FMath Documentation

Release 0.0.0

Gideon Mueller

Jun 13, 2020

Introduction:

CHAPTER 1

General Concepts

The library revolves around the `Field` template class, which is a convenience wrapper around `std::vector` with added operators and functions. The Eigen library is used for vector operations.

Expression templates ensure that mathematical operations are used efficiently, avoiding temporaries.

The library provides convenience `typedef`s:

- `FMath::scalar` = default is `double`, can be defined differently by `FMATH_SCALAR_TYPE`
- `FMath::ScalarField` = `FMath::Field<FMath::scalar>`
- `FMath::Vector3` = `Eigen::Vector3<FMath::scalar>`
- `FMath::VectorX` = `Eigen::VectorX<FMath::scalar>`
- `FMath::VectorField` = `FMath::Field<FMath::Vector3>`

CHAPTER 2

Using

Requirements:

- C++17
- OpenMP 4.5 (optional)
- (later maybe a CUDA version which supports C++17 features)

Adding this library to your project should be trivial. You can do it manually:

- copy the `FMath` folder into your directory of choice
- copy the `thirdparty/Eigen` folder or provide your own
- make sure the `FMath` and `Eigen` folders are in your include-directories
- optionally define `FMATH_SCALAR_TYPE`
- optionally add OpenMP compiler flags to activate parallelisation

or using CMake:

- copy the entire repository folder into your directory of choice
- TODO...

CHAPTER 3

Basics

3.1 Reductions

```
#include <FMath/Core>

using FMath::Field;
using FMath::scalar;
using FMath::Vector3;

// Single field reduction
int N = 10;
Field<Vector3> vf(N, Vector3{0,0,1});
Vector3 mean = vf.mean();

// N-dimensional dot-product
FMath::ScalarField sf1(N), sf2(N);
sf1 *= sf2;
FMath::scalar dot = sf1.sum();

// More efficient version of the dot product:
dot = (sf1*sf2).sum();
```

3.2 Operators

```
#include <FMath/Core>

FMath::VectorField vf1(N), vf2(N);
FMath::ScalarField sf1(N);

// This will produce an expression object, due to auto
auto vf = vf1 + vf2*vf2;
```

(continues on next page)

(continued from previous page)

```
// This will actually evaluate the expression
FMath::ScalarField sf_result = vf.dot(vf1);
```

3.3 Convenience math functions

```
#include <FMath/Core>

FMath::VectorField vf1(N), vf2(N);

// Element-wise dot product
FMath::ScalarField sf_dot = vf1.dot(vf2);
// Element-wise cross product
FMath::VectorField vf_cross = vf1.cross(vf2);
```

3.4 Other convenience functions

Copying or re-interpreting a `Field` as an `Eigen::VectorX`

```
#include <FMath/Core>

FMath::ScalarField sf(N);
FMath::VectorField vf(N);

// Copy a scalar field to a new N-dimensional vector
FMath::VectorX vec1 = sf.asRef<VectorX>();
// Copy a vector field to a new 3N-dimensional vector
FMath::VectorX vec2 = vf.asRef<VectorX>();

// Interpret a scalar field as a N-dimensional vector without copying
Eigen::Ref<VectorX> vecRef1 = sf.asRef<VectorX>();
// Interpret a vector field as a 3N-dimensional vector without copying
Eigen::Ref<VectorX> vecRef2 = vf.asRef<VectorX>();
```

CHAPTER 4

Subsets

4.1 Indexed subset

Extracting and operating on an indexed subset of a `Field` can be performed by passing a `std::vector<std::size_t>` or anything compatible, such as a `Field<int>` into the access operator.

```
#include <FMath/Core>

using FMath::Field;
using FMath::scalar;

// A Field of size N1 and an index list of size N2<N1
Field<scalar> sf1(N1);
Field<int>    index_list1(N2);

// Set the indices of the Field entries you wish to extract...
// (this can also be used to re-order a Field)

// Extract the indexed set
Field<scalar> sf_subset1 = sf1[index_list1];

// Extract a small set via an initializer list
Field<scalar> sf_subset2 = sf1[{0,3,22}];

// Operate on subsets, combining different index lists
Field<scalar> sf2(N1);
Field<scalar> sf3(N3);
Field<int>    index_list2(N2);
sf1[index_list1] = sf2[index_list1] + sf3[index_list2];
```

4.2 Contiguous slices

`Field.slice()` takes:

- `std::size_t begin = 0`
- `std::optional<std::size_t> end = {}`
- `std::size_t stride = 1`

```
#include <FMath/Core>

using FMath::Field;
using FMath::scalar;

Field<scalar> sf1{1,2,3,4,5};

// Create field from strided slice of other field
Field<scalar> sf2 = sf1.slice(0, {}, 2);
// Expected contents
// sf2.size() == 3
// sf2[0] == 1
// sf2[1] == 3
// sf2[2] == 5

// Assign slice from slice
sf2.slice(0,1) = sf1.slice(2,3);
// Expected contents
// sf2[0] == 3
// sf2[1] == 4

// Resize to size `5` and set entire field to value `2`
sf2.resize(5);
sf2 = 2;
// Assign value `3` to strided slice
sf2.slice(0, {}, 2) = 3;
// Expected contents
// sf2[0] == 3
// sf2[1] == 2
// sf2[2] == 3
// sf2[3] == 2
// sf2[4] == 3
```

CHAPTER 5

Lambdas

5.1 Simple example

```
#include <FMath/Core>

using FMath::Field;
using FMath::scalar;

Field<scalar> sf(5, 0.0);

auto lambda = [](std::size_t i, scalar& val)
{
    // Every value of the field is set to a calculated value
    val = i+5;
};

sf.apply_lambda(lambda);

// Example entries of `sf`
// sf[0] == 5
// sf[3] == 8
```

5.2 Complex example

This example shows a reasonably complex calculation:

1. each entry of `intermediate` should contain the sum of its "neighbours" in `orientations`
2. each entry of `res` should contain the scalar product of said sum with `orientations`

As should be, the first operation (in form of a lambda) and the second will be performed in parallel over the elements of the result `Field`.

```cpp
#include <FMath/Core>

using FMath::Field;
using FMath::scalar;
using FMath::Vector3;

int N = 5;
std::vector<int> relative_indices({-1,+1});
Field<Vector3> orientations(N, Vector3{0,0,1});
Field<Vector3> intermediate(N, Vector3{0,0,0});
Field<scalar> res(N, 0.0);

auto lambda = [&](std::size_t idx, Vector3& val)
{
    // Inside entries get both directions
    if( idx > 0 && idx < orientations.size()-1 )
    {
        for( auto& rel : relative_indices )
        {
            val += orientations[idx+rel];
        }
    }
    // Left end: right-hand side
    else if( idx == 0 )
        val += orientations[idx+1];
    // Right end: left-hand side
    else if( idx == orientations.size()-1 )
        val += orientations[idx-1];
};

// Result: scalar field of products of vectors in `orientations` and `intermediate`
res = orientations.dot(intermediate.applied_lambda(lambda));

// Contents of result:
// res[0] == 1
// res[1] == 2
// res[2] == 2
// res[3] == 2
// res[4] == 1
```

CHAPTER 6

Indices and tables

- genindex
- modindex
- search