High-level programming of OpenFOAM applications, and a first glance at C++
High-level programming of OpenFOAM applications, and a first glance at C++

Prerequisites

• You are familiar with basic usage of OpenFOAM.
• You are familiar with the directory and file structure of OpenFOAM.
• You have some programming experience.
• You are aware of the `wmake` command, and how it picks up the instructions for the compilation.

Learning outcomes

• You will get a suggested way of working with your own developments of applications.
• You will understand the very basic parts of a C++/OpenFOAM code.
• You will understand the `wmake` compilation procedure for applications, and how it is related to compilation with the `g++` compiler and `make`.
• You will step-by-step from scratch implement and understand the purpose of the most general high-level parts of OpenFOAM solvers.
High-level programming of OpenFOAM applications, and a first glance at C++

Contents

• We will start by setting up a code from the very scratch, and make it compile using the OpenFOAM procedures.

• We will discuss the fundamental concepts of CFD, and piece-by-piece implement a code at the very highest level - meaning that we will at this point not know exactly what is happening at the lower levels, where the actual code is found. We do not have enough knowledge at the moment to look at the lower levels. That we will get later.

The instructions are inspired by the codes found in $FOAM_APP/test and of course in $FOAM_APP. See also the Programmers guide, linked to from: https://www.openfoam.com/documentation/.
A VERY first glance at C++, from a high-level OpenFOAM perspective

A C++ code must at least have one function, named `main()`:

```cpp
int main()
{
    return 0;
}
```

The above means that:

- The function takes no arguments (nothing between the brackets)
- The function returns an integer (`int`)  
- The function *definition* is between the curly brackets
- The returned integer is zero (`return 0`).
- The code will not give any output to the terminal window.
A VERY first glance at C++, from a high-level OpenFOAM perspective

Let’s implement and compile this using the OpenFOAM procedures.

• Create a directory for the code, and go there (inspired by the original test directory):

  mkdir -p $WM_PROJECT_USER_DIR/applications/myTests/onlyMainFunction
  cd $WM_PROJECT_USER_DIR/applications/myTests/onlyMainFunction

• Add the code in the previous slide to onlyMainFunction.C (named as directory)

At this point we can actually compile the code by

  g++ onlyMainFunction.C -o Test-onlyMainFunction

Run the code by

  ./Test-onlyMainFunction

Alternatively, put it in $FOAM_USER_APPBIN, as when compiling OpenFOAM applications

  g++ onlyMainFunction.C -o $FOAM_USER_APPBIN/Test-onlyMainFunction

Run the code by

  Test-onlyMainFunction

Remember that the $PATH is used to find the executable, but . is not in the $PATH
A VERY first glance at C++, from a high-level OpenFOAM perspective

Now we will compile it the OpenFOAM way:

- **Clean up**

  ```
  rm Test-onlyMainFunction $FOAM_USER_APPBIN/Test-onlyMainFunction
  ```

- **Create Make/files with**

  ```
  onlyMainFunction.C
  EXE = $(FOAM_USER_APPBIN)/Test-onlyMainFunction
  ```

- **Create an empty Make/options file:**

  ```
  touch Make/options
  ```

- **Compile:**

  ```
  wmake
  ```

- **Run the code by:**

  ```
  Test-onlyMainFunction
  ```
A VERY first glance at C++, from a high-level OpenFOAM perspective

Have a look at the output in the terminal window from the wmake command:

```bash
g++ -std=c++11 -m64 -DOPENFOAM_PLUS=1706 -Dlinux64 -DWM_ARCH_OPTION=64 -DWM_DP -DWM_LABEL_SIZE=32 -Wall -Wextra -Wold-style-cast -Wnon-virtual-dtor -Wno-unused-parameter -Wno-invalid-offsetof -O3 -DNDoRepository -ftemplate-depth-100 -IlInInclude -I. -I/home/oscfd/OpenFOAM/OpenFOAM-plus/src/OpenFOAM/lnInclude -I/home/oscfd/OpenFOAM/OpenFOAM-plus/src/OSspecific/POSIX/lnInclude -fPIC -Xlinker --add-needed -Xlinker --no-as-needed Make/linux64GccDPInt32Opt/onlyMainFunction.o -L/home/oscfd/OpenFOAM/OpenFOAM-plus/platforms/linux64GccDPInt32Opt/lib -lOpenFOAM -ldl -lm -o $FOAM_USER_APPBIN/Test-onlyMainFunction
```

The difference from when we compiled with `g++` directly is that the `wmake` command added a lot of flags. They are in fact not needed for this code, but they also don’t hurt.

From where did the flags come?
Solving PDEs with OpenFOAM

Let’s start building our code by looking at the smallest elements of CFD.

- In CFD we solve partial differential equations (PDEs)
- The PDEs we wish to solve involve derivatives of tensor fields with respect to time and space
- The PDEs must be discretized in time and space before we solve them

- We will start by having a look at algebra of tensors in OpenFOAM at a single point
- We will then have a look at how to generate tensor fields from tensors, and how to relate those fields to a mesh
- Finally we will see how to discretize the PDEs and how to set boundary conditions using high-level coding in OpenFOAM

- For further details, see the ProgrammersGuide (found at https://www.openfoam.com/documentation/)
Basic tensor classes in OpenFOAM

- OpenFOAM provides pre-defined classes for tensors of rank 0-3:

<table>
<thead>
<tr>
<th>Rank</th>
<th>Common name</th>
<th>Basic name</th>
<th>Access function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Scalar</td>
<td>scalar</td>
<td>x(), y(), z()</td>
</tr>
<tr>
<td>1</td>
<td>Vector</td>
<td>vector</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Tensor</td>
<td>tensor</td>
<td>xx(), xy(), xz(), ...</td>
</tr>
</tbody>
</table>

A tensor \( T = \begin{bmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \\ 31 & 32 & 33 \end{bmatrix} \) is defined line-by-line in OpenFOAM:

tensor T( 11, 12, 13, 21, 22, 23, 31, 32, 33);

However, we can’t just put this line in the main() function definition, since C++ does not know what a tensor is. You can try if you like!

We need to declare the tensor class and link to its definition, which is in the shared object file of the library to which the tensor class belongs.
Basic tensor classes in OpenFOAM

We can find the implementation of the tensor class by

find $FOAM_SRC -name tensor

which gives

$FOAM_SRC/OpenFOAM/primitives/Tensor/tensor

We see that the tensor class belongs to the OpenFOAM library, since it is in the OpenFOAM directory we find the Make directory - showing that it is the top level of that library. In the OpenFOAM directory we also find the lnInclude directory, in which there is a link to

$FOAM_SRC/OpenFOAM/primitives/Tensor/tensor/tensor.H

For all declaration files in OpenFOAM you find both a link in an lnInclude directory and the file itself: find $FOAM_SRC -name tensor.H yields

$FOAM_SRC/OpenFOAM/lnInclude/tensor.H
$FOAM_SRC/OpenFOAM/primitives/Tensor/tensor/tensor.H

That file contains the declaration of the class, and we need to include that file at the header of our code. We also have to link to the OpenFOAM library, where the compiled definition of the class can be found.

Let’s implement this...
mkdir -p $WM_PROJECT_USER_DIR/applications/myTests/myTensor
cd $WM_PROJECT_USER_DIR/applications/myTests/myTensor

Add the following code to myTensor.C

#include "tensor.H"
using namespace Foam;
int main()
{
tensor T( 11, 12, 13, 21, 22, 23, 31, 32, 33);
Info << "T: " << T << endl;
Info << "Txz: " << T.xz() << endl;
return 0;
}

• The first line includes the declaration of the tensor class, so that it can be used below.

• The second line says that we should use the namespace Foam, in which everything in OpenFOAM is implemented. We will get back to this later.

• The OpenFOAM Info output stream allows us to write to the terminal window, which we will get back to later. The second Info line uses a call to the access function xz(), which belongs to the tensor class. This is how you typically call functions of a class.
Basic tensor classes in OpenFOAM

Now we need to set up the Make directory so that wmake finds the tensor.H file and links to the OpenFOAM library.

Create Make/files with

myTensor.C
EXE = $(FOAM_USER_APPBIN)/Test-myTensor

Create an empty Make/options file:

touch Make/options

Compile:

wmake

Run the code by:

Test-myTensor

MAGIC? How does it find tensor.H and the OpenFOAM library? Have a look again at the default wmake flags, showing that the compilation is now done in two steps. The first step is for compilation of the intermediate object file, and the second step is for the linking. In the first step the flag -I/home/oscfd/OpenFOAM/OpenFOAM-plus/src/OpenFOAM/lnInclude tells wmake where to find tensor.H. In the second step the flag -L$WM_PROJECT_DIR/platforms/linux64GccDPInt32Opt/lib tells wmake where the libraries can be found, and the flag -lOpenFOAM tells wmake to link to that library. NO MAGIC!
Algebraic tensor operations in OpenFOAM

- Tensor operations operate on the entire tensor entity instead of a series of operations on its components
- The OpenFOAM syntax closely mimics the syntax used in written mathematics, using descriptive functions or symbolic operators

<table>
<thead>
<tr>
<th>Operation</th>
<th>Comment</th>
<th>Mathematical description</th>
<th>Description in OpenFOAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td></td>
<td>$\mathbf{a} + \mathbf{b}$</td>
<td>$\mathbf{a} + \mathbf{b}$</td>
</tr>
<tr>
<td>Outer product</td>
<td>Rank $\mathbf{a}, \mathbf{b} \geq 1$</td>
<td>$\mathbf{a} \mathbf{b}$</td>
<td>$\mathbf{a} \times \mathbf{b}$</td>
</tr>
<tr>
<td>Inner product</td>
<td>Rank $\mathbf{a}, \mathbf{b} \geq 1$</td>
<td>$\mathbf{a} \cdot \mathbf{b}$</td>
<td>$\mathbf{a} \cdot \mathbf{b}$</td>
</tr>
<tr>
<td>Cross product</td>
<td>Rank $\mathbf{a}, \mathbf{b} = 1$</td>
<td>$\mathbf{a} \times \mathbf{b}$</td>
<td>$\mathbf{a} \times \mathbf{b}$</td>
</tr>
</tbody>
</table>

**Examples:**

**Operations exclusive to tensors of rank 2**

- Transpose
  
  $\mathbf{T}'$

  T.T()

- Determinant

  $\text{det}\mathbf{T}$

  det(T)

**Operations exclusive to scalars**

- Positive (boolean)

  $s \geq 0$

  pos(s)

- Hyperbolic arc sine

  $\text{asinh} s$

  asinh(s)

Find more examples in the Programmer’s guide (linked to at https://www.openfoam.com/documentation/)
Algebraic tensor operations in OpenFOAM

Add the following to our myTensor.C file (before return), compile and run:

```cpp
// Line to remove copy-paste problem
tensor t1(1, 2, 3, 4, 5, 6, 7, 8, 9);
tensor t2(1, 2, 3, 1, 2, 3, 1, 2, 3);
tensor t3 = t1 + t2;
Info<< t3 << endl;
tensor t4(3,-2,1,-2,2,0,1, 0, 4);
Info<< inv(t4) << endl;
Info<< (inv(t4) & t4) << endl;
Info<< t1.x() << t1.y() << t1.z() << endl;
Info<< t1.T() << endl;
Info<< det(t1) << endl;
scalar s1(0.75);
Info<< pos(s1) << endl;
Info<< Foam::asinh(s1) << endl;
```

We do not have to add an include line for scalar, since that is done through the tensor.H file (through Tensor.H -> Vector.H -> VectorSpace.H).

The Foam:: at the final line is because the compiler complained that the function is ambiguous, i.e. it exists in more than one namespace. Therefore we explicitly say that we want to use the function that belongs to namespace Foam. It has to be done without using namespace Foam.
A quick note on the tensor class

This slide is actually a bit early since it requires a bit more knowledge, but you can see it as a teaser... It shows how you can learn what a tensor is and how you can learn more on its usage.

- In `tensor.H`, `Tensor.H` is included (located in `$FOAM_SRC/OpenFOAM/primitives/Tensor`), which defines the access functions and includes `TensorI.H`, which defines the tensor operations. The capital `T` means that it is a template class.

- See also `vector`, `symmTensorField`, `sphericalTensorField` and many other examples.
Dimensional units in OpenFOAM

- OpenFOAM checks the dimensional consistency

**Declaration of a tensor with dimensions:**

```
dimensionedTensor sigma
(
   "sigma",
   dimensionSet( 1, -1, -2, 0, 0, 0, 0),
   tensor( 1e6, 0, 0, 0, 1e6, 0, 0, 0, 1e6)
);
```

The values of `dimensionSet` correspond to the powers of each SI unit:

<table>
<thead>
<tr>
<th>No.</th>
<th>Property</th>
<th>Unit</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Mass</td>
<td>kilogram</td>
<td>kg</td>
</tr>
<tr>
<td>2</td>
<td>Length</td>
<td>metre</td>
<td>m</td>
</tr>
<tr>
<td>3</td>
<td>Time</td>
<td>second</td>
<td>s</td>
</tr>
<tr>
<td>4</td>
<td>Temperature</td>
<td>Kelvin</td>
<td>K</td>
</tr>
<tr>
<td>5</td>
<td>Quantity</td>
<td>moles</td>
<td>mol</td>
</tr>
<tr>
<td>6</td>
<td>Current</td>
<td>ampere</td>
<td>A</td>
</tr>
<tr>
<td>7</td>
<td>Luminous intensity</td>
<td>candela</td>
<td>cd</td>
</tr>
</tbody>
</table>

`sigma` then has the dimension \([kg/ms^2]\)
Dimensional units in OpenFOAM

- Add the following to myTensor.C:
  Before main():
  #include "dimensionedTensor.H"
  Before return(0):

    dimensionedTensor sigma
    (
      "sigma",
      dimensionSet( 1, -1, -2, 0, 0, 0, 0),
      tensor( 1e6, 0, 0, 0, 1e6, 0, 0, 0, 1e6)
    );
    Info<< "Sigma: " << sigma << endl;

- Compile, run again, and you will get:

  Sigma: sigma [1 -1 -2 0 0 0 0] (1e+06 0 0 0 1e+06 0 0 0 1e+06)

  You see that the object sigma belongs to the dimensionedTensor class that contains both the name, the dimensions and values.

- See $FOAM_SRC/OpenFOAM/dimensionedTypes/dimensionedTensor
Dimensional units in OpenFOAM

- Try some member functions of the `dimensionedTensor` class:

  ```cpp
  Info<< "Sigma name: " << sigma.name() << endl;
  Info<< "Sigma dimensions: " << sigma.dimensions() << endl;
  Info<< "Sigma value: " << sigma.value() << endl;
  ```

  You now also get:

  Sigma name: sigma
  Sigma dimensions: [1 -1 -2 0 0 0 0]
  Sigma value: (1e+06 0 0 0 1e+06 0 0 0 1e+06)

- Extract one of the values:

  ```cpp
  Info<< "Sigma yy value: " << sigma.value().yy() << endl;
  ```

  Note here that the `value()` member function first converts the expression to a tensor, which has a `yy()` member function. The `dimensionedTensor` class does not have a `yy()` member function, so it is not possible to do `sigma.yy()`.
Construction of a tensor field in OpenFOAM

- A tensor field is a list of tensors
- The use of typedef in OpenFOAM yields readable type definitions: scalarField, vectorField, tensorField, symmTensorField, ...

- Algebraic operations can be performed between different fields, and between a field and a single tensor, e.g. Field U, scalar 2.0: U = 2.0 * U;

- Add the following to myTensor.C:
  Before main():
  #include "tensorField.H"
  Before return(0):

  tensorField tf1(2, tensor::one);
  Info<< "tf1: " << tf1 << endl;
  tf1[0] = tensor(1, 2, 3, 4, 5, 6, 7, 8, 9);
  Info<< "tf1: " << tf1 << endl;
  Info<< "2.0*tf1: " << 2.0*tf1 << endl;

- However, this kind of tensor field is not related to any mesh, and can therefore not be discretized.
#include "fvCFD.H"

- OpenFOAM uses the finite volume method (fvm) to discretize the PDEs, and there are many classes in OpenFOAM that are related to fvm.

- OpenFOAM provides the header file `fvCFD.H` that only includes other header files related to fvm, including the tensor classes we have discussed. It can therefore be used to reduce the number of header files. It in fact also ends with `using namespace Foam`.

- Exchange all the lines before the `main()` function with:

  ```cpp
  #include "fvCFD.H"
  ```

- That file is located in a sub-directory of `$FOAM_SRC/finiteVolume`, so add the following to `Make/options` (Note that we do not have to link. Why?):

  ```
  EXE_INC = \n    -I$(LIB_SRC)/finiteVolume/lnInclude \n    -I$(LIB_SRC)/meshTools/lnInclude
  ```

  The `meshTools` line is needed for a `cyclicAMILduInterface.H` file that is included through `fvCFD.H`, and obviously includes something from the `meshTools` library.

- Compile and test!
Discretization of a tensor field in OpenFOAM

- OpenFOAM uses the finite volume method (fvm) to discretize the PDEs of the tensor fields on a mesh.
- The tensor fields must thus be related to a mesh.
- The `polyMesh` class can be used to construct a polyhedral mesh using the minimum information required. I.e. that in `<case>/constant/polyMesh`
- The `fvMesh` class extends the `polyMesh` class to include additional data needed for the fvm discretization (see `test/mesh`)
- The `geometricField` class relates a tensor field to an `fvMesh` (can also be typedef `volField`, `surfaceField`, `pointField`)
- A `geometricField` inherits all the tensor algebra of its corresponding field, has dimension checking, and can be subjected to specific discretization procedures

We will now investigate a `polyMesh`, an `fvMesh`, and a `geometricField`, but for that we need a base code...
Base code

- For the base code we are inspired both by \$FOAM_APP/test/mesh/Test-mesh.C and \$FOAM_SOLVERS/incompressible/icoFoam/icoFoam.C

- We need a code that starts with (explanations coming later):

```cpp
#include "setRootCase.H"
#include "createTime.H"

where the <header files> can preferrably be fvCFD.H.

- We use a special functionality to generate a template application:

```bash
cd $WM_PROJECT_USER_DIR/applications/myTests
foamNewApp meshAndField
cd meshAndField
wmake
meshAndField
```

It complains that it can’t find a system/controlDict. We will see why soon.
Base code

The base code (meshAndField.C) contains a commented header, and:

```c
#include "fvCFD.H"

// ******************************************

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

    // ******************************************

    Info<< nl;
    runTime.printExecutionTime(Info);

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

    return 0;
}

Some explanations in the coming slides...
Base code

- `int argc, char *argv[]`
  The arguments to the main function are the number of flags and the flags supplied when running the code.

- `setRootCase.H (find it with find $FOAM_SRC -name setRootCase.H)`
  Not a header file, just a piece of code:
  ```cpp
  Foam::argList args(argc, argv);
  if (!args.checkRootCase())
  {
    Foam::FatalError.exit();
  }
  ```
  This constructs the object `args`, and uses it to check that we are running the code in a case directory. It simply checks if there is an appropriate `system/controlDict()`.

- `createTime.H (find it with find $FOAM_SRC -name createTime.H)`
  Not a header file, just a piece of code:
  ```cpp
  Foam::Info<< "Create time\n" << Foam::endl;
  Foam::Time runTime(Foam::Time::controlDictName, args);
  ```
  This writes some text and constructs the `runTime` object of the class `Time`. 
Base case

We need a case to run our code (here the cavity case with only four cells):

cp -r $FOAM_TUTORIALS/incompressible/icoFoam/cavity/cavity $FOAM_RUN/cavityFourCells
esed -i s/"20 20 1"/"2 2 1"/g $FOAM_RUN/cavityFourCells/system/blockMeshDict
blockMesh -case $FOAM_RUN/cavityFourCells/meshAndField -case $FOAM_RUN/cavityFourCells

Now we can move on to examine a polyMesh and an fvMesh, following by a geometricField, in the form of a volScalarField...
Examine a polyMesh

Add after the line with `#include "createTime.H" (from test/mesh):

```cpp
Info<< "Create mesh" << endl;
polyMesh mesh
(
    IOobject
    (
        fvMesh::defaultRegion,
        runTime.timeName(),
        runTime,
        IOobject::MUST_READ
    ),
);

Info<< "Cell centres" << nl << mesh.cellCentres() << endl;
Info<< "Cell volumes" << nl << mesh.cellVolumes() << endl;
Info<< "Cell face centres" << nl << mesh.faceCentres() << endl;

Compile and run.
Some descriptions follow...
Examine a polyMesh

- **polyMesh mesh ( ... )**
  An object named `mesh` is constructed from the `polyMesh` class. There is only one argument to the constructor, which is an `IOobject`. The `IOobject` itself takes four arguments that are not obvious. We can make the educated guess that it reads the mesh from `constant/polyMesh`. At some point we have to stop figuring out exactly how things are done. Now we are fine relying on how OpenFOAM reads from files.

- **cellCentres()**
  gives the center of all cells and boundary faces.

- **cellVolumes()**
  gives the volume of all the cells.

- **faceCentres()**
  gives the center of all the faces.

A detailed note is that the above functions are actually inherited from the `primitiveMesh` class.

However, a `polyMesh` only has the very basic information of the mesh...
Examine an fvMesh

An fvMesh builds on top of a polyMesh, and has additional attributes.

- Let us examine an fvMesh:

  ![Image](image.png)

  Start by changing the mesh class from polyMesh to fvMesh in meshAndField.C

  ```
  sed -i s/'polyMesh mesh'/'fvMesh mesh'/g meshAndField.C
  ```

- Compile and run.

- Note that we are using the same member functions as before. They still belong to the polyMesh class. However, since the fvMesh class inherits from the polyMesh class they also belong to the fvMesh class. We will discuss inheritance more later.

- Add the following after our previous insertion in meshAndField.C, compile and run again:

  ```
  Info<< mesh.C() << endl;
  Info<< mesh.V() << endl;
  Info<< mesh.Cf() << endl;
  ```

  This could not be done with the polyMesh, since those member functions are defined in the sub-class fvMesh. They give the corresponding information as before, but with additional information.
Examine an fvMesh

The construction of thefvMesh is done the same way for all solvers that use a static mesh. Instead of replicating the call for the fvMesh constructor in all the solvers they just have after
#include "createTime.H" the line (see e.g. icoFoam.C)

#include "createMesh.H"

This is as well not a real header file, but just inserts the code in:
find $FOAM_SRC -name createMesh.H
Clean up your code by doing this, and check that it still works!
Examine a volScalarField

Now we are at a point where we can start implementing a particular PDE solver. Depending on the problem at hand we want to solve equations for different fields. I.e. the fields to be constructed depends on the solver. That is why all the solvers include a local file named createFields.H. See e.g. the original icoFoam directory:

createFields.H  icoFoam.C  Make

Inside the main(...) function of icoFoam.C we find:

    #include "createFields.H"

This refers to the local file in the same directory, which is found by the compiler thanks to the compiler flag -I.

Add such a line below #include "createMesh.H", create the file:

touch createFields.H

and we will continue constructing a field in that file.
Examine a volScalarField

We are here inspired by icoFoam/createFields.H, and construct a volScalarField (which is typedef of a geometricField, and therefore relates to a mesh).

Add in createFields.H:

```cpp
volScalarField p
{
    IOobject
    {
        "p",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    },
    mesh
};
Info<< p << endl;
Info<< p.boundaryField()[0] << endl;
```

Compile, run and have a look at the output! Note that the above Info statement is in createFields.H, which is in the beginning of the execution.

Although we are not interested in all the details here we will still have a quick look...
Examine a volScalarField

```cpp
cfScalarField p( IOobject (...), mesh );
```

The object `p` is constructed as a `volScalarField`, and the constructor takes two arguments.

- `IOobject ("p", runTime.timeName(), mesh, IOobject::MUST_READ, IOobject::AUTO_WRITE)`
  - The first argument is an internal name `p`, so that we at the end can ask our object for its name (i.e. the internal name should be the same as the object name).
  - The second argument uses the `runTime` object to determine which time directory to read from (according to settings in `system/controlDict`).
  - The third argument states that the `volScalarField` should be related to the `fvMesh` corresponding to our object `mesh`.
  - The fourth argument states that it must be read.
  - The fifth argument states that the field should be written in the coming time directories according to the settings in `system/controlDict`.

- `mesh`
  If the `volScalarField` is read from a file, as in this case, you just specify `mesh` here. I haven’t been interested in checking the details of this, but I guess it is used to specify the size and structure of the field. If the `volScalarField` is NOT read from a file the second argument can be an existing `volScalarField` that corresponds to the same mesh.
Implement a steady-state thermal conduction solver

Let’s start from scratch and implement a steady-state thermal conduction solver:

cd $WM_PROJECT_USER_DIR/applications/myTestsfoamNewApp
foamNewApp myThermalConductionSolver
cd myThermalConductionSolver
wmake

Add after #include "createTime.H":

    #include "createMesh.H"
    #include "createFields.H"

Add in createFields.H:

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

This is the point where we were before. Now we need to specify and discretize the equation...
Equation specification and discretization in OpenFOAM

- We need to convert a PDE into a linear equation system, $\mathbf{A}\mathbf{x} = \mathbf{b}$. Here $\mathbf{x}$ and $\mathbf{b}$ are volFields (geometricFields) and $\mathbf{A}$ is an fvMatrix that is created by a discretization of a geometricField on a mesh according to the PDE and the discretization schemes used for each term in the PDE.

- The fvm (Finite Volume Method) and fvc (Finite Volume Calculus) classes contain static functions for the differential operators, and discretize any geometricField. fvm returns an fvMatrix, and fvc returns a geometricField (see $\$FOAM\_SRC/finiteVolume/finiteVolume/fvc$ and fvm).

Examples (see more in Programmer’s guide):

<table>
<thead>
<tr>
<th>Term description</th>
<th>Mathematical expression</th>
<th>fvm::/fvc:: functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Laplacian</td>
<td>$\nabla \cdot \Gamma \nabla \phi$</td>
<td>laplacian(Gamma,phi)</td>
</tr>
<tr>
<td>Time derivative</td>
<td>$\partial \phi / \partial t$</td>
<td>ddt(phi)</td>
</tr>
<tr>
<td></td>
<td>$\partial \rho \phi / \partial t$</td>
<td>ddt(rho, phi)</td>
</tr>
<tr>
<td>Convection</td>
<td>$\nabla \cdot (\psi)$</td>
<td>div(psi, scheme)</td>
</tr>
<tr>
<td></td>
<td>$\nabla \cdot (\psi \phi)$</td>
<td>div(psi, phi, word)</td>
</tr>
<tr>
<td></td>
<td>$\phi$</td>
<td>div(psi, phi)</td>
</tr>
<tr>
<td>Source</td>
<td>$\rho \phi$</td>
<td>Sp(rho, phi)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SuSp(rho, phi)</td>
</tr>
</tbody>
</table>

$\phi$: vol<type>Field, $\rho$: scalar, volScalarField, $\psi$: surfaceScalarField
A familiar example

A call for solving the equation

\[
\frac{\partial \rho \vec{U}}{\partial t} + \nabla \cdot \phi \vec{U} - \nabla \cdot \mu \nabla \vec{U} = -\nabla p
\]

has the OpenFOAM representation

```
solve
(
    fvm::ddt(rho, U) + fvm::div(phi, U) - fvm::laplacian(mu, U) == -fvc::grad(p)
)
```

In this case all terms except the pressure gradient contribute to the coefficient matrix. The pressure gradient thus ends up as an explicit source term.

The convecting velocity is treated using the flux \( \phi \), and there is a viscosity \( \mu \) defined somewhere else. We will get back to that later.
Implement a steady-state thermal conduction solver

In our steady-state thermal conduction solver we want to solve the equation

\[ \nabla \cdot k \nabla T = 0 \]

We thus add in our code (after `#include "createFields.H"`):

```cpp
solve( fvm::laplacian(k, T) );
runTime++;
runTime.write();
```

We see that the right hand side of the equation is omitted. For OpenFOAM this means that it is zero.

```cpp
runTime++ increases the time by the value of deltaT specified in controlDict, so that we do not overwrite the T file in the startTime directory.
runTime.write() tells the code to write out all the fields that are specified with IOobject::AUTO_WRITE, which is the case for our T field.
This means that in our case we must make sure that the fields are written at time startTime+deltaT
```

We need to specify the thermal conductivity \(k\)...
Read thermal conductivity from dictionary

We could hard-code the thermal conductivity in `createFields.H` as (remember how we did this for a tensor before):

```c++
dimensionedScalar k
(
    "k",
    dimensionSet( 0, 2, -1, 0, 0, 0, 0),
    scalar(4e-05)
);
```

However, we would probably prefer that the value can be modified when we run the case.

Have a look at how the kinematic viscosity is read from a dictionary in `createFields.H` of the `laplacianFoam` solver, and copy-paste from the next two slides into our `createFields.H` file.
Read thermal conductivity from dictionary

Copy-paste to end of createFields.H:

```cpp
IOdictionary transportProperties
{
    IOobject
    (
        "transportProperties",
        runTime.constant(),
        mesh,
        IOobject::MUST_READ_IF_MODIFIED,
        IOobject::NO_WRITE
    )
};
```

This means that the file named transportProperties in the constant directory will be read (and read again if it is modified) into an object named transportProperties of the class IOdictionary. At this point the contents of that dictionary file is just kept in the object transportProperties.
Read thermal conductivity from dictionary

Copy-paste to end of `createFields.H`:

```cpp
dimensionedScalar k
(
    "k",
    dimArea/dimTime,
    transportProperties
);
```

This is similar to the hard-coded way of doing it, as discussed before, but:

- The dimension is set using the pre-defined `dimensionSets` defined in
  `$FOAM_SRC/OpenFOAM/dimensionSet/dimensionSets.C`

- The `transportProperties` object is used to set the value. It should be noted here that
  if the `constant/transportProperties` file changes, the `transportProperties` object
  changes, and thus also the value of the `k` object changes.

- We need to provide a `constant/transportProperties` file with a `k` entry in our case.

Compile using `wmake`, and proceed to set up a test case...
A test case for myThermalConductionSolver

Copy-paste to the terminal window (and understand the purpose of each line):

```bash
pushd $FOAM_RUN
cp -r $FOAM_TUTORIALS/incompressible/icoFoam/cavity/cavity thermalSquare
cd thermalSquare
mv 0/U 0/T; rm 0/p
sed -i s/volVectorField/volScalarField/g 0/T
sed -i s/U/T/g 0/T
sed -i s"1 -1 0"/"0 0 1"/g 0/T
sed -i s"(0 0 0)"/0/g 0/T
sed -i s"(1 0 0)"/1/g 0/T
sed -i s"noSlip;"/"fixedValue; value uniform 0;"/g 0/T
sed -i s/icoFoam/myThermalConductionSolver/g system/controlDict
sed -i s"0.005"/1/g system/controlDict
sed -i s"20"/1/g system/controlDict
sed -i s/Euler/steadyState/g system/fvSchemes
sed -i s/U/T/g system/fvSolution
sed -i s/nu/k/g constant/transportProperties
sed -i s"0.01"/"4e-05"/g constant/transportProperties
```

Run blockMesh, myThermalConductionSolver and check in paraFoam.

**Now, spend some time to clean up the case for this specific solver, not to fool any future user with settings that are not affecting the solver! Tell me when you’re done!**
Add source terms

Go back to the code using popd
Let us add a linearized source term \((S(x) = S_u + S_p x)\). Add to `createFields.H`:

dimensionedScalar su
(
   "su",
   dimTemperature/dimTime,
   transportProperties
);

dimensionedScalar sp
(
   "sp",
   pow(dimTime,-1),
   transportProperties
);

Info << "k: " << k << endl;
Info << "su: " << su << endl;
Info << "sp: " << sp << endl;
Add source terms

Change in `myThermalConductionSolver.C`:

```c
solve( fvm::laplacian(k, T) + su + fvm::Sp(sp, T) );
```

Compile with `wmake`.
Go to the case with `pushd $FOAM_RUN/thermalSquare`

Add to `constant/transportProperties`:

```c
// Line to remove copy-paste problem
su 0.02;
sp 0.03;
```

Run the case and investigate the result.

Later we can have a look at the code to figure out how the source terms are treated exactly. Now we simply see that `su` is a `dimensionedScalar`. It means that it must be expanded and treated as a field covering the entire computational domain. It will be added to the source term, `b`, of the linear system `Ax=b`. The `sp` contribution is implemented using the `fvm` namespace, which tells us that it will contribute to the coefficient matrix, `A`, rather than the source term, `b`.

Let’s play with this...
Add source terms

It is indeed possible to add the source terms as:

\[
\text{solve( fvm::laplacian(k, T) + su + sp*T );}
\]

If you do that you see that the results change drastically, and the number of iterations is greatly reduced. Why?

You get exactly the same effect if you add it like:

\[
\text{solve( fvm::laplacian(k, T) + su + fvc::Sp(sp, T) );}
\]

Try changing your code to:

```cpp
for (int i=0; i<10; i++)
{
    solve( fvm::laplacian(k, T) + su + fvc::Sp(sp, T) );
    runTime++;
    runTime.write();
}
```

In the final time directory we have good results!

Have a look at the log file...
Add source terms

The log file:

smoothSolver: Solving for T, Initial residual = 1, Final residual = 9.94501e-06, No Iterations 197
smoothSolver: Solving for T, Initial residual = 0.0211164, Final residual = 9.54079e-06, No Iterations 153
smoothSolver: Solving for T, Initial residual = 0.00557356, Final residual = 9.58851e-06, No Iterations 130
smoothSolver: Solving for T, Initial residual = 0.00196027, Final residual = 9.64058e-06, No Iterations 109
smoothSolver: Solving for T, Initial residual = 0.000734073, Final residual = 9.60551e-06, No Iterations 89
smoothSolver: Solving for T, Initial residual = 0.000283075, Final residual = 9.81723e-06, No Iterations 69
smoothSolver: Solving for T, Initial residual = 0.000113519, Final residual = 9.93168e-06, No Iterations 50
smoothSolver: Solving for T, Initial residual = 4.9317e-05, Final residual = 9.87474e-06, No Iterations 33
smoothSolver: Solving for T, Initial residual = 2.48817e-05, Final residual = 9.85407e-06, No Iterations 19
smoothSolver: Solving for T, Initial residual = 1.55733e-05, Final residual = 9.56393e-06, No Iterations 10

We see that the Initial residual jumps up a lot from the previous Final residual, and is decreasing every time we solve the equation.

The reason is that with this way of writing the source term is given explicitly, and the temperature field of the source term is considered constant each time we solve the equation. We therefore need to iterate to get the correct solution. This is not efficient, and should be avoided if possible.

Change fvc to fvm, and you see that the linear solver will only iterate the first time, i.e. we reach the correct solution directly. Then the sp part of the source term is treated implicitly, as it should.
Add source terms using fvOptions

Just a note to say that source terms can be added using `fvOptions`, for the solvers that have that functionality included. This is similar to User Defined Functions in Fluent for example.

See:

$FOAM_SRC/fvOptions/sources

We are not covering that now.
Add a time term

A next step is to add a time term. Instead of doing that ourselves, we have a look at the existing code `laplacianFoam`...
A tutorial example: laplacianFoam, the source code

Solves $\frac{\partial T}{\partial t} - \nabla \cdot k \nabla T = 0$ (see `$FOAM_SOLVERS/basic/laplacianFoam/laplacianFoam.C`)

Here omitting the lines corresponding to `fvOptions` (version dependent):

```cpp
#include "fvCFD.H" // Include the class declarations
#include "simpleControl.H" // Prepare to read the SIMPLE sub-dictionary
int main (int argc, char *argv[])
{
    # include "setRootCase.H" // Set the correct path
    # include "createTime.H" // Create the time
    # include "createMesh.H" // Create the mesh
    simpleControl simple(mesh); // Read the SIMPLE sub-dictionary
    # include "createFields.H" // Temperature field T and diffusivity DT
    while (simple.loop()) // SIMPLE loop
    {
        while (simple.correctNonOrthogonal())
        {
            solve( fvm::ddt(T) - fvm::laplacian(DT, T) ); // Solve eq.
        }
    }
    # include "write.H" // Write out results at specified time instances
    return 0; // End with 'ok' signal
}
```
A tutorial example: laplacianFoam, discretization and boundary conditions

See $FOAM_TUTORIALS/basic/laplacianFoam/flange

Discretization:
dictionary fvSchemes, read from file:

ddtSchemes
{
    default Euler;
}

laplacianSchemes
{
    default none;
    laplacian(DT,T) Gauss linear corrected;
}

Boundary conditions:
Part of class volScalarField object T, read from file:

boundaryField{
    patch1{ type zeroGradient;}
    patch2{ type fixedValue; value uniform 273;}}