Reverse Mode Automatic Differentiation: Unraveling Expression Graphs & Library Magic

Steve Bronder

October 2023

Who am I

I'm a software engineer here in CCM with a background in C++ and statistics.

- Most of my work revolves automatic differentiation within the Stan language's math library

What are we talking about?

- What's Automatic Differentiation (AD)?
  - Evaluates partial derivatives of a program
- Why should you care?
  - It's important and used a lot
- What's an expression graph?
  - Graphic to describe dependencies for AD
- How is AD implimented
  - Source code is transformed or objects are made for intermediate ops
- What are the tradeoffs between different AD packages
  - Flexibility, Efficiency, and Scale

What's Automatic Differentiation?

Computational technique to evaluate partial derivatives of a program.

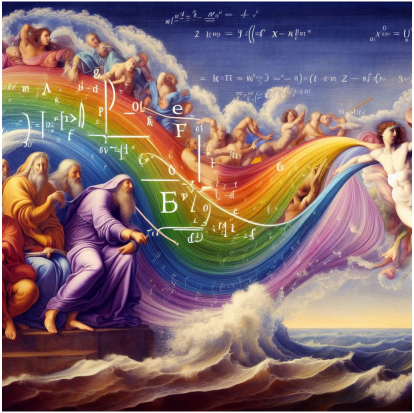


Figure: Asking Dall-E 3 to make a physical representation of automatic differentiation.

Why use Automatic Differentiation?

Many algorithms need derivatives!

$$x_{n+1} = \frac{f(x_n)}{f'(x_n)}$$

Think about HMC, BFGS, SGD, etc.

- Choices
  - Write by hand
  - finite difference,
  - symbolic differentiation
  - spectral differentiation
  - automatic differentiation

Why use Automatic Differentiation?

- Faster than finite difference, more flexible than symbolic differentiation
- Allows for unknown length while and for loops
- Accurate to floating point precision
- Reverse Mode AD can compute partials derivatives of inputs at the same time
- Reverse Mode AD complexity around 4x original function

What's Automatic Differentiation?

Suppose we have a function

$$z = \log(x_0) * x_1 + \sin(x_0)$$

We want to calculate our row vector Jacobian

$$J = \left\{ \frac{\partial z}{\partial x_0}, \frac{\partial z}{\partial x_1} \right\}$$

How to calculate $\frac{\partial z}{\partial x_1}$?

What's Automatic Differentiation?

How to calculate $\frac{\partial z}{\partial x_0}$?

Let v_k be the sequence of intermediate expressions for the input x and output z and let $v_K = z$ and $v_0 = x_0; v_1 = x_1$. Let $\overline{v}_k$ be the partial gradient of the kth intermediate step. Then we can apply the chain rule to each intermediate step to get the partial gradient.

$$z = \log(x_0) * x_1 + \sin(x_0))$$

i.e.

$v_0 = x_0; v_1 = x_1;\ v_2 = \log(v_0); v_3 = sin(v_0); v_4 = v_2 v_1; v_5 = v_4 + v_3$

$$\frac{\partial z_i}{\partial x_j} = \frac{\partial v_K}{\partial v_{K-1}}\overline{v}_K + \cdots + \frac{\partial v_1}{\partial v_0}\overline{v}_1 = \sum_{k=K}^{0} \frac{\partial v_k}{\partial v_{k-1}}\overline{v}_k$$

What's Automatic Differentiation?

- Given a function f with inputs $x \in \mathbb{R}^n$ and outputs $z \in \mathbb{R}^m$ we want to calculate the Jacobian J with size (m, n)
- To get the full Jacobian, use the chain rule to differentiate from each output to each input.

$$J_{i,1:j} = \left\{ \frac{\partial z_i}{\partial x_1}, \cdots, \frac{\partial z_i}{\partial x_j} \right\}$$

Automatic Differentiation can do higher order partials, but here we just focus on the Jacobian

Cool Math, but how do we do this in a computer??

For Reverse Mode AD, we perform two functions.

- Forward Pass:

$$z = f(x_0, x_1)$$

- Reverse Pass: Given z's adjoint (gradient) $\overline{z}$

$$chain(z, x_0, x_1) = \left\{ \frac{\partial z}{\partial x_0}\overline{z}, \frac{\partial z}{\partial x_1}\overline{z} \right\}$$

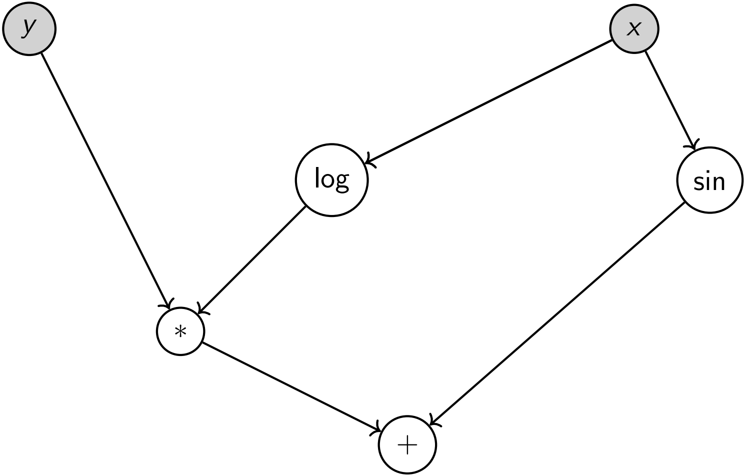
  Calculate the adjoint-jacobian update for x_0 and x_1.
- The calculations needed are represented as an expression Graph

What's an Expression Graph?

- Dependency graph of intermediate computations
- Think of both data and operations as objects
- Do a forward pass to calculate the values of the intermediates, then a reverse pass to calculate the adjoint-jacobian updates.

Forward Pass

$$z = \log(x) * y + \sin(x)$$

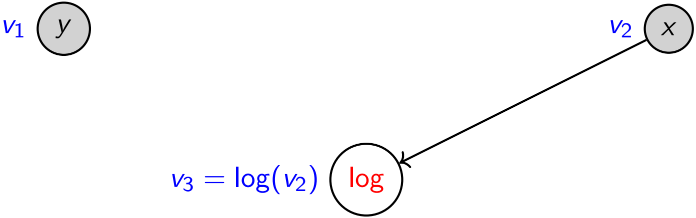


Forward Pass

$$z = \log(x) * y + \sin(x)$$

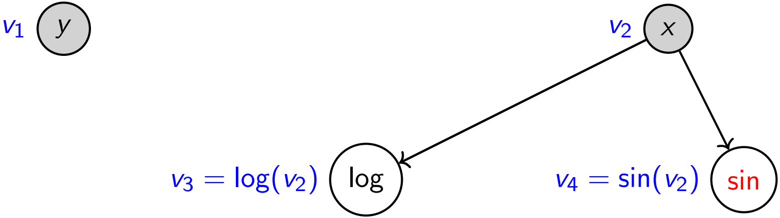
Forward Pass

$$z = \log(x) * y + \sin(x)$$



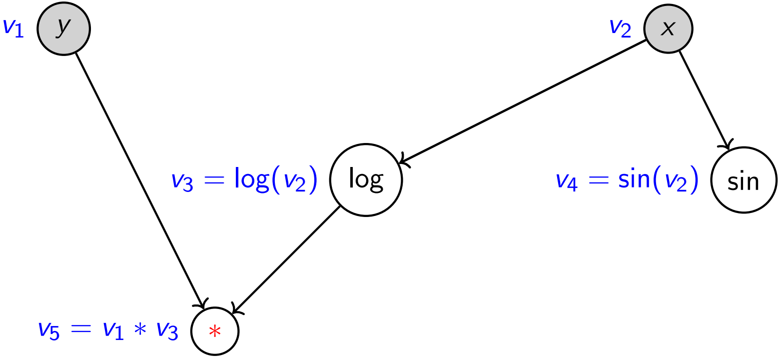
Forward Pass

$$z = \log(x) * y + \sin(x)$$



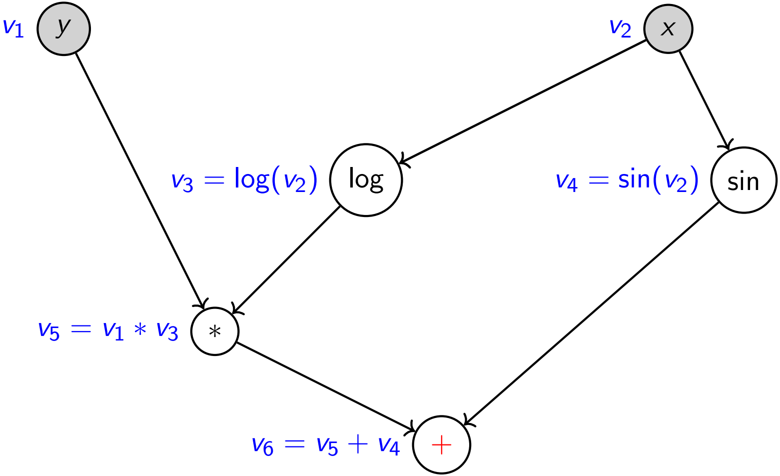
Forward Pass

$$z = log(x) * y + \sin(x)$$



Forward Pass

$$z = log(x) * y + \sin(x)$$



How do we calculate the adjoint jacobian?

Let $\overline{v}_i$ be the adjoint of v_i

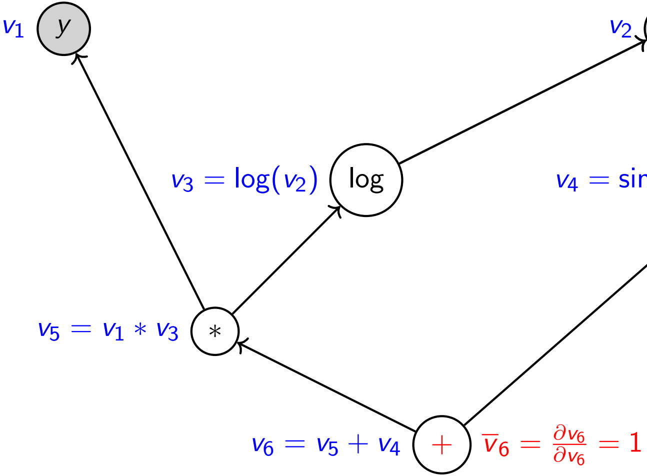
$$\overline{v}_i = \frac{\partial v_{i+1}}{\partial v_i}\overline{v}_{i+1}$$

Automatic Differentiation only needs the partials of the intermediates

$$\begin{aligned}
z &= x + y & \frac{\partial z}{\partial x} = 1, \frac{\partial z}{\partial y} = 1 \\
z &= x * y & y, x \\
z &= \log(x) & \frac{1}{x} \\
z &= sin(x) & \cos(x)
\end{aligned}$$

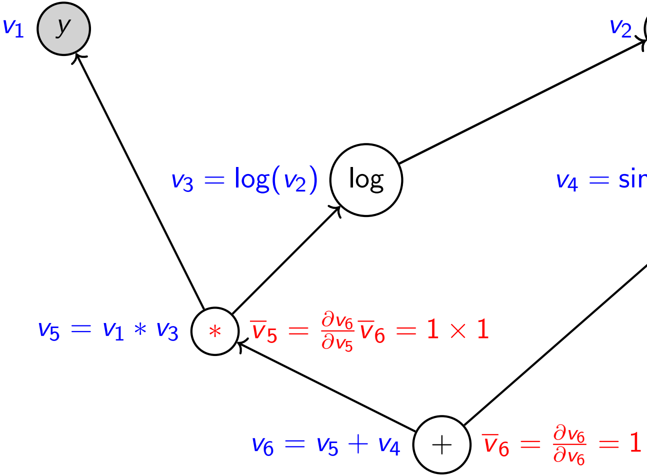
Reverse Pass

$$z = \log(x) * y + \sin(x)$$

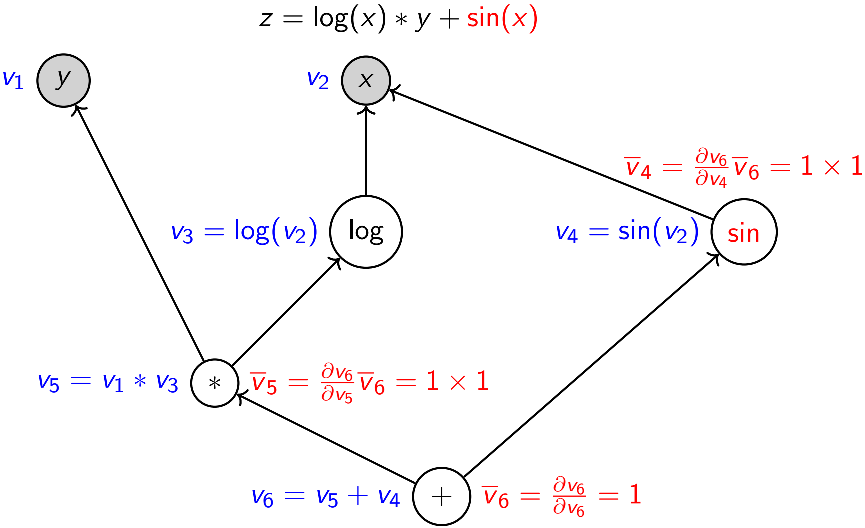


Reverse Pass

$$z = \log(x) * y + \sin(x)$$

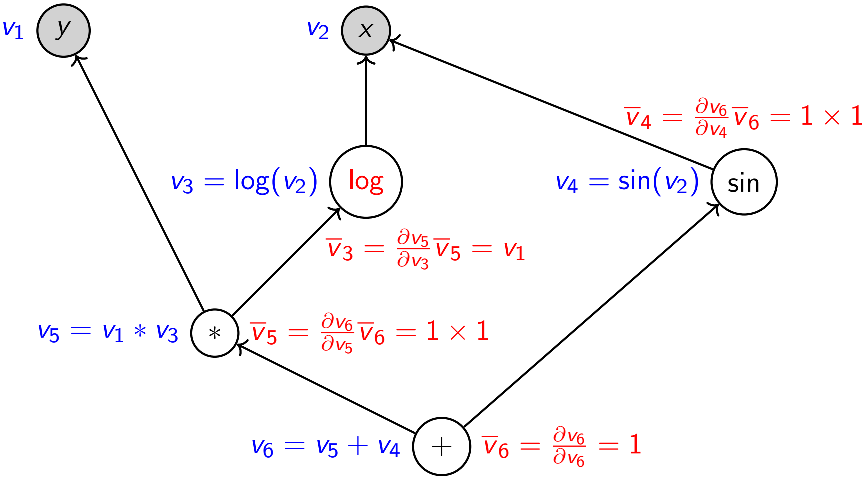


Reverse Pass

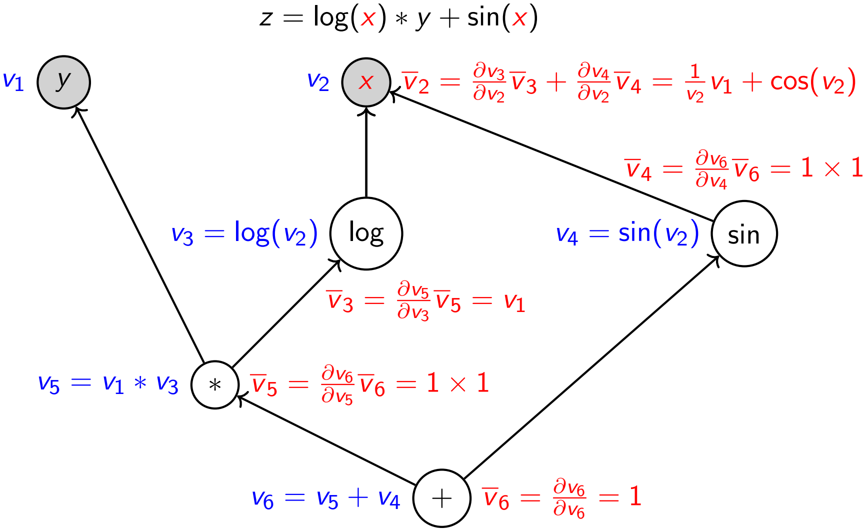


Reverse Pass

$$z = \log(x) * y + \sin(x)$$

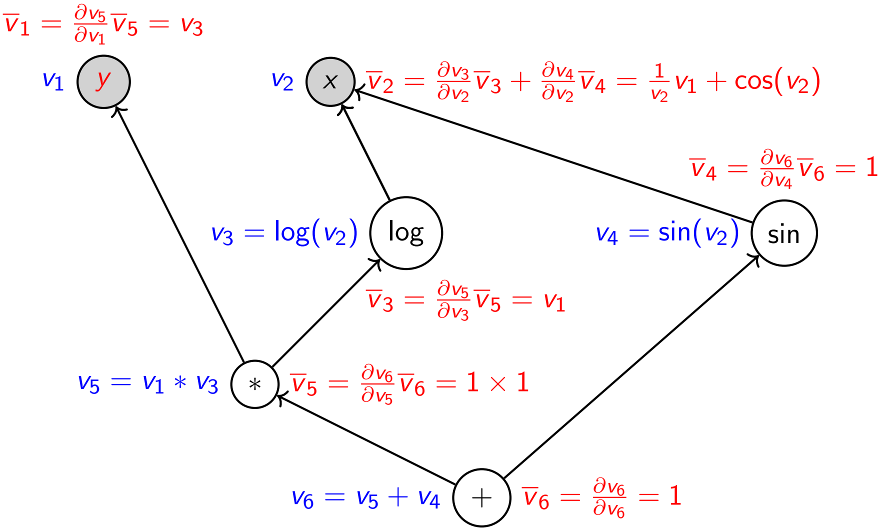


Reverse Pass



Reverse Pass

$$z = \log(x) * y + \sin(x)$$



Cool graph math, how do we do this in a computer?

We can think of AD operations as a function returning functions for the forward pass and the reverse pass

- Forward Pass:

```
f(value(x), value(y)) = z
```

- Reverse Pass:

```
adjoint(x) += adjoint(z) * partial_x(z, y, z)
adjoint(y) += adjoint(z) * partial_y(z, y, z)
```

Store reverse pass as a "tape"

Make A Tape

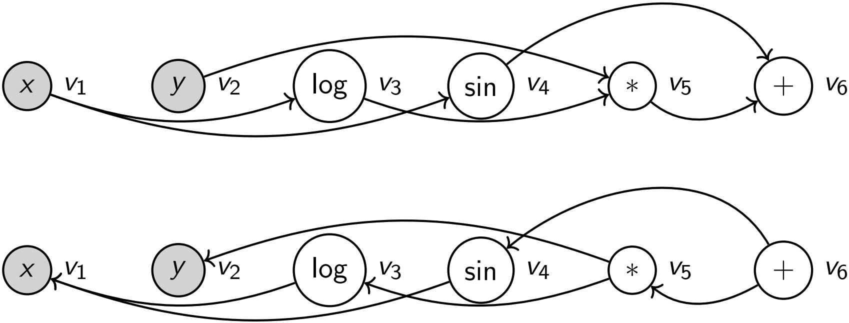


Figure: Topological sort of expression graph

How do we keep track of our reverse pass?

- Source code transformation
  - Unroll all forward passes and reverse passes into two functions

  Good: Fast

  Bad: Hard to implement, very restrictive
- Object oriented operator overloading
  - Nodes in the expression graph are objects which store a forward and reverse pass function

  Good: Easier to implement, more flexible

  Bad: Less optimization opportunities
- Newer AD packages use a combination of both of these ideas

How do we keep track of our reverse pass?

Most of the choices made are a balance between flexibility, performance, and developer time

- Static (Fast) vs. Dynamic (Flexible) graphs
  - Will our graph size change depending on conditionals? (Dynamic)
  - Do I know the size of my expression graph at compile time? (Static)
  - Can I allow reassignment of variables (Dynamic easy, Static v hard!)
- How much time do I have? (human time)

Source Code Transform Ex:

```
double z = log(x) * y + sin(x)
```

Break it down

```
double v1 = x
double v2 = y
double v3 = log(x)
double v4 = sin(x)
double v5 = v1 * v3
double v6 = v5 + v4
double z = v6
```

Source Code Transform Ex:

```
double z = x * y + sin(x)
```

Break it down

```
double v1 = x
double v2 = y
double v3 = log(x)
double v4 = sin(x)
double v5 = v1 * v3
double v6 = v5 + v4
double z = v6
double dv6 = 1
double dv5 = 1 * dv6
double dv4 = 1 * dv6
double dv3 = v1 * dv5
// Final output
double dv2 = 1 / v2 * dv3 + cos(v2) * dv4
double dv1 = v3 * dv5
```

Source Code Transform Ex:

Code like the following very hard / impossible in source code transform

```
while(error < tolerance) {
  // ...
}
```

Object Oriented Approach

- The object oriented approach usually involves:
  - A vector or list to track the expression graph for the reverse pass function calls
  - A pair to hold the value and adjoint
- Very flexible: Allows conditional loops and reassignment of values in matrices
- Nodes of expression graph can be collapsed

Example Godbolt

Object Oriented Approach: Matrices

- Either Array of Structs (AOS) or Struct of Arrays (SoA)

```
struct var {
  double value_;
  double adjoint_;
};
struct MatrixVar {
  var* data_;
  MatrixVar(std::size_t N) :
   data_(static_cast<var*>(malloc(sizeof(var) * N)) {}
}
MatrixVar aos_matrix;

struct VarMatrix {
  double* value_;
  double* adjoint_;
}
VarMatrix soa_matrix;
```

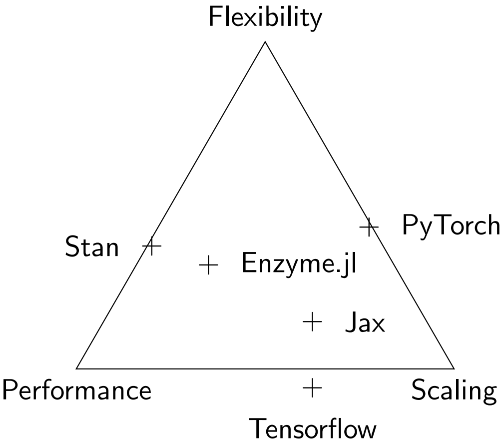
Object Oriented Approach: Matrices

- Array of Structs:
  - Simple, most algorithms Just Work™
  - Adds a lot to expression graph
  - turns off SIMD
- Struct of Arrays:
  - Hard, everything written out manually
  - Collapses matrix expressions in tree
  - SIMD can be used on values and adjoints

What Do AD Libraries Care About?

- Flexibility:
  - Debugging, exceptions, conditional loops, matrix subset assignment
- : Efficiency:
  - Efficiently using a single CPU/GPU
- Scaling
  - Efficiently using clusters with multi-gpu/cpu nodes

What are the AD packages like?



What are the AD packages like?

Disclaimer: Just pick the package that does the things you like, the ones here are performant enough

Common Autodiff Packages

- Static Graph
  - TensorFlow, Jax, Enzyme
- Dynamic Graph
  - Pytorch and Stan
- TF, Jax, and Pytorch now have both

Stan!

Good:

- Very flexible language
- Exceptions, conditionals loops, matrix subsetting
- Only known CPU AD package faster than Stan math is FastAD
- Simple C like Domain Specific Language (DSL)

Bad:

- Very limited GPU support at the language level
- Poor scaling for TB of data
- Simple C like Domain Specific Language (DSL)
- Compilation times

Pytorch

Good:

- Good multi-gpu support
- Exceptions, conditional loops, debugging first priority
- Builtins for neural networks
- Extensible (see pytorch-finufft)

Bad:

- Subset assignment to matrices and vectors is a full hard copy
- Backend is very hard to parse

Tensorflow

Good:

- Made for scalability

Bad:

- Just a very gross language imo
- No conditional loops
- No subset assignment to matrices and vectors
- No exceptions

Jax

Good:

- Built on top of autograd and XLA
- Well documented
- Extendable
- Write python, jit to near C++ speed

Bad:

- No Exceptions, conditional loops, subset assignment to matrices and vectors is a full hard copy

Enzyme.jl

Good:

- JIT compiled to llvm
- Can use a large amount of julia packages

Bad:

- Only one main maintainer
- Not yet 1.0 (0.1)
- No GC or dynamic dispatch support

What did we talk about?

- What's Automatic Differentiation (AD)?
  - Evaluates partial derivatives of a program
- Why should you care?
  - It's important and used a lot
- What's an expression graph?
  - Graphic to describe dependencies for AD
- How is AD implemented
  - Source code is transformed (static graph) or objects are made for intermediate ops (dynamic graph)
- What are the tradeoffs between different AD packages
  - Flexibility, Efficiency, and Scale