Experiments with Vector-Valued Nonlinear Functions in JuMP

Benoît Legat, Sophie Lequeu

SIAM Conference on Optimization (OP26), June 5th 2026

What is Automatic Differentiation (AD)?

When **computing a derivative**, you can:

- Compute it by hand,
- Use Symbolic differentiation,
- Use Finite Differences,
- Use Automatic Differentiation

In short:

1. Decompose your function into elementary operations,

$$f(x) = (f_n \circ f_{n-1} \circ \ldots \circ f_2 \circ f_1)(x)$$

2. Apply the Chain Rule automatically on this sequence of elementary operations.

$$J_f(x) = J_{f_n}(s_{n-1}) \cdot J_{f_{n-1}}(s_{n-2}) \cdot \ldots \cdot J_{f_2}(s_1) \cdot J_{f_1}(s_0)$$

There is Forward and Reverse AD. We focus on Reverse AD.

What is Automatic Differentiation (AD)?

When **computing a derivative**, you can:

- Compute it by hand,
- Use Symbolic differentiation,
- Use Finite Differences,
- Use Automatic Differentiation

In short:

1. Decompose your function into elementary operations,

$$f(x) = (f_n \circ f_{n-1} \circ \ldots \circ f_2 \circ f_1)(x)$$

2. Apply the Chain Rule automatically on this sequence of elementary operations.

$$\mathrm{J}_f(x)\,\mathbf{u} = \mathrm{J}_{f_n}(s_{n-1}) \cdot \mathrm{J}_{f_{n-1}}(s_{n-2}) \cdot \ldots \cdot \mathrm{J}_{f_2}(s_1) \cdot \mathrm{J}_{f_1}(s_0)\,\mathbf{u}$$

There is Forward and Reverse AD. We focus on Reverse AD.

What is Automatic Differentiation (AD)?

When **computing a derivative**, you can:

- Compute it by hand,
- Use Symbolic differentiation,
- Use Finite Differences,
- Use Automatic Differentiation

In short:

1. Decompose your function into elementary operations,

$$f(x) = (f_n \circ f_{n-1} \circ \ldots \circ f_2 \circ f_1)(x)$$

2. Apply the Chain Rule automatically on this sequence of elementary operations.

$$\mathbf{v}^\top \mathrm{J}_f(x) = \mathbf{v}^\top \mathrm{J}_{f_n}(s_{n-1}) \cdot \mathrm{J}_{f_{n-1}}(s_{n-2}) \cdot \ldots \cdot \mathrm{J}_{f_2}(s_1) \cdot \mathrm{J}_{f_1}(s_0)$$

There is Forward and Reverse AD. We focus on Reverse AD.

How does it work?

1. Construct the ***expression graph***,

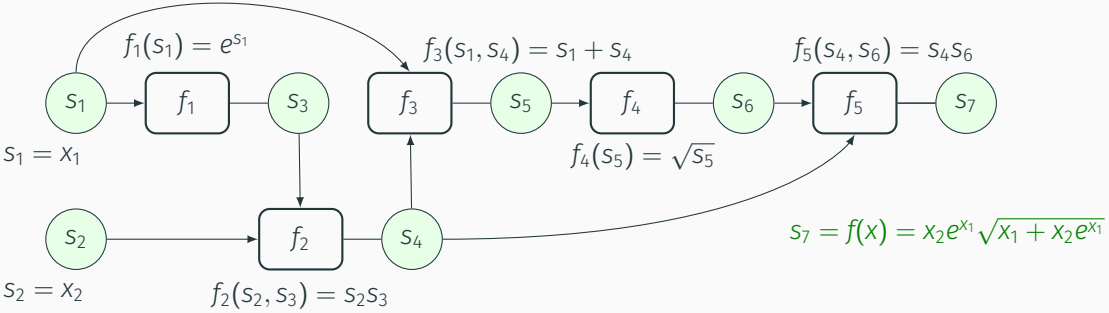


Figure 1: Expression graph of function $f(x) = x_2 e^{x_1} \sqrt{x_1 + x_2 e^{x_1}}$.

How does it work?

1. Construct the ***expression graph***,
2. Propagate **forward** the intermediate quantities,
3. Propagate **backward**, multiplying the local Jacobians.

$$\mathbf{v}^\top \mathrm{J}_f(x) = \mathbf{v}^\top \mathrm{J}_{f_n}(s_{n-1}) \cdot \mathrm{J}_{f_{n-1}}(s_{n-2}) \cdot \ldots \cdot \mathrm{J}_{f_2}(s_1) \cdot \mathrm{J}_{f_1}(s_0)$$

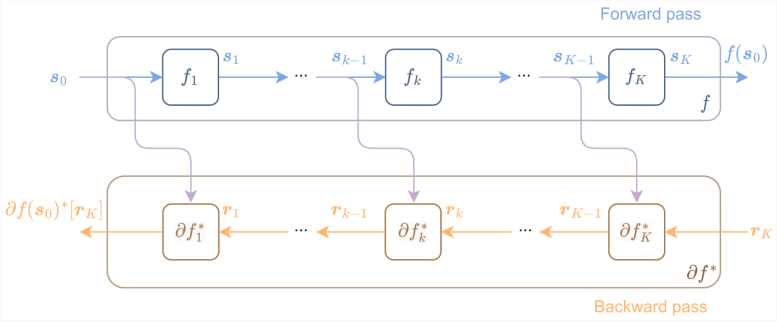
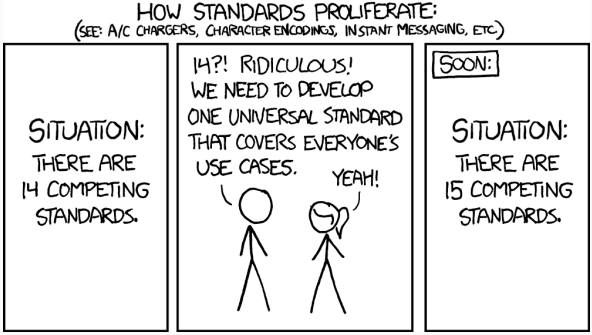


Figure 2: from Blondel and Roulet 2025

What is already available in Julia?

- **Mooncake**, **Zygote**: Julia compiler level
- **Enzyme**: LLVM level
- **Reactant**: MLIR
- ...
- JuMP's **ReverseAD**



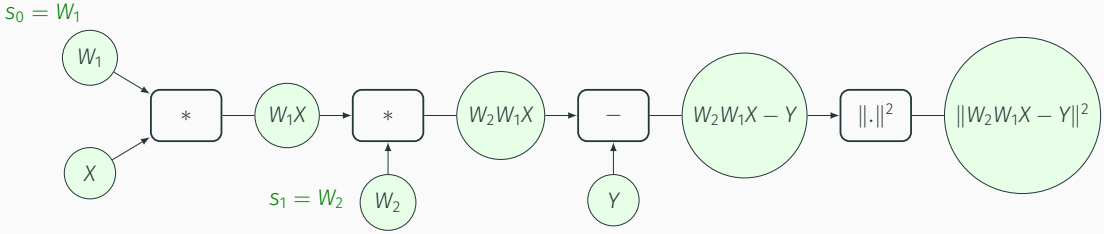
JuMP is using its own AD system (ReverseAD), but it is **only scalar**. For functions with vectors or matrices, vectorization could allow faster computations!

→ Goal with ArrayDiff: support **vector** and **matrix** variables and functions.

What is different from the scalar case?

→ Nodes in the expression graph do not contain only scalar values anymore, but **arrays**.

Example for: $f(W_1, W_2) = \|W_2 W_1 X - Y\|^2$,



In the implementation of ReverseAD, values are stored in the tape.
Here, since values in the nodes can have **arbitrary dimensions**, we **also** need to store the **shapes** of the values in the nodes.

What is different from the scalar case?

$\rightarrow$ We will not compute, store and multiply all the local Jacobians on the backward pass anymore.

Example for a product of matrices: $Y = AX, \quad A \in \mathbb{R}^{m \times n}, \quad X \in \mathbb{R}^{n \times p}$

Jacobians as matrices:

$$\text{vec}(Y) = (I_p \otimes A)\ \text{vec}(X)$$

Local Jacobian:

$$J = I_p \otimes A$$

Potentially huge, structured matrix

Jacobians as linear maps:

$$Y = AX$$

On the backward pass:

$$\frac{\partial f}{\partial X} = A^\top \frac{\partial f}{\partial Y}$$

Apply local Jacobian as linear maps

Details on implementation

Goal: avoid runtime dispatch as much as possible.

Common shapes and operators are handled by explicit *if*/*elseif* branches.

→ Actually similar to what is done in MOI.Nonlinear.ReverseAD or by Julia compiler for small unions[1].

- **Shape dispatch:** if-else for scalars, vectors, and matrices.

$$scalar \;/\; vector \;/\; matrix$$

- **Operator dispatch:** if-else for a predefined list of operators.

$$+, \quad *, \quad dot, \quad norm, \quad sum, \quad \ldots$$

- **Fallback:** dynamic dispatch for higher-order tensors or user-defined functions.

[1] *https://julialang.org/blog/2018/08/union-splitting/*

Integration with existing systems

- User-defined **array** functions with *ChainRules*

→ For arbitrary Julia code with vector or matrix input, JuMP will use ChainRules for this specific part of your function.

- Embedding a **Neural Network** in JuMP via **MathOptAI**

→ ArrayDiff allows to differentiate your NN with respect to inputs x, given your trained NN imported with ONNX.

A small benchmark and comparisons

Benchmark on function $f(W_1, W_2) = \|W_2 \tanh(W_1 X) - Y\|^2$

	Mooncake + Lux	PyTorch	ArrayDiff
Min	656.703 µs	293.197 µs	226.312 µs
Mean ± σ	827.964 µs ± 1.821 ms	882.135 µs ± 162.453 µs	266.470 µs ± 67.781 µs

On NVIDIA RTX PRO 1000 Blackwell Generation Laptop GPU

Optimizing on GPU:

- Make it allocation-free on CPU
- Get rid of *scalar indexing* errors
- Use dot (.) syntax → we do not depend on CUDA.jl or KernalAbstraction.jl!

Deep learning with JuMP ?

```
model = GenericModel{Float32}(solver)
@variable(model, W1[1:h, 1:d],
    container = ArrayDiff.ArrayOfVariables)
@variable(model, W2[1:out_dim, 1:h],
    container = ArrayDiff.ArrayOfVariables)
Y_hat = W2 * tanh.(W1 * X)
loss = sum((Y_hat - Y) .^ 2)
@objective(model, Min, loss)
```

Deep learning with JuMP ?

CPU / Float64 with $n_{\text{out}} = 2, n_{\text{hidden}} = 4096, n_{\text{in}} = 13, n_{\text{data}} = 178$

NLopt with LBFGS

```
solution_summary(; result = 1, verbose = false)
├ solver_name          : NLopt
└ Termination
  ├ termination_status : LOCALLY_SOLVED
  ├ result_count       : 1
  └ raw_status         : FTOL_REACHED
└ Solution (result = 1)
  ├ primal_status         : FEASIBLE_POINT
  ├ dual_status           : NO_SOLUTION
  ├ objective_value       : 9.34980e-14
  └ dual_objective_value : 0.00000e+00
└ Work counters
  └ solve_time (sec)    : 4.97072e+00
```

Optimisers.Adam(0.05)

```
solution_summary(; result = 1, verbose = false)
├ solver_name          : NLPModels with OptimisersSolver
└ Termination
  ├ termination_status : LOCALLY_SOLVED
  ├ result_count       : 1
  └ raw_status         : first-order stationary
└ Solution (result = 1)
  ├ primal_status         : FEASIBLE_POINT
  ├ dual_status           : NO_SOLUTION
  ├ objective_value       : 8.09767e-14
  └ dual_objective_value : 0.00000e+00
└ Work counters
  └ solve_time (sec)    : 7.57489e+01
```

- No minibatch/stochastic gradient
- No distributed support / NCCL

Take-home message

Our objective with ArrayDiff:

- Is **not** to differentiate arbitrary Julia code,
- Is **not** to be a competitor to existing AD systems in Julia,

- But to **extend** JuMP's scalar AD ReverseAD to support **vectors and matrices** variables and functions,
- Leading to faster computations for these types of models.
- And supporting **user-defined vector/matrix functions** and **neural networks**.

Future work

Documentation

- Training a neural network/transformer with JuMP
- How to extend ArrayDiff with custom operators, DifferentiationInterface.jl

Second-order

- Sparsity : coloring, decompression using SparseMatrixColorings
- Hessian : Forward over reverse

Merge into JuMP/MOI ?

- ArrayDiff started with MOI.Nonlinear.ReverseAD
- Should be equivalent for scalar expressions
- Could it replace it ?