On the architecture agnostic Field Operation And Manipulation: Arithmetic operations

February 25, 2025

1 Introduction

In this proposal, we outline the changes required to perform arithmetic field operations in an architecture-agnostic manner using C++ standard library constructs and style. Throughout the paper, we refer to "arithmetic field operations" as any operations that do not require mesh connectivity information and can be applied to any of the OpenFOAM fields. We note that the approach presented here could be easily extended to matrix types.

Although we draw upon features from the C++20 standard (such as ranges, concepts, etc.), we aim to target C++17 with the proposed approach and reference implementation.

While this proposal is written in the imperative, we are merely presenting the approach we have found useful in our in-house GPU library. We are, of course, open to any discussion or feedback.

1.1 Expression Templates and Their Relation to C++ Standard Ranges

Expression templates are a metaprogramming technique used in C++ to optimize the evaluation of complex expressions involving operator overloading. Rather than computing intermediate results, expression templates build a representation of the expression at compile time, allowing the compiler to generate optimized code. This technique is commonly employed in numerical computing libraries (e.g., Eigen, Blaze) to eliminate temporary objects and improve performance.

Expression templates enable deferred evaluation by constructing an abstract syntax tree (AST) at compile time. Instead of evaluating an expression immediately, the compiler represents it as a hierarchical structure.

$$R = A + B + C + D$$

Rather than creating temporary results for each intermediate sum, the compiler generates a generic, nested expression object:

$$\text{Expression} = \text{Add}(\text{Add}(A, B), \text{Add}(C, D))$$

This expression is only evaluated when the result is needed, typically when constructing a new type that holds the data. This approach eliminates unnecessary temporaries, reduces memory usage, and allows for loop fusion, leading to more efficient computation.

In modern C++ (C++20 and later), the Ranges Library introduces views, which provide lazy, composable transformations over sequences without materializing intermediate results. This is conceptually similar to expression templates, as both defer computation until the final evaluation step. Below is an example of how the previously described addition expression could be implemented using C++20 ranges.

Listing 1: Operator+ for two generic range types. The code can be tested here

```
template<class Lhs, class Rhs>
auto operator+(const Lhs& lhs, const Rhs& rhs)
{
    return
    std::views::zip(lhs, rhs)
    | std::views::transform([](auto pair) {
        return
              std::get<0>(pair)
            + std::get<1>(pair);
    });
}

int main() {
    std::vector<double> A = {1.0, 2.0, 3.0};
    std::vector<double> B = {4.0, 5.0, 6.0};
    std::vector<double> C = {7.0, 8.0, 9.0};
    std::vector<double> D = {10.0, 11.0, 12.0};

    auto expression = A + B + C + D;

    // Convert to a concrete vector (materialize results)
    std::vector<double> result
    (
        expression.begin(), expression.end()
    );
    for (auto e : result){
        std::cout << e << std::endl;
    }



    return 0;
}
}
```

2 OpenFOAM Ranges

To implement the ranges approach in OpenFOAM, there are a few special considerations. First, the constructed expressions must hold *all* the necessary data to construct full objects. In the example presented in 1, the construction of a new vector is straightforward since a vector only stores the begin and end pointers. This is very similar to the OpenFOAM `Field` type, for which enabling lazy evaluation of arithmetic operations is simple. However, in OpenFOAM, there are other field types that have additional properties like name, mesh, dimensions, etc.

2.1 Field Concepts

To construct higher-level abstractions of different expression types, it is helpful to consider the properties that different field types have. We introduce four concepts that describe the OpenFOAM field types. Note that here the word "concept" is used loosely and does not (necessarily) refer to the C++20 concepts. In earlier versions of C++, the "concepts" could be implemented using type traits.

Listing 2: A field concept. Here an additional requirement could be that the iterators are checked to dereference a specific arithmetic element type.

```
FieldConcept
{
    hasMember(begin()) -> Iterator
    hasMember(end()) -> Iterator
}
```

Listing 3: A dimensioned field concept. Note that it is important to have a geomesh() member so the volumetric and surface fields can be separated.

```
DimensionedFieldConcept
{
    hasMember(primitiveField()) -> FieldConcept
    hasMember(geomesh()) -> GeoMesh
    hasMember(dimensions()) -> Dimensions
    hasMember(name()) -> Word
}
```

Listing 4: A field of fields concept.

```
FieldFieldConcept
{
    hasMember(operator[](label)) -> FieldConcept
    hasMember(size()) -> SizeType
}
```

Listing 5: A geometric field concept.

```
GeometricFieldConcept
{
    hasMember(internalField()) -> DimensionedFieldConcept
    hasMember(boundaryField()) -> FieldFieldConcept
}
```

2.2 Range types that satisfy the field concepts

2.2.1 Special iterator types

With the concepts defined, we must now implement Range types which not only satisfy these concepts but also enable the building of the nested expression objects. Let us consider the field expressions first. Assuming that we have the following iterators available:

Listing 6: A zip iterator satisfies the standard requirements for random access iterator and holds a tuple of iterators. The return type of the dereference operator of a zip iterator is a tuple of dereferences of the underlying iterators. A possible implementation of a zip iterator is given here.

```
zip_iterator
{
    IsRandomAccessIterator
    hasMember(operator*()) -> Tuple<Iterator, ...>
}
```

Listing 7: A transform iterator which satisfies the standard requirements for a random access iterator type and holds a random access iterator and a generic function F. The dereference operator of a transform iterator is the return type of the generic function F when given the dereference of the underlying iterator. A possible implementation of a transform iterator is given here.

```
transform_iterator
{
    IsRandomAccessIterator
    hasMember(operator*()) -> F(*Iterator)
}
```

Listing 8: A constant iterator which satisfies the standard requirements for a random access iterator type and holds a single constant value. The dereference operator of a constant iterator returns the constant value. A possible implementation of a constant iterator is given here.

```
constant_iterator
{
    IsRandomAccessIterator
    hasMember(operator*()) -> ValueType
}
```

2.2.2 Field range types

With the help of the three special iterator types, it is now possible to implement lightweight range types that allow for building the expression trees and while satisfying the field concepts. As an example, for the Field operations we can define ranges:

Listing 9: Lightweight range types for building Field expression trees.

```
template<class Iterator>
Range{
    using iterator = Iterator;

    Range(Iterator begin, Iterator end)
        : m_begin(begin), m_end(end) {}

    iterator m_begin, m_end;

};

template<class Func, Iterator>
TransformRange : public Range<transform_iterator<Func, Iterator>>
{
    //...
};

template<class Value>
ConstantRange : public Range<constant_iterator<Value>>
{
    //...
};

template<IteratorTuple>
ZipRange : public Range<zip_iterator<IteratorTuple>>
{
    //...
};
```

With these ranges defined, it is possible to implement the `operator+` for two types that satisfy the FieldConcept. We note that if further constraints on the two input types for addition operator are required, they could be added using `static_assert` producing clear compile-time error messages.

Listing 10: operator+ for two types satisfying a FieldConcept.

```
template<class Lhs, class Rhs, REQUIRE(FieldConcept<Lhs, Rhs>)>
constexpr auto operator+
(
    const Lhs& lhs,
    const Rhs& rhs
)
{
    using value_type1 = decltype(*lhs.begin());
    using value_type2 = decltype(*rhs.begin());
    static_assert
    (
        FurtherConstaints<value_type1, value_type2>,
        "NICE ERROR MESSAGE"
    );
    return make_transform_range
    (
        make_zip_range(lhs, rhs),
        std::plus{}
    );
}
```

2.2.3 Other range types

As mentioned earlier, the other field types in OpenFOAM store additional data, which must be propagated in the expression tree to allow for construction of new fields from the expressions. The DimensionedRange implementation could be as follows:

Listing 11: A lightweight DimensionedRange type.

```
template<class Iterator, class GeoMeshType>
struct DimensionedRange
{

    constexpr const Range<Iterator>& primitiveField()
    {
        return m_primitive;
    }
    //...

private:
    Range<Iterator> m_primitive;
    dimensionSet    m_dimensions;
    GeoMeshType     m_mesh;
    word            m_name;
};
```

And the corresponding addition operator could look as follows

Listing 12: operator+ for two types satisfying a DimensionedFieldConcept.

```
template<class Lhs, class Rhs,
REQUIRE(DimensionedFieldConcept<Lhs, Rhs>)>
constexpr auto operator+
(
    const Lhs& lhs,
    const Rhs& rhs
)
{
    return make_dimensioned_transform_range
    (
        make_dimensioned_zip_range(lhs, rhs),
        std::plus{},
        lhs.dimensions() + rhs.dimensions(),
        lhs.name() +'+'+ rhs.name()
    );
}
```

For the FieldField ranges, because the number of Fields is generally not known compile time, slightly different approach is required. Essentially, there are two options:

1. Make FieldFieldRange store a `std::vector<Range<Iterator>>`
2. Make FieldFieldRange store a reference to underlying field(s) and construct a correct 1D Range type when the `operator[](label)` is called

Option 1 leads to cleaner code while option 2 is (slightly) more efficient since there is no need to do a dynamic memory allocation when a FieldFieldRange is constructed. In the reference implementation, option 2 is chosen.

2.3 A note on the field inheritance model

Currently, in OpenFOAM the field types are implemented such that `GeometricField` inherits `DimensionedField`, `Field` and `List`. This makes it slightly difficult to construct concepts that are unique for `GeometricField` since by definition `GeometricField` satisfies all concepts that its child classes satisfy. Moreover, having this type of inheritance leads to following confusions which are not obvious to new developers:

- What does `GeometricField::begin()` return?
- Is `GeometricField::operator[](label)` dimensioned element, element or a boundary field type?
- If a function takes a `UList&`, it is possible to pass in a GeometricField directly which may lead to unwanted behaviour.

Thus, we propose that the inheritance chain is done using private inheritance or, for even more clarity, switching to different field types having a HAS-A relation with other field types. Any access to internalField/primitiveField should be explicit to avoid confusion.

2.4 Proposed changes

To allow for lazy evaluation of field arithmetic the following modifications should be made:

1. Break the inheritance chain of the different field types
2. Removal of `tmp` overloads for all arithmetic expressions
3. Change the return type of a `<FieldType>::New` to be `autoPtr` or `std::unique_ptr`
4. Make all arithmetic field operations return a Range
5. Add constructors for field types from a corresponding Range
6. Change all function return types to be either concrete field types or Ranges
7. Replace all functions which take a `tmp<FieldType>` to take a concrete field type or a Range

Many of the proposed changes are related to the use of the `tmp` class in OpenFOAM. Currently, tmp is for two purposes: 1) Possible reuse of memory in chained operations. 2) Allowing overriding of a base class function which returns a constant reference to a field with a derived class implementation which

does computation and returns an allocated field. The first purpose is automatically handled with the proposed lazy evaluation. The second purpose could be handled by introducing a more explicit type `RefOrValue<T>` which is easy to implement.

2.5 Evaluation of advantages and disadvantages of the proposed approach

The lazy evaluation approach proposed herein has both advantages and disadvantages. Here we try to list both but we note that these are opinionated.

- + Performance (see the benchmarks)
- + Possiblity of better error messages for illegal expressions
- + Usage of standard library constructs increasing portability to different hardware
- + Possibility of having compile time dimension checking in the future
- + Reduction of preprocessor usage in field operations
- + Limiting the number of "evaluation" function calls makes the code more easy to optimize
- - More work is done compile time which may lead to longer compilation times
- - The user must understand object life time when returning a Range from a function which can be error prone (see note below)
- - This is a large change and it is not possible to foresee all negative effects

Listing 13: Example of a possible misuse

```
volScalarField func1 //We return a field
(
    const volScalarField& f1,
    const volScalarField& f2
)
{
    volScalarField expr = mag(f1) + f2 * f1 * sum(f1);
    return expr + f1; //Ok since we cast to volScalarField
}

auto func2 //We return a range
(
    const volScalarField& f1,
    const volScalarField& f2
)
{
    auto expr = mag(f1) + f2 * f1 * sum(f1);
    return expr + f1; //Ok since expr does not allocate
}

auto func3 //We return a range
(
    const volScalarField& f1,
    const volScalarField& f2
)
{
    volScalarField expr = mag(f1) + f2 * f1 * sum(f1);
    return expr + f1; //Not ok since expr is deallocated
}
```

References