AD beyond Python and ML

Vassil Vassilev, Princeton

compiler-research.org

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

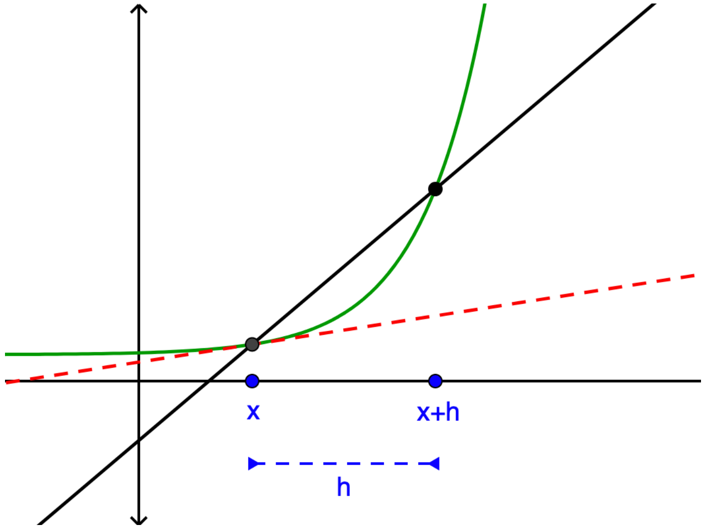
Algorithmic or Automatic Differentiation (AD)

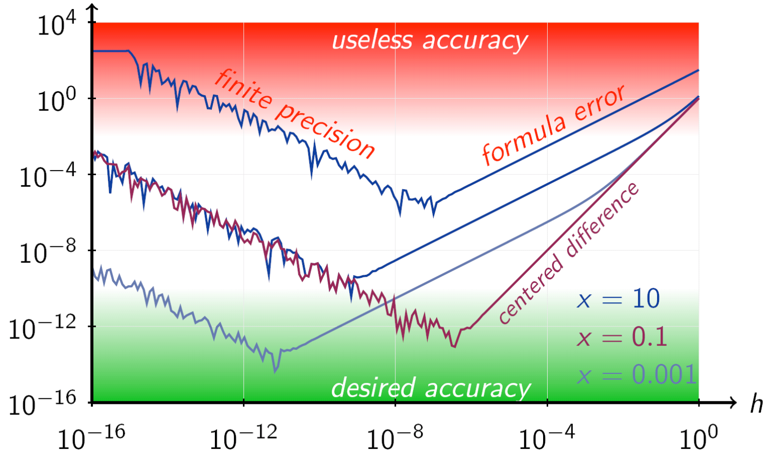
- Automatically generate a C++ program to compute the derivative of a given function

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





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

$$f(x) = e^{e^{e^{e^{e^{x}}}}} \xrightarrow{\text{Symbolic via Wolfram Alpha}} \frac{d}{dx}\left(e^{e^{e^{e^{e^{x}}}}}\right) = e^{x+e^{e^{e^{e^{x}}}}+e^{e^{e^{x}}}+e^{e^{x}}+e^{x}}$$

Figure out the analytical fn

Handcode

Handcode

```
// f(x)=e^(e^(e^(e^(e^x))))
#include <cmath>
double f (double x) {
  double result = x;
  for (unsigned i = 0; i < 5; i++)
    result = std::exp(result);
  return result;
}
```

AD →

```
double f_dx(double x) {
  double result = x;
  double d_result = 1;
  for (unsigned i = 0; i < 5; i++) {
    result = std::exp(result);
    d_result *= result;
  }
  return d_result;
}
```

Chain Rule & AD

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

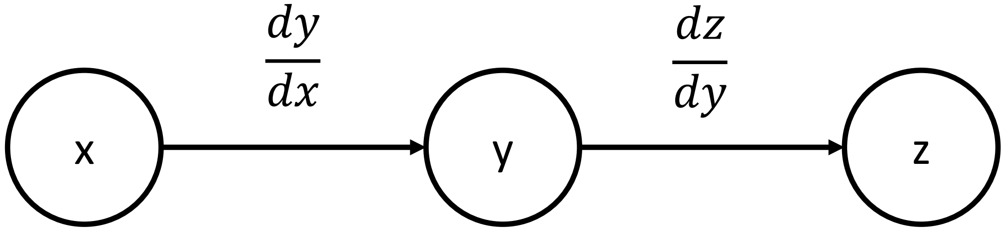
Intuitively, the chain rule states that knowing the instantaneous rate of change of *z* relative to *y* and that of *y* relative to *x* allows one to calculate the instantaneous rate of change of *z* relative to *x* as the product of the two rates of change.

“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 × 4 = 8 times as fast as the man.” G. Simmons

AD. Algorithm Decomposition

```
y = f(x)
z = g(y)
```

```
dydx = dfdx(x)
dzdy = dgdy(y)
dzdx = dzdy * dydx
```

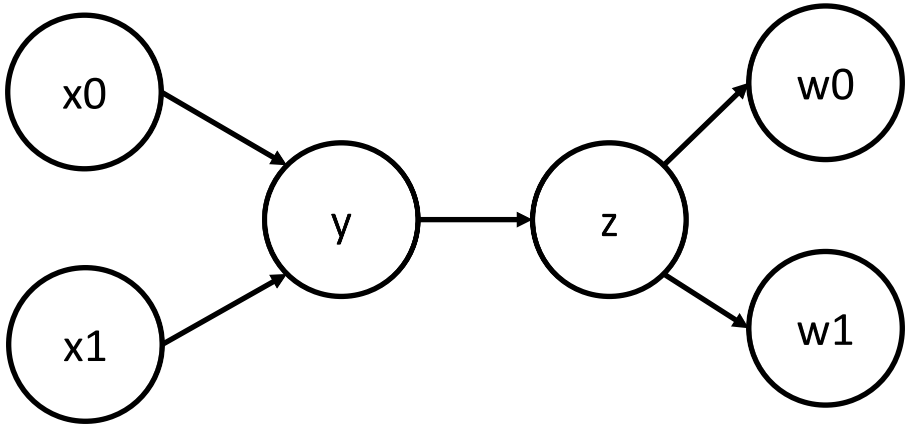


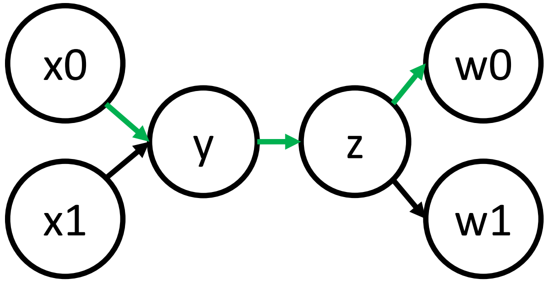
In the computational graph each node is a variable and each edge is derivatives between adjacent edges

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

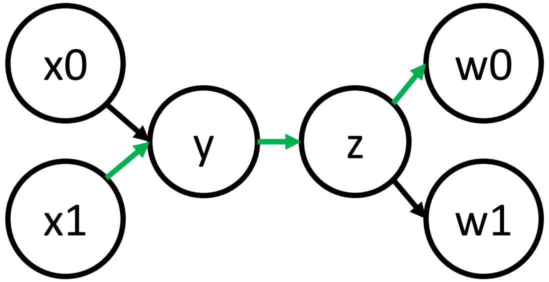
AD. Chain Rule

```
y = f(x0, x1)
z = g(y)
w0, w1 = l(z)
```

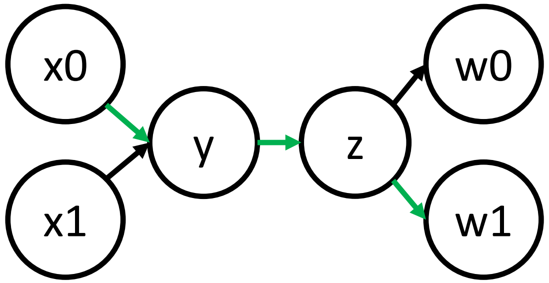




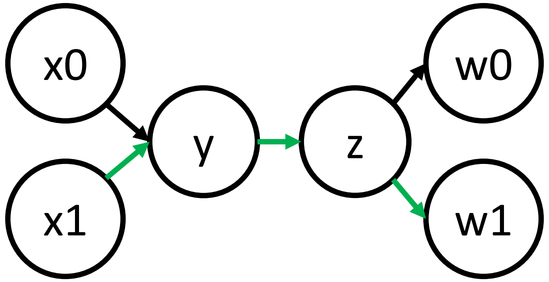
$$\frac{\partial w0}{\partial x0} = \frac{\partial w0}{\partial z}\frac{\partial z}{\partial y}\frac{\partial y}{\partial x0}$$



$$\frac{\partial w0}{\partial x1} = \frac{\partial w0}{\partial z}\frac{\partial z}{\partial y}\frac{\partial y}{\partial x1}$$



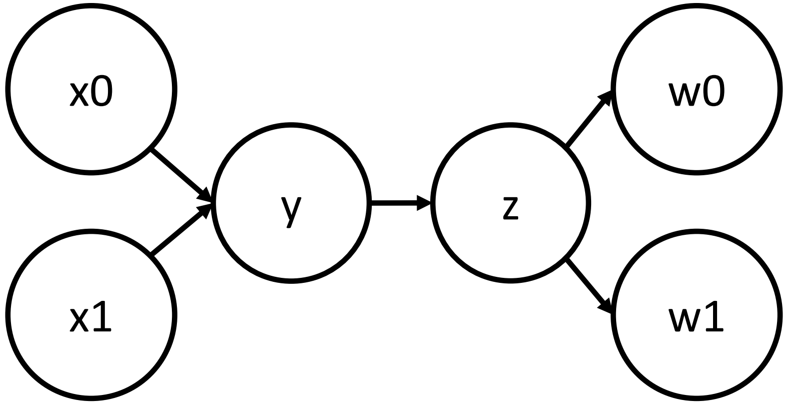
$$\frac{\partial w1}{\partial x0} = \frac{\partial w1}{\partial z}\frac{\partial z}{\partial y}\frac{\partial y}{\partial x0}$$



$$\frac{\partial w1}{\partial x1} = \frac{\partial w1}{\partial z}\frac{\partial z}{\partial y}\frac{\partial y}{\partial x1}$$

AD step-by-step. Forward Mode

```
dx0dx = {1, 0}
dx1dx = {0, 1}
y = f(x0, x1)
dydx = df(x0, dx0dx, x1, dx1dx)
z = g(y)
dzdx = dg(y, dydx)
w0, w1 = l(z)
dw0dx, dw1dx = dl(z, dzdx)
```



AD step-by-step. Reverse Mode

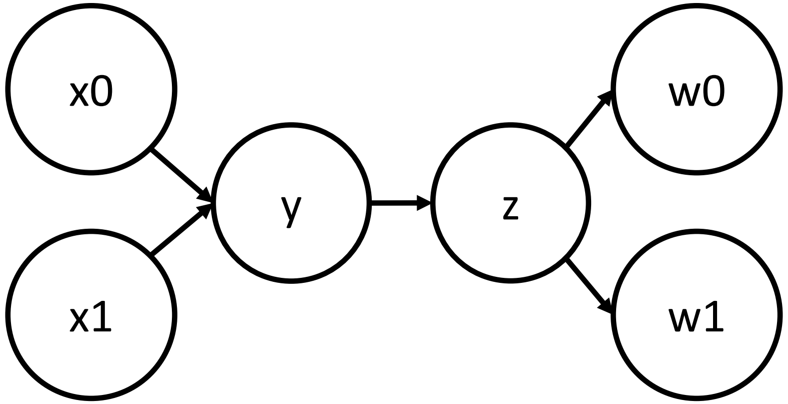
```
y = f(x0, x1)
z = g(y)
w0, w1 = l(z)

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 mater 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.

Implementation Techniques

Components of an AD-Aware System

- Core AD Transformation
  How do we generate a derivative? Usually is a transformation pass over a data structure representing the code. Challenge: performance
- User Interface/API
  How do we request and use a derivative? Usually is a trigger for the AD transformation. Challenge: cross-translation unit support, tool interoperability.
- Framework
  How do we express a solution apt to AD? Usually is a complex system that enables differentiable programming, that is provides users with facilities to solve problems end-to-end. Challenge: complexity, tools work well for a single domain.

Core AD Transformation

AD tools can be categorized by how much work is done before program execution:

- Tracing/taping/operator overloading – constructs and processes the computational graph at the time of execution, each time a function is invoked.
  Records the linear sequence of computation operations at runtime into a tape (or Wengert list). The control flow is flattened to produce a derivative. A typical implementation is via operator overloading, defining a special floating type with overloaded elementary operations. Algorithms use this type to trigger differentiation by calling a special function. There are numerous C++ AD tools based on tracing including ***ADOL-C, CppAD, Adept,*** **Zygote.jl, Diffractor.jl and JAX**.
- Source Transformation – constructs the computation graph and produces a derivative function at ahead of time.
  More compiler optimizations can be applied, such as reorganizing or evaluating simple constant expressions at compile time and common subexpression elimination. Source trans- formation is more difficult to implement as it requires a significant investment in developing and maintaining a language parser. ***Tapenade*** *is an example for a source transformation tool with custom parsers for C and Fortran.*
- Compiler-based Source Transformation – constructs and transforms the computation graph as part of the translation phase.
  Historically, toolmakers made trade offs between ease of use, performance and ease to integration. AD now benefits from better language support to avoid such trade offs. Recent advancements of production quality compilers like Clang allow tools to reuse the language parsing infrastructure. ***ADIC, Enzyme and CLAD*** *are compiler-based tools using source transformation.*

AD Frameworks

Other systems offer AD-aware environments to differentiate subsets of a language for domain-specific purposes:

- Halide offers AD-aware environment for image and array processing in C++
- JAX is also an AD system that differentiates a sublanguage of Python oriented towards ML
- Dex aims to give better AD asymptotic and parallelisation guarantees than JAX for loops with indexing
- Swift and Julia integrate AD deeply into the language itself
- Tensorflow/PyTorch/Theano/…/

Differentiable Programming

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]

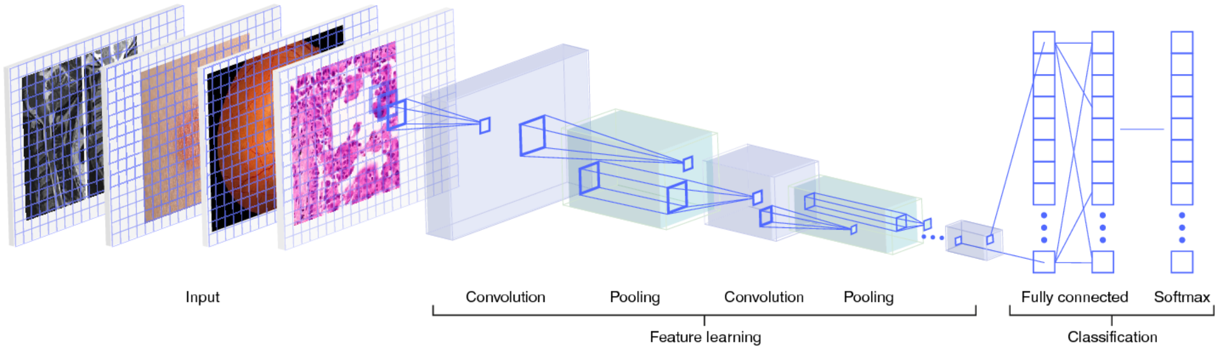
- Deep learning drives recent advancements in automatic differentiation
- AD is useful also in bayesian inference, uncertainty quantification, modeling, simulation
- The concept of AD dates back from dual number algebra from 19^{th} century
- In 1970’s AD was used to estimate roundoff errors
- In the 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), …

Deep Learning & Automatic Differentiation

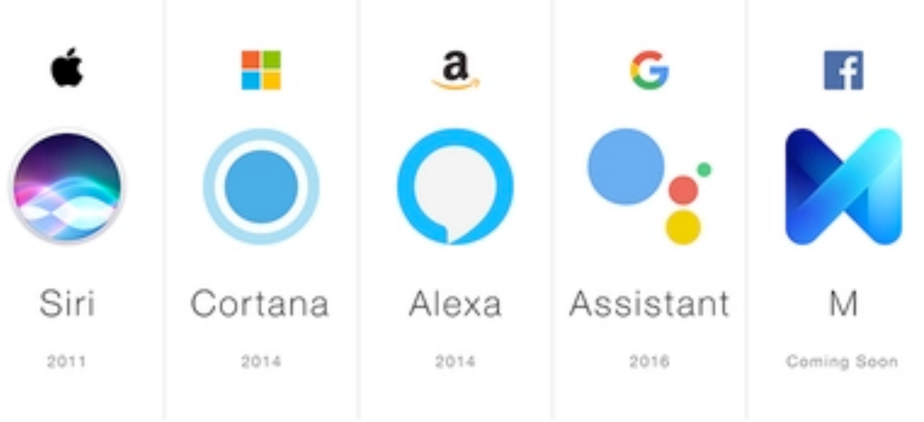
Imagined by GAN,
ThisPersonDoesNotExist.com

Image colorization

Tesla Autopilot, tesla.com



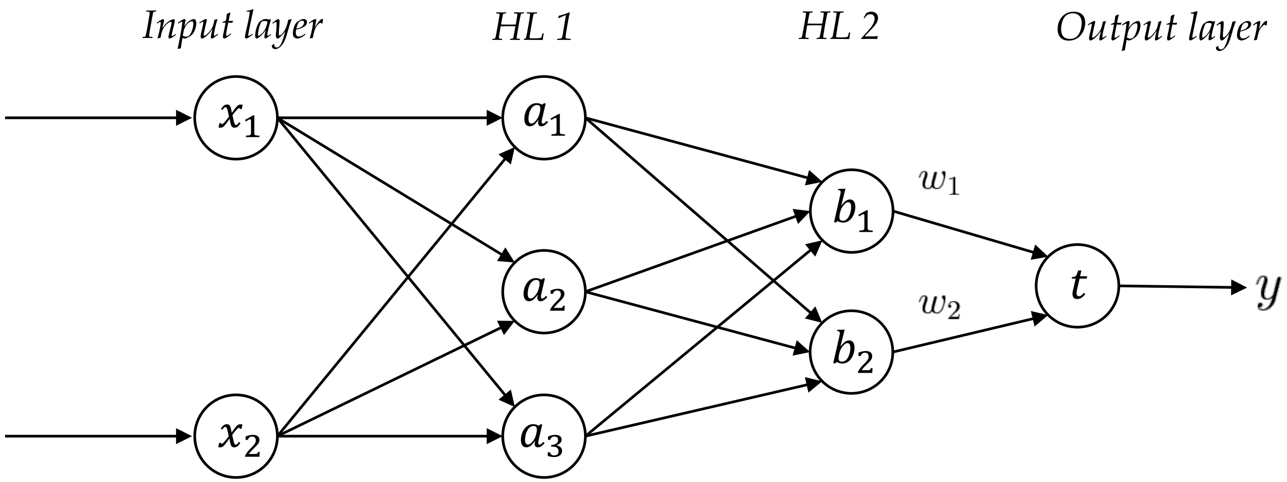
Medical Imaging, CNN, A. Esteva et al, A guide to deep learning in healthcare



Speech Recognition

Backpropagation As Data Flow Optimization

Error at output, the error between observed and desired state. Computed from the output y and seen desired output t.



$$y = \phi\left(\sum_i w_i x_i\right) \qquad E = \frac{1}{2}(t - y)^2$$

Forward pass – make a prediction

$$\Delta w_i = \eta \frac{\partial E}{\partial w_i}$$

Calculate Loss

Training pattern is fed, forward generating corresponding output

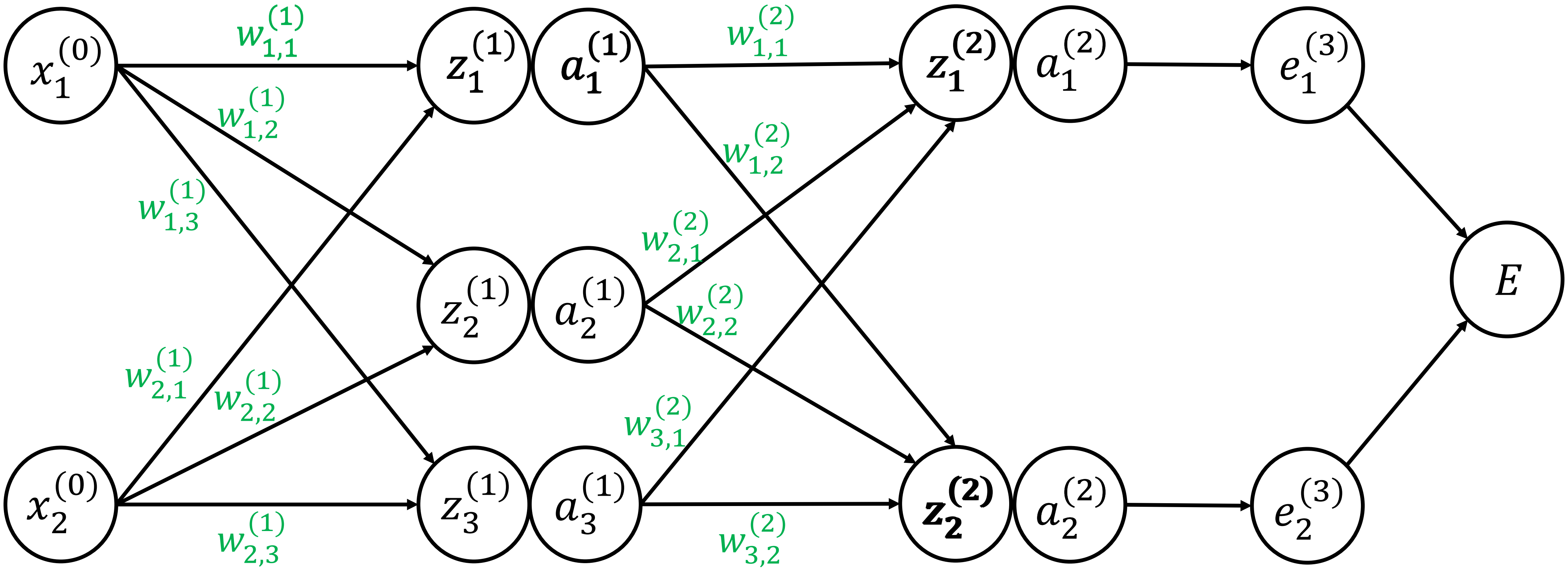
Backpropagation – adjust the weights to minimize loss

The error propagates back, through updates of the subtracted gradient ratio from the weights.

x_i, w_i, ϕ, η *are inputs, input weights, activation function and learning rate of the neuron*

Backpropagation

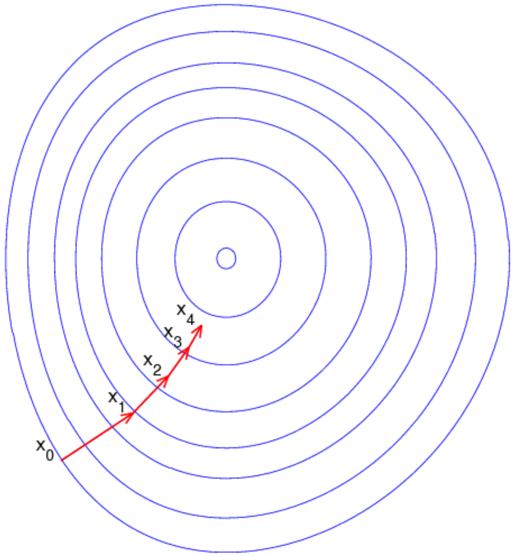
$$\frac{\partial}{\partial} = \left(\frac{\partial}{\partial}\frac{\partial}{\partial}\frac{\partial}{\partial} + \frac{\partial}{\partial}\frac{\partial}{\partial}\frac{\partial}{\partial}\right)\frac{\partial}{\partial}\frac{\partial}{\partial}$$

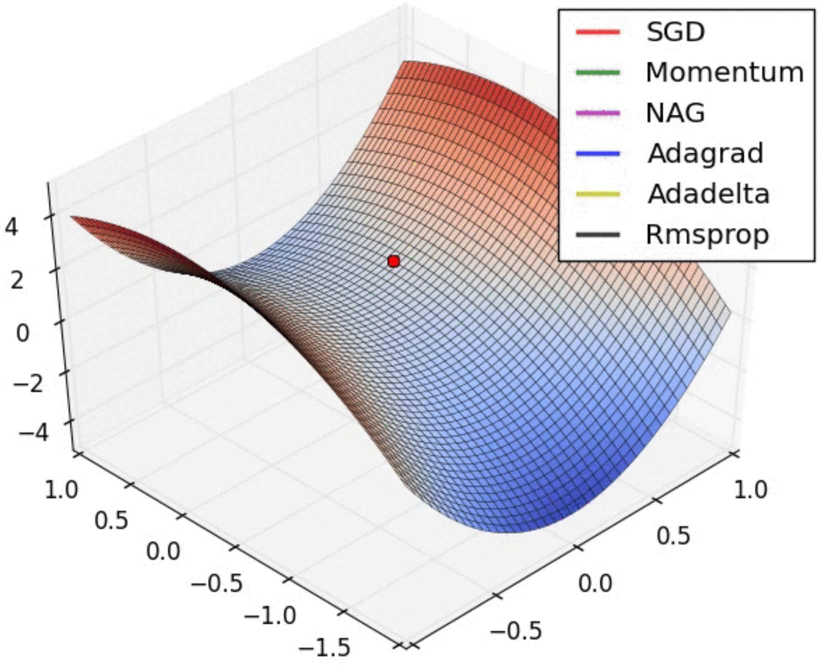


Gradient Descent

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

$$\nabla f(x1, \dots, x_n) = \begin{bmatrix} \frac{\partial f}{x_1}(x1, \dots, x_n) \\ \cdot \\ \cdot \\ \cdot \\ \frac{\partial f}{x_n}(x1, \dots, x_n) \end{bmatrix}$$

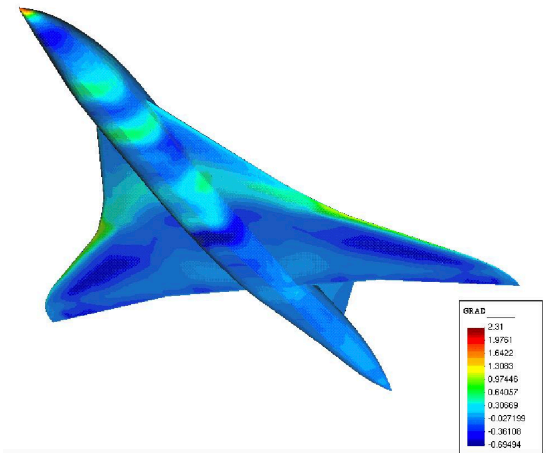




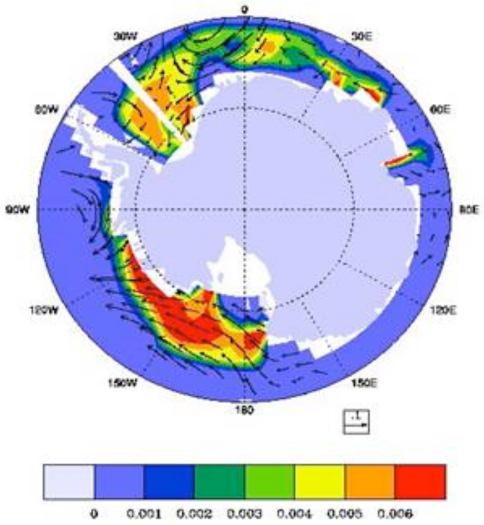
Plot credits: https://ruder.io/optimizing-gradient-descent/

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

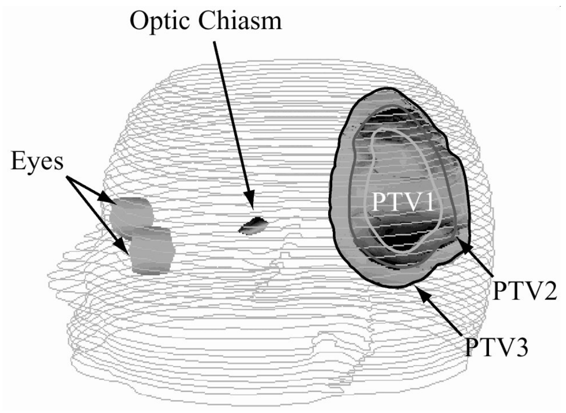
Uses of AD Beyond Python & ML



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

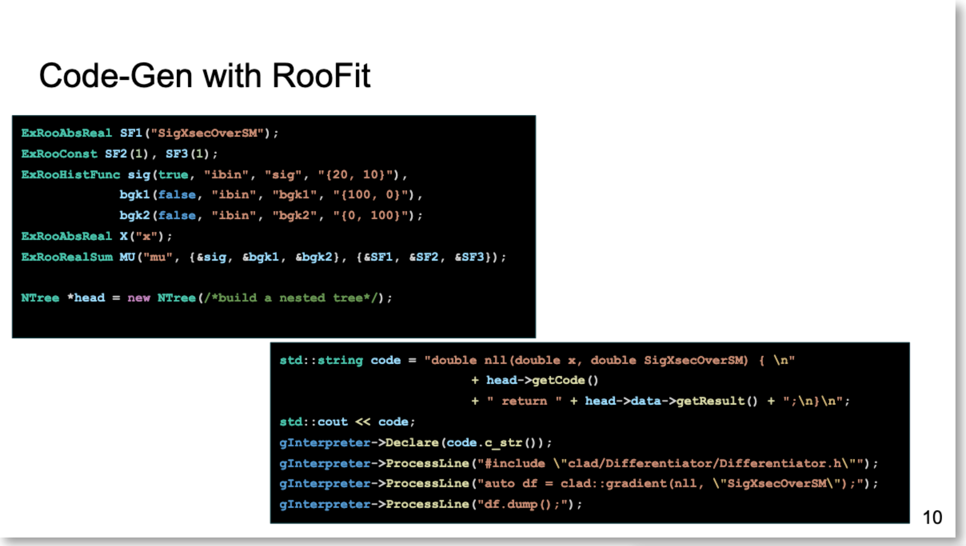


Sensitivities of a Global
Sea-Ice Model, Climate, Jong G. Kim et al

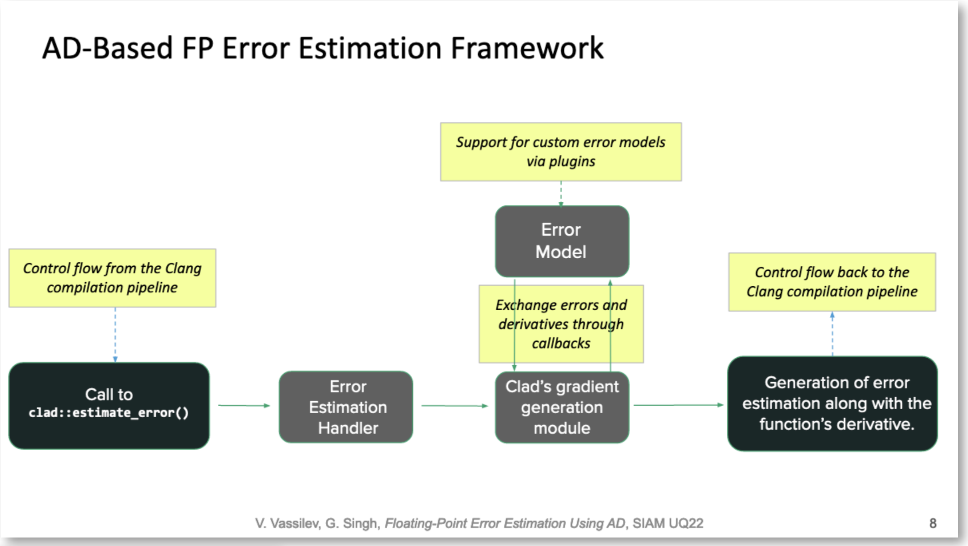


Intensity Modulated Radiation Therapy, Biomedicine, Kyung-Wook Jee et al

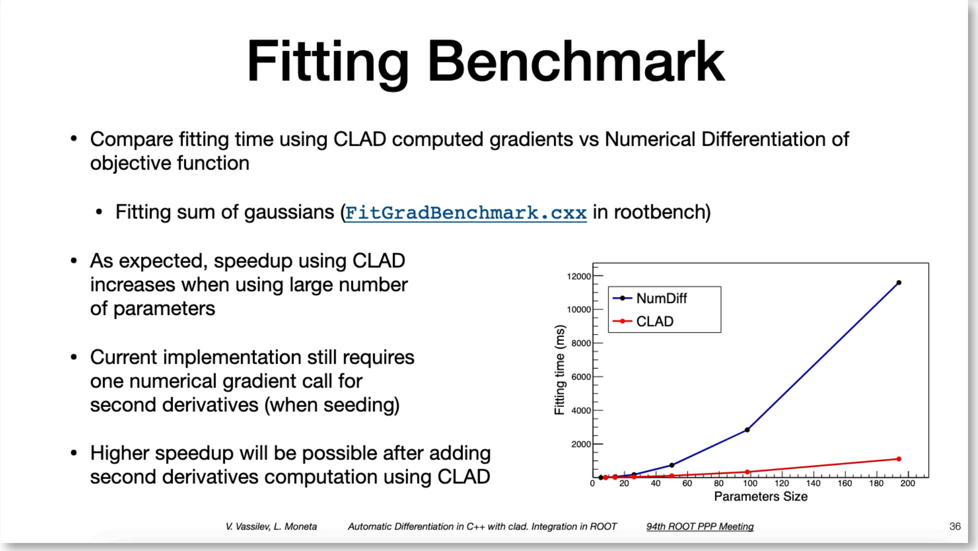
AD in ROOT and Beyond



RooFit & AD prototype, G. Singh et al.



Floating point error estimation& AD, G. Singh et al.



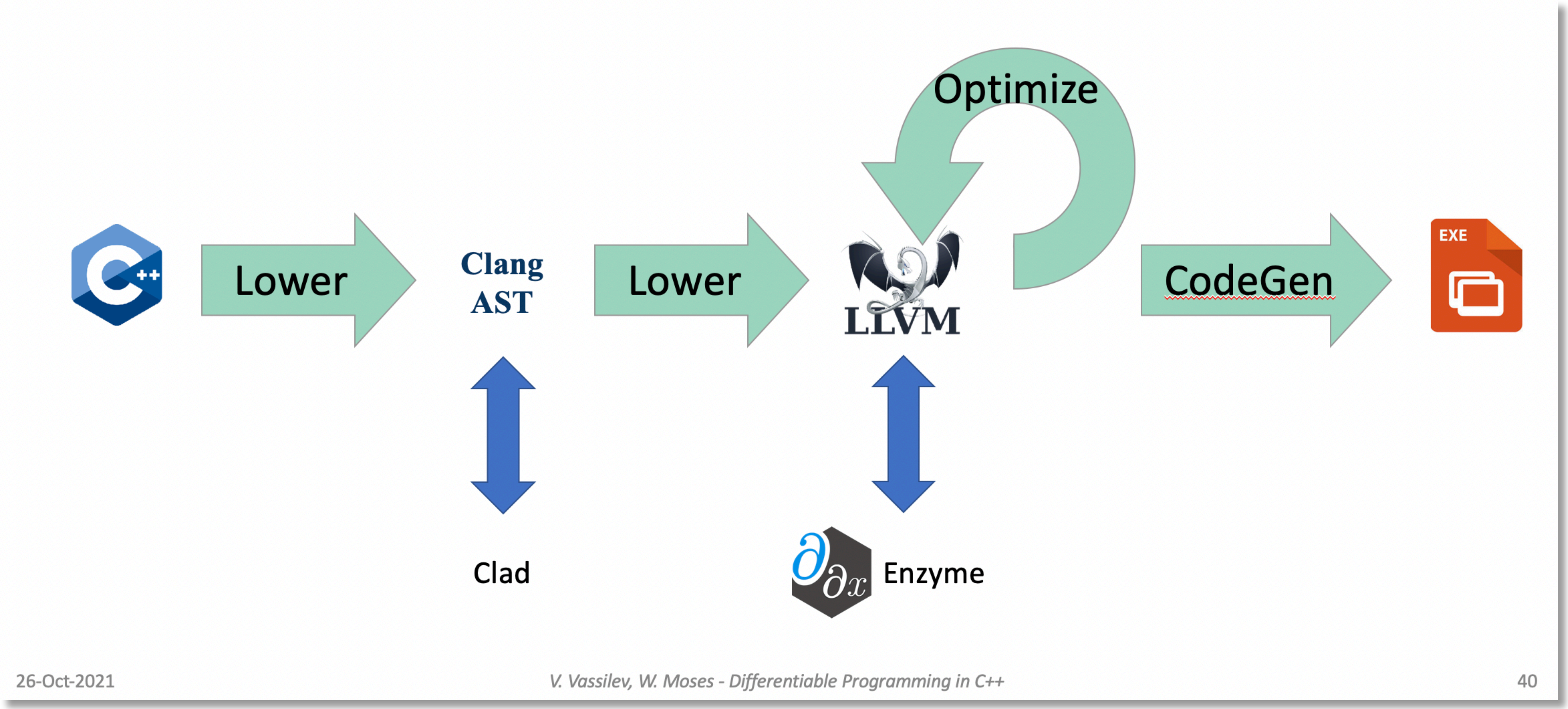
ROOT hist fitting & AD, L. Moneta et al.

AD Community & Trends

AD Community

- Relatively small and well-connected community
- autodiff.org – a community portal which is mostly kept up-to-date
- Once a year an EuroAD workshop
- Next year there will be a major AD event taking place in ANL
- juliadiff.org – a Julia AD community which captures the uprise of AD infrastructure in Julia

LLVM-Based Source Transformation AD Tools



Clad. Usage

```cpp
// 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;
}
```

Enzyme. Usage

```
// clang test.c -S -emit-llvm -o input.ll -O2 -fno-vectorize -fno-slp-vectorize -
fno-unroll-loops

#include <stdio.h>
extern double __enzyme_autodiff(void*, double);
double square(double x) {
    return x * x;
}
double dsquare(double x) {
    // This returns the derivative of square or 2 * x
    return __enzyme_autodiff((void*) square, x);
}
int main() {
    for(double i=1; i<5; i++)
        printf("square(%f)=%f, dsquare(%f)=%f", i, square(i), i, dsquare(i));
}
```

Conclusion

Differentiable Programming a programming paradigm which relies on well developed theory and technology. It can enable gradient descent optimizations and make our systems more sensitive or resistant to particular data inputs.

I personally think that differentiable programming will disrupt science modeling and simulation.

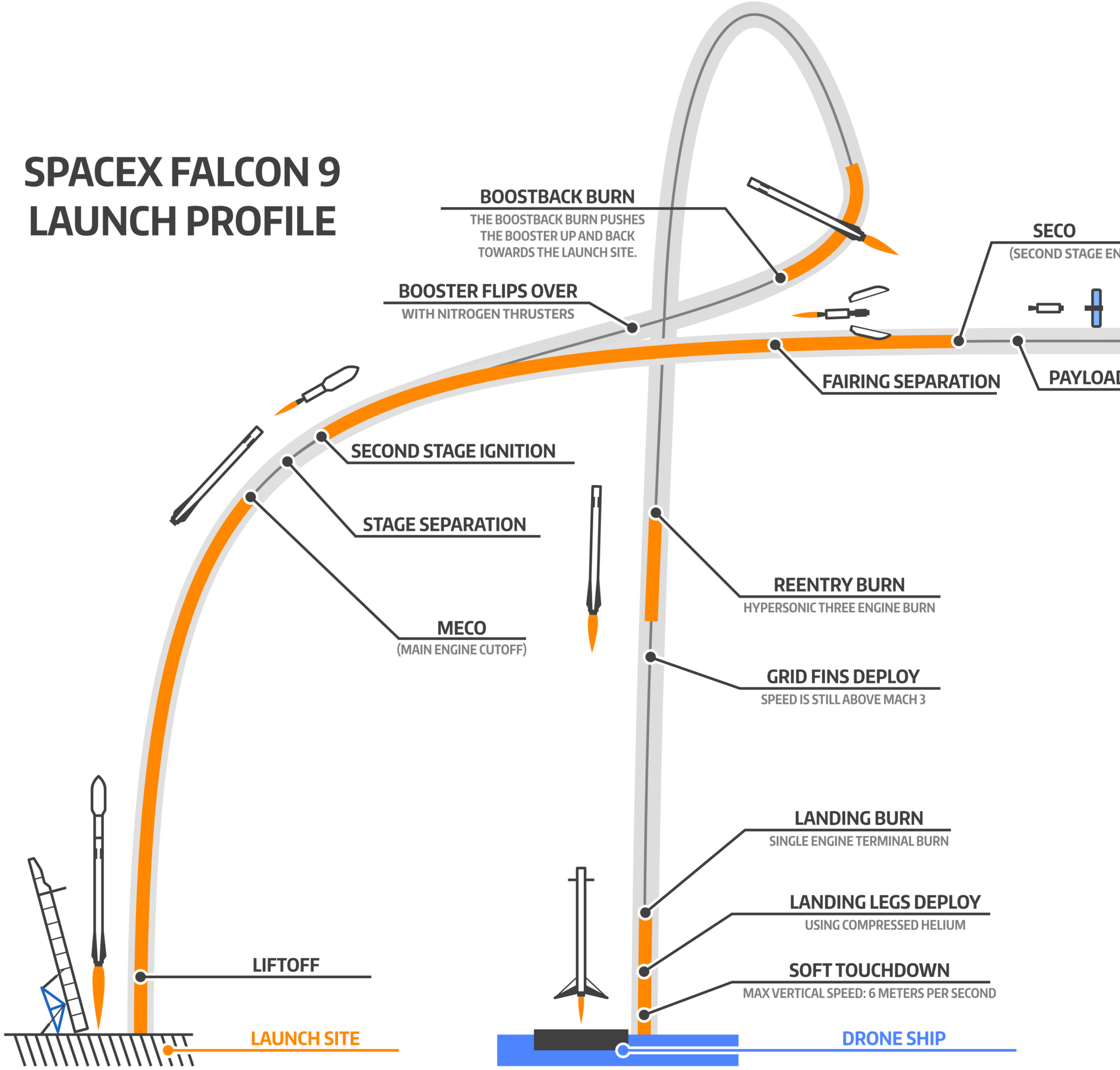
Can HEP re-ogranize its software to benefit from this paradigm beyond the canonical use of ML?

Thank you!

Supplementary Slides

Optimal Control

Can steer a process towards a reference trajectory automatically?



Credit: imgur.com

Credit: Reddit.com

Controllir

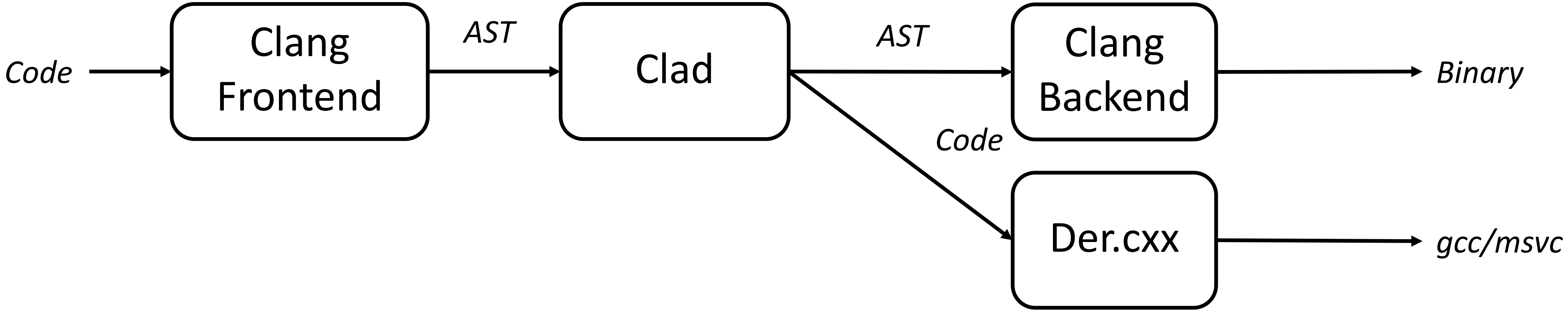
Credit: Official SpaceX Photos

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

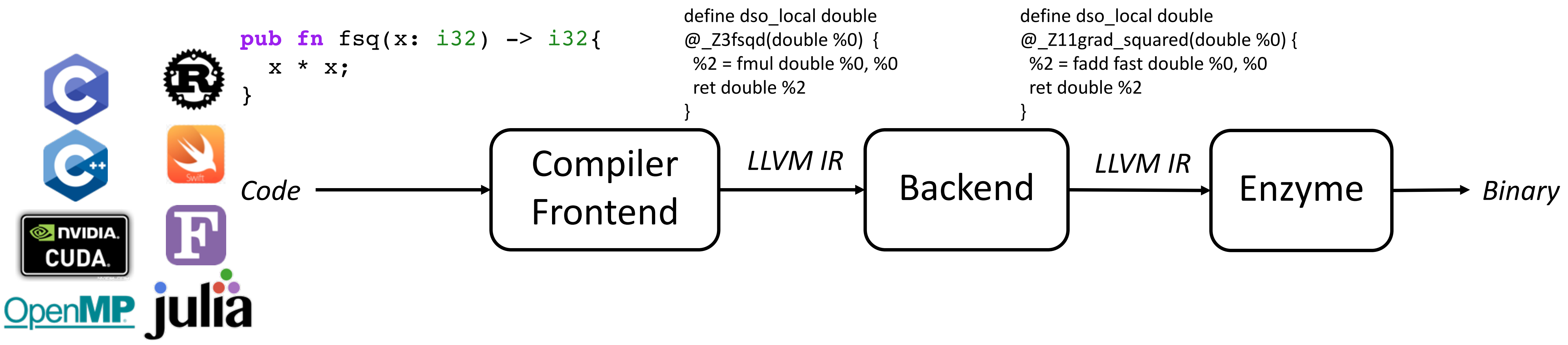
```
FunctionDecl 0x7f7f801dbff8 <<invalid sloc>> <invalid sloc> f_darg0 'double (double)'
|-ParmVarDecl 0x7f7f801dc090 <<invalid sloc>> <invalid sloc> used x 'double'
`-CompoundStmt 0x7f7f801dc3d0 <<invalid sloc>>
  |-DeclStmt 0x7f7f801dc190 <<invalid sloc>>
  | `-VarDecl 0x7f7f801dc118 <<invalid sloc>> <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'
```

```
FunctionDecl fsq '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;
}
```




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


pub fn fsq(x: i32) -> i32{
x * x;
}
Code
Compiler Frontend
define dso_local double
@_Z3fsqd(double %0) {
%2 = fmul double %0, %0
ret double %2
}
LLVM IR
Backend
define dso_local double
@_Z11grad_squared(double %0) {
%2 = fadd fast double %0, %0
ret double %2
}
LLVM IR
Enzyme
Binary