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)

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

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

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

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

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

```c
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**

```c
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 d_r;
}
What automatic differentiation is

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

\[ f'(x) \approx \frac{f(x+h) - f(x)}{h} \]
At the moment supports functions with:
- **multiple** (*scalar*) inputs
- **single scalar** output value

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

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

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

Can be extended with:
- **multiple outputs**

```cpp
double f(double x0, double x1, ..., double xn);
double f(vector<double> x);
double f(double* x);
```

```cpp
vector<double> f(vector<double> x);
```
Automatic differentiation in Clad

- For $f : \mathbb{R}^n \rightarrow \mathbb{R}$ can generate:
  - single derivative $\frac{\partial f}{\partial x_i}$: \texttt{clad::differentiate}(f, i);
  - gradient $\nabla f = \left( \frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n} \right)$: \texttt{clad::gradient}(f);

- Supports both \textit{forward} and \textit{reverse} mode AD:
  - \texttt{clad::differentiate} uses forward mode
  - \texttt{clad::gradient} uses reverse mode
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 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]
double f(double x) {
    return x * x;
}
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](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

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

- Or in ROOT:
  
  `#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 `l`, indicating the index of independent variable
      ```cpp
      clad::differentiate(f, l);
      ```
    - 2) string literal with the name of independent variable (as written in the definition)
      ```cpp
      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
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 following signature
  - `void f_grad(/* ..., same inputs as in f*/, double* _result);`
  - `_result` is an output parameter to get gradient vector
Using \texttt{clad::gradient}

double \texttt{f}(double \texttt{x}, double \texttt{y}, double \texttt{z}) \{ return ...; \} 

\texttt{// Will be generated by Clad:} 
\texttt{// void \texttt{f} \_\texttt{grad}(double \texttt{x}, double \texttt{y}, double \texttt{z}, double* \_\texttt{result}) \{ \_\texttt{result}[0] += ...; \_\texttt{result}[1] += ...; \_\texttt{result}[2] += ...; ... \} \}

\texttt{int main() \{ \}
\texttt{\hspace{1em}auto \texttt{gf} = \texttt{clad::gradient}(f);} 
\texttt{\hspace{1em}// or specify subset of params: auto \texttt{df} = \texttt{clad::differentiate(f, \texttt{"x, z"});}}
\texttt{\hspace{1em}// will generate the function \texttt{f} \_\texttt{grad} in the current namespace} 
\texttt{\hspace{1em}// You must pre-allocate and initialize storage for the gradient:} 
\texttt{\hspace{1em}double \texttt{result}[3] = \{\};} 
\texttt{\hspace{1em}f \_\texttt{grad}(1.0, 2.0, 3.0, \texttt{result});} 
\texttt{\hspace{1em}// or: \texttt{gf.execute}(1.0, 2.0, 3.0, \texttt{result});} 
\texttt{\}}
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();

```cpp
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;
}
```
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: \texttt{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: \texttt{clad::differentiate(&C::m, “x, y, C::a, C::b”)};

● Labeling input as a vector, selecting subinterval:

\texttt{clad::gradient(f, “x[:], y[0:100], z[0:n]”)};
Towards better API

Returning gradients:

```cpp
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:

```cpp
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, dy;
  }
  ```
- 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