Recursive compile time adjoint in C++

23rd EuroAD Workshop

Dominic Jones

Siemens PLM, London
dominic.jones@cd-adapco.com

11–13 Aug 2020

Introduction

- The compile time differentiation of a C++ function which only calls built-in functions has been demonstrated in previous talks.
- This talk describes the approach for differentiating a function which may call any nested function.
- Siemen's Simcenter Star-CCM+ simulation software has an implementation of this approach, and is used to differentiate the Spalart Allmaras turbulence model, among other things.

What's new

Flat structure (old)

```
hypotenuse(A const &a,
           B const &b,
           R &r)
{
  auto a2 = a * a;
  auto b2 = b * b;
  auto d = a2 + b2;
  r = sqrt(d);
}
```

Nested structure (new)

```
hypotenuse(A const &a,
           B const &b,
           R &r)
{
  auto d = discrimiant(a, b);
  r = sqrt(d);
}
```

```
discrimiant(A const &a,
            B const &b)
{
  auto a2 = a * a;
  auto b2 = b * b;
  return a2 + b2;
}
```

Built-in functions

Hypotenuse

$$r = \sqrt{a^2 + b^2}$$

```
float a = 3;
float b = 4;
float r;

{
  auto d = a*a + b*b;
  r = sqrt(d);
}

std::cout << r << std::endl;  // r = 5
```

Primal of Hypotenuse

$$r = \sqrt{a^2 + b^2}$$

```
float a = 3;
float b = 4;
float r;

auto constexpr mode = DrvMode::PRIMAL;

Drv<mode, float>  a_{a};  // input
Drv<mode, float>  b_{b};  // input
Drv<mode, float&> r_{r};  // output

{
  auto d = a_*a_ + b_*b_;
  r_ = sqrt(d);
}

std::cout << r << std::endl;  // r = 5
```

Tangents of Hypotenuse

$$\frac{dr}{da}$$

```
float a = 3, a_drv = 1;  // w.r.t. 'a'
float b = 4, b_drv = 0;
float        r_drv;

auto constexpr mode = DrvMode::TANGENT;

Drv<mode, float>  a_{a, a_drv};
Drv<mode, float>  b_{b, b_drv};
Drv<mode, float&> r_{r_drv};

{
  auto d = a_*a_ + b_*b_;
  r_ = sqrt(d);
}

std::cout << r_drv << std::endl;  // dr/da = 0.6
```

Tangents of Hypotenuse

$$\frac{dr}{db}$$

```
float a = 3, a_drv = 0;
float b = 4, b_drv = 1;  // w.r.t. 'b'
float        r_drv;

auto constexpr mode = DrvMode::TANGENT;

Drv<mode, float>  a_{a, a_drv};
Drv<mode, float>  b_{b, b_drv};
Drv<mode, float&> r_{r_drv};

{
  auto d = a_*a_ + b_*b_;
  r_ = sqrt(d);
}

std::cout << r_drv << std::endl;  // dr/db = 0.8
```

Adjoint of Hypotenuse

$$\begin{bmatrix} \frac{dr}{da} & \frac{dr}{db} \end{bmatrix}^T$$

```
float a = 3, a_drv = 0;
float b = 4, b_drv = 0;
float        r_drv = 1;  // w.r.t. ‘r’

auto constexpr mode = DrvMode::ADJOINT;

Drv<mode, float>  a_{a, a_drv};
Drv<mode, float>  b_{b, b_drv};
Drv<mode, float&> r_{r_drv};

{
  auto d = a_*a_ + b_*b_;
  r_ = sqrt(d);
}

std::cout << a_drv << std::endl;  // dr/da = 0.6
std::cout << b_drv << std::endl;  // dr/db = 0.8
```

User defined functions

Defining a function

```
struct Discrimiant
{
  template<DrvMode mode>  // ‘float’ could be templated, too
  static void
  evaluate(Drv<mode, float> const &a,
           Drv<mode, float> const &b,
           Drv<mode, float&> r)
  {
    auto a2 = a * a;
    auto b2 = b * b;
    r = a2 + b2;
  }
};
```

– All function arguments are treated as differentiable terms, even if in the function body some may be treated passively, e.g.

```
auto a2 = primal(a * a);  // i.e.  float a2 = a * a;
```

Calling it via a free function

```
template<typename E0, typename E1>
auto
discrimiant(E0 &&e0, E1 &&e1)
->
DrvVariadicNode<GetDrvMode<E0, E1>::value,   // mode
                decltype(primal(e0 + e1)),   // result
                Bind_evaluate<Discrimiant>,  // operator
                E0, E1>                      // children
{
  return {std::forward<E0>(e0), std::forward<E1>(e1)};
}
```

Unified call syntax

```
Drv<mode, float>  a_{a, a_drv};
Drv<mode, float>  b_{b, b_drv};
Drv<mode, float&> r_{r_drv};

{
  auto d = discrimiant(a_, b_);
  r_ = sqrt(d);
}
```

- `discrimiant` can be used just like any built-in function, such as `sqrt`.
- `a_` or `b_` could be replaced with `a` or `b`, respectively, to treat either passively.

The variadic expression node

Unary, binary, . . . and *variadic*

- Allied with variadic templates, perfect forwarding and `std::tuple`, unary and binary expression nodes can be generalised into an N-ary form.
- In principle, all built-in functions, such as `+ - * /` and `sin cos tan pow`, could return an N-ary node.
- A mechanism to distinguish between built-in and user defined functions is still required. This is done with an operator binding, such as `Bind_evaluate<OP>`.

Base of all nodes

```
template<DrvMode m, typename R, typename E>
struct DrvExpression
{
  static constexpr auto mode = m;
  using Result = R;
  using Expression = E;
};
```

Variadic node*

```
template<DrvMode m,            // mode
         typename R,           // result
         typename OP,          // operator
         typename... EE>       // children
struct DrvVariadicNode
  : DrvExpression<m, R, DrvVariadicNode>
{
  DrvVariadicNode(EE &&... ee) : _ee(ee...) {}

  template<int I> auto const &node() const {
    return std::get<I>(_ee);
  }

  private: std::tuple<EE...> _ee;
};
```

Variadic node

- The *N*-ary node is minimal, providing nothing more than access to its children.
- Children will be associated to the node by value, for in-place expressions or literals, or by reference. This difference is distinguished by perfect forwarding.
- The values and references are held in the `std::tuple` member.

Evaluating the primal*

```
template<DrvMode m,
         typename OP, typename R, typename... EE,
         typename AA, int... I>
auto
evaluatePrimal(
  DrvVariadicNode<m, R, OP, EE...> const &expr,
  AA const &aa,                          // primal values: a, b
  std::index_sequence<I...>)
{
  auto constexpr lm = DrvMode::PRIMAL; // local mode
  auto r_pri = R(0);

  OP::evaluate(
    Drv<lm, decltype(DrvVariadicNode::node<I>())::Result>
      {std::get<I>(aa)}...,              // inputs
    Drv<lm, R&>{r_pri});                 // output

  return r_pri;
}
```

Evaluating the primal

- All work relating to differentiable expressions is done by free functions.
- The expression mode, `m`, is distinguished from the local mode, `lm`, in order to generate the correct instance of the user defined function.
- The arguments for `OP::evaluate` are constructed in-place, generating a new (and independent) differentiation context inside the user defined function.

Evaluating the adjoint*

```
template<DrvMode m,
         typename OP, typename R, typename... EE,
         typename AA, typename RHS, int... I>
void
evaluateAdjoint(
  DrvVariadicNode<m, R, OP, EE...> const &expr,
  AA const &aa,                // primal values: a, b
  AA &aa_drv,                  // adjoint refs:  a_drv, b_drv
  RHS const &rhs,              // seed value:    r_drv
  std::index_sequence<I...>)
{
  OP::evaluate(
    Drv<m, decltype(DrvVariadicNode::node<I>())::Result>
      {std::get<I>(aa), std::get<I>(aa_drv)}...,
    Drv<m, R&>{rhs});
}
```

Evaluating the adjoint

- Here, `OP::evaluate` is compiled in adjoint mode, as deduced from the arguments.

- Despite the fact that `OP::evaluate` will have been evaluated in primal mode already, the values of any intermediate results cannot be captured and used here because there is no monolithic expression tree 'view'.

- The work presented in EuroAD 2019 did have such a view, and so avoided duplicate primal evaluations. But compilation time proved to be too expensive.

Summary

- The variadic template expression node, combined with perfect forwarding, facilitates generic user defined functions.
- User defined virtual functions can be supported, and so too functions with multiple results.
- Despite being difficult to implement, the added functionality of variadic nodes has proved to be a major advancement in deploying automatic differentiation in Star-CCM+.
- The tool is fully `constexpr` qualified, and with aggressive compiler optimisations performs very well: 2.82x on the harmonic test (331 nodes, 5 inputs); cf. EuroAD 2019.

Recursive compile time adjoint in C++

23^{rd} EuroAD Workshop

Dominic Jones

Siemens PLM, London
`dominic.jones@cd-adapco.com`

11–13 Aug 2020