Add support for differentiating functor objects in clad

Parth Arora

Mentors: Vassil Vassilev and David Lange

Project Overview

Many computations are modelled using functors and lambda expressions. These constructs are becoming more and more popular in modern C++. Thus there's a growing need for support to differentiate these constructs in clad. The primary goal of the project focuses on this issue and therefore adds support for differentiating functors and lambda expressions in clad.

The secondary goal of the project is to make general improvements to clad, focusing majorly on making clad more robust and adding support for differentiating more C++ syntax.

Clad Overview

Clad is an automatic differentiation clang plugin for C++. It can differentiate mathematical functions represented as C++ functions.

To differentiate a function, user have to specify the function to be differentiated, independent parameters and the differentiation mode. For each function to be differentiated, clad creates another function that computes the requested derivative.

Clad uses automatic differentiation coupled with analysing and transforming abstract syntax tree (AST) to produce derived functions. Automatic differentiation avoids all the usual disadvantages of symbolic and numerical differentiation such as precision loss and high computation time.

Clad provides an interface to execute the differentiated function and obtain its source code. A simple example to demonstrate clad usage:

```
double fn(double i, double j) {
  return i*j;
}
// Create a function that computes derivative of 'fn' w.r.t 'i'
auto d_fn = clad::differentiate(fn, "i");
// Computes derivative of 'fn' w.r.t 'i' when (i, j) = (3, 4)
double res = d_fn.execute(3, 4);  // res is equal to 4
// 'getCode' returns string representation of the generated derived function.
const char* derivative_code = d_fn.getCode();
```

Please read more about clad here.

Benefits of functor objects over ordinary functions

Functor object, also known as function object, is an object that can act as a function. In C++, an object of a class type that defines the call operator (*operator()*) member function is a functor object.

Functor objects offer several advantages over ordinary functions:

- Functor objects are stateful. They can have states in the form of member variables. States can be used to store values which do not change very often and can thus prevent call operator parameter list from being cluttered.
- Functors can be used to create configurable algorithms. Configurations can be stored as member variables and class template parameter types.
- Calls to functors are often inlined by the compilers as opposed to the calls to function pointers. Thus, using functors lead to better performance.

In light of these advantages, functors are being preferred more and more over ordinary functions for complicated tasks.

Add support for differentiating functor objects

What is meant by differentiating functors?

Differentiating functors means differentiating the call operator (*operator()*) member function defined by the functor type and executing the differentiated function using a reference to the functor object. Thus, the differentiated call operator have access to the member variables and member functions of the functor object.

Despite the differences, differentiating functors is remarkably similar to differentiating functions in clad.

```
class ElectrostaticForce {
  double q1, q2;
  const double k = 8.99e9;
public:
  ElectrostaticForce(double p_q1, double p_q2) : q1(p_q1), q2(p_q2) {}
  double operator()(double radius) {
      return k * q1 * q2 / (radius * radius);
  }
};
ElectrostaticForce E(3.00, 5.00);
auto d_E = clad::differentiate(&E, "radius");
std::cout<<d_E.execute(4)<<"\n";
```

Differentiating functors example

Results:

- All clad differentiation functions now support differentiating functor objects.
- As a prerequisite for the implementation of differentiating functors, differentiation of member functions using clad reverse mode differentiation functions have been improved and support for differentiation of member function is added to `clad::hessian`.
- Just like functions, functors can also be passed both by pointers and by reference to the clad differentiation functions.
- Functors of template types and template specialization types are also supported.
- Differentiating functors using the forward mode AD (`clad::differentiate`), also supports differentiating with respect to member variables.

Add support for differentiating lambda expressions

Lambda expressions are C++ way of creating a closure, that is, an anonymous function object capable of capturing variables in scope.

```
auto momentum = [](double mass, double velocity) {
  return mass * velocity;
};
auto d_momentum = clad::differentiate(momentum, "velocity");
// Computes derivative w.r.t to 'velocity' when (mass, velocity) = (5, 7)
std::cout << d_momentum.execute(5, 7) << "\n";
```

Differentiating lambda expressions example

Results:

- All clad differentiation functions now support differentiating lambda expressions.
- Just like functions and functors, lambda expressions can also be passed both by pointers and by reference to the clad differentiation functions.

Add support for differentiating more C++ syntax

Long term goal of clad is to support all C++ syntax. Support for more C++ syntax is added to bring clad one step closer to this goal.

Results:

- `while` and `do-while` statements are now supported in both the forward and the reverse mode automatic differentiation in clad.
- `switch`, `continue`, and `break` statements are now supported in the forward mode automatic differentiation in clad.

Automatic verification of the reverse mode AD

An assert-based testing framework, which can be optionally enabled using a compile-time flag, is added to verify derivatives produced by the reverse mode AD using the forward mode AD.

The main goal of this is to make clad more robust by making it easier to find any inconsistencies and errors in the implementation of the forward and the reverse mode automatic differentiation.

The design of the automatic verification is very simple yet effective. It involves modifying the reverse mode gradient function to cross-verify the derivatives produced by the gradient function using the corresponding forward mode derivative functions.

Automatic verification modifies the gradient function as follows:

```
double fn_grad(double i, double j) {
  // make copy of all the arguments.
  // These will be used to call the forward mode differentiated functions.
  double _p_i = i;
  double _p_j = j;
  ...
  // usual code to calculate the function gradient
  ...
  // cross-verify derivative w.r.t 'i'.
  clad::VerifyResult(*_d_i, fn_darg0(_p_i, _p_j));
  // cross-verify derivative w.r.t 'j'.
  clad::VerifyResult(*_d_j, fn_darg1(_p_i, _p_j));
}
```

If verification fails, then the function prints an *Assertion failed* message and aborts the program.

Future work

- Add support for differentiating calls to member functions.
- Add support for differentiating functors with multiple overloaded call operators.
- Add automatic verification of the forward mode AD using the reverse mode AD.
- Add automatic verification of the automatic differentiation using numerical differentiation.

Acknowledgements

I would like to thank my mentors, Vassil Vassilev and David Lange, for their constant guidance, suggestions, code reviews and help. The goals accomplished in this project would not have been possible without their help.