Implementation of simple solvers

Implementation of simple solvers

Prerequisites

- You are familiar with the directory and file structure of OpenFOAM.
- You have a basic knowledge of how OpenFOAM application implementations are structured, and how objects are used in a top-level code.
- You understand the very basic parts of a C++/OpenFOAM code.
- You understand the `wmake` compilation procedure for applications, and how it is related to compilation with the `g++` compiler and `make`.

Learning outcomes

- You will get a suggested way of working with your own developments of applications.
- You will step-by-step from scratch implement and understand the purpose of the most general high-level parts of OpenFOAM solvers.

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/myTests
foamNewApp myThermalConductionSolver
cd myThermalConductionSolver
wmake
```

Add in `myThermalConductionSolver.C` 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, **Ax=b**. Here **x** and **b** are volFields (geometricFields) and **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) namespaces 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):

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

ϕ: vol<type>Field, ρ: scalar, volScalarField, ψ: 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 ϕ, 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"`):

```
    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.

`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):

```
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`:

```
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`:

```
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):

```
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`:

```
    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`:

```
// 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:

```
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:

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

Try changing your code to:

```
    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 explicitely, 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 implicitely, 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.

Implement a convection-diffusion solver

In this part, we implement a solver for convection-diffusion problem with the governing equation

$$\nabla \cdot (UT) = \nabla \cdot k \nabla T$$

In high-level programming language, this equation is translated to

```
fvm::div(phi, T) - fvm::laplacian(k, T) == 0
```

Note that the second term is exactly the same as the governing equation in our steady-state thermal conduction solver, so we can use this solver as a base for our implementation.

```
cd $WM_PROJECT_USER_DIR/applications/myTests
cp -r myThermalConductionSolver/ myConvectionDiffusionSolver
cd myConvectionDiffusionSolver
mv myThermalConductionSolver.C myConvectionDiffusionSolver.C
sed -i s/myThermalConductionSolver/myConvectionDiffusionSolver/g Make/files
```

We then replace the `for` loop in `myConvectionDiffusionSolver.C` with the governing equation for convection diffusion problem

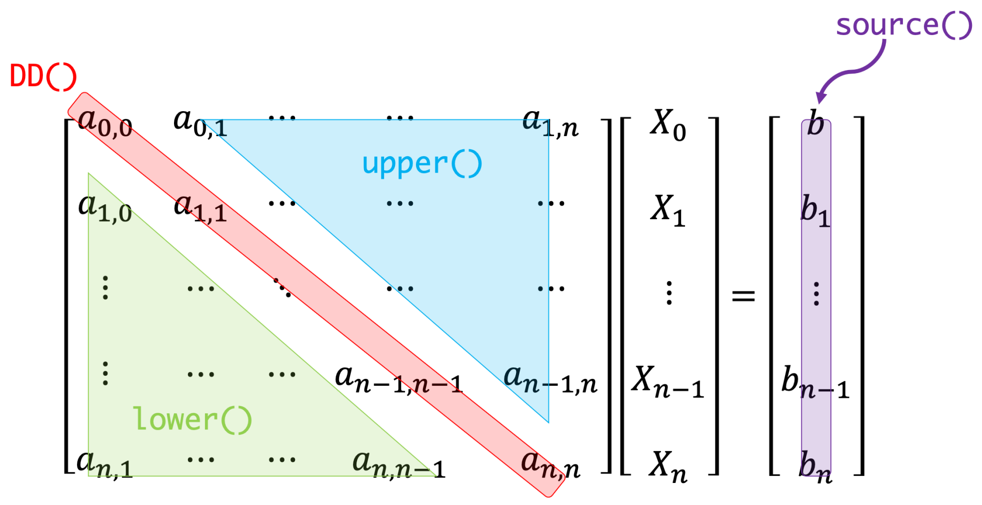
```
fvScalarMatrix TEqn (fvm::div(phi, T)-fvm::laplacian(k, T));

TEqn.solve();
runTime++;
runTime.write();
```

Implement a convection-diffusion solver

We include the `printOutfvMatrixCoeffs.H` file after the line which includes the governing equations.

```
#include "printOutfvMatrixCoeffs.H"
```

The code in this file prints out the coefficients in the linear system **Ax=b** for an internal cell. Let's have a quick look in this file.



$$\begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & \cdots & a_{1,n} \\ a_{1,0} & a_{1,1} & \cdots & \cdots & \cdots \\ \vdots & \cdots & \ddots & \cdots & \cdots \\ \vdots & \cdots & \cdots & a_{n-1,n-1} & a_{n-1,n} \\ a_{n,1} & \cdots & \cdots & a_{n,n-1} & a_{n,n} \end{bmatrix} \begin{bmatrix} X_0 \\ X_1 \\ \vdots \\ X_{n-1} \\ X_n \end{bmatrix} = \begin{bmatrix} b \\ b_1 \\ \vdots \\ b_{n-1} \\ b_n \end{bmatrix}$$

Since we do not have any source term in the governing equation, we remove the respective lines from `createFields.H`

Implement a convection-diffusion solver

As the governing equations includes `phi` which is the flux of `U` at faces, we add the following to `createFields.H`,

```
Info<< "Reading field U\n" << endl;
volVectorField U
(
    IOobject
    (
        "U",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);

#include "createPhi.H"
```

We compile using `wmake`.

Two-dimensional convection-diffusion problem

We need to create a test case for our new solver. We copy an existing tutorial and modify it by copying the following lines in the terminal.

```
// Line to remove copy-paste problem
cd $FOAM_RUN
mkdir 2DConvectionDiffusionCase
cd 2DConvectionDiffusionCase
cp -r $FOAM_TUTORIALS/verificationAndValidation/schemes/divergenceExample/{0.orig,constant,system} .
mv  0.orig 0
sed -i s/"DT.*"/"k 0.0012;"/g constant/transportProperties
sed -i s/"100;"/"1;"/g system/controlDict
sed -i s/"0.005;"/"1;"/g system/controlDict
```

Since the governing equation has a divergence term, we need to specify the discretization scheme for this term. As a first try, we select `linear` scheme.

```
sed -i s/"div(phi,T).*"/"div(phi,T) Gauss linear;"/g system/fvSchemes
```

We then create the mesh using the `blockMesh` and run the solver.

If we check the output of the solver in the terminal, we can see that the solution did not converged.

```
DILUPBiCG:  Solving for T, Initial residual = 1, Final residual = 7.14412e+13, No Iterations 1000
```

We can check the solution using `paraFoam` to see that it looks strange.

Two-dimensional convection-diffusion problem

The reason for divergence of the solution can be explained by examining the sufficient condition for a convergent iterative method. This condition can be expressed as:

$$\frac{\sum |a_{nb}|}{|a_p|} \begin{cases} \leq 1 & \text{at all nodes} \\ < 1 & \text{at one node at least} \end{cases}$$

Here, a_{np} is the off-diagonal coefficients in matrix A and a_p is the diagonal coefficients in this matrix. If a matrix of coefficients satisfies the above condition, we call it diagonally dominant.

Let's check if the above condition is true for the cell for which we printed out the coefficients.

```
diagonal coefficient for a(1275,1275) = 0.00048
source term due to discretization  b(1275) = 0
off-diagonal coefficients:
a(1275,1225) : -0.00112
a(1275,1274) : -0.00112
a(1275,1276) : 0.00088
a(1275,1325) : 0.00088
```

As we can see, for the selected cell, the above stability conditions is not satisfied. That is why the solver does not converge.

$$\frac{\sum |a_{nb}|}{|a_p|} = \frac{0.004}{0.00048} > 1$$

Two-dimensional convection-diffusion problem

There are a couple of ways to make the solution converge. One way is to increase the mesh resolutions. Let's try that.

```
sed -i s/"(50 50 1)"/"(500 500 1)"/g system/blockMeshDict
```

We create the mesh and run the solver. We can see that we have a converged solution now.

```
DILUPBiCG:  Solving for T, Initial residual = 1, Final residual = 7.54955e-11, No Iterations 106
```

We can also see that the coefficients for the selected cell satisfies the sufficient convergence condition.

```
diagonal coefficient for a(1275,1275) = 0.00048
source term due to discretization  b(1275) = 0
off-diagonal coefficients:
a(1275,775) : -0.00022
a(1275,1274) : -0.00022
a(1275,1276) : -2e-05
a(1275,1775) : -2e-05
```

$$\frac{\sum |a_{nb}|}{|a_p|} = \frac{0.00048}{0.00048} \leq 1$$

Two-dimensional convection-diffusion problem

Another way to improve the stability of the solution is to change the divergence scheme to `upwind`. To compare with the convergence of this scheme with `linear` scheme, we use the original coarse mesh.

```
sed -i s/"(500 500 1)"/"(50 50 1)"/g system/blockMeshDict
sed -i s/"div(phi,T).*"/"div(phi,T) Gauss upwind;"/g system/fvSchemes
```

We create the mesh and run the solver. We can see that we have a converged solution

```
DILUPBiCG:  Solving for T, Initial residual = 1, Final residual = 4.81385e-12, No Iterations 15
```

and the coefficients also satisfy the sufficient convergence conditions.

```
diagonal coefficient for a(1275,1275) = 0.00448
source term due to discretization  b(1275) = 0
off-diagonal coefficients:
a(1275,1225) : -0.00212
a(1275,1274) : -0.00212
a(1275,1276) : -0.00012
a(1275,1325) : -0.00012
```

$$\frac{\sum |a_{nb}|}{|a_p|} = \frac{0.00448}{0.00448} \leq 1$$

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 $\partial T/\partial t - \nabla \cdot k \nabla T = 0$ (see `$FOAM_SOLVERS/basic/laplacianFoam/laplacianFoam.C`)
Here omitting the lines corresponding to `fvOptions` (version dependent):

```
#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())
    {   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;}}
```