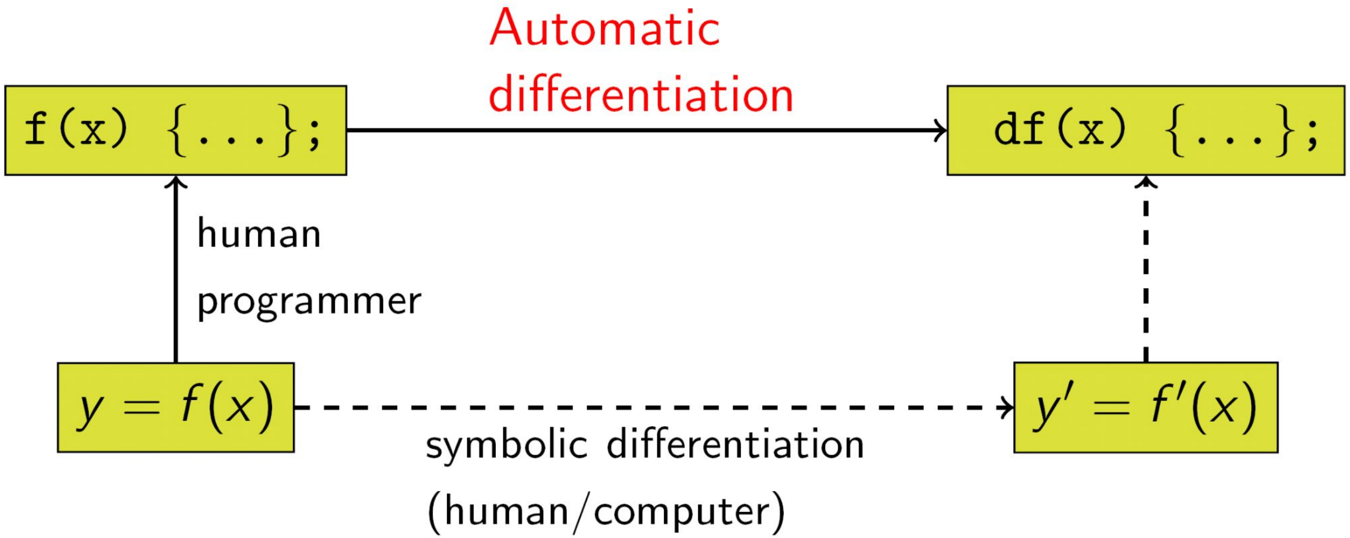
Extending Clad, an automatic differentiation system for C++

Vaibhav Thakkar
Supervisor: Dr Vassil Vassilev (CERN / Princeton University)

Brief Intro of Automatic Differentiation



Example

$$f(x) = e^{e^{e^{e^{e^x}}}}$$

Symbolic via Wolfram Alpha

$$\frac{d}{dx}\left(e^{e^{e^{e^{e^x}}}}\right) = e^{x+e^{e^{e^{e^x}}}+e^{e^{e^x}}+e^{e^x}+e^x}$$

Handcode

Figure out the analytical fn

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

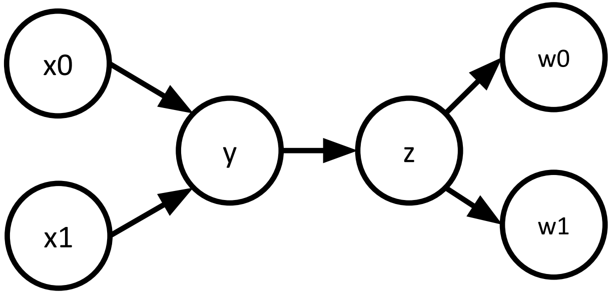
AD

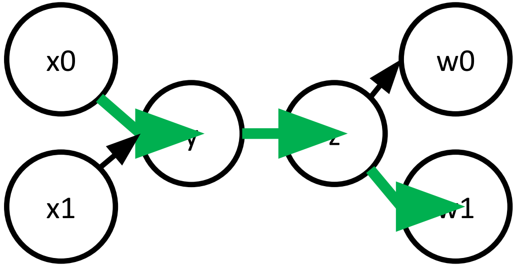
```cpp
double f_dx(double x, int N=5) {
  double result = x;
  double d_result = 1;
  for (unsigned i = 0; i < N; i++) {
     result = std::exp(result);
     d_result *= result;
  }
  return d_result;
}
```

Reference: *V. Vassilev – Accelerating Large Scientific Workflows Using Source Transformation Automatic Differentiation*

Crux of AD - Computational graph + Chain rule

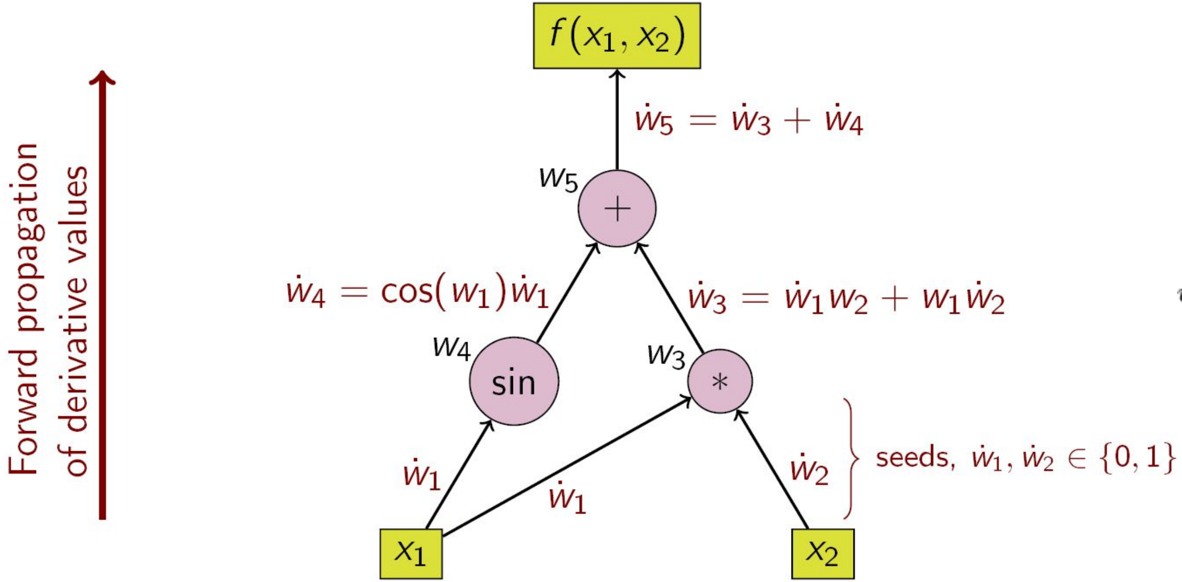
```
y = f(x0, x1)
z = g(y)
w0, w1 = l(z)
```





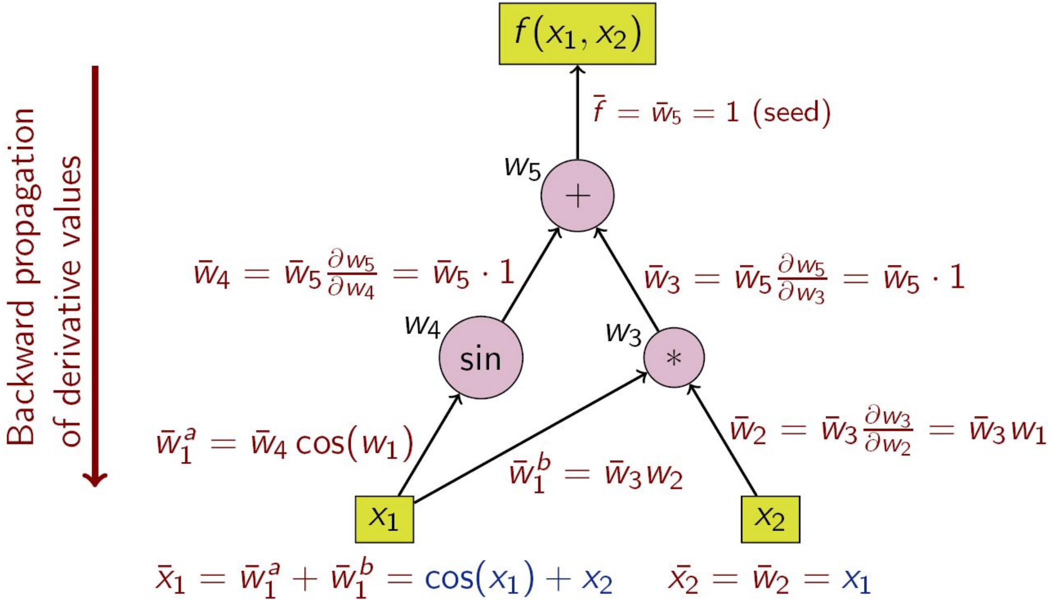
$$\frac{\partial w1}{\partial x0} = \frac{\partial w1}{\partial z}\frac{\partial z}{\partial y}\frac{\partial y}{\partial x0}$$

Forward mode AD



$$\dot{w}_i = \sum_{j \in \{\text{predecessors of i}\}} \frac{\partial w_i}{\partial w_j} \dot{w}_j$$

Reverse mode AD



$$\bar{w}_i = \sum_{j \in \{\text{successors of i}\}} \bar{w}_j \frac{\partial w_j}{\partial w_i}$$

About Clad

- ***Source transformation based AD tool for C++***
    - *Runs at compile time* - clad generates the code for derivatives using the Abstract Syntax Tree (AST) of the original / primal function as the computational graph.
    - *Implemented as a Clang plugin* - uses the APIs and robust infrastructure of LLVM/Clang for traversing over the parsed graph and generating the derivative code.
- ***Supports both forward and reverse mode, also provide functionality for higher order derivatives, Jacobians and Hessians.***

About Clad - usage example

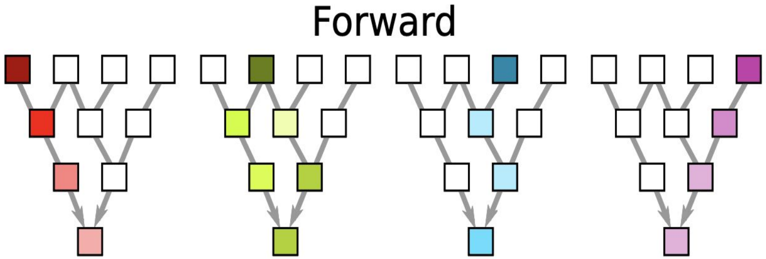
```
#include "clad/Differentiator/Differentiator.h"
#include <iostream>

double f (double x, double y) {
  return x*y;
}

double main() {
  // Call clad to generate the derivative of f wrt x.
  auto f_dx = clad::differentiate(f, "x");

  // Execute the generated derivative function.
  std::cout << f_dx.execute(/*x=*/3, /*y=*/4) << std::endl;
  std::cout << f_dx.execute(/*x=*/9, /*y=*/6) << std::endl;

  // Dump the generated derivative code to stdout.
  f_dx.dump();
}
```

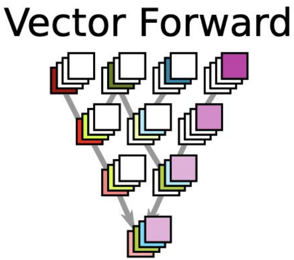
My major contributions (past, present and future)

Complete list of my contributions can found here: https://github.com/vgvassilev/clad/commits?author=vaithak

Vector Forward mode AD for higher order derivatives

- Vector forward mode AD allows computing the entire gradient in a single vectorized forward pass.
    - *This was implemented in Clad in my Google Summer of Code project*





- Can we use this for efficient computation of the Hessian / Jacobian ?
    - How about computing just the diagonal matrix of the Hessian?

Pointer support in reverse mode AD

- Pointers are a separate beast - specifically for reverse pass
  - Memory allocations and deallocations - when exactly can we deallocate a memory in reverse pass?
  - Keeping track of not just the value in the pointer (the address), but also the value(s) inside that address.

- Still in progress and improving incrementally.

Differentiating lambda functions

- Main benefit of Clad is to allow users to enjoy the richness of C++ and give them the power of differentiability.
  - This means we have to add support for modern C++ features, includes lambda expressions, standard library functions and containers, ...
- Clad already contains some support for lambdas, mainly when the primal function is a lambda itself. Need to improve it to conditions where lambdas are used as a call expression inside another function.

```
double primal_func(double i, double j) {
 auto myLambda = [](double t) {
   return t*t + 1.0;
 };
 return i + myLambda(j);
}
```

Thank you

Questions or Comments ?