Aggregated type handling in AD tape implementations

M. Sagebaum, N.R. Gauger

AG Scientific Computing
TU Kaiserslautern

24th EuroAD workshop
Overview

- Motivation - What are aggregated types and why do we need to handle them?
- Idea of handling them
- Implementation
- Performance example
Motivation - Aggregated types

Examples:

- Complex numbers
- Matrices
- Vectors, arrays
- Tensors
- Computational grids
- Layers in neural networks
- etc.

Aggregated type

A structured type that can be represented by a set of floating point values.

AD handling: e.g. std::complex<adouble>
Motivation - Advantage of expression templates

Example statement:

\[ w = \sqrt{\text{pow}(u,2) + \text{pow}(v,2)} \]

Example statement with intermediates:

\[
\begin{align*}
t1 &= \text{pow}(u,2); \\
t2 &= \text{pow}(v,2); \\
t3 &= t1 + t2; \\
w &= \sqrt{t3}
\end{align*}
\]

<table>
<thead>
<tr>
<th>Type</th>
<th>Jacobian entries</th>
<th>Statement entries</th>
<th>Total bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>AD double with expression templates</td>
<td>2</td>
<td>1</td>
<td>29</td>
</tr>
<tr>
<td>AD double without expression templates</td>
<td>5</td>
<td>4</td>
<td>80</td>
</tr>
</tbody>
</table>

Memory reduction by 70% (Just for this example.)

Sagebaum, Gauger
Motivation - Advantage of expression templates

Example statement:

\[ w = \sqrt{\text{pow}(u,2) + \text{pow}(v,2)} \]

Example statement with intermediates:

\[
\begin{align*}
t1 &= \text{pow}(u,2); \\
t2 &= \text{pow}(v,2); \\
t3 &= t1 + t2; \\
w &= \sqrt{t3}
\end{align*}
\]

<table>
<thead>
<tr>
<th>Type</th>
<th>Jacobian entries</th>
<th>Statement entries</th>
<th>Total bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>AD double with expression templates</td>
<td>2</td>
<td>1</td>
<td>29</td>
</tr>
<tr>
<td>AD double without expression templates</td>
<td>5</td>
<td>4</td>
<td>80</td>
</tr>
</tbody>
</table>

Memory reduction by 70% (Just for this example.)
**Motivation - Complex numbers and expression templates**

**Example statement:**

\[ w = \sqrt{\text{pow}(u, 2) + \text{pow}(v, 2)} \]

**Example statement with intermediates:**

\[
\begin{align*}
t1 &= \text{pow}(u, 2); \\
t2 &= \text{pow}(v, 2); \\
t3 &= t1 + t2; \\
w &= \sqrt{t3}
\end{align*}
\]

<table>
<thead>
<tr>
<th>Type</th>
<th>Jacobian entries</th>
<th>Statement entries</th>
<th>Total bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>AD double with expression templates</td>
<td>2</td>
<td>1</td>
<td>29</td>
</tr>
<tr>
<td>AD double without expression templates</td>
<td>5</td>
<td>4</td>
<td>80</td>
</tr>
<tr>
<td>AD complex with expression templates</td>
<td>16</td>
<td>8</td>
<td>232</td>
</tr>
</tbody>
</table>

Memory reduction by 50% (Just for this example.)

Sagebaum, Gauger

Aggregated type handling in AD tape implementations
Motivation - Complex numbers and expression templates

Example statement:

\[ w = \sqrt{\text{pow}(u, 2) + \text{pow}(v, 2)} \]

Example statement with intermediates:

\[
\begin{align*}
  t1 &= \text{pow}(u, 2); \\
  t2 &= \text{pow}(v, 2); \\
  t3 &= t1 + t2; \\
  w &= \sqrt{t3}
\end{align*}
\]

<table>
<thead>
<tr>
<th>Type</th>
<th>Jacobian entries</th>
<th>Statement entries</th>
<th>Total bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>AD double with expression templates</td>
<td>2</td>
<td>1</td>
<td>29</td>
</tr>
<tr>
<td>AD double without expression templates</td>
<td>5</td>
<td>4</td>
<td>80</td>
</tr>
<tr>
<td>AD complex with expression templates</td>
<td>16</td>
<td>8</td>
<td>232</td>
</tr>
<tr>
<td>AD complex with potential expression templates</td>
<td>8</td>
<td>2</td>
<td>106</td>
</tr>
</tbody>
</table>
Motivation - Complex numbers and expression templates

Example statement:

```
w = sqrt(pow(u,2) + pow(v,2))
```

Example statement with intermediates:

```
t1 = pow(u,2);
t2 = pow(v,2);
t3 = t1 + t2;
w = sqrt(t3)
```

<table>
<thead>
<tr>
<th>Type</th>
<th>Jacobian entries</th>
<th>Statement entries</th>
<th>Total bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>AD double with expression templates</td>
<td>2</td>
<td>1</td>
<td>29</td>
</tr>
<tr>
<td>AD double without expression templates</td>
<td>5</td>
<td>4</td>
<td>80</td>
</tr>
<tr>
<td>AD complex with expression templates</td>
<td>16</td>
<td>8</td>
<td>232</td>
</tr>
<tr>
<td>AD complex with potential expression templates</td>
<td>8</td>
<td>2</td>
<td>106</td>
</tr>
</tbody>
</table>

Memory reduction by 50% (Just for this example.)
Aggregated types - Expected memory increase

Let the aggregated type have $n$ entries.

**Jacobian taping** (Store $\frac{d\phi}{du}(u)$)

- *Per assignment:* Factor of $n$ for statements. (Each entry creates one statement.)
- *Per argument:* Factor of $n \times n$ for Jacobians. (Each entry creates $n$ entries per argument for each statement.)
Aggregated types - Expected memory increase

Let the aggregated type have $n$ entries.

**Jacobian taping** (Store $\frac{d\phi}{du} (u)$)

- *Per assignment*: Factor of $n$ for statements. (Each entry creates one statement.)
- *Per argument*: Factor of $n \times n$ for Jacobians. (Each entry creates $n$ entries per argument for each statement.)

**Primal value taping** (Store $\phi$)

- *Per assignment*: Factor of 1 for statements. (One statement per assignment.)
- *Per argument*: Factor of $n$ for primal values. (Each entry creates $n$ entries per argument.)
Aggregated types - Expected memory increase

Let the aggregated type have \( n \) entries.

**Jacobian taping** (Store \( \frac{d\phi}{du} (u) \))

- **Per assignment**: Factor of \( n \) for statements. (Each entry creates one statement.)
- **Per argument**: Factor of \( n \times n \) for Jacobians. (Each entry creates \( n \) entries per argument for each statement.)

**Primal value taping** (Store \( \phi \))

- **Per assignment**: Factor of 1 for statements. (One statement per assignment.)
- **Per argument**: Factor of \( n \) for primal values. (Each entry creates \( n \) entries per argument.)

For complex numbers:

<table>
<thead>
<tr>
<th></th>
<th>Per assignment</th>
<th>Per value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacobian taping</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Primal value taping</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

If aggregated types are not added to the expression templates, then memory usage increases further.
Restrictions:

- Use already available tape implementations.
  - No new fancy stuff or template meta programming magic.
- Make it “simple” to add custom aggregated types.
  - To the expression templates.
  - To the specialized handling.
Aggregated types - Handling idea - Statement level handling
(Jacobian taping, Primal value taping)

**Assumption:** Aggregated types are already added to the expression framework.

**Assignment:** $w = expr$

- $w$ is an aggregated type with $n$ entries.
- $expr$ is an aggregated expression that matches $w$. 
Aggregated types - Handling idea - Statement level handling  
(Jacobian taping, Primal value taping)

**Assumption:** Aggregated types are already added to the expression framework.

**Assignment:** \( w = expr \)

- \( w \) is an aggregated type with \( n \) entries.
- \( expr \) is an aggregated expression that matches \( w \).

**Idea:** Reduce to assignments of single entries.

```
// Recording:
for i = 1 ... n
  w[i] = expr[i]
end
```

- \( w \) and \( expr \) are treated as array types.
Aggregated types - Handling idea - Expression level handling
(Primal value taping)

**Assumption:** Aggregated types are already added to the expression framework.

**Assignment:** \( w = expr \)

- \( w \) is an aggregated type with \( n \) entries.
- \( expr \) is an aggregated expression that matches \( w \).
Aggregated types - Handling idea - Expression level handling
(Primal value taping)

**Assumption:** Aggregated types are already added to the expression framework.

**Assignment:** \( w = expr \)

- \( w \) is an aggregated type with \( n \) entries.
- \( expr \) is an aggregated expression that matches \( w \).

**Idea:** Store full expression on the tape, treat as array types in the reverse evaluation.

```c
// Recording:
tape.store(w, expr);

// Reversal:
for i = 1 ... n
  expr[i].reverse(w_b[i]);
end
```

- Requires small changes on the tape layout for the primal value tape.
Aggregated types - Implementation - Expression templates

Expression

AggregatedType

AggregatedTraits

Tape

Sagebaum, Gauger
Aggregated type handling in AD tape implementations

10/18
Aggregated types - Implementation - Expression templates

Type for the integration into the expression template structure:

```cpp
<
    typename HOT
>
    // HOT = HigherOrderType

struct AggregatedType : Expression {

    using Traits = AggregatedTraits<HOT>;
    HOT value_;  

    // Expression Interface
    Traits::ValueType getValue() const {
        return Traits::getValue(value_);
    }

    // Expression Interface
    void calcGradient(Traits::ValueType const &multiplier) const {
        for (int i = 0; i < Traits::nElements(value_); i += 1) {
            Traits::extract(value_, i).calcGradient(Traits::extract(multiplier, i));
        }
    }

    <
        typename Arg
    >
    AggregatedType & operator= (Expression<Arg> const &arg) {
        Traits::getTape().store(*this, arg);
    }
};
```

General implementation that uses the traits implementation for the logic.

AD handling: e.g. `AggregatedType<std::complex<adouble>>`
Aggregated types - Implementation - Expression templates

Type for the integration into the expression template structure:

```cpp
<
<typename HOT>  // HOT = HigherOrderType
struct AggregatedType  : Expression {

  using Traits = AggregatedTraits<HOT>;
  HOT value_;  // Expression Interface

  Traits::ValueType getValue() const { return Traits::getValue(value_); }

  // Expression Interface
  Traits::ValueType getValue() const { return Traits::getValue(value_); }

  // Expression Interface
  void calcGradient(Traits::ValueType const& multiplier) const {
    for(int i = 0; i < Traits::nElements(value_); i += 1) {
      Traits::extract(value_, i).calcGradient(Traits::extract(multiplier, i));
    }
  }

  // Expression Interface

Aggregated type handling in AD tape implementations
```
Aggregated types - Implementation - Expression templates

Type for the integration into the expression template structure:

```cpp
<
typename HOT> // HOT = HigherOrderType
struct AggregatedType : Expression {
    using Traits = AggregatedTraits<HOT>;
    HOT value_;  
    
// Expression Interface
    Traits::ValueType getValue() const {return Traits::getValue(value_);}

// Expression Interface
    void calcGradient(Traits::ValueType const& multiplier) const {
    for(int i = 0; i < Traits::nElements(value_); i += 1) {
        Traits::extract(value_, i).calcGradient(Traits::extract(multiplier, i));
    }
    }

<
typename Arg>
AggregatedType& operator=(Expression<Arg> const& arg) {
    Traits::getTape().store(*this, arg);
}
};
```

- General implementation that uses the traits implementation for the logic.
- **AD handling**: e.g. AggregatedType<std::complex<adouble>>
Aggregated types - Implementation - Expression templates

**General traits definition:**

```cpp
template<typename _Type, typename = void>
struct AggregatedTypeTraits {

  using Type = CODI_DD(_Type, CODI_ANY);

  using ValueType; // Primal value representation
  using IdentifierType; // Identifier representation

  static size_t nElements(Type& type); // Number of elements

  static ValueType getValue(Type& type);
  static void setValue(Type& type, ValueType const& value);

  static IdentifierType getIdentifier(Type& type);
  static void setIdentifier(Type& type, IdentifierType const& identifier);

  static double& extract(ValueType& value, size_t i);
  static int& extract(IdentifierType& value, size_t i);
};
```

- Needs to be specialized for each aggregated type – e.g. std::complex.
Aggregated types - Implementation - Statement level handling
(Jacobian taping, Primal value taping)

Specialize the store method for aggregated types.

```cpp
// In JacobianTape
<typename HOT, typename Rhs>
void store(AggregatedType<HOT>& lhs, Expression<Rhs> const& rhs) {
    using Traits = AggregatedTraits<HOT>;

    Traits::ValueType value = Traits::getValue(lhs.value_);
    Traits::IdentifierType identifier = Traits::getIdentifier(lhs.value_);

    for (int i = 0; i < Traits::nElements(lhs.value_); i += 1) {
        RealReverseWrapper wrapper(Traits::extract(value, i),
                                   Traits::extract(identifier, i));
        wrapper = ExtractExpression<Rhs>(rhs, i);
    }
    Traits::setValue(lhs.value_, value);
    Traits::setIdentifier(lhs.value_, identifier);
}
```

For loop over entries. Value and identifier need to be extracted because of self assignments. e.g. 
```
a = a * a;
```
Aggregated types - Implementation - Statement level handling
(Jacobian taping, Primal value taping)

**Specialize** the store method for aggregated types.

```cpp
// In JacobianTape
<typename HOT, typename Rhs>
void store(AggregatedType<HOT>& lhs, Expression<Rhs> const& rhs) {
  using Traits = AggregatedTraits<HOT>;

  Traits::ValueType value = Traits::getValue(lhs.value_);
  Traits::IdentifierType identifier = Traits::getIdentifier(lhs.value_);

  for(int i = 0; i < Traits::nElements(lhs.value_); i += 1) {
    RealReverseWrapper wrapper(Traits::extract(value, i),
                              Traits::extract(identifier, i));
    wrapper = ExtractExpression<Rhs>(rhs, i);
  }
  Traits::setValue(lhs.value_, value);
  Traits::setIdentifier(lhs.value_, identifier);
}
```

For loop over entries. Value and identifier need to be extracted because of self assignments. e.g. 
a = a * a;
Specialize the store method for aggregated types.

```cpp
// In JacobianTape
<typename HOT, typename Rhs>
void store(AggregatedType<HOT>& lhs, Expression<Rhs> const& rhs) {
    using Traits = AggregatedTraits<HOT>;

    Traits::ValueType value = Traits::getValue(lhs.value_);
    Traits::IdentifierType identifier = Traits::getIdentifier(lhs.value_);

    for(int i = 0; i < Traits::nElements(lhs.value_); i += 1) {
        RealReverseWrapper wrapper(Traits::extract(value, i),
                                   Traits::extract(identifier, i));
        wrapper = ExtractExpression<Rhs>(rhs, i);
    }

    Traits::setValue(lhs.value_, value);
    Traits::setIdentifier(lhs.value_, identifier);
}
```

- For loop over entries.
- Value and identifier need to be extracted because of self assignments.
  - e.g. `a = a * a;`
Aggregated types - Implementation - Expression level handling
(Primal value taping)

**Change** the tape structure:

- **Old:** \( 4 \cdot nArgs + 8 \cdot nPas + 8 \cdot nCon + 8 + 4 + 8 + 1 \)
- **New:** \( 4 \cdot nArgs + 8 \cdot nPas + 8 \cdot nCon + 8 \cdot nOut + 4 \cdot nOut + 8 + 1 + 1 \)
  - Store number of outputs: 1 byte (nOut)
  - Store a primal value for each entry: 8 byte each.
  - Store an identifier for each entry: 4 byte each.

**Implementation:**

\[ \]
Aggregated types - Implementation - Expression level handling
(Primal value taping)

**Change** the tape structure:

- **Old:** \( 4 \cdot nArgs + 8 \cdot nPas + 8 \cdot nCon + 8 + 4 + 8 + 1 \)
- **New:** \( 4 \cdot nArgs + 8 \cdot nPas + 8 \cdot nCon + 8 \cdot nOut + 4 \cdot nOut + 8 + 1 + 1 \)
  - Store number of outputs: 1 byte (nOut)
  - Store a primal value for each entry: 8 byte each.
  - Store an identifier for each entry: 4 byte each.

**Implementation:**

```cpp
// To much to show :(
```
Performance example - Burgers equation

Performance test:

- Forward numerical scheme for the coupled Burgers’s equation

\[
\begin{align*}
    u_t + uu_x + vu_y &= \frac{1}{R}(u_{xx} + u_{yy}) \\
    v_t + uv_x + vv_y &= \frac{1}{R}(v_{xx} + v_{yy})
\end{align*}
\]

- Solved on the unit square.
- Elwetritsch cluster compute node: Two Intel Xeon 6126 CPUs with a total of 24 cores and 384 GB of main memory.
- Solved with real and complex numbers.
Performance example - Burgers equation - Memory

![Bar chart showing memory usage for different tape implementations and types.]  
- Linear Real
- Reuse Real
- Linear Complex
- Reuse Complex
- Linear Complex Statement level handling
- Reuse Complex Statement level handling
- Primal Vector
- Adj. Vector
- Doubles
- Identifiers
- Statements

Jacobian taping
Performance example - Burgers equation - Memory

Jacobian taping

Primal value taping
Performance example - Burgers equation - Timing

![Chart showing timing for different types of Jacobian and Primal handling methods.]

- **Linear Jacobian**
  - Real: 0.76, 3.54
  - Complex: 0.84, 2.72
  - Expression: 0.74, 2.37

- **Reuse Jacobian**
  - Real: 2.1, 7.69
  - Complex: 2.37, 4.28
  - Expression: 1.91, 2.92

- **Linear Primal**
  - Real: 2.1, 0.85
  - Complex: 3.72, 0.76
  - Expression: 4.28, 0.74

- **Reuse Primal**
  - Real: 7.69, 8.58
  - Complex: 4.28, 2.37
  - Expression: 2.92, 1.91

**Recording time**
Performance example - Burgers equation - Timing

Linear Jacobian
Reuse Jacobian
Linear Primal
Reuse Primal

Recording time
Reversal time

Real
Complex
Statement level handling
Expression level handling
Conclusion & Outlook

Conclusion:
- Demonstration of overhead for aggregated types in expression template AD frameworks.
- Demonstration of aggregated type handling in standard AD tapes.
- Good memory and timing results.

Outlook:
- Member functions in expression template AD frameworks.
- Specialized tapes for aggregated type handling.
Conclusion & Outlook

Conclusion:
- Demonstration of overhead for aggregated types in expression template AD frameworks.
- Demonstration of aggregated type handling in standard AD tapes.
- Good memory and timing results.

Outlook:
- Member functions in expression template AD frameworks.
- Specialized tapes for aggregated type handling.

Thank you very much for your attention.