

Differentiable Programming in C++

VASSIL VASSILEV, WILLIAM MOSES



20
21



Speakers



Vassil Vassilev,
Research Software Engineer,
Princeton/CERN



William Moses,
Ph.D. Candidate, MIT

What is this talk about?



$$f(x)$$



Outline

- A warmup
 - Measuring the rate of change
- Introduction
 - Computing derivatives. Approaches
 - A gentle introduction to AD. Chain rule
 - Applications using AD
- Differentiable Programming
 - Deep learning & AD
 - Backpropagation
 - Existing tools & Frameworks
- Implementation
 - Discuss possible implementation approaches
 - Showcase tools built as part of the Clang/LLVM compiler toolchain.
 - Explain how such tools work and what are the benefits
- Briefly outline standardization efforts (as per <https://wg21.link/P2072>)
- Conclusion



How fast he ran?

How fast he ran? What does that even mean?

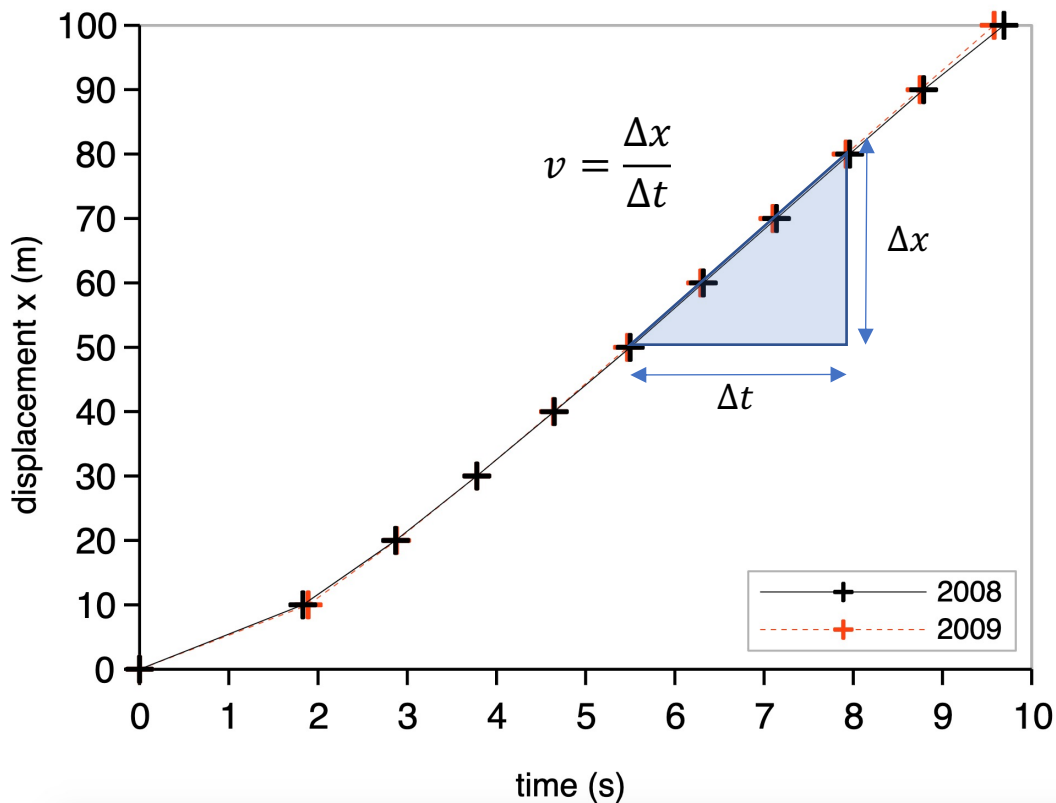
Displacement = velocity * time

$100/9.58 = 10.44 \text{ m/s} \Rightarrow 37.58 \text{ km/h}$ on average

- Did he accelerate until the end?
- When did he slow down?
- What was his top speed?

Measuring the rate of change

Usain Bolt 100m stats - (t,x) graph



Plot credits: A. Penev

To find the time and velocity at some interval we could calculate the *gradient* graph at *different* times.

Data from SportEndurance.com

Bolt (m)	2008 (s)	2009 (s)
0	0	0
10	1.83	1.89
20	2.87	2.88
30	3.78	3.78
40	4.65	4.64
50	5.5	5.47
60	6.32	6.29
70	7.14	7.1
80	7.96	7.92
90	8.79	8.75
100	9.69	9.58

For example the velocity of Bolt from the 50th to the 80th meter was:

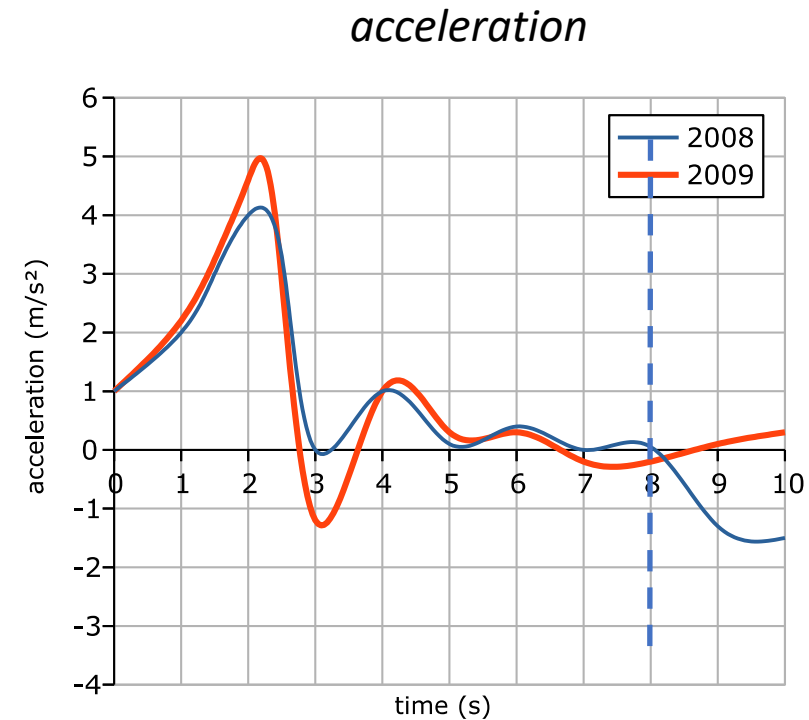
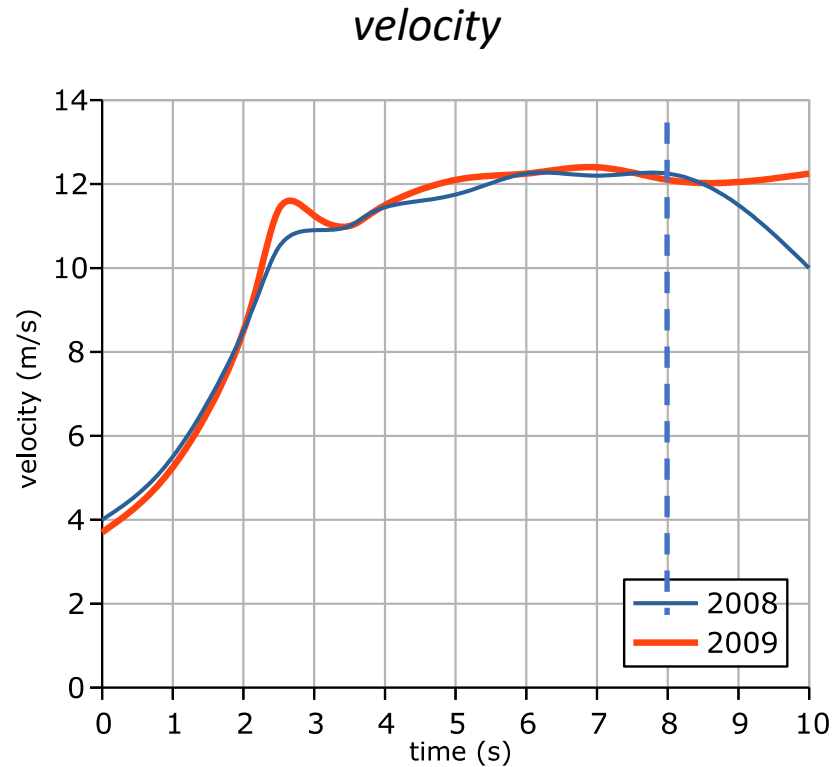
$$v = \frac{\Delta x}{\Delta t} = \frac{80 - 50}{7.96 - 5.5} = 12.19 \text{ m/s}$$

Could he do better in 2009?



Bolt, 100m dash, Beijing Olympics, 2008, source quantamagazine.org

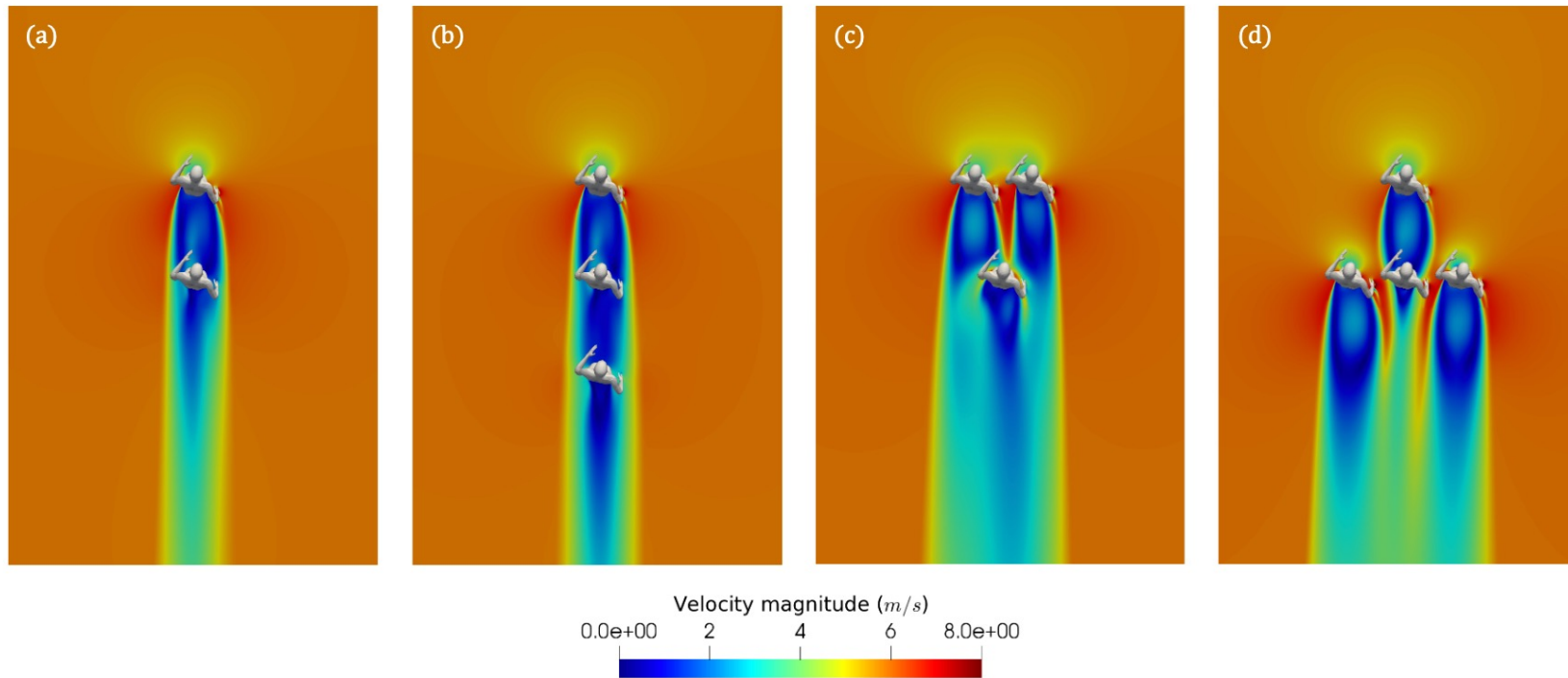
Derivatives: measure the rate of change



A derivative measures the rate of a function's output value wrt a change in its input:

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(a+h) - f(a)}{h}$$

The longer the distance the more parameters



Tactics are skills required in a competition that allow a player or team to effectively use their talent and skill to the best possible advantage. Usually means to empirically develop an intuition how to win and apply it.

Building a reference trajectory with a goal of maximizing performance (output) while minimizing the set of inputs.

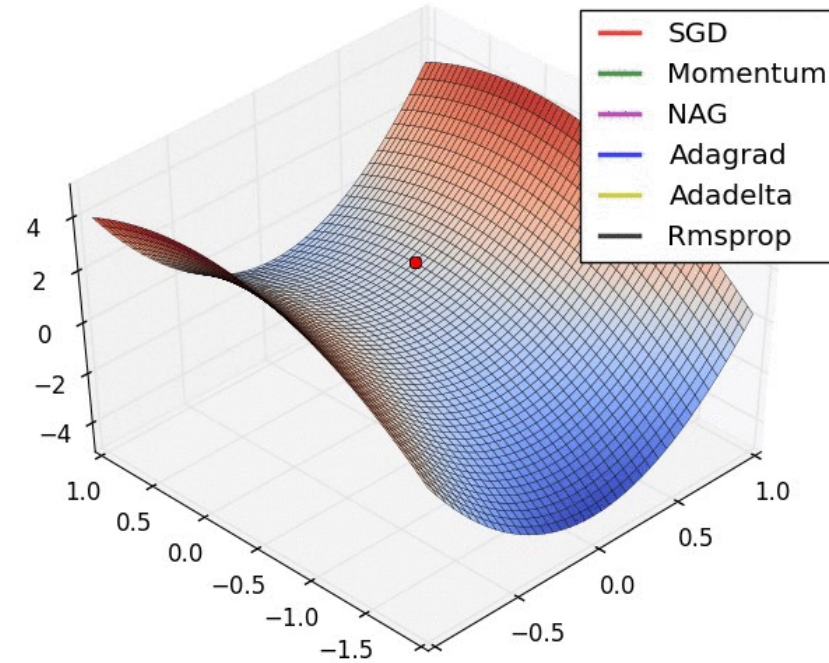
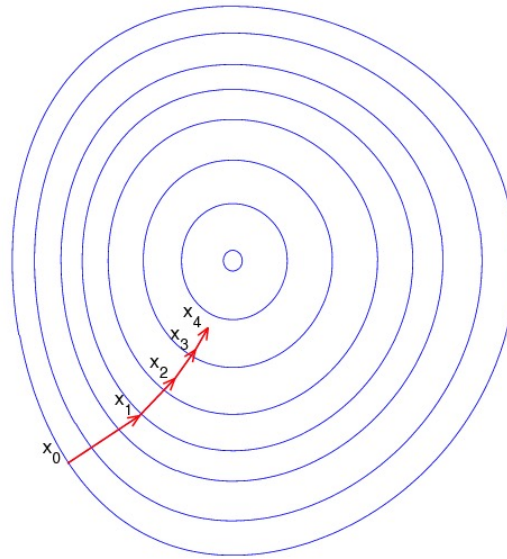
Thus, we need to know how each input parameter affects the output.

Schickhofer, Lukas, and Henry Hanson. "Aerodynamic effects and performance improvements of running in drafting formations." *Journal of Biomechanics* 122 (2021): 110457.

Gradient Descent

A gradient is the vector of values of the function; each entry is the output of the function's derivative wrt a parameter...

$$\nabla f(x_1, \dots, x_n) = \begin{bmatrix} \frac{\partial f}{\partial x_1}(x_1, \dots, x_n) \\ \vdots \\ \frac{\partial f}{\partial x_n}(x_1, \dots, x_n) \end{bmatrix}$$



Plot credits: <https://ruder.io/optimizing-gradient-descent/>

The gradient vector can be interpreted as the "direction and rate of fastest increase"

Computing Derivatives

Computing Derivatives

Manual

- Error prone

Numerical Differentiation (ND)

- Precision errors
- High computational complexity
- Higher order problem (formula approximated by missing higher order terms)

Symbolic Differentiation (SD)

- Only works on single mathematical expressions (no control flow)
- May require transcribing result back into code

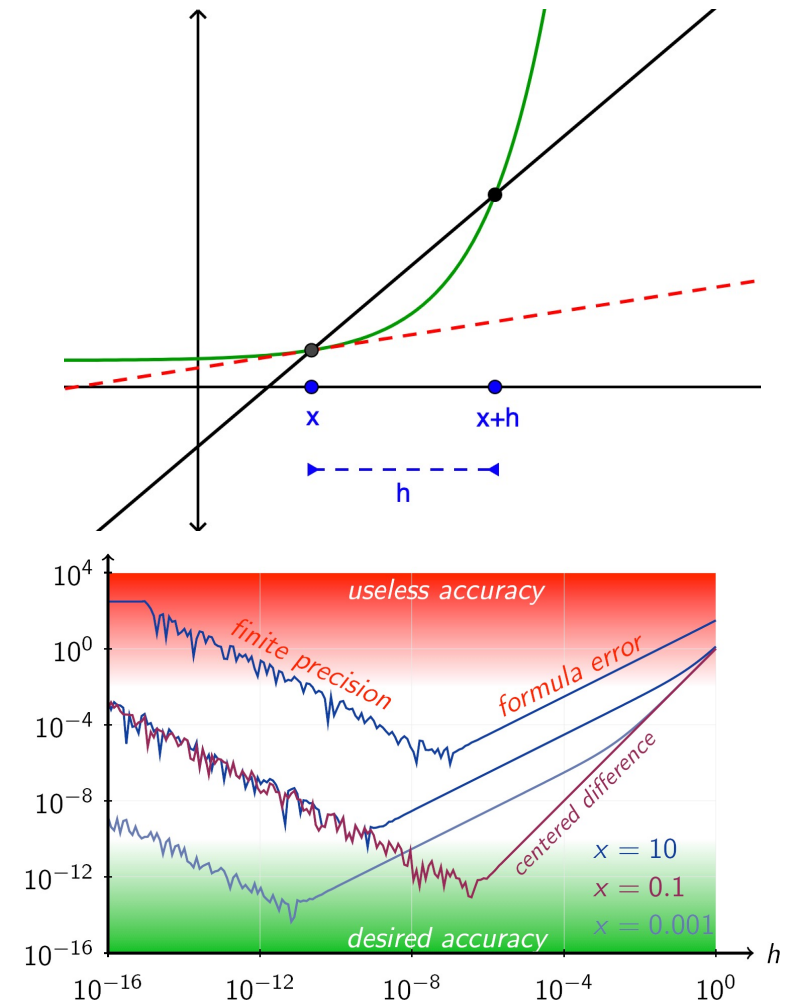
Algorithmic or Automatic Differentiation (AD)

- Automatically generate a C++ program to compute the derivative of a given function

Numerical Differentiation

$$\frac{df(x)}{dx} \approx \frac{f(x) - f(x+h)}{h}$$

- The choice of h is problem-dependent.
- Too big step h makes the approximation too poor
- Too small h makes the floating point round-off error too big
- The computational complexity is $O(n)$, where n is the number of parameters – for a function with 100 parameters we need 101 evaluations



Symbolic Differentiation

- Limited to closed form expressions
- Requires a symbolic processing system (eg Mathematica, Maple) and transcribing back the algorithm
- Suffers from expression swell (subexpression accumulation), especially challenging when going to higher order derivatives

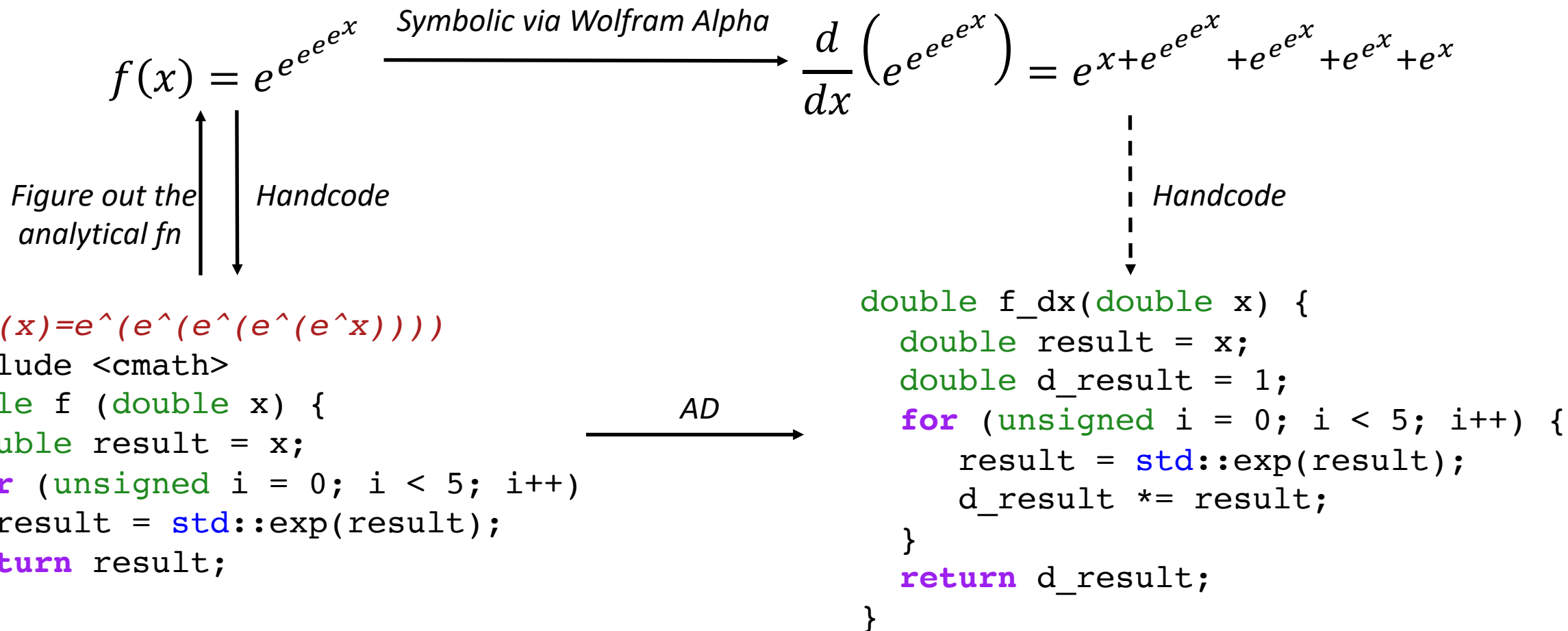
```
// Supports  
double pow3(double x) {  
    return x * x * x;  
}  
  
// Does not support  
double pow3_(double x) {  
    if (x == 0) return 0;  
    return x * x * x;  
}
```

Automatic Differentiation

”[AD] is a set of techniques to evaluate the derivative of a function specified by a computer program. AD exploits the fact that every computer program, no matter how complicated, executes a sequence of elementary arithmetic operations (addition, subtraction, multiplication, division, etc.) and elementary functions (exp, log, sin, cos, etc.).” [Wikipedia]

Known as algorithmic differentiation, autodiff, algodiff, computational differentiation.

Automatic and Symbolic Differentiation



AD. Chain Rule

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$$

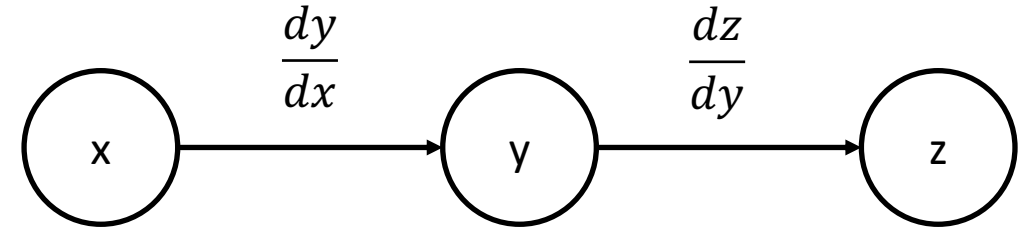
Intuitively, the chain rule states that knowing the instantaneous rate of change of z relative to y and that of y relative to x allows one to calculate the instantaneous rate of change of z relative to x as the product of the two rates of change.

“if a car travels twice as fast as a bicycle and the bicycle is four times as fast as a walking man, then the car travels $2 \times 4 = 8$ times as fast as the man.” G. Simmons

AD. Algorithm Decomposition

$$\begin{aligned}y &= f(x) \\ z &= g(y)\end{aligned}$$

$$\begin{aligned}dydx &= dfdx(x) \\ dzdy &= dgdy(y) \\ dzdx &= dzdy * dydx\end{aligned}$$

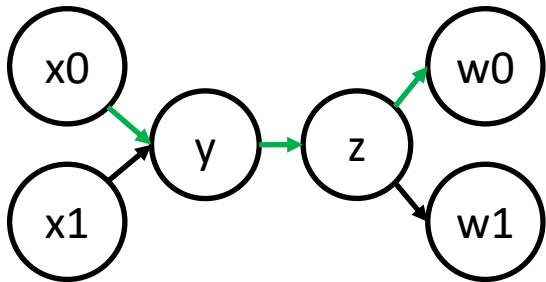
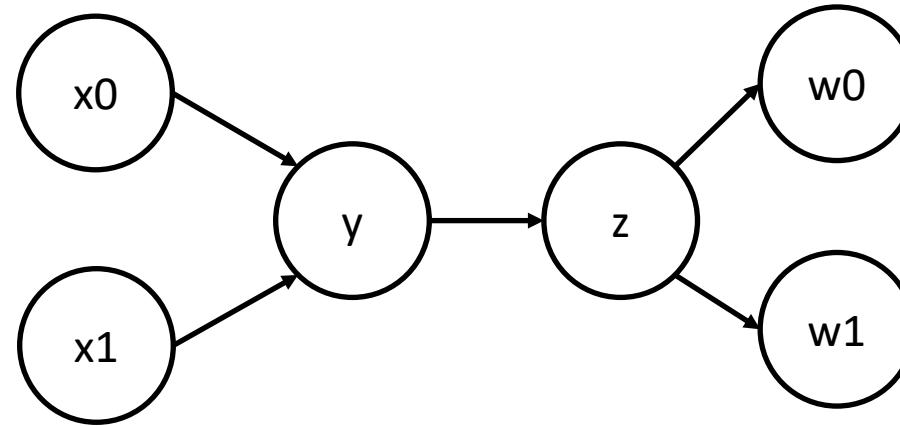


In the computational graph each node is a variable and each edge is derivatives between adjacent edges

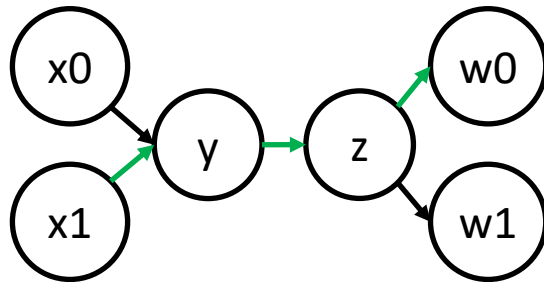
We recursively apply the rules until we encounter an elementary function such as addition, multiplication, division, sin, cos or exp.

AD. Chain Rule

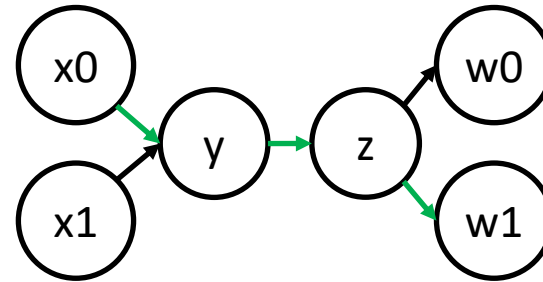
$$\begin{aligned} y &= f(x_0, x_1) \\ z &= g(y) \\ w_0, w_1 &= l(z) \end{aligned}$$



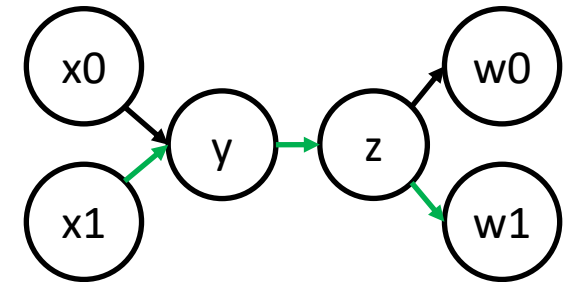
$$\frac{\partial w_0}{\partial x_0} = \frac{\partial w_0}{\partial z} \frac{\partial z}{\partial y} \frac{\partial y}{\partial x_0}$$



$$\frac{\partial w_0}{\partial x_1} = \frac{\partial w_0}{\partial z} \frac{\partial z}{\partial y} \frac{\partial y}{\partial x_1}$$



$$\frac{\partial w_1}{\partial x_0} = \frac{\partial w_1}{\partial z} \frac{\partial z}{\partial y} \frac{\partial y}{\partial x_0}$$



$$\frac{\partial w_1}{\partial x_1} = \frac{\partial w_1}{\partial z} \frac{\partial z}{\partial y} \frac{\partial y}{\partial x_1}$$

AD step-by-step. Forward Mode

$\frac{dx_0}{dx} = \{1, 0\}$

$\frac{dx_1}{dx} = \{0, 1\}$

$y = f(x_0, x_1)$

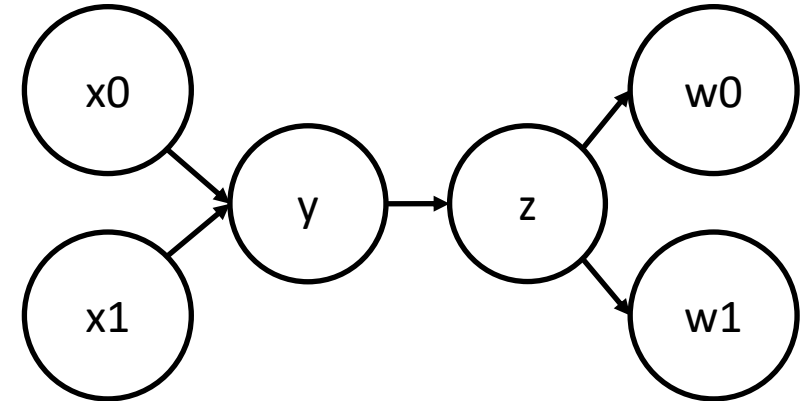
$\frac{dy}{dx} = df(x_0, \frac{dx_0}{dx}, x_1, \frac{dx_1}{dx})$

$z = g(y)$

$\frac{dz}{dx} = dg(y, \frac{dy}{dx})$

$w_0, w_1 = l(z)$

$\frac{dw_0}{dx}, \frac{dw_1}{dx} = dl(z, \frac{dz}{dx})$



AD step-by-step. Reverse Mode

```
y = f(x0, x1)
```

```
z = g(y)
```

```
w0, w1 = l(z)
```

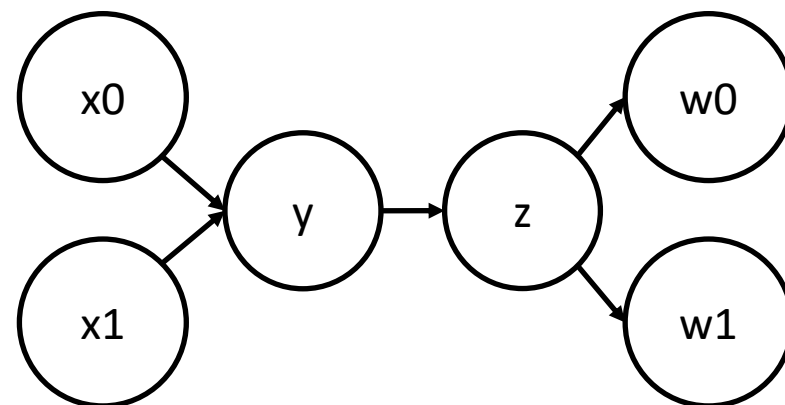
```
dwdw0 = {1, 0}
```

```
dwdw1 = {0, 1}
```

```
dwdz = dl(dwdw0, dwdw1)
```

```
dwdy = dg(y, dwdz)
```

```
dwx0, dwx1 = df(x0, x1, dwdy)
```



AD Control Flow

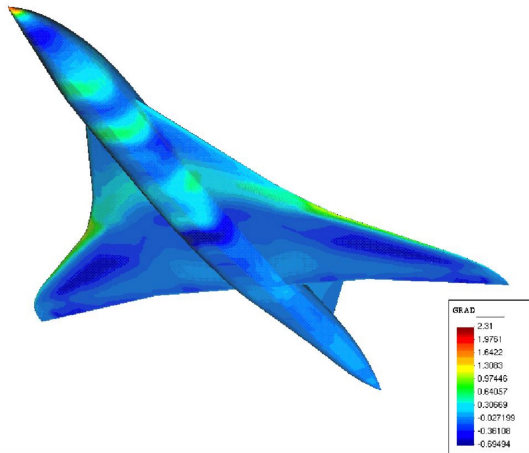
- Control Flow and Recursion fall naturally in forward mode.
- Extra work is required for reverse mode in reverting the loop and storing the intermediaries.

```
double f_reverse (double x) {  
    double result = x;  
    std::stack<double> results;  
    for (unsigned i = 0; i < 5; i++) {  
        results.push(result);  
        result = std::exp(result);  
    }  
    double d_result = 1;  
    for (unsigned i = 5; i; i--) {  
        d_result *= std::exp(results.top());  
        results.pop();  
    }  
    return d_result;  
}
```

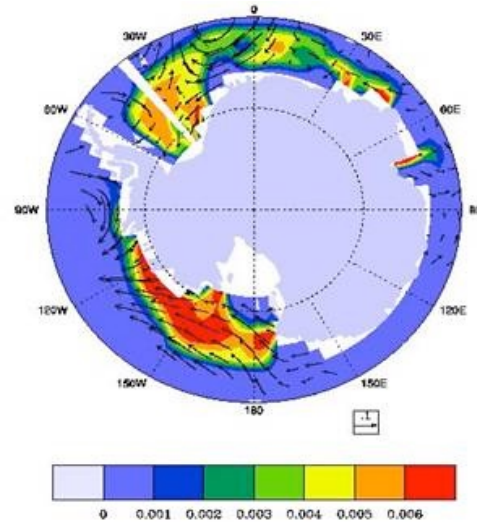
AD. Cheap Gradient Principle

- The computational graph has **common subpaths** which can be precomputed
- If a function has a single input parameter, no matter how many output parameters, **forward mode** AD generates a **derivative** that has the **same time complexity** as the original function
- More importantly, if a function has a **single output** parameter, **no matter how many input** parameters, reverse mode AD generates **derivative** with the **same time complexity** as the original function.

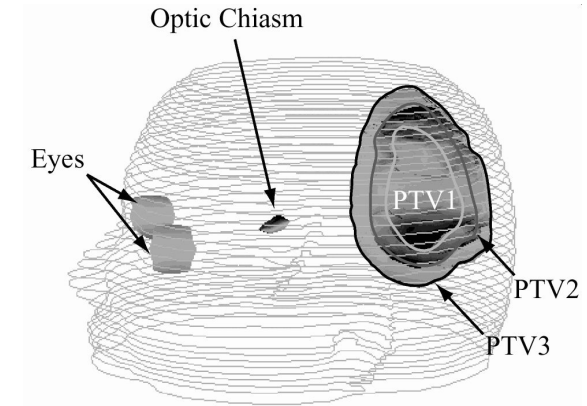
Uses of AD outside of Deep Learning



Gradient of the Sonic Boom objective function
on the skin of the plane, CFD, Laurent Hascoët
et al.



Sensitivities of a Global Sea-Ice Model, Climate, Jong G. Kim et al



Intensity Modulated Radiation
Therapy, Biomedicine, Kyung-Wook
Jee et al

Differentiable Programming

Deep Learning & Automatic Differentiation



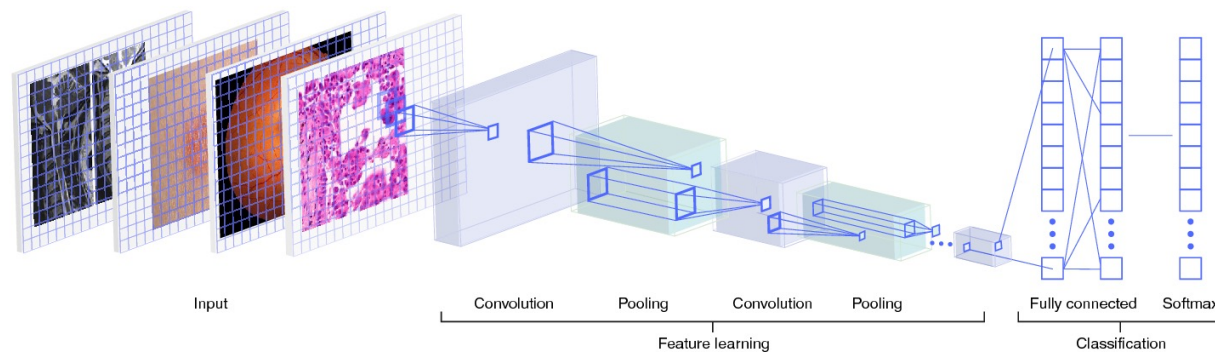
Imagined by GAN,
ThisPersonDoesNotExist.com



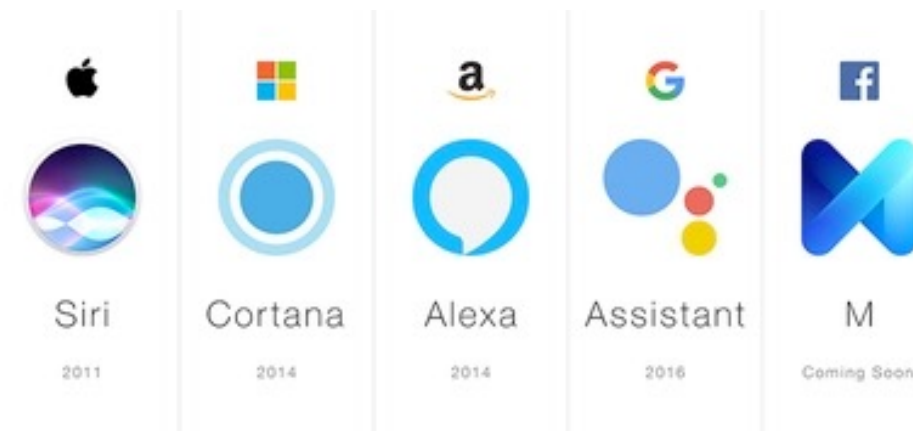
Image colorization



Tesla Autopilot, tesla.com

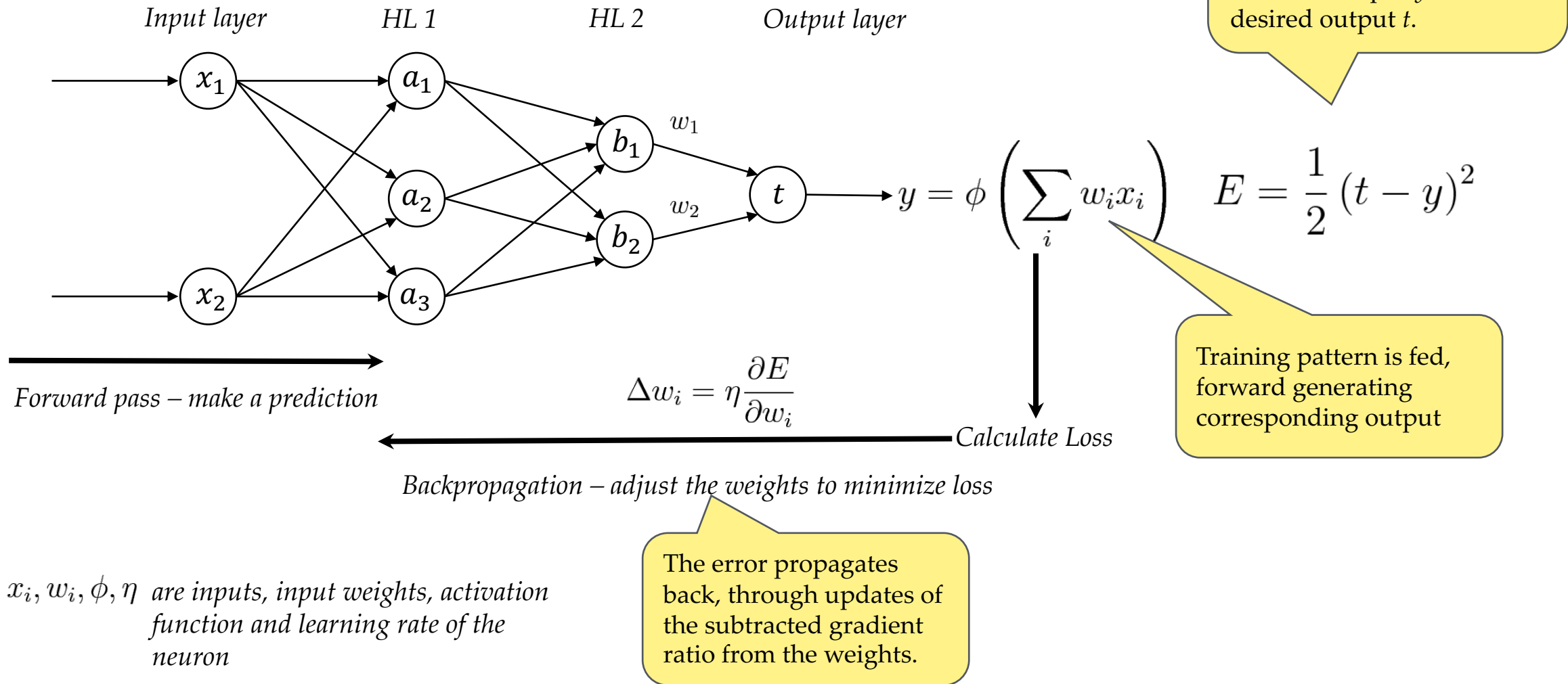


Medical Imaging, CNN, A. Esteva et al, A guide to deep learning in healthcare



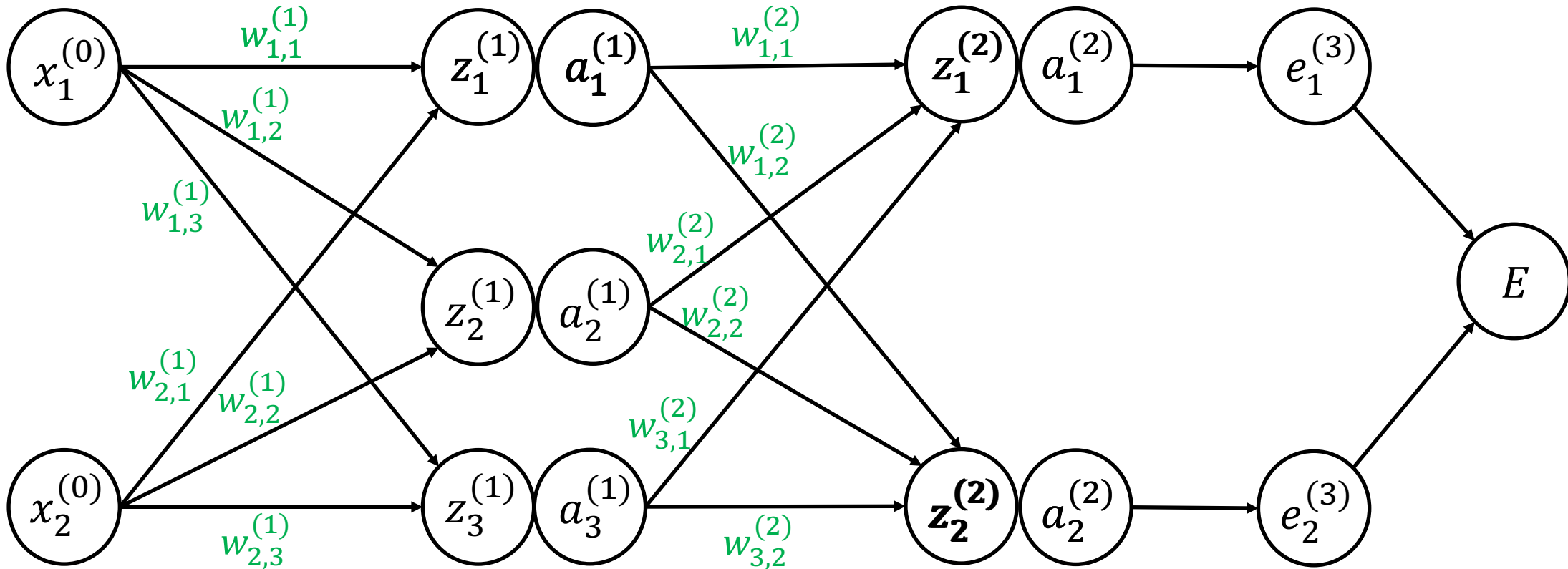
Speech Recognition

Backpropagation



Backpropagation

$$\frac{\partial}{\partial} = \left(\frac{\partial}{\partial} \frac{\partial}{\partial} \frac{\partial}{\partial} + \frac{\partial}{\partial} \frac{\partial}{\partial} \frac{\partial}{\partial} \right) \frac{\partial}{\partial} \frac{\partial}{\partial}$$



Differentiable Programming

“A programming paradigm in which a numeric computer program can be differentiated throughout via **automatic differentiation**. This allows for gradient based optimization of parameters in the program, often via gradient descent.”
[Wikipedia]

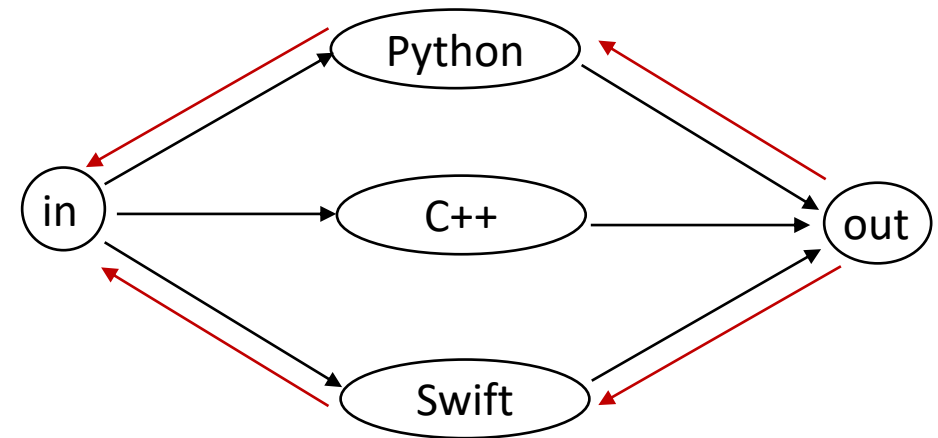
- Deep learning drives recent advancements in automatic differentiation
- AD is useful also in bayesian inference, uncertainty quantification, modeling, simulation
- Several programming languages and frameworks have enabled the differential programming paradigm by adding support for AD.
- Swift, Kotlin, and Julia have made AD a first-class citizen.

Automatic Differentiation & C++

Interoperable Machine Learning

“[The key challenge of scientific ML is that] if there is just one part of your loss function that isn’t AD-compatible, then the whole network won’t train.” -Rackauckas

- Limited support for C++ automatic differentiation hinders the use of C++ within machine learning
- Cannot easily use the vast set of existing C++ codebases in ML applications



C++ Automatic Differentiation Wish-List

- Fast
 - Compilation Time (ideally not JIT)
 - Execution Time
- Works on existing code
 - Doesn't require rewriting user code
 - Supports (most) C++
- Easily Maintainable
 - Minimal impact outside of AD (e.g. no rewrite of STL)
 - Keeps up with evolving standards

Existing AD Approaches (1/3)

- Differentiable DSL (TensorFlow, PyTorch, DiffTaichi, Halide)
 - Provide a new language designed to be differentiated
 - Requires rewriting everything in the DSL and the DSL must support all operations in original code
 - Fast if DSL matches original code well

```
#include "tensorflow/core/public/session.h"

GraphDef graph_def;
session->Create(graph_def);

...
session->Run(inputs, {"output_class/Softmax:0"}, {}, &outputs);
```

Existing AD Approaches (2/3)

- Operator overloading (Adept [C++], JAX [Python])
 - Provide differentiable versions of existing language constructs
 - May require writing to use non-standard utilities
 - Often dynamic: storing instructions/values to later be interpreted

```
template<typename T> square(T val) { return val * val; }

adept::Stack stack;
adept::adouble inp = 3.14;
adept::adouble out(square(inp));
out.set_gradient(1.00);

double derivative = out.get_gradient(3.14);
```

Existing AD Approaches (3/3)

- Source rewriting
 - Statically analyze program to produce a new gradient function in the source language
 - Re-implement parsing and semantics (hard for C++ & must keep up with standard)
 - Requires all code to be available ahead of time

```
double square(double val) { return val * val; }
```



```
tapenade -b -o out.c -head "square(val)/(out)" square.c
```

```
double grad_square(double val) { return 2 * val; }
```

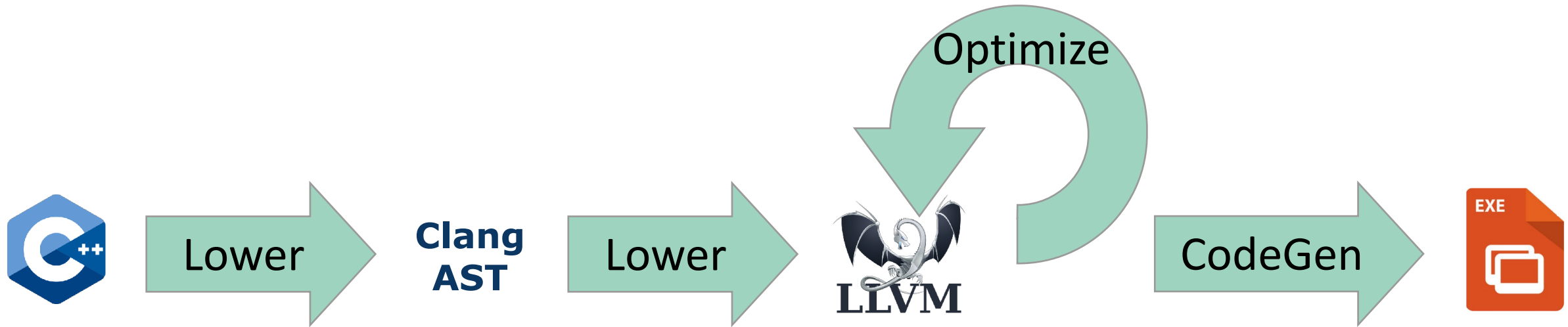
Idea: Compiler-Based AD!

- Want the no user-rewriting, speed, and low STL-rewriting impact of source AD
- Do not want the extra maintenance burden
- Since the compiler already implements parsing, semantic analysis, etc, we can use the compiler to perform source-based AD without maintaining a second parser!

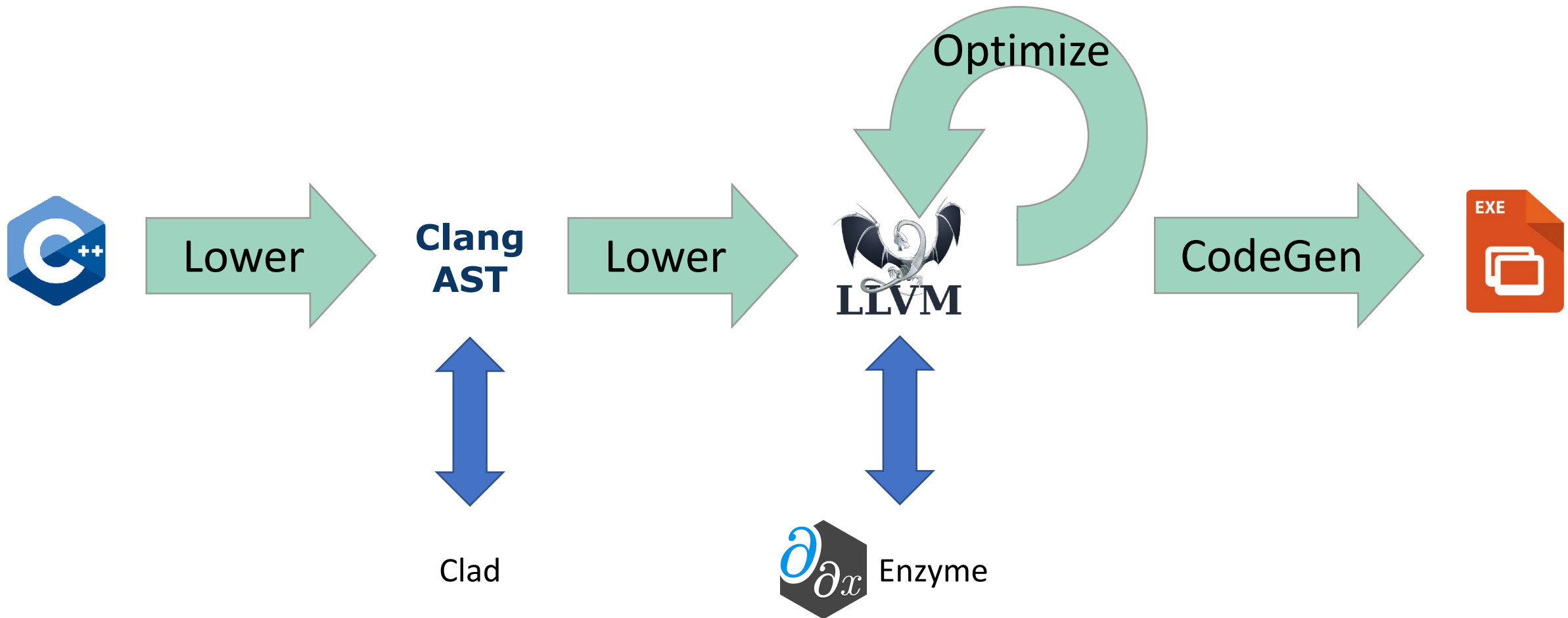


Two Case Studies of Compiler-Based AD

Implementation of AD in Clang/LLVM



Implementation of AD in Clang/LLVM



Case Study 1: Clad – AD of Clang AST

```
double square(double val) {  
    return val * val;  
}
```



Clang

```
FunctionDecl square_double (double)  
|-ParmVarDecl val double  
`-CompoundStmt  
  `-ReturnStmt  
    `-BinaryOperator double *  
      |-ImplicitCastExpr double <LValueToRValue>  
      | `DeclRefExpr double ParmVar val  
      `-ImplicitCastExpr double <LValueToRValue>  
        `DeclRefExpr double ParmVar val
```

Clad



```
FunctionDecl square_darg0 double (double)  
|-ParmVarDecl val double  
`-CompoundStmt  
  |-DeclStmt  
  | `VarDecl d_val double  
  | `ImplicitCastExpr double <IntegralToFloating>  
  |   `IntegerLiteral int 1  
  `-ReturnStmt  
    `-BinaryOperator double +  
      |-BinaryOperator double *  
      | |-ImplicitCastExpr double <LValueToRValue>  
      | | `DeclRefExpr double Var d_val  
      | `ImplicitCastExpr double <LValueToRValue>  
      |   `DeclRefExpr double ParmVar val  
      `-BinaryOperator double *  
        |-ImplicitCastExpr double <LValueToRValue>  
        | `DeclRefExpr double ParmVar val  
        `-ImplicitCastExpr double <LValueToRValue>  
          `DeclRefExpr double lvalue Var d_val
```

Case Study 1: Clad – AD of Clang AST

```
#include "clad/Differentiator/Differentiator.h"
double square(double val) {
    return val * val;
}

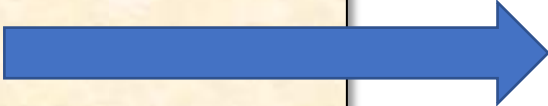
int main() {
    auto dfdx = clad::differentiate(pow2, 0);

    double res = dfdx.execute(1);

    // OR
    auto dfdxFnPtr = dfdx.getFunctionPtr();
    dfdx = dfdxFnPtr(2);

    printf("%s\n", dfdx.getCode());

    ...
}
```



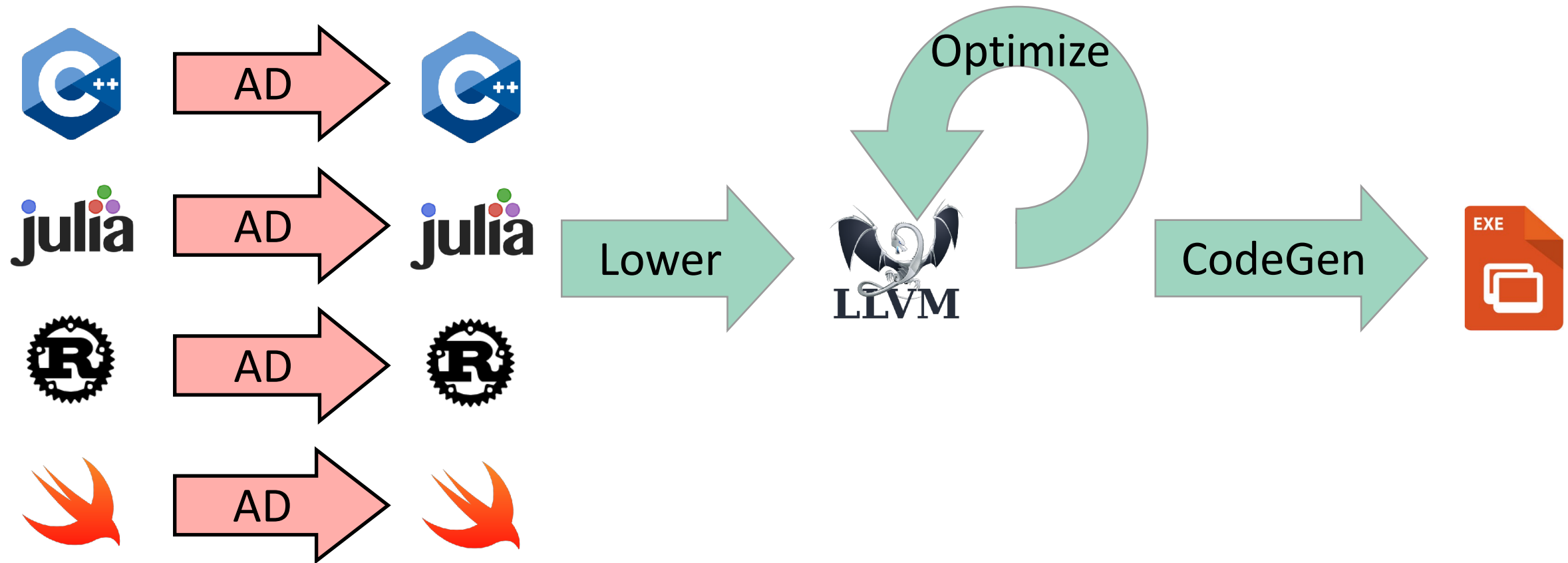
```
double square_darg0(double val) {
    double d_val = 1;
    return d_val * val + val * d_val;
}
```

Clad Key Insights

- Works on the compiler frontend level and uses the tree-rebuilding approach like the C++ template instantiator
- Can produce valid C++ source code

<https://clad.readthedocs.io> / <https://github.com/vgvassilev/clad>

Existing Automatic Differentiation Pipelines



Vector Normalization

$O(n^2)$

```
//Compute magnitude in O(n)
double magnitude(const double[] x);

//Compute norm in O(n^2)
void normalize(double[] __restrict__ out,
               const double[] __restrict__ in) {


    for (int i=0; i<N; i++) {
        out[i] = in[i] / magnitude(in);
    }
}
```

Vector Normalization: LICM

$O(n)$

```
//Compute magnitude in O(n)
double magnitude(const double[] x);

//Compute norm in O(n)
void normalize(double[] __restrict__ out,
               const double[] __restrict__ in) {
    double res = magnitude(in);
    for (int i=0; i<N; i++) {
        out[i] = in[i] / res;
    }
}
```



Vector Normalization: LICM then AD


$O(n)$

```
void grad_normalize(double[] out, double[] dout,
                  double[] in, double[] din) {
    double res = magnitude(in);
    for (int i=0; i<N; i++) {
        out[i] = in[i] / res;
    }
    double d_res = 0;
    for (int i=N-1; i>=0; i--) {
        dres += -in[i]*in[i]/res * dout[i];
        din[i] += dout[i]/res;
    }
    grad_magnitude(in, din, n, dres);
}
```

Vector Normalization: AD, then LICM

$O(n^2)$

```
void grad_normalize(double[] out, double[] dout,
                  double[] in, double[] din) {
    double res = magnitude(in);
    for (int i=0; i<N; i++) {
        out[i] = in[i] / res;
    }
    for (int i=N-1; i>=0; i--) {
        double dres = -in[i]*in[i]/res * dout[i];
        din[i] += dout[i]/res;
        grad_magnitude(in, din, n, dres);
    }
}
```



Can't LICM as uses loop-local variable dres

Optimization & Automatic Differentiation

Differentiating after optimization can create *asymptotically faster* gradients!

$O(n^2)$

```
for i=0..n {  
  out[i] /= mag(in)  
}
```

Optimize

$O(n)$

```
res = mag(in)  
for i=0..n {  
  out[i] /= res  
}
```

AD

$O(n)$

```
d_res = 0.0  
for i=n..0 {  
  d_res += d_out[i]...  
}  
∇mag(d_in, d_res)
```

$O(n^2)$

```
for i=0..n {  
  out[i] /= mag(in)  
}
```

AD

$O(n^2)$

```
for i=n..0 {  
  d_res = d_out[i]...  
  ∇mag(d_in, d_res)  
}
```

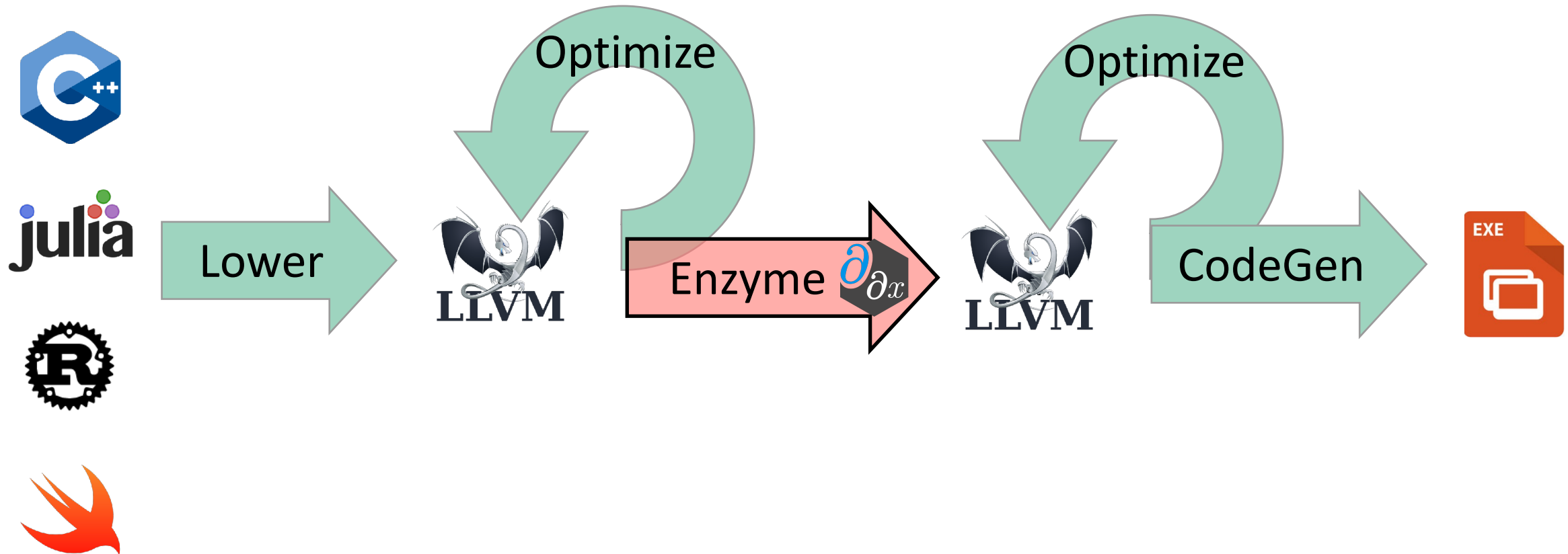
Optimize

$O(n^2)$

```
for i=n..0 {  
  d_res = d_out[i]...  
  ∇mag(d_in, d_res)  
}
```

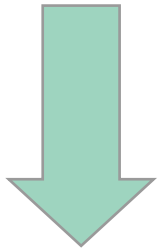
Enzyme Approach [MC20]

Performing AD at low-level lets us work on ***optimized*** code!



Case Study 2: Enzyme – AD of LLVM IR

```
double square(double val) {  
    return val * val;  
}  
  
double __enzyme_autodiff(void*, ...);  
  
double grad_square(double val) {  
    return __enzyme_autodiff((void*)square, val);  
}
```



Clang

```
define double @square(double %val) {  
    %sq = fmul double %val, %val  
    ret double %sq  
}
```

 Enzyme



```
define double @grad_square(double %val) {  
    %res = fadd double %val, %val  
    ret double %res  
}
```

Case Study 2: Enzyme – AD of LLVM IR

```
double square(double val) {  
    return val * val;  
}  
  
double __enzyme_autodiff(void*, ...);  
  
double grad_square(double val) {  
    return __enzyme_autodiff((void*)square, val);  
}
```



Clang

```
define double @square(double %val) {  
    %sq = fmul double %val, %val  
    ret double %sq  
}
```

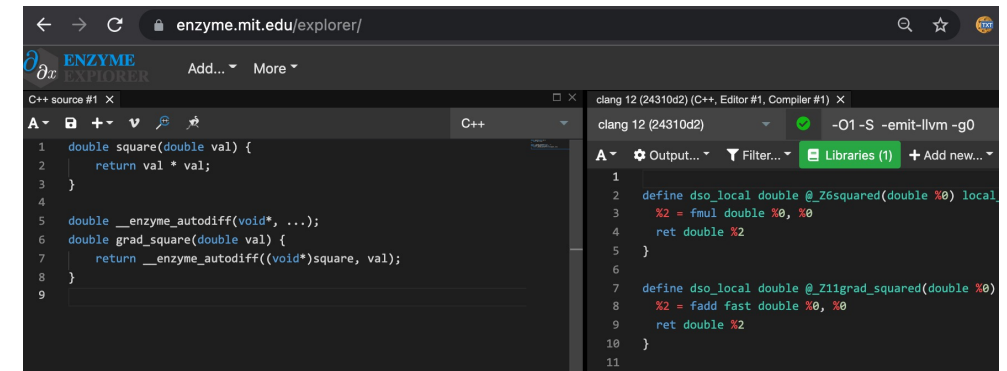
 Enzyme



```
define double @grad_square(double %val) {  
    %res = fadd double %val, %val  
    ret double %res  
}
```

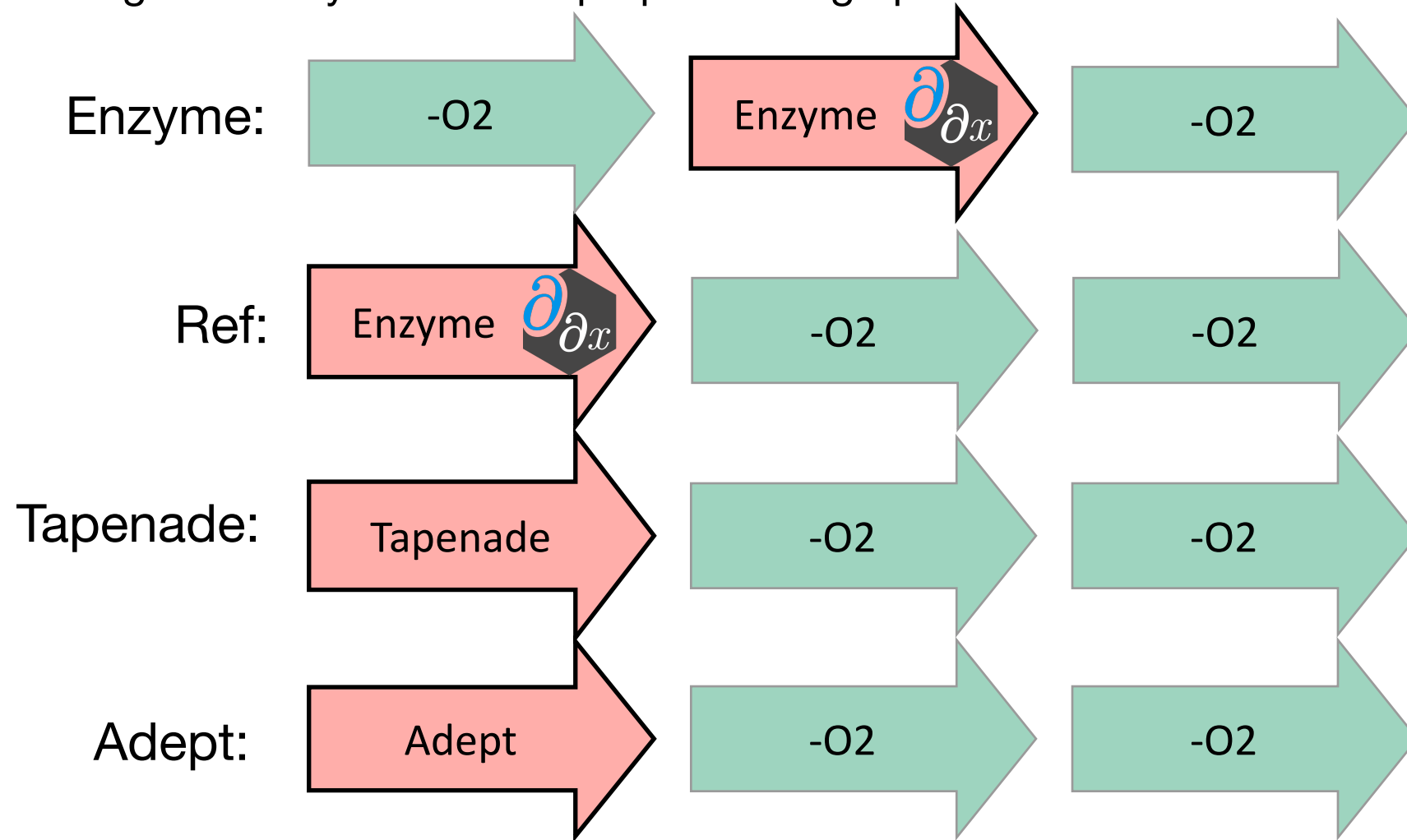
Try Online On
The Enzyme Compiler Explorer!

<https://bit.ly/3aNP6bB>

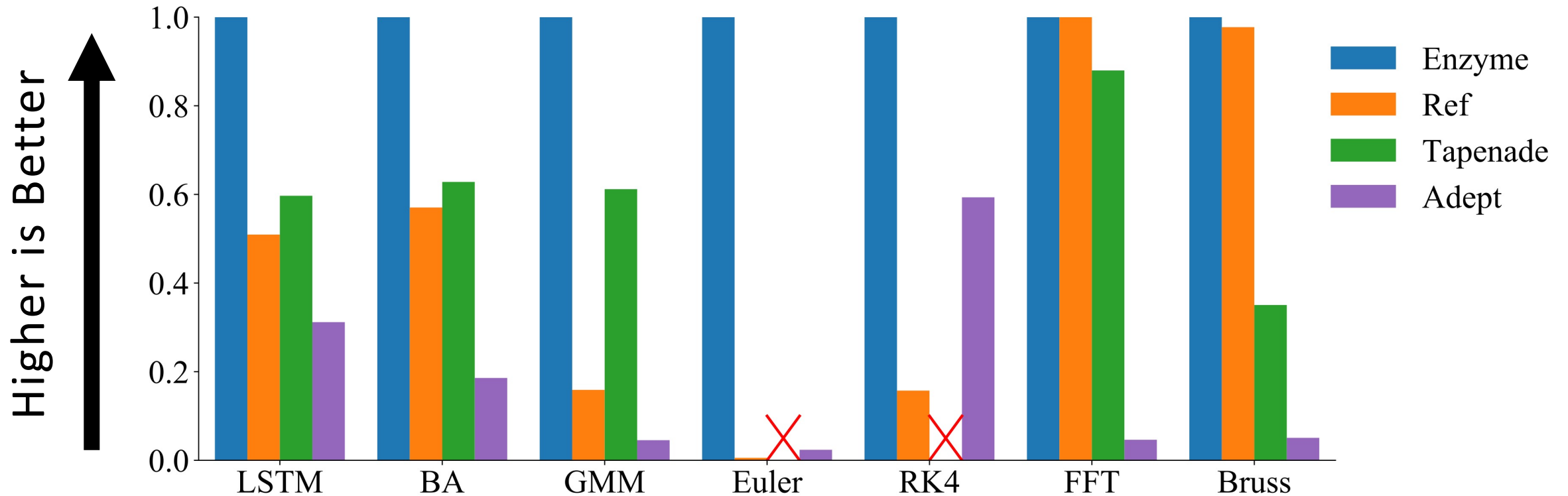


Enzyme Evaluation

Compared against Enzyme without preprocessing optimizations and two fastest AD tools



Speedup of Enzyme



Enzyme is **4.2x faster** than Reference!

Key Enzyme Insights

- Running AD after/alongside optimization enables substantial speedups, including 4.2x on a suite of ML/scientific codes
- Enzyme achieves state-of-the art performance
- Enzyme is the first AD tool to differentiate arbitrary GPU kernels (including AMD and NVIDIA) [MCPH+21]
- Enzyme has support for generic forms of parallelism including OpenMP, MPI, and other frameworks that build upon them like Kokkos and RAJA

<https://enzyme.mit.edu> / <https://github.com/wsmoses/Enzyme>

[MCPH+21] Moses et al. Reverse-Mode Automatic Differentiation and Optimization of GPU Kernels via Enzyme. *To appear at SC, 2021.*

Overall AD Compiler Insights

- Existing code does not need to be rewritten to be differentiated.
- Being within the compiler AD tools to continue function as the frontend languages & standards evolve.
- Has access to source locations and can issue precise diagnostics
- Can be successfully implemented at either a high or low level

...but this requires using a conformant compiler

=> Can we standardize this?

Standardization Efforts

- We believe first class support of differentiable programming paradigm is an important feature which will become central for various data science, research and industry communities
- We believe that compiler-aided AD is the most viable path forward to supporting high-performance
- We have produced an overview paper “Differentiable programming for C++” <https://wg21.link/P2072>
- We have solicited feedback from the ML study group of isocpp (aka SG19) but also from various other parties
- We are keen on turning the overview paper into a concrete plan!

Conclusion

- Differentiable Programming is a new and promising programming paradigm which relies on well developed theory and technology
- The presented tools are being developed and integrated in various fields
- The standardization efforts are ramping up and we hope to solicit support after this talk

Thank you!

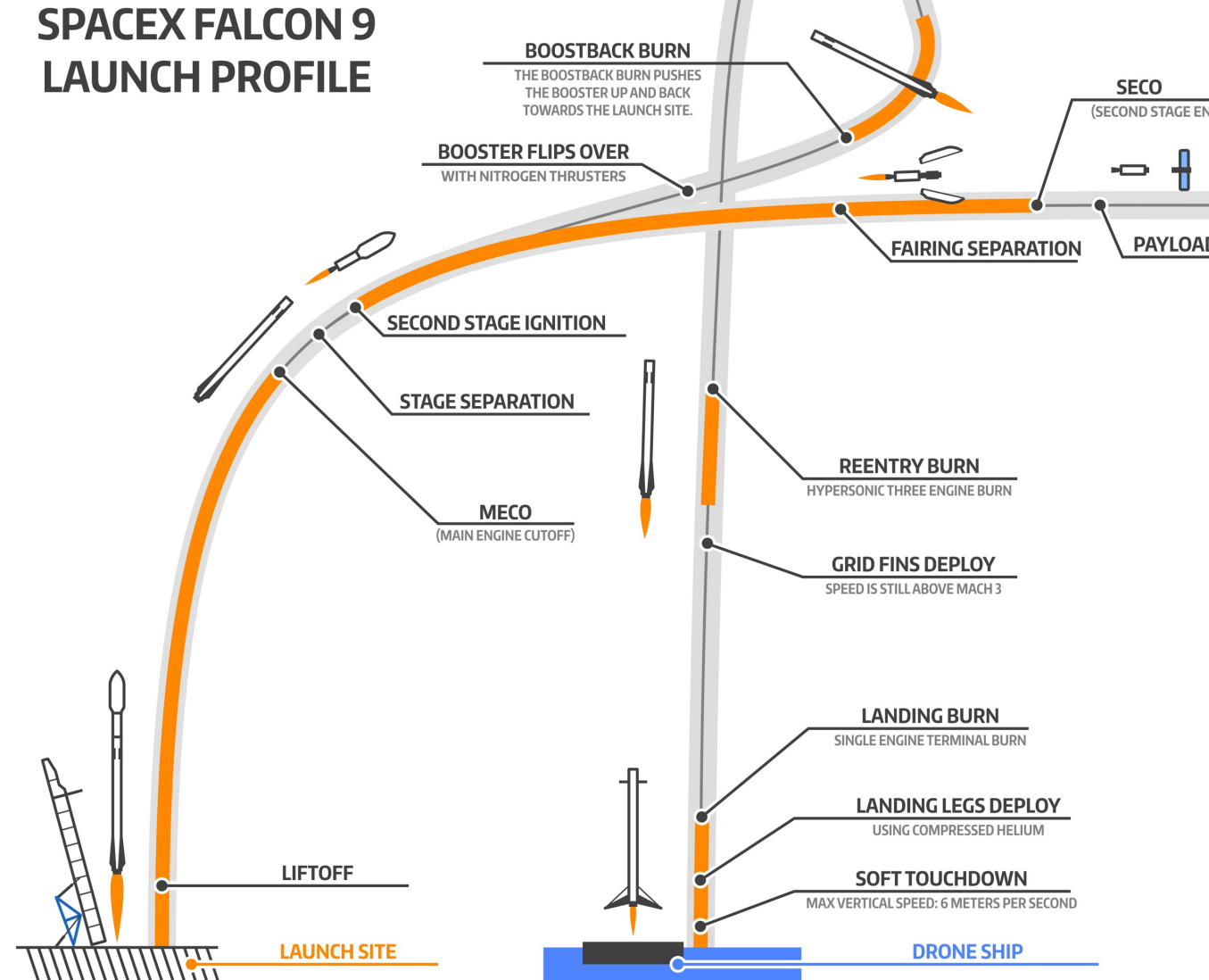
Q & A

Backup

Optin

Can steer a process
towards a reference
trajectory automatically?

Credit: imgur.com



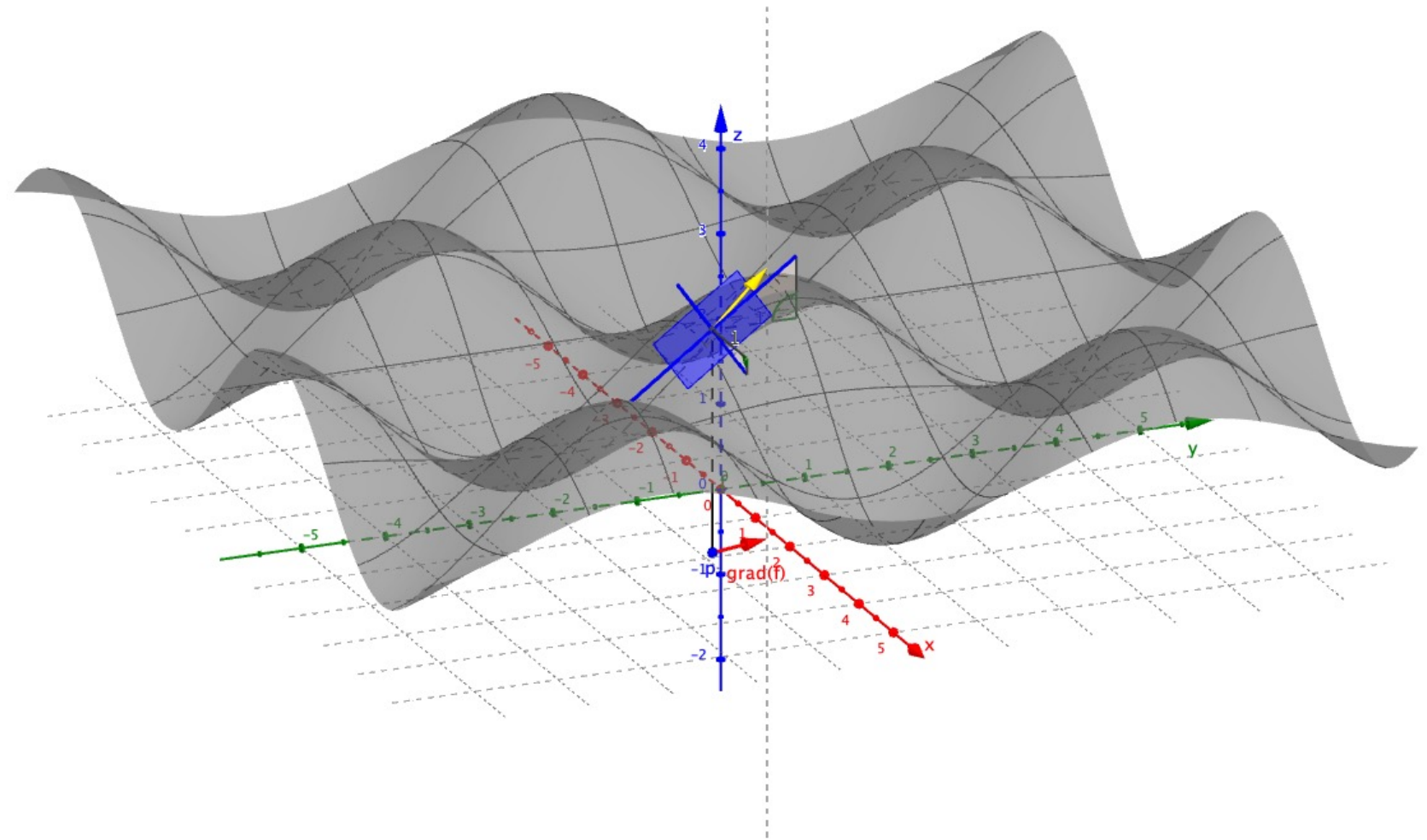
Credit: Reddit.com

Differentiable Programming. Birds' eye

- Sees computer programs as components which interact with each other. Each (sub)component represents a process which can be described with a function (in the math sense).
- The AD can provide information about key properties of the process to enable optimization and possibly changing the algorithm.
- The differentiable programming paradigm encourages software to be written in such a way that AD can be applied to (almost) all components.
- In many cases the software tends to optimize a process – e.g. minimize unemployment, minimize expenses and maximize profit, reach the moon with minimal fuel consumption, minimize log likelihood given data and constraints

Today's Automatic Differentiation Ecosystem

- First-Class Support (Swift)
 - (Almost) everything in the language is differentiable
- Domain-Specific Language/Libraries (PyTorch, TensorFlow, Halide, Taichi)
 - A new language or library has been created where every operation is differentiable
 - Requires rewriting everything in the new language, and cannot handle programs outside it
- Meta Programming (Julia, C++)
 - Packages (Zygote.jl, Diffractor.jl, Adept) build off of meta-programming (such as generated functions, templates, or operator overloading) to c
- External Tools (C)
 - A tool (e.g. Tapenade) takes in input code and generates a new source file which computes the derivative



Implementation Classification

Implementations vary on how much work is done at compile-time:

- Tracing/Taping – the compute graph is constructed as the program is executed, just like tracing JITs, the execution is recorded, transformed, and compiled “just-in-time”.
 - Typical implementation in C++ is using metaprogramming using operator overloading on a special type.
 - Easy to implement
 - Inefficient, needs code modification
- Source transformation – the compute graph is constructed before compilation and then transformed and compiled.
 - Typical implementation is a custom parser building code representation and producing the transformed code.
 - Efficient as many computations and optimizations are done ahead of time
 - Hard to implement (especially for C++)

Derivatives + Programming

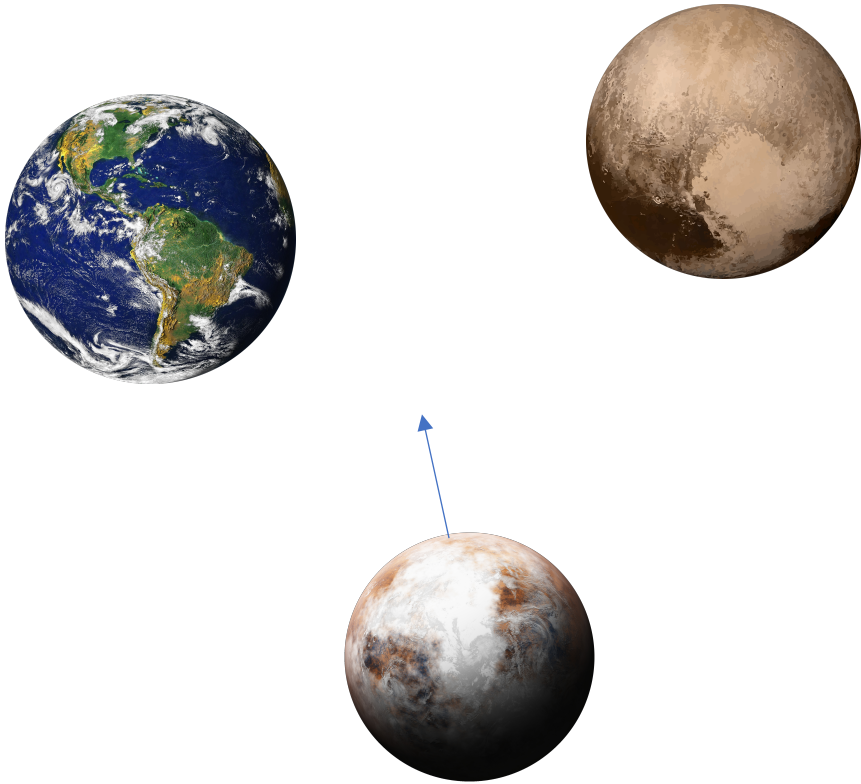
- Many algorithms require the derivatives of various functions

```
//Compute magnitude in  $O(n)$ 
double magnitude(const double[] x);

//Compute norm in  $O(n^2)$ 
double PotentialEnergy(double[] __restrict__ out,
                      const double[] __restrict__ in) {

    for (int i=0; i<N; i++) {
        out[i] = in[i] / magnitude(in);
    }
}
```

Derivatives + Programming

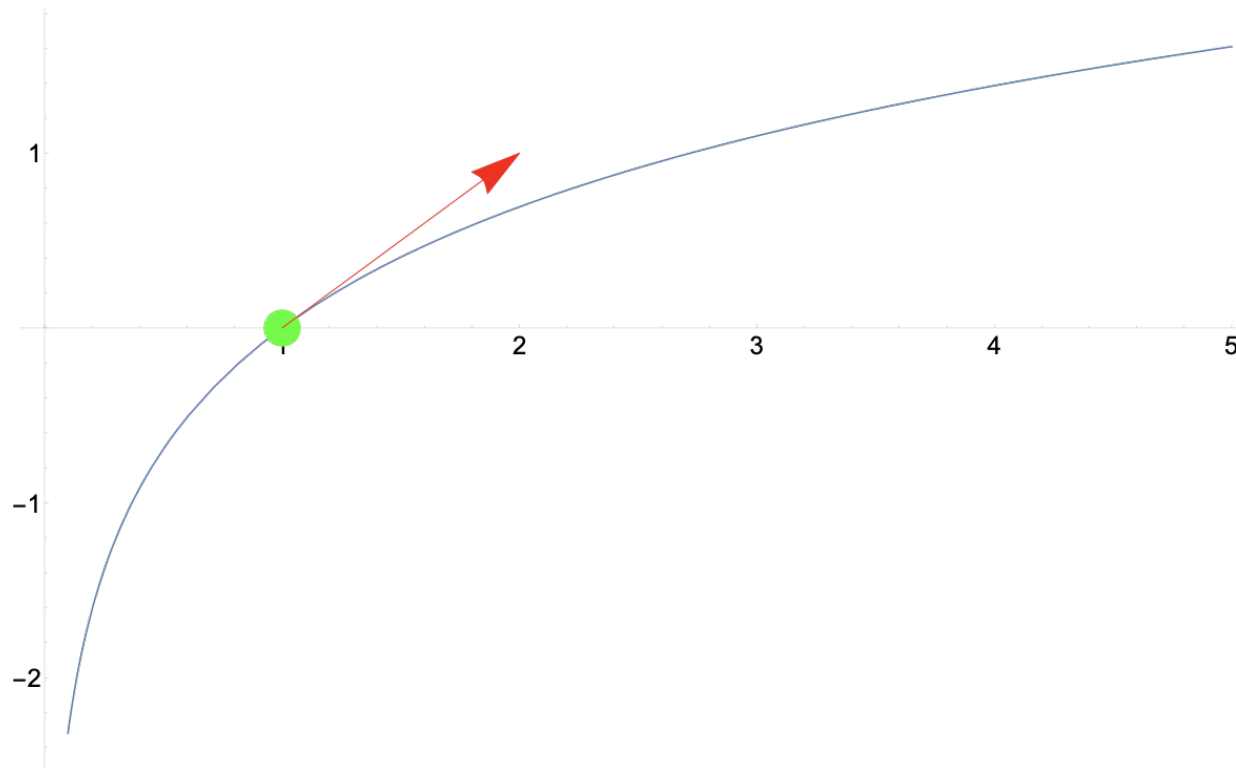


```
//Compute magnitude in O(n)
double magnitude(const double[] x);

// Compute Compute norm in O(n^2)
double PotentialEnergy(vector<Planet> in) {
    //  $G_{\mu\nu} + \Lambda g_{\mu\nu} = 8\pi G * T_{\mu\nu}/c^4$ 
    // ...
}
```

Derivatives (Gradients)

- From calculus, the derivative of function as a rate of change
- Given a function $f(x)$, the derivative $f'(x)$ describes the slope.



Derivatives + Programming

- Many algorithms require the derivatives of various functions
 - Machine learning (back-propagation, Bayesian inference, uncertainty quantification)
 - Scientific computing (modeling, simulation)
- When working with large codebases or dynamically-generated programs, manually writing derivative functions becomes intractable
- Several languages / frameworks (Swift, PyTorch/TensorFlow/JaX in Python, Julia) have made differentiation a first-class primitive

Derivatives (Gradients)

- Remember from calculus the derivative of function Give some lightweight description about the chain rule
- Introduce the concept of a derivative and gradient and their high-level use
- Give examples how else one can produce a gradient – numerically or symbolically
- Describe existing approaches – source transformation, using metaprogramming or other reflection tools (such as trace-based ad)
- Enumerate prominent tools

Differentiable Programming

- The concept of AD dates back from dual number algebra from 19th century
- In 1970's AD was used to estimate roundoff errors
- In the ML era was rebranded as backpropagation
- In an [essay](#), LeCun coined the term Differentiable Functional Programming
- Now there are efforts in enabling differentiable programming in computer graphics (differentiable rendering), computer vision, physics simulators (fluid dynamics), ...

Controlling

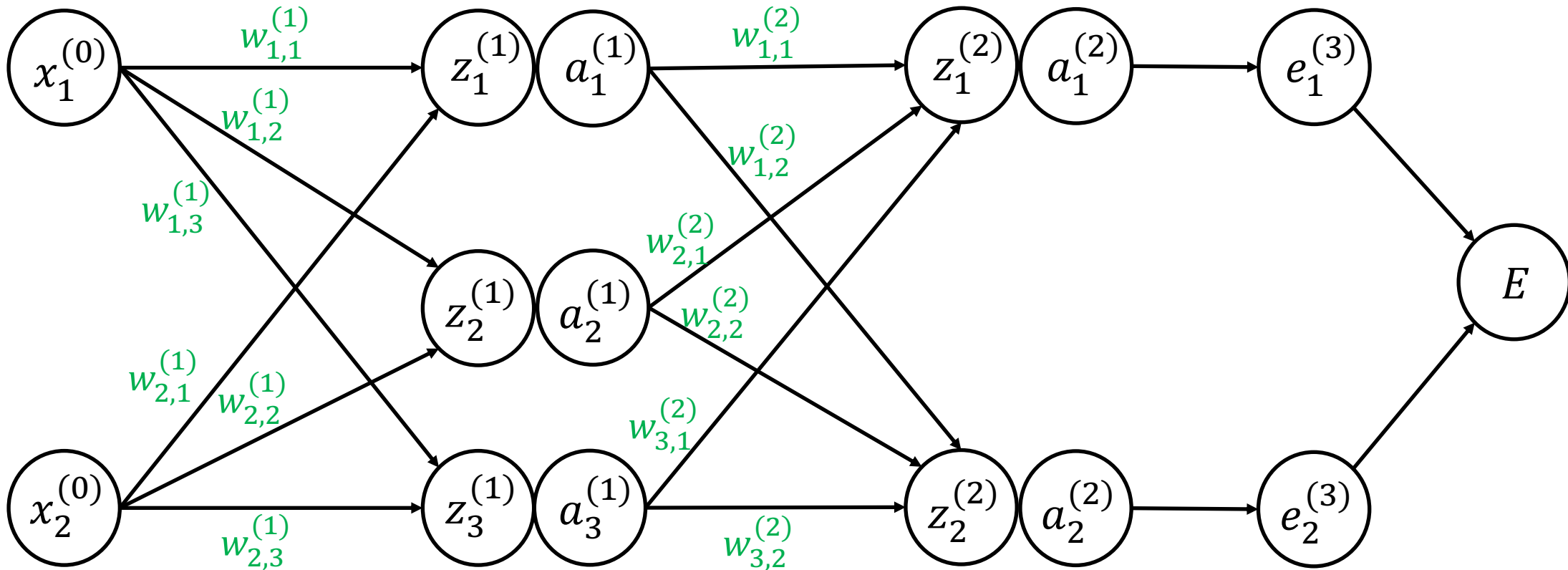
The goal is to reach zero altitude with zero vertical velocity given tight constraints of landing area and fuel.



Credit: Official SpaceX Photos

Backpropagation

$$\frac{\partial E}{\partial w_{1,1}^{(1)}} = \left(\frac{\partial e_1^{(3)}}{\partial a_1^{(2)}} \frac{\partial a_1^{(2)}}{\partial z_1^{(2)}} \frac{\partial z_1^{(2)}}{\partial a_1^{(1)}} + \frac{\partial e_2^{(3)}}{\partial a_2^{(2)}} \frac{\partial a_2^{(2)}}{\partial z_2^{(2)}} \frac{\partial z_2^{(2)}}{\partial a_1^{(1)}} \right) \frac{\partial a_1^{(1)}}{\partial z_1^{(1)}} \frac{\partial z_1^{(1)}}{\partial w_{1,1}^{(1)}}$$



Differentiable Programming Frameworks

- Define-and-run
- Theano..
- Define-by-run
- Pytorch..

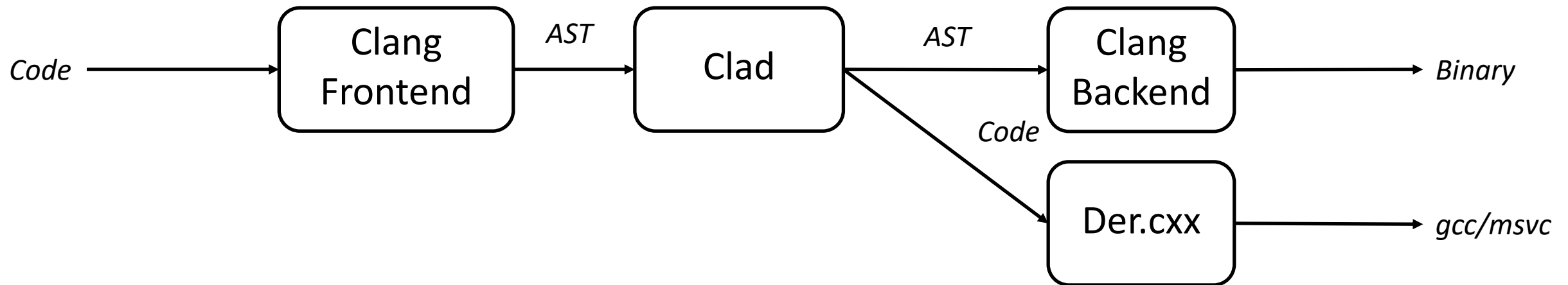
Static compute graph

Dynamic compute graph

Case Study 1: Clad – AD of Clang AST

```
FunctionDecl f 'double (double)'
| -ParmVarDecl x 'double'
| -CompoundStmt
|   -ReturnStmt
|     -BinaryOperator 'double' '*'
|       -ImplicitCastExpr 'double' <LValueToRValue>
|         -DeclRefExpr 'double' lvalue ParmVar 'x' 'double'
|       -ImplicitCastExpr 'double' <LValueToRValue>
|         -DeclRefExpr 'double' lvalue ParmVar 'x' 'double'
|
double fsq(double x) {
    return x * x;
}
```

```
FunctionDecl 0x7f7f801dbff8 <<invalid sloc>> <invalid sloc> f_darg0 'double (double)'
| -ParmVarDecl 0x7f7f801dc090 <<invalid sloc>> <invalid sloc> used x 'double'
| -CompoundStmt 0x7f7f801dc3d0 <<invalid sloc>>
|   -DeclStmt 0x7f7f801dc190 <<invalid sloc>>
|     -VarDecl 0x7f7f801dc118 <<invalid sloc>> <invalid sloc> used _d_x 'double' cinit
|       -ImplicitCastExpr 0x7f7f801dc178 <<invalid sloc>> 'double' <IntegralToFloating>
|         -IntegerLiteral 0x7f7f801dc0f8 <<invalid sloc>> 'int' 1
|   -ReturnStmt 0x7f7f801dc398 <<invalid sloc>>
|     -BinaryOperator 0x7f7f801dc318 <<invalid sloc>> 'double' '+'
|       -BinaryOperator 0x7f7f801dc298 <<invalid sloc>, T.cpp:3:32> 'double' '*'
|         -ImplicitCastExpr 0x7f7f801dc268 <<invalid sloc>> 'double' <LValueToRValue>
|           -DeclRefExpr 0x7f7f801dc1a8 <<invalid sloc>> 'double' lvalue Var 0x7f7f801dc118 '_d_x' 'double'
|         -ImplicitCastExpr 0x7f7f801dc280 <col:32> 'double' <LValueToRValue>
|           -DeclRefExpr 0x7f7f801dc208 <col:32> 'double' lvalue ParmVar 0x7f7f801dc090 'x' 'double'
|       -BinaryOperator 0x7f7f801dc2f0 <col:30, <invalid sloc>> 'double' '*'
|         -ImplicitCastExpr 0x7f7f801dc2c0 <col:30> 'double' <LValueToRValue>
|           -DeclRefExpr 0x7f7f801dc1d0 <col:30> 'double' lvalue ParmVar 0x7f7f801dc090 'x' 'double'
|         -ImplicitCastExpr 0x7f7f801dc2d8 <<invalid sloc>> 'double' <LValueToRValue>
|           -DeclRefExpr 0x7f7f801dc1a8 <<invalid sloc>> 'double' lvalue Var 0x7f7f801dc118 '_d_x' 'double'
```



```
double f_darg0(double x) {
    double _d_x = 1;
    return _d_x * x + x * _d_x;
}
```

Clad. Usage

```
// clang -fplugin=../../libclad.so
// Necessary for clad to work include
#include "clad/Differentiator/Differentiator.h"
double pow2(double x) { return x * x; }
```

```
double pow2_darg0(double); // to be filled by clad
```

```
int main() {
```

```
    auto dfdx = clad::differentiate(pow2, 0);
```

```
    // Function execution can happen in 3 ways:
```

```
    // 1) Using CladFunction::execute method.
```

```
    double res = dfdx.execute(1);
```

```
    // 2) Using the function pointer.
```

```
    auto dfdxFnPtr = dfdx.getFunctionPtr();
```

```
    res = dfdxFnPtr(2);
```

```
    // 3) Using direct function access through fwd declaration.
```

```
    printf(pow2_darg0(3);
```

```
    printf("The derivative code is: %s\n", dfdx.getCode());
```

```
    return res;
```

```
}
```

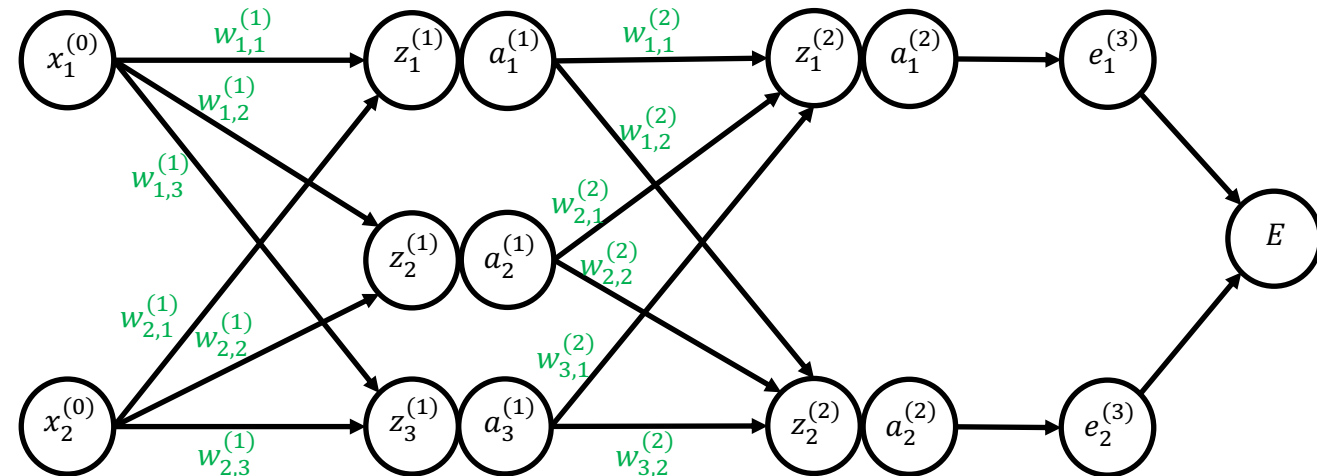
Tells the plugin to create pow2_darg0.

The programmer can use the derivative via a wrapper object, function pointer or forward declaration.

```
performance-test@vv-nuc ~/clad-build-llvm11 $ ./a.out
6.000000
The derivative code is: double pow2_darg0(double x) {
    double _d_x = 1;
    return _d_x * x + x * _d_x;
}
```

Differentiable Programming as Generalization of DL

- Each layer describes a subprocess algorithmically
- Each layer can be considered as a reusable software building block
- Layers can be combined and their sensitivity can be tuned by gradient descent using AD



LeCun: “Deep Learning est mort. Vive Differentiable Programming!”

Andrej Karpathy: Software 2.0 is written in much more abstract, human unfriendly language [...]. Instead, our approach is to specify some goal on the behavior of a desirable program (e.g., “satisfy a dataset of input output pairs of examples”, or “win a game of Go”).

Implementation of AD in Clang/LLVM

```
double square(double val) {  
    return val * val;  
}
```

```
double square_darg0(double val) {  
    double d_val = 1;  
    return d_val * val + val * d_val;  
}
```



Lower

Clang

Lower

```
FunctionDecl square double (double)  
|-ParmVarDecl val double  
`-CompoundStmt  
  |-ReturnStmt  
  |-BinaryOperator double *  
    |-ImplicitCastExpr double <LValueToRValue>  
    | |-DeclRefExpr double ParmVar val  
    |-ImplicitCastExpr double <LValueToRValue>  
    | |-DeclRefExpr double ParmVar val
```

```
FunctionDecl square_darg0 double (double)  
|-ParmVarDecl val double  
|-CompoundStmt  
  |-DeclStmt  
  | |-VarDecl d_val double  
  | |-ImplicitCastExpr double <IntegralToFloating>  
  | | |-IntegerLiteral int 1  
  |-ReturnStmt  
  |-BinaryOperator double +  
    |-BinaryOperator double *  
    | |-ImplicitCastExpr double <LValueToRValue>  
    | | |-DeclRefExpr double Var d_val  
    | |-ImplicitCastExpr double <LValueToRValue>  
    | | |-DeclRefExpr double ParmVar val  
    |-BinaryOperator double *  
    | |-ImplicitCastExpr double <LValueToRValue>  
    | | |-DeclRefExpr double ParmVar val  
    |-ImplicitCastExpr double <LValueToRValue>  
    | |-DeclRefExpr double lvalue Var d_val
```



Implementation of AD in LLVM/Clang

Two options – transformation of the Clang AST or transformation of the LLVM IR each with pros and cons:

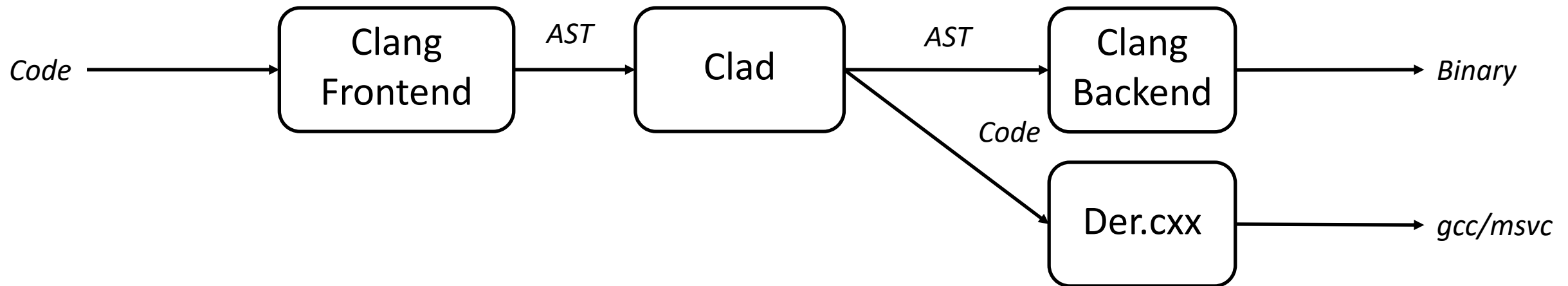
- Clad – a Clang Plugin transforming the AST of the supported languages by clang (C++ but also CUDA, C, ObjC...)
- Enzyme – an LLVM IR plugin transforming the LLVM IR

Case Study 1: Clad – AD of Clang AST

```
FunctionDecl f 'double (double)'
| -ParmVarDecl x 'double'
| -CompoundStmt
|   -ReturnStmt
|     -BinaryOperator 'double' '*'
|       -ImplicitCastExpr 'double' <LValueToRValue>
|         -DeclRefExpr 'double' lvalue ParmVar 'x' 'double'
|       -ImplicitCastExpr 'double' <LValueToRValue>
|         -DeclRefExpr 'double' lvalue ParmVar 'x' 'double'
```

```
double fsq(double x) {
    return x * x;
}
```

```
FunctionDecl 0x7f7f801dbff8 <<invalid sloc>> <invalid sloc> f_darg0 'double (double)'
| -ParmVarDecl 0x7f7f801dc090 <<invalid sloc>> <invalid sloc> used x 'double'
| -CompoundStmt 0x7f7f801dc3d0 <<invalid sloc>>
|   -DeclStmt 0x7f7f801dc190 <<invalid sloc>>
|     -VarDecl 0x7f7f801dc118 <<invalid sloc>> <invalid sloc> used _d_x 'double' cinit
|       -ImplicitCastExpr 0x7f7f801dc178 <<invalid sloc>> 'double' <IntegralToFloating>
|         -IntegerLiteral 0x7f7f801dc0f8 <<invalid sloc>> 'int' 1
|   -ReturnStmt 0x7f7f801dc398 <<invalid sloc>>
|     -BinaryOperator 0x7f7f801dc318 <<invalid sloc>> 'double' '+'
|       -BinaryOperator 0x7f7f801dc298 <<invalid sloc>, T.cpp:3:32> 'double' '*'
|         -ImplicitCastExpr 0x7f7f801dc268 <<invalid sloc>> 'double' <LValueToRValue>
|           -DeclRefExpr 0x7f7f801dc1a8 <<invalid sloc>> 'double' lvalue Var 0x7f7f801dc118 '_d_x' 'double'
|         -ImplicitCastExpr 0x7f7f801dc280 <col:32> 'double' <LValueToRValue>
|           -DeclRefExpr 0x7f7f801dc208 <col:32> 'double' lvalue ParmVar 0x7f7f801dc090 'x' 'double'
|       -BinaryOperator 0x7f7f801dc2f0 <col:30, <invalid sloc>> 'double' '*'
|         -ImplicitCastExpr 0x7f7f801dc2c0 <col:30> 'double' <LValueToRValue>
|           -DeclRefExpr 0x7f7f801dc1d0 <col:30> 'double' lvalue ParmVar 0x7f7f801dc090 'x' 'double'
|         -ImplicitCastExpr 0x7f7f801dc2d8 <<invalid sloc>> 'double' <LValueToRValue>
|           -DeclRefExpr 0x7f7f801dc1a8 <<invalid sloc>> 'double' lvalue Var 0x7f7f801dc118 '_d_x' 'double'
```



```
double f_darg0(double x) {
    double _d_x = 1;
    return _d_x * x + x * _d_x;
}
```

Clad. Usage

```
// clang -fplugin=../../libclad.so
// Necessary for clad to work include
#include "clad/Differentiator/Differentiator.h"
double pow2(double x) { return x * x; }
```

```
double pow2_darg0(double); // to be filled by clad
```

```
int main() {
```

```
    auto dfdx = clad::differentiate(pow2, 0);
```

```
    // Function execution can happen in 3 ways:
```

```
    // 1) Using CladFunction::execute method.
```

```
    double res = dfdx.execute(1);
```

```
    // 2) Using the function pointer.
```

```
    auto dfdxFnPtr = dfdx.getFunctionPtr();
```

```
    res = dfdxFnPtr(2);
```

```
    // 3) Using direct function access through fwd declaration.
```

```
    printf(pow2_darg0(3);
```

```
    printf("The derivative code is: %s\n", dfdx.getCode());
```

```
    return res;
```

```
}
```

Tells the plugin to create pow2_darg0.

The programmer can use the derivative via a wrapper object, function pointer or forward declaration.

```
performance-test@vv-nuc ~/clad-build-llvm11 $ ./a.out
6.000000
The derivative code is: double pow2_darg0(double x) {
    double _d_x = 1;
    return _d_x * x + x * _d_x;
}
```