How to implement your own application

- The applications are located in the `$WM_PROJECT_DIR/applications` directory (equivalent to `$FOAM_APP`. Go there using alias `app`).
- Copy an application that is similar to what you would like to do and modify it for your purposes. In this case we will make our own copy of the `icoFoam` solver and put it in our `$WM_PROJECT_USER_DIR` with the same file structure as in the OpenFOAM installation:

```
cd $WM_PROJECT_DIR
cp -r --parents applications/solvers/incompressible/icoFoam $WM_PROJECT_USER
cd $WM_PROJECT_USER_DIR/applications/solvers/incompressible
mv icoFoam passiveScalarFoam
cd passiveScalarFoam
wclean
mv icoFoam.C passiveScalarFoam.C
```

- Modify `Make/files` to:

```
passiveScalarFoam.C
EXE = $(FOAM_USER_APPBIN)/passiveScalarFoam
```

- Compile with `wmake` in the `passiveScalarFoam` directory. `rehash` if necessary.

Add a passive scalar transport equation (1/3)

- Let's add, to `passiveScalarFoam`, the passive scalar transport equation

$$\frac{\partial s}{\partial t} + \nabla \cdot (\mathbf{u}\ s) = 0$$

- We must modify the solver:
  - Create `volumeScalarField s` (do the same as for `p` in `createFields.H`)
  - Add the equation `solve(fvm::ddt(s) + fvm::div(phi, s));` before `runTime.write();` in `passiveScalarFoam.C`.
  - Compile `passiveScalarFoam` using `wmake`
- We must modify the case - next slide ...

Add a passive scalar transport equation (2/3)

- We must modify the case:
  - Use the `icoFoam/cavity` case as a base:
    ```
    run
    cp -r $FOAM_TUTORIALS/incompressible/icoFoam/cavity passiveCavity
    cd passiveCavity
    ```
  - Copy the `0/p` file to `0/s` and modify `p` to `s` in that file. Choose approprate dimensions for the scalar field (not important now).
  - In `fvSchemes`, add (if you don't, it will complain):
    ```
        div(phi,s)      Gauss linearUpwind Gauss;
    ```
  - In `fvSolution`, add (if you don't, it will complain):
    ```
        s PBiCG
        {
            preconditioner   DILU;
            tolerance        1e-05;
            relTol           0;
        };
    ```
    (if you use `PCG`, as for `p`, it will complain - try it yourself!)
- We must initialize and run the case - next slide ...

Add a passive scalar transport equation (3/3)

- We must initialize `s`:
  - `cp $FOAM_TUTORIALS/multiphase/interFoam/laminar/damBreak/system/setFieldsDict system`
  - Set `defaultFieldValues`:
    `volScalarFieldValue s 0`
  - Modify the bounding box to:
    `box (0.03 0.03 -1) (0.06 0.06 1);`
  - Set `fieldValues`:
    `volScalarFieldValue s 1`
- Run the case:
  `blockMesh`
  `setFields`
  `passiveScalarFoam >& log`
  `paraFoam` - mark `s` in Volume Fields, color by `s` (cell value) and run an animation.
- You can see that although there is no diffusion term in the equation, there is massive diffusion in the results. This is due to mesh resolution, numerical scheme etc. The `interfoam` solver has a special treatment to reduce this kind of diffusion.

A look inside icoFoam

- The `icoFoam` directory consists of the following:

  ```
  createFields.H  Make/  icoFoam.C
  ```

- The `Make` directory contains instructions for the `wmake` compilation command.
- `icoFoam.C` is the main file, and `createFields.H` is an inclusion file, which is included in `icoFoam.C`.
- In the header of `icoFoam.C` we include `fvCFD.H`, which contains all class definitions that are needed for `icoFoam`. `fvCFD.H` is included from (see `Make/options`):
  `$WM_PROJECT_DIR/src/finiteVolume/lnInclude`, but that is actually only a link to
  `$WM_PROJECT_DIR/src/finiteVolume/cfdTools/general/include/fvCFD.H`.
  `fvCFD.H` in turn only includes other files that are needed (see next slide).
- Hint: Use `find PATH -iname "*LETTERSINFILENAME*"` to find where in `PATH` a file with a file name containing `LETTERSINFILENAME` in its file name is located.
  In this case: `find $WM_PROJECT_DIR -iname "*fvCFD.H*"`
- Hint: Use `locate fvCFD.H` to find all files with `fvCFD.H` in their names. Note that `locate` is much faster than `find`, but is not frequently updated when files are added and removed!

A look inside icoFoam, fvCFD.H

```
#ifndef fvCFD_H
#define fvCFD_H

#include "parRun.H"

#include "Time.H"
#include "fvMesh.H"
#include "fvc.H"
#include "fvMatrices.H"
#include "fvm.H"
#include "linear.H"
#include "uniformDimensionedFields.H"
#include "calculatedFvPatchFields.H"
#include "fixedValueFvPatchFields.H"
#include "adjustPhi.H"
#include "findRefCell.H"
#include "mathematicalConstants.H"
#include "OSspecific.H"
#include "argList.H"
#include "timeSelector.H"

#ifndef namespaceFoam
#define namespaceFoam
    using namespace Foam;
#endif

#endif
```

The inclusion files are all class definitions that are used in `icoFoam`. Dig further into the source file to find out what these classes actually do.
At the end we say that we will use all definitions made in `namespace Foam`.

A look inside icoFoam

- `icoFoam` starts with

  ```
  int main(int argc, char *argv[])
  ```

  where `int argc, char *argv[]` are the number of parameters, and the actual parameters used when running `icoFoam`.

- The case is initialized by:

  ```
  #    include "setRootCase.H"

  #    include "createTime.H"
  #    include "createMesh.H"
  #    include "createFields.H"
  #    include "initContinuityErrs.H"
  ```

  where all inclusion files except `createFields.H` are included from `src/OpenFOAM/lnInclude` and `src/finiteVolume/lnInclude`. Have a look at them yourself. (find them using the `find` or `locate` commands)

- `createFields.H` is located in the `icoFoam` directory. It initializes all the variables used in `icoFoam`. Have a look inside it and see how the variables are created from files.

A look inside icoFoam

- The time loop starts by:

  ```
  while (runTime.loop())
  ```

  and the rest is done at each time step.

- The `fvSolution` subdictionary `PISO` is read, and the Courant number is calculated and written to the screen by (use the `find` command)

  ```
  #       include "readPISOControls.H"
  #       include "CourantNo.H"
  ```

- We will now discuss the PISO algorithm used in `icoFoam`, in words, equations and code lines.

The PISO algorithm: The incompressible flow equations (1/7)

(Acknowledgements to Professor Hrvoje Jasak)

- In strictly incompressible flow the coupling between density and pressure is removed, as well as the coupling between the energy equation and the rest of the system.
- The incompressible continuity and momentum equations are given by:

  $$\nabla \cdot \mathbf{u} = 0$$

  $$\frac{\partial \mathbf{u}}{\partial t} + \nabla \cdot (\mathbf{u}\mathbf{u}) - \nabla \cdot (\nu \nabla \mathbf{u}) = -\nabla p$$

- The non-linearity in the convection term ($\nabla \cdot (\mathbf{u}\mathbf{u})$) is handled using an iterative solution technique, where

  $$\nabla \cdot (\mathbf{u}\mathbf{u}) \approx \nabla \cdot (\mathbf{u}^{\mathrm{o}}\mathbf{u}^{\mathrm{n}})$$

  where $\mathbf{u}^{\mathrm{o}}$ is the currently available solution and $\mathbf{u}^{\mathrm{n}}$ is the *new* solution. The algorithm cycles until $\mathbf{u}^{\mathrm{o}} = \mathbf{u}^{\mathrm{n}}$.
- There is no pressure equation, but the continuity equation imposes a scalar constraint on the momentum equation (since $\nabla \cdot \mathbf{u}$ is a scalar).

The PISO algorithm: The idea behind the algorithm (2/7)

(Acknowledgements to Professor Hrvoje Jasak)

- The idea of PISO is as follows:
  - Pressure-velocity systems contain two complex coupling terms:
    - Non-linear convection term, containing u-u coupling.
    - Linear pressure-velocity coupling.
  - On low Courant numbers (small time-step), the pressure-velocity coupling is much stronger than the non-linear coupling.
  - It is therefore possible to repeat a number of pressure correctors without updating the discretization of the momentum equation (without updating $\mathbf{u}^o$).
  - In such a setup, the first pressure corrector will create a conservative velocity field, while the second and following will establish the pressure distribution.
- Since multiple pressure correctors are used with a single momentum equation, it is not necessary to under-relax neither the pressure nor the velocity.
- On the negative side, the derivation of PISO is based on the assumption that the momentum discretization may be safely frozen through a series of pressure correctors, which is true only at small time-steps. Experience also shows that the PISO algorithm is more sensitive to mesh quality than the SIMPLE algorithm.

The PISO algorithm: Derivation of the pressure equation (3/7)

(Acknowledgements to Professor Hrvoje Jasak)

- As previously mentioned, there is no pressure equation for incompressible flow, so we use the continuity and momentum equations to derive a pressure equation.
- Start by discretizing the momentum equation, keeping the pressure gradient in its original form:

$$a_P^{\mathbf{u}}\mathbf{u}_P + \sum_N a_N^{\mathbf{u}}\mathbf{u}_N = \mathbf{r} - \nabla p$$

- Introduce the $\mathbf{H}(\mathbf{u})$ operator:

$$\mathbf{H}(\mathbf{u}) = \mathbf{r} - \sum_N a_N^{\mathbf{u}}\mathbf{u}_N$$

so that:

$$\begin{aligned} a_P^{\mathbf{u}}\mathbf{u}_P &= \mathbf{H}(\mathbf{u}) - \nabla p \\ \mathbf{u}_P &= (a_P^{\mathbf{u}})^{-1}(\mathbf{H}(\mathbf{u}) - \nabla p) \end{aligned}$$

- Substitute this in the incompressible continuity equation ($\nabla \cdot \mathbf{u} = 0$) to get a pressure equation for incompressible flow:

$$\nabla \cdot \left[(a_P^{\mathbf{u}})^{-1}\nabla p\right] = \nabla \cdot \left[(a_P^{\mathbf{u}})^{-1}\mathbf{H}(\mathbf{u})\right]$$

The PISO algorithm: Sequence of operations (4/7)

(Acknowledgements to Professor Hrvoje Jasak)

- The following description corresponds to the operations at each time step.
- Use the conservative fluxes, `phi`, derived from the previous time step, to discretize the momentum equation. Now, `phi` represents the 'old' velocity, u^o, in the convective term.

```
fvVectorMatrix UEqn
(
    fvm::ddt(U)
  + fvm::div(phi, U)
  - fvm::laplacian(nu, U)
);
```

- Solve the momentum equations using the pressure from the previous time step.

```
solve(UEqn == -fvc::grad(p));
```

  This is the momentum predictor step.

- We will re-use `UEqn` later, which is the reason not to do both these steps as a single operation `solve(fvm::ddt(U)+fvm::div(phi, U)-fvm::laplacian(nu, U)==-fvc::grad(p));`

The PISO algorithm: Sequence of operations (5/7)

(Acknowledgements to Professor Hrvoje Jasak)

- Loop the pressure-corrector step a fixed number of times (`nCorr`):
  - Store `rUA*UEqn.H()` (corresponding to $(a_P^{\mathbf{u}})^{-1}\mathbf{H}(\mathbf{u})$) in the `U` field, representing the velocity solution without the pressure gradient. Calculate interpolated face fluxes from the approximate velocity field (corrected to be globally conservative so that there is a solution to the pressure equation) to be used in the `fvc::div` operator.
  - Loop the non-orthogonal corrector step a fixed number of times (`nNonOrthCorr`):
    * Calculate the new pressure:
      ```
      fvScalarMatrix pEqn ( fvm::laplacian(rUA, p) == fvc::div(phi) );
      pEqn.setReference(pRefCell, pRefValue);
      pEqn.solve();
      ```
      where `rUA` corresponds to $(a_P^{\mathbf{u}})^{-1}$.
    * Correct finally `phi` for the next pressure-corrector step (see also next slide):
      ```
      if (nonOrth == nNonOrthCorr){ phi -= pEqn.flux(); }
      ```
  - Calculate and write out the continuity error.
  - Correct the approximate velocity field using the corrected pressure gradient.
- Do the next pressure-corrector step.

The PISO algorithm: Conservative face fluxes (6/7)

(Acknowledgements to Professor Hrvoje Jasak)

- Here we derive the conservative face fluxes used in `pEqn.flux()` in the previous slide.
- Discretize the continuity equation:

  $$\nabla \cdot \mathbf{u} = \sum_f \mathbf{s}_f \cdot \mathbf{u} = \sum_f F$$

  where F is the face flux, $F = \mathbf{s}_f \cdot \mathbf{u}$.
- Substitute the expression for the velocity in 'PISO slide (3/7)' ($\mathbf{u}_P = (a_P^\mathbf{u})^{-1}(\mathbf{H}(\mathbf{u}) - \nabla p)$), yielding

  $$F = -(a_P^\mathbf{u})^{-1}\mathbf{s}_f \cdot \nabla p + (a_P^\mathbf{u})^{-1}\mathbf{s}_f \cdot \mathbf{H}(\mathbf{u})$$
- A part of the above appears during the discretization of the pressure Laplacian, for each face:

  $$(a_P^\mathbf{u})^{-1}\mathbf{s}_f \cdot \nabla p = (a_P^\mathbf{u})^{-1}\frac{|\mathbf{s}_f|}{|\mathbf{d}|}(p_N - p_P) = a_N^P(p_N - p_P)$$

  where $|\mathbf{d}|$ is the distance between the owner and neighbour cell centers, and $a_N^P = (a_P^\mathbf{u})^{-1}\frac{|\mathbf{s}_f|}{|\mathbf{d}|}$ is the off-diagonal matrix coefficient in the pressure Laplacian. For the fluxes to be fully conservative, they must be completely consistent with the assembly of the pressure equation (e.g. non-orthogonal correction).

The PISO algorithm: Rhie & Chow interpolation (7/7)

(Acknowledgements to Dr. Fabian Peng-Kärrholm and Professor Hrvoje Jasak)

- When using a colocated FVM formulation it is necessary to use a special interpolation to avoid unphysical pressure oscillations.
- OpenFOAM uses an approach 'in the spirit of Rhie & Chow', but it is not obvious how this is done. Fabian presents a discussion on this in his PhD thesis, and here is the summary of the important points:
  - In the explicit source term `fvc::div(phi)` of the pressure equation, `phi` does not include any effect of the pressure.
  - `rUA` does not include any effect of pressure when solving the pressure equation and finally correcting the velocity.
  - The Laplacian term, `fvm::laplacian(rUA, p)`, of the pressure equation uses the value of the gradient of p on the cell faces. The gradient is calculated using neighbouring cells, and not neighbouring faces.
  - `fvc::grad(p)` is calculated from the cell face values of the pressure.
- See *Rhie and Chow in OpenFOAM*, by Fabian Peng Kärrholm at the course homepage, 2007, for a detailed description of the PISO algorithm and Rhie and Chow in OpenFOAM.

A look inside icoFoam, write statements

- At the end of icoFoam there are some write statements:

```
runTime.write();

Info<< "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
    << "  ClockTime = " << runTime.elapsedClockTime() << " s"
    << nl << endl;
```

- `write()` makes sure that all variables that were defined as an `IOobject` with `IOobject::AUTO_WRITE` are written to the time directory accoring to the settings in the `controlDict` dictionary.
- `elapsedCpuTime()` is the elapsed CPU time.
- `elapsedClockTime()` is the elapsed wall clock time.

A look inside icoFoam, summary of the member functions

- Some of the member functions used in `icoFoam` are described below. The descriptions are taken from the classes of each object that was used when calling the functions.
  `A()`: Return the central coefficient of an `fvVectorMatrix`.
  `H()`: Return the H operation source of an `fvVectorMatrix`.
  `Sf()`: Return cell face area vectors of an `fvMesh`.
  `flux()`: Return the face-flux field from an `fvScalarMatrix`
  `correctBoundaryConditions()`: Correct boundary field of a `volVectorField`.

- Find the descriptions by identifying the object type (class) and then search the OpenFOAM Doxygen at: `http://foam.sourceforge.net/doc/Doxygen/html/` (linked to from `www.openfoam.com`).

- You can also find the Doxygen documentation by doing:
  `firefox file://$WM_PROJECT_DIR/doc/Doxygen/html/index.html`
  This requires that the Doxygen documentation was compiled. If so, it would correspond to the exact code that you have currently installed rather than the version the documentation was originally compiled for, found at `www.openfoam.com`. Unfortunately, the search functionality only works when running firefox through a php server.

- See the presentation by Martin Beaudoin at the 2007 course, on how to adapt the Doxygen documentation, and include your own development.