Five Basic Classes in OpenFOAM

Hrvoje Jasak, Wikki

United Kingdom and Germany

Outline

Objective

- Present a narrative on numerical simulation software development

Topics

- Design and limitations of contemporary CFD software
- Model representation through equation mimicking
- Object orientation, generic programming and library design
- Top-level solvers and utilities: interactivity
- Five Basic Classes in OpenFOAM
  - 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
- Summary

Background

FSB

State of the Art in CFD Software Use

- Numerical modelling is a part of product design
  - Improvements in computer performance: CPU speed and memory
  - Improved physical modelling and numerics, covering more physics
  - Sufficient validation and experience with modelling capabilities
- Two-fold requirements in CFD development
  - Integration into the CAD-based design process
  - Quick and reliable implementation of new and complex physical models
- Complex geometry support, high-performance computing, automatic meshing, dynamic mesh capabilities etc. needed across the spectrum
- Opening new areas of CFD simulation
  - Non-traditional physics: complex heat and mass transfer models, electromagnetics, fluid-structure interaction
  - New solution techniques, eg. flow and shape optimisation; robust design

Design of Modern Solvers

Physics and Numerics

- A single discretisation method (FVM, FEM), parallelism and vectorisation
- User-defined modifications inefficient and limiting: proprietary software
- Model-to-model interaction matrix is becoming increasingly complex

Software Organisation

- Functional approach: centralised data and multiple functions operating on it
- Monolithic implementation and integrated software: single executable for all cases
- A CFD software is a large project: order of 1-2 million lines of source code
- Complex solver-to-solver interaction or embedding virtually impossible: two solvers, solver and mesh generator, embedding in a CAD environment

Consequences

- Difficulties in development, maintenance and support
- Some new simulation techniques cannot be accommodated at all: sensitivity
- In spite of the fact the all components are present, it is extremely difficult to implement “non-traditional” models

Based on the above, change of paradigm is overdue!

Object Orientation

Object-Oriented Software: Create a Language Suitable for the Problem

- Analysis of numerical simulation software through object orientation: “Recognise main objects from the numerical modelling viewpoint”
- Objects consist of **data** they encapsulate and **functions** which operate on the data

Example: Sparse Matrix Class

- **Data members**: protected and managed
  - Sparse addressing pattern (CR format, arrow format)
  - Diagonal coefficients, off-diagonal coefficients
- Operations on matrices or data members: **Public interface**
  - Matrix algebra operations: $+, -, *, /,$
  - Matrix-vector product, transpose, triple product, under-relaxation
- Actual data layout and functionality is important only internally: efficiency

Example: Linear Equation Solver

- Operate on a system of linear equations $[A][x] = [b]$ to obtain $[x]$
- It is irrelevant how the matrix was assembled or what shall be done with solution
- Ultimately, even the solver algorithm is not of interest: all we want is new x!
- Gauss-Seidel, AMG, direct solver: all answer to the same interface

Object-Oriented Numerics for CCM

Implementation of Complex Physics Models

- Flexible handling of arbitrary equations sets is needed
- Natural language of continuum mechanics: partial differential equations
- Example: turbulence kinetic energy equation

$$\frac{\partial k}{\partial t} + \nabla\bullet(\mathbf{u}k) - \nabla\bullet[(\nu + \nu_t)\nabla k] = \nu_t \left[\frac{1}{2}(\nabla\mathbf{u} + \nabla\mathbf{u}^T)\right]^2 - \frac{\epsilon_o}{k_o}\,k$$

- Objective: **Represent differential equations in their natural language**

```
solve
(
    fvm::ddt(k)
  + fvm::div(phi, k)
  - fvm::laplacian(nu() + nut, k)
 == nut*magSqr(symm(fvc::grad(U)))
  - fvm::Sp(epsilon/k, k)
);
```

- Correspondence between the implementation and the original equation is clear

Object Orientation

FSB

Object Oriented Analysis

- Main object = **operator**, eg. time derivative, convection, diffusion, gradient
- How do we represent an operator in CFD?
  - A **field**: explicit evaluation
  - A **matrix**: implicit algorithm
- . . . but we need many other components to assemble the equation
  - Representation of space and time: computational domain
  - Scalars, vectors and tensors
  - Field representation and field algebra
  - Initial and boundary condition
  - Systems of linear equations and solvers
  - Discretisation methods
- The above does not complete the system
  - Physical models, (eg. turbulence) and model-to-model interaction
  - Pre- and post-processing and related utilities
  - Top-level solvers

Object Orientation

Main Objects

- Computational domain

Object	Software representation	C++ Class
Tensor	(List of) numbers + algebra	vector, tensor
Mesh primitives	Point, face, cell	point, face, cell
Space	Computational mesh	polyMesh
Time	Time steps (database)	time

- Field algebra

Object	Software representation	C++ Class
Field	List of values	Field
Boundary condition	Values + condition	patchField
Dimensions	Dimension set	dimensionSet
Geometric field	Field + mesh + boundary conditions	geometricField
Field algebra	$+ \; - \; * \; / \; tr(), sin(), exp() \ldots$	field operators

Object Orientation

Main Objects

- Linear equation systems and linear solvers

Object	Software representation	C++ Class
Linear equation matrix	Matrix coefficients	lduMatrix
Solvers	Iterative solvers	lduMatrix::solver

- Numerics: discretisation methods

Object	Software representation	C++ Class
Interpolation	Differencing schemes	interpolation
Differentiation	ddt, div, grad, curl	fvc, fec
Discretisation	ddt, d2dt2, div, laplacian	fvm, fem, fam

- Top-level code organisation

Object	Software representation	C++ Class
Model library	Library	eg. turbulenceModel
Application	main()	–

Generic Programming

Generic Implementation of Algorithms: Templating

- A number of algorithms operate independently of the type they operate on
- Examples: Containers (list of objects of the same type), list sorting
- . . . but also in CFD: field algebra, discretisation, interpolation, boundary conditions
- Ideally, we want to implement algorithms generically and produce custom-written code for compiler optimisation
- Templating mechanism in C++
  - Write algorithms with a generic type holder

    ```
    template<class Type>
    ```
  - Use generic algorithm on specific type

    ```
    List<cell> cellList(3);
    ```
  - Compiler to expand the code and perform optimisation after expansion
- Generic programming techniques massively increase power of software: less software to do more work
- Debugging is easier: if it works for one type, it will work for all
- . . . but writing templates is trickier: need to master new techniques
- Templating allows introduction of side-effects: forward derivatives

Model-to-Model Interaction

Common Interface for Model Classes: Model Libraries

- Physical models grouped by functionality, eg. material properties, viscosity models, turbulence models etc.
- Each model answers the interface of its class, but its implementation is separate and independent of other models
- The rest of software handles the model through a generic interface: breaking the complexity of the interaction matrix

```
class turbulenceModel
{
    virtual volSymmTensorField R() const = 0;
    virtual fvVectorMatrix divR
    (
        volVectorField& U
    ) const = 0;
    virtual void correct() = 0;
};
```

- New turbulence model implementation : Spalart-Allmaras

```
class SpalartAllmaras : public turbulenceModel{};
```

Model-to-Model Interaction

Common Interface for Model Classes: Model Libraries

- Model user only sees the virtual base class
- Example: steady-state momentum equation with turbulence

```
autoPtr<turbulenceModel> turbulence =
    turbulenceModel::New(U, phi, laminarTransport);

fvVectorMatrix UEqn
(
    fvm::ddt(rho, U)
  + fvm::div(phi, U)
  + turbulence->divR(U)
 ==
  - fvc::grad(p)
);
```

- Implementation of a new model does not disturb the existing models
- Consumer classes see no changes whatsoever, just a new choice

Run-Time Selection

Handling Model Libraries

- New components do not disturb existing code: fewer new bugs
- Run-time selection tables: dynamic binding for new functionality
- Used for every implementation: “user-coding”
  - Differencing schemes: convection, diffusion, rate of change
  - Gradient calculation
  - Boundary conditions
  - Linear equation solvers
  - Physical models, eg. viscosity, turbulence, evaporation, drag etc.
  - Mesh motion algorithms
- Library components may be combined at will: new use for existing software!
- Ultimately, there is no difference between pre-implemented models and native library functionality: no efficiency concerns
- Implemented models are examples for new model development

Layered Development

OpenFOAM Software Architecture

- Design encourages code re-use: developing shared tools
- Development of model libraries: easy model extension
- Code developed and tested in isolation
  - Vectors, tensors and field algebra
  - Mesh handling, refinement, mesh motion, topological changes
  - Discretisation, boundary conditions
  - Matrices and solver technology
  - Physics by segment
  - Custom applications
- Custom-written top-level solvers optimised for efficiency and storage
- Substantial existing capability in physical modelling
- Mesh handling support: polyhedral cells, automatic mesh motion, topological changes
- **Ultimate user-coding capabilities!**

Five Basic Classes

Five Basic Classes in OpenFOAM

- 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 `IOobject`s, including mesh, fields and dictionaries registered with time class
  - Main simulation control dictionary: `controlDict`
  - Holding paths to `<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 `polyPatch`es

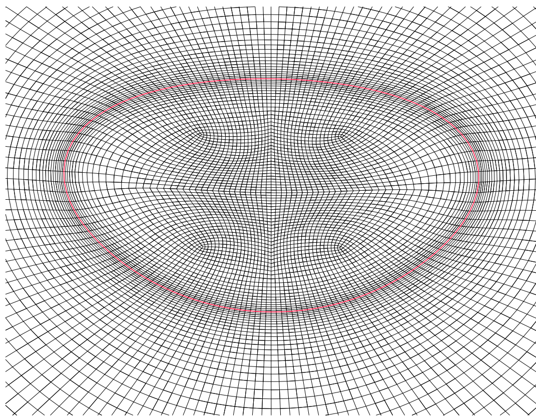
Finite Volume Mesh

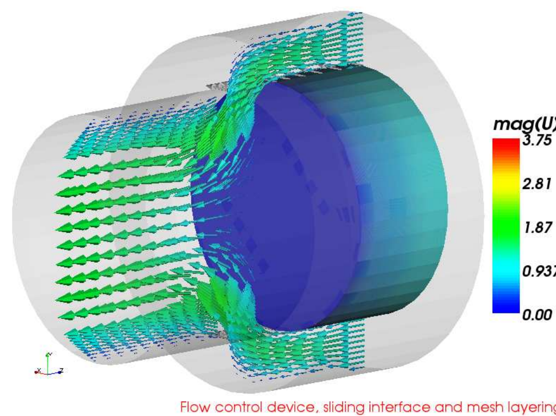
- `polyMesh` class provides mesh data in generic manner: it is used by multiple applications and discretisation methods
- For convenience, each discretisation wraps up primitive mesh functionality to suit its needs: mesh metrics, addressing etc.
- `fvMesh`: mesh-related support for the Finite Volume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

Space and Time

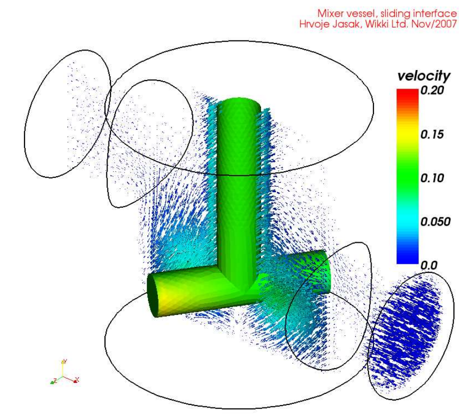
Representation of Space

- 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



Mixer vessel, sliding interface
Hrvoje Jasak, Wikki Ltd. Nov/2007

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

FSB

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

FSB

Summary and Conclusions: Numerical Simulation Software

- Object-oriented design offers a way to manage large software projects more easily. Numerical simulation software definitely qualifies
- There are **no performance issues** with object-orientation, generic programming and library structure in scientific computing
- Object orientation and generic programming is by no means the only way: for smaller projects or limited scope, “old rules still apply”

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