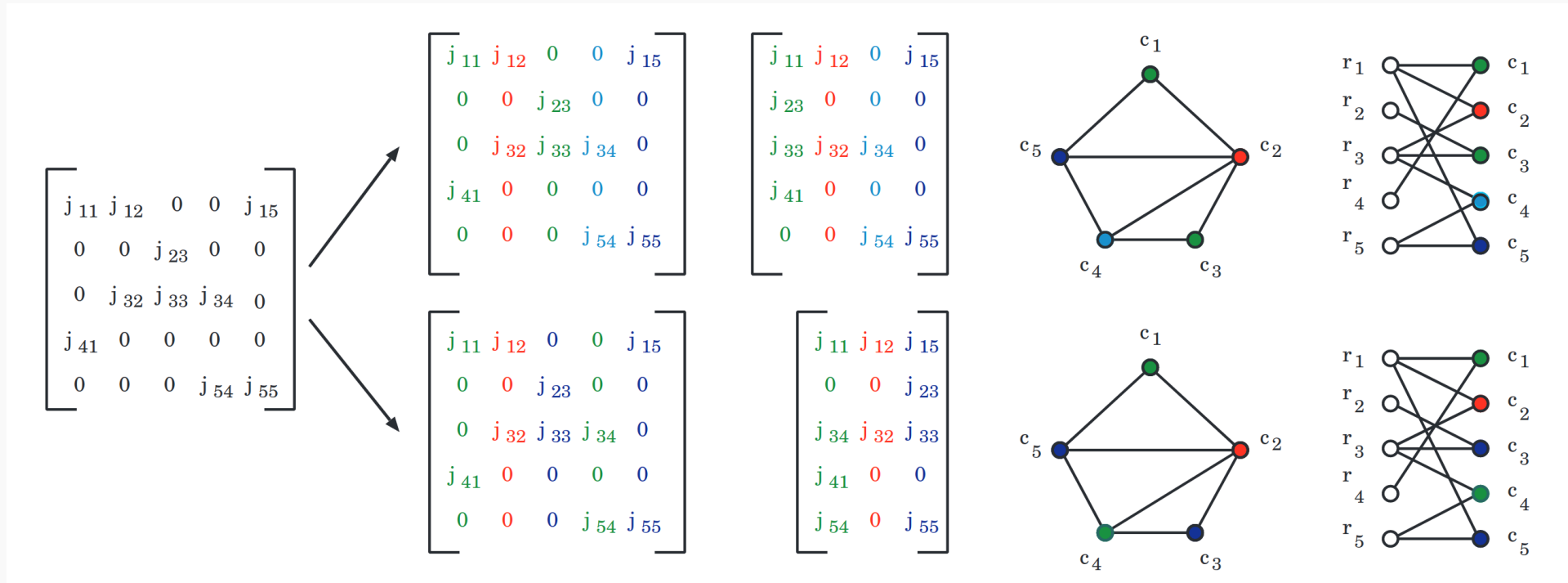
Partial distance-2 coloring of a bipartite graph $\mathcal{G} = (\mathcal{I} \cup \mathcal{J}, \mathcal{E})$

If $(i, j_1) \in \mathcal{E}$ and $(i, j_2) \in \mathcal{E}$, then vertices j_1 and j_2 have different colors $c(j_1) \neq c(j_2)$.

⁴Gebremedhin et al. (2005)

Coloring for Jacobians, illustrated

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

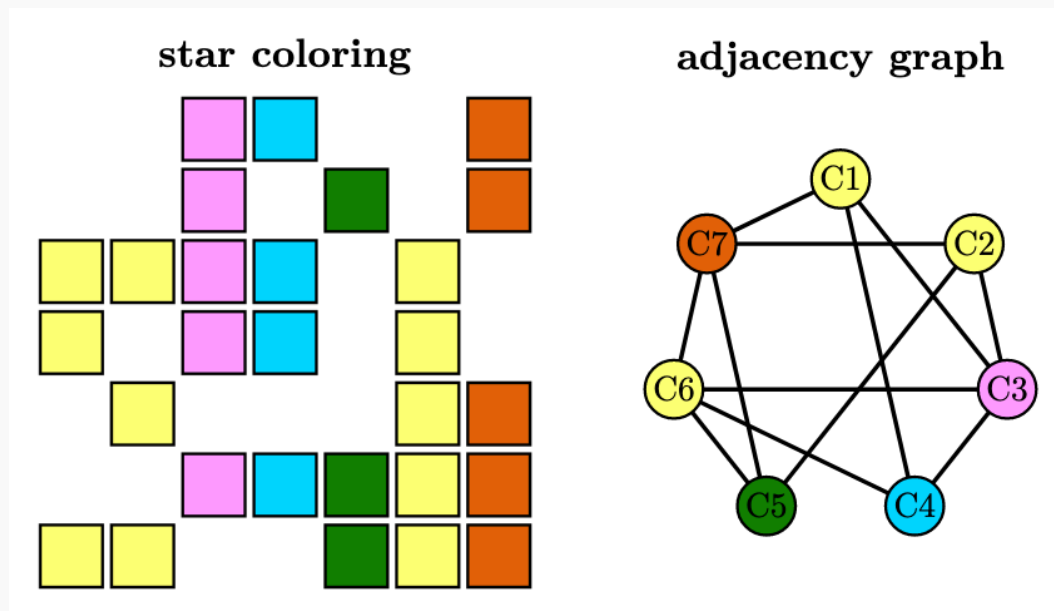


Gebremedhin et al. (2005)

Coloring for Hessians

What if our matrix has structure, like $A_{i,j} = A_{j,i}$?

We can compute a slightly different coloring⁵ with fewer colors.

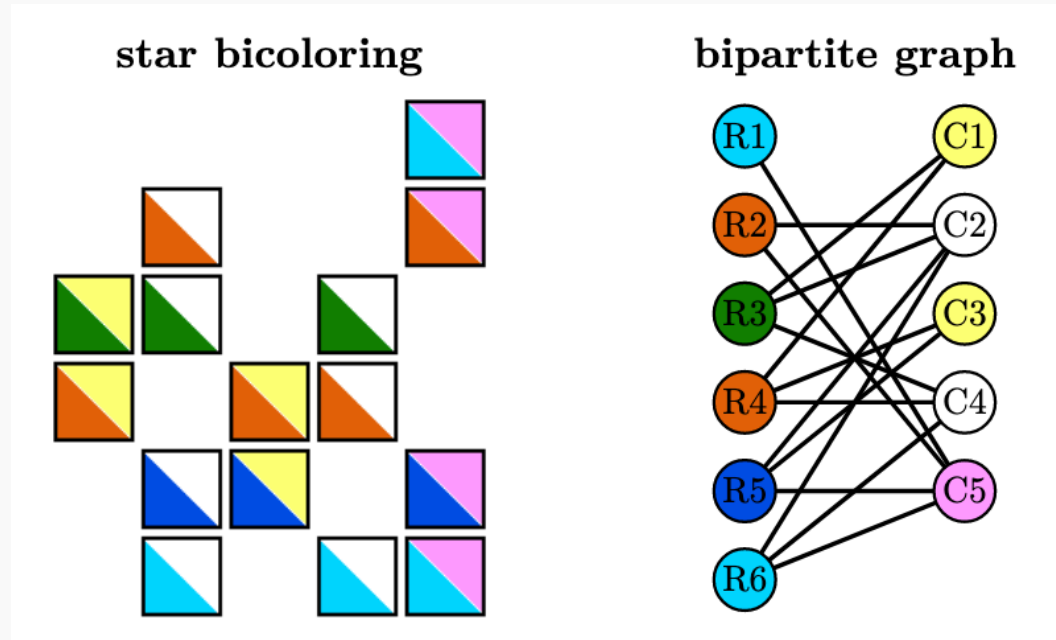


⁵Coleman & Moré (1984)

Coloring for bidirectional Jacobians

What if the columns are not orthogonal enough?

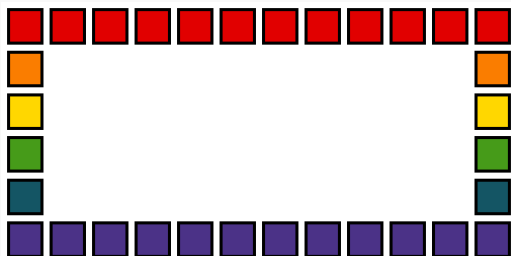
We can use both rows and columns⁶ inside our coloring.



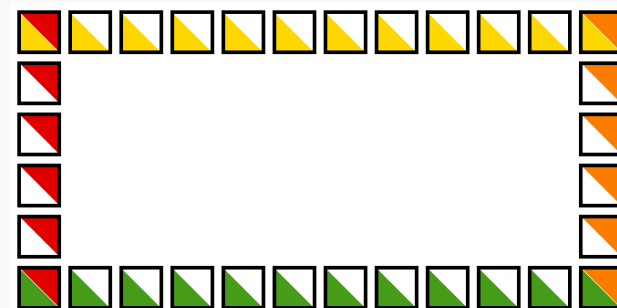
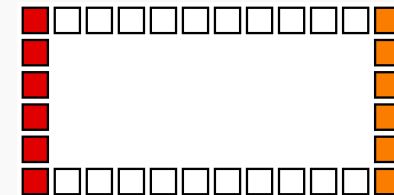
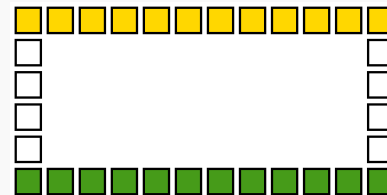
⁶Coleman & Verma (1998)

Benefits of bidirectional coloring

Compute Jacobians with a dense row **and** a dense column, using forward-mode AD + reverse-mode AD.



Unidirectional



Bidirectional

From bidirectional to symmetric [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

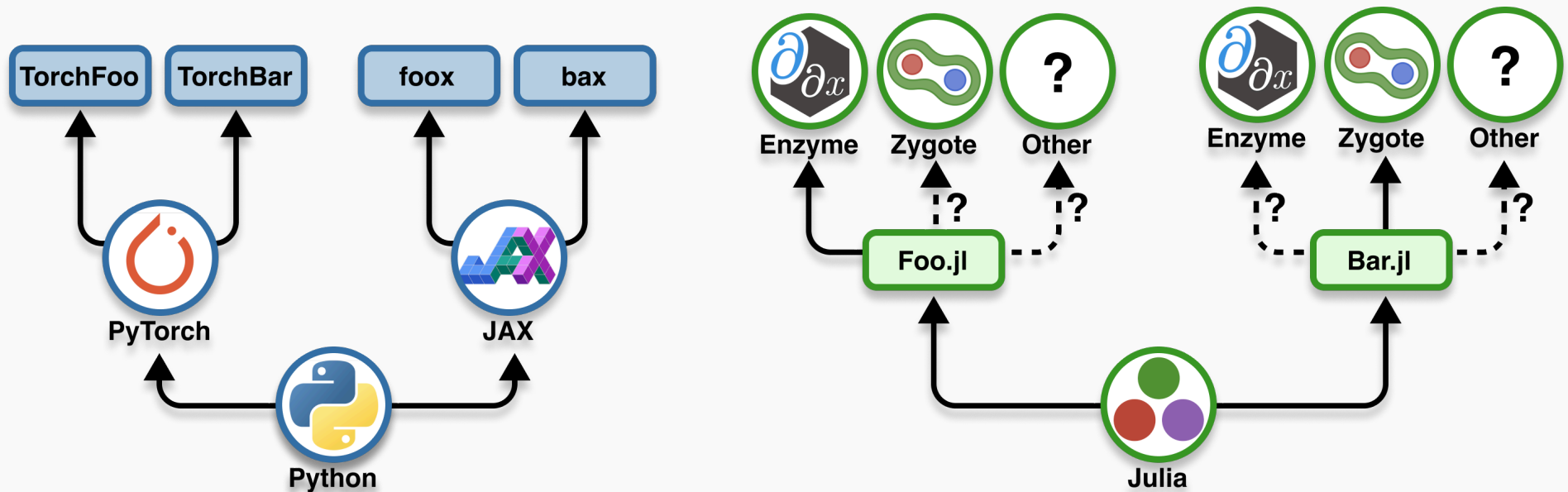
- Local versus global sparsity
- Control flow
- Linear and nonlinear interactions

Coloring

- Only heuristic algorithms
- Vertex ordering matters a lot

Implementation

AD in Python & Julia



Interfaces for experimenting [new]

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

```
import keras

model = keras.Sequential([
    keras.layers.Input(shape=(num_features,)),
    keras.layers.Dense(512, activation="relu"),
    keras.layers.Dense(512, activation="relu"),
    keras.layers.Dense(num_classes, activation="softmax"),
])
model.summary()

model.compile(
    optimizer=keras.optimizers.AdamW(learning_rate=1e-3),
    loss=keras.losses.CategoricalCrossentropy(),
    metrics=[
        keras.metrics.CategoricalAccuracy(),
        keras.metrics.AUC(),
    ],
)

history = model.fit(
    x_train, y_train, batch_size=64, epochs=8, validation_split=0.2
)
evaluation_scores = model.evaluate(x_val, y_val, return_dict=True)
predictions = model.predict(x_test)
```



\$ python example.py
Using TensorFlow backend



\$ python example.py
Using PyTorch backend



\$ python example.py
Using JAX backend

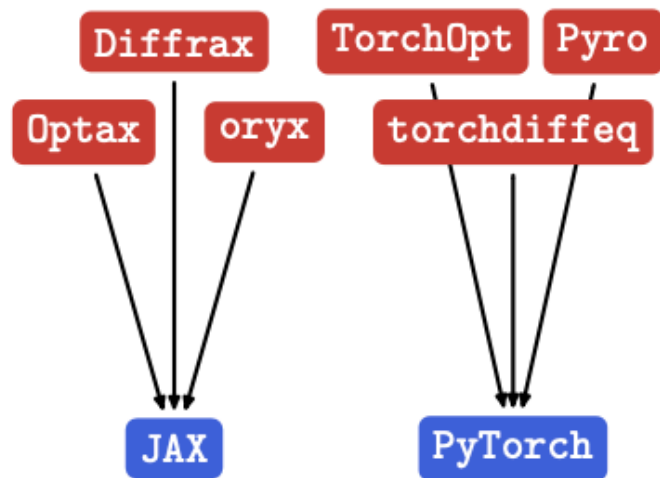


In Python, Keras supports
Tensorflow, PyTorch and JAX.

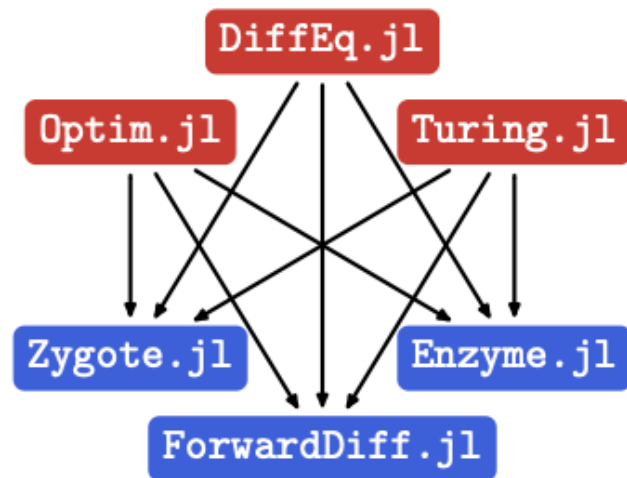
In Julia, 14 AD backends inside
DifferentiationInterface.jl

One API to rule them all

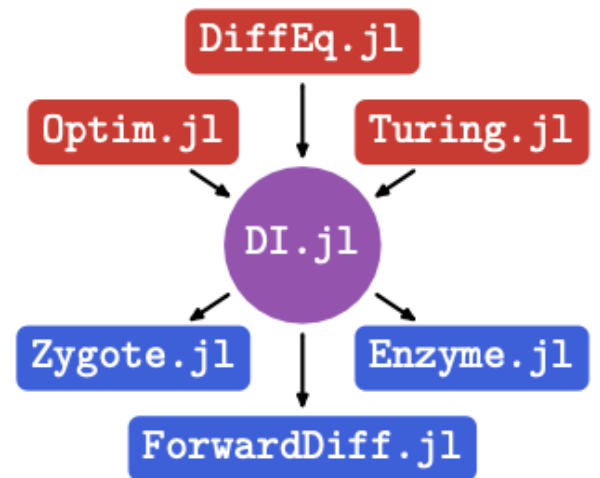
Decouple the scientific libraries from the AD underneath.



(a) Python



(b) Julia (before DI)



(c) Julia (now)

Dalle & Hill (2025)

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

Python bindings in construction:

- `pysparsematrixcolorings`
- `sparsediffax`

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)
- Hill et al. (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>
- 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>
- 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, May). *A Common Interface for Automatic Differentiation*. arXiv. <https://doi.org/10.48550/arXiv.2505.05542>
- 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>
- Hill, A., Dalle, G., & Montoison, A. (2025, April). An Illustrated Guide to Automatic Sparse Differentiation. *ICLR Blogposts 2025*. <https://iclr-blogposts.github.io/2025/blog/sparse-autodiff/>

- 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>
- Montoisson, A., Dalle, G., & Gebremedhin, A. (2025, May). *Revisiting Sparse Matrix Coloring and Bicoloring*. arXiv. <https://doi.org/10.48550/arXiv.2505.07308>
- Pearlmutter, B. A. (1994). Fast Exact Multiplication by the Hessian. *Neural Computation*, 6(1), 147–160. <https://doi.org/10.1162/neco.1994.6.1.147>