Adding Numerical Differentiation Support To Clad

Garima Singh • Google Summer of Code 2021
Mentors: Vassil Vassilev and Alexander Penev

Introduction

What is Clad?

Clad is an automatic differentiation library implemented as a Clang plugin.

What is Automatic Differentiation (AD)?

Very generally speaking, AD is a set of techniques to evaluate the derivative of a computer program. It computes the exact derivative of a program (if any exists) unlike numerical differentiation which aims to compute an estimate instead.

Why Numerical Differentiation for an AD library?

Due to some constraints it might be inefficient or even impossible to use AD for a function, this is where numerical differentiation comes in. Numerical differentiation will be used as a fallback mechanism in case clad is not able to differentiate the given function.

Basic Idea

The formula to implement:

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h}$$

This is called the 'Finite Central Difference' method. While we do not use this formula directly in clad, it makes things easier to understand. As one can see, this formula will likely take care of single argument function, but what about multi argument functions?

A general formula:

$$f'(x_0, x_1, ..., x_n)_{x_i} = \frac{f(..., x_i + h, ...) - f(..., x_i - h, ...)}{2h}$$

Basic Idea

For multi-argument functions, have a ‘magic’ function to pick and update the correct parameter (in this case the i^{th} parameter), modify that and forward the rest unchanged. This allows us to get those two calls we saw before and use those to calculate the estimated derivative.

Since these derivatives are mere estimates, we also want to have a functionality of printing associated errors. This also open up an avenue of forward propagating these errors to see what kind of results we get.

Lastly, we want to be able to switch off numerical differentiation completely. There might be cases where it is needed to “error-out” instead of building call to numerical differentiation.

Basic Implementation

Implement the ‘magic’ function using templates, parameter packs and index_sequences to pick the correct ith parameter.

This allows us to be very concise and flexible in our implementation (as will be described later). We avoid rewriting a lot of code and being more general as a whole.

An algorithmic overview of the implemented function is as follows:

```
for each i in args, do:

fx1 :=
f(updateIndexParamValue(args, i,
sequence)...)

fx2 :=
f(updateIndexParamValue(args, i,
sequence)...)

grad[i][0] := (fx1 - fx2)/(2 * h)

end for
```

Basic Implementation contd

There is one thing of interest in the algorithm we saw before - ‘UpdateIndexParamValue’. An overview of that function is given below:

```
// This function basically enables us to 'select' the correct parameter to update.
// Without this function, we will not be able to figure out which x should be updated to x ± h.
template <typename T>
T updateIndexParamValue(T arg, std::size_t idx, std::size_t currIdx, int multiplier, precision& h_val,...) {
    if (idx == currIdx) {
          // selects the correct ith term.
          // assigns it an h_val (h)
          // and returns arg + multiplier * h_val essentially.
    }
    return arg;
  }
```

Here, Idx is the current parameter we are on and currIdx is the parameter we want to differentiate with respect to in this pass. If the indices do not match, we return the argument unchanged.

Examples: with clad

For the clad use-case, we synthesize a call to either the “forward” numerical differentiation function (for differentiation with respect to single arguments) or the “reverse” numerical differentiation function (for differentiation with respect to all the arguments.).

Some examples are given in the following slides.

Examples: with clad contd

Reverse Mode (`clad::gradient`):

```
double test_1(double x){
   return tanh(x);
}
```

```
void test_1_grad(double x, clad::array_ref<double> _d_x) {
    double _t0;
    _t0 = x;
    double test_1_return = tanh(_t0);
    goto _label0;
  _label0:
    {
        double _r0 = 1 * numerical_diff::forward_central_difference(tanh, _t0, 0, 0, _t0);
        * _d_x += _r0;
    }
}
```

Examples: with clad contd

Forward Mode (`clad::differentiate`):

```
double test_2(double x){
  return std::log10(x);
}
```

```
double test_2_darg0(double x) {
    double _d_x = 1;
    return numerical_diff::forward_central_difference(std::log10, x, 0, 0, x) * _d_x;
}
```

Examples: standalone

Standalone, the numerical differentiation is capable of a lot. We have the many ways listed below:

- For in-built scalar and non-scalar types. These includes doubles, floats, double*, etc.
- Support for user defined types as input (both value and pointer forms). This can be achieved by overloading the `UpdateIndexParamValue` with the special type.
- Can differentiate overloaded operators.
- Can differentiate functors.

Examples: standalone

```
struct myStruct {
 double data;
 bool effect;
 myStruct(double x, bool eff) {
   data = x;
   effect = eff;
 }
};

myStruct operator+(myStruct a, myStruct b) {
 myStruct out(0, false);
 out.data = a.data + b.data;
 out.effect = a.effect || b.effect;
 return out;
}

double func3(myStruct a, myStruct b) {
return (a + b + a).data; }
```

```
myStruct
updateIndexParamValue(myStruct arg, std::size_t idx, std::size_t
currIdx, int multiplier, numerical_diff::precision& h_val,
std::size_t n = 0, std::size_t i = 0) {
 if (idx == currIdx) {
   h_val = (h_val == 0) ? numerical_diff::get_h(arg.data) : h_val;
   if (arg.effect)
     return myStruct(arg.data + h_val * multiplier, arg.effect);
 }
 return arg;
}
```

Current Implementation

We achieved some really interesting results that are enumerated below:

- Differentiating calls with pointer/array input
- Differentiating user-defined types
- Lightweight dedicated interface
- Printing of error estimates

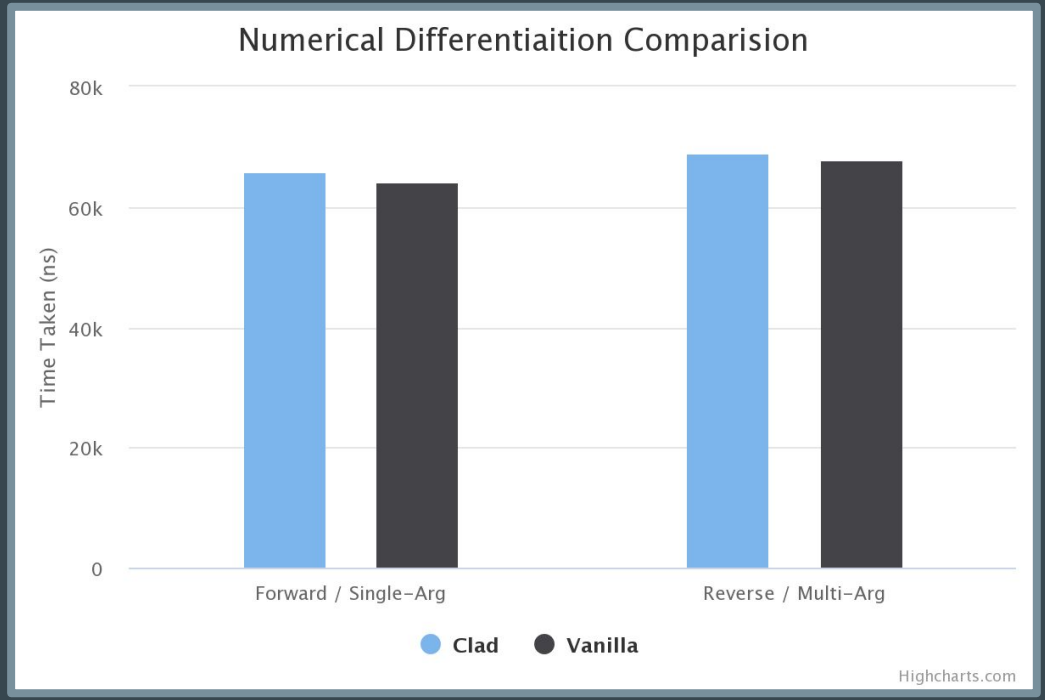
Future work:

The error estimates may also be propagated further through the function to see what kind of effects they have on the final output. They may also be coupled with clad's floating-point error estimation framework to get interesting insights into the stability of functions.

Performance Analysis

A small comparison between the vanilla and clad version of numerical differentiation for both single and multiple argument calls.

As is observed, the performance drop is very small and considering that we do not take the dev effort into account while adapting the vanilla method for multi-argument function calls (something which clad does automatically), clad's numerical differentiation becomes a nice general solution for numerical differentiation.



Key Takeaways

- Using automatic differentiation coupled with numerical differentiation as a failsafe for AD libraries.
- Using a template based numerical differentiation for custom user types allows differentiation of a varied variety of functions.
- Error diagnostics and their propagation for numerical differentiation may also be carried out to gain interesting insights into functions.

Thank you!

You can find more about my work in this gist.

Or, for any comments or suggestions, you can contact me here.