Add Support for differentiating functor objects in Clad

Parth Arora
Mentors: Vassil Vassilev, David Lange

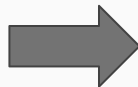
What is Clad?

- Clad is an automatic differentiation clang plugin for C++.

- For each function that is to be differentiated, Clad creates another function that computes its derivative.

Clad in Action

```
double fn(double i, double j) {
  return i*i*j;
}
```



Differentiating w.r.t 'i'

```
double fn_darg0(double i, double j) {
    double _d_i = 1;
    double _d_j = 0;
    double _t0 = i * i;
    return (_d_i * i + i * _d_i) * j + _t0 * _d_j;
}
```

```
auto d_fn = clad::differentiate(fn, "i");
// compute derivative of 'fn' w.r.t 'i' when (i, j) = (3, 5)
std::cout<<d_fn.execute(3, 5)<<"\n";
```

Basic usage

Project Goals

- Add support for differentiating functor objects and lambda expressions in both the forward and the reverse mode automatic differentiation.

- General improvements to clad.
    - Add support for differentiating more C++ syntax and constructs.
    - Add an automatic assert-based testing framework.

Add support for differentiating functors and lambda expressions

What is a functor?

- A functor object is an object that can act as a function.

```
// A class that defines the call operator
class Functor {
  public:
    double operator()(double i) {}
  }
}

Functor functor;
double a = functor(3);
```

Why use functor instead of a function?

- Functor objects are stateful.

```
class Pressure {
  double m_initial_pressure, m_density, m_g;

public:
  Pressure(double density, double g = 9.8, double initial_pressure = 0)
      : m_density(density), m_g(g), m_initial_pressure(initial_pressure) {}
  double operator()(double h) {
    return m_initial_pressure + m_density * m_g * h;
  }
};

Pressure P(/*density=*/1, /*g=*/9.9);

// Pressure at height h = 10m
std::cout<<P(10)<<"\n";
// Pressure at height h = 15m
std::cout<<P(15)<<"\n";
```

Why use functor instead of a function?

- Functors can be used to create configurable algorithms.
- Functors fit well into the OOP paradigm.
- Calls to functor objects are often inlined by the compilers. This lead to better performance.

Add support for differentiating functor objects

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.

Add support for differentiating functor objects

```
class ElectrostaticForce {
  double q1, q2;
  const double k = 8.99e9; // electrostatic constant
  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");
auto d_ERef = clad::differentiate(E, "radius");
// Computes derivative w.r.t to 'radius' when (q1, q2, radius) = (3.00, 5.00 , 5.00)
std::cout<<d_E.execute(5.00)<<"\n";
```

Add support for differentiating lambda expressions

```
auto momentum = [](double mass, double velocity) {
  return mass * velocity;
};

// both ways are equivalent
auto d_momentum = clad::differentiate(&momentum, "velocity");
auto d_momentumRef = clad::differentiate(momentum, "velocity");
// Computes derivatives w.r.t to 'velocity' when (mass, velocity) = (5, 7);
std::cout << d_momentum.execute(5, 7) << "\n";

auto d_momentumGrad = clad::gradient(&momentum);
double d_mass=0, d_velocity=0;
// computes derivatives w.r.t to 'mass' and 'velocity' when (mass, velocity) = (5, 7)
d_momentumGrad.execute(5, 7, &d_mass, &d_velocity);
std::cout<<d_mass<<" "<<d_velocity<<"\n";
```

Differentiating with respect to member variables.

```
class Momentum {
  double mass, velocity;

public:
  Momentum(double p_mass, double p_velocity)
      : mass(p_mass), velocity(p_velocity) {}

  double operator()() { return mass * velocity; }
};

Momentum momentum(3, 5);
auto d_momentum_velocity = clad::differentiate(momentum, "velocity");
auto d_momentum_mass = clad::differentiate(momentum, "mass");

// computes derivative w.r.t 'velocity' when (mass, velocity) = (3, 5)
std::cout << d_momentum_velocity.execute() << "\n";
// computes derivative w.r.t 'mass' when (mass, velocity) = (3, 5)
std::cout << d_momentum_mass.execute() << "\n";
```

General improvements to clad

Extend clad by adding support for differentiating more C++ syntax and constructs.

- Added support for differentiating *while* and *do-while* statements in the forward and reverse mode automatic differentiation.

- Added support for differentiating *switch* statement in the forward mode automatic differentiation

Automatic testing of the reverse mode AD using the forward mode AD

- Verifies at runtime if the produced derivative result is consistent using both forward and reverse mode AD. If the verification fails, then aborts the program with 'Assertion Failed' message.

- It helped to discover few inconsistencies and errors in the implementation of the forward and the reverse mode AD.

Automatic testing of the reverse mode AD using the forward mode AD

```
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
  ...

  // Verify derivative w.r.t 'i' is equal using both reverse and forward mode.
  clad::
      VerifyResult(*_d_i, fn_darg0(_p_i, _p_j), /*assertMessage=*/
                   "Inconsistent differentiation result with respect to the "
                   "parameter 'i' in forward and reverse differentiation mode",
                   "FileName.cpp", "fn_grad");
  // Verify derivative w.r.t 'j' is equal using both reverse and forward mode.
  clad::
      VerifyResult(*_d_j, fn_darg1(_p_i, _p_j), /*assertMessage=*/
                   "Inconsistent differentiation result with respect to the "
                   "parameter 'j' in forward and reverse differentiation mode",
                   "FileName.cpp", "fn_grad");
}
```

Future work

- Add support for differentiating calls to member functions.
- Add support for differentiating overloaded call operators.
- Add support for differentiating functors and lambda expressions used in a functor or a function.

Future work

- Add support for differentiating break, continue and switch statements in the reverse mode automatic differentiation.
- Add assert-based automatic testing of the forward mode AD using the reverse mode AD.
- Add numerical differentiation as another way to verify results in the assert-based automatic testing.

Thank you

Acknowledgement: I would like to thank my mentors, Vassil Vassilev and David Lange, for the constant guidance and help.