IAEA
International Atomic Energy Agency

Multi-physics modeling and simulation of nuclear reactors using OpenFOAM

30 Aug 2022 – 6 October 2022 (every Tuesday & Thursday)

*Contact: ONCORE@iaea.org*

# Lecture 2, part C:
# A practical introduction to OpenFOAM - Source code

8 September 2022
Ivor Clifford
Paul Scherrer Institut
ivor.clifford@psi.ch

Course Enrollment : Multi-physics modelling and simulation of nuclear reactors using OpenFOAM

ONCORE: Open-source Nuclear Codes for Reactor Analysis

# Lecture Outline

- Introduction
- OpenFOAM library structure
- Basic classes
- Structure of a typical solver
- Beyond the basics
- Time for Q&A

# Goal of Lecture

This lecture is intended to give a flavour of how the source code of OpenFOAM is designed and structured

- Approached from a programming perspective
- Detailed knowledge of C++ and object-oriented programming is not needed, but will help
- Detailed knowledge of Linux is not needed, but will help

You will not be able to create your own solver from scratch, but this will hopefully save you a lot of reading and point you in the right direction.

**An additional note**

OpenFOAM is HUGE and can take years to master

# Useful Resources

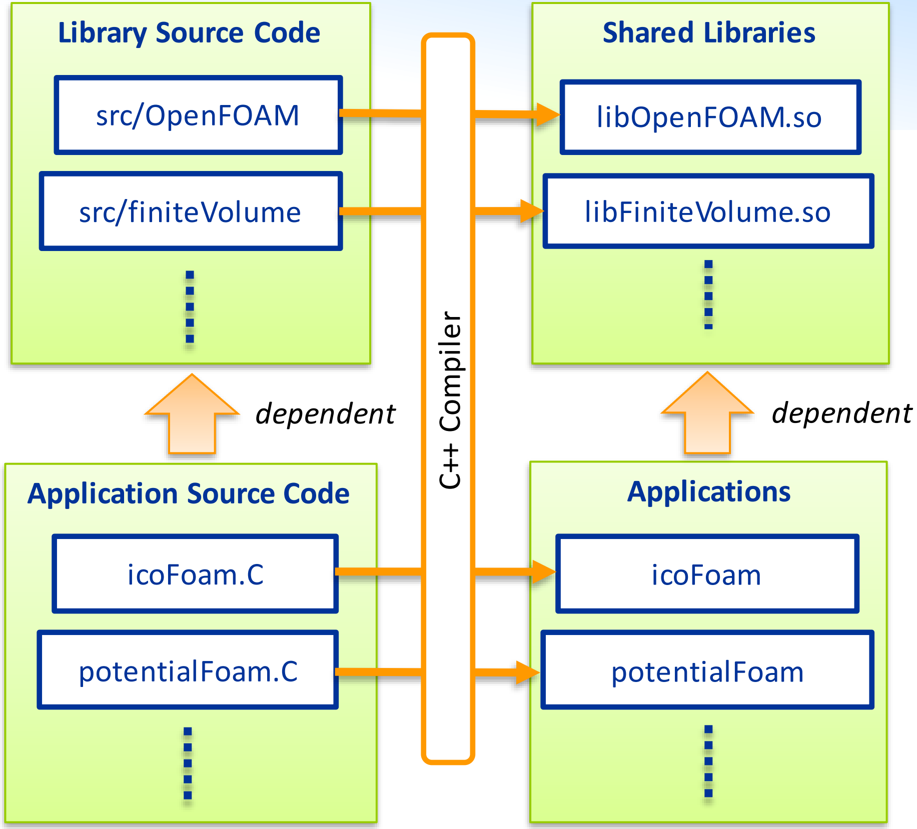
- H. Jasak and H. Rusche, **Five Basic Classes in OpenFOAM**, Advanced Training at the OpenFOAM Workshop, 21.6.2010, Gothenborg, Sweden
- H. Jasak, **OpenFOAM: Introduction and Basic Class Layout**, OpenFOAM in Industrial Combustion Simulations, Pohang University Feb/2015
- H. Jasak, **OpenFOAM: Linear System and Linear Solver**, OpenFOAM in Industrial Combustion Simulations, Pohang University Feb/2015
- H. Jasak, **Finite Volume Discretisation in OpenFOAM: Best Practice Guidelines**, OpenFOAM in Industrial Combustion Simulations, Pohang University Feb/2015
- **Proceedings of CFD with OpenSource Software**, 2020, Edited by H. Nilsson, http://dx.doi.org/10.17196/OS_CFD#YEAR_2020

IAEA
International Atomic Energy Agency

# OpenFOAM Library Structure

# What Exactly is OpenFOAM?

- OpenFOAM is a set of C++ libraries and applications
  - **Library**: Collection of constants, static variables, functions and classes with common functionality; compiled as a Linux shared object (.so) library
  - **Application**: Top-level executable run from the command-line



# Getting Started with OpenFOAM

- OpenFOAM v2112 contains ~100 libraries and ~4600 classes
- Luckily, the average user/developer only needs to be concerned with a very small subset of these

- Full OpenFOAM API can be browsed online
  https://www.openfoam.com/documentation/guides/latest/api/index.html

- For beginners, recommended approach is to find existing solvers and developments that closely match your requirements and build from these

# Basic Libraries

- For very basic or first-of-a-kind solvers, only two libraries are needed
- For more advanced (CFD) solvers, additional libraries are available

| Library | Source Location | Description |
|---|---|---|
| libOpenFOAM.so | src/OpenFOAM | OpenFOAM base library<br>(basic types, containers, I/O, fields, meshes, matrices, matrix solvers |
| libfiniteVolume.so | src/finiteVolume | Finite-volume classes<br>(finite-volume meshes, discretisation schemes, fields, standard boundary conditions and matrices) |
| Others | src/* | Thermophysical models, turbulence modelling, chemistry, ODE solvers, mesh manipulation, non-conformal mapping, … |

# OpenFOAM Coding Style

- OpenFOAM is coded almost entirely in C++
  - OpenFOAM heavily uses C++ features
  - Latest versions support the C++11 standard
  - OpenFOAM **does not** use the C++ standard template library (STL); development started before STL existed. Many STL classes are represented in OpenFOAM equivalent classes, .e.g. smart pointers (`AutoPtr`), maps (`HashTable`), lists (`List`).
  - Coding guidelines are comprehensive and consistent throughout. Source code layout and commenting style differ from many other C++ projects, but basic rules are consistent.

# OpenFOAM Build System

- OpenFOAM uses a custom build system, **wmake**, built on top of GNU Make
  - Handles the detailed options and rules of a traditional makefile
  - Effectively, the user just provides a list of **source code** files, **prerequisite libraries** and the name of the resulting **executable** or **library** and wmake does the rest
- **Pros**
  - Perfect for building typical OpenFOAM solvers. It's simple and it works
- **Cons**
  - Programmer IDEs don't directly support wmake. Workarounds are needed.
  - Less perfect for coupling or embedding other codes since there may be build system incompatibilities. This issue is not unique to OpenFOAM, though.

## Useful OpenFOAM Environment Variables

| Environment Variable | Description |
|---|---|
| `$WM_PROJECT_DIR` | OpenFOAM installation directory |
| `$FOAM_APPBIN` | Location of OpenFOAM compiled executables |
| `$FOAM_LIBBIN` | Location of OpenFOAM libraries (.so files) |
| `$WM_PROJECT_USER_DIR` | OpenFOAM user directory (user-developed solvers, libraries, etc.) |
| `$FOAM_USER_APPBIN` | User's executables |
| `$FOAM_USER_LIBBIN` | User's libraries (.so files) |

# 2 Minute Guide to Object-Oriented Programming (OOP)

- Real-world objects have state and behaviour
  - State = information about the object, e.g. size, colour, position
  - Behaviour = what can the object do, e.g. change size, move, interact with another object
- C++ classes embrace this idea
  - Each class allocates and manages its own data
  - Each class provides a clear API for interacting with the outside world
  - Classes can be inherited, creating variants of the base class, e.g. different boundary conditions, turbulence models, discretisation schemes (**polymorphism**)

```
class vector
{
    scalar x_, y_, z_;

public:

    vector(scalar x, scalar y, scalar z);

    // Magnitude of vector
    scalar mag();

    // Scalar multiplication, scalar*vector
    vector operator*(scalar&);

    // Vector addition, vector + vector
    vector operator+(vector&);

    // Dot-product, vector & vector
    scalar operator&(vector&);
}
```

# 2 Minute Guide to Object-Oriented Programming (OOP)

**So what's the advantage?**

- As an outside user you only need to know how to construct class instances and interact with them
- If classes are well designed, it's very difficult to break them as an outside user
- Since classes are self-encapsulated, it's extremely easy to reuse functionality.
- Polymorphism combined with runtime selection provides freedom to extend existing code without risk of breaking it; No more updating `if-then-else` blocks to add new materials/models.

**Are there any disadvantages?**

- To be effective, classes must be carefully designed; Requires additional up-front planning.

Luckily, OpenFOAM was very carefully planned and the available classes are extremely effective.

IAEA
International Atomic Energy Agency

# Basic Classes of OpenFOAM

# The Basic Classes of OpenFOAM

- OpenFOAM was originally conceived as a library for computational continuum mechanics (CCM)
- Basic classes of OpenFOAM closely relate to the requirements for this
  - Time
  - Space
  - Time and space-dependent variables
  - Physical equations (PDEs) and their solution

# The Basic Classes of OpenFOAM

## A short note on C++ templates

OpenFOAM uses template classes extensively; For some classes, the templates are too complex to present effectively here. For simplicity, I have therefore avoided all template parameters, e.g.

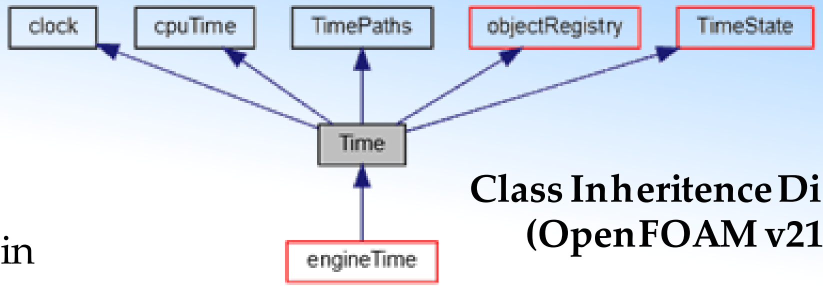
- `GeometricField<Type, PatchField, GeoMesh>` is simply `GeometricField`
- `GeometricField<scalar, fvPatchField, volMesh>` is `volScalarField`

## A short note on C++ namespaces

OpenFOAM defines one major namespace containing virtually all of its functionality, `namespace Foam`. Unless otherwise mentioned, all classes and functions included in this presentation are in this namespace.

# Time Control

# `Foam::Time`



Class Inheritence Diagram
(OpenFOAM v2112)

## Key Responsibilities

- Defines the current simulation time (in seconds)
- Controls the time stepping ($t + \Delta t$)
- Controls the reading and writing of data (e.g. simulation inputs/outputs). What to write and when to write it.

## Other lesser known responsibilities

- Provides a mapping between user time and physical time (s)
- Controls the file paths for the simulation
- Keeps a registry of top-level global objects, e.g. meshes, fields, thermophysical models
- Manages the execution of functionObjects

## Example

```
#include "createTime.H"

while (runTime.loop())
{
    // Print current timestep
    Info<< "Time = " << runTime.timeName()
        << nl << endl;

    // DO STUFF HERE

    // Write simulation outputs
    runTime.write();
}

// End of simulation
```

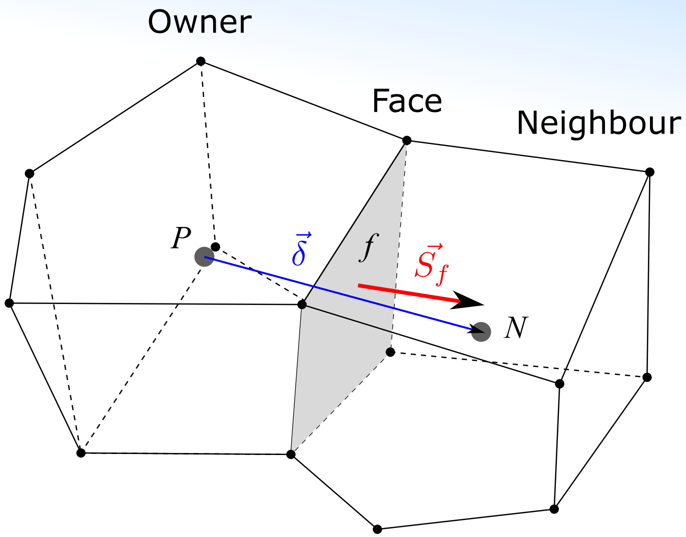
# Space

# `Foam::fvMesh`

**Key Responsibilities**

- Store and calculate topological and geometric information related to the finite-volume mesh (points, faces, cells, boundary definitions) and associated connectivity information

- Computational mesh consists of
  - List of **points** (3D vectors)
  - List of **faces** (list of point IDs)
  - List of **cells** (list of bounding face IDs)
  - **Owner-neighbour** addressing (owner and neighbour cell IDs for each face)
  - List of **patches** / boundary definitions



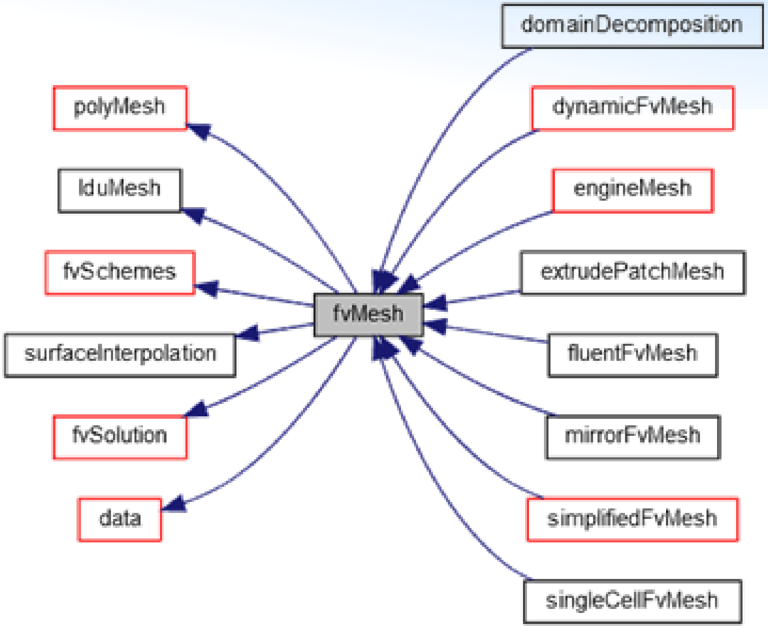
**Finite-volume Mesh Cells and Associated Connectivity**

# Space

## Foam::fvMesh

**Other lesser known responsibilities**

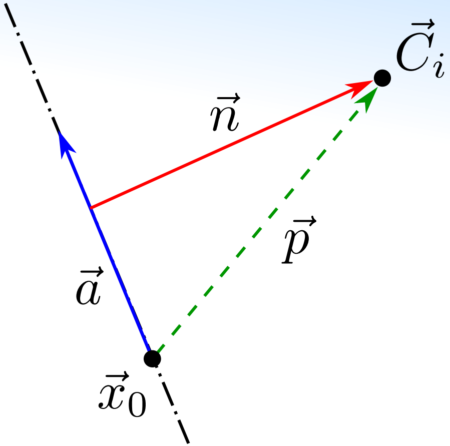
- Move and adapt the mesh for cases with refinement, morphing of geometric change
  - Managed by derived classes
- Define addressing for solution matrix



**Class Inheritence Diagram (OpenFOAM v2112)**

# Vector and Tensor Mathematics

- OpenFOAM provides several classes for vector and tensor algebra
- Both 2D and 3D vectors and tensors are defined, e.g. `vector2D` and `vector`
- Many operators and functions are provided, e.g.
  - Addition, subtraction, scaling
  - magnitude
  - dot product
  - outer product
  - transpose
  - eigenvalues
  - many more …



```
// Calculate radius to the centre
// of a cell from a given axis
vector p = mesh.C()[cellI] - x0;
vector n = p - (p & a)*a;
scalar radius = mag(n);
```

# Vector and Tensor Mathematics

| Description | Class | Rank | Components | Definition |
|---|---|---|---|---|
| Scalar value | `scalar` | 0 | 1 | $v$ |
| 3D vector | `vector` | 1 | 3 | $\vec{v} = (v_x, v_y, v_z)$ |
| 3×3 tensor | `tensor` | 2 | 9 | $\mathbf{T} = \begin{pmatrix} T_{xx} & T_{xy} & T_{xz} \\ T_{yx} & T_{yy} & T_{yz} \\ T_{zx} & T_{zy} & T_{zz} \end{pmatrix}$ |
| 3×3 symmetric tensor | `symmTensor` | 2 | 6 | $\mathbf{T} = \begin{pmatrix} T_{xx} & T_{xy} & T_{xz} \\ \vdots & T_{yy} & T_{yz} \\ & \dots & T_{zz} \end{pmatrix}$ |
| 3×3 diagonal tensor | `diagTensor` | 2 | 3 | $\mathbf{T} = \begin{pmatrix} T_{xx} & 0 & 0 \\ 0 & T_{yy} & 0 \\ 0 & 0 & T_{zz} \end{pmatrix}$ |
| 3×3 spherical tensor | `sphericalTensor` | 2 | 1 | $\mathbf{T} = \begin{pmatrix} T & 0 & 0 \\ 0 & T & 0 \\ 0 & 0 & T \end{pmatrix}$ |

# Physical Units

- Almost all physical quantitites have units (mass, velocity, temperature)
- OpenFOAM provides basic unit handling and checking during run time. Code will print an error if units are inconsistent.
- Handled by class `dimensionSet`
- Units either in terms of the 7 base SI units (mass, length, time, temperature, moles, current and luminous intensity) or a large selection of predefined units, e.g. `dimPower`, `dimAcceleration`, `dimHeatCapacity`.
- Basic dimensioned types provided, e.g. `dimensionedScalar`, `dimensionedVector`. Equivalent to the original types but with a name and units.

```
#include "dimensionSets.H"
…
// Energy of a mass at fixed velocity
dimensionedScalar m(1000, dimMass);
dimensionedVector v(vector(20,10,0), dimVelocity);
dimensionedScalar E = 0.5*m*magSqr(v);
```

# Physical Variables

## `Foam::GeometricField`

Templated for scalar, vector, tensor and symmetric tensor variables and FV discretisation (cell centres and faces)

**volFields – defined at centre of cells**

- `volScalarField`, scalar variable at cell centres
- `volVectorField`, 3D vector variable "
- `volTensorField`, 3x3 tensor variable "

**surfaceFields – defined at faces**

- `surfaceScalarField`, scalar variable at face centres

**Key Functionality and Responsibilities**

- Store current field value, previous iteration and previous time step values at cell centres or faces
- Each field has a name and is associated with a mesh
- Provides operators for field manipulation (field arithmetic)
- Store separate fields (`fvPatchField`) containing boundary values and providing boundary conditions

```
// Read velocity field
volVectorField U
(
    IOobject
    (
        "U",
        runtime.timeName(),
        runTime,
        IOobject::MUST_READ
        IOobject::AUTO_WRITE
    )
);

// Calculate mass flux at faces
surfaceScalarField phi = (
    fvc::interpolate(rho*U) & mesh.Sf()
);
```

$$\phi_f = (\rho U)_f \cdot \vec{S}_f$$

# Physical Variables

GeometricField is derived multiple times from several parent classes

- UList = list/vector/array of values without memory allocation
- List = UList + memory allocation
- Field = List + linear algebra
- DimensionedField = Field + name + I/O + dimensions + reference to mesh
- GeometricField = DimensionedField + boundary fields + previous iteration and time step values

**Field Algebra**

- Basic algebraic operators apply (+, -, *, /, &)
- Many mathematical functions supported (mag, pow, sqr, magSqr, min, max, sum, …)
- For GeometricField, everything is parallelised
- For Field and List, you need to be aware of how OpenFOAM handles parallelism and design your code accordingly

```
// Read velocity field
volVectorField U
(
    IOobject
    (
        "U",
        runtime.timeName(),
        runTime,
        IOobject::MUST_READ
        IOobject::AUTO_WRITE
    )
);

// Calculate mass flux at faces
surfaceScalarField phi = (
    fvc::interpolate(rho*U) & mesh.Sf()
);
```

$$\phi_f = (\rho U)_f \cdot \vec{S}_f$$

# Boundary Conditions
# `Foam::fvPatchField`

- In OpenFOAM, each subset the domain boundary is referred to as a patch (`fvPatch`)
  - Each patch has a name, ID, a list of faces and connectivity for the boundary-adjacent cells
- For each `volField`, boundary conditions (`fvPatchField`) are provided for each patch
- OpenFOAM includes a large collection of BCs, subclassed from `fvPatchField`

**Key responsibilities**

- Calculate and store the boundary values at each boundary face
- Provide coefficients for calculating the field value and surface normal gradient on the boundary, i.e.

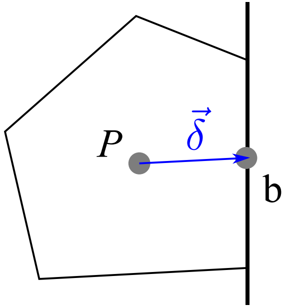
$$x_b = a_P x_P + a_b$$
$$(\nabla x)_b \cdot \vec{S}_b = a_P x_P + a_b$$

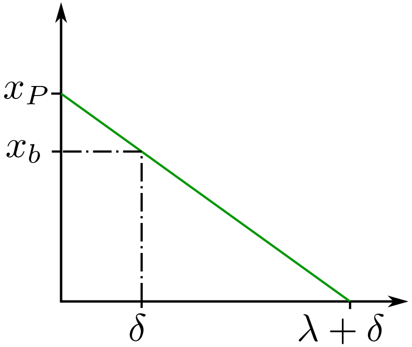
$P$ refers to the the boundary-adjacent cell, $b$ refers to the boundary.
$a_P$ is the **internal coefficient**. $a_b$ is the **boundary coefficient**.

# Boundary Conditions

## `Foam::fvPatchField`

**Example** – Extrapolated length boundary condition for neutron diffusion ($x$ goes to zero some distance $\lambda$ beyond the boundary)





$$x_b = \frac{\lambda}{\lambda + \delta} x_P$$

$$(\nabla x)_b \cdot \vec{S}_b = -\frac{1}{\lambda + \delta} x_P$$

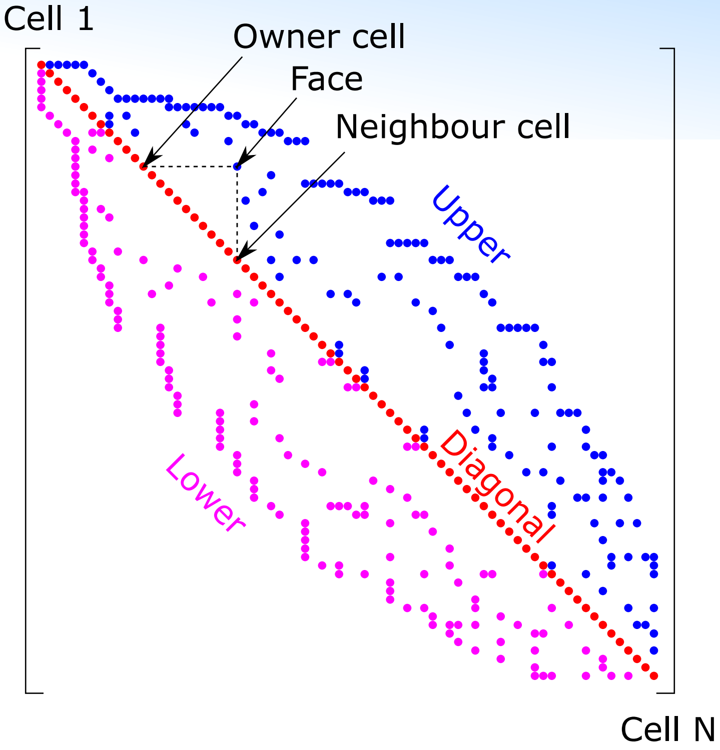
- Boundaries may optionally be "coupled", i.e. the boundary value depends on the value in another region and is obtained by data exchange
  - e.g. periodic BCs (`cyclicFvPatchField`) or processor BCs (`processorFvPatchField`)
  - In this case the boundary coefficient $a_b$ is implicit

$$x_b = a_P x_P + a_b x_b$$

$$(\nabla x)_b \cdot \vec{S}_b = a_P x_P + a_b x_b$$

# Sparse Matrix Linear Systems

- OpenFOAM provides a comprehensive set of linear sparse matrix solvers and preconditioners
- Linear matrix system (`fvMatrix`): $\mathbf{A}x = B$
  - $\mathbf{A}$ is the sparse matrix (N×N)
  - $B$ is the source vector
  - $x$ is the solution vector, i.e. the variable you are solving for
- Matrix store as separate vectors (arrays) for diagonal, lower triangular matrix and upper triangular matrix ($\mathbf{A} = \mathbf{L} + \mathbf{D} + \mathbf{U}$).
- Matrix structure is strongly linked to finite-volume mesh; matrix diagonal values correspond to cells, while off-diagonal values correspond to faces
  - Ordering of matrix diagonal and off-diagonal entries corresponds to ordering of mesh cells and faces
  - Extremely convenient because it simplifies the matrix construction considerably



# Partial Differential Equations

- PDEs combine simple algebra with spatial and time-dependent operations, e.g. gradients and time-derivatives
- OpenFOAM provides a comprehensive collection of typical operators to build up PDEs using finite-volume discretisation
- Two separate namespaces are defined:
  - `Foam::fvm`: fully- or semi-implicit operators, which return matrices (`fvMatrix`).
  - `Foam::fvc`: fully explicit operators that return `GeometricField`, i.e. they simply evaluate the current value
- OpenFOAM provides all necessary algebra to add, subtract, scale, etc., fields and matrices to build full equation systems, e.g.

```
fvScalarMatrix + volScalarField → fvScalarMatrix
volScalarField × fvScalarMatrix → fvScalarMatrix
fvScalarMatrix == volScalarField → fvScalarMatrix
```

- Equality operator (==) simply subtracts everything on the right hand side of the equation from the left hand side

# Equation Operators

The list of available operators should suffice for 90% of problems

| Operator | Function | Description |
|---|---|---|
| $\frac{\partial \rho x}{\partial t}$ | `fvm::ddt(rho,x)`<br>`fvc::ddt(rho,x)` | Time-derivative |
| $\frac{\partial^2 \rho x}{\partial t^2}$ | `fvm:d2dt2(rho,x)`<br>`fvc:d2dt2(rho,x)` | Second-order time-derivative |
| $\nabla \cdot \gamma \nabla x$ | `fvm:laplacian(gamma,x)`<br>`fvc:laplacian(gamma,x)` | Laplacian |
| $\nabla \cdot \phi x$ | `fvm:div(phi,x)`<br>`fvc::div(phi,x)` | Divergence |
| $Kx$ | `fvm::Su(K,x)`<br>`fvm::Sp(K,x)`<br>`fvm::SuSp(K,x)` | Linearised source<br>Fully implicit (matrix diagonal)<br>Fully explicit (matrix source)<br>Semi-implicit (based on sign of $K$) |

# Some PDE Examples

Mass conservation (continuity) equation

$$\frac{\partial \rho}{\partial t} + \nabla \cdot \phi = S$$

```
solve(
    fvm::ddt(rho)
  + fvc::div(phi)
 ==
    S
);
```

# Some PDE Examples

One-group neutron diffusion equation

$$\frac{1}{v}\frac{\partial\varphi}{\partial t} - \nabla\cdot D\nabla\varphi + \left(\Sigma_a - \nu\Sigma_f\right)\varphi = S$$

```
solve(
    fvm::ddt(phi)/v
  - fvm::laplacian(D, phi)
  + fvm::Sp(sigma_a - nuSigma_f, phi)
 ==
    S
);
```

# Some PDE Examples

Solid mechanics momentum equation

$$\frac{\partial}{\partial t}\rho\frac{\partial\vec{u}}{\partial t^2} = \nabla\cdot\sigma$$

where

$$\sigma = \mu\nabla\vec{u} + \mu(\nabla\vec{u})^T + \lambda\,\mathrm{tr}(\vec{u})\mathbf{I}$$

**Simplest approach (not best)**

```
solve(
    fvm::d2dt2(rho,U)
 ==
    fvm::laplacian(mu, U)
  + fvc::div(mu*grad(U).T() + lambda*tr(U)*I)
);
```

**Requires iteration**

Implicit

Explicit

# Other Field Operators

`Foam::fvc` contains some additional functions that can be extremely useful

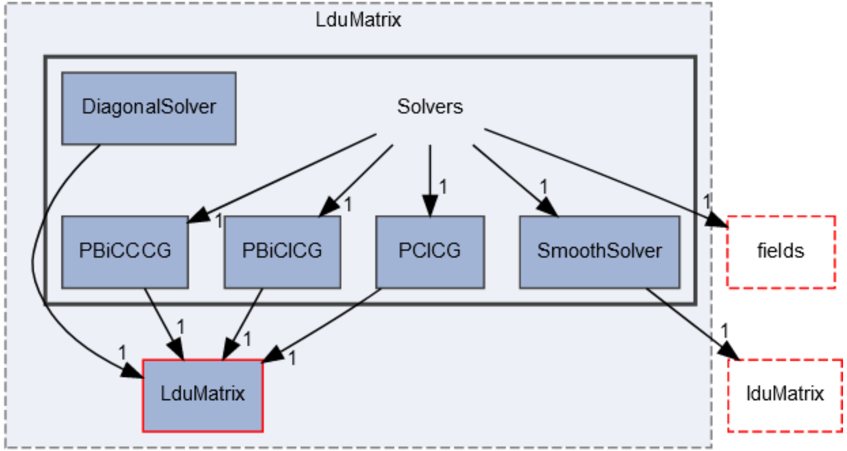
| Function | Description |
|---|---|
| `fvc::interpolate` | Interpolate the cell centre values to the faces<br>Receives a volField. Returns a surfaceField |
| `fvc::snGrad` | Surface normal gradient of a field, $(\nabla x)_f \cdot \overrightarrow{S_f}$<br>Receives a volField. Returns a surfaceField |
| `fvc::average` | Area-weighted average of the faces bounding each cell.<br>Returns a volField |
| `fvc::volumeIntegrate` | Volume integral in each cell.<br>Returns a volField |
| `fvc::domainIntegrate` | Volume integral over the entire domain, $\int x \, dV$<br>Returns a dimensioned value |

# Equation Solution

- Once the matrix is constructed, it may be kept in memory and/or solved as needed

```
fvScalarMatrix LHS = fvm::ddt(rho) + fvc::div(phi);
solve(LHS == S);
```

- Linear iterative solvers are subclassed from `lduSolver`. Normal code users and developers don't touch this.



# Numerical Considerations

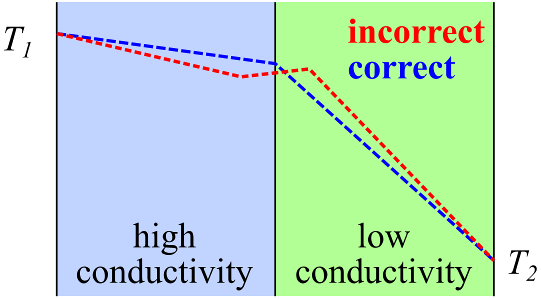
OpenFOAM usage can appear simple at first, but …

- Numerical aspects should always be considered when solving new equation systems

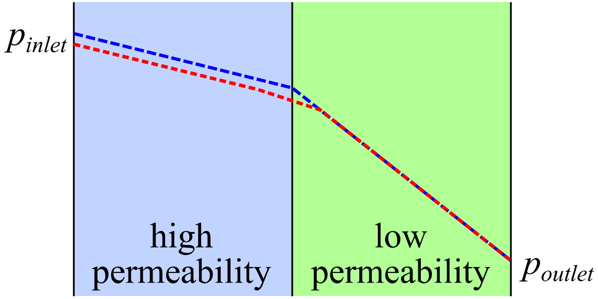
**Example:** Applying FV method to $\nabla \cdot \Gamma\nabla\phi$ gives you $\sum_f (\Gamma\nabla\phi)_f \cdot \vec{S}_f$. Which of these is correct?

```
fvc::div(fvc::interpolate(gamma) * fvc::snGrad(phi) * mesh.magSf())
fvc::div(fvc::interpolate(gamma * fvc::grad(phi)) & mesh.Sf())
```

- Have a clear understanding of what will end up in your matrices
- Have a clear understanding of what discretisation schemes could/should be used
- Finite-volume methods are generally conservative if the domain is continuous and smooth. When not continuous and smooth, the consistency of spatial terms may be critical



Heat Conduction



Porous Flow Pressure Drop

# Additional Considerations

- OpenFOAM's vector and tensor matrices have scalar coefficients, i.e. vector and tensor components are partially decoupled and added to the source term
  - Some past efforts to address this… despite these efforts, this is still the default in OpenFOAM
  - Can cause problems for, e.g. multi-group diffusion.

- Almost all equation systems will contain some non-linear explicit terms
  - Equation coupling terms
  - Non-constant material properties (e.g. temperature dependent)
  - Mesh non-orthogonal and skewness corrections

It is virtually impossible to avoid outer (fixed-point) iterations in OpenFOAM

Advanced non-linear acceleration techniques are relatively unexplored in OpenFOAM. OpenFOAM developers prefer the simplicity of linear solvers in combination with fixed-point iteration.

# Wrap Up - Basic Classes in OpenFOAM

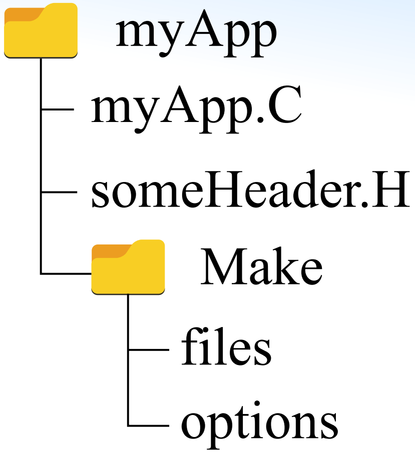
- Time control; `Time`
- Computational domain; `fvMesh`
- Basic types; `scalar`, `vector`, `tensor`, …
- Physical units handling; `dimensionedScalar`, …
- Physical variables; `volFields` and `surfaceFields`
- Boundary conditions; `fvPatchField`
- Matrix systems; `fvScalarMatrix`, `fvVectorMatrix`, `fvTensorMatrix`

# Typical Solver Structure

# Typical Solver Structure

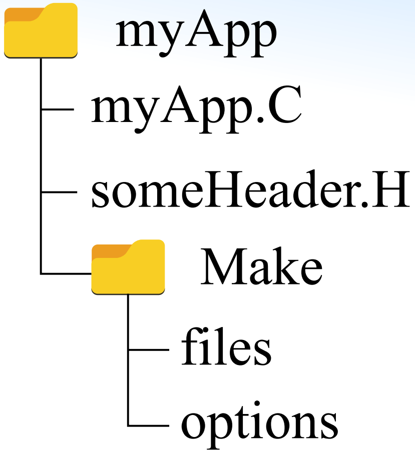
- OpenFOAM solvers have a distinct structure
  - Basic solver layout varies very little from one to the next

- Many well-written tutorials on the OpenFOAM solver structure, compiling your own applications, libraries, new boundary conditions, etc.



# Directory Structure

- wmake build system imposes a fairly simple directory structure
  - – Single directory containing all source code (can be in subdirectories)
  - – 'Make' subdirectory containing
    - files – List of source code files to compile (not headers)
    - options – Compilation options, specifically include directories and libraries to link
  - – Each directory can build a single executable or shared library
- If you want something more complex (e.g. statically link external library, multi-library executable or coupled code), you will have to think carefully on how to proceed.
  - – Easiest is generally to split the complex parts into simpler parts and compile each separately as a shared library.



myApp

myApp.C

someHeader.H

Make

files

options

# scalarTransportFoam

- A basic OpenFOAM solver to model convective and diffusive transport of a passive scalar in a carrier fluid, e.g. boron transport.
- Solves the following equation

$$\frac{\partial T}{\partial t} + \nabla \cdot \vec{\phi} T - \nabla \cdot D \nabla T = S_T$$

where

  - $T$ is a scalar quantity
  - $\vec{\phi}$ is the mass flux of a fluid
  - $D$ is a constant diffusion coefficient
  - $S_T$ is a source term

- In reality, probably not used much due to simplicity; still makes for a good example for more advanced solver development.
- Let's take a look at **selected parts** of the source code

# scalarTransportFoam.C

```
int main(int argc, char *argv[])
{
    #include "setRootCaseLists.H"
    #include "createTime.H"
    #include "createMesh.H"

    simpleControl simple(mesh);

    #include "createFields.H"
```

#include within functions

- Very OpenFOAM-specific
- Only appears in top-level applications
- Helps to avoid redundant clutter

# createFields.H

```
Info<< "Reading field T\n" << endl;

volScalarField T
(
    IOobject
    (
        "T",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);
```

Read/write from/to time-directories

Initial and BCs supplied by user

Write to file by default

# createFields.H

```
Info<< "Reading transportProperties\n" << endl;
IOdictionary transportProperties
(
    IOobject
    (
        "transportProperties",
        runTime.constant(),
        mesh,
        IOobject::MUST_READ_IF_MODIFIED,
        IOobject::NO_WRITE
    )
);
```

Application-specific dictionary (input file) located in constant/transportProperties

Will reread if modified by user

```
Info<< "Reading diffusivity DT\n" << endl;
dimensionedScalar DT
(
    transportProperties.lookup("DT")
);
```

Read constant diffusivity from constant/transportProperties

# scalarTransportFoam.C

```
while (simple.loop(runTime))
{
    Info<< "Time = " << runTime.timeName() << nl << endl;

    fvModels.correct();

    while (simple.correctNonOrthogonal())
    {
        … SOLVE TRANSPORT EQUATION …
    }

    runTime.write();
}

Info<< "End\n" << endl;
```

Time loop

Non-orthogonal corrections

Write (output) fields to file

# scalarTransportFoam.C

```
fvScalarMatrix TEqn
(
    fvm::ddt(T)
  + fvm::div(phi, T)
  - fvm::laplacian(DT, T)
 ==
    fvModels.source(T)
);

TEqn.relax();
fvConstraints.constrain(TEqn);
TEqn.solve();
fvConstraints.constrain(T);
```

User-defined source

Equation underrelaxation

Apply constraints to equation, e.g. fix value in certain region of mesh

Apply constraints to field, e.g. limit value between certain bounds

# Beyond the Basics

# Input/Output

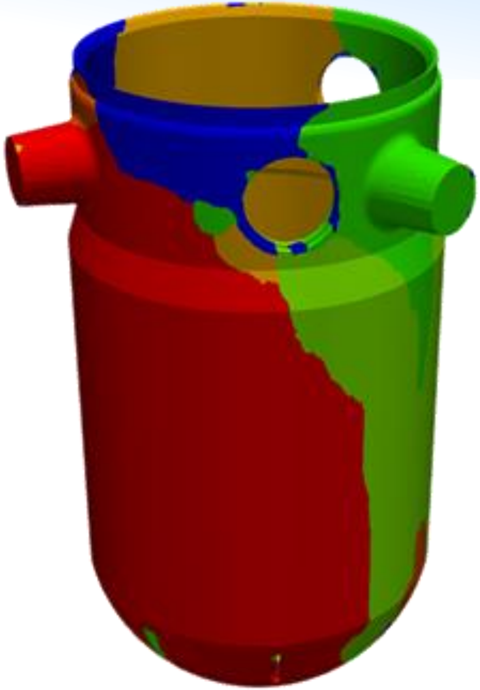
- OpenFOAM has a unique I/O format
  - Fully text-based format (easy to read/modify)
  - Similar to JSON, but designed for CFD, not metadata
- Based on the idea of representing underlying objects in a hierarchy with keywords and values
  - Whitespace is unimportant (less mistakes)
  - Uses human readable keywords (dictionaries are self-documenting)
  - Each level of hierarchy is a `dictionary` containing entries and/or subdictionaries
- Virtually all classes on OpenFOAM have constructors that take a `dictionary` as an argument
  - Objects construct themselves by reading from file
  - Structure of dictionaries often mimics that of objects
  - Even extremely complex objects can be created with just a few lines of code
- Objects that need to be written to file at certain time points are typically subclassed from `IOdictionary`

```
// Example for solver specification
solvers
{
    U
    {
        solver          BiCGStab;
        preconditioner  DILU;
        tolerance       1e-6;
    }
}
```

```
// Example for volField
U
{
    internalField   uniform (0 0 0);

    boundaryField
    {
        inlet
        {
            type    fixedValue;
            value   nonuniform 6 (
                (0 0 1) (0 0 2) (0 0 5)
                (0 0 5) (0 0 5) (0 0 1)
            );
        }
    }
}
```

# Parallelisation

- OpenFOAM parallelisation is based on MPI
  - Domain (mesh) is decomposed into submeshes with processor boundaries (`processorFvPatchField`) connecting them
  - Separate MPI process is spawned for each submesh
- For top-level solvers and applications, developers typically don't need to worry about parallelisation
  - Basic classes of OpenFOAM are parallel-aware
- Lower-level classes are not necessarily parallel aware, i.e. they do their thing without any knowledge of what's happening on other MPI tasks
  - Developers should understand how MPI works to develop new low-level functionality
  - Most important for operations where we collect and evaluate global information



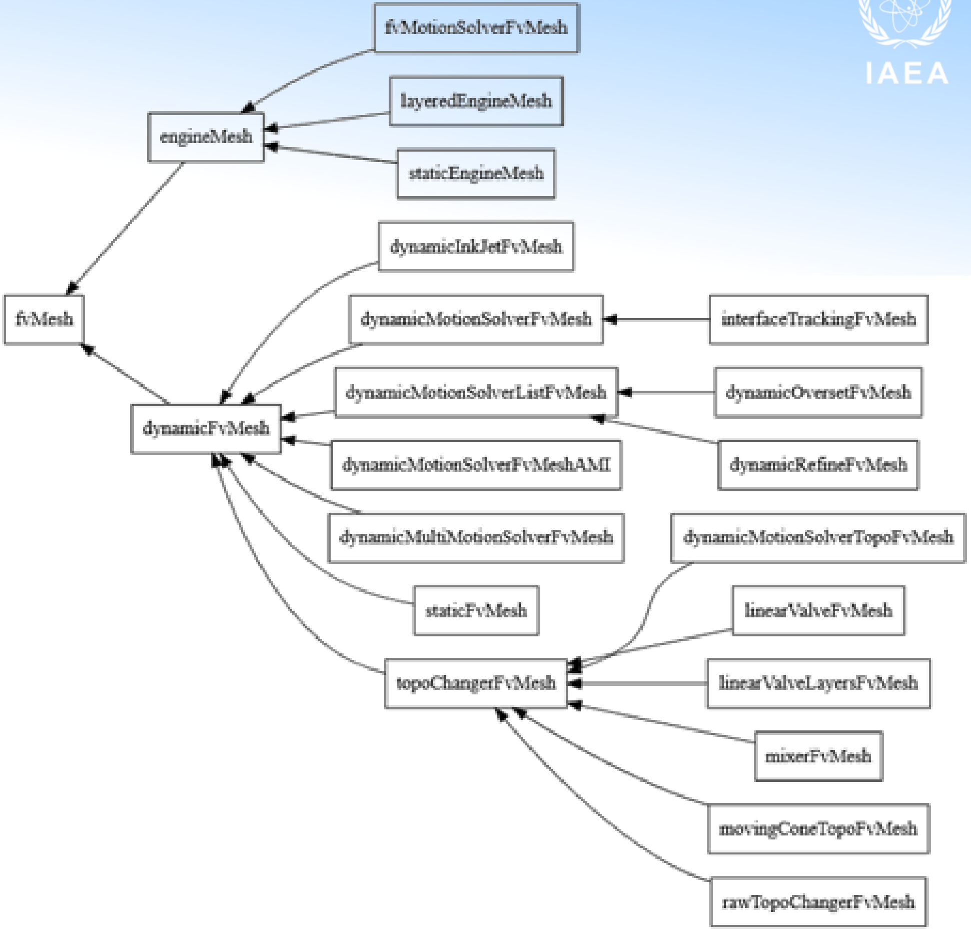
# Parallelisation

- Situations where we need to take care **when not using volFields**
  - Integration of values over the domain, e.g. volume-averaging, surface integration and averaging, finding minima, maxima
  - Working **at a low level** with data exchange between different regions of the mesh, e.g. mesh to mesh mapping or boundary mapping.
- Situations where we are generally safe
  - When using volFields
  - Evaluation of physical properties for **individual cells**

```
const cellZone &fuel =
    mesh.cellZones().findZone("fuel");
const scalarField& V = mesh.V();

// Local integration
scalar Tavg = 0.0;
scalar Vtot = 0.0;

forAll(fuel, i)
{
    const label cellI = fuel[i];
    Tavg += T[cellI]*V[cellI];
    Vtot += V[cellI];
}

// Add contributions from all MPI tasks
reduce(Tavg, sumOp<scalar>());
reduce(Vtot, sumOp<scalar>());

// Calculate final average
Tavg /= Vtot;
```

# Moving Meshes

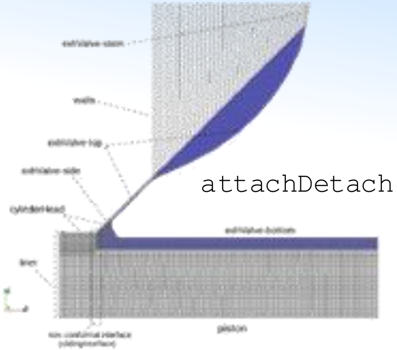
- Mesh motion algorithm and top levels solvers are separated
  - A moving mesh is simply another type of mesh
  - Finite-volume discretisation by default includes necessary terms for moving meshes
  - Any solver can be quickly adapted to support moving meshes
- Mesh movement is handled by class `dynamicFvMesh` or `engineMesh`
  - Many functions available out the box



# Moving Meshes

- For custom applications, you probably want to use `rawTopoChangerFvMesh`
  - User defines a combination of mesh modifiers
    - `attachDetach` – Turn selected faces into boundaries and vice versa. Typically used to isolate two regions, e.g. valve operation
    - `layerAdditionRemoval` – Add cell layers to selected boundaries. Used to extend the domain, e.g. engine piston
    - `slidingInterface` – Slides two boundaries along each another and ensures that the surfaces remain coincident, e.g. rotating components and turbomachinery

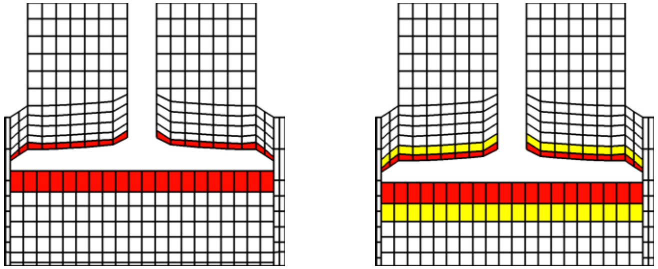
- For anything outside of the existing functionality, significant development work will be needed, e.g. on-the-fly remeshing



`attachDetach`

A. Montofarno et al.,
SAE World Congress 2015

`layerAdditionRemoval`



H. Jasak,
AIAA 2009-341

# Dense Matrix Operations

- Vector and tensor operations already discussed
- OpenFOAM also provides more general Matrix classes
  - `SquareMatrix` – Templated N×N square matrix
  - `RectangularMatrix` – Templated M×N matrix
  - `DiagonalMatrix` – Templated N×N diagonal matrix
  - `SymmetricSquareMatrix` - Templated N×N symmetric square matrix
- Basic matrix algebra supported, e.g. matrix addition, scaling, multiplication and inversion
- More advanced operations also supported, e.g.
  - Solution of a simple matrix system $\mathbf{Ax} = \mathbf{B}$ using `simpleMatrix`
  - QR decomposition using `QRMatrix`
  - Eigendecomposition using `EigenMatrix`
  - Cholesky decomposition using `LLTMatrix`
- Matrix algebra is not parallelised

# ODE Solvers

- OpenFOAM provides the `ODESolver` base class for solving coupled sets of ODEs
  - Solves equation system of form $\frac{d\mathbf{Y}}{dx} = \mathbf{AY}$, where $\mathbf{Y}$ is the solution vector
  - Several runtime-selectable ODE solvers, e.g. Euler implicit, Runge-Kutta, Rosenbrock
- User-defined equation system defined as a class derived from `ODESystem`
  - User provides evaluation function $\mathbf{AY}$ and Jacobian evaluation.

- Not parallelised
  - Most appropriate for solving ODE systems locally in each cell, e.g. chemical reactions, delayed neutron precursor
  - Alternatively, solve the complete system on the master MPI task and distribute results to all tasks, e.g. point kinetics.

# ODE System Example

Point kinetics equation for reactor power with 1 delayed neutron group

$$f_1 = \frac{dn}{dt} = \frac{\rho - \beta}{\Lambda} n + \lambda C$$

$$f_2 = \frac{dC}{dt} = \frac{\beta}{\Lambda} n - \lambda C$$

$$J = \begin{bmatrix} \frac{\partial f_1}{\partial n} & \frac{\partial f_1}{\partial C} \\ \frac{\partial f_2}{\partial n} & \frac{\partial f_2}{\partial C} \end{bmatrix} = \begin{bmatrix} \frac{\rho - \beta}{\Lambda} & \lambda \\ \frac{\beta}{\Lambda} & -\lambda \end{bmatrix}$$

```
class pointKinetics : public ODESystem
{
    ...

    public:

        virtual label nEqns() const
        {
            return 2;
        }

        virtual void derivatives
        (
            const scalar x,
            const scalarField &y,
            scalarField &dydx
        ) const
        {
            dydx[0] = (rho_ - beta_)/Lambda_*y[0]
                      + lambda_*y[1];
            dydx[1] = beta_/Lambda_*y[0]
                      - lambda_*y[1];
        }
```

# ODE System Example

Point kinetics equation for reactor power with 1 delayed neutron group

$$f_1 = \frac{dn}{dt} = \frac{\rho - \beta}{\Lambda} n + \lambda C$$

$$f_2 = \frac{dC}{dt} = \frac{\beta}{\Lambda} n - \lambda C$$

$$J = \begin{bmatrix} \frac{\partial f_1}{\partial n} & \frac{\partial f_1}{\partial C} \\ \frac{\partial f_2}{\partial n} & \frac{\partial f_2}{\partial C} \end{bmatrix} = \begin{bmatrix} \frac{\rho - \beta}{\Lambda} & \lambda \\ \frac{\beta}{\Lambda} & -\lambda \end{bmatrix}$$

```
    virtual void jacobian
    (
        const scalar x,
        const scalarField &y,
        scalarField &dfdx,
        scalarSquareMatrix &dfdy
    ) const
    {
        dfdx[0] = 0.0;
        dfdx[1] = 0.0;

        dfdy[0][0] = (rho_ - beta_)/Lambda_;
        dfdy[0][1] = lambda_;
        dfdy[1][0] = beta_/Lambda_;
        dfdy[1][1] = -lambda_;
    }
} // End of class pointKinetics
```

# Closing Remarks

- OpenFOAM API is extremely large and diverse
  - Basic classes will get you pretty far for simple problems
  - Many examples for other classes of problems
  - Beyond these, the learning curve is very steep and information can be scattered in different places

- OpenFOAM user forums are very active and are regularly visited by key developers of OpenFOAM
  - Don't be surprised if CFD Direct or ESI developers respond directly to your questions

- With a bit of imagination, you'd be amazed by what you can model

# *Thank you!*

*Contact: ONCORE@iaea.org*

Course Enrollment : Multi-physics modelling and simulation of nuclear reactors using OpenFOAM

ONCORE: Open-source Nuclear Codes for Reactor Analysis