From Bayesian Inference to LLMs

Modern C++ Optimizations for Reverse-Mode Automatic Differentiation

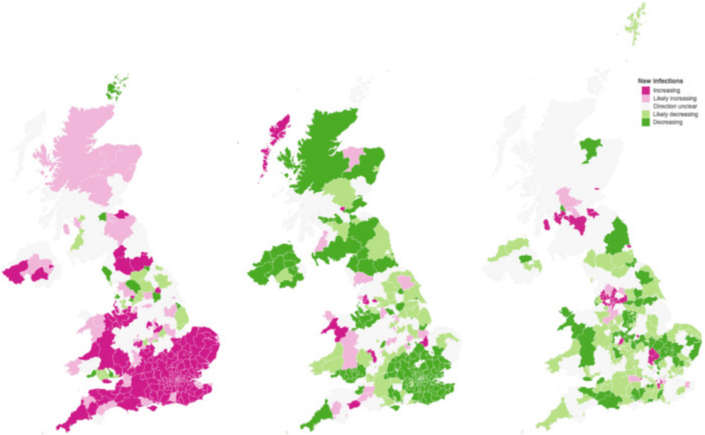
STEVE BRONDER

Cppcon
The C++ Conference

2025

September 13 - 19

Estimating COVID Infection Rates For Policy



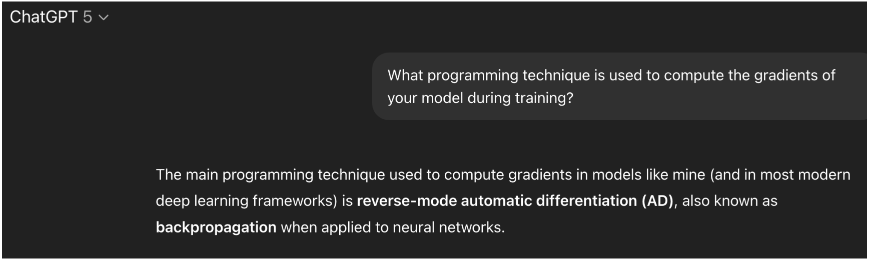
(a) 25 Dec 2020

(b) 21 Jan 2021

(c) 30 May 2021

Figure: Probability of epidemic growth by local area

Automatic Differentiation Affects Your Day to Day



What programming technique is used to compute the gradients of your model during training?

The main programming technique used to compute gradients in models like mine (and in most modern deep learning frameworks) is **reverse-mode automatic differentiation (AD)**, also known as **backpropagation** when applied to neural networks.

What is Automatic Differentiation?

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

Why use Automatic Differentiation

Ex: Newton's root finding method

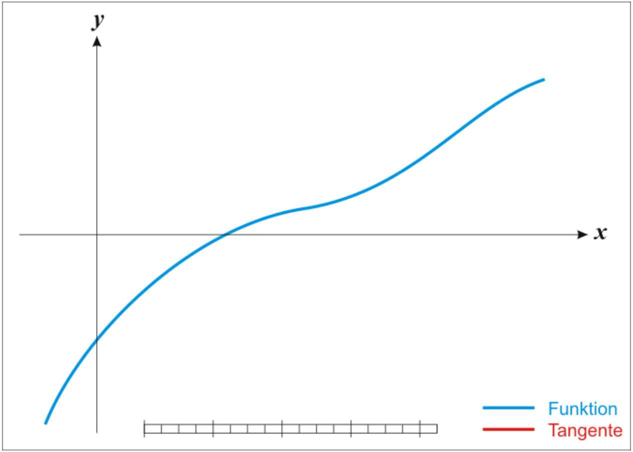
$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

find $f(x) = y$ where $y = 0$

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?

- Hamiltonian Monte Carlo:
  - $\frac{dp}{dt} = -\nabla_\theta \log p(\theta|y)$
- BFGS:
  - $s_k = -H_k \nabla_\theta f(\theta_k)$
- Stochastic Gradient Descent:
  - $\theta_{t+1} = \theta_t - \eta \nabla_\theta L(\theta_t; x_t)$

Why use Automatic Differentiation?

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

Why use Automatic Differentiation?

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

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

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

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

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

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

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

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

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

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

Implementation Matters!

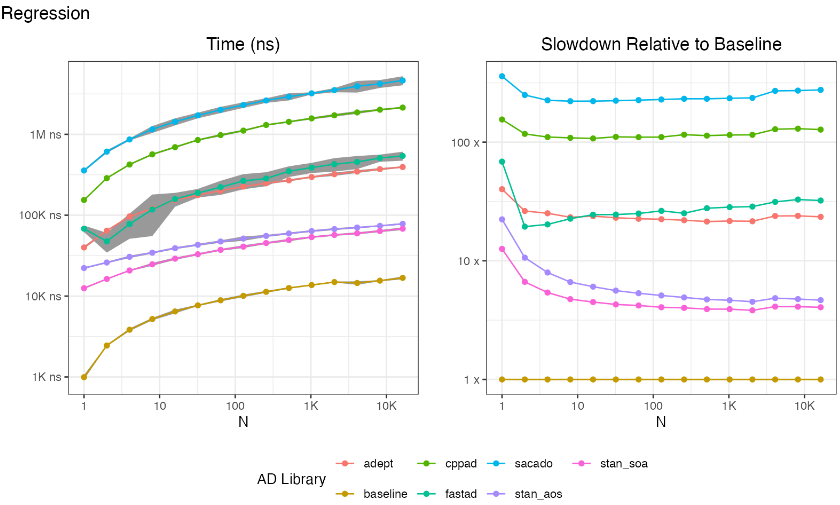


Figure: Benchmark for f' given $f(y) = N(y|X\theta, \sigma)$

Goal of this talk

- Explain how Reverse Mode Automatic Differentiation works
- Performance of high throughput memory intensive programs
- Show how modern C++ has led to cleaner and more efficient AD

What is Automatic Differentiation?

- AD computes gradients of a program by applying the chain rule to its subexpressions.

$$f(x, y) = \log(x \cdot y)$$

$$f_1(x, y) = x \cdot y \qquad \frac{\partial f_1}{\partial x} = y \qquad \frac{\partial f_1}{\partial y} = x$$

$$f_2(u) = \log(u) \qquad \frac{\partial f_2}{\partial u} = \frac{1}{u}$$

$$f(x, y) = f_2\big(f_1(x, y)\big)$$

What is Automatic Differentiation?

- AD computes gradients of a program by applying the chain rule to its subexpressions.

$$f(x, y) = \log(x \cdot y)$$

$$f_1(x, y) = x \cdot y \qquad \frac{\partial f_1}{\partial x} = y \qquad \frac{\partial f_1}{\partial y} = x$$

$$f_2(u) = \log(u) \qquad \frac{\partial f_2}{\partial u} = \frac{1}{u}$$

$$f(x, y) = f_2\big(f_1(x, y)\big)$$

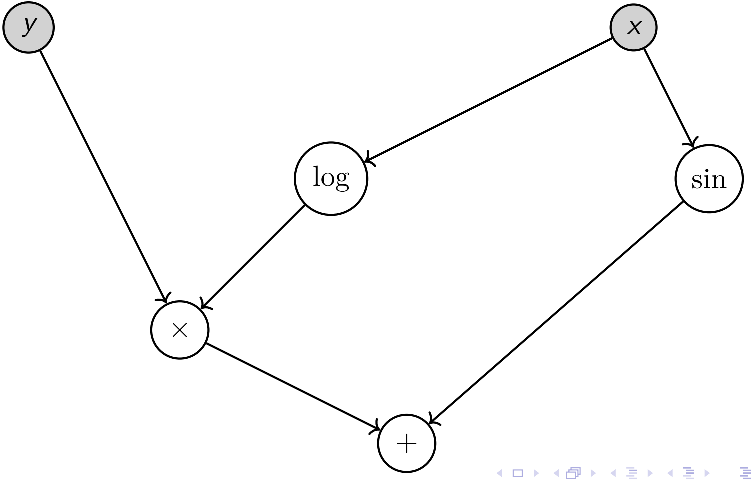
$$\frac{\partial f}{\partial x} = \frac{\partial f_2}{\partial f_1} \cdot \frac{\partial f_1}{\partial x} = \frac{1}{f_1(x, y)} \cdot y = \frac{y}{x \cdot y} = \frac{1}{x},$$

$$\frac{\partial f}{\partial y} = \frac{\partial f_2}{\partial f_1} \cdot \frac{\partial f_1}{\partial y} = \frac{1}{f_1(x, y)} \cdot x = \frac{x}{x \cdot y} = \frac{1}{y}.$$

Data Tye: Expression Graph

Goal: Calculate the full gradient by accumulating partial gradients (adjoints) through the a graph of subexpressions.

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



What is Automatic Differentiation

For Reverse Mode AD, each node performs two functions.

- Forward Pass:

$$v_2 = f(v_0, v_1)$$

What is Automatic Differentiation

For Reverse Mode AD, each node performs two functions.

- Forward Pass:

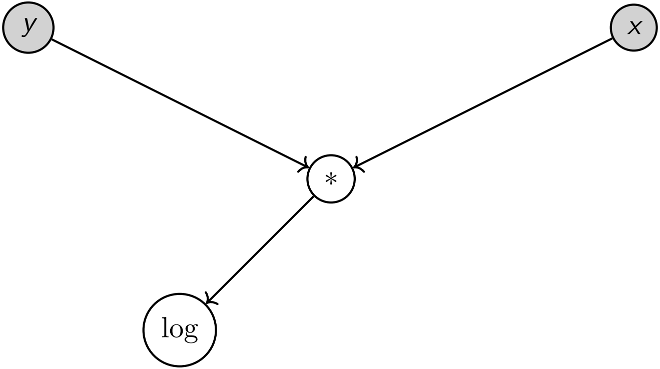
$$v_2 = f(v_0, v_1)$$

- Reverse Pass: Given v_2's adjoint (partial gradient) $\overline{v_2}$ Calculate the local adjoint-Jacobian update for v_0 and v_1.

$$chain(\overline{v_2}, v_0, v_1) = \left\{ \overline{v_0} \mathrel{+}= \frac{\partial v_2}{\partial v_0} \overline{v_2}, \overline{v_1} \mathrel{+}= \frac{\partial v_2}{\partial v_1} \overline{v_2} \right\}$$

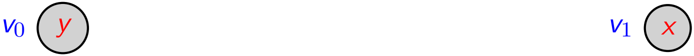
Forward Pass

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



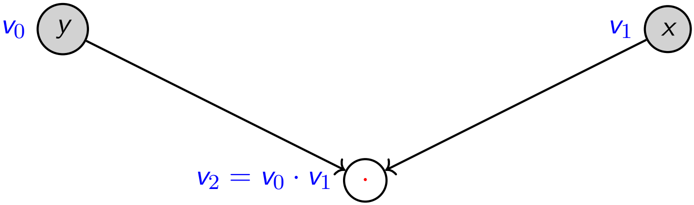
Forward Pass

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



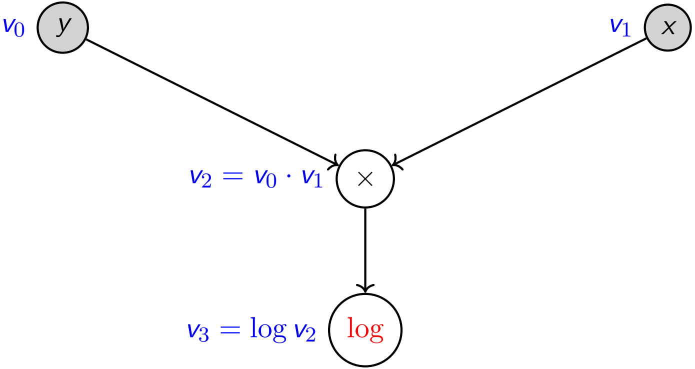
Forward Pass

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



Forward Pass

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



How do we calculate the adjoint Jacobian?

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

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

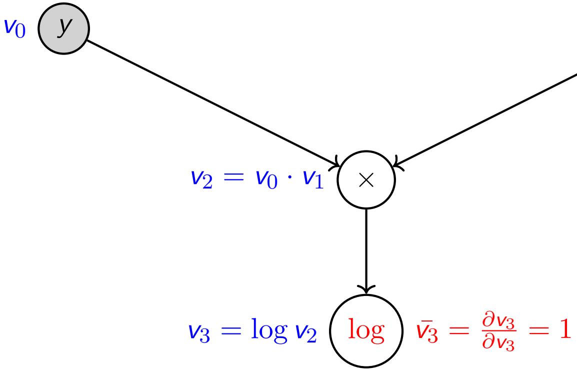
Automatic Differentiation only needs the partials of the intermediates

$$v_2 = v_0 \cdot v_1 \qquad \frac{\partial v_2}{\partial v_0} = v_1, \frac{\partial v_2}{\partial v_1} = v_0$$

$$v_3 = \log(v_2) \qquad \frac{\partial v_3}{\partial v_2} = \frac{1}{v_2}$$

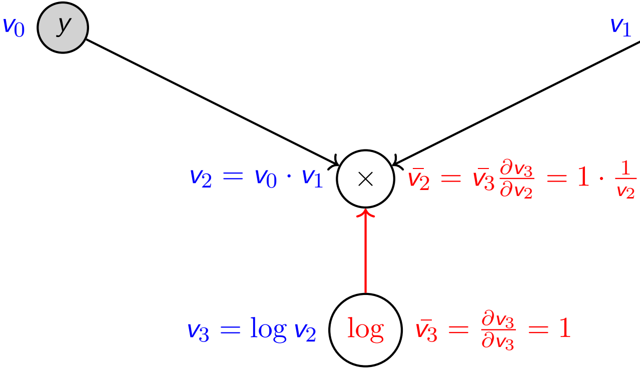
Reverse Pass

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



Reverse Pass

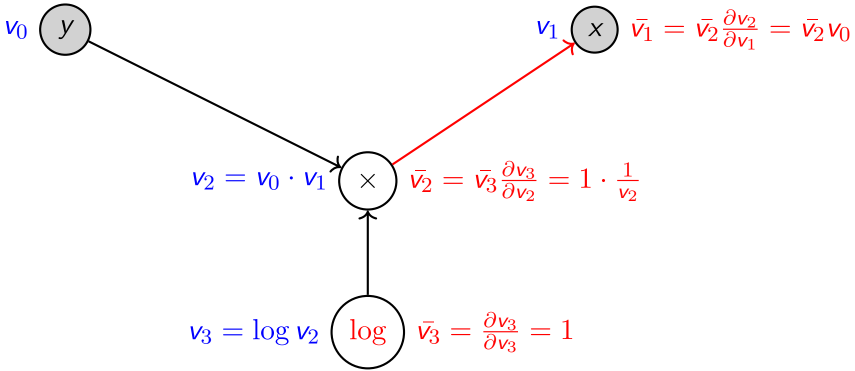
$$z = \log(x \cdot y)$$



v_0
y
v_1
x
$v_2 = v_0 \cdot v_1$
$\times$
$\bar{v_2} = \bar{v_3}\frac{\partial v_3}{\partial v_2} = 1 \cdot \frac{1}{v_2}$
$v_3 = \log v_2$
log
$\bar{v_3} = \frac{\partial v_3}{\partial v_3} = 1$

Reverse Pass

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



v_0

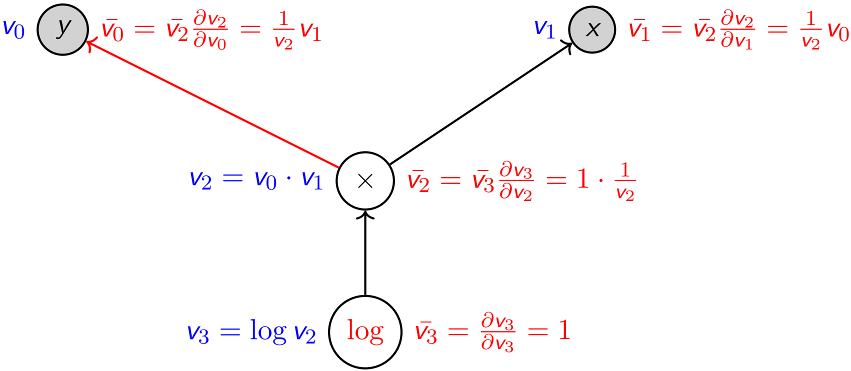
v_1 $\quad \bar{v_1} = \bar{v_2}\frac{\partial v_2}{\partial v_1} = \bar{v_2}\, v_0$

$v_2 = v_0 \cdot v_1$ $\quad \bar{v_2} = \bar{v_3}\frac{\partial v_3}{\partial v_2} = 1 \cdot \frac{1}{v_2}$

$v_3 = \log v_2$ $\quad \bar{v_3} = \frac{\partial v_3}{\partial v_3} = 1$

Reverse Pass

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



v_0 y $\bar{v_0} = \bar{v_2}\frac{\partial v_2}{\partial v_0} = \frac{1}{v_2}v_1$

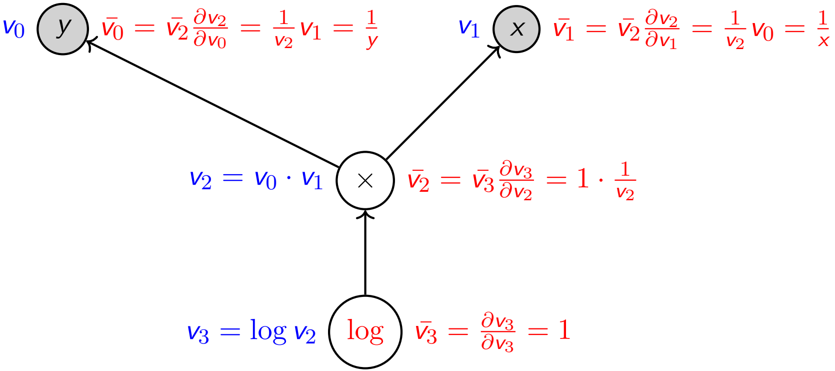
v_1 x $\bar{v_1} = \bar{v_2}\frac{\partial v_2}{\partial v_1} = \frac{1}{v_2}v_0$

$v_2 = v_0 \cdot v_1$ $\times$ $\bar{v_2} = \bar{v_3}\frac{\partial v_3}{\partial v_2} = 1 \cdot \frac{1}{v_2}$

$v_3 = \log v_2$ log $\bar{v_3} = \frac{\partial v_3}{\partial v_3} = 1$

Reverse Pass

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



v_0 y $\bar{v_0} = \bar{v_2}\frac{\partial v_2}{\partial v_0} = \frac{1}{v_2}v_1 = \frac{1}{y}$

v_1 x $\bar{v_1} = \bar{v_2}\frac{\partial v_2}{\partial v_1} = \frac{1}{v_2}v_0 = \frac{1}{x}$

$v_2 = v_0 \cdot v_1$ $\times$ $\bar{v_2} = \bar{v_3}\frac{\partial v_3}{\partial v_2} = 1 \cdot \frac{1}{v_2}$

$v_3 = \log v_2$ log $\bar{v_3} = \frac{\partial v_3}{\partial v_3} = 1$

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
- 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

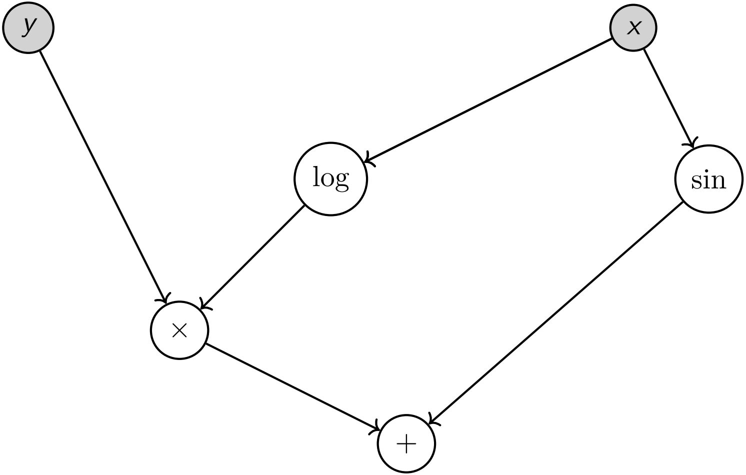
How do we keep track of our reverse pass?

Static (Fast) vs. Dynamic (Flexible) graphs

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

Make A Tape

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



Tape of Expression Graph

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

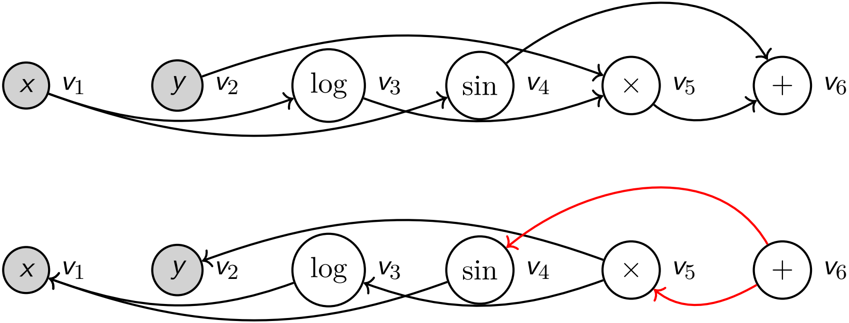


Figure: Topological sort of expression graph

Operator Overloading Approach

The operator overloading approach usually involves:

- Tracking the expression graph of reverse pass function calls
- A pair scalar type to hold the value and adjoint

Allows conditional loops and reassignment of values in matrices
Nodes of expression graph can be collapsed

Example Godbolt

Operator Overloading: Simple

```
struct var_impl {
  double val_;
  double adj_;
  virtual void chain() {}
  var_impl(double val) : val_(val), adj_(0.0) {}
};
static std::vector<std::shared_ptr<var_impl>> tape;
struct var {
  std::shared_ptr<var_impl> vi_;
  var(std::shared_ptr<var_impl>& vi) : vi_(vi) {
    tape.push_back(vi);
  }
};
```

Operator Overloading: Simple

```
struct mul_vv final : public var_impl {
  var op1_;
  var op2_;
  mul_vv(double val, var op1, var op2) :
   ↪   var_impl(val), op1_(op1), op2_(op2) {}
  void chain() {
    op1_.adj() += op2_.val() * this->adjoint_;
    op2_.adj() += op1_.val() * this->adjoint_;
  }
};
operator*(var x, var y) {
  return var{
    std::make_shared<mul_vv>(
      x.val() * y.val(), x, y)};
}
```

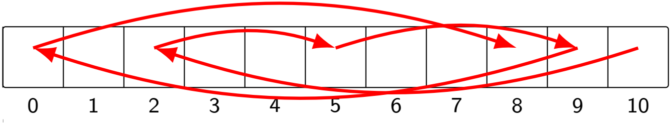
Operator Overloading: Simple

```cpp
void grad(var& v) {
  v.adj() = 1.0;
  for (auto&& x : tape | std::views::reverse) {
    x->chain();
  }
}
var x(2.0);
var y(4.0);
auto z = x;
while (value(z) < 10) {
  z += x * log(y) + log(x * y) * y;
}
grad(z);
```

Full Example

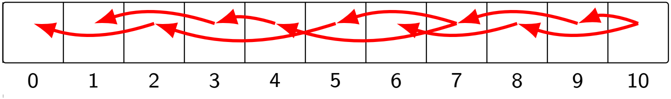
Issues

Noncontiguous Access Pattern



Issues

Contiguous Access Pattern



Monotonic Buffer

```
struct Tape {
  monotonic_buffer_resource mbr{1<<16};
  polymorphic_allocator<std::byte> pa{&mbr};
  std::vector<var_impl*> tape;
  void clear() {
    tape.clear();
    mbr.release();
  }
  template <typename T, typename Types>
  auto* add(Types&&... args) {
    return g_ad.tape.emplace_back(
      g_ad.pa.new_object<T>(args...));
  }
};
static Tape g_ad{};
```

Monotonic Buffer

```
struct var_impl {
  double val;
  double adj;
  virtual void chain() {};
}; // 24 bytes
struct var {
  var_impl* vi_;
  var(double val) :
    vi_(g_ad.template add<var_impl>(val)) {}
}; // 8 bytes
```

Monotonic Buffer

```cpp
struct mul_vv : public var_impl {
  var op1_;
  var op2_;
  void chain() {
    op1_->adjoint_ += this->adjoint_ * op2_->value_;
    op2_->adjoint_ += this->adjoint_ * op1_->value_;
  }
}; // 40 bytes
operator*(var op1, var op2) {
  return var{g_ad.template add<mul_vv>(
    op1.val() * op2.val(), op1, op2)};
}
```

Monotonic Buffer

```
void compute_grads(var x, var y) {
  var z = -10;
  while (z.val() < 20) {
    z += log(x * y);
  }
  grad(z);
}
// Later in program
for (int i = 0; i < 1e10; ++i) {
  var x = compute_x(...);
  var y = compute_y(...);
  compute_grads(x, y);
  do_something_with_grads(x.adj(), y.adj());
  tape.clear();
  mbr.release();
}
```

So Many Operator Classes

```
struct mul_vv;
struct mul_vd;
struct mul_dv;
struct add_vv;
struct add_dv;
struct add_vd;
struct subtract_vv;
struct subtract_dv;
struct subtract_vd;
struct divide_vv;
struct divide_dv;
struct divide_vd;
```

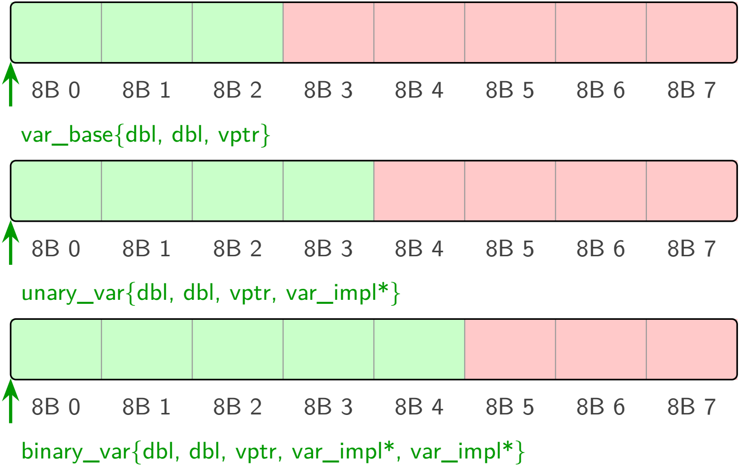
Reduce Boilerplate

```
template <typename F>
struct callback_var_impl : public var_impl {
  F rev_functor_;
  template <std::floating_point S>
  explicit callback_var_impl(S&& value,
    F&& rev_functor)
    : var_impl(value),
      rev_functor_(rev_functor) {}
  void chain() final { rev_functor_(*this); }
}; // 24 + 8B * N
// Helper for callback_var_impl
template <std::floating_point FwdVal, typename F>
auto lambda_var(FwdVal val, F&& rev_functor) {
  return var(
    new_var_impl<callback_var_impl<F>>(
      val,
      std::forward<F>(rev_functor)));
}
```

Reduce Boilerplate

```
template <typename T1, typename T2>
requires any_var<T1, T2>
inline auto operator*(T1 op1, T2 op2) {
  return lambda_var(value(op1) * value(op2),
    [op1, op2](auto&& ret) mutable {
      if constexpr (is_var_v<T1>) {
        adjoint(op1) += adjoint(ret) * value(op2);
      }
      if constexpr (is_var_v<T2>) {
        adjoint(op2) += adjoint(ret) * value(op1);
      }
    });
}
```

Poor Cache Use



var_base{dbl, dbl, vptr}

unary_var{dbl, dbl, vptr, var_impl*}

binary_var{dbl, dbl, vptr, var_impl*, var_impl*}

Source Code Transform Ex:

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

Break it down

```
double v0 = x;
double v1 = y;
double v2 = x * y;
double v3 = log(v2)
double bar_v3 = 1;
double bar_v2 = bar_v3 * 1/v2;
double bar_v1 = bar_v2 * v0;
double bar_v0 = bar_v2 * v1;
```

Source Code Transform Ex:

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

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

Source Code Transform Ex:

```
struct var {
    double values_;
    double adjoints_;
    var(double x) : values_(x), adjoints_(0) {}
};
```

Source Code Transform Ex:

```
template <typename F, typename... Exprs>
struct expr {
   var ret_;
   std::tuple<deduce_ownership_t<Exprs>...> exprs_;
   std::decay_t<F> f_;
   template <typename FF, typename... Args>
   expr(double x, FF&& f, Args&&... args) :
     ret_(x), f_(std::forward<F>(f)),
     exprs_(std::forward<Args>(args)...) {}
};
```

Source Code Transform Ex:

```cpp
template <typename T1, typename T2>
requires any_var_or_expr<T1, T2>
inline auto operator*(T1&& lhs, T2&& rhs) {
  return make_expr(value(lhs) * value(rhs),
  [](auto&& ret, auto&& lhs, auto&& rhs) {
    if constexpr (!std::is_arithmetic_v<T1>) {
      adjoint(lhs) += adjoint(ret) * value(rhs);
    }
    if constexpr (!std::is_arithmetic_v<T2>) {
      adjoint(rhs) += adjoint(ret) * value(lhs);
    }
  }, std::forward<T1>(lhs), std::forward<T2>(rhs));
}
```

Source Code Transform Ex:

```cpp
template <typename Expr>
inline void grad(Expr&& z) {
  adjoint(z) = 1.0;
  auto nodes = collect_bfs(z);
  eval_breadthwise(nodes);
}
```

Source Code Transform Ex:

```
auto z = x * log(y) + log(x * y) * y;
expr<Lambda<Plus>,
  expr<Lambda<Mult>, var, expr<Lambda<Log>, var>>,
  expr<Lambda<Mult>,
   expr<Lambda<Log>, expr<Lambda<Mult>, var, var>>,
   var>>
```

Comparison

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

Method	CPU Time	% Improvement
Shared Ptr	508ns	1.0
MonoBuff	121ns	3.9x
Lambda	112ns	4.2x
Source Code Transform	26.5ns	19x
Baseline	2.82ns	180x

Operator Overloading Approach: Matrices

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

Operator Overloading Approach: Matrices

```
template <typename MatrixType>
struct arena_matrix :
  public Eigen::Map<MatrixType> {
  using Base = Eigen::Map<MatrixType>
  template <typename T>
  arena_matrix(T&& mat) :
   Base(copy_to_arena(mat.data(), mat.size()),
     mat.rows(), mat.cols()) {}
};
```

Operator Overloading Approach: Matrices

```cpp
// Array of Structs
struct Matrix<var> {
  var* data_;
};
// Struct of Arrays
struct var_impl<Matrix<double>> {
  arena_matrix<double> value_;
  arena_matrix<double> adjoint_;
  virtual void chain() {}
};
```

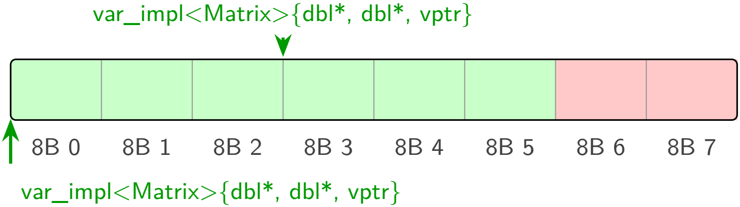
Operator Overloading Approach: Matrices

- Array of Structs `Matrix<var>`:
  - Simple, most algorithms Just Work™
  - Adds a lot to expression graph
  - turns off SIMD

Operator Overloading Approach: Matrices

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

Operator Overloading Approach: Matrices



var_impl<Matrix>{dbl*, dbl*, vptr}

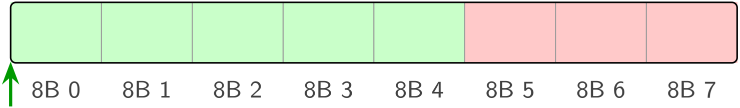
8B 0 8B 1 8B 2 8B 3 8B 4 8B 5 8B 6 8B 7

var_impl<Matrix>{dbl*, dbl*, vptr}

Matrix Multiplication Example

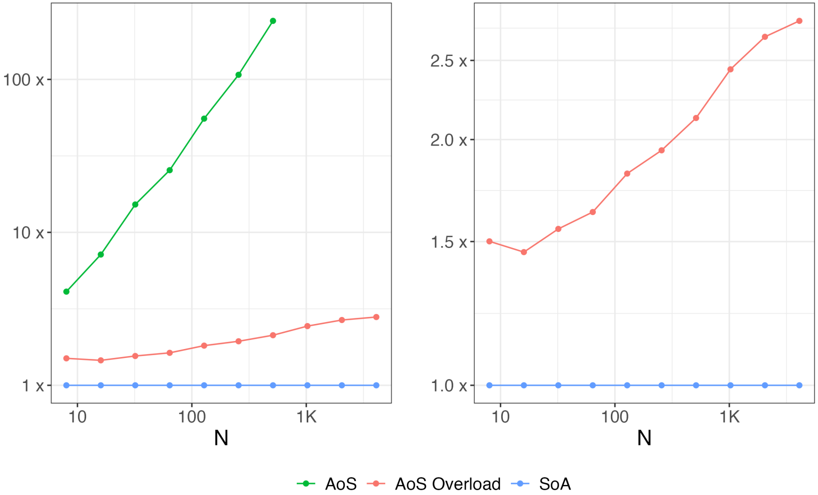
```
template <typename T1, typename T2>
requires any_var<T1, T2>
inline auto operator*(T1&& op1, T2&& op2) {
  return lambda_var(value(op1) * value(op2),
    [op1, op2](auto&& ret) mutable {
      if constexpr (is_var_matrix_v<T1>) {
        adjoint(op1) += adjoint(ret) *
         ↪  value(op2).transpose();
      }
      if constexpr (is_var_matrix_v<T2>) {
        adjoint(op2) += value(op1) * adjoint(ret);
      }
    });
}
```

Operator Overloading Approach: Matrices



mat_mul_vv{dbl*, dbl*, vptr, lambda[var_impl<Matrix>*, var_impl<Matrix>*]}

Matrix Multiplication Benchmark



Scaling Up: CPU

- Within a node, parallelism is easy

Scaling Up: CPU

- Within a node, parallelism is easy
- Parallel + static globals????

Scaling Up: CPU

- Within a node, parallelism is easy
- Parallel + static globals????
- Yes,

Scaling Up: CPU

- Within a node, parallelism is easy
- Parallel + static globals????
- Yes, sometimes

Embarassingly Parallel Not Possible

But reduce sum style very easy