Clad - Clang plugin for Automatic Differentiation

Alex Efremov

What Clad does

- Clad performs **automatic differentiation** on C++ functions
- For a C++ function, creates another C++ function that computes its derivative(s)

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

Clad

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

What automatic differentiation is

- Technique for evaluating the derivatives of mathematical functions
- Applies differentiation rules to each arithmetical operation in the code

```
double c = a + b;
```

➔

```
double d_c = d_a + d_b;
```

```
double c = a * b;
```

➔

```
double d_c = a * d_b + d_a * b;
```

...

What automatic differentiation is

- Not limited to closed-form expressions
- Can take **derivatives of algorithms** (conditionals, loops, recursion)

Example: loops

```
double pow(double x, int n) {
    double r = 1;
    for (int i = 0; i < n; i++)
        r = r * x;
    return r;
}
```

```
double pow_darg0(double x, int n) {
    double d_r = 0;
    double r = 1;
    for (int i = 0; i < n; i++) {
        d_r = d_r*x + r*1;
        r = r*x;
    }
    return dr;
}
```

What automatic differentiation is

- Alternative to numerical differentiation

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

- Creates a function that computes the derivative(s) for you
- Without additional precision loss
- Small constant factor more arithmetic operations than the original function

Automatic differentiation in Clad

- At the moment supports functions with:
  - **multiple** (*scalar*) inputs
  - **single scalar** output value

$$f : \mathbb{R}^n \to \mathbb{R}$$

```
double f(double x0, double x1, ..., double xn);
```

- Will be extended soon with:
  - **vector** inputs

```
double f(vector<double> x);
```

```
double f(double* x);
```

- Can be extended with:
  - multiple outputs

$$f : \mathbb{R}^n \to \mathbb{R}^{m}$$

```
vector<double> f(vector<double> x);
```

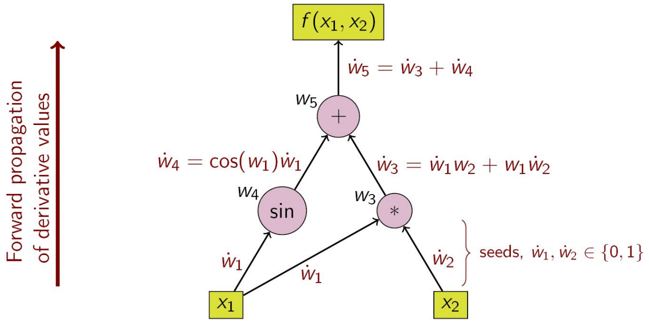
Automatic differentiation in Clad

- For $f : \mathbb{R}^n \to \mathbb{R}$ can generate:
  - single derivative $\dfrac{\partial f}{\partial x_i}$: `clad::differentiate(f, i);`
  - gradient $\nabla f = \left(\dfrac{\partial f}{\partial x_1}, \ldots, \dfrac{\partial f}{\partial x_n}\right)$: `clad::gradient(f);`
- Supports both ***forward*** and ***reverse*** mode AD:
  - `clad::differentiate` uses forward mode
  - `clad::gradient` uses reverse mode

Forward mode

`clad::differentiate`

- **Forward mode AD** algorithm computes derivatives w.r.t. any **(single)** variable
- *W.r.t. all outputs at once (only single output is supported for now)*
- Constant factor overhead:
  - **At most 3 times** more arithmetic operations than the original function
- Must be run **N** times separately if you have **N** input variables and need gradient
  - Use reverse mode instead



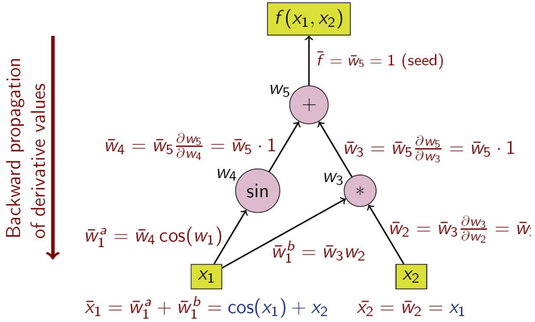
$$f(x_1, x_2) = \sin(x_1) + x_1 x_2$$

[Wikipedia, Automatic differentiation]

Reverse mode

`clad::gradient`

- **Reverse mode AD** computes gradients (w.r.t to **all** inputs at once)
- **At most 4 times** more arithmetic operations than the original function
    - **No matter how many inputs you have**



$$f(x_1, x_2) = \sin(x_1) + x_1 x_2$$

[Wikipedia, Automatic differentiation]

How Clad works

```cpp
double f(double x) {
      return x * x;
}
```

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'
```

Clad

derivative AST

codegen

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

- Clad is a **Clang compiler plugin**
- Performs C++ **source code transformation**
- Operates on Clang AST
- AST transformation with `clang::StmtVisitor`

How to use Clad

- You will need **clang-5.0** (doesn’t work with newer versions yet)
- Download and build **Clad**: https://github.com/vgvassilev/clad
- Attach **libclad.so/libclad.dylib** to clang to compile with enabled Clad:

```
clang -cc1 -x c++ -std=c++11 -load libclad.so -plugin clad SourceFile.cpp
```

In your C++ source

- `#include "clad/Differentiator/Differentiator.h"`
- You want to differentiate some existing C++ function:

```
double f(double x, ...) {...}
```

No source modification needed

Definition MUST be visible for the compiler (or interpreter - Cling), otherwise we cannot analyze it

- Use `clad::differentiate(f, ARG);` or `clad::gradient(f, ARG);`
- Clad will detect that call and process **f** in compile time to create the derivative

In your C++ source

- ```cpp
  #include "clad/Differentiator/Differentiator.h"
  ```

- Or in **ROOT:**
  ```cpp
  #include "Math/CladDerivator.h"
  ```
  ```

Using clad::differentiate

- ```
  auto df = clad::differentiate(f, ARG);
  ```
  - **f** is a pointer to your function
  - `ARG` is either:
    - 1) integral literal **I**, indicating the index of independent variable
      ```
      clad::differentiate(f, 0);
      ```
    - 2) string literal with the name of independent variable (as written in the definition)
      ```
      clad::differentiate(f, “x”);
      ```
- Will generate a function **f_dargI**, with the same signature as **f**
  - **f_dargI** returns the value of the derivative for given inputs
      ```

Using clad::differentiate

```
double f(double x) { return x*x; }
// Will be generated by Clad: double f_darg0(double x) { return 1*x + x*1; }

int main() {
  auto df = clad::differentiate(f, 0);
  // or: auto df = clad::differentiate(f, “x”);
  // will generate the function f_darg0 in the current namespace
  // df is a functor object with pointer to f_darg0 inside
  double val = f_darg0(2.0);
  // or: double val = df.execute(2.0);
  // val is 4.0
}
```

Using clad::gradient

- ```
  auto gf = clad::gradient(f, ARG);
  ```
  - **f** is a pointer to your function
  - `ARG` is *optional*:
    - string literal with comma-separated names of independent variables

      ```
      clad::gradient(f, “x, y, z”);
      ```

    - if not provided, will **f** will be differentiated w.r.t. to all parameters

      ```
      clad::gradient(f);
      ```

- Will generate a function **f_grad**, with the the following signature

```
void f_grad(/* ..., same inputs as in f*/, double* _result);
```

  - **_result** is an output parameter to get gradient vector
```

Using clad::gradient

```
double f(double x, double y, double z) { return ...; }
// Will be generated by Clad:
// void f_grad(double x, double y, double z, double* _result) {
//  _result[0] += ...; _result[1] += ...; _result[2] += ...; ... }

int main() {
  auto gf = clad::gradient(f);
  // or specify subset of params: auto df = clad::differentiate(f, "x, z");
  // will generate the function f_grad in the current namespace
  // You must pre-allocate and initialize storage for the gradient:
  double result[3] = {};
  f_grad(1.0, 2.0, 3.0, result);
  // or: gf.execute(1.0, 2.0, 3.0, result);                         }
```

Custom derivatives

- Sometimes function's definition is not available to you (e.g. part of a library)
- Or you know efficient analytical expression

```
double f(double x) { ... ; y = std::sin(x); ...  }
```

- Define a custom derivative yourself:

```
namespace custom_derivatives { namespace std {
  double sin_darg0(double x) { return ::std::cos(x); }
}}
```

- Will be detected by Clad:

```
double f_darg0(double x) { ... ; dy = custom_derivatives::std::sin(x); ...   }
```

- Derivatives for some math functions (from **<cmath>**) defined in **"clad/Differentiator/BuiltinDerivatives.h"**

How the generated C++ code looks like?

- `auto df = clad::differentiate(f, ...);`
- Can print the generated code: `df->dump();`

```
double pow(double x, int n) {
    double r = 1;
    for (int i = 0; i < n; i++)
        r = r * x;
    return r;
}
```

```
double pow_darg0(double x, int n) {
    double _d_x = 1; int _d_n = 0;
    double _d_r = 0; double r = 1;
    { int _d_i = 0;
      for (int i = 0; i < n; i++) {
        _d_r = _d_r * x + r * _d_x;
        r = r * x;
      }
    }
    return _d_r;
}
```

Current state

Support of C++ constructs:

- Tested with built-in floating point types: **float**, **double**
- In principle, should work with user-defined scalar types, needs testing
- Arithmetic operators, function calls, variable declarations, if statements …
- In **forward** mode:
  - variable mutation (reassignments), for loops

TODO:

- In **reverse** mode: variable mutation (reassignments), for loops
- Arrays/vectors, struct/class methods, custom datastructures…
- Occasional missing C++ constructs
- Rigorous documentation, error/warning handling

Future work

- Better integration into **ROOT** (through **TFormula**)
- Benchmarking, replace numerical differentiation in **ROOT**
- Better API

Towards better API

Done:

- Selecting subset of function's parameters: `clad::gradient(f, “x, y, z”);`

Why string literals?

- Some advanced things are hard to represent with standard C++ syntax

TODO:

- Differentiate w.r.t. class field: `clad::differentiate(&C::m, “x, y, C::a, C::b”);`
- Labeling input as a vector, selecting subinterval:

```
clad::gradient(f, “x[:], y[0:100], z[0:n]”);
```

Towards better API

Returning gradients:

```
auto gf = clad::gradient(f, “x, z”);
double result[2] = {};
gf.execute(1.0, 2.0, 3.0, result);
double dx = result[0]; double dy = result[1];
```

Now:

- Must allocate gradient storage
- Mapping between parameter and index in the result array

Proposal:

- Create and return special type (struct) with named fields:

```
auto gf = clad::gradient(f, “x, z”);
auto result = gf.execute(1.0, 2.0, 3.0);
double dx = result.dx; double dy = result.dy;
```

Towards better API

Proposal:

- Create and return special type (struct) with named fields:

```
auto gf = clad::gradient(f, "x, z");
```

- At this point the compiler knows that x and y of type T are requested
- At compile time create special struct type:

```
struct f_grad_x_z_return_type_xxx {T dx, dx;};
```

- Generate gradient function for the requested parameters:

```
F_grad_x_z_return_type_xxx f_grad_x_z_xxx__(double x, double y, double z) {
  F_grad_x_z_return_type_xxx result{};
  ...
  result.dx += ...; result.dy += ...;
  return result; }
```

- **Clad: https://github.com/vgvassilev/clad**