AM119: HW3 and OpenFOAM tutorial

Prof. Trask

March 14, 2016

1 Assignment 3: Burgers equation component

In class, we learned how the balance of momentum gives rise to non-linear flux terms. To obtain an understanding of how this translates to our 1D periodic example, we're going to rerun last weeks code solving Burger's equation instead. Whereas the advection equation had the form:

$$\partial_t u + a \partial_x u = 0$$

We will now we solving what is referred to as **Burger's equation**:

$$\partial_t u + u \partial_x u = 0$$

We can write this in conservative form as:

$$\partial_t u + \partial_x \left(\frac{1}{2} u^2 \right) = 0$$

or

$$\frac{du}{dt} + \nabla \cdot F = 0$$

$$F = \frac{1}{2} u^2$$

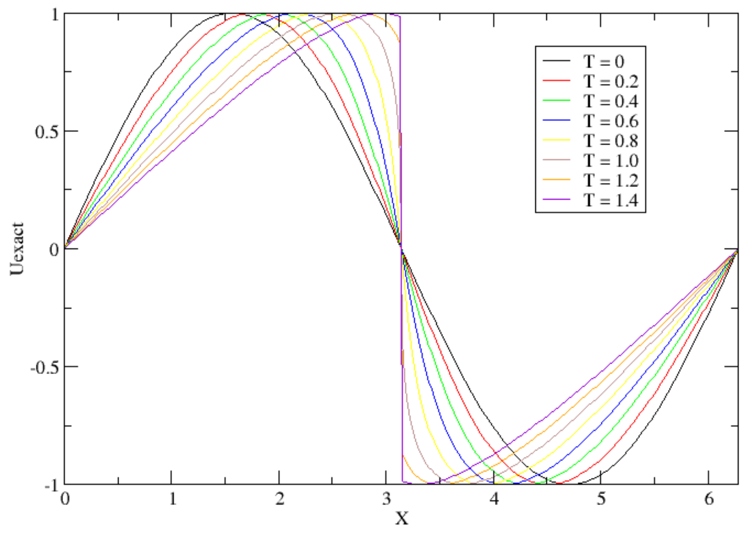
This code will be **identical to your code from HW2**, and will only require the modification of the flux calculation. Whereas previously we treated the advection flux using upwinding, the upwind direction for Burgers equation will depend upon whether u is positive or negative. We can write this mathematically via

$$V_i \frac{u_i^{n+1} - u_n^n}{\Delta t} = -\left(F_{i+1/2} - F_{i-1/2} \right)$$

where $F_{i+1/2} = f(u_i, u_{i+1})$ and $F_{i-1/2} = f(u_{i-1}, u_i)$, where f is calculated via

$$f(u,v) = \begin{cases} min\left(F(u), F(v)\right) & \text{if } u \leq v \\ max\left(F(u), F(v)\right) & \text{if } u \geq v \end{cases}.$$

For this assignment, implement the new flux term, run your solver to $T_{final} = 1.4$ and demonstrate that your solution matches the following analytic result.



2 Incompressible flow through a pipe

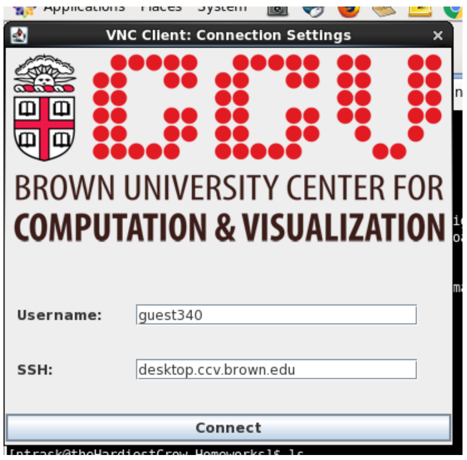
In the notes for Lecture 3 we discussed an example where initially uniform pipe flow settles into a parabolic velocity profile, and through a mass conservation argument we derived a relationship between the inlet velocity U_∞ and the downstream velocity profile. Today, we will set up an OpenFOAM simulation to demonstrate numerically that this relationship is true - we will see later in the course how to derive the velocity profile analytically.

Before we start, you'll need to log into Oscar through CCV's VNC client.

In our last homework assignment we implemented the finite difference method to solve the 1D advection diffusion equation. Today, we've introduced the finite volume method, which we will use at length for the remainder of the class. For this assignment, we will be repeating the previous homework but swapping in a finite volume discretization. This should require minimal changes to the previous homework assignment. A solution to the previous assignment will be put on the website tomorrow (3/1) - if you had difficulty completing the previous assignment feel free to use the solution as a starting point.

3 Get onto Oscar

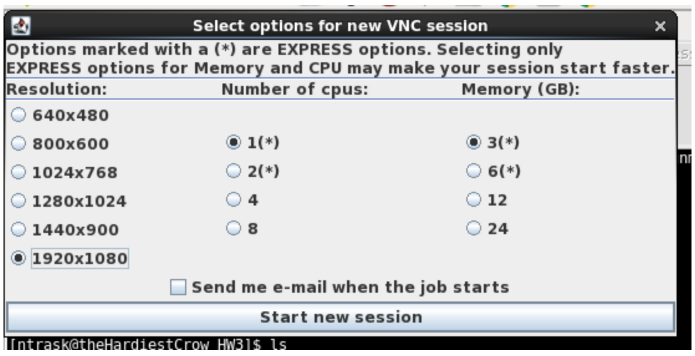
We've gone over this in class but I'll gather the relevant bits and pieces here for reference. First download the VNC client from the CCV webpage (`https://www.ccv.brown.edu/technologies/vnc`). You'll need Java to launch this - Java is on all of the machines in the CIT, but see me if you need help getting this up on your home machine. Once you launch the jar file, you should see a scene that looks like:


VNC Client: Connection Settings
CCV
BROWN UNIVERSITY CENTER FOR
COMPUTATION & VISUALIZATION
Connect

Username: guest340

SSH: desktop.ccv.brown.edu

Enter your username and password (case sensitive, and be careful as 5 unsuccessful logins will lock your account). Next some configuration options will pop up. You want the default options, but you might want to play with the resolution settings to find a desktop that matches your screen resolution best. For the purposes of this class, there is no benefit to choosing more processors, and doing so will make your session take longer to load. You might get a message that your session hasn't loaded yet - just keep clicking retry until you're successfully logged in.



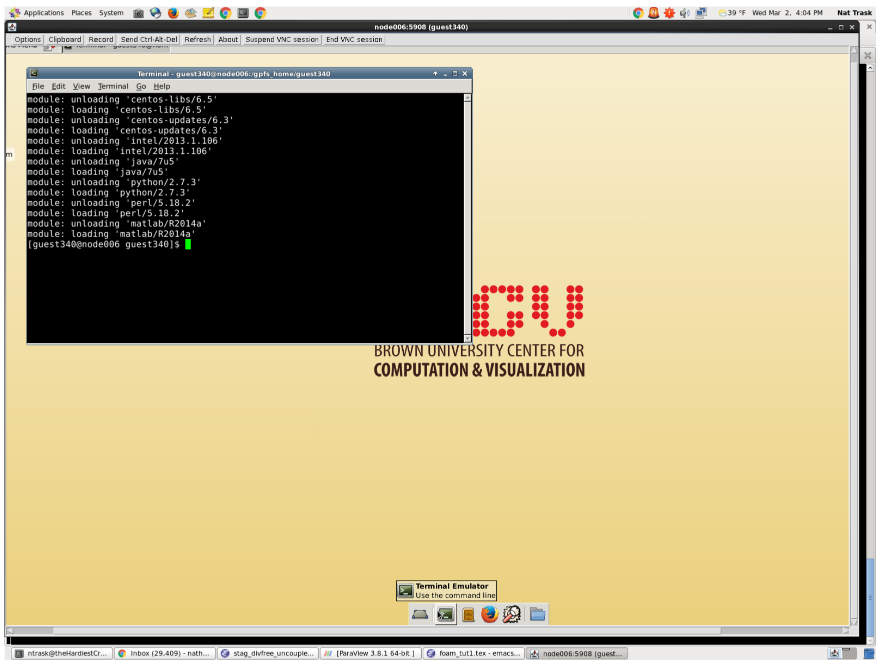
Options marked with a (*) are EXPRESS options. Selecting only EXPRESS options for Memory and CPU may make your session start faster.

Resolution:	Number of cpus:	Memory (GB):
640x480		
800x600	[✓ 1(*)]	[✓ 3(*)]
1024x768	2(*)	6(*)
1280x1024	4	12
1440x900	8	24
[✓ 1920x1080]		

☐ Send me e-mail when the job starts

```
[ntrask@theHardiestCrow HW3]$ ls
```

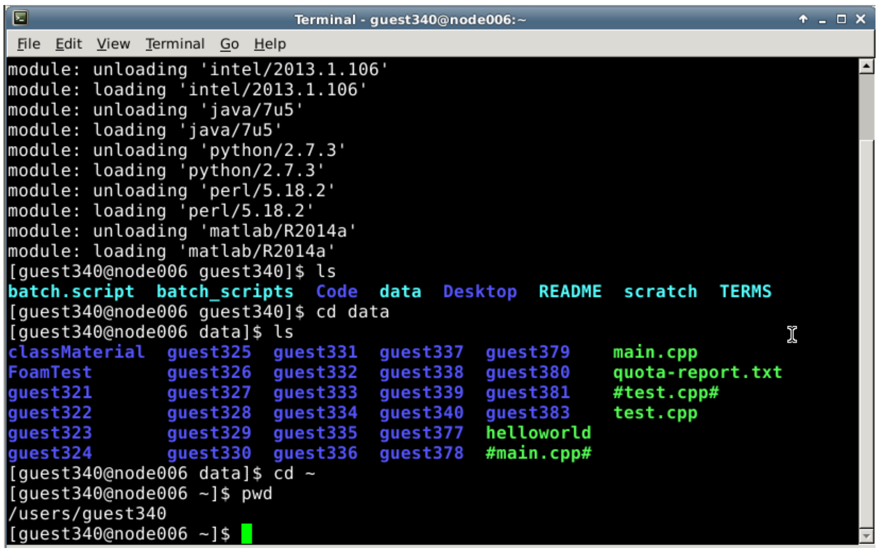
You should now have a virtual desktop setup - congratulations!!! you are now remotely logged into the department cluster. What we'll do now is start up a terminal - click on the black boxy looking thing in the tray at the bottom of the screen.



```
module: unloading 'centos-libs/6.5'
module: loading 'centos-libs/6.5'
module: unloading 'centos-updates/6.3'
module: loading 'centos-updates/6.3'
module: unloading 'intel/2013.1.106'
module: loading 'intel/2013.1.106'
module: unloading 'java/7u5'
module: loading 'java/7u5'
module: unloading 'python/2.7.3'
module: loading 'python/2.7.3'
module: unloading 'perl/5.18.2'
module: loading 'perl/5.18.2'
module: unloading 'matlab/R2014a'
module: loading 'matlab/R2014a'
[guest340@node006 guest340]$
```

This terminal is where we'll spend all of our time. I've included a cheatsheet of useful Unix commands on the course website. To get warmed up, open a terminal and run the following set of commands to: look at the contents of your home directory, move into the data directory, look around there, and move back home. You should see the following output.

```
$ ls
$ cd data
$ ls
$ cd ~
```

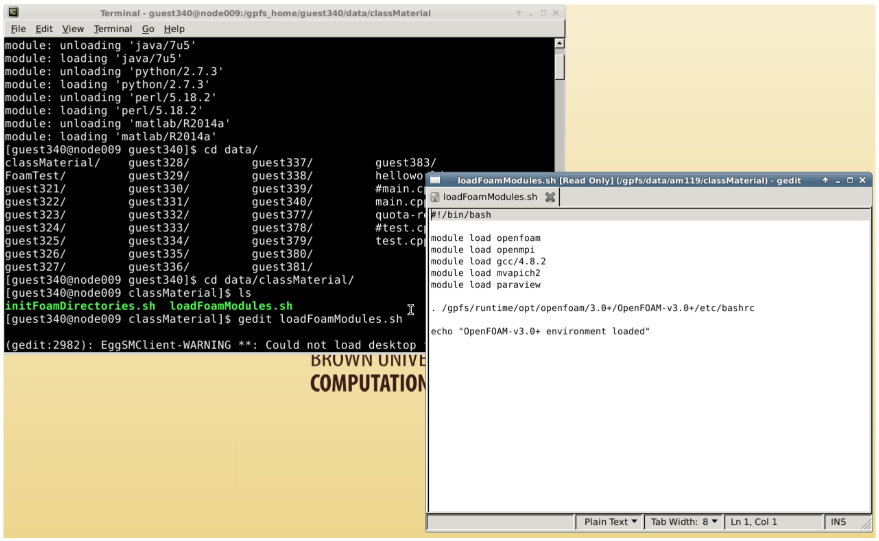

Terminal - guest340@node006:~
File Edit View Terminal Go Help

```
module: unloading 'intel/2013.1.106'
module: loading 'intel/2013.1.106'
module: unloading 'java/7u5'
module: loading 'java/7u5'
module: unloading 'python/2.7.3'
module: loading 'python/2.7.3'
module: unloading 'perl/5.18.2'
module: loading 'perl/5.18.2'
module: unloading 'matlab/R2014a'
module: loading 'matlab/R2014a'
[guest340@node006 guest340]$ ls
batch.script  batch_scripts  Code  data  Desktop  README  scratch  TERMS
[guest340@node006 guest340]$ cd data
[guest340@node006 data]$ ls
classMaterial  guest325  guest331  guest337  guest379    main.cpp
FoamTest       guest326  guest332  guest338  guest380    quota-report.txt
guest321       guest327  guest333  guest339  guest381    #test.cpp#
guest322       guest328  guest334  guest340  guest383    test.cpp
guest323       guest329  guest335  guest377  helloworld
guest324       guest330  guest336  guest378  #main.cpp#
[guest340@node006 data]$ cd ~
[guest340@node006 ~]$ pwd
/users/guest340
[guest340@node006 ~]$ 
```

If this is your first Unix experience - don't worry. It'll take some time to get comfortable moving around using a command line instead of a GUI, but you'll get the hang of it. Next up we're going to set up the OpenFoam environment. I've stored some useful scripts in the `data/classMaterial\verb` directory. A script is just a collection of terminal commands bundled together in a convenient way. To check them out, move to the `classMaterial` directory, and take a look at the setup script

```
$ cd ~
$ cd data/classMaterial
$ ls
$ gedit loadFoamModules.sh
```

You should be able to see a bunch of commands that this script will execute.


Terminal - guest340@node009:/gpfs_home/guest340/data/classMaterial
File Edit View Terminal Go Help
BROWN UNIVE
COMPUTATION
loadFoamModules.sh [Read Only] (/gpfs/data/am119/classMaterial) - gedit
loadFoamModules.sh
Plain Text
Tab Width: 8
Ln 1, Col 1
INS

```
module: unloading 'java/7u5'
module: loading 'java/7u5'
module: unloading 'python/2.7.3'
module: loading 'python/2.7.3'
module: unloading 'perl/5.18.2'
module: loading 'perl/5.18.2'
module: unloading 'matlab/R2014a'
module: loading 'matlab/R2014a'
[guest340@node009 guest340]$ cd data/
classMaterial/    guest328/        guest337/        guest383/
FoamTest/         guest329/        guest338/        hellowo
guest321/         guest330/        guest339/        #main.c
guest322/         guest331/        guest340/        main.cp
guest323/         guest332/        guest377/        quota-r
guest324/         guest333/        guest378/        #test.c
guest325/         guest334/        guest379/        test.cp
guest326/         guest335/        guest380/
guest327/         guest336/        guest381/
[guest340@node009 guest340]$ cd data/classMaterial/
[guest340@node009 classMaterial]$ ls
initFoamDirectories.sh  loadFoamModules.sh
[guest340@node009 classMaterial]$ gedit loadFoamModules.sh

(gedit:2982): EggSMClient-WARNING **: Could not load desktop
```

```
#!/bin/bash

module load openfoam
module load openmpi
module load gcc/4.8.2
module load mvapich2
module load paraview

. /gpfs/runtime/opt/openfoam/3.0+/OpenFOAM-v3.0+/etc/bashrc

echo "OpenFOAM-v3.0+ environment loaded"
```

To execute the scripts in this directory, enter

```
$ source ~/data/classMaterial/initFoamDirectories.sh
$ source ~/data/classMaterial/loadFoamModules.sh
```

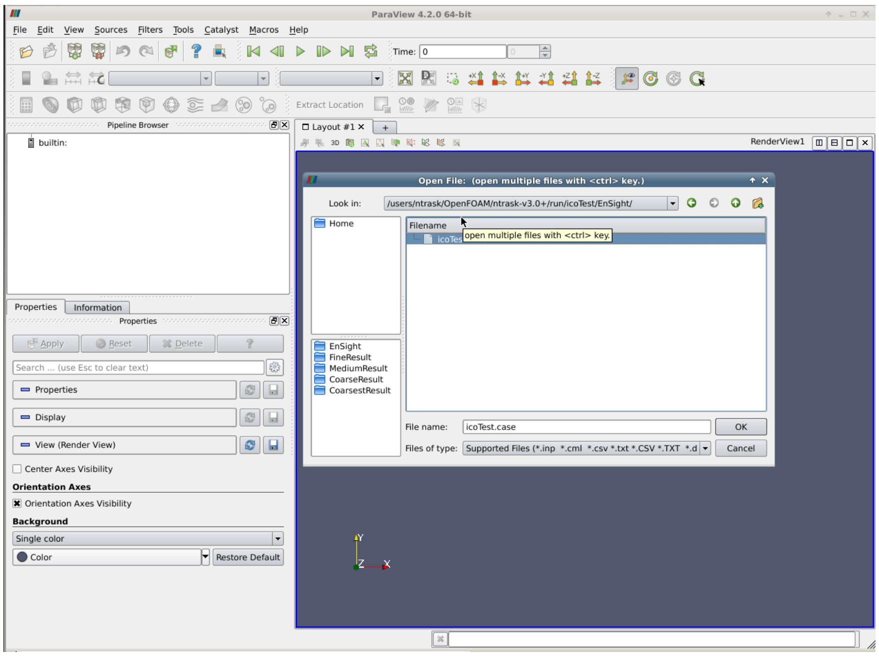
At this point OpenFoam is set up to run! In the future, you only need to run `loadFoamModules.sh` - the other script only needs to be run once to initialize your home directory to be set up with OpenFoam. To get a handle on how to use OpenFoam applications, we're going to simulate the fully developed pipeflow case that we looked at in class. In the data directory you can find a folder called `foamCases`. We're going to copy that into your user directory and run it. OpenFoam has a bunch of terminal shortcuts to make it easy to jump around - for example, typing `run` will take you to your runtime directory.

```
$ run
$ pwd
$ cp ~/data/foamCases/channelCase . -r
$ ls
```

Now you've got a copy of the tutorial case in your run directory that you can mess around with. This case is set up to run the channel flow problem we went over in class. We'll explore the settings for this case, but first let's run the incompressible fluid flow solver and take a look what the flow looks like. We'll run the solver (icoFoam), convert the output to a file format that works with the CCV setup (Ensight), and open a post-processing tool (paraview)

```
$ icoFoam
$ foamToEnsight
$ paraview
$
```

Paraview will start up, and we'll load the case output by going to `File > Load` and opening `Ensight/icoTest.case`.



Time: 0

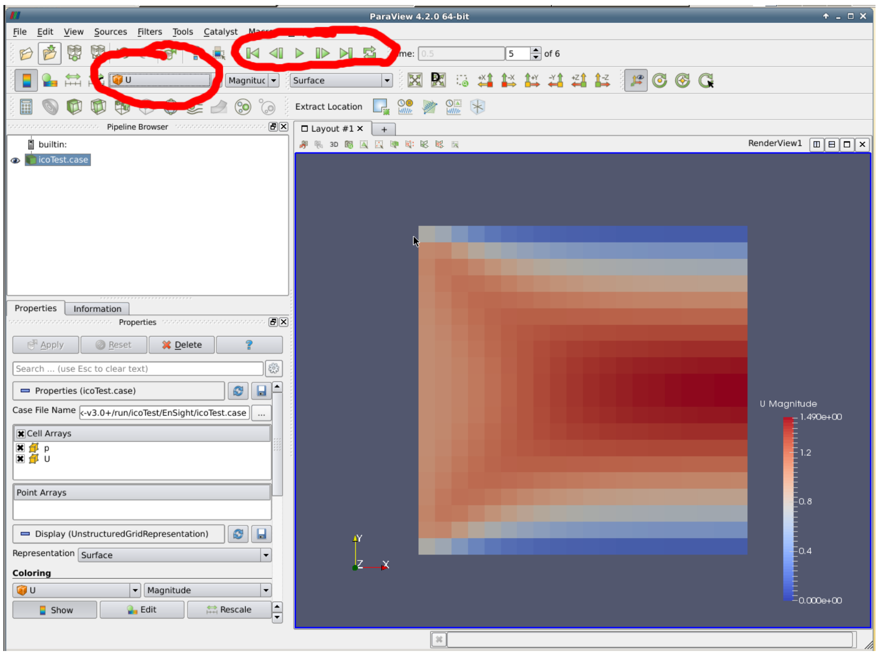
Look in: /users/ntrask/OpenFOAM/ntrask-v3.0+/run/icoTest/EnSight/

File name: icoTest.case

Files of type: Supported Files (*.inp *.cml *.csv *.txt *.CSV *.TXT *.d

Background: Single color

If you click the green button that says "Apply", the geometry will load. We're looking at a 0.1×0.1 box, with a specified uniform flow coming in for the left and walls at the top and bottom. What you're looking at right now is the initial value ($t = 0$) of the pressure - you can move to different timesteps or switch between the velocity and pressure field by playing with the options circled in red below.



Time: 5 of 6

U Magnitude Surface

Properties (icoTest.case)

Case File Name …-v3.0+/run/icoTest/EnSight/icoTest.case

Display (UnstructuredGridRepresentation)

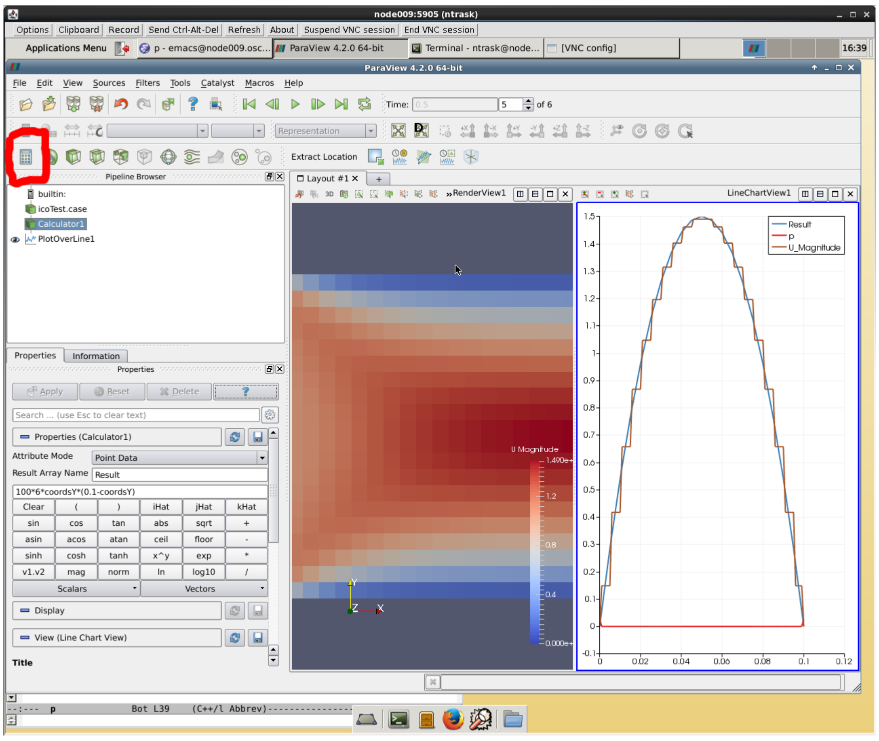
Representation Surface

Coloring: U Magnitude

Let's make a plot of the downstream velocity profile. Select from the menu `Filters > Alphabetical > Plot over line`, enter values for the endpoints of the line as $(0.1, 0, 0.005)$ and $(0.1, 0.1, 0.005)$, and click apply. The velocity profile that you get should look pretty parabolic. Let's see how well it matches up with our theoretical profile. We know from the derivation in class that the velocity profile should satisfy

$$u(y) = \frac{6U_\infty}{L^2} y \left(L - y\right)$$

Let's punch that into the calculator to generate a field we can compare to, and plot how that matches up with what we expect.

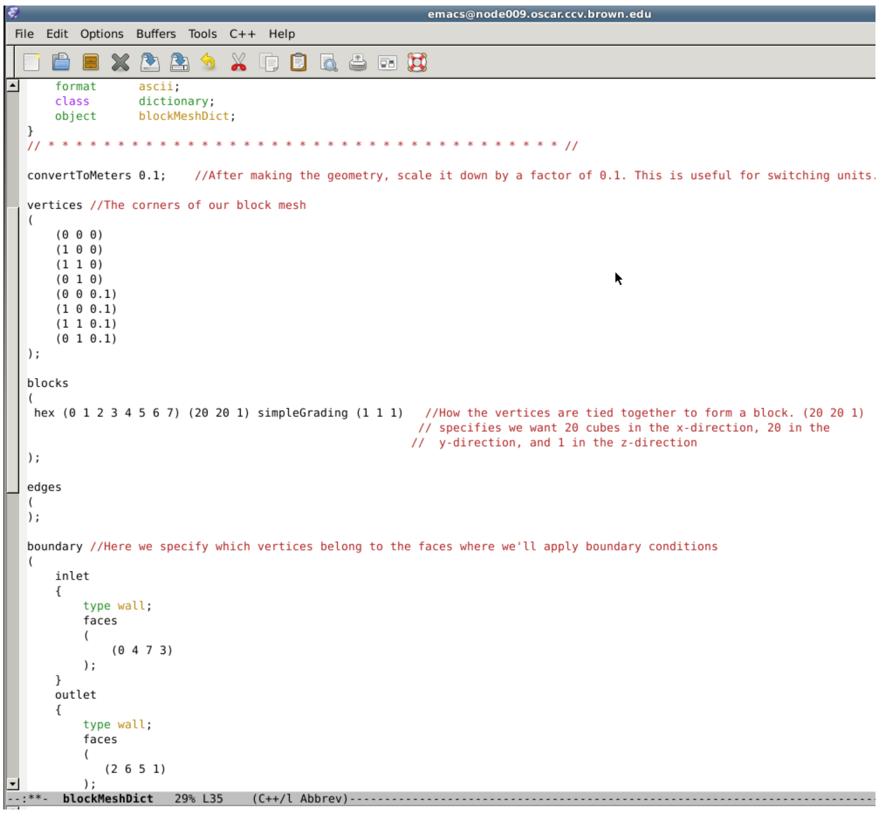


Properties (Calculator1)
Attribute Mode: Point Data
Result Array Name: Result
100*6*coordsY*(0.1-coordsY)

Looks good! Now, for the rest of the tutorial, what we're going to do is explore the case directory and I'll show you how you can tweak the case setup so that you can tackle the questions in HW3. You can go ahead and close paraview. Let's take a look at the directory - you'll see: some numbered folders corresponding to timesteps, a folder called constant where we specify physical parameters, and a folder called system where we setup the case. To start with, lets go into the system folder and take a look at all of the files. We'll start with `blockMeshDict`.

```
$ ls
$ cd system
$ ls
$ gedit blockMeshDict
```

This is where we specify the geometry for our case. In general, making a mesh can be a pretty intense job. blockMesh is an OpenFoam utility for handling very simple geometries. We do this by defining the vertices of blocks and explaining how to stitch them together. I'll explain the details of this on the board.


emacs@node009.oscar.ccv.brown.edu
File Edit Options Buffers Tools C++ Help

```
    format      ascii;
    class       dictionary;
    object      blockMeshDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

convertToMeters 0.1;    //After making the geometry, scale it down by a factor of 0.1. This is useful for switching units.

vertices //The corners of our block mesh
(
    (0 0 0)
    (1 0 0)
    (1 1 0)
    (0 1 0)
    (0 0 0.1)
    (1 0 0.1)
    (1 1 0.1)
    (0 1 0.1)
);

blocks
(
 hex (0 1 2 3 4 5 6 7) (20 20 1) simpleGrading (1 1 1)   //How the vertices are tied together to form a block. (20 20 1)
                                                        // specifies we want 20 cubes in the x-direction, 20 in the
                                                       //  y-direction, and 1 in the z-direction
);

edges
(
);

boundary //Here we specify which vertices belong to the faces where we'll apply boundary conditions
(
    inlet
    {
        type wall;
        faces
        (
            (0 4 7 3)
        );
    }
    outlet
    {
        type wall;
        faces
        (
           (2 6 5 1)
        );
--:**-  blockMeshDict   29% L35    (C++/l Abbrev)-----------------------------------------------------------------
```

To rebuild the mesh according to any changes we make in the blockMesh dictionary, we'll drop back down to the case directory and run the utility

```
$ cd ..
$ blockMesh
```

If you pop open the other dictionaries in the system folder, you'll see that controlDict is where we specify timestepping parameters, fvSchemes is where we specify which sort of fluxes we want to use, and fvSolution is where we specify how to solve linear systems (we haven't talked about the last one in class yet). For HW3, we'll only need to play with the geometry settings so I won't discuss those guys, other than to say they control the foam solver in much the same way as your 1D solver parameters did.

Next up, we'll dig into the constant/transportProperties dictionary. You can see that the only thing there is a definition of the viscosity coefficient. You can tweak this to change the viscosity of the fluid in the simulation.

Finally, one of the numbered directories in the case directory is called 0. This is the initial condition. When you run the solver, it'll make a bunch of other numbered folders saving the output of the simulation at the corresponding times. We initialize the flow by going into the initial condition directory and specifying the initial values and boundary conditions there. If you go into this directory, you'll see a file called U and another called p, corresponding to the velocity and pressure fields. If you pop open the velocity field, you can see the components of what we call a *field* in OpenFoam. A field is a list of values at cell centers in the interior of the domain (*internalField*) and a list of values at face centers on the boundary of the domain (*boundaryField*). You can see that this case is set up so that the velocity on the interior is initially at rest. At the inlet we impose a uniform unit velocity. The walls maintain what's called a no-slip boundary condition, and at the outlet we enforce a zeroGradient boundary condition, which meants that $\partial_n u = 0$ for each component of the velocity vector. We'll get into the details of boundary conditions later in the class, but for HW3 you'll be tweaking the inlet velocity.

4 Assignment 3: Foam component

This exercise is going to be pretty informal, and should be completed in the lab session following lecture today. It turns out that for this problem, the distance downstream that it takes for the velocity to settle into a parabolic profile is roughly inversely proportional to the viscosity. Your job is to:

- Change the viscosity from 0.01 to 0.0005
- Run the solver again - if we generate the plot we can see that at the outlet we're still pretty far from the parabolic downstream profile.
- Double the resolution. Convince me that your answer is refined enough that you trust your answer.
- Get into the blockMesh dictionary and extend the domain. Do some experimenting - if you make it too long the code will get progressively more expensive. Give me a bunch of plots showing me that you can characterize (roughly) the point downstream where the profile is pretty much parabolic.
- Play with the boundary conditions: use our formula relating the downstream flow to the inlet velocity to show that we get the correct answer when you change the inlet from $U = 1$ to $U = 2$ or $U = 3$.

5 Compiling your own solvers

OpenFOAM has a number of solvers already written to solve a whole bunch of different types of problems. In the last section we used the solver *icoFoam* to simulate incompressible, viscous fluid flow. In class today, we derived the Euler equations, which pose the governing equations for compressible, inviscid fluid flow. In the tutorial that follows, we will show how to compile a user-defined solver that I've written to solve the advection-diffusion equation. Next week we will use this as the foundation to solve the Euler equations and start looking at some real fluid mechanics problems!

In the classMaterial folder in the data directory, you will find a folder called solvers with another folder called *advectionDiffusionFoam* in it. In your OpenFOAM user directory, you will find a folder called solvers that is empty. We'll start by copying that solver over into your solver directory.

```
$ run
$ cd ../solvers
$ cp ~/data/classMaterial/advectionDiffusionFoam . -r
$ cd advectionDiffusionFoam
```

You now have a personal copy of the solver that you can do whatever you'd like to. In general this will be how we interact with OpenFOAM - we can find a solver or utility that somebody else has already written, modify it to do what we'd like it to do, and then compile it as a stand-alone custom user solver/utility.

This solver will solve the advection diffusion equation:

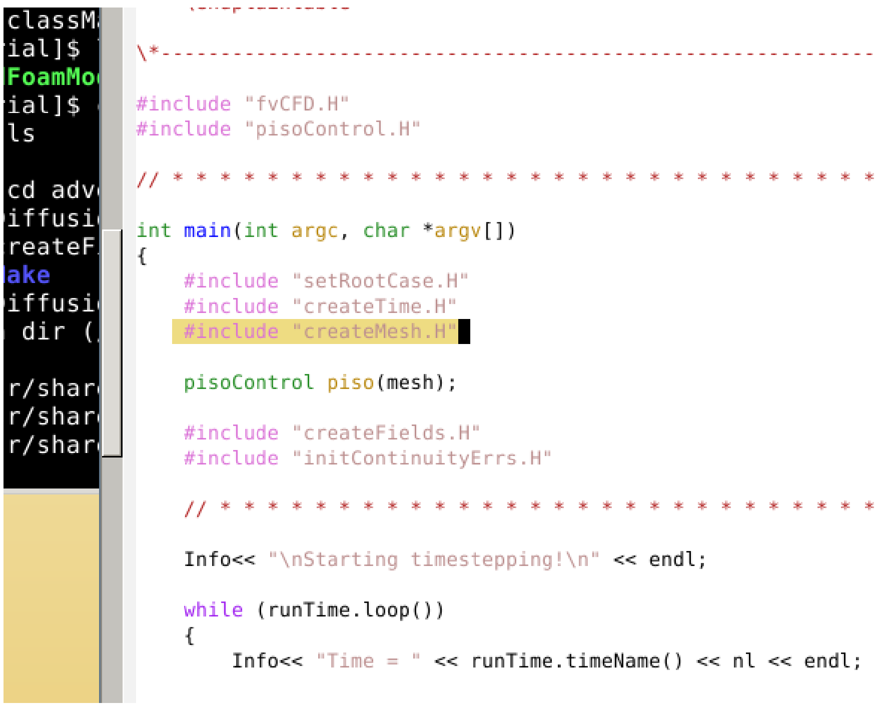
$$\partial_t T + \nabla \cdot (\mathbf{a}T) = \nu \nabla^2 T$$

In order to do this, we will need to:

- Define a field called T
- Define a way to read in the parameters $\mathbf{a}$ and ν from the user
- Define and solve the governing equation

Once we've done that, we'll compile our code and run a sample case to see that this works exactly the same workflow that *icoFoam* did in the previous tutorial.

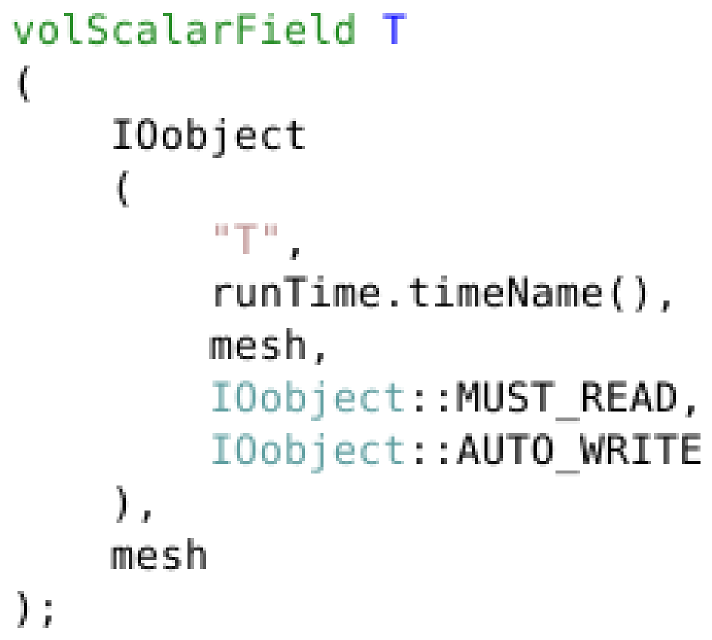
Let's take a look around. If we open up the file *advectionDiffusionFoam.C* in a text editor, we can see the main function that gets called when the solver is executed. In our solver, we use a whole bunch of standard libraries and header files, but the *createMesh.H* include line is where we've done some customization. This is where we will define our custom fields



```
classM
ial]$
FoamMo
ial]$
ls

cd adv
iffusi
reateF
ake
iffusi
 dir (

r/shar
r/shar
r/shar
```

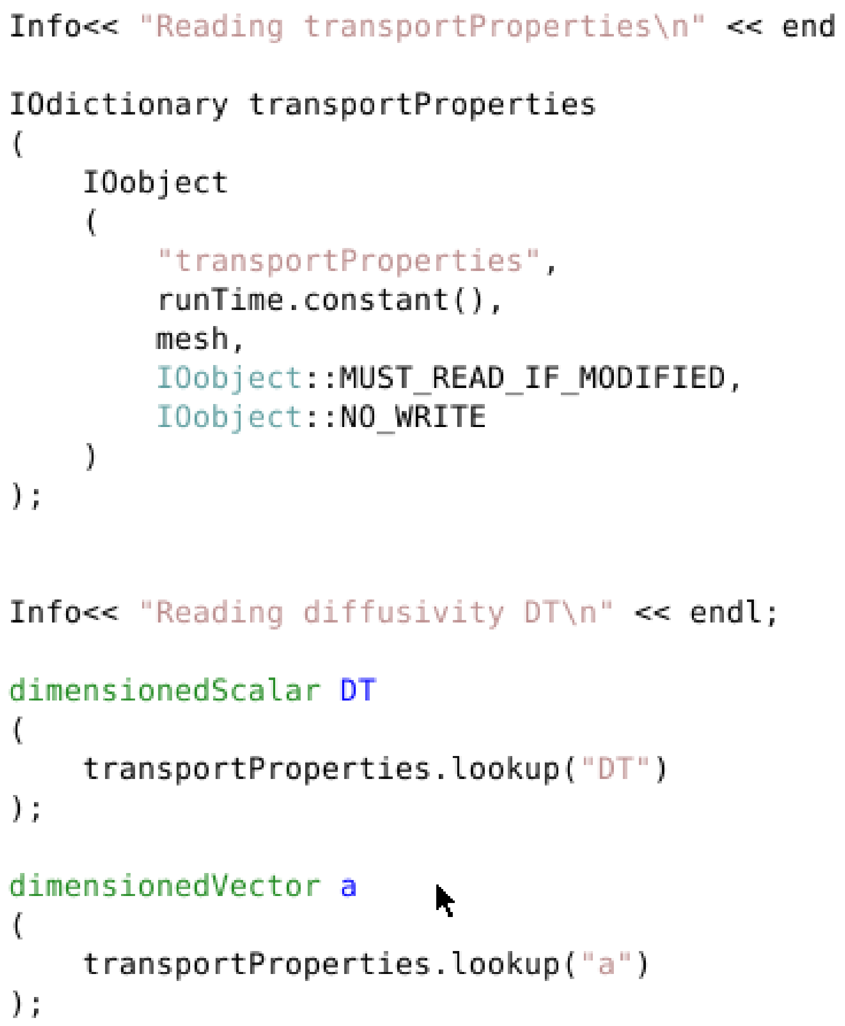
```
\*---------------------------------------------------------------

#include "fvCFD.H"
#include "pisoControl.H"

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

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

    pisoControl piso(mesh);

    #include "createFields.H"
    #include "initContinuityErrs.H"

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

    Info<< "\nStarting timestepping!\n" << endl;

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

If we open up *createMesh.H* you can see where we initialize everything. We start off by defining our field T, and pass along some parameters that will: explain how it hooks into the timestepping scheme, use the mesh to determine how many degrees of freedom need to be stored, explain how it should be initialized when the solvers starts up, and explain how output should be handled. This is the direct analogue of your 1D code when you wrote “vector¡double¿ unew” - in OpenFoam there’s just a whole lot more details to keep track of.



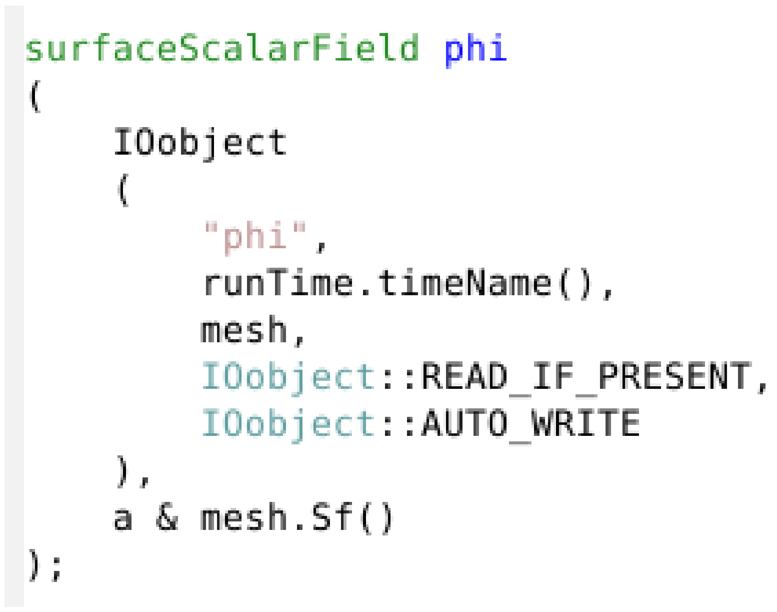
```
volScalarField T
(
    IOobject
    (
        "T",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);
```

After that we define a lookup dictionary for our material parameters. This says that we will pull the diffusion and advection parameters from a dictionary that we will ultimately stick in the *constant* directory.



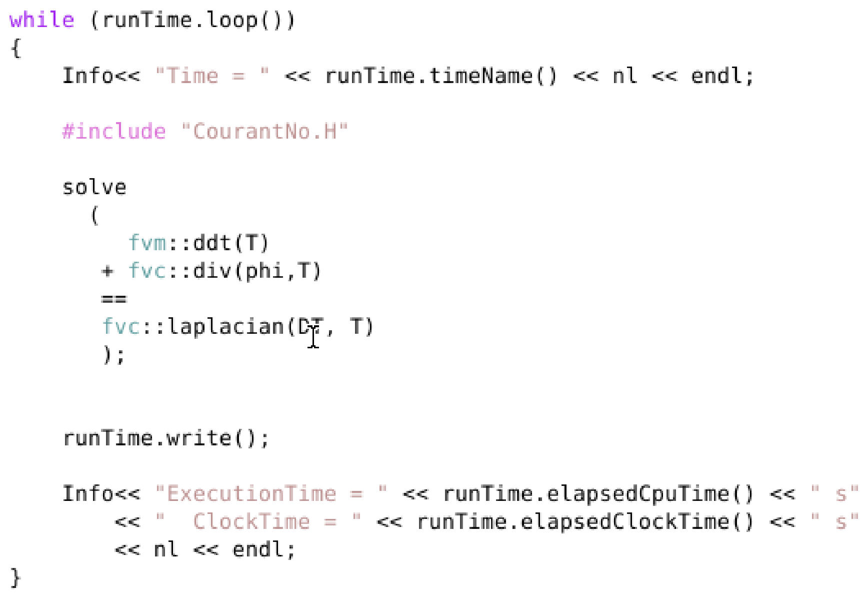
```
Info<< "Reading transportProperties\n" << end

IOdictionary transportProperties
(
    IOobject
    (
        "transportProperties",
        runTime.constant(),
        mesh,
        IOobject::MUST_READ_IF_MODIFIED,
        IOobject::NO_WRITE
    )
);


Info<< "Reading diffusivity DT\n" << endl;

dimensionedScalar DT
(
    transportProperties.lookup("DT")
);

dimensionedVector a
(
    transportProperties.lookup("a")
);
```

Finally, we define a *surfaceScalarField* object called *phi*. Our field T is a *volScalarField* - this is a list of scalars that is associated with the interior of every cell in the domain together with a list of face values on the boundary where boundary conditions are imposed. We know from our 1D cases that fluxes need to be defined at face, not at cells. So what we will do is store the value of $\mathbf{a} \cdot \hat{n}$ integrate against every face of the domain - this will be the advective flux.



```
surfaceScalarField phi
(
    IOobject
    (
        "phi",
        runTime.timeName(),
        mesh,
        IOobject::READ_IF_PRESENT,
        IOobject::AUTO_WRITE
    ),
    a & mesh.Sf()
);
```

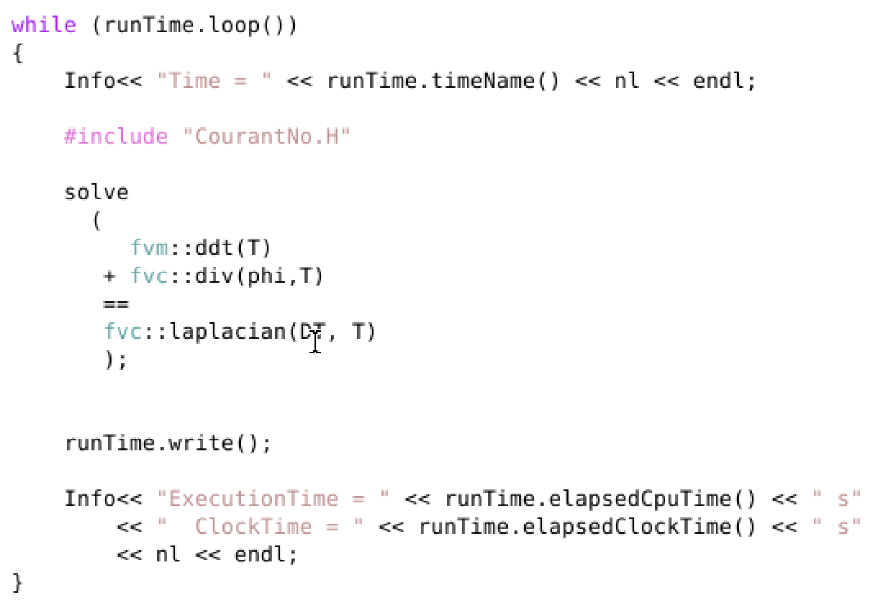
All we have left to do now is define the PDE that we'll solve at every timestep. Switching back to *advectionDiffusionFoam.C*, you can see that each timestep is defined as follows



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

    #include "CourantNo.H"

    solve
      (
         fvm::ddt(T)
       + fvc::div(phi,T)
       ==
       fvc::laplacian(DT, T)
       );


    runTime.write();

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

Here the *solve* function plays the same role as the *update* function in your 1D codes. This is much more sophisticated though - at each timestep the solver will lookup up in the *system* directory how the fluxes are defined, how the time discretization is handled, and so on.

At this point we've got our solver written - let's compile it. Compiling a big piece of code can get pretty complicated - OpenFOAM uses what's called a Makefile system. Instead of writing all of the (possibly hundred of) different *cpp* and header files that make up a big piece of code, a Makefile keeps a big list of all of their interdependencies to see how they rely on one another. If we pop open *advectionDiffusionFoam/Make/files* we can see two lines. One we list all of the *cpp* files that we have defined in our code. Then we list what our executable should be called and where we would like the compiler to stick it. Think of this second line as corresponding to the *-o executableName* flag that we've seen before in the gcc compiler.



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

    #include "CourantNo.H"

    solve
      (
         fvm::ddt(T)
       + fvc::div(phi,T)
       ==
       fvc::laplacian(DT, T)
       );


    runTime.write();

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

If we pop open *advectionDiffusionFoam/Make/options*, we can see where we give the compiler a list of all the folders that our header files are hidden in, along with a list of OpenFOAM libraries that we use. Makefiles are straightforward in premise but can get complicated in their technical details, for now we'll leave it at that and go back to the solver folder. If we run the command

```
$ run
$ cd ../solvers/advectionDiffusionFoam
$ wclean
$ wmake
```

The compiling utility *wmake* will execute the makefile and put all of our pieces together. If everything went as planned, you should be able to autocomplete the solver in the terminal by starting to type *advectionDiffusionFoam* and then hitting tab.

At this point we've got a working solver - let's take it for a spin. In the *classMaterial* directory you will find a folder called *advectionDiffusionTest*. If you build a mesh with blockMesh and run the solver in the usual way:

```
$ blockMesh
$ advectionDiffusionFoam
```

You will get a solution in the same way as in the previous assignment.

5.1 Homework

The additional objective for this weeks homework assignment is to:

- Set the advection velocity to $\mathbf{a} = \{2, 1, 0\}$
- Plot results for the case where $\nu = 10^{-15}$ and $\nu = 10^{-1}$.
- For both of these cases, present your results for a series of increasingly refined meshes (for example, for 16×16,32×32, and 64×64 cells)

The objectives for this part of the assignment have been left intentionally vague. Please try to figure out how to do this on your own without talking to your classmates - the best way to learn how code works is to explore it on your own and try to reverse engineer how it should work. I will give you the hint that everything you need to complete this task is in the case directory.