Five Basic Classes in OpenFOAM

Hrvoje Jasak

h.jasak@wikki.co.uk

Wikki Ltd, United Kingdom

Outline

Objective

- Present in detail the implementation and functionality of five basic classes in OpenFOAM, concentrating on Finite Volume discretisation

Topics

- Space and time: `polyMesh`, `fvMesh`, `Time`
- Field algebra: `Field`, `DimensionedField` and `GeometricField`
- Boundary conditions: `fvPatchField` and derived classes
- Sparse matrices: `lduMatrix`, `fvMatrix` and linear solvers
- Finite Volume discretisation: `fvc` and `fvm` namespace

Space and Time

Representation of Time

- Main functions of `Time` class
  - Follow simulation in terms of time-steps: start and end time, delta t
  - Time is associated with I/O functionality: what and when to write
  - `objectRegistry`: all `IOobjects`, including mesh, fields and dictionaries registered with time class
  - Main simulation control dictionary: `controlDict`
  - Holding paths to `<root>`, `<case>` and associated data
- Associated class: `regIOobject`: database holds a list of objects, with functionality held under virtual functions

Space and Time

Representation of Space

- Computational mesh consists of
  - **List of points**. Point index is determined from its position in the list
  - **List of faces**. A face is an ordered list of points (defines face normal)
  - **List of cells** OR **owner-neighbour addressing** (defines left and right cell for each face, saving some storage and mesh analysis time)
  - List of boundary patches, grouping external faces
- `polyMesh` class holds mesh definition objects
- `primitiveMesh`: some parts of mesh analysis extracted out (topo changes)
- `polyBoundaryMesh` is a list of `polyPatches`

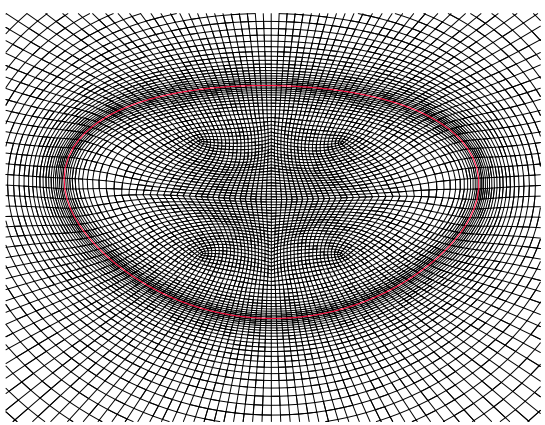
Finite Volume Mesh

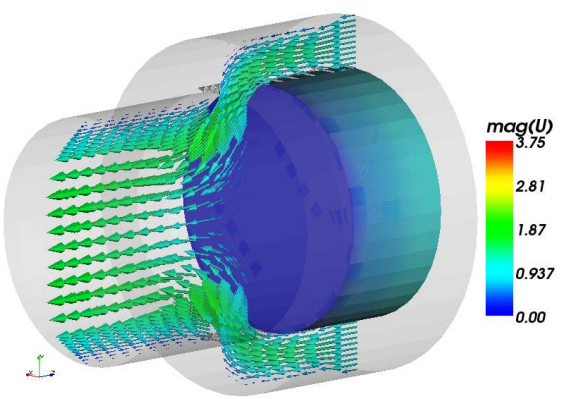
- `polyMesh` class provides mesh data in generic manner: it is used by multiple applications and discretisation methods
- For convenience, each discretisation wraps up primitive mesh functionality to suit its needs: mesh metrics, addressing etc.
- `fvMesh`: mesh-related support for the Finite Volume Method

Space and Time

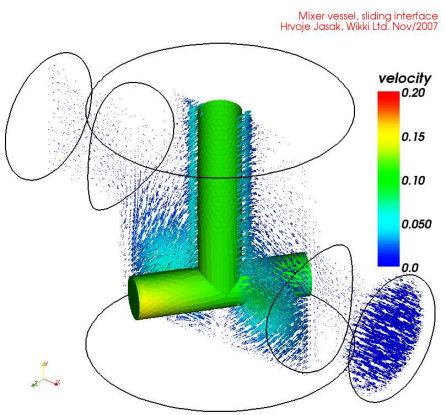
Representation of Space

- Further mesh functionality is generally independent of discretisation
  - Mesh motion (automatic mesh motion)
  - Topological changes
  - Problem-specific mesh templates: mixer vessels, moving boxes, pumps, valves, internal combustion engines etc.
- Implementation is separated into derived classes and mesh modifier objects (changing topology)
- Functionality located in the `dynamicMesh` library





Flow control device, sliding interface and mesh layering



Field Algebra

Field Classes: Containers with Algebra

- Class hierarchy of field containers
  - Unallocated list: array pointer and access
  - List: allocation + resizing
  - Field: with algebra
  - Dimensioned Field: I/O, dimension set, name, mesh reference
  - Geometric field: internal field, boundary conditions, old time

List Container

- Basic contiguous storage container in OpenFOAM: `List`
- Memory held in a single C-style array for efficiency and optimisation
- Separate implementation for list of objects (`List`) and list of pointers (`PtrList`)
  - Initialisation: `PtrList` does not require a null constructor
  - Access: dereference pointer in `operator[]()` to provide object syntax instead pointer syntax
  - Automatic deletion of pointers in `PtrList` destructor
- Somewhat complicated base structure to allow slicing (memory optimisation)

Field Algebra

Field

- Simply, a list with algebra, templated on element type
- Assign unary and binary operators from the element, mapping functionality etc.

Dimensioned Field

- A field associated with a mesh, with a name and mesh reference
- Derived from `IOobject` for input-output and database registration

Geometric Field

- Consists of an internal field (derivation) and a `GeometricBoundaryField`
- Boundary field is a field of fields or boundary patches
- Geometric field can be defined on various **mesh entities**
  - Points, edges, faces, cells
- . . . with various **element types**
  - scalar, vector, tensor, symmetric tensor etc
- . . . on various **mesh support classes**
  - Finite Volume, Finite Area, Finite Element
- Implementation involves a LOT of templating!

Boundary Conditions

Finite Volume Boundary Conditions

- Implementation of boundary conditions is a perfect example of a virtual class hierarchy
- Consider implementation of a boundary condition
  - Evaluate function: calculate new boundary values depending on **behaviour**: fixed value, zero gradient etc.
  - Enforce boundary type constraint based on matrix coefficients
  - Multiple if-then-else statements throughout the code: asking for trouble
  - **Virtual function interface**: run-time polymorphic dispatch
- Base class: `fvPatchField`
  - Derived from a field container
  - Reference to `fvPatch`: easy data access
  - Reference to internal field
- Types of `fvPatchField`
  - **Basic**: fixed value, zero gradient, mixed, coupled, default
  - **Constraint**: enforced on all fields by the patch: cyclic, empty, processor, symmetry, wedge, GGI
  - **Derived**: wrapping basic type for physics functionality

Sparse Matrix and Solver

Sparse Matrix Class

- Some of the oldest parts of OpenFOAM: about to be thrown away for more flexibility
- Class hierarchy
  - Addressing classes: `lduAddressing`, `lduInterface`, `lduMesh`
  - LDU matrix class
  - Solver technology: preconditioner, smoother, solver
  - Discretisation-specific matrix wrapping with handling for boundary conditions, coupling and similar

LDU Matrix

- Square matrix with sparse addressing. **Enforced strong upper triangular ordering in matrix and mesh**
- Matrix stored in 3 parts in **arrow format**
  - Diagonal coefficients
  - Off-diagonal coefficients, upper triangle
  - Off-diagonal coefficients, lower triangle
- Out-of-core multiplication stored as a list of `lduInterface` with coupling functionality: executed eg. on vector matrix multiplication

Sparse Matrix and Solver

LDU Matrix: Storage format

- Arbitrary sparse format. Diagonal coefficients typically stored separately
- Coefficients in 2-3 arrays: diagonal, upper and lower triangle
- Diagonal addressing implied
- Off-diagonal addressing in 2 arrays: “owner” (row index) “neighbour” (column index) array. Size of addressing equal to the number of coefficients
- The matrix structure (fill-in) is assumed to be symmetric: presence of a_{ij} implies the presence of a_{ji}. Symmetric matrix easily recognised: efficiency
- If the matrix coefficients are symmetric, only the upper triangle is stored – a symmetric matrix is easily recognised and stored only half of coefficients

```
vectorProduct(b, x) //   [b] = [A] [x]
{
    for (int n = 0; n < coeffs.size(); n++)
    {
        int c0 = owner(n);
        int c1 = neighbour(n);
        b[c0] = upperCoeffs[n]*x[c1];
        b[c1] = lowerCoeffs[n]*x[c0];
    }
}
```

Sparse Matrix and Solver

Finite Volume Matrix Support

- Finite Volume matrix class: `fvMatrix`
- Derived from `lduMatrix`, with a reference to the solution field
- Holding dimension set and out-of-core coefficient
- Because of derivation (insufficient base class functionality), all FV matrices are currently always scalar: **segregated** solver for vector and tensor variables
- Some coefficients (diagonal, next-to-boundary) may locally be a higher type, but this is not sufficiently flexible
- Implements standard matrix and field algebra, to allow matrix assembly at equation level: adding and subtracting matrices
- "Non-standard" matrix functionality in `fvMatrix`
  - `fvMatrix::A()` function: return matrix diagonal in FV field form
  - `fvMatrix::H()`: vector-matrix multiply with current `psi()`, using off-diagonal coefficients and rhs
  - `fvMatrix::flux()` function: consistent evaluation of off-diagonal product in "face form". See derivation of the pressure equation
- New features: coupled matrices (each mesh defines its own addressing space) and matrices with block-coupled coefficients

Finite Volume Method

Finite Volume Discretisation

- Finite Volume Method implemented in 3 parts
  - **Surface interpolation**: cell-to-face data transfer
  - **Finite Volume Calculus** (`fvc`): given a field, create a new field
  - **Finite Volume Method** (`fvm`): create a matrix representation of an operator, using FV discretisation
- In both cases, we have **static functions** with no common data. Thus, `fvc` and `fvm` are implemented as **namespaces**
- Discretisation involves a number of choices on how to perform identical operations: eg. gradient operator. In all cases, the signature is common

```
volTensorField gradU = fvc::grad(U);
```

- Multiple algorithmic choices of gradient calculation operator: Gauss theorem, least square fit, limiters etc. implemented as run-time selection
- Choice of discretisation controlled by the user on a per-operator basis: `system/fvSchemes`
- Thus, each operator contains basic data wrapping, selects the appropriate function from run-time selection and calls the function using virtual function dispatch

Finite Volume Method

Example: Gradient Operator Dispatch

```
template<class Type>
tmp
<
    GeometricField
    <
        outerProduct<vector,Type>::type, fvPatchField, volMesh
    >
>
grad
(
    const GeometricField<Type, fvPatchField, volMesh>& vf,
    const word& name
)
{
    return fv::gradScheme<Type>::New
    (
        vf.mesh(),
        vf.mesh().gradScheme(name)
    )().grad(vf);
}
```

Finite Volume Method

Example: Gradient Operator Virtual Base Class

- Virtual base class: `gradScheme`

```
template<class Type>
class gradScheme
:
    public refCount
{
    //- Calculate and return the grad of the given field
    virtual tmp
    <
        GeometricField
        <outerProduct<vector, Type>::type, fvPatchField, volMesh>
    > grad
    (
        const GeometricField<Type, fvPatchField, volMesh>&
    ) const = 0;
};
```

Finite Volume Method

Example: Gauss Gradient Operator, Business End

```
forAll(owner, facei)
{
    GradType Sfssf = Sf[facei]*issf[facei];
    igGrad[owner[facei]] += Sfssf;
    igGrad[neighbour[facei]] -= Sfssf;
}

forAll(mesh.boundary(), patchi)
{
    const unallocLabelList& pFaceCells =
        mesh.boundary()[patchi].faceCells();
    const vectorField& pSf = mesh.Sf().boundaryField()[patchi];
    const fvsPatchField<Type>& pssf = ssf.boundaryField()[patchi];

    forAll(mesh.boundary()[patchi], facei)
    {
        igGrad[pFaceCells[facei]] += pSf[facei]*pssf[facei];
    }
}

igGrad /= mesh.V();
```

Summary

Summary: Five Basic Classes in OpenFOAM (FVM Discretisation)

- Representation of space: hierarchy of mesh classes
- Representation of time: `Time` class with added database functions
- Basic container type: `List` with contiguous storage
- Boundary condition handling implemented as a virtual class hierarchy
- Sparse matrix support: arrow format, separate upper and lower triangular coefficients
- Discretisation implemented as a calculus and method namespaces. Static functions perform dispatch using run-time selection and virtual functions