Clad - Clang plugin for Automatic Differentiation

Alex Efremov
What Clad does

- Clad performs **automatic differentiation** on C++ functions
- Based on **source code transformation**
- 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;
}
```
Gradient-based optimization

Gradient descent:

\[ x_{i+1} = x_i - \alpha \nabla f(x_i) \]

Applications:

- Function minimization
- Backpropagation for machine learning
- Fitting models to data

In ROOT:

- RooFit
- TFo\textit{r}mul\textit{a} and histograms

[Wikipedia, Gradient descent]
Given $f : \mathbb{R}^n \rightarrow \mathbb{R}$, how to find $\nabla f = \left( \frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n} \right)$?

**Differentiation methods:**

- Symbolic differentiation
- Numerical differentiation (ROOT)
  
  $$f'(x) \approx \frac{f(x+h) - f(x)}{h} \quad f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} \quad \ldots$$

- Automatic differentiation
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;
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**

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

```c
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**
- More efficient than numerical differentiation
- Small constant factor more arithmetic operations than the original function

\[ f'(x) \approx \frac{f(x+h) - f(x)}{h} \]
Automotive differentiation in Clad

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

\[
f : \mathbb{R}^n \rightarrow \mathbb{R}
\]

\[
double f(double a, double b[], double* c, double** d);
\]

- Can be extended with:
  - custom data structures
  - multiple outputs

\[
f : \mathbb{R}^n \rightarrow \mathbb{R}^m
\]

\[
double f(vector<double> x);
\]

\[
vector<double> f(vector<double> x);
\]
Automatic differentiation in Clad

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

- Supports both **forward** and **reverse** mode AD:
  - `clad::differentiate` uses forward mode
  - `clad::gradient` uses reverse mode
Forward mode 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

$\begin{align*}
  f(x_1, x_2) &= \sin(x_1) + x_1 x_2 \\
  \text{[Wikipedia, Automatic differentiation]}
\end{align*}$
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]
To compute a gradient w.r.t. $N$ inputs:

**Numerical differentiation:**
- has to evaluate $f$
  - $N+1$ times $\frac{\partial f(x)}{\partial x_i} \approx \frac{f(x+h_i) - f(x)}{h}$
  - $2N$ times $\frac{\partial f(x)}{\partial x_i} \approx \frac{f(x+h_i) - f(x-h_i)}{2h}$

**Reverse mode AD:**
- will transform $f$ to $f\_grad$
- increasing number of arithmetic operations by a small constant factor (up to $\sim 4$)
- single evaluation of $f\_grad$ gives the gradient
- $O(N)$ times faster than numerical differentiation

**Disadvantage:**
- Increased memory usage to reverse program state
- Dynamic allocations on loops
How Clad works

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

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

```cpp
double f_darg0(double x) {
    return 1*x + x*1;
}
```
How to use Clad

In ROOT via Cling interpreter

- **TFormula::GradientPar**

```cpp
TFormula* f1 = new TFormula("f1", "gaus");
...
f1->GradientPar(x, result);
```

- Uses **clad::gradient** internally
How to use Clad

As a standalone clang plugin for C++

- Attach `libclad.so/libclad.dylib` to clang to compile with enabled 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
```
Example of usage from C++

- You want to differentiate some existing C++ function:
  ```cpp
double f(double x, ...) {...}
  ```

- Use:
  ```cpp
clad::differentiate(f, ARG);
  ```
  or
  ```cpp
clad::gradient(f, ARG);
  ```

- Clad will detect that call and process `f` in compile time to create the derivative

- No source modification needed

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

- #include "clad/Differentiator/Differentiator.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, 0);
      ```
    - 2) string literal with the name of independent variable (as written in the definition)
      ```cpp
      clad::differentiate(f, "x");
      ```
  - Will generate a function `f_dargl`, with the same signature as `f`
    - `f_dargl` 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 = df.execute(2.0);
    // val is 4.0
}
Using \texttt{clad::gradient}

- \texttt{auto \_gf = clad::gradient(\_f, ARG);} \\
  \begin{itemize}
  \item \texttt{f} is a pointer to your function \\
  \item \texttt{ARG} is \textit{optional}:
    \begin{itemize}
    \item string literal with comma-separated names of independent variables \texttt{clad::gradient(f, \texttt{"x, y, z"});}
    \item if not provided, \texttt{f} will be differentiated w.r.t. to all parameters \texttt{clad::gradient(f);} 
    \end{itemize}
  \end{itemize}
- \textbf{Will generate a function} \texttt{f\_grad}, with the the following signature \texttt{void f\_grad(/* ..., same inputs as in f*/, \texttt{double* _result});} \\
  \begin{itemize}
  \item \texttt{_result} is an output parameter to get gradient vector
  \end{itemize}
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] = {};  
  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

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

- Define a custom derivative yourself:

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

- Will be detected by Clad:

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

- Derivatives for some math functions (from `<cmath>`) defined in "clad/Differentiator/BuiltinDerivatives.h"
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;
}
Support of C++ constructs:
- Tested with built-in floating point types: float, double
- Arithmetic operators, function calls, variable declarations, if statements ...
- variable mutation (reassignments), for loops
- C arrays as inputs and intermediate variables

TODO:
- Custom data structures (e.g. std::vector)
- Improved differentiation of struct/class methods (differentiate w.r.t. members)
- Occasional missing C++ constructs (e.g. pointers, references, lambdas…)
- Memory optimization of reverse mode
Future work

- Integration into RooFit
- Better API
Benchmarks: in ROOT

TF1* h1 = new TF1("f1", "formula");
TFormula* f1 = h1->GetFormula();
f1->GenerateGradientPar(); // clad

Clad:
f1->GradientPar(x, result);

Numerical:
h1->GradientPar(x, result);

- gaus: Npar = 3
- expo: Npar = 2
- crystalball: Npar = 5
- breitwigner: Npar = 5
- cheb2: Npar = 4

~10x faster!

In RootBench: https://github.com/root-project/rootbench/pull/86

Numerical:

- gaus: 963 ns
- expo: 223 ns
- crystalball: 80 ns
- breitwigner: 339 ns
- cheb2: 291 ns
Why is the speedup factor higher than theoretical limit of ~Npar?

From TF1::GradientPar():

```cpp
// save original parameters
Double_t par0 = parameters[ipar];
parameters[ipar] = par0 + h;
f1 = func->EvalPar(x, parameters);
parameters[ipar] = par0 - h;
f2 = func->EvalPar(x, parameters);
parameters[ipar] = par0 + h / 2;
g1 = func->EvalPar(x, parameters);
parameters[ipar] = par0 - h / 2;
g2 = func->EvalPar(x, parameters);

// compute the central differences
h2 = 1 / (2. * h);
d0 = f1 - f2;
d2 = 2 * (g1 - g2);
T grad = h2 * (4 * d2 - d0) / 3.;

// restore original value
parameters[ipar] = par0;
return grad;
```

- some initial bookkeeping
- 4 calls to f
- additional ops to improve accuracy
Benchmarks: C++, out of ROOT

Tested function:

```c
double sum(double* p, int dim) {
    double r = 0.0;
    for (int i = 0; i < dim; i++)
        r += p[i];
    return r;
}
```

Numerical:

```c
double* Numerical(double* p, int dim, double eps = 1e-8) {
    double result = new double[dim]{};
    for (int i = 0; i < dim; i++) {
        double pi = p[i];
        p[i] = pi + eps;
        double v1 = sum(p, dim);
        p[i] = pi - eps;
        double v2 = sum(p, dim);
        result[i] = (v1 - v2)/(2 * eps); // \frac{\partial f(x)}{\partial x_i} \approx \frac{f(x+h_i)-f(x-h_i)}{2h}
        p[i] = pi;
    }
    return result;
}
```

Clad:

```c
double* Clad(double* p, int dim) {
    auto result = new double[dim]{};
    auto sum_grad = clad::gradient(sum, "p");
    sum_grad.execute(p, dim, result);
    return result;
}
```

Tested function:  
Clad:
Benchmarks: C++, out of ROOT

Original function:

```c
double sum(double* p, int dim) {
    double r = 0.0;
    for (int i = 0; i < dim; i++)
        r += p[i];
    return r;
}
```

Clad’s gradient:

```c
void sum_grad_0(double *p, int dim, double *_result) {
    double _d_r = 0;
    unsigned long _t0;
    int _d_i = 0;
    clad::tape<int> _t1 = {};
    double r = 0.;
    _t0 = 0;
    for (int i = 0; i < dim; i++) {
        _t0++;
        r += p[clad::push(_t1, i)];
    }
    double sum_return = r;
    _d_r += 1;
    for (; _t0; _t0--) {
        double _r_d0 = _d_r;
        _d_r += _r_d0;
        _r_d0 += _d_r;
        _result[clad::pop(_t1)] += _r_d0;
        _d_r -= _r_d0;
    }
}
```
Benchmarks: C++, out of ROOT

![Graph showing benchmark results for Clad and Numerical methods over dimensions (dim) and time (ns). The graph illustrates a clear increase in time as the dimension increases, with Clad showing a more linear increase and Numerical method showing a steeper rise, especially at higher dimensions.](image-url)
Benchmarks: C++, out of ROOT

~dim/4 times faster!
Original function:

```cpp
double gaus(double* x, double* p /*means*/, double sigma, int dim) {
    double t = 0;
    for (int i = 0; i < dim; i++)
        t += (x[i] - p[i])*(x[i] - p[i]);
    t = -t / (2*sigma*sigma);
    return std::pow(2*M_PI, -n/2.0) * std::pow(sigma, -0.5) * std::exp(t);
}
```

\[
\frac{1}{\sqrt{(2\pi)^{\text{dim}} \sigma}} e^{-\frac{|x-p|^2}{2\sigma^2}}
\]
Benchmarks: C++, out of ROOT

~$\text{dim/25 times faster}$
• Clad: https://github.com/vgvassilev/clad