Differentiation of Eigen and Softsusy library using Clad.

•••

Mentors: Dr. Vassil Vassilev, Dr. David Lange
Student: Parth Arora

Contents

- Project Goals
- Current Progress
- Next Goals and challenges

Project Goals

- Differentiation of as much missing C++ language features as required.
- Differentiation of class type objects.
- Differentiate Eigen library features.
- Differentiate Softsusy Library features
- **Project Vision:** Out of the box differentiation of arbitrary types using Clad

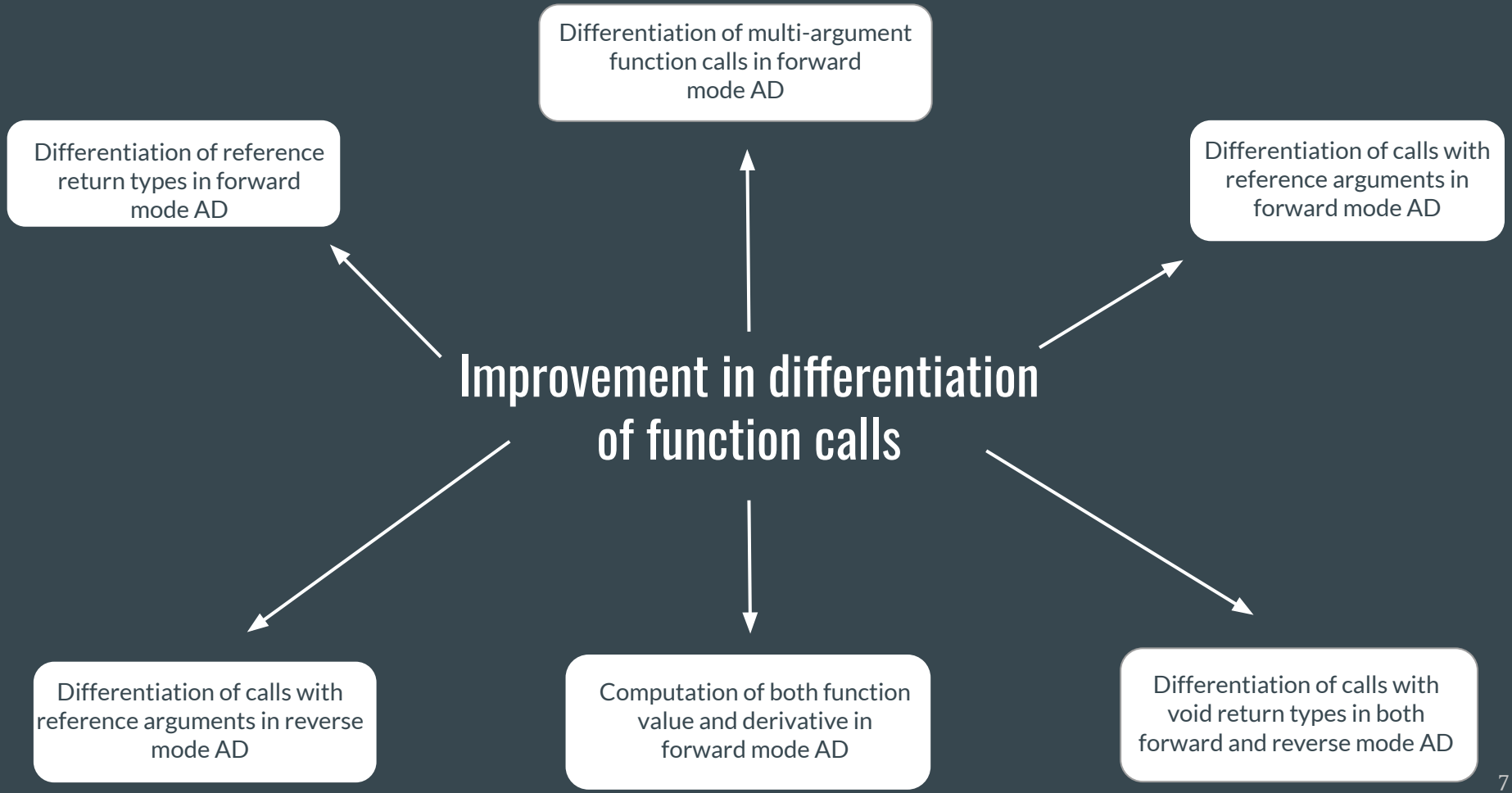
Current Progress

Major Features Added

- Differentiation of ‘break’ and ‘continue’ statements in loops and differentiation of switch statements in reverse mode AD.
- Improved differentiation of function calls in both forward and reverse mode AD.
- Improved custom derivatives functionality.
- Differentiation of pointers in forward mode AD.

Major Features Added

- Differentiation of basic class types in both forward and reverse mode AD.
- Differentiation of calls to member functions in both forward and reverse mode AD.
- Differentiation of calls to overloaded operators in forward mode AD.
- Added testing utilities for efficient and convenient testing of *clad::differentiate* and *clad::gradient*.


Improvement in differentiation of function calls
Differentiation of multi-argument function calls in forward mode AD
Differentiation of reference return types in forward mode AD
Differentiation of calls with reference arguments in forward mode AD
Differentiation of calls with reference arguments in reverse mode AD
Computation of both function value and derivative in forward mode AD
Differentiation of calls with void return types in both forward and reverse mode AD

Improvements in Custom Derivatives Functionality

Custom derivatives are now searched in proper context. For example: custom derivative for *A::B::C::fn* should be defined in *clad::custom_derivatives::A::B::C::*

Now custom derivatives of member functions can also be specified.

Custom derivatives has been modified to utilize pushforward and pullback functions.

Custom Derivative Pushforward Function

Custom derivative of functions in *::std* namespace are defined in namespace *::clad::custom_derivatives::std::*

More optimal custom derivatives: users can design pushforward custom derivative in a way that allows for maximum reusability of computations

```
namespace clad {
namespace custom_derivatives {
namespace std {
template <typename T> ValueAndPushforward<T, T>
sin_pushforward(T x, T d_x) {
  return {::std::sin(x), ::std::cos(x) * d_x};
}

template <typename T1, typename T2>
ValueAndPushforward<decltype(::std::pow(T1(), T2())),
                    decltype(::std::pow(T1(), T2()))>
pow_pushforward(T1 x, T2 exponent, T1 d_x, T2 d_exponent) {
  auto x_e = ::std::pow(x, exponent);
  return {x_e, (exponent * ::std::pow(x, exponent - 1)) * d_x +
               (x_e * ::std::log(x)) * d_exponent};
}
} // namespace std
} // namespace custom_derivatives
} // namespace clad
```

Custom Derivative Pullback Function

d_y – output tangent – propagates derivatives to the input space

```
namespace clad {
namespace custom_derivatives {
namespace std {
template <typename T1, typename T2>
void pow_pullback(
    T1 x, T2 exponent,
    decltype(::std::pow(T1(), T2())) d_y,
    clad::array_ref<decltype(::std::pow(T1(), T2()))> d_x,
    clad::array_ref<decltype(::std::pow(T1(), T2()))> d_exponent) {
  auto t = pow_pushforward(x, exponent, static_cast<T1>(1),
static_cast<T2>(0));
  *d_x += t.pushforward * d_y;
  t = pow_pushforward(x, exponent, static_cast<T1>(0),
static_cast<T2>(1));
  *d_exponent += t.pushforward * d_y;
}
} // namespace std
} // namespace custom_derivatives
} // namespace clad
```

Differentiation of pointers in forward mode

Differentiation of pointers includes support for the following language features:

- Address-of operator (*&*)
- Dereference operator (***)
- *new* operator
- *delete* operator

Differentiable Class Types

- Class should represent a real vector space
- Should have a default constructor that zero initializes the object of the class.
- Copy initialisation should perform deep copy initialisation
- The assignment operator should perform deep copy.

Derivatives of class type objects in forward-mode.

- Forward mode AD evaluates derivatives of each output value w.r.t a single input parameter specified as an independent parameter.
- If an object of class type is the function return value, then forward mode AD evaluates derivatives of each class field w.r.t independent parameter.
- Forward mode AD restricts independent parameter to be a scalar built-in numerical type.

```
std::pair<double, double> fn(double i, double j) {
  std::pair<double, double> c;
  c.first = 7 * i;
  c.second = 9 * i + c.first * j;
  return c;
}

int main() {
  auto d_fn = clad::differentiate(fn, "i");
  auto p = d_fn.execute(3, 5);
  // derivative of 'first' data member of the 'fn'
  // value w.r.t argument 'i'.
  std::cout << p.first << "\n";
  // derivative of 'second' data member of the 'fn'
  // value w.r.t argument 'i'.
  std::cout << p.second << "\n";
}
```

Derivatives of class types in reverse-mode

- Reverse mode AD evaluates derivatives of the function value w.r.t each input parameter.
- If an object of class type is an input parameter, then reverse-mode AD evaluates derivatives of the function value w.r.t each class field.
- Reverse mode AD restricts the function value to be a scalar built-in numerical type.

```
double fn(double i, std::pair<double, double> c) {
  double res = 0;
  res = 7 * i;
  res += c.first * i + c.second * c.first;
  return res;
}

int main() {
  auto fn_grad = clad::gradient(fn);
  double i = 3, d_i = 0;
  std::pair<double, double> p(5, 7), d_p;
  fn_grad.execute(i, p, &d_i, &d_p);
  // derivative of 'fn' value w.r.t 'p.first'.
  std::cout << d_p.first << "\n";
  // derivative of 'fn' value w.r.t 'p.second'.
  std::cout << d_p.second << "\n";
}
```

Differentiation of calls to member functions and operator overloads

```
c.real_pushforward(9 * i, &_d_c, 0 * i + 9 * _d_i);
c.imag_pushforward(11 * i, &_d_c, 0 * i + 11 * _d_i);
auto _t0 = operator_star_pushforward(3., c, 0., _d_c);
auto _t1 =
    res.operator_plus_equal_pushforward(_t0.value,
&_d_res, _t0.pushforward);
```

```
std::complex<double> fn(double i, std::complex<double> c) {
  std::complex<double> res;
  c.real(9*i);
  c.imag(11*i);
  res += 3.0*c;
  return res;
}

int main() {
  auto d_fn = clad::differentiate(fn, "i");
  auto c = d_fn.execute(3, 5);
  // derivative of 'real' part of the 'fn' value
  // w.r.t argument 'i'.
  std::cout << c.real() << "\n";
  // derivative of 'imag' part of the 'fn' value
  // w.r.t argument 'i'.
  std::cout << c.imag() << "\n";
}
```

Differentiation of *std::vector* and *std::map* with the help of custom derivatives

Derivative of *v* will get initialised as:
vectorD _d_v = clad::custom_derivatives::zero_tangent (v);

```
using vectorD = std::vector<double>;
using mapID = std::map<int, double>;

mapID fn(double i, double j, vectorD v) {
  v.clear();
  v.resize(5, i*j);
  mapID m;
  m[0] = v[0];
  m[1] = v[1]*i;
  return m;
}

int main() {
  auto d_fn = clad::differentiate(fn, "i");
  vectorD v(3, 0);
  auto d_m = d_fn.execute(3, 5, v);
  // Derivative of element '0' of the 'fn' value
  // w.r.t argument 'i'.
  std::cout<<d_m[0]<<"\n";
  // Derivative of element '1' of the 'fn' value
  // w.r.t argument 'i'.
  std::cout<<d_m[1]<<"\n";
}
```

Basic differentiation of Eigen Matrix with the help of custom derivatives

Specified derivatives using custom derivatives

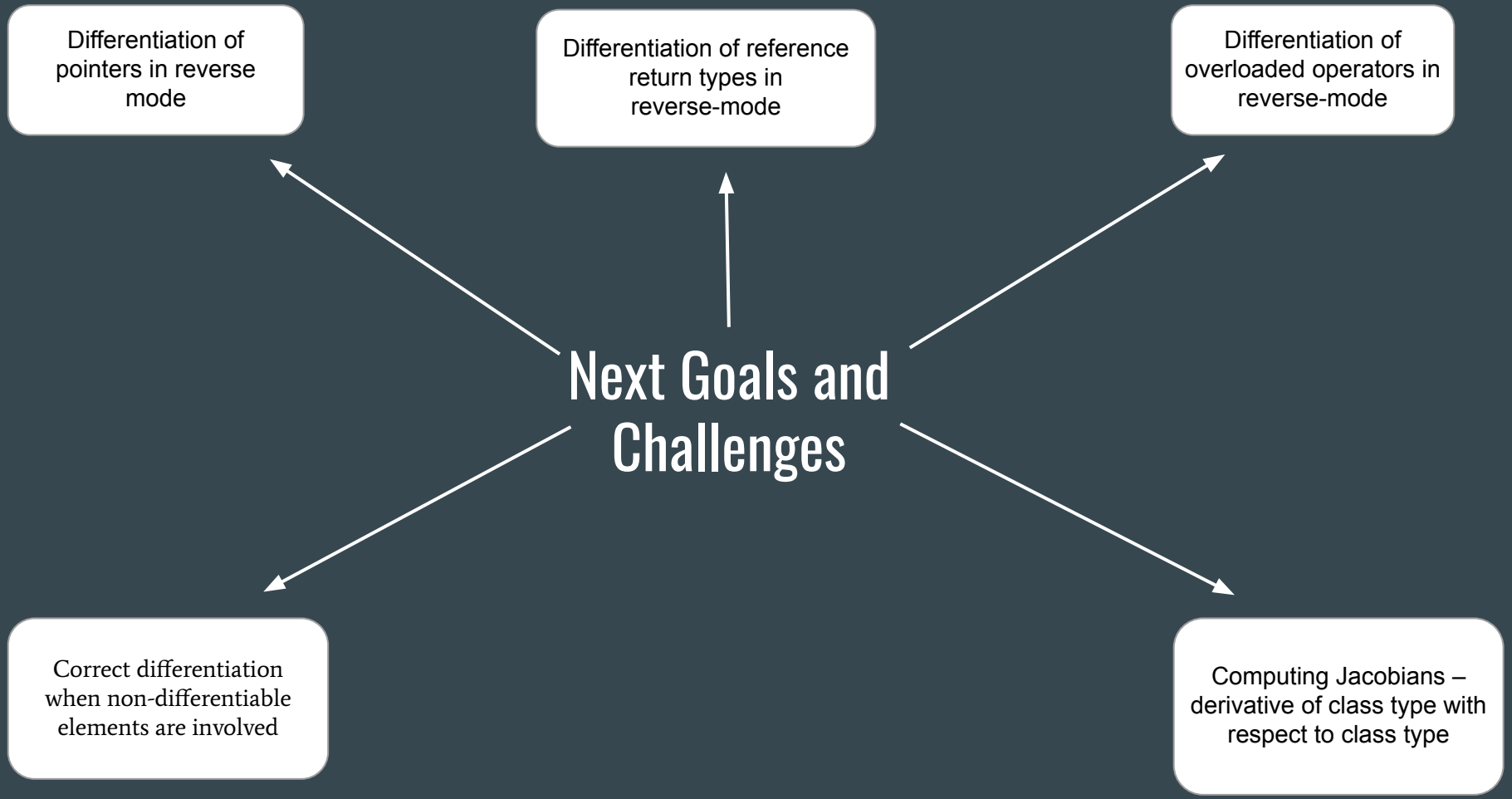
To automatically differentiate most of the Eigen features we need to enhance Clad to correctly differentiate when non-differentiable elements are involved and add support for differentiating constructors.

```
using Eigen::Matrix2d;
using Eigen::Vector2d;

Vector2d fn(double i, Matrix2d m) {
  Vector2d v;
  m(0, 0) = i;
  m(0, 1) = 2 * i;
  m(1, 0) = 3 * i;
  v(0) = m(0, 0);
  v(1) = m.sum();
  return v;
}

int main() {
  auto d_fn = clad::differentiate(fn, "i");
  Matrix2d m;
  auto res = d_fn.execute(3, m);
  // derivative of '0'th element of the 'fn' value
  // w.r.t argument 'i'.
  std::cout << res(0) << "\n";
  // derivative of '1'th element of the 'fn' value
  // w.r.t argument 'i'.
  std::cout << res(1) << "\n";
}
```

Next Goals and Challenges



Thank you!