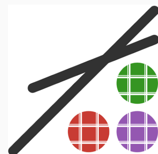
Adding array support for JuMP’s Automatic Differentiation

Sophie Lequeu, Benoît Legat

JuMP-dev 2026, May 31st



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**,

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

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

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

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.

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

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.

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

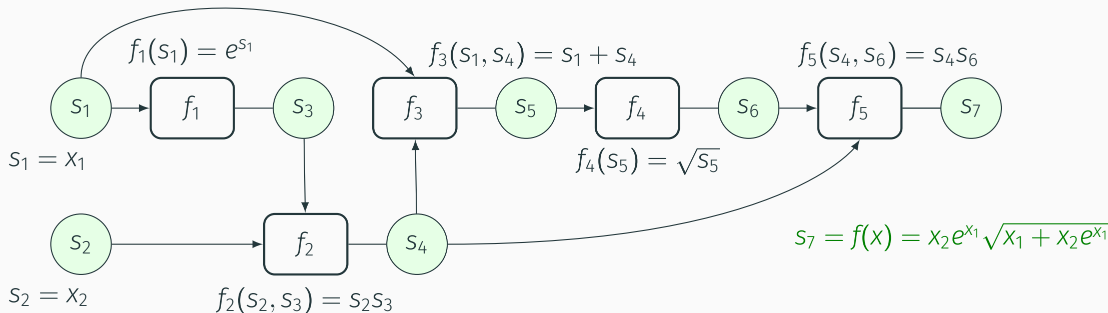


Figure 1: Expression graph of function $f(x) = x_2 e^{x_1} \sqrt{x_1 + x_2 e^{x_1}}$.

How does it work?

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

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

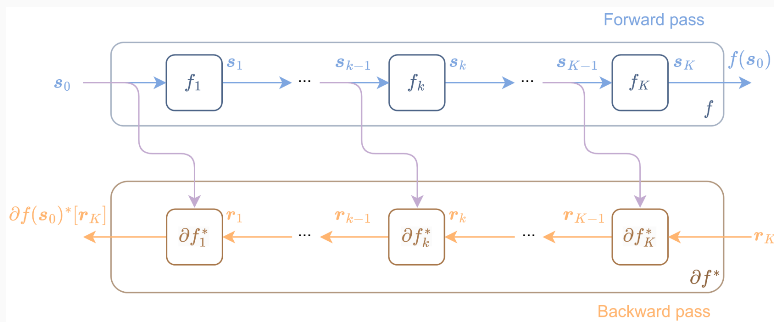
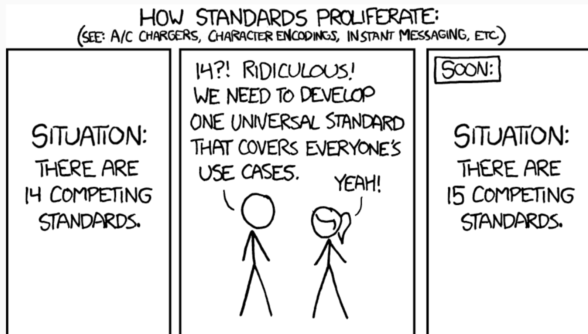


Figure 2: from Blondel and Roulet 2025

What is already available in Julia?

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



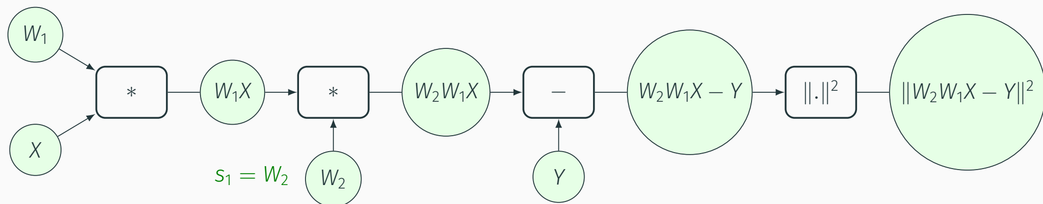
JuMP is using its own AD system (ReverseAD), but it is **only scalar**. For functions with vectors or matrices, vectorization could allow faster computations!

→ 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.

$\rightarrow$ Actually similar to what is done in MOI.Nonlinear.ReverseAD.

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

$$\mathit{scalar} \quad / \quad \mathit{vector} \quad / \quad \mathit{matrix}$$

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

$$+, \quad *, \quad \mathit{dot}, \quad \mathit{norm}, \quad \mathit{sum}, \quad \dots$$

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

Integration with existing systems

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

$\rightarrow$ 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**

$\rightarrow$ 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 $f(W_1, W_2) = \|W_2 \,\mathbf{tanh}\,(W_1 X) - Y\|^2$

	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

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**.