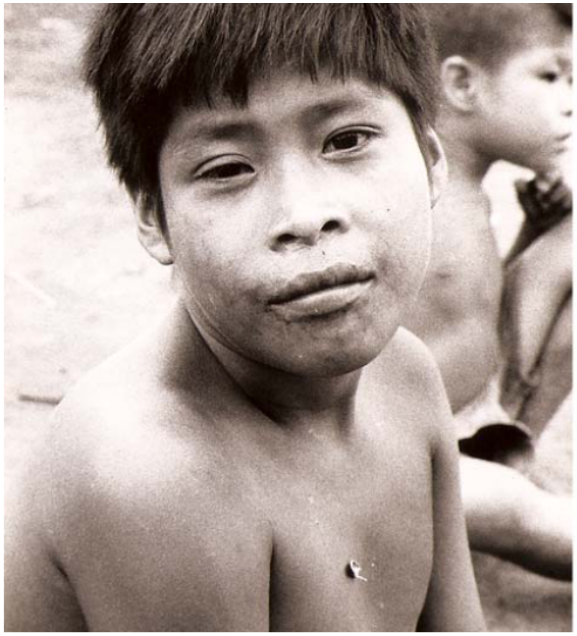
Reverse Mode Automatic Differentiation: Unraveling Expression Graphs & Library Magic

Steve Bronder

October 2023



1978

2009

From:

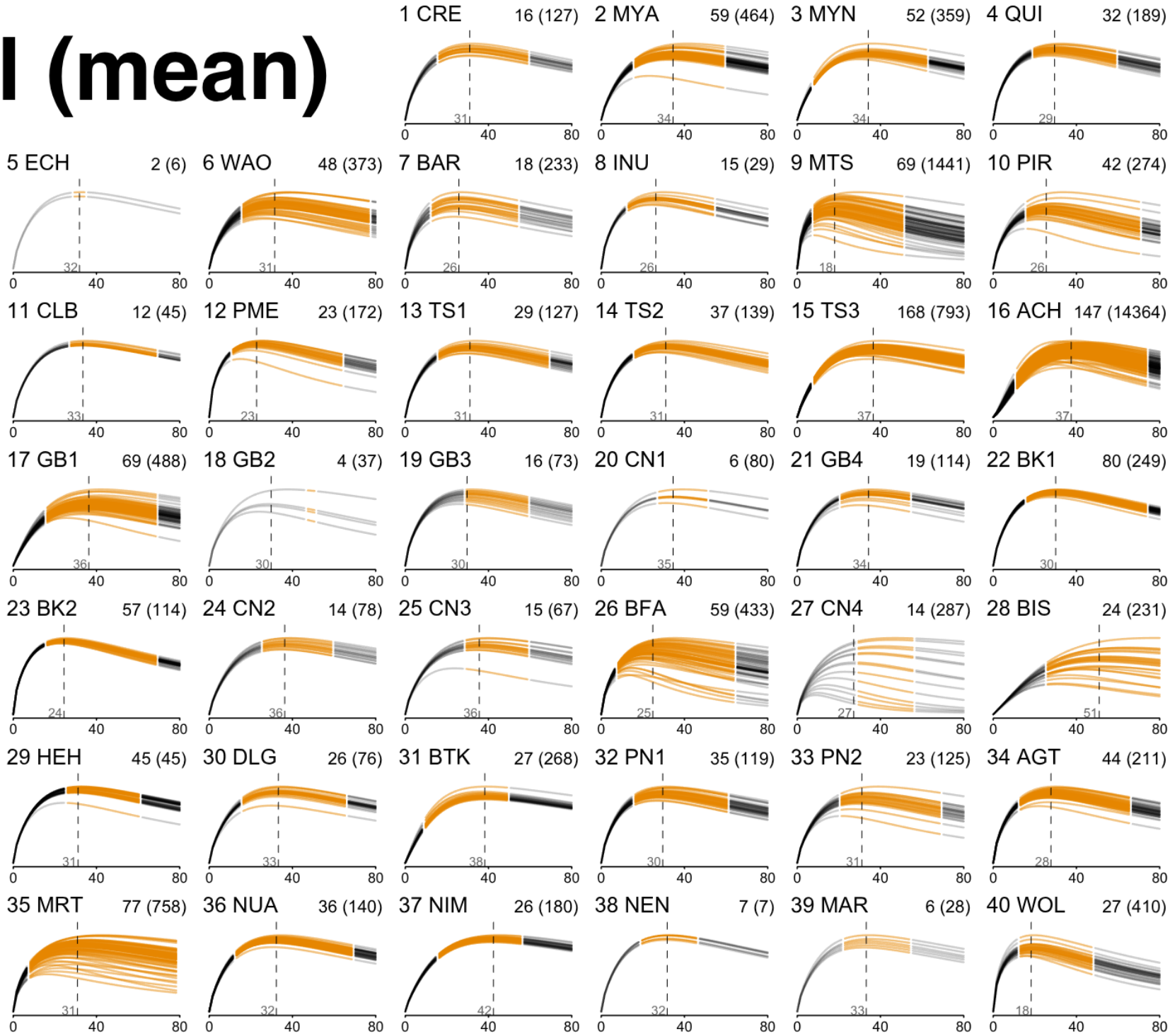
Can Anthropologists Distinguish Good and Poor Hunters? Implications for Hunting Hypotheses, Sharing Conventions, and Cultural Transmission

Kim Hill and **Keith Kintigh**

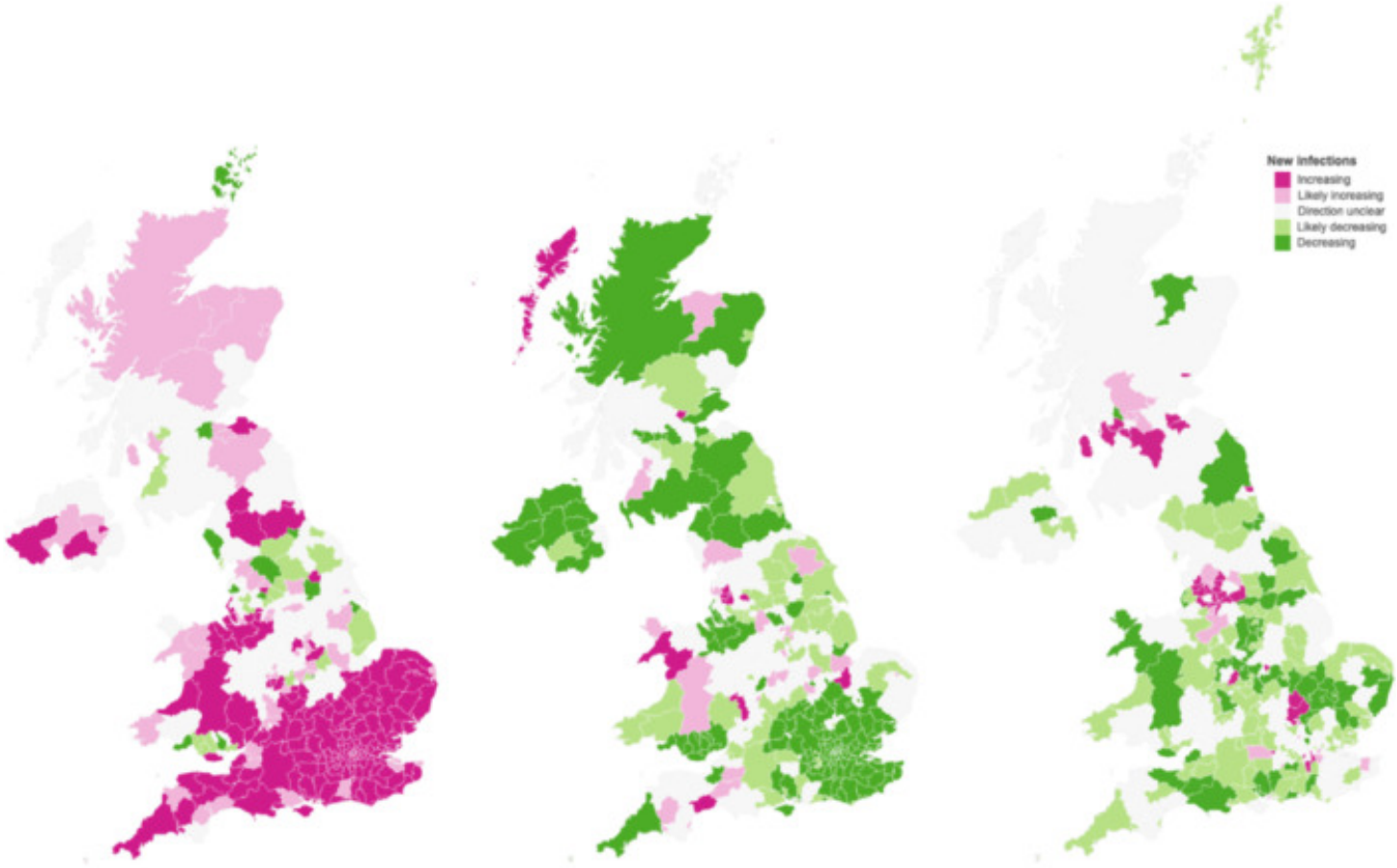
Life History of Production Skill

- Goals
  - Estimate skill development
  - Take variation seriously
  - Develop stats machinery
- Sample
  - 40 research sites
  - 1821 individuals
  - 21,160 foraging trips
  - 23,747 harvests
  - Uncountable headaches

Skill (mean)


1 CRE 16 (127)
2 MYA 59 (464)
3 MYN 52 (359)
4 QUI 32 (189)
5 ECH 2 (6)
6 WAO 48 (373)
7 BAR 18 (233)
8 INU 15 (29)
9 MTS 69 (1441)
10 PIR 42 (274)
11 CLB 12 (45)
12 PME 23 (172)
13 TS1 29 (127)
14 TS2 37 (139)
15 TS3 168 (793)
16 ACH 147 (14364)
17 GB1 69 (488)
18 GB2 4 (37)
19 GB3 16 (73)
20 CN1 6 (80)
21 GB4 19 (114)
22 BK1 80 (249)
23 BK2 57 (114)
24 CN2 14 (78)
25 CN3 15 (67)
26 BFA 59 (433)
27 CN4 14 (287)
28 BIS 24 (231)
29 HEH 45 (45)
30 DLG 26 (76)
31 BTK 27 (268)
32 PN1 35 (119)
33 PN2 23 (125)
34 AGT 44 (211)
35 MRT 77 (758)
36 NUA 36 (140)
37 NIM 26 (180)
38 NEN 7 (7)
39 MAR 6 (28)
40 WOL 27 (410)

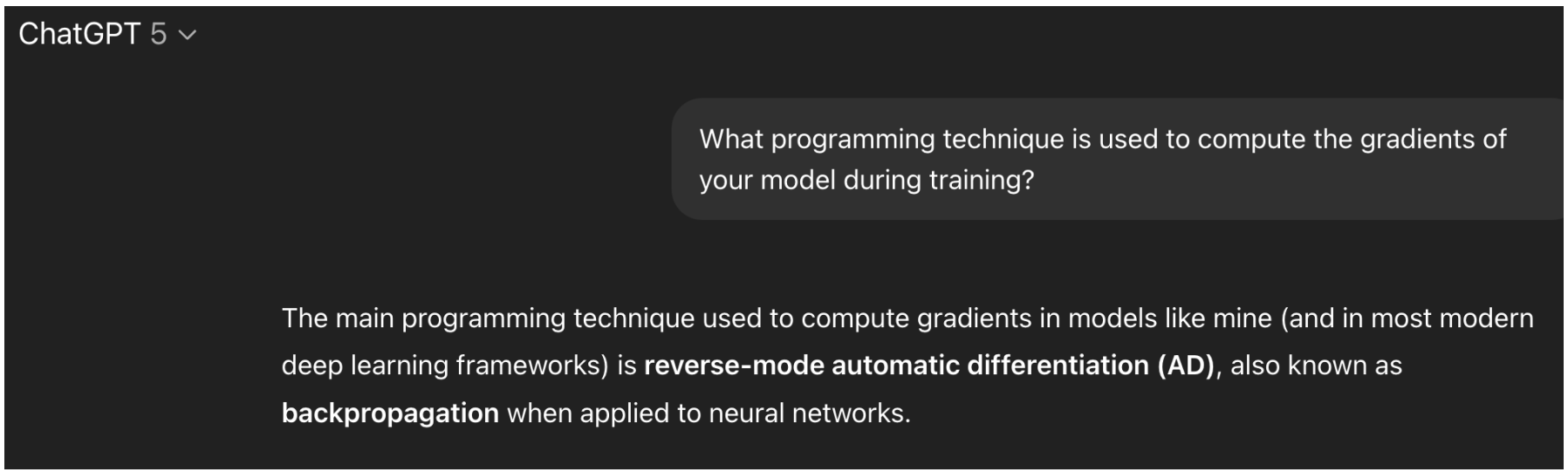
Estimating COVID Infection Rates For Policy



(a) 25 Dec 2020 (b) 21 Jan 2021 (c) 30 May 2021

Figure: Probability of epidemic growth by local area

Automatic Differentiation Probably Effects Your Day to Day



What's Automatic Differentiation?

Computational technique for evaluating derivatives of functions expressed as computer programs by systematically applying the chain rule.

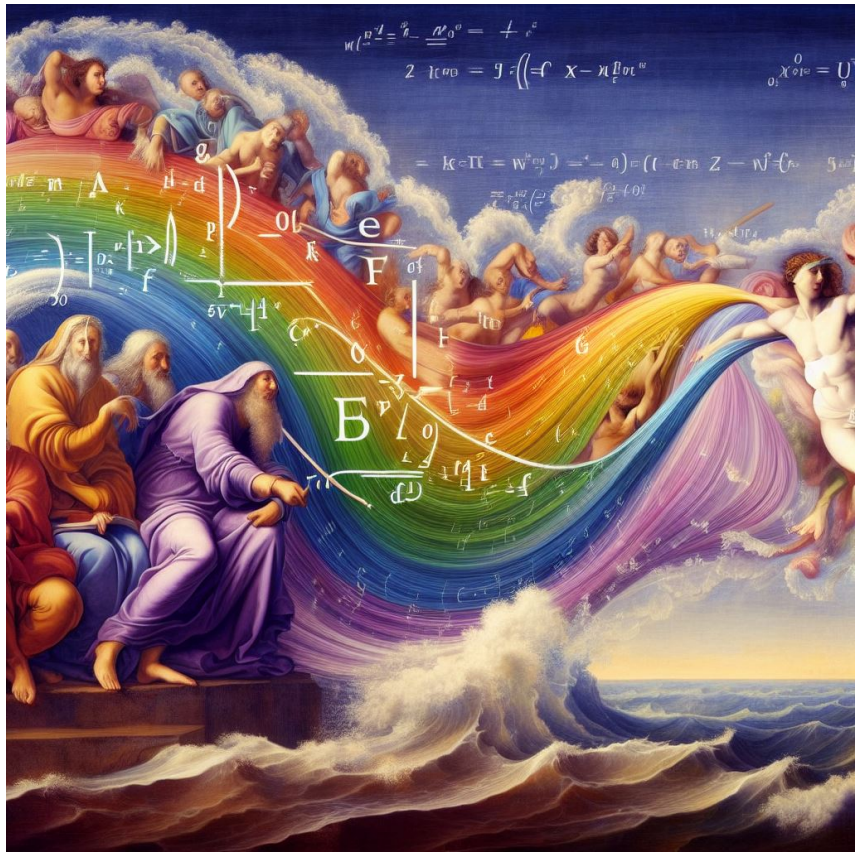


Figure: Asking ChatGPT to make a physical representation of automatic differentiation.

Why use Automatic Differentiation?

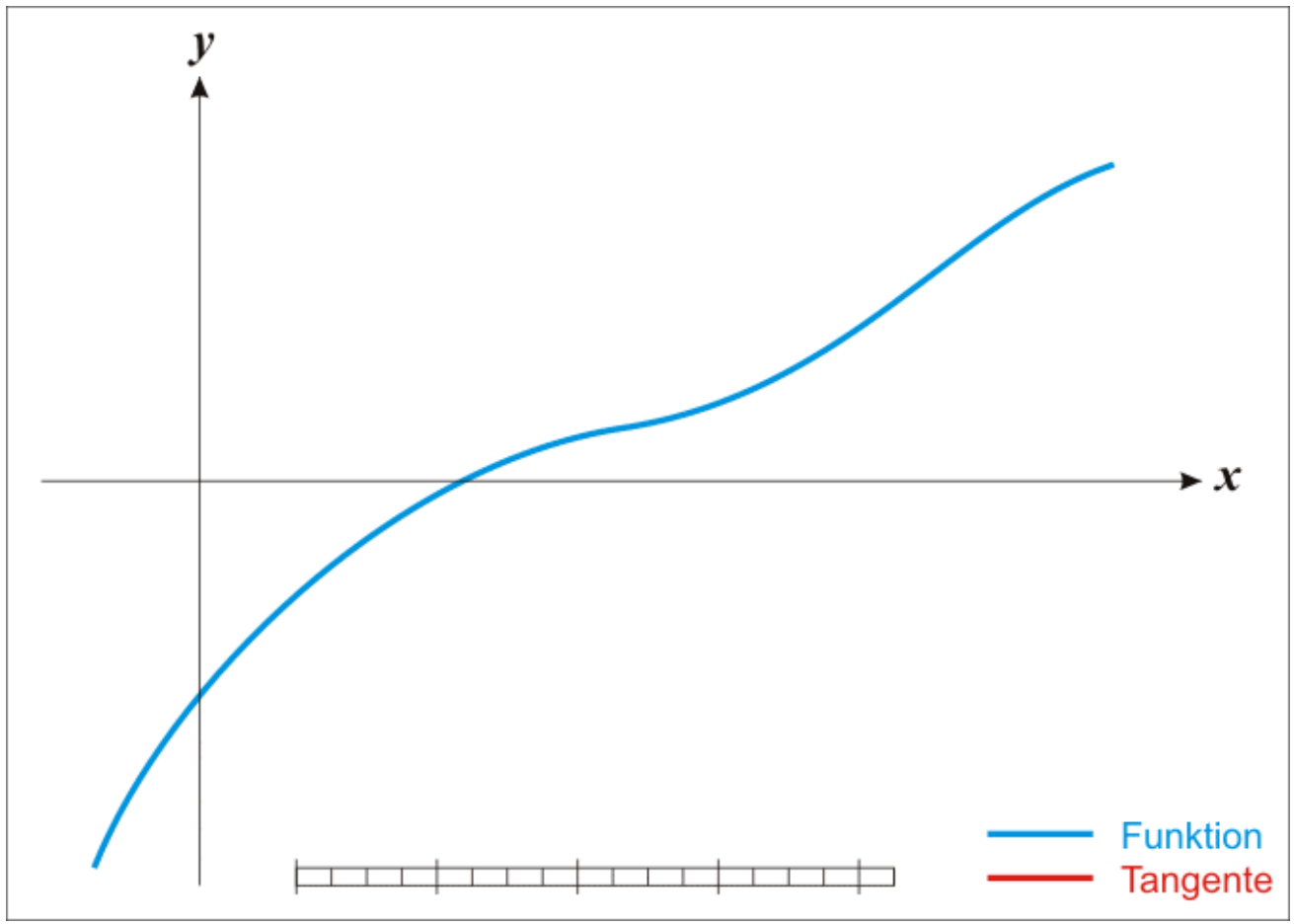
Ex: Newton's method for finding root of function (where output is 0)

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

Why use Automatic Differentiation?

$$f(x) = x^3 + x^2 + x \tag{1}$$

$$f'(x) = 3x^2 + 2x + 1 \tag{2}$$



Why use Automatic Differentiation?

Think about HMC, BFGS, SGD, etc.

- HMC: $\frac{dp}{dt} = -\nabla_\theta \log p(\theta|y)$
- BFGS: $s_k = -H_k \nabla_\theta f(\theta_k)$
- SGD: $\theta_{t+1} = \theta_t - \eta \nabla_\theta L(\theta_t; x_t)$
- Choices
  - Write by hand
  - finite difference,
  - symbolic differentiation
  - spectral differentiation
  - automatic differentiation

Why use Automatic Differentiation?

$$\underbrace{p(\boldsymbol{\theta} \mid \mathbf{y})}_{\textbf{posterior}} \propto \prod_{i=1}^{N} \left\{ \sum_{\mathbf{z}_i \in \{1,2,3\}^{T_i}} \left[\underbrace{\pi_{z_{i,1}} \prod_{t=2}^{T_i} \Pi_{z_{i,t-1},\, z_{i,t}}}_{\text{3-state HMM prior}} \prod_{t=1}^{T_i} \underbrace{\mathcal{N}\left(y_{i,t} \,\middle|\, \eta_{i,t},\ \sigma^2_{z_{i,t}}\right)}_{\text{state-dependent emission}} \right] \right\}$$

where $\eta_{i,t} = \underbrace{\mathbf{x}_{i,t}^\top \boldsymbol{\beta}}_{\text{fixed}} + \underbrace{\mathbf{z}_{i,t}^{(G)\top} \mathbf{b}_{g[i]} + \mathbf{z}_{i,t}^{(C)\top} \mathbf{c}_{c[i]} + u_{g[i]}}_{\text{crossed random effects}} + \underbrace{f(t_{i,t})}_{\text{GP}} + \underbrace{\mu_{z_{i,t}} + \mathbf{r}_{z_{i,t}}^\top \mathbf{w}_i}_{\text{state-specific offset + slope}}, \qquad z_{i,t} \in \{1, 2, 3\}.$

$$p(\mathbf{f} \mid \boldsymbol{\psi}) = (2\pi)^{-T/2} \, |\mathbf{K}|^{-1/2} \, \exp\Big(-\tfrac{1}{2} \mathbf{f}^\top \mathbf{K}^{-1} \mathbf{f} \Big),$$

$$\mathbf{K} = \sigma_f^2 \Big(\mathbf{K}_{\mathsf{LP}}(\ell, p, \lambda) \ + \ \rho \, \mathbf{K}_{\mathsf{SE}}(\tilde{\ell}) \Big) + \sigma_n^2 \mathbf{I}, \quad \left[\mathbf{K}_{\mathsf{LP}}\right]_{tt'} = \exp\left(-\frac{(t-t')^2}{2\ell^2} - \frac{2\sin^2\left(\pi |t-t'|/p\right)}{\lambda^2} \right),$$

$$\left[\mathbf{K}_{\mathsf{SE}}\right]_{tt'} = \exp\left(-\frac{(t-t')^2}{2\tilde{\ell}^2} \right), \qquad \mathbf{K} = \mathbf{L}_K \mathbf{L}_K^\top \ \Rightarrow \ \log|\mathbf{K}| = 2 \sum_{j=1}^{T} \log\left((\mathbf{L}_K)_{jj} \right).$$

Hierarchical mixed effects (non-centered, LKJ prior):

$$\mathbf{b}_g = \left(\mathbf{I}_{p_G} \otimes \operatorname{diag}(\boldsymbol{\tau}_b) \, \mathbf{L}_R \right) \tilde{\mathbf{b}}_g, \quad \tilde{\mathbf{b}}_g \sim \mathcal{N}(\mathbf{0}, \mathbf{I}), \quad \mathbf{c}_c = \operatorname{diag}(\boldsymbol{\tau}_c) \, \tilde{\mathbf{c}}_c, \quad \tilde{\mathbf{c}}_c \sim \mathcal{N}(\mathbf{0}, \mathbf{I}),$$

$$\mathrm{LKJ}_{p_G}(\eta) \text{ prior on } \mathbf{R}, \quad \mathbf{L}_R \mathbf{L}_R^\top = \mathbf{R}, \quad \boldsymbol{\tau}_b \sim \prod_{j=1}^{p_G} \text{Half-}t_{\nu_b}(0, s_b), \quad \boldsymbol{\tau}_c \sim \prod_{j=1}^{p_C} \text{Half-}t_{\nu_c}(0, s_c),$$

$$u_g \sim \mathcal{N}(0, \sigma_u^2).$$

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

How Fast is AutoDiff?

Figure: AuToDiFf rUnS iN $\Theta(C(f))$ TiMe

Impl Matters!

TODO: QR code here

Regression

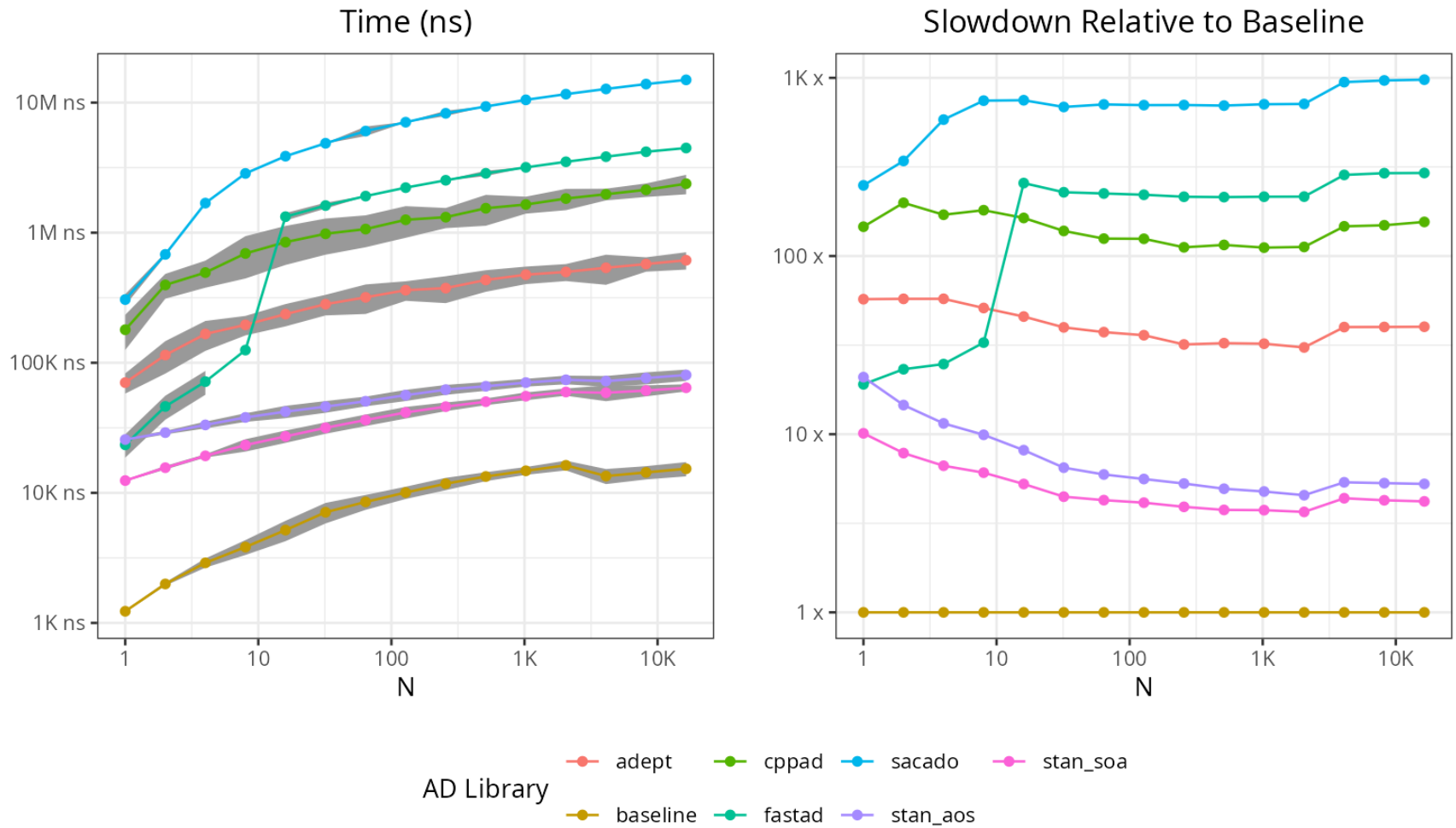


Figure: Benchmark for f' given $f = y \sim N(X\theta, \sigma)$

Goal of this talk

- Understand performance of high throughput memory intensive programs
- Show how modern C++ through time has led to cleaner and more efficient AD

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) $\bar{z}$

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

  Calculate the adjoint-jacobian update for x_0 and x_1.

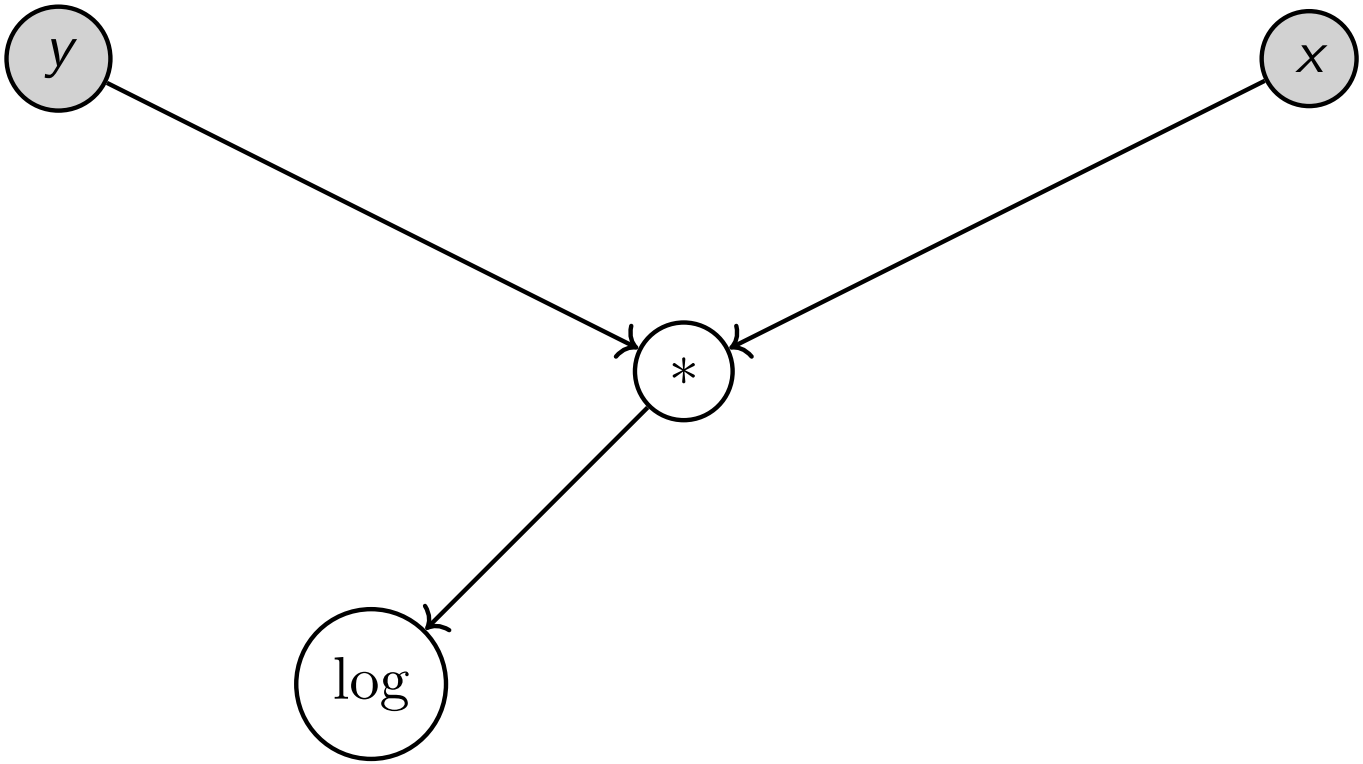
- The calculations needed are represented as an expression Graph

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.

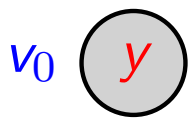
Forward Pass

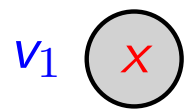
$$z = \log(x * y)$$



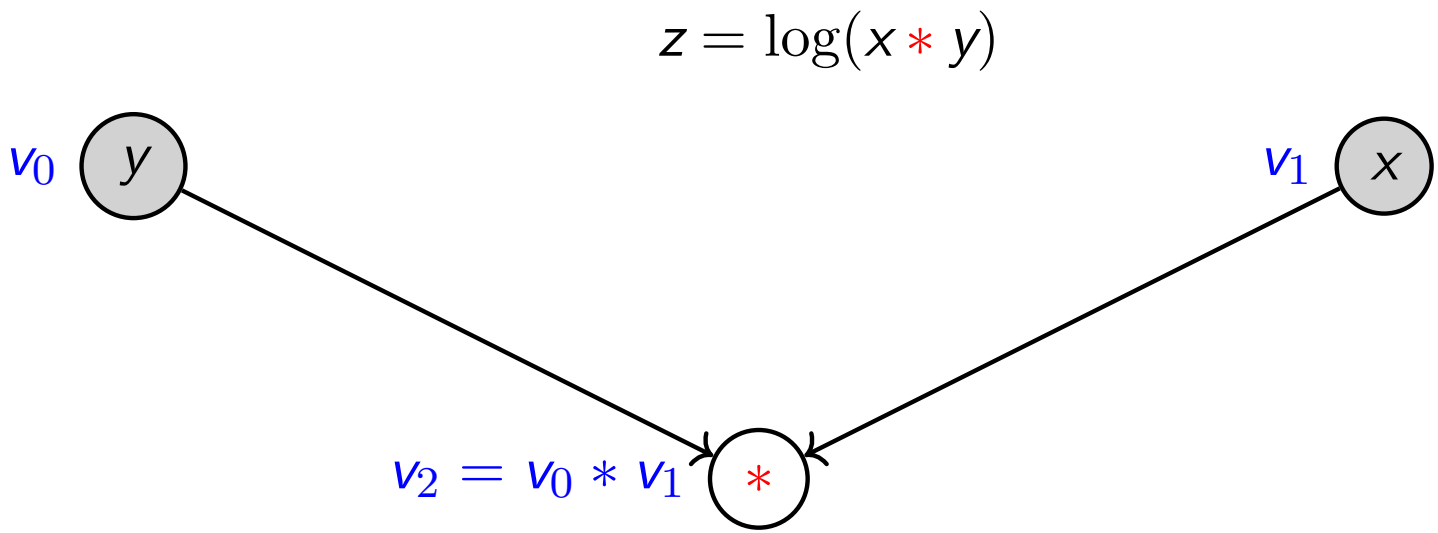
Forward Pass

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

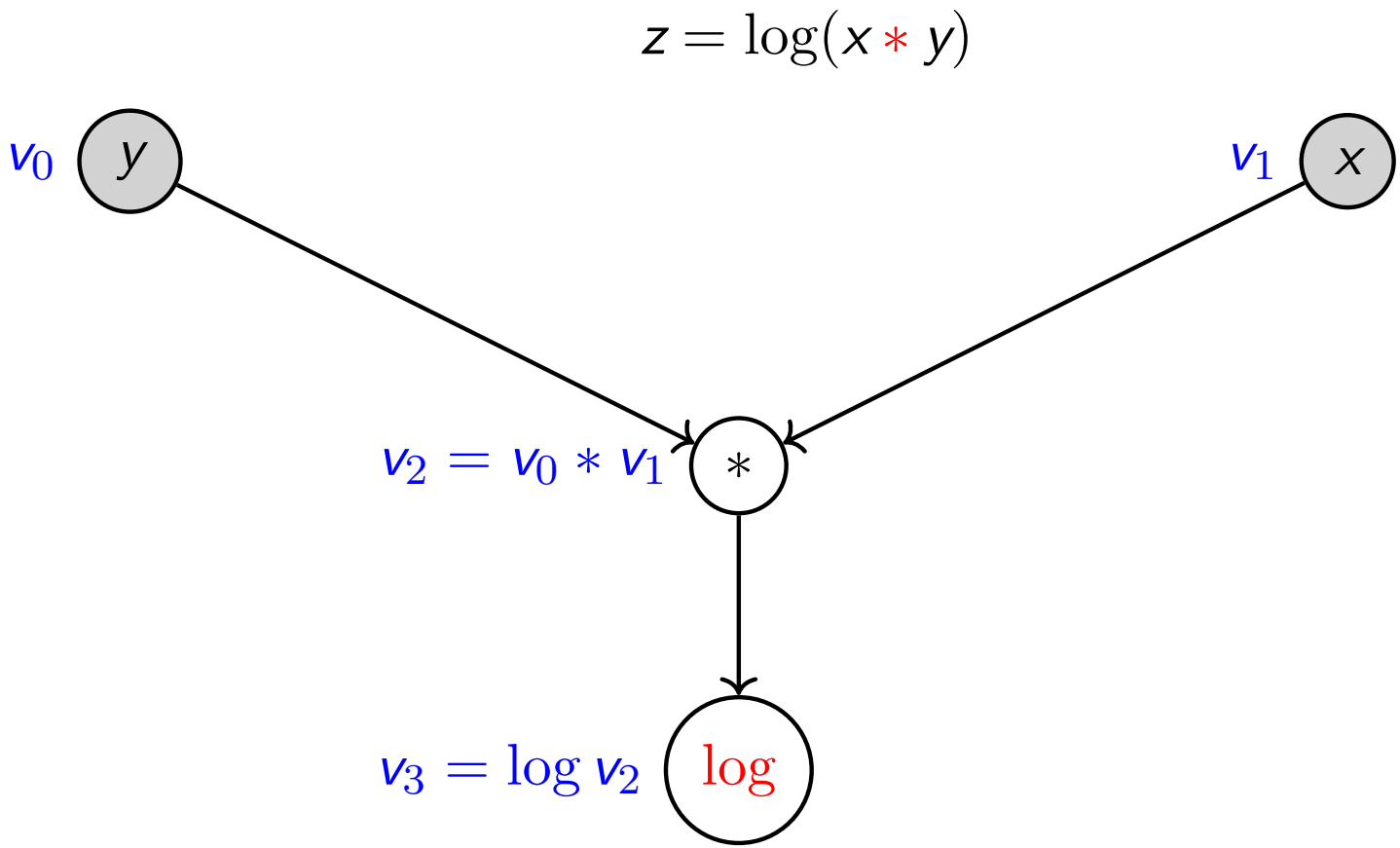




Forward Pass



Forward Pass



How do we calculate the adjoint jacobian?

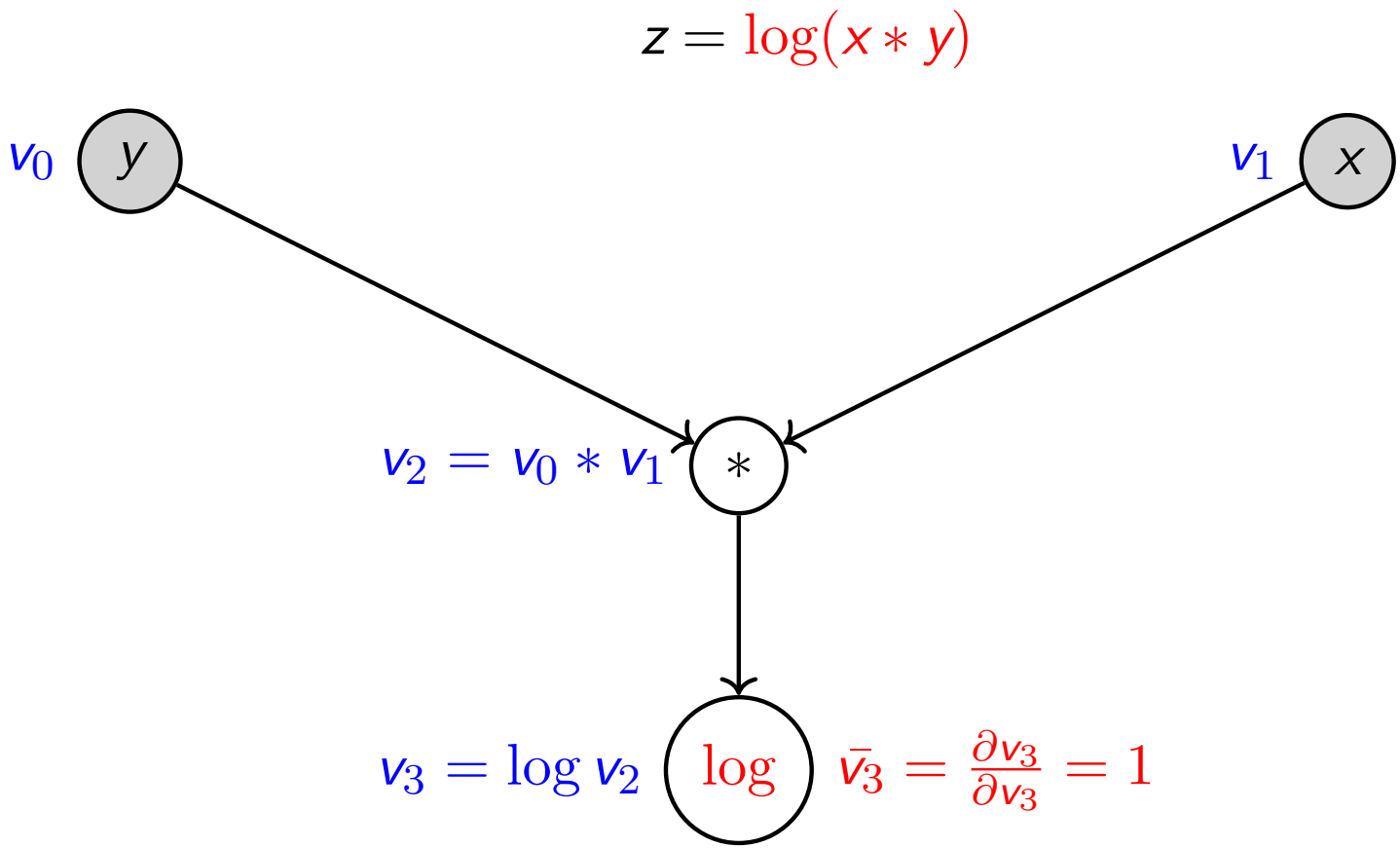
Let $\overline{v}_i$ be the adjoint of v_i

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

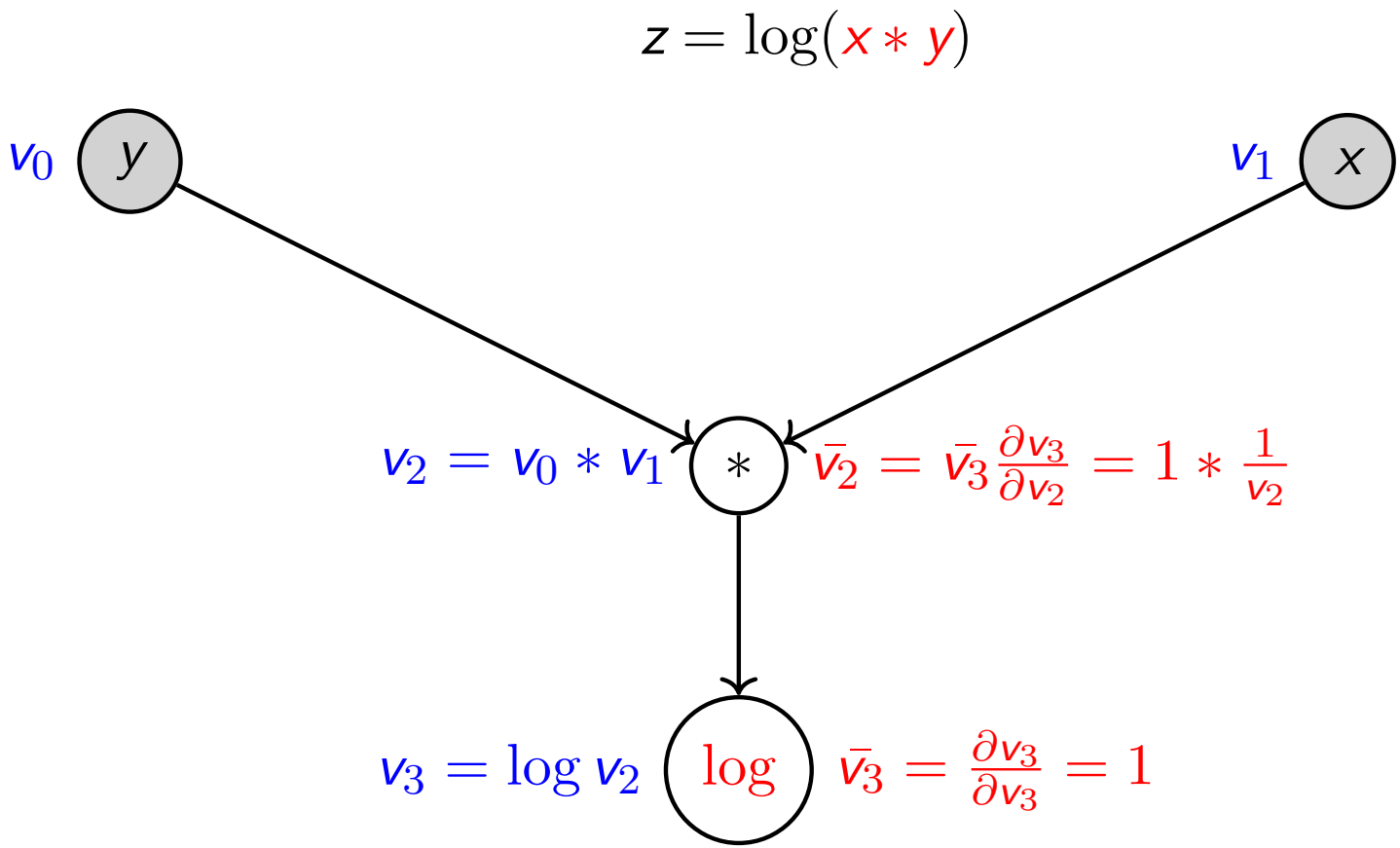
Automatic Differentiation only needs the partials of the intermediates

$z = x * y$	$\frac{\partial z}{\partial x} = y, \frac{\partial z}{\partial y} = x$
$z = \log(x)$	$\frac{1}{x}$

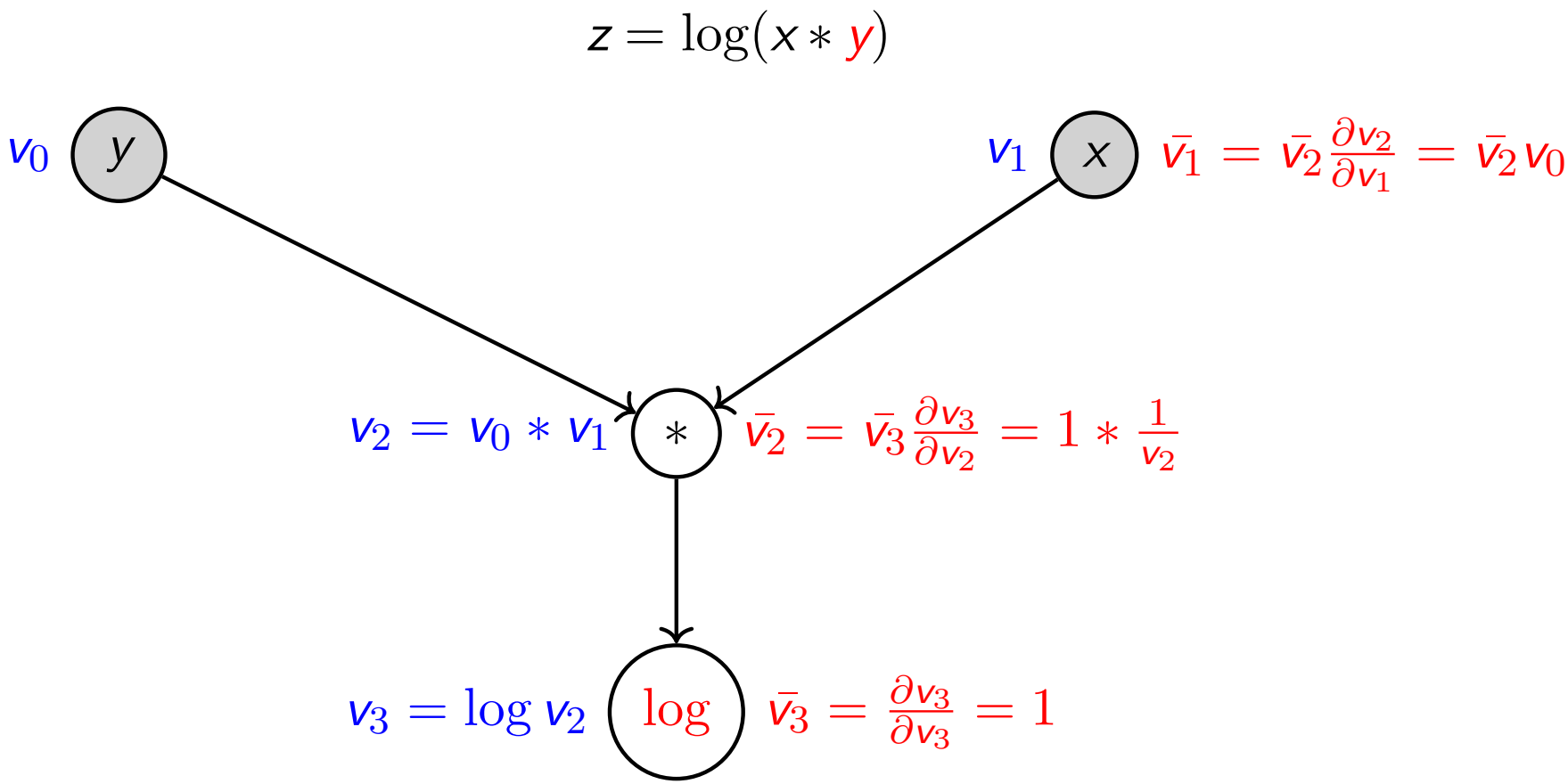
Reverse Pass



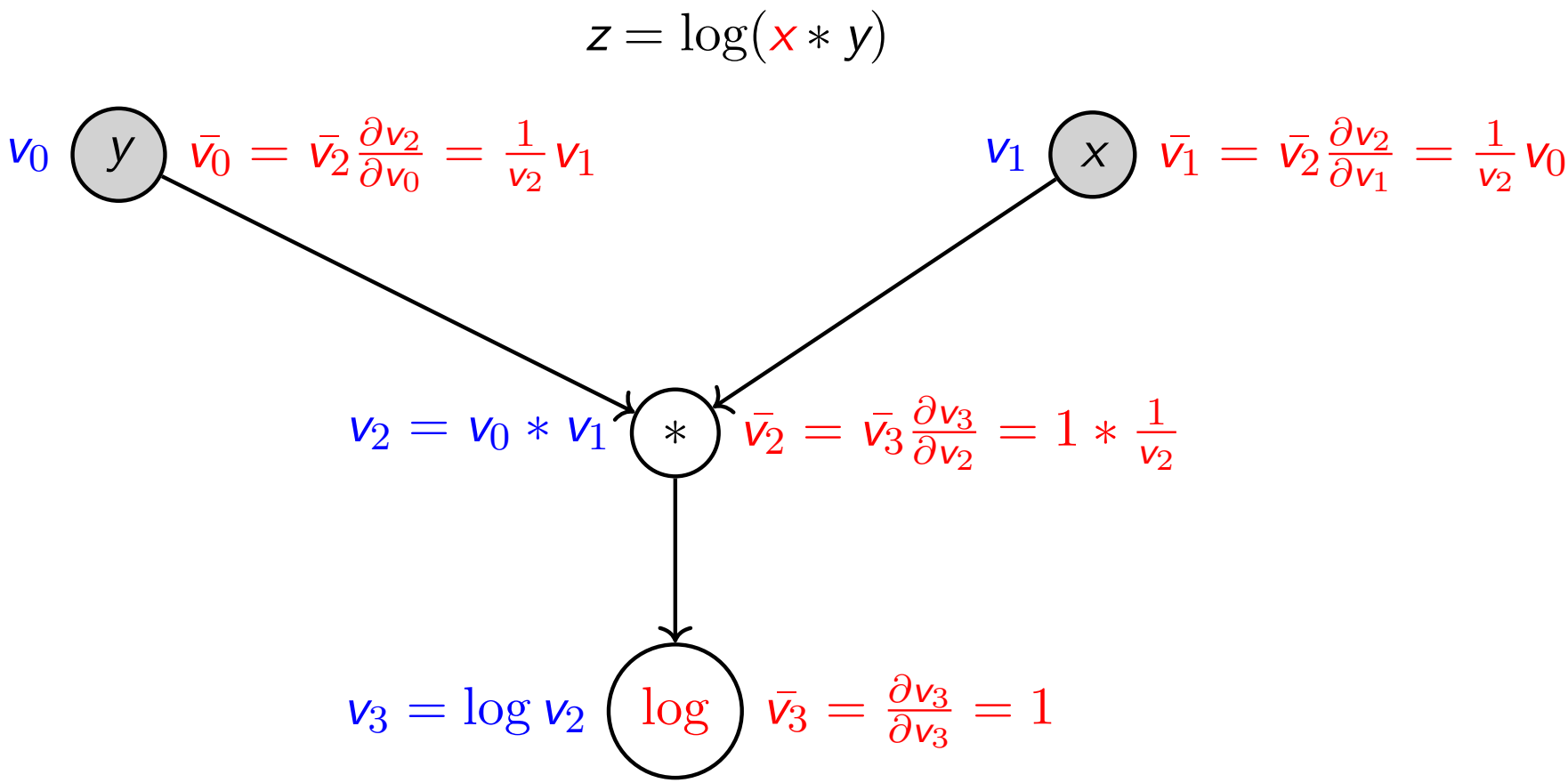
Reverse Pass



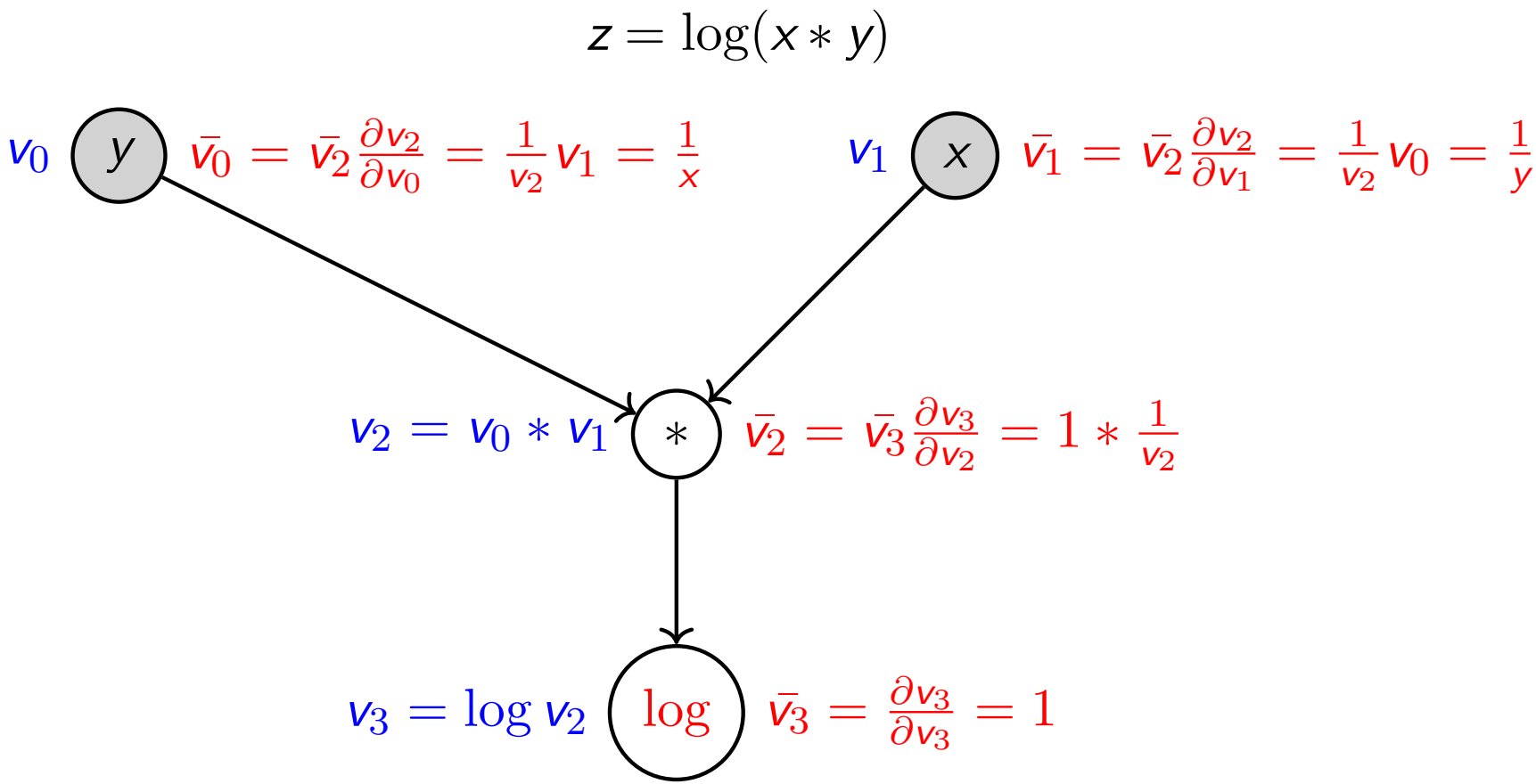
Reverse Pass



Reverse Pass



Reverse Pass



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

How do we keep track of our reverse pass?

- Source code transformation
  - Unroll all forward passes and reverse passes into one function

  Good: Fast

  Bad: Hard to implement, very restrictive

How do we keep track of our reverse pass?

- Source code transformation
  - Unroll all forward passes and reverse passes into one function

  Good: Fast

  Bad: Hard to implement, very restrictive
- 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

How do we keep track of our reverse pass?

Static (Fast) vs. Dynamic (Flexible) graphs

- Known expression graph size at compile time? (Static)
- Reassignment of variables (dynamic easy, Static vv hard!)
- How much time do I have? (dynamic)

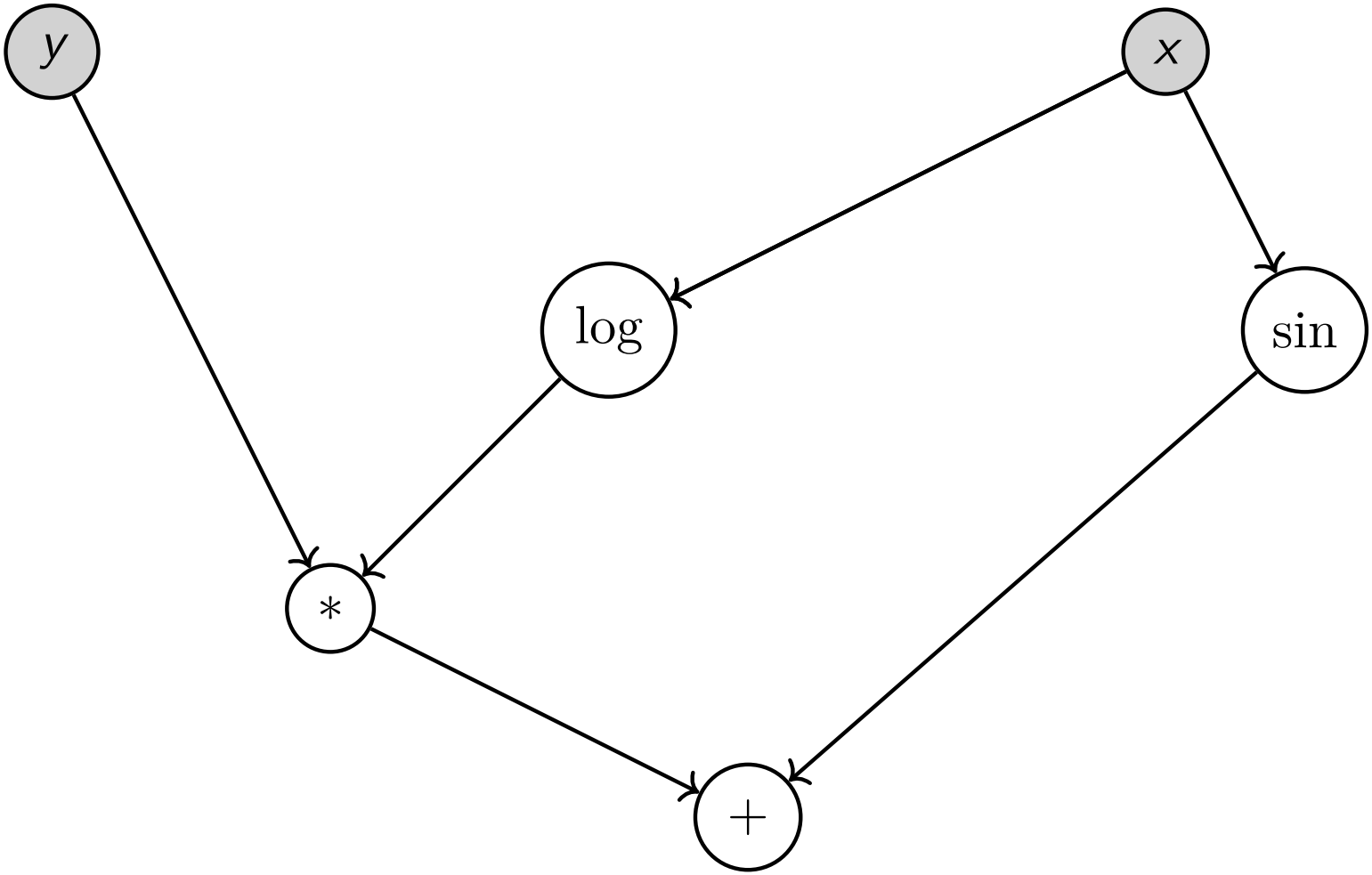
How do we keep track of our reverse pass?

Static (Fast) vs. Dynamic (Flexible) graphs

- Known expression graph size at compile time? (Static)
- Reassignment of variables (dynamic easy, Static vv hard!)
- How much time do I have? (dynamic)
- Do I actually need to track my graph :)

Make A Tape

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



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

```
1  auto foo_fwd(double x, double y);
```

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

```
1  auto foo_fwd(double x, double y);
2
3  template<ADType Ret, ADType T1, ADType T2>
4  auto foo_rev(Ret&& z, T1& x, T2& y) {
5    adjoint(x) += adjoint(z) * compute_adj(x);
6    adjoint(y) += adjoint(z) * compute_adj(y);
7  }
```

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

```
auto foo_fwd(double x, double y);
template<ADType Ret, ADType T1, ADType T2>
auto foo_rev(Ret&& z, T1&& x, T2&& y) {
  adjoint(x) += adjoint(z) * compute_adj(x);
  adjoint(y) += adjoint(z) * compute_adj(y);
}
template<ADType T1, ADType T2>
auto foo(T1&& x, T2&& y) {
  // Fwd pass
  auto fwd_ret = foo_fwd(value(x), value(y));
  // Setup reverse pass
  auto foo_rev = [](auto&& z, auto& x, auto& y) {
    return my_func_rev(z, x, y);
  };
  return tuple{foo_rev, fwd_ret, x, y};
}
```

Operator Overloading Approach

- The operator overloading approach usually involves:
  - Tracking the expression graph for the reverse pass function calls
  - A pair scalar type 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

Operator Overloading: Simple

```
1  struct var_impl {
2    double val_;
3    double adj_;
4    virtual void chain() {}
5    var_impl(double val) : val_(val), adj_(0.0) {}
6  };
7  struct var {
8    std::shared_ptr<var_impl> vi_;
9    var(double x) :
       ↪  vi_(std::make_shared<var_impl>(x)) {
0    }
1  };
```

Operator Overloading: Simple

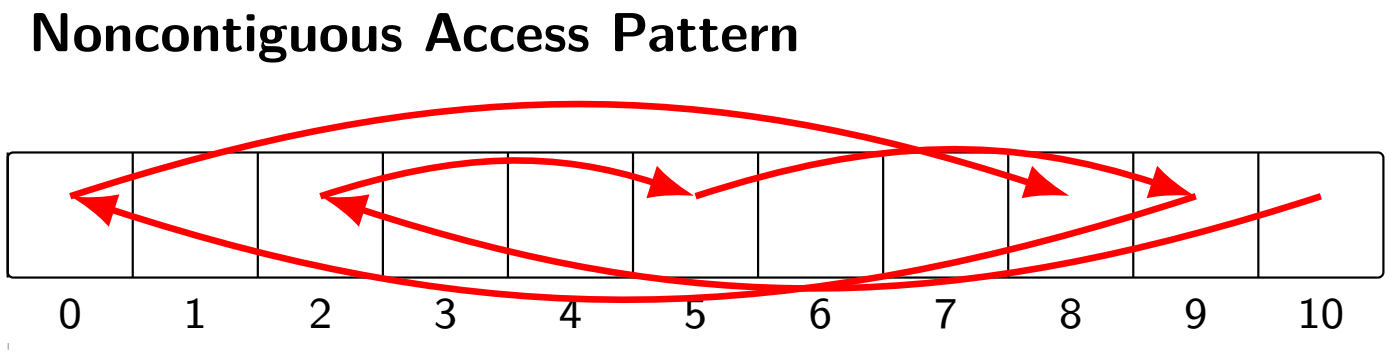
```
1  struct mul_vv final : public var_impl {
2    var lhs_;
3    var rhs_;
4    mul_vv(double val, var lhs, var rhs) :
       ↪  var_impl(val), lhs_(lhs), rhs_(rhs) {}
5    void chain() {
6      lhs_.adj() += rhs_.val() * this->adjoint_;
7      rhs_.adj() += lhs_.val() * this->adjoint_;
8      lhs_.chain();
9      rhs_.chain();
0    }
1  };
2  operator*(var x, var y) {
3    return var{
4      std::make_shared<mul_vv>(
5        x.val() * y.val(), x, y)};
6  }
```

Operator Overloading: Simple

```
1  void grad(var& v) {
2    v.adj() = 1.0;
3    v.chain();
4  }
5  var x(2.0);
6  var y(4.0);
7  auto z = x;
8  while (value(z) < 10) {
9    z += x * log(y) + log(x * y) * y;
0  }
1  grad(z);
```

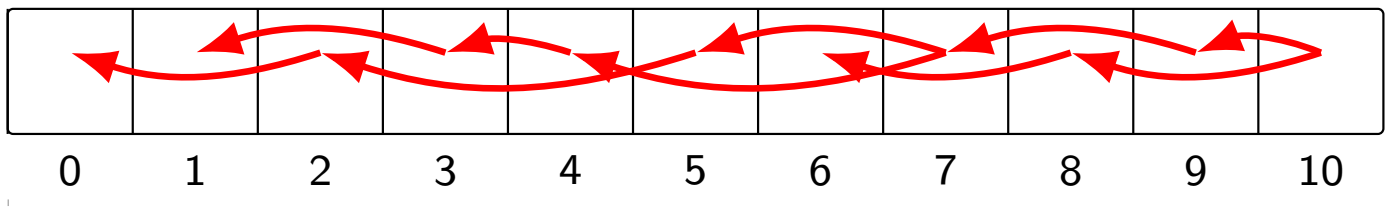
Full Example

Issues



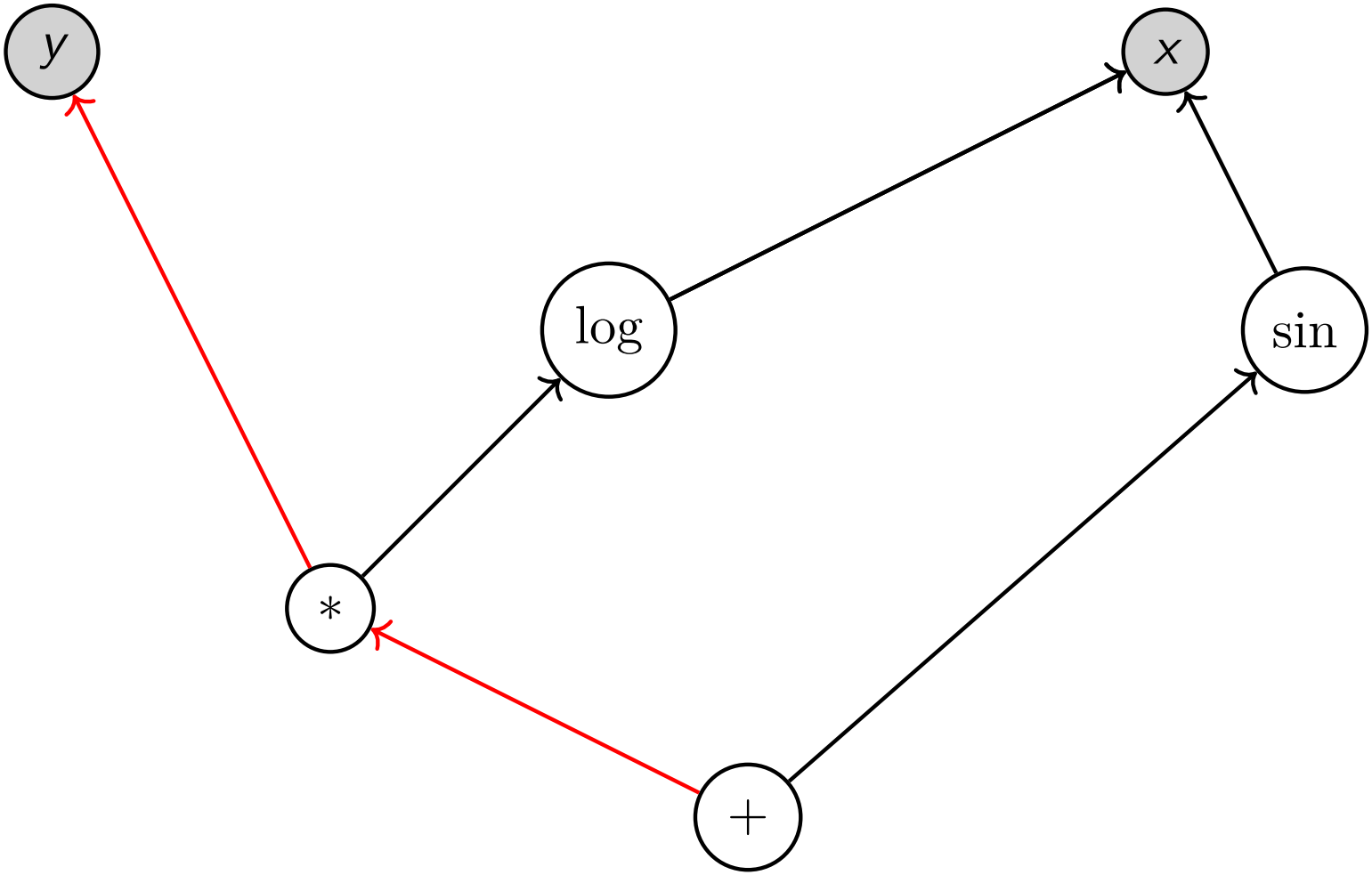
Issues

Contiguous Access Pattern



DepthWise

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



Breadthwise

$$f(x, y) = \log(x)y + \sin(x)$$

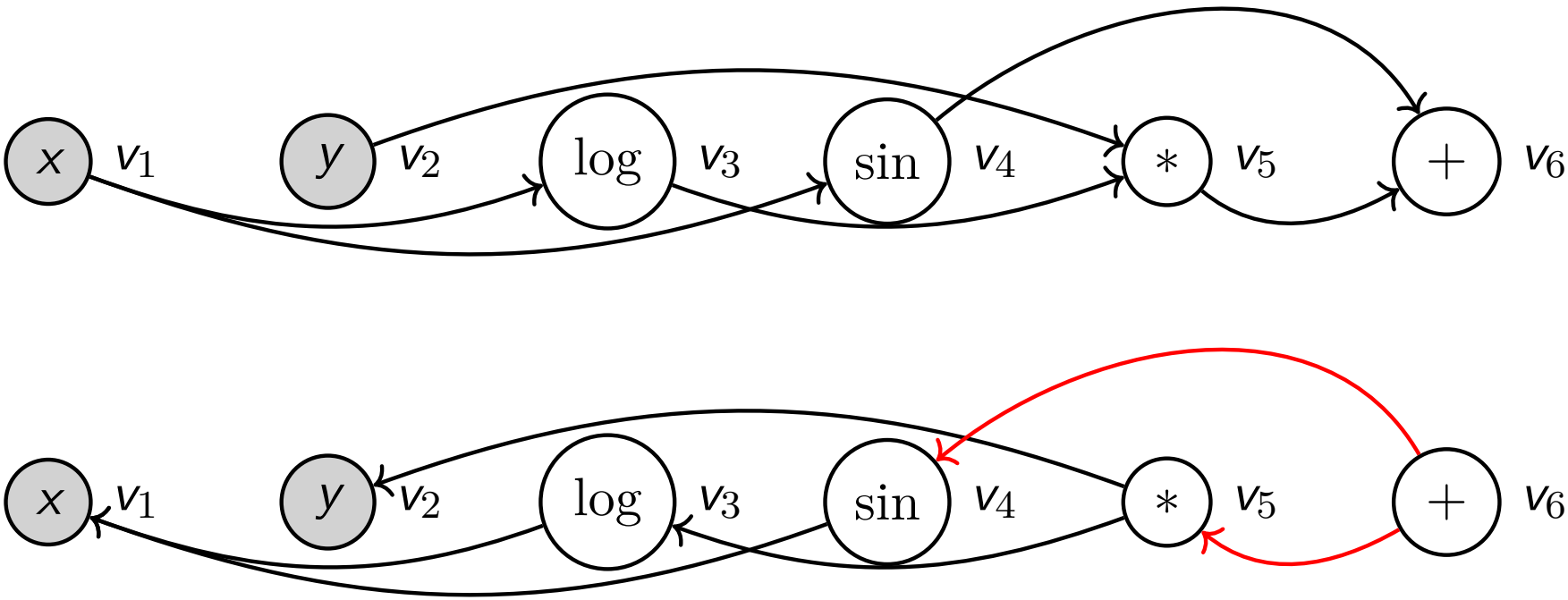
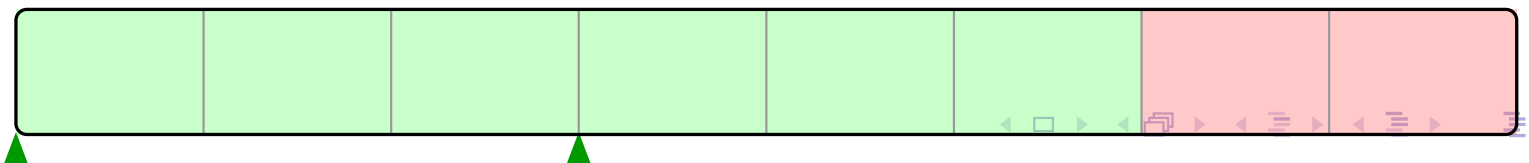


Figure: Topological sort of expression graph

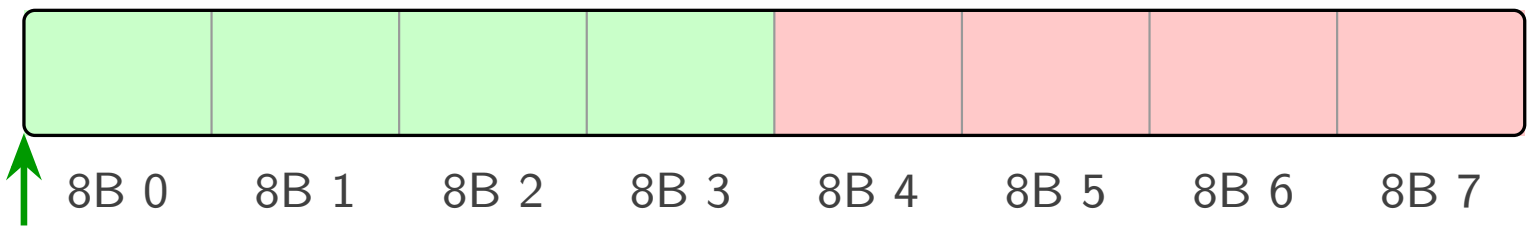
Monotonic Buffer

```
1  static monotonic_buffer_resource mbr{1<<16};
2  static polymorphic_allocator<std::byte> pa{&mbr};
3  static std::vector<vari*> vari_vec;
4  template <typename T>
5  auto& new_vari(double x) {
6    return *vari_vec.emplace_back(pa.new_object<T>(x,
       ↪  0.0));
7  }
8  struct vari {
9    double val;
0    double adj;
1    virtual void chain() {};
2  }; // 24 bytes
3  struct var {
4    vari* vi_;
5    var(double val) :
6    vi_(new_vari<vari>(val)) {}
7  }; // 8 bytes
```



Monotonic Buffer

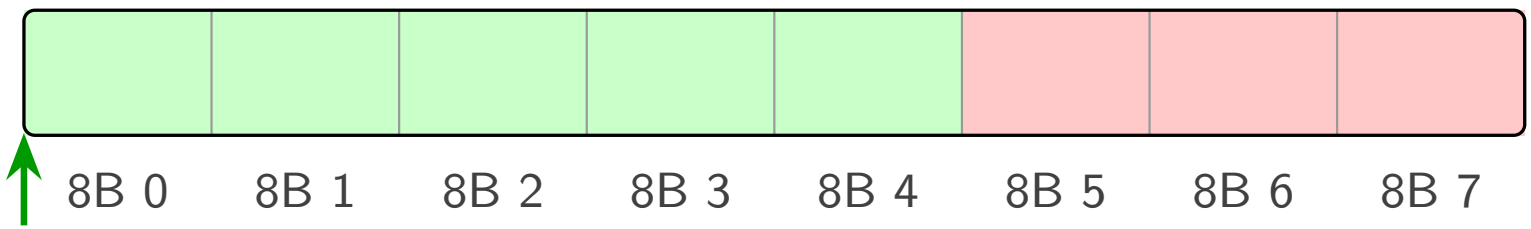
```
1  struct LogVari : vari {
2    vari* in_;
3    LogVari(var x) : in_(std::log(x.vi_->val)) {}
4    virtual void chain() override {
5      in_->adj += adj / in_->val;
6    }
7  }; // 32 bytes
8  inline var log(var x) {
9    return var(new_vari<LogVari>(x));
0  }
```



LogVari{dbl, dbl, vptr, vari*}

Monotonic Buffer

```
1  struct mul_vv : public var_impl {
2    vari* lhs_;
3    vari* rhs_;
4    void chain() {
5      lhs_->adjoint_ += this->adjoint_ * rhs_->value_;
6      rhs_->adjoint_ += this->adjoint_ * lhs_->value_;
7    }
8  }; // 40 bytes
```



AddVari{dbl, dbl, vptr, vari*, vari*}

Monotonic Buffer

```
1  void grad(var z) {
2    z.adj() = 1;
3    for (auto& x : vari_vec | std::views::reverse) {
4      x->chain();
5    }
6  }
7  var x = 1;
8  var y = 2;
9  var z = log(x * y);
0  grad(z);
1  // Do what we want with x and y adjoints then clear
2  vari_vec.clear();
3  mbr.release();
```

Monotonic Buffer

```
1  for (auto& x : vari_vec | std::views::reverse) {
2    x->chain();
3  }
```

Reduce Boilerplate

```
1  template <typename F>
2  struct callback_vari : public vari {
3    F rev_functor_;
4    template <std::floating_point S>
5    explicit callback_vari(S&& value, F&& rev_functor)
6      : vari(value),
7        rev_functor_(rev_functor) {}
8    void chain() final { rev_functor_(*this); }
9  }; // 24 + 8B * N
0  template <std::floating_point T, typename F>
1  auto lambda_var(T val, F&& rev_functor) {
2    return var(
3      new_vari<callback_vari<T, F>>(
4        std::forward<T>(val),
5        std::forward<F>(rev_functor)));
6  }
```

Reduce Boilerplate

```
1  template <float_or_var T1, float_or_var T2>
2  inline auto operator+(T1 lhs, T2 rhs) {
3    return lambda_var(value(lhs) + value(rhs),
4      [lhs, rhs](auto&& ret) mutable {
5        if constexpr (is_var_v<T1>) {
6          adjoint(lhs) += adjoint(ret);
7        }
8        if constexpr (is_var_v<T2>) {
9          adjoint(rhs) += adjoint(ret);
0        }
1      });
2  }
```

Comparison

Table: $f(x, y) = x * \log(y) + \log(x * y) * y$;

Method	CPU Time	% Improvement
Shared Ptr	508ns	1.0
MonoBuff	130ns	3.9x
Lambda	120ns	4.2x
SCT	84ns	6.0x

Operator Overloading Approach: Matrices

```
1  Matrix<var> B(M, M);
2  Matrix<var> X(M, M);
3  Matrix<var> Z = X * B.transpose();
```

- 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

Operator Overloading Approach: Matrices

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

```
1  struct var {
2    double value_;
3    double adjoint_;
4  };
5  struct MatrixVar {
6    var* data_;
7    MatrixVar(std::size_t N) :
8     data_(static_cast<var*>(malloc(sizeof(var) *
       ↪  N))) {}
9  }
0  MatrixVar aos_matrix;
1
2  struct VarMatrix {
3    double* value_;
4    double* adjoint_;
5  }
6  VarMatrix soa_matrix;
```

Source Code Transform Ex:

```
1  double z = log(x * y);
```

Break it down

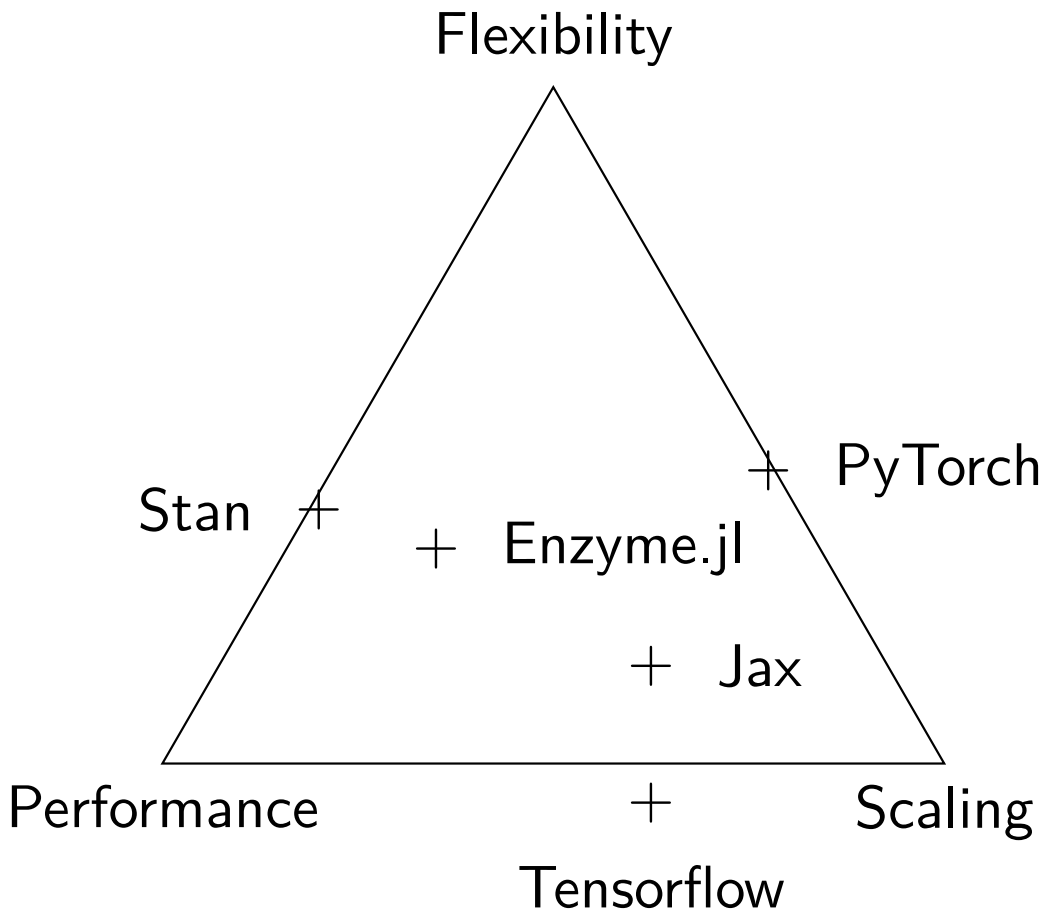
```
1  double v0 = x;
2  double v1 = y;
3  double v2 = x * y;
4  double v3 = log(v2)
5  double bar_v3 = 1;
6  double bar_v2 = bar_v3 * 1/v2;
7  double bar_v1 = bar_v2 * v0;
8  double bar_v0 = bar_v2 * v1;
```

Source Code Transform Ex:

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

```
1  while(error < tolerance) {
2     //  ...
3  }
```

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:

- 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