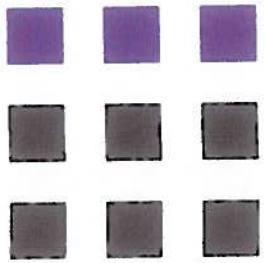
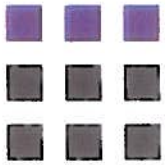


CFD Direct Training Manual

Programming CFD



CFD Direct

Programming CFD

OpenFOAM Training

C.J. Greenshields

Notes v4.1 rev 1. 22/11/2016

- Introduction to programming
1.1 Coding and compiling
1.2 OpenFOAM core classes
1.3 Tools and practice

- Utility applications
2.1 Data Constuction
2.2 Data Access
2.3 Data Input
2.4 Data Operations

- Model development
3.1 Object Orientation
3.2 Interfaces

- Boundary conditions
4.1 Class hierarchy
4.2 Generic Programming
4.3 Class Functions / Data
4.4 Design and implementation

- Solver applications
5.1 Fields
5.2 Equation and algorithms
5.3 Derivatives and algebra

- Data Analysis
6.1 Data Processing
6.2 Data Output

- Task: create an application called cppSandbox that:
- ...creates an integer $a = 2$ and real number $b = 3.4$; calculates $a \times b$;
- ...and writes the answer to standard output (the terminal)

- From the `run` directory, make a `cppSandbox` directory; go into it

```
>> run
>> mkdir cppSandbox
>> cd cppSandbox
```

- Open an editor, *e.g.* gedit, and create a file named `cppSandbox.C`

```
#include <iostream>

int main()
{
    int a = 2;
    double b = 3.4;

    std::cout<< a << " * " << b << " = " << a*b << "\n";

    return 0;
}
```

- Use wmake compilation tool
- Run wmakeFilesAndOptions: creates `Make` dir. with `files` & `options` files

```
>> wmakeFilesAndOptions
```

- Edit `files`:

```
cppSandbox.C                                  <-- source code file

EXE = $(FOAM_USER_APPBIN)/cppSandbox          <-- compiled executable
      \_____directory path_____/ \_file name_/
```

- Compile with wmake

```
>> wmake
```

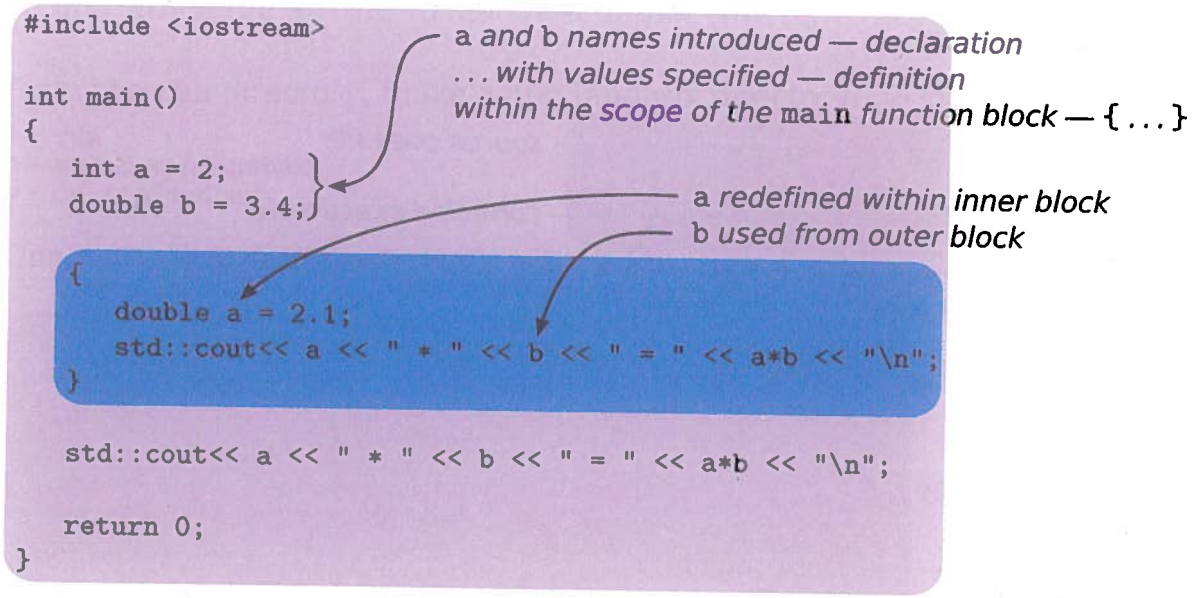
- Run the application

```
>> cppSandbox
2 * 3.4 = 6.8      <-- from cout<<...
```

- Check the return value

```
>> echo $?
0      <-- from return (0);
```

- Task: add an additional calculation of $a \times b$ with $a = 2.1$
- Add an inner block to `cppSandbox.C` and recompile




```
#include <iostream>

int main()
{
    int a = 2;
    double b = 3.4;
    {
        double a = 2.1;
        std::cout<< a << " * " << b << " = " << a*b << "\n";
    }

    std::cout<< a << " * " << b << " = " << a*b << "\n";

    return 0;
}
```

- a, b are objects
- All objects have type: `int` = integer; `double` = double precision number
- Run the cppSandbox application

- Large software → large amount of code outside scope of `main` function
- Lots of non-local names ⇒ risk of name conflicts
- `namespace` = named scope, *e.g.*

```
namespace myNS
{
   int a;
}

int main()
{
   myNS::a = 2;
   ...
```

names used normally within scope

names use namespace qualifier `::` *when used outside of scope*

- C++ standard library → `std` namespace, *e.g.* `std::cout<<`
- `using` directive avoids qualifiers

```
using namespace std

int main()
{
   ...
   cout<< ...
```

insert directive for `std` *namespace*

remove `std::` *qualifier on* `cout<<` *functions*

- Task: add a `cout<<` statement that writes b^2
- Before `return (0);` line, insert

```
cout<< "sqr(" << b << ") = " << pow(b, 2) << "\n"
```

- Compiler does not recognise the name of the `pow` function

```
cppSandbox.C:16:45: error: 'pow' was not declared in this scope
     cout<< "sqr(" << b << ") = " << pow(b, 2) << "\n";
```

- Before `int main()`, `#include` the header file containing declaration of `pow`

```
#include <math.h>
```
← *Header (`.h`) file for C numerics* `library`

- Header files: mainly declarations so compilers recognise names, *e.g.* of functions
- Compile and run

- Application → source code with `main` function, compiled
- Namespace → named scope of a software package, *e.g.* `std::`, `Foam::`
- Library → module of functionality within package
- Header file → interface, *e.g.* functions, for functionality within library
- Declaration → type and name, *e.g.* `int a;`
- Definition → storage additionally allocated, *e.g.* `a = 2;`
- Online resources:
  LearnCpp.com: `http://www.learncpp.com`

eckerG
O'Reilly.

- We have used C++ types / functions, *e.g.* `int`, `double`, `cout<<`
- OpenFOAM contains ∼3,000 classes (defined types) with ∼50,000 functions
- . . . beginning with enhanced versions of C++ primitive types
- Task: create new foamSandbox application using OpenFOAM types

OpenFOAM header file for streaming I/O

OpenFOAM namespace

`int`→`label` *length selectable*

`double`→`scalar` *precision selectable*

`cout` → `Info` *handles parallel running*

```
#include "IOstreams.H"

using namespace Foam;

int main()
{
    label a = 2;
    scalar b = 3.4;

    Info<< a << " * " << b << " = " << a*b << endl;
    Info<< "sqr(" << b << ") = " << sqr(b) << endl;

    return 0;
}
```

means "end line"

- Compile and run
- Names convey meaning: "cell `label`"; "`scalar` value"; "`Info`rmation"

Env variables for compiler flags.

OpenFOAM core classes | scalar, vector, tensor

- Classes for tensors in 3D Cartesian space
- *e.g.* scalar b, vector $\mathbf{u} = u_i = (u_1, u_2, u_3)$, tensor $\mathbf{T} = T_{ij}$
- Task: instantiate $\mathbf{u} = (1, 2, 3)$, $\mathbf{v} = (2, 3, 1)$ and $\mathbf{T}$

```
vector u(1, 2, 3);
vector v(2, 3, 1);
tensor T(1, 2, 3, 4, 5, 6, 7, 8, 9);
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

- Need `vector.H` to declare `vector`; and `tensor.H`...
- `fieldTypes.H` includes both files, and more

```
#include "fieldTypes.H"
```
← `$FOAM_SRC/OpenFOAM/lnInclude/fieldTypes.H`

- Add some algebraic operations: $\times$, $+$, $\bullet$ (inner product)

```
Info<< "b*u   = " << b*u     << endl;
Info<< "u + v = " << u + v   << endl;
Info<< "u & v = " << (u & v) << endl;
Info<< "T & v = " << (T & v) << endl;
```

$= b \times \mathbf{u}$
$= \mathbf{u} + \mathbf{v}$
$= \mathbf{u} \bullet \mathbf{v}$
$= \mathbf{T} \bullet \mathbf{v}$

& precedes a '<<'
∴ brackets.

- Compile and run

- Task: create a list of Fibonnaci numbers 1, 1, 2, 3, 5, 8, 13, 21, ... $\sim 10^6$
- Create a new `fibonnaci` directory and change into it
- Create a `fibonnaci.C` file

List<Type> *template*
<Type> = *any type*
e.g. `scalar`, `vector`, `List`

Loop until $a_{n-1} < 10^6$

Appends new value
$a_n = a_{n-1} + a_{n-2}$

```
#include "IOstreams.H"
#include "List.H"

using namespace Foam;

int main()
{
    List<label> a(2, 1);

    while (a.last() < 1e6)
    {
        a.append(a.last() + a[a.size() - 2]);
    }
    Info << a << endl;

    return 0;
}
```

- Create `Make/files` and `Make/options`, compile and run
- What happens if we increase 10^6? Replace `label`→`scalar`?

emacs with github.

- Task: obtain list of 2× Fibonnaci numbers, *i.e.* 2, 2, 4, 6, 10, 16, 26, 42, ...
- Before `Info<<`: loop over `List`, doubling each element

```
forAll(a, i)
{
    a[i] *= 2;
}
```

`forAll` *macro function: shorthand for*
`for (int i=0; i<a.size(); i++)`

- Compile and run → works, but there is a better alternative...
- `Field<Type>`: derived from `List` class, but with additional field algebra
- ⇒ replace `List<Type>` → `Field<Type>` (including `List.H` → `Field.H`)
- Compiles and runs ⇒ `Field` inherits `List` functionality, behaviour
- Use field algebra: delete the `forAll` loop and change `Info<<`

```
Info<< 2*a << endl;
```

- We need to handle text in programming
- C++ has classes `char` (character) and `string`, *e.g.* `"hello"`
- OpenFOAM has `word` class → string without ␣ / \ ; { } [ ]
- . . . good input/output handling
- Add a simple word and include in `Info` statement

```
word mathematician = "Fibonacci";

Info<< "The series of 2x " << mathematician
    << " numbers is:" <<  nl << 2*a << endl;
```

inserts newline character `"\n"`

- Tensor classes → `scalar`, `vector`, `tensor`, `symmTensor`, `sphericalTensor`
- Tensor algebra → `b*v`, `u + v`, `u & v`, `T & v`
- `List<Type>` → template (class)
- `forAll` → loop over `List`
- `Field<Type>` → `List` with field algebra
- `Field` is derived class of `List` base class
- Inheritance → `Field` has functionality/behaviour of `List`
- Text → `string` base class, `word` derived class

- We have used OpenFOAM classes and functions — where do they come from?
- Task: learn to find classes, functions, *etc.*
- Programmer's "Doxygen" Documentation — `http://cpp.openfoam.org`
- Open in browser: search for "Field"

`Field<Type> a(2, 1);` → *constructor*

`a.last()` → `Type`

`a.append(...Type...)`

`a.size()` → `label`

- Very important for managing complexity
- Self-describing code → good naming and code structure — not comments
- `typedef` simplifies names — especially with template classes
- *e.g.* `Field<scalar>` is clumsy for "scalar field"
- `typedef` searching not effective in Doxygen; command line is better

```
>> find $FOAM_SRC -type f | xargs grep -E 'typedef *Field<scalar>'
```

find in `src` *dir* — *all files* — → — *lines with* `typedef...Field<scalar>`

`-E` = *regular expressions;* `␣*` = *0+ spaces*

- Modify fibonnaci to make it easier to read

```
#include "IOstreams.H"
#include "Field.H"
#include "primitiveFields.H"

using namespace Foam;

int main()
{
    labelField a(2, 1);
    ...
```

← `scalarField.H`, `vectorField.H`, *etc. all included from* `primitiveFields.H` — *reduces the number of header files*

- Standard style is essential for avoiding errors, easier analysis, maintainability
- OpenFOAM code style guide: `http://openfoam.org/dev/coding-style-guide`
- In particular, code indentation avoids errors
- OpenFOAM uses Allman-style indentation with 4 spaces, NOT Tabs
  `https://en.wikipedia.org/wiki/Indent_style#Allman_style`
- For example, a conditional statement

```
if (condition)
{
    code;
}
```

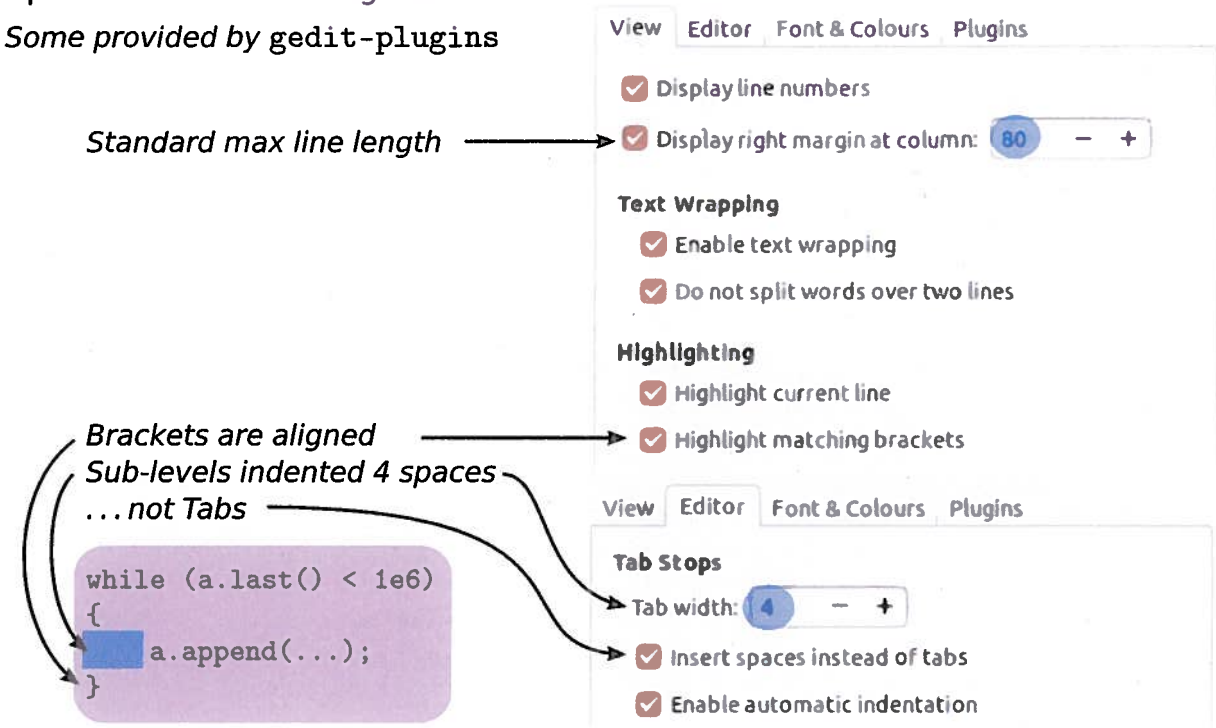
- Also, variable assignment and function handling

```
variableName =
    longClassName:longFunctionName
    (
        longArgument1,
        longArgument2
    );
```

- Max 80 character lines; avoid trailing whitespace
- `a + b, a - b, a*b, a/b`

- Requires good, configured text editor, *e.g.* gedit, emacs, atom, vim
- Open `Preferences` in gedit

  Some provided by `gedit-plugins`



- OpenFOAM contains scripts to create new code: foamNew...
- Provide a consistent framework for development
- Task: set up new application called setBubble — to be used later
- Run foamNewApp to create setBubble; compile the code

```
>> run
>> foamNewApp setBubble
>> wmake setBubble
```

- View setBubble.C

single header file that includes header files for common classes

arguments passed to program e.g. setBubble -help

creates argList *named* args
checks args *against valid args/options*

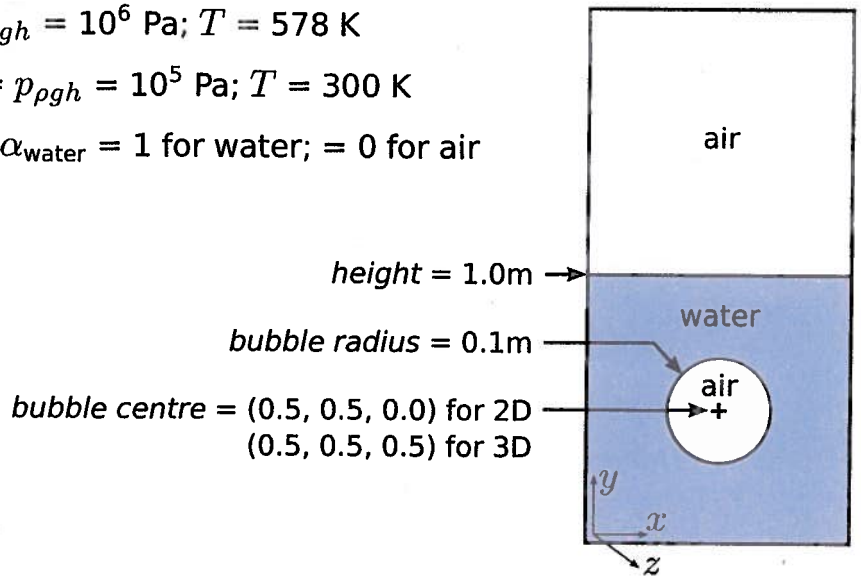
```
#include "fvCFD.H"

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

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

    Info<< "End\n" << endl;
    return 0;
}
```

- Communication is essential for code maintainability
- Files should be documented in the header
- The `Description` section provides space for this
- Complete the `Description` after the code is written

- Doxygen documentation → lists of classes & functions
- . . . especially good for providing inherited functions
- Code readability → good naming, use of `typedef`
- Programming style → indentation reduces errors, consistency aids analysis
- Good text editor → configure to support style
- foamNew. . . scripts → good templates that conform to style

- Some simulations need non-uniform initialisation of fields, *e.g.* p, T
- *e.g.* `depthCharge2D` and `...3D` tutorial cases for `compressibleInterFoam`
- Task: program `setBubble` to initialise an air bubble and air/water interface

- Bubble: $p = p_{\rho gh} = 10^6$ Pa; $T = 578$ K
- Elsewhere: $p = p_{\rho gh} = 10^5$ Pa; $T = 300$ K
- Phase fraction $\alpha_{\text{water}} = 1$ for water; $= 0$ for air



- `Time` class: database (`objectRegistry`) and time data (*e.g.* time step)
- Code template includes `createTime.H`, a short piece of code — in effect:

```
Time runTime("controlDict", args);
```

class *object* *read file* *case info*

← *just a heavily reused piece of code NOT a class declaration (header) file*

- After `Time` is created, data is read from file, starting with mesh, then fields

- `fvMesh` class: mesh data and functions, for finite volume calculations
- Conveniently added by including `createMesh.H`; add at line 42

```
42 #include "createMesh.H"
```

line number

class *object*

```
fvMesh mesh
(
    IOobject
    (
        "region0",
        runTime.timeName(),
        runTime,
        IOobject::MUST_READ
    )
);
```

name of registered object
directory data is read from
object registry
data must be read in

- `GeometricField` template class: field (*e.g.* p, T), including BCs, dimensions
- Create a local `createFields.H` file and include it in `setBubble.C`

```
43 #include "createFields.H"
```

- We need field creation code ⇒ copy an existing `createFields.H` file, *e.g.*

```
>> cp $FOAM_UTILITIES/preProcessing/boxTurb/createFields.H .
```

- Modify `createFields.H` to read in pressure field `p`

typedef for `GeometricField` *of scalars defined at cell centres* (`vol`)

e.g. reads from `0/p` *file*: *file/object name*, *time directory*, *object registry*, *must read data*, *"automatic" writing*

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

- Compile the application

- Duplicate the entire entry for `p` 3 times in `createFields.H`
- Replace `p` with `p_rgh` (3 occurences) in the first duplicate, `T` in the second
- ...and `alpha.water` in the last duplicate, but name the object just `alpha`

```
Info<< "Reading field alpha.water\n" << endl;
volScalarField alpha
(
    IOobject
    (
        "alpha.water",
        ...
```

- Task: initialise `alpha` field to 1 in all cells by "`alpha = ...`"
- Fields contain a `dimensionSet`: dimensional units `[M L T K Q C L]`
- Field algebra does dimension checking on left/right of +, −, =
- ...so generally we need dimensions on right of `alpha = ...`

- `dimensioned<Type>` class → <Type> (scalar, vector, . . .) with dimensions
- Data → `word`, `dimensionSet`, <Type>

```
56 alpha = dimensionedScalar("alpha", dimensionSet(0, 0, 0, 0, 0), 1.0);
```

word dimensionSet scalar

- Here, `dimensionedScalar` is constructed "on-the-fly"

```
class1 object1(args);
object2 = object1;
```

→

```
object2 = class1(args);
```

- For readability → predefined dimension sets with familiar names
- . . . `extern` (global) variables, *e.g.* `dimless`, `dimMass`, `dimPressure`
  see `$FOAM_SRC/OpenFOAM/dimensionSet/dimensionSets.C`

modify

```
56 alpha = dimensionedScalar("alpha", dimless, 1.0);
```

- Compile again

- Construct `centre` = (0.5, 0.5, 0)
- `centre` is fixed: construct with `const` to restrict modifying it (by mistake)

```
58 const point centre(0.5, 0.5, 0);
```

`typedef` *for* `vector`

construct `const`

- Construct `radius` = 0.1, `height` = 1.0, *e.g.*

```
59 const scalar radius = 0.1;
```

generally use assignment (=) *for primitives, instead of* `"radius(0.1);"`

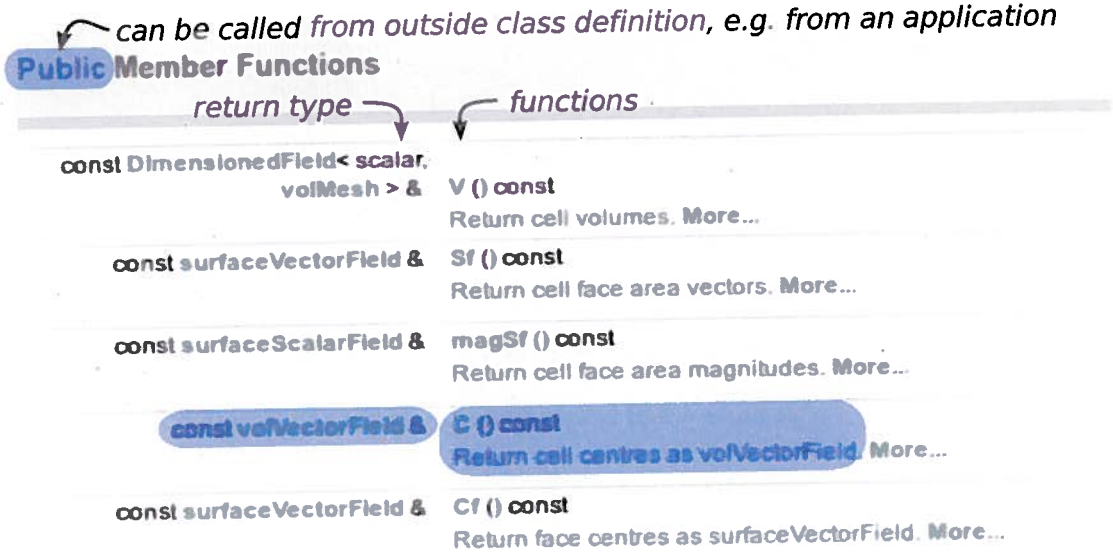
- Construct bubble pressure pB = 10^6, TB = 578
- Compile again; warnings issued for unused primitives — that's OK

```
createFields.H:59:14: warning: unused variable 'radius' [-Wunused-variable]
 const scalar radius(0.1);
```

- `Time` → object registry and time data
- `fvMesh` → mesh for finite volume calculations
- `vol<Type>Field` → fields with values at cell centres
- `dimensioned<Type>` → scalar, vector, . . . with dimensions

- `const` → for any object that will not be modified
- `scalar`, `label` → construct using =, *e.g.* "`label i = 0;`"
- `typedef` → use to improve understanding

- Classes: defined entities with data and functions
- Object: *e.g.* `mesh` → construction of a particular class, *e.g.* `fvMesh`
- Object contains data → how do we access it?
- Task: setting $\alpha_{water} = 0$ inside bubble and above interface
- ⇒ for every cell, check if centre $\mathbf{C}$ is inside bubble or above interface
- ⇒ need $\mathbf{C}$ data → part of `mesh` → look at `fvMesh` class → Doxygen



- `C() function`: returns cell centres — a set of vectors
- `volVectorField`: vectors defined at cell centres

- We want to access cell centre data already calculated and stored
- ... with a reference to the data → like a pointer, but not reallocatable, so safer
- ... equivalent to symbolic link or shortcut in a filing system

- Create a const reference, so data modification is restricted

```
45 const volVectorField& C = mesh.C();
```

& means "reference"

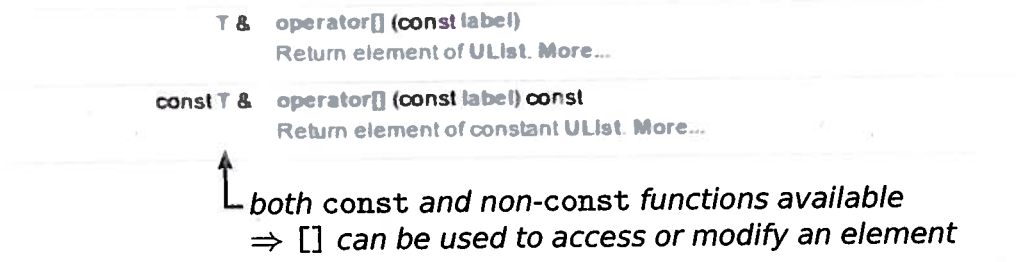
- Now start looping over cells to set values of `alpha`

```
47 forAll(alpha, celli)
48 {
```

loop index

- Now what? Try writing the code to set `alpha` in bubble and above interface

- `UList` elements can be accessed with the array subscript `operator[]`



- ⇒ cell centre values accessed by "`C[celli]`"; create a reference...

```
49 const vector& Ci = C[celli];
```

- Why? 1) Need the data twice → more efficient to call the function once
- ...2) Ensures the specific `const` version of the function
- ...3) Improves readability

- Write the code to set alpha

```
47 forAll(alpha, celli)
48 {
49     const vector& Ci = C[celli];
50
51     if (mag(Ci - centre) <= radius)
52     {
53         alpha[celli] = 0;
54     }
55     else if (Ci.y() > height)
56     {
57         alpha[celli] = 0;
58     }
59 }
```

Global function of style `function(arg1, arg2, ...)` `mag()` *calculates magnitude*

Member access function of style `object.function()`

relational operator

assignment operator

- Global functions: commonly used for maths operations
  see `$FOAM_SRC/OpenFOAM/primitives/VectorSpace/VectorSpaceI.H`
- Operator functions: `http://wikipedia.org/wiki/Operators_in_C_and_C++`
- Function overloading: single function/operator name, defined multiple times
- ... operates differently depending on context, *e.g.* for `operator&`
  `(bool && bool)` = *AND operation;* `(tensor && tensor)` = *double inner product* $\mathbf{T} : \mathbf{T}$

- alpha has been inititialised; we now need to write data into `0/alpha.water`

```
61 alpha.write();
```
← *writes to location according to* `IOobject`
dir: `runTime.timeName()`; *file:* `"alpha.water"`

- Compile; test the application on the `depthCharge2D` case

```
>> run
>> cp -r $FOAM_TUTORIALS/multiphase/compressibleInterFoam/laminar/depthCharge2D .
>> cd depthCharge2D
>> blockMesh
>> cp -r 0.org 0
>> setBubble
```

- Check `0/alpha.water` in ParaView
- Set `p = pB` and `T = TB` in bubble; and `p_rgh = p`

```
54 p[celli] = pB;
55 T[celli] = TB;
```

```
62 p_rgh = p;
```

- Write out alpha, p, T, p_rgh; instead of 4 `write()`, replace `alpha.write()` by

```
64 runTime.writeNow();
```
← *writes all objects registered to* `runTime`
with `IOobject::AUTO_WRITE` *option*

- Access functions → *e.g.* `mesh.C()`
- Reference → (`&`) points to existing data stored in memory
- `const` declaration → restricts modification of data
- Global functions → *e.g.* `mag(C)`, often used for better style
- Function overloading → *e.g.* `alpha.write()` writes a field
- `...runTime.write()` writes database (more to follow...)

- Input data is currently hard coded → inflexible; bad practice
- Task: modify the code to read data from file
- Case properties are in `constant/...Properties` files — a `dictionary`
- Plan: read input parameters from `constant/bubbleProperties`
- In `createFields.H`, construct `IOdictionary`: registered dictionary object
- Start at line 1 because nothing can happen without this data
- Copy/paste a sample of similar code, *e.g.* lines 1-13 of `$FOAM_SOLVERS/incompressible/icoFoam/createFields.H`

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

file/object name → `"bubbleProperties"`,
`constant` *directory* → `runTime.constant()`,
object registry → `mesh`,
must read data → `IOobject::MUST_READ`,
no writing → `IOobject::NO_WRITE`

- OpenFOAM uses dictionary I/O format with keyword lookup
- Test the lookup of `centre` parameter
- Copy `turbulenceProperties` to create `bubbleProperties` in `depthCharge2D`
- Delete existing entry and add `centre` entry

```
18 centre  (0.5 0.5 0);
```
← *note: input data, not code → no commas*
data is delimited by spaces

- Data is looked up with the `lookup("keyword")` function; add

```
14 const point centre
15 (
16     bubbleProperties.lookup("centre")
17 );
```
→ `centre  (0.5 0.5 0);`
← `IOdictionary` *in memory*

- Remove old constructor of `centre` (line ~75); recompile
- Test again; view in ParaView; move the `centre`

- Task: add code for other properties
- Some properties relate to bubble, others to liquid (*e.g.* height level)
- To lookup `centre`, construct a reference to the `bubble` sub-dictionary

```
14 const dictionary& bubbleDict(bubbleProperties.subDict("bubble"));
```

- Create sub-dictionaries for `bubble` and `liquid` in `bubbleProperties`
- Modify the `centre lookup()`

```
15 const point centre
16 (
17     bubbleDict.lookup("centre")
18 );
```

- Remove the old `centre` constructor
- Compile and test

include the liquid properties also

```
bubble
{
    centre (0.5 0.5 0);
    radius 0.1;
    p      1e6;
    T      578;
}

liquid
{
    height 1.0;
    p      1e5;
    T      300;
}
```

- `lookup("chars")` recognises data type based on form of `"chars"`
- *e.g.* `(0 1 2)` is a `vector`
- Cannot distinguish `scalar` and `label`, *e.g.* "32"
- ...so need type conversion functions `readScalar(...)`, `readLabel(...)`
- Add code to lookup other entries for `bubble`

```
19 const scalar radius(readScalar(bubbleDict.lookup("radius")));
20 const scalar pB(readScalar(bubbleDict.lookup("p")));
21 const scalar TB(readScalar(bubbleDict.lookup("T")));
```

- Your tasks:
- Remove the redundant constructors for `radius`, `pB`, `TB`
- Create `liquid` sub-dictionary and lookup `height`, `pL`, `TL` (liquid pressure, temp)
- Assign `pL` to `p` and `p_rgh`; and `TL` to `T`
- Compile and test

- We have the assignment of one GeometricField to another with
  `p_rgh = p;`
- This"assigns values of `p` to `p_rgh`" in cells
- ...but what happens on boundary patch faces?

- Answer: values are assigned on some BCs, but not on others
- *e.g.* values are not assigned on `fixedValue`, the BC is unchanged
- ...because the `fixedValue` code has no `operator=` function
- ...but it does have an `operator==` function that sets equivalent values

- $\Rightarrow$ values on `fixedValue` BCs can be over-ridden by:
  `p_rgh == p;`

- Execution parameters can be provided through command line
- `argList` class manages mandatory arguments and options
- Includes default options, *e.g.* `-case`; options can be added and removed

- Task: add option to prevent overwriting original fields by incrementing time
- Add a bool option to `argList::validOptions` static data member
- ... before `args` is constructed (`setRootCase.H`)

```
37 argList::addBoolOption
38 (
39     "noOverwrite",
40     "increment time to avoid overwriting initial fields"
41 );
```

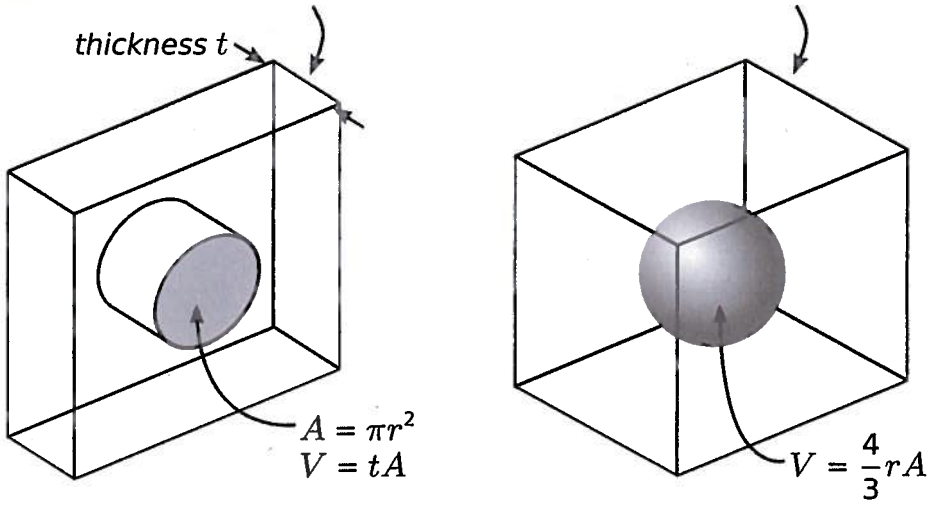
- Increment time before writing data if `noOverwrite` option selected

```
69 if (args.optionFound("noOverwrite"))
70 {
71     runTime++;
72 }
```

- Compile and test

- `IOdictionary` → reads dictionary and registers object
- `lookup()` → looks up data from dictionary
- `subDict()` → references a sub-dictionary
- `readScalar()/readLabel()` → needed for `scalar/label` lookup
- `dimensioned<Type>` → think "does this assignment need dimensions?"
- `argList` → stores configurable arguments and options

- setBubble utility initialises a bubble by setting $\alpha_{\text{water}} = 0$ in cells
- Task: compare numerical bubble volume with analytical calculation
- Assumption: 2D $\rightarrow$ 'cylindrical' bubble; 3D $\rightarrow$ spherical bubble



- Which class provides cell volumes, 2D/3D information (maybe?), thickness t, *etc.*?
- Open documentation in Doxygen

- Cell volumes available by the following function, see `fvMesh.H` / page 31

```
294 //- Return cell volumes
295 const DimensionedField<scalar, volMesh>& V() const;
```

return type

`const` *function: does not modify private data*

- Create a reference to cell volumes and define `bubbleVolume`

```
51 const DimensionedField<scalar, volMesh>& V = mesh.V();
52 scalar bubbleVolume = 0.0;
```

only internal.

- Increment `bubbleVolume` by `V[celli]` for cells within the bubble

```
63 bubbleVolume += V[celli];
```

`a += b;` → "*increase* `a` *by* `b`"
`a = a + b;` → "*take* `a`, *add* `b`, *assign back to* `a`"

- Write `bubbleVolume` to standard output

```
71 Info<< "bubbleVolume = " << bubbleVolume << endl;
```

- Compile and test

- Test in parallel on 2 processors with `(1 2 1)` and `(2 1 1)` decompositions
- Copy the `depthCharge3D` case locally and use its `decomposeParDict` file
- Run using `mpirun`

```
>> mpirun -np 2 setBubble -parallel
```

- `bubbleVolume` = 0.0065 for `(1 2 1)`; = 0.00325 for `(2 1 1)` Why?

only shows processor 0 doesn't parallelise well.

- `sum/min/max` functions operate on individual domains
- `reduce` function → reduction operation across all domain

```
71 reduce(bubbleVolume, sumOp<scalar>());
```

More examples...

```
>> find $FOAM_SRC -type f | xargs grep -h "Op<"
```

- Compile and test
- Fields include `global` operators that embed reductions, *e.g.* gSum, gMax, *e.g.*

```
Info<< "Domain volume = " << gSum(V) << endl;
```

see `$FOAM_SRC/OpenFOAM/fields/Fields/Field/FieldFunctions.C`

- Search "dimension" in Doxygen `fvMesh` to locate functions determining 2D/3D

polyMesh.H

```
442 //- Return the vector of solved-for directions in mesh.
443 const Vector<label>& solutionD() const;
444
445 //- Return the number of valid solved-for dimensions in the mesh
446 label nSolutionD() const;
```

- Implement basic `switch` statement to calculate input:actual ratio bubble volume

```
73 scalar bubbleRatio =
74     constant::mathematical::pi*sqr(radius)/bubbleVolume;
75
76 switch (mesh.nSolutionD())
77 {
78     case 2:
79     {
80         bubbleRatio *= 0.2; // Hard-coded thickness
81         break;
82     }
83     case 3:
84     {
85         bubbleRatio *= 4.0*radius/3.0;
86         break;
87     }
88 }
```

$= A/V_B$

constants defined in various namespaces in `$FOAM_SRC/OpenFOAM/global/constants`

$= tA/V_B$

$= \frac{4}{3} r\, A/V_B$

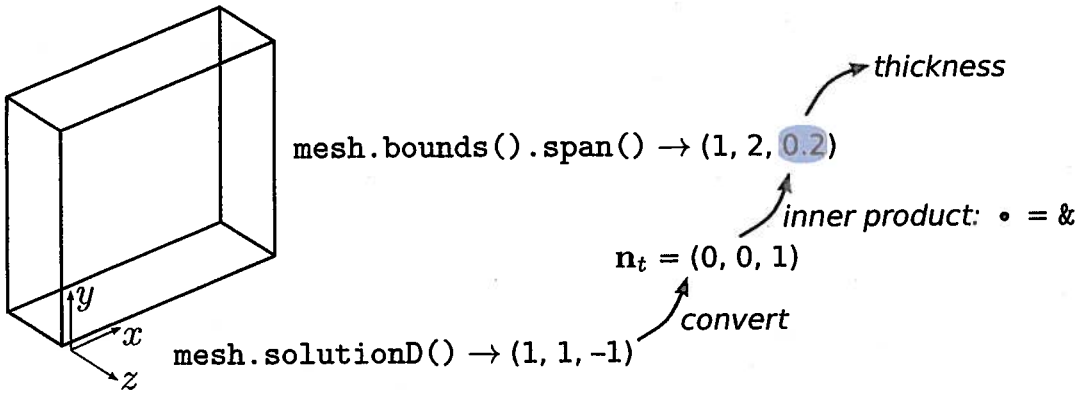
- Always include a `default` in a `switch` statement
- `FatalErrorInFunction` provides location of error in messaging
- ...and causes executable to terminate

```
76 switch (mesh.nSolutionD())
77 {
   ...
88    default:
89    {
90       FatalErrorInFunction
91           << "Case does not appear to be 2D or 3D"
92           << exit(FatalError);
93    }
94 }
```

- Report `bubbleRatio` and `bubbleVolume` in `Info` statement

```
95 Info<< "Specified bubble volume = " << bubbleRatio
96     << " x initialised volume (= " << bubbleVolume << " m^3)" << endl;
```

- Task: calculate thickness instead of hard-coding it
- Tools: `solutionD()` returns a `Vector<label>`
- ...with component 1 for solved direction, -1 for non-solved
- `bounds()` returns `BoundBox`, then `span()` returns `vector` (≡ `Vector<scalar>`)



- Replace bubbleVolume scaling for 2D with:

```
80 const vector nT
81 (
82     0.5*vector(Vector<label>::one - mesh.solutionD())
83 );
84 bubbleRatio *= (nT & mesh.bounds().span());
```

Vector<label> *type:* ***explicitly converted to*** `vector`, *i.e.* `Vector<scalar>`

Vector is three components.

Vector == Vector<scalar>

- Task: adjust bubble temperature and pressure for error in volume
- ...assuming isentropic expansion; delete assignments to pB, TB, lines 61-62
- In the first loop, build a shortened list of cells for the bubble
- Construct a DynamicList<label> named bubbleCells

```
53 DynamicList<label> bubbleCells;
```

← DynamicList: *efficient memory allocation by doubling its size when needed*

- In the loop, append each bubble cell to bubbleCells list

```
63 bubbleCells.append(celli);
```

- Scale TB, pB assuming isentropic expansion

namespace qualifier, ~~Octave~~ on scalar functions

```
102 const scalar gamma = 1.4;
103 pB *= Foam::pow(bubbleRatio, gamma);
104 TB *= Foam::pow(bubbleRatio, (gamma - 1));
```

- Compile and debug
- Write the loop to set p and T in the bubble; and set p_rgh = p
- Compile and test

Think in terms of vector algebra.

- `a += b` → compound assignment "increase a by b"
- `reduce(a, sumOp<scalar>())` → reduction operation across parallel domains
- `gSum(a)` → field operation with built in reduction
- `switch` statement → useful conditional for options
- `FatalErrorInFunction` → error message which causes code to terminate
- `vector(Vector<label>)` → explicit type conversion to `vector`
- `DynamicList` → efficient extensible list class

- OpenFOAM includes a library of strain-rate dependent viscosity models:
  `$FOAM_SRC/transportModels/incompressible/viscosityModels`
- Models are documented in the User Guide: `http://cfd.tips/frhe`
- Task: add the Cross model specified as follows:
  `https://en.wikipedia.org/wiki/Cross_fluid`

$$\nu = \frac{\nu_0}{1 + (k\dot{\gamma})^{1-n}}$$

  where: $\dot{\gamma}$ = strain rate; μ_0, k are model parameters

- Similar to the existing `CrossPowerLaw` model:
  `$FOAM_SRC/transportModels/incompressible/viscosityModels/CrossPowerLaw`
- Each model is specified within its own class (`.C` and `.H` files)
- ⇒ We need to create a new class — named `Cross`

- Create a new `Cross` class by 'cloning' `CrossPowerLaw`
- Copy the `CrossPowerLaw` directory, renaming it `Cross`

```
>> run
>> MODELS=$FOAM_SRC/transportModels/incompressible/viscosityModels
>> cp -r $MODELS/CrossPowerLaw Cross
```

- Rename `CrossPowerLaw.C` → `Cross.C`; `CrossPowerLaw.H` → `Cross.H`

```
>> cd Cross
>> rename 's/CrossPowerLaw/Cross/g' *
```

- Replace the word `CrossPowerLaw` → `Cross` inside both files

```
>> sed -i 's/CrossPowerLaw/Cross/g' *
```

*stream **ed**itor*
*edits files '**i**n place'*
*substitute. . . **g**lobablly (= all instances)*
all files

- Create `Make/files` and `Make/options` files

```
>> wmakeFilesAndOptions
```

- Edit `files`:

```
Cross.C

LIB = $(FOAM_USER_LIBBIN)/libCross
```

source code file
compiled library
output = library
path
`lib` *prefix for library names*

- Compile the library with wmake → compilation error... why?

```
In file included from Cross.C:26:0:
Cross.H:38:28: fatal error: viscosityModel.H: No such file or directory
 #include "viscosityModel.H"
```

- Cross class is derived from a base `viscosityModel` class, see `Cross.H`

```
53 class Cross
54 :
55     public viscosityModel
56 {
```

declares `Cross` *class*

inherits public members (functions, data) from `viscosityModel` *class*

- ⇒ `viscosityModel.H` must be included in `Cross.H` (line 38)
- wmake must locate `viscosityModel.H` to compile
- ⇒ add include directory (`-I`) and link library (`-l`) to `Make/options`

```
EXE_INC = \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/transportModels/incompressible/lnInclude

LIB_LIBS = \
    -lfiniteVolume \
    -lincompressibleTransportModels
```

line continuation

directory of links to all `.H` *files in viscosity models library incl.* `viscosityModel.H`

library with compiled `viscosityModel` *class*

libraries linked to our library

- Modify model parameters ⇒ remove `nuInf`; replace `m` by `k`
- In `Cross.H`:

```
62 dimensionedScalar nuInf_;
63 dimensionedScalar k_;
```

- In `Cross.C`:

```
100 CrossCoeffs_.lookup("nuInf") >> nuInf_;
101 CrossCoeffs_.lookup("k") >> k_;
```

```
70 nuInf_("nuInf", dimViscosity, CrossCoeffs_),
71 k_("k", dimTime, CrossCoeffs_),
```

- Modify `calcNu()` function in `Cross.C`:

```
50 Foam::tmp<Foam::volScalarField>
51 Foam::viscosityModels::Cross::calcNu() const
52 {
53     return nu0_/(scalar(1) + pow(k_*strainRate(), (1 - n_)));
54 }
```

- `strainRate()` is defined in `viscosityModel.C` → `Cross` inherits it

- Test the model on `offsetCylinder` case of nonNewtonianIcoFoam solver

```
>> run
>> cp -r $FOAM_TUTORIALS/incompressible/nonNewtonianIcoFoam/offsetCylinder .
>> cd offsetCylinder
>> blockMesh
```

- Select Cross model in `transportProperties`

```
transportModel  Cross;

CrossCoeffs
{
   nu0      [0 2 -1 0 0 0 0]  0.01;
   k        [0 0  1 0 0 0 0]     2;
   n        [0 0  0 0 0 0 0]   0.5;
}
```

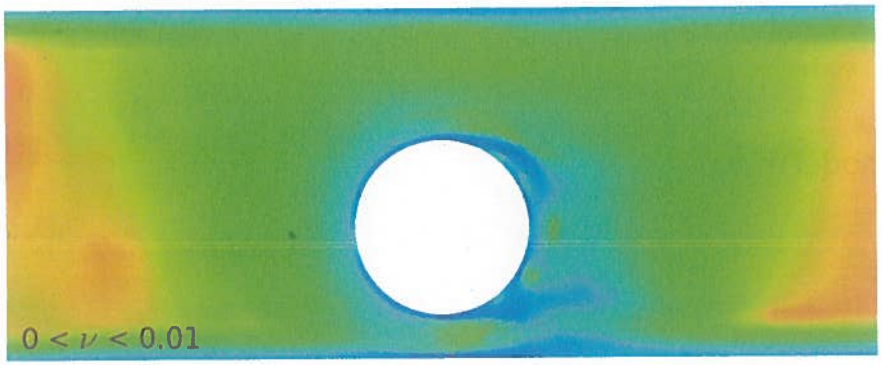
- Load the libCross.so library dynamically at run-time by setting in `controlDict`

```
libs    ( "libCross.so" );
```

requires full name with `.so`

- Run the case and view the nu field in ParaView

- `nu` field shows shear-thinning behaviour for $n < 1$



- Try $n = 1.5 \rightarrow$ code blows up in `calcNu()` function
- $(k\dot{\gamma})^{-0.5} \rightarrow \infty$ (NaN), as $\dot{\gamma} \rightarrow 0$
- $\Rightarrow$ add `SMALL` (`static const scalar`) in `calcNu()`

```
53 return nu0_/(scalar(1) + pow(k_*strainRate() + SMALL, (1 - n_)));
```

see `$FOAM_SRC/OpenFOAM/primitives/Scalar/doubleScalar/doubleScalar.H`
`$FOAM_SRC/OpenFOAM/primitives/Scalar/scalar/scalar.H`

- Recompile and test

- The Cross model is used when selected in `transportProperties`
- How does the application use the model, compiled in a separate library?
- How does the application get the viscosity from the model?

- The interface is revealed by comparing nonNewtonianIcoFoam and icoFoam
- The meld application can display the comparison, *e.g.*

```
>> meld $FOAM_SOLVERS/incompressible/*coFoam/createFields.H
```

`nonNewtonianIcoFoam/createFields.H`

```
34 singlePhaseTransportModel fluid
35 (
36     U,
37     phi
38 );
```

`icoFoam/createFields.H`

```
15 dimensionedScalar nu
16 (
17     "nu",
18     dimViscosity,
19     transportProperties.lookup("nu")
20 );
```

- ⇒ icoFoam constructs `nu` — a single-valued viscosity
- ...nonNewtonianIcoFoam constructs `fluid` — a `singlePhaseTransportModel`

- Compare the nonNewtonianIcoFoam and icoFoam main files

```
>> meld $FOAM_SOLVERS/incompressible/*coFoam/*.C
```

nonNewtonianIcoFoam.C

```
33 #include "singlePhaseTransportModel.H"

59 fluid.correct();

67 - fvm::laplacian(fluid.nu(), U)
```

icoFoam.C

```
64 - fvm::laplacian(nu, U)
```

model updates viscosity with `correct()` *function*

model 'delivers' viscosity with `nu()` *function*

- ⇒ Models are contructed, corrected and delivered
- `singlePhaseTransportModel` is a 'wrapper' around `viscosityModel`
- ...construction of `fluid` → constructs a `viscosityModel` by

```
autoPtr<viscosityModel> fluidPtr
(
    viscosityModel::New("fluid", transportProperties, U, phi)
);

viscosityModel& fluid = fluidPtr();
```

constructs a pointer to `viscosityModel`

'dereferences' pointer so `fluid` ***is*** *the* `viscosityModel`

- nonNewtonianIcoFoam is compiled with `viscosityModel` base class only
- ...but uses `Cross` functions (`nu()`, `correct()`) — through virtual functions
- `viscosityModel` base class includes general features of all viscosity models
- ...including delivery, correction of viscosity → `nu()`, `correct()`

`viscosityModel.H`

```
148 //- Return the strain rate
149 tmp<volScalarField> strainRate() const;
150
151 //- Return the laminar viscosity
152 virtual tmp<volScalarField> nu() const = 0;
...
157 //- Correct the laminar viscosity
158 virtual void correct() = 0;
```

defined in `viscosityModel.C` *as* `sqrt(2.0)*mag(symm(fvc::grad(U_)))`

undefined; defined in model class files e.g. `Cross` ⇒ *declared* `virtual`

- Derived class, *e.g.* `Cross`, defines virtual functions, *e.g.* `Cross.H`

```
111 void correct()
112 {
113     nu_ = calcNu();
114 }
```

`calcNu()` *defined in* `Cross.C`

- Compiler points a virtual function to function defined in derived class

- Model is selected at run-time, *e.g.* through `transportProperties`. How?
- Base class declares/defines a run-time selection 'hash' table
- ... of pointers to constructors, indexed by `typeName`
- Base class has `New` selector function → returns pointer for `typeName`
- Each derived class defines its `typeName` → its model name
- ... and adds its constructor pointer to the table, *e.g.* `Cross.C`

```
36 defineTypeNameAndDebug(Cross, 0);
37
38 addToRunTimeSelectionTable
39 (
40     viscosityModel,
41     Cross,
42     dictionary
43 );
```

macro functions defined in `runTimeSelectionTables.H`

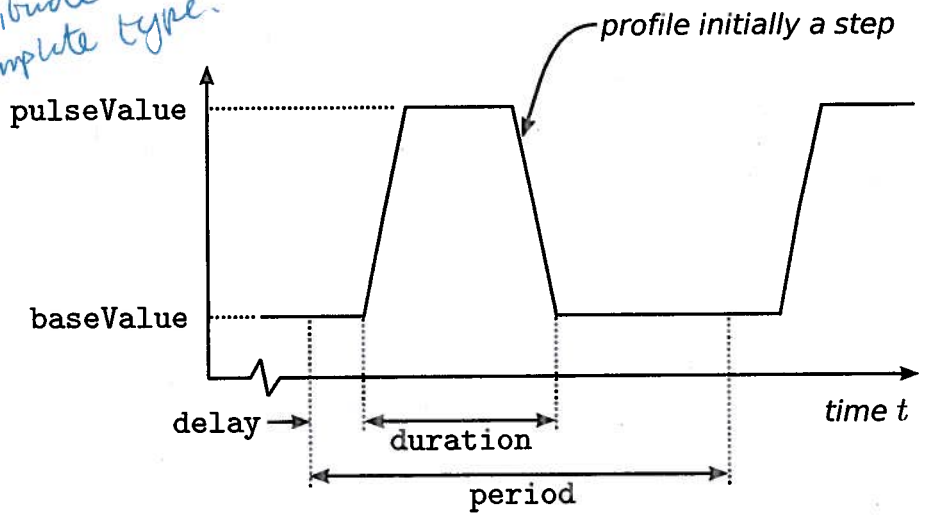
- Hash table constructed at run-time
- ⇒ Classes in libraries linked at run-time (through `libs`) included in hash table

- Models 'plugged into' applications by constructing (pointer to) base class
- Virtual functions defined in base class for use in application
- Virtual functions over-ridden in derived classes
- Application uses functions of the selected model
- Models are selected at run-time through a similar 'virtual' mechanism

- Some inlets are pulsed or time varying, *e.g.* from a valve
- Task: develop boundary condition (BC) for pulse

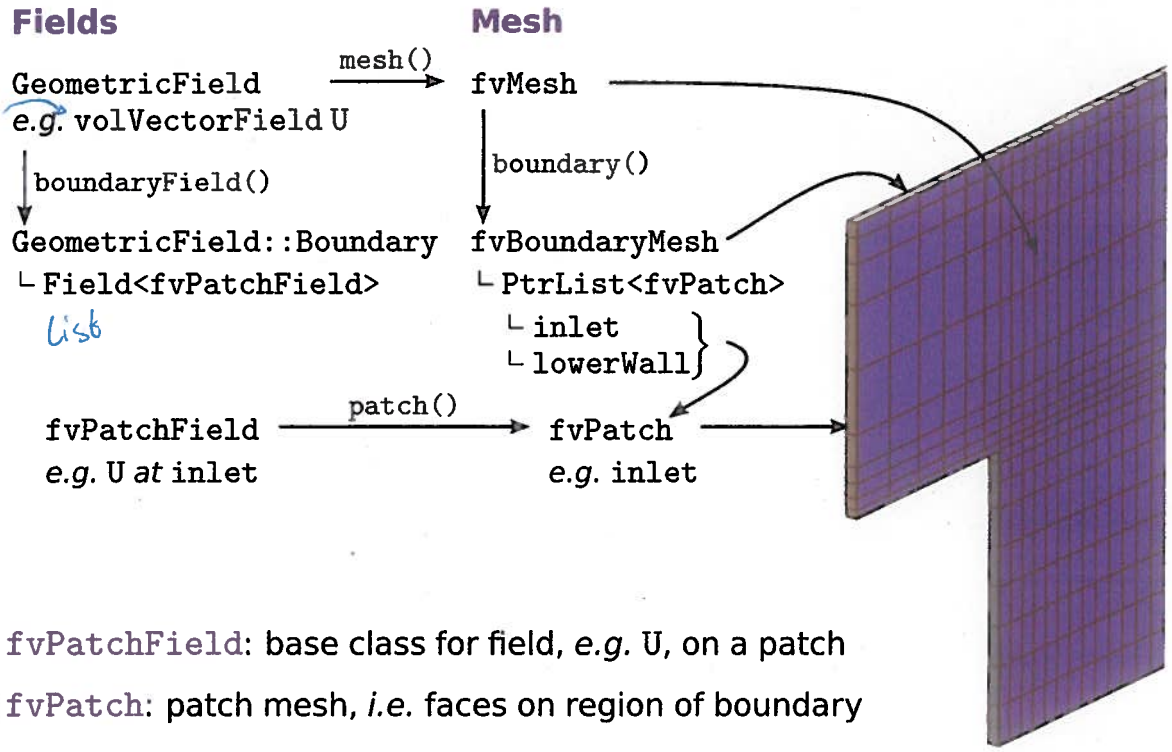
- Define the user inputs
- ... with `duration` and `delay` (optional) specified as a fraction of `period`

amplitude is template type.

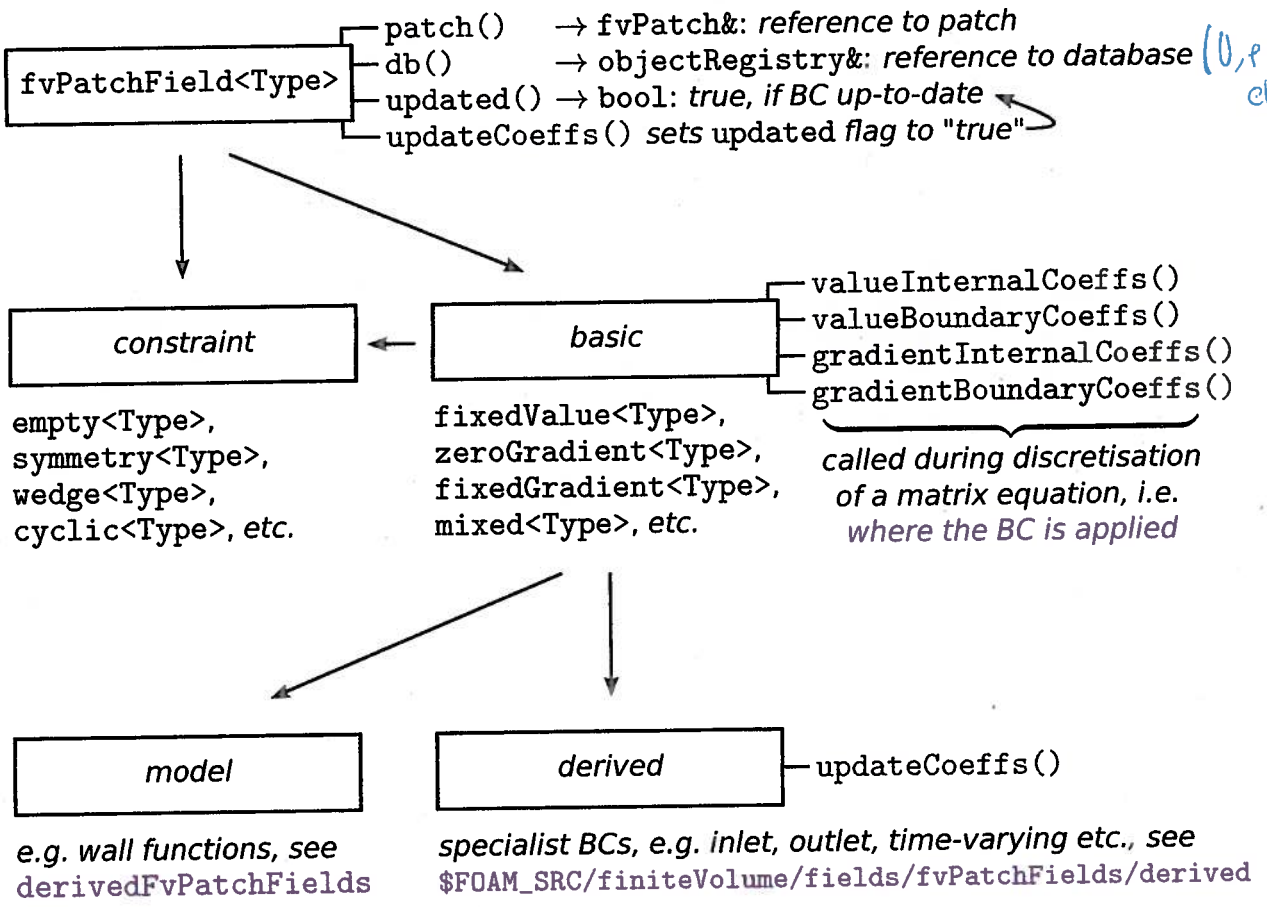


etc/templates
snappyStuff.

- Consider the mesh and boundary of a case, *e.g.* inlet section of `pitzDaily`
- How are BCs stored for fields, *e.g.* at `inlet` for velocity `U`?



- `fvPatchField`: base class for field, *e.g.* `U`, on a patch
- `fvPatch`: patch mesh, *i.e.* faces on region of boundary



- New class → unique name
- Derived typically from `fixedValue` (simplest) or `mixed` (advanced)
- BC function, *e.g.* $\mathbf{Q} = \mathbf{Q}_0(1 + At)$ implemented in `updateCoeffs()`
- Model coefficients, *e.g.* $\mathbf{Q}_0$, A → private data
- Case data, *e.g.* time t → accessed by `patch()`, `db()` functions
- The foamNewBC script creates BC class files
- Create the new BC named `pulseFixedValue` as follows

```
>> run
>> foamNewBC -fixedValue -all pulseFixedValue
>> cd pulseFixedValue
>> wmake
```

`-fixedValue` option → derived from `fixedValue`

`-all` option → for all types: `scalar` (*e.g.* p), `vector` (*e.g.* $\mathbf{U}$), `tensor`, *etc.*

- Compiles into a library named libpulseFixedValue.so

- Modify template BC code in `pulseFixedValueFvPatchField.H` / `.C` files
- Open `pulseFixedValueFvPatchField.H` in an editor
- Lines 99-122 provide common types of private data, *e.g.* `scalar`, `word`, *etc.*

- Model parameters: `period`, `duration`, `pulseValue`, *etc.* What types are they?
- First, delete the types we do not need
- Lines 111-122: delete `timeVsData_`, `wordData_`, `labelData_`, `boolData_`

Fields files contain macros to compile with scalar version, vector version etc.

Type generic for scalar, vector etc.

- Constructor functions → creates an object, initializing data
- . . . function name is the same as the class name

- `fvPatchFields` are constructed when a field, *e.g.* `U`, is created
- . . . often by reading data from file, *e.g.* `0/U`,
- . . . but also empty, as a copy, during decomposition, reconstruction, *etc.*
- ⇒ `5` different constructors defined in `pulseFixedValueFvPatchField.C`

- Delete initializations of removed private data in **all 5 constructors** starting with. . .

```
45 pulseFixedValueFvPatchField
46 (
47     const fvPatch& p,
48     const DimensionedField<Type, volMesh>& iF
49 )
50 :
51     fixedValueFvPatchField<Type>(p, iF),
52     scalarData_(0.0),
53     data_(Zero),
54     fieldData_(p.size(), Zero),
55     timeVsData_(),
56     wordData_("wordDefault"),
57     labelData_(-1),
58     boolData_(false)
```

constructor function same name as class

initializations separated by commas (`,`)

`scalarData_` *set to 0*

⋆ *delete final comma* ⋆

deconstruct
reconstruct
copy
null
user specified

- Try compiling; compiler reports a lot of error messages, *e.g.*

```
pulseFixedValueFvPatchField.C:174:21: error: 'timeVsData_' not declared...
```

file name *line number* *yes, removed from the* `.H` *file*

- Delete removed private data from `updateCoeffs()` to enable compilation

```
170 fixedValueFvPatchField<Type>::operator==
171 (
172     data_
173   + fieldData_
174   + scalarData_*timeVsData_->value(t())
175 );
```

- Similarly, delete removed private data from `write()` function
- . . . referring to compilation error messages for line numbers
- Ensure the code compiles without error

- More code → more maintenance ⇒ avoid code duplication
- What if we have common algorithms/function for different types?
- ...*e.g.* addition `operator+(a, b)` for `scalar/vector` → "add components"
- Use generic programming → templates in C++
- `operator+` defined once in `VectorSpaceI.H` for `VectorSpace` template class

```
75 template<class Form, class Cmpt, int nCmpt>
76 class VectorSpace
77 { ...
```

template arguments, i.e. types specified later (in <...>)

- Works for `vector` class → `Cmpt = scalar`, `nCmpt = 3`
- ...`tensor` class → `Cmpt = scalar`, `nCmpt = 9`, `sphericalTensor`, *etc.*
- Templates used extensively in OpenFOAM for `Type = scalar, vector, tensor + 2`
- One set of code for 5 types ⇒ big reduction in duplication

- Many BCs are template classes, *e.g.* `fixedValue` → applicable to all types
- `pulseFixedValue` is a template class → any field, *e.g.* `p`, `U` can be "pulsed"

```
93 template<class Type>
94 class pulseFixedValueFvPatchField
95 :
96     public fixedValueFvPatchField<Type>
97 { ...
```

template argument `Type` : `scalar`, `vector`, `tensor`, *etc.*

derived from `fixedValue` *template class*

- Private data, line 106, `Type`: single scalar, vector, *etc.*
- Let's use this for `baseValue`, which we can default to zero
- Private data, line 110, `Field<Type>` is "set of values on patch faces, of `Type`"
- Let's use this for `pulseValue`, allowing different values on each patch face
- Rename `data_` and `fieldData_` in `.H` file

```
103 //- Base value of the field
104 Type baseValue_;
105
106 //- Value during the pulse
107 Field<Type> pulseValue_;
```

OpenFOAM standard: private data names end with underscore _ to avoid ambiguity with function arguments

- Private data requires initialization in constructors
- ⇒ need to rename `data` and `fieldData` in `.C` file, *e.g.*

```
43 template<class Type>
44 Foam::pulseFixedValueFvPatchField<Type>::
45 pulseFixedValueFvPatchField
46 (
47     const fvPatch& p,
48     const DimensionedField<Type, volMesh>& iF
49 )
50 :
51     fixedValueFvPatchField<Type>(p, iF),
52     scalarData_(0.0),
53     baseValue_(Zero),
54     pulseValue_(p.size(), Zero)
55 {
56 }
```

← *template function*
← *class scope*
← *function name*
← *arguments*
← *initialization follows...*
← *base class initialized*
← *scalar initialized to 0*
← Type *initialized to zero*

- How do we initialize zero for Type → scalar, vector, *etc.* ('0' won't work)
- Use static variable `Zero`, defined for all primitive types
- Compile, ensuring all `data` and `fieldData` entries are updated in `.C` file

- Constructor from `dictionary` (lines 59-84) used when a field is read from file
- For example, in `0/p`, an `inlet` could use `pulseFixedValue` by

```
inlet
{
    type            pulseFixedValue;   ← BC typeName
    pulseValue      uniform 10;        ← Field<Type> ⇒ uniform/nonuniform
    baseValue       0;                 ← Type ⇒ single value
    ...
```

- Default values can be included → used if the keyword entry is omitted
- `baseValue` currently looked up; requires explicit type conversion to Type

```
70 baseValue_(pTraits<Type>(dict.lookup("baseValue"))),
```

like `readScalar`, *converting* `Istream` *(characters) to* `scalar`

gets around scalar vector conversion in readScalar

- Change lookup to include a default of zero

```
70 baseValue_(dict.lookupOrDefault<Type>("baseValue", Zero)),
```

default value

Type *is known* ⇒ *conversion is built-in*

no need for pTraits as lookupOrDefault returns a Type?

pTraits<Type>::zero is built in for arbitrary 0 or (0, 0, 0)

- Add scalar parameters `period` and `duration` by
- ...1. replace `scalarData` with `period` in `.H` and `.C` files
- ...2. duplicate `period` declarations/initializations, renaming `duration`, *e.g.*

```
100 //- Cycle period
101 scalar period_;
102
103 //- Pulse duration as fraction of cycle period
104 scalar duration_;
```

- Terminate run if input does not satisfy $0 < \texttt{duration} < 1$; in `.C` file

```
70     period_(readScalar(dict.lookup("period"))),
71     duration_(readScalar(dict.lookup("duration"))),
       ...
74 {
75     if (duration_ < 0 || duration_ > 1)
76     {
77         FatalErrorInFunction
78             << "duration = " << duration_ << " is invalid (0 < duration < 1)"
79             << exit(FatalError);
80     }
       ...
```

- BC can be specified as $\alpha \times$ `pulseValue` $+(1-\alpha) \times$ `baseValue`
- Create `pulseFraction()` function for α
- $\Rightarrow$ Rename the example function `t()` (current time) $\rightarrow$ `pulseFraction()`

```
114     //- Return pulse fraction
115     scalar pulseFraction() const;
116
117 public:
```

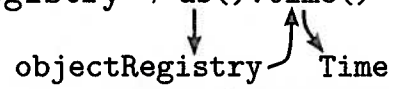
← *declared before* `public` *specifier*
$\Rightarrow$ *private (the default) to class code*

- `pulseFraction()` is a private function $\Rightarrow$ can only be called from in this class
- $\Rightarrow$ For an object constructed from this class named `obj`, you cannot call:
  `obj.pulseFraction()`

this → reserved pointer which will point to something you will create used for templates as its going to be created.

class → template → object

- Define `pulseFraction()` in `.C` file
- The function requires time t — already exists in the example `t()` function
- During a run, time stored by `Time` object on `objectRegistry` ⇒ `db().time()`
- 'db()' → "call db() on object of this class"
- If ambiguous → use reserved `*this` pointer



Function can't change values/data

```
34 template<class Type>
35 Foam::scalar Foam::pulseFixedValueFvPatchField<Type>::pulseFraction() const
36 {
37     const scalar t = this->db().time().timeOutputValue();
38
39     scalar cycleFraction = fmod(t/period_, 1.0);
40     if (cycleFraction > duration_)
41     {
42         return 0.0;
43     }
44     else
45     {
46         return 1.0;
47     }
48 }
```

this->db(): *call* db() *on object of* *this *pointer*

fractional part of t/τ

fmod = floating point modulus.

if you want things in future use db()

- BC expression implemented in `updateCoeffs()` function
- Called when matrix equation is constructed
- Update mechanism avoids multiple calls for a given matrix equation

```
183 template<class Type>
184 void Foam::pulseFixedValueFvPatchField<Type>::updateCoeffs()
185 {
186     if (this->updated())
187     {
188         return;
189     }
190
191     fixedValueFvPatchField<Type>::operator==
192     (
193         baseValue_*(1.0 - pulseFraction())
194       + pulseValue_*pulseFraction()
195     );
196
197     fixedValueFvPatchField<Type>::updateCoeffs();
198 }
```

returns updated_ (true/false)
*this *pointer required because*
updated() *is inherited, class is template*

*this *pointer not required*
pulseFraction() *defined in this class*

sets updated_ = true;

assign to BC's

- Compile, eliminating any errors

sets flag to true so there's no recalculating.

- Test the BC on a basic, transient tutorial case, *e.g.* `pitzDaily`
- Go to the `run` directory and copy the case

```
>> run
>> cp -r $FOAM_TUTORIALS/incompressible/pimpleFoam/pitzDaily .
```

- Edit `controlDict`, linking libpulseFixedValue.so, modifying the following

```
libs            ( "libpulseFixedValue.so" );
endTime         0.1;
deltaT          0.0001;
writeControl    runTime;
writeInterval   0.002;
```

- Activate the `pulseFixedValue` BC in `0/U` as follows:

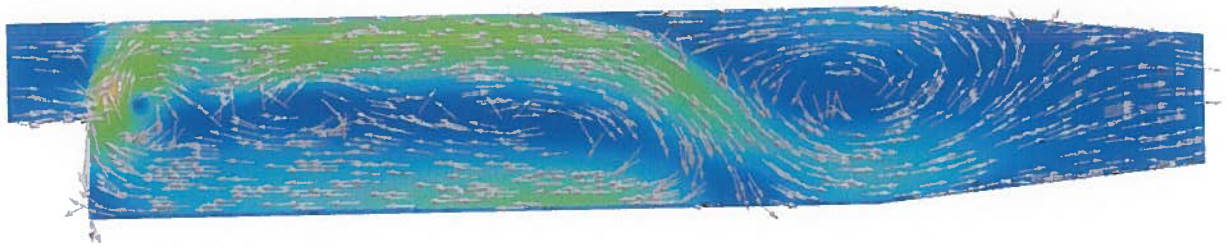
```
inlet
{
    type          pulseFixedValue;   <-- TypeName: see .H file
    period        0.01;              <-- scalar
    duration      0.4;               <-- scalar: 0→1
    pulseValue    uniform (10 0 0);  <-- Field<vector>: requires uniform
    baseValue     (1 0 0);           <-- vector
    value         uniform (10 0 0);  <-- optional, to keep ParaView happy
}
```

- Run the pimpleFoam solver
- Check results manually, *e.g.* strip `inlet` entry from `U` files:

```
>> grep -A 8 inlet 0.004/U
>> grep -A 8 inlet 0.008/U
```

next 8 lines

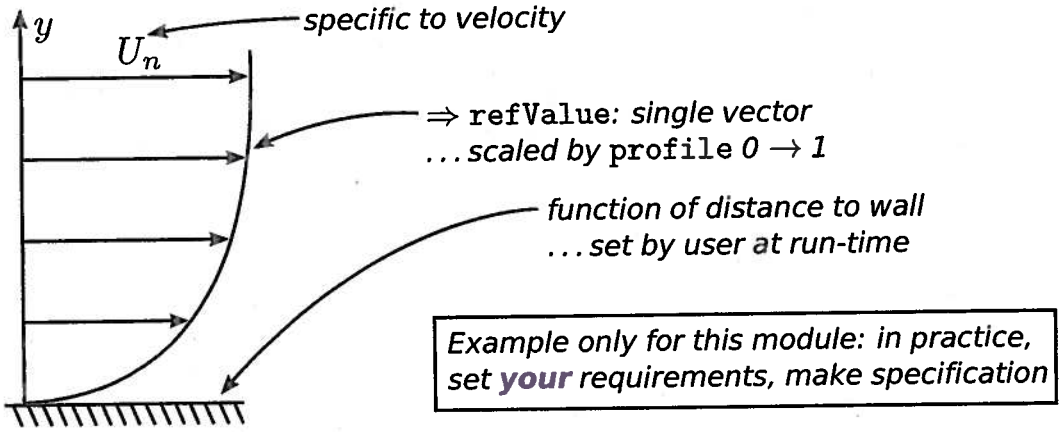
- Examine the simulation in ParaView



- Correct the `Description` section at the top of the `.H` file
- Format of `Description` designed for special typesetting in Doxygen HTML

```
30 Description
31     This boundary condition provides a pulseFixedValue condition,
32     calculated as:
33
34         \f[
35             Q = (f - 1)*baseValue + f*pulseValue
36         \f]
37
38     where
39         f = 1, when fractional part of (time/period) < duration
40         f = 0, otherwise
41
42     \heading Patch usage
43
44     \table
45         Property   | Description                  | Req'd? | Default
46         period     | cycle period [s]             | yes    |
47         duration   | duration, fraction of period | yes    |
48         baseValue  | single Type value            | no     | zero
49         pulseValue | Type field across patch      | yes    |
50     \endtable
```

- Task: create a velocity BC with a spatial profile
- Requirements?: immediate application? future applications? data input?
- ...usability? options? extensibility? maintainability? efficiency?



- Specification: new `fixedValue`, `vector` BC → `profileVelocity`
- ...using `Function1` for run-time function; `wallDist` provides distance to wall

- From the `run` directory, create new BC code
  `foamNewBC -f -v profileVelocity`
  `-f` → derived from `fixedValue`
  `-v` → for `vector` field only
- Open `profileVelocityFvPatchVectorField.H` in an editor
- Model parameters: `data_` → `refValue`; `timeVsData_` → `profile`
- Lines 113-121: delete the types we do not need `wordData_`, `labelData_`, `boolData_`
- Lines 106-109: delete `fieldData_`
- Lines 99-101: delete `scalarData_`

- Edit **all 5 constructors** in the `.C` file, deleting initializations, starting with...

```
43 profileVelocityFvPatchVectorField
44 (
45     const fvPatch& p,
46     const DimensionedField<vector, volMesh>& iF
47 )
48 :
49     fixedValueFvPatchVectorField(p, iF),
50     scalarData_(0.0),
51     data_(Zero),
52     fieldData_(p.size(), Zero),
53     timeVsData_()
54     wordData_("wordDefault"),
55     labelData_(-1),
56     boolData_(false)
```

⋆ *delete final comma* ⋆

- Delete private data in `updateCoeffs()` function to enable compilation

```
156 fixedValueFvPatchVectorField::operator==
157 (
158     data_
159   + timeVsData_->value(t())
160 );
```

- Patch fields can be constructed during decomposition and reconstruction
- Construction of `patchField` data (one value per face) is complex
- ...generally requiring mapping of values onto new `patchField`
- The `autoMap` and `rmap` functions perform the mapping; delete for `fieldData_`

```
124 void Foam::profileVelocityFvPatchVectorField::autoMap
125 (
126     const fvPatchFieldMapper& m
127 )
128 {
129     fixedValueFvPatchVectorField::autoMap(m);
130     fieldData_.autoMap(m);
131 }
```

```
140 fixedValueFvPatchVectorField::rmap(ptf, addr);
141
142 const profileVelocityFvPatchVectorField& tiptf =
143     refCast<const profileVelocityFvPatchVectorField>(ptf);
144
145 fieldData_.rmap(tiptf.fieldData_, addr);
```

- Correct the `write()` function and compile

- The `Function1` class provides a run-time function of single dependent variable
- ... often a function of time t; here distance to wall y
- Rename `timeVsData_` → `profile_`; change to `scalar` function, *e.g.* `.H` file

```
103 //- Profile as function of distance to wall
104 autoPtr<Function1<scalar>> profile_;
```

- Update `.C` file, renaming to `profile_` and ensuring it is a `scalar` function
- Rename `data_` → `refValue_`, *e.g.* `.H` file

```
99 //- Max velocity of the profile
100 vector refValue_;
```

- Update `.C` file, renaming to `profile_`
- ... checking the initializations in dictionary constructor

```
65 refValue_(dict.lookup("refValue")),
66 profile_(Function1<scalar>::New("profile", dict))
```

- `wallDist` class: distance to the wall, a common requirement in CFD
- ...a `MeshObject` → constructed once only if needed by `New()` function

```
static const wallDist& wallDist::New(const fvMesh& mesh)
```

- Distance to the wall accessed from `wallDist` by `y()` function

```
const volScalarField& y() const
```

- Task: create a distance to wall function `y()` on the patch
- ...replacing/reusing the `t()` function
- Use Doxygen → 'chain' access function, try starting with `patch()`

1. Get reference to `mesh` (`fvMesh` object)
2. Use `mesh` reference in `wallDist::New`
3. Get reference to distance to the wall `y` (`volScalarField`) from `wallDist`
4. Return field `y()` → `y` on the patch

- Evaluate the profile function in the `updateCoeffs()` function
  `profile_->value(y())`
- Normalise the profile values (≤ 1)
- Assign the values in the `operator==` function in `updateCoeffs()`
- Compile the library

- Copy the `pitzDaily` case for simpleFoam, renaming `pitzDailyProfile`
- Edit `controlDict`, linking libprofileVelocity.so, modifying the following

```
libs        ( "libprofileVelocity.so" );
```

- Activate the `profileVelocity` BC in `0/U` as follows:

```
inlet
{
    type            profileVelocity;
    refValue        (10 0 0);
    profile         table ( (0 0) (0.008 1) );
    value           uniform (10 0 0);
}
```

profile ramps to 1 at 8mm

- Profile ramps from 0 to 1 at 0.008m from wall
- Run the case and check the profile manually and with ParaView

- Solver application → governing equations, model interface and algorithm code
- ...to simulate particular physics in CFD
- Prototype solver → existing solver + additional physics
- Additional physics later transferred to modelling libraries

- Task: develop an application for laminar flow with region of porous media
- Note: porous media code already exists in `fvOptions`, custom solvers
- Modify code from icoFoam, renaming porousIcoFoam

```
>> run
>> cp -r $FOAM_SOLVERS/incompressible/icoFoam porousIcoFoam
>> cd porousIcoFoam
>> mv icoFoam.C porousIcoFoam.C
```

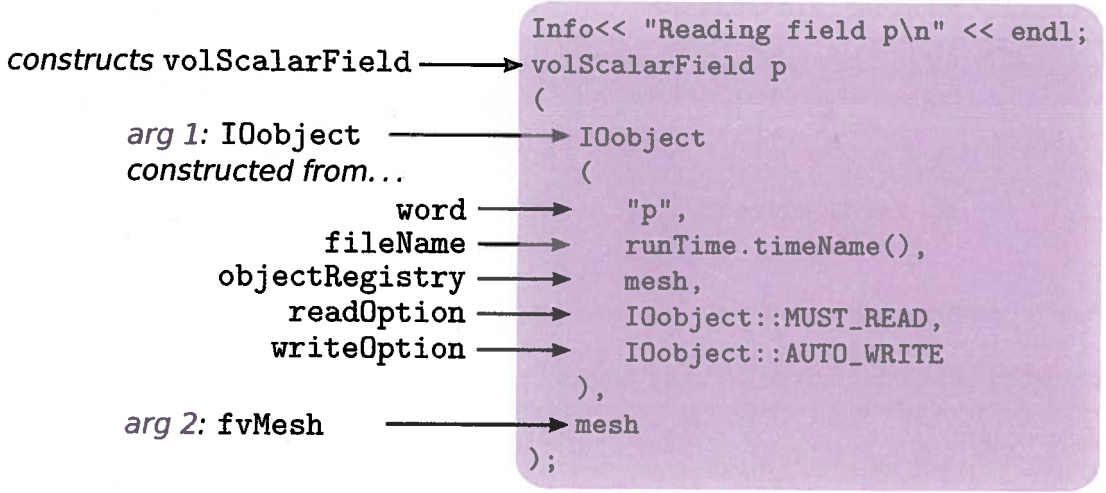
- Edit `Make/files`:

```
porousIcoFoam.C

EXE = $(FOAM_USER_APPBIN)/porousIcoFoam
```

compiles in ~/OpenFOAM/ubuntu-4.1/platforms/.../bin

- Run wmake

- `createFields.H` 1-20 → viscosity `nu` `lookup()` with `IOdictionary` (see sl. 36)
- line 23 → `volScalarField p` is constructed with `IOobject` and `fvMesh`



typeDef

- What is the `volScalarField` class?

```
>> find $FOAM_SRC -type f | xargs grep -h 'volScalarField;'
typedef GeometricField<scalar, fvPatchField, volMesh> volScalarField;
```

actual class name — template with 3 arguments

semi-colon follows name in typedefs

no semi-colon after class declaration → it will be a typedef.

- Example pressure p: `GeometricField<scalar, fvPatchField, volMesh>`

`GeometricField::Boundary` (sub-class)
`FieldField<fvPatchField<scalar>>`
`→PtrList<fvPatchField<scalar>>`

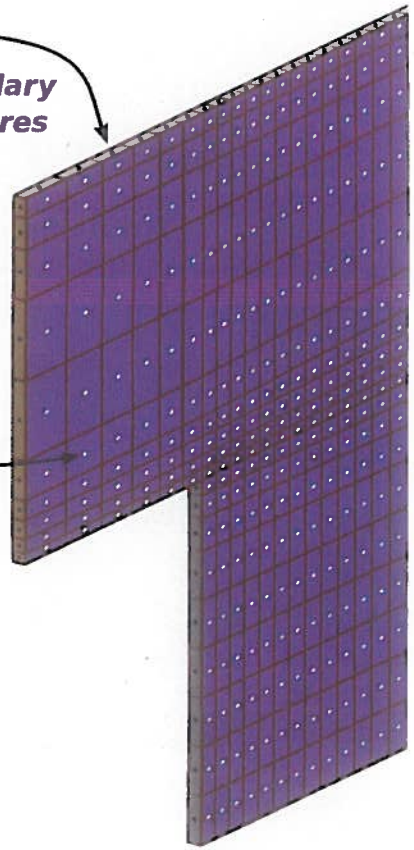
values at boundary face centres

```
boundaryField
{
    inlet
    {
        type  zeroGradient;
    }
    ...
```

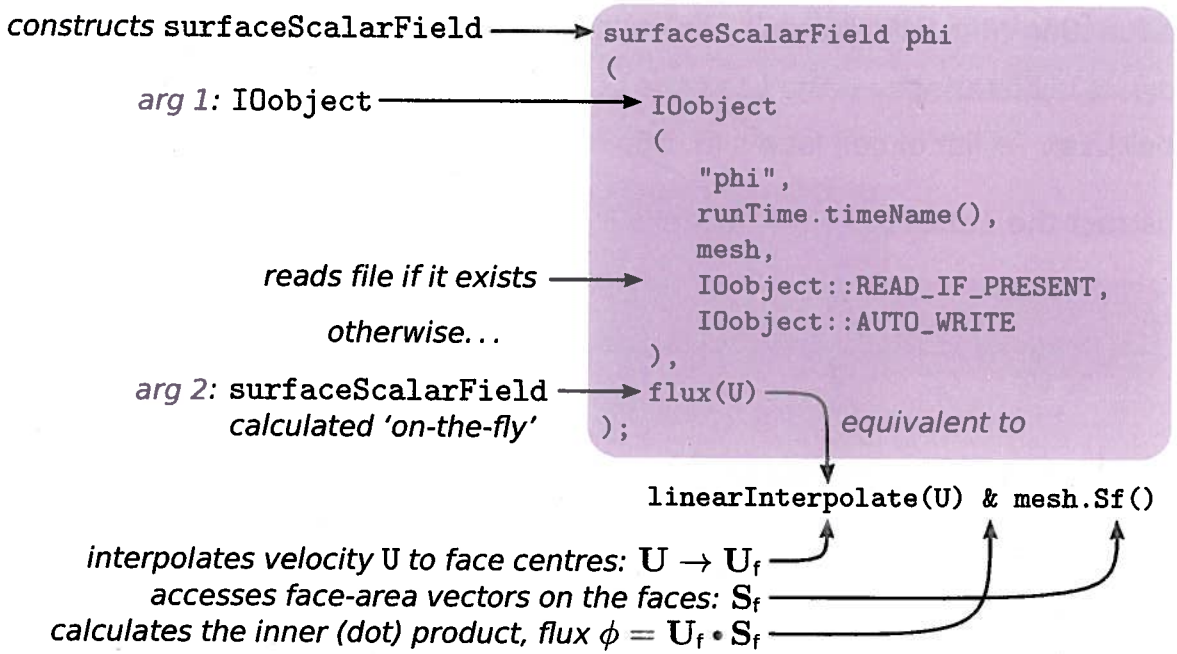
values at cell centres

`GeometricField::Internal` (typedef)

```
dimensions                  regIOobject data
[0 2 -2 0 0 0 0];
                                          db()
internalField
nonuniform List<scalar>     IOobject data time()
2048                                      name()
(
0.347229
0.345293                    fvMesh reference mesh()
...
```



- Lines 37-49: `volVectorField U` constructed → like `p`, but `vector`
- Line 52 includes the file `createPhi.H`
  `$FOAM_SRC/finiteVolume/cfdTools/incompressible/createPhi.H`



- `surfaceScalarField` → `GeometricField` with values defined at face centres

- Assume Darcy Law, body force $\mathbf{f} = -\nu d\,\mathbf{U}$, isotropic porosity d = 1000 m^{-2}
- ... defined in a single `cellZone` in the mesh

- `mesh` holds reference to the cell zones → find the access function
- The function (find it!) returns `const cellZoneMesh &`
- Show `cellZoneMesh` → `PtrList<cellZone>` → `PtrList<labelList>`
- `labelList` → list of cell labels (3, 56, 72, ...); `PtrList` → multiple lists

- Construct the `labelList` for the cells in the `cellZone`

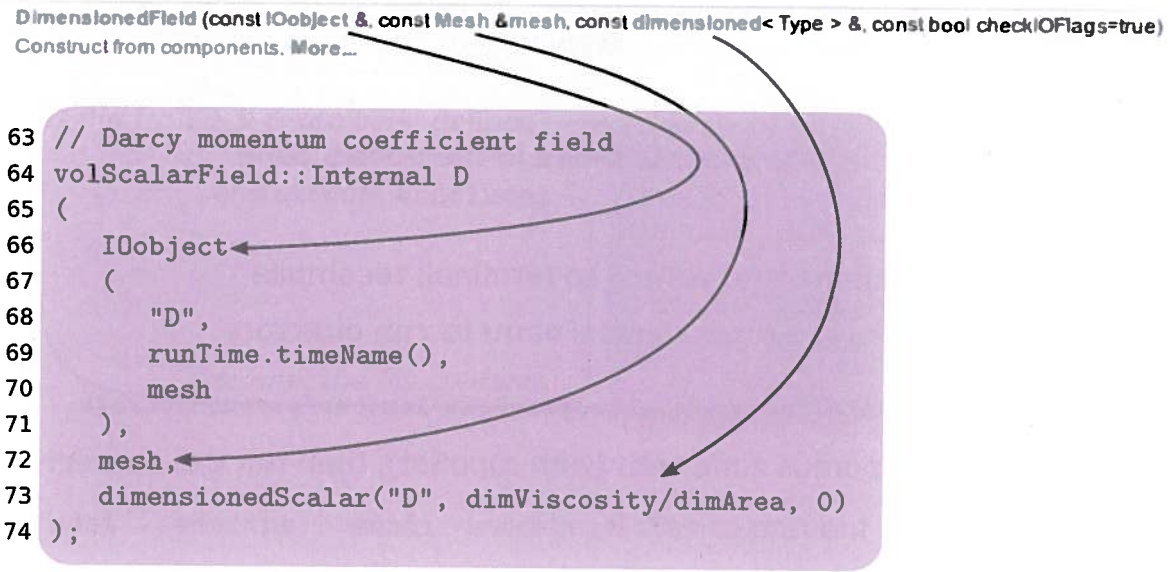
```
60 // Porous zone is hard-coded as the first cellZone in the list
61 const labelList& porousCells(mesh.cellZones()[0]);
```

first element in the `PtrList`

- Create a field $D = \nu d$, with $d = 1000$ m^{-2} in cells within zone; 0, elsewhere
- Body force → no patch field needed ⇒ use `GeometricField::Internal`
- Initialise as zero, then set values in cell zone

define as internal field
no need to have ~~boundary~~ boundaries

- `GeometricField::Internal` is a typedef for the `DimensionedField` class




```
63 // Darcy momentum coefficient field
64 volScalarField::Internal D
65 (
66     IOobject
67     (
68         "D",
69         runTime.timeName(),
70         mesh
71     ),
72     mesh,
73     dimensionedScalar("D", dimViscosity/dimArea, 0)
74 );
```

- `readOption` and `writeOption` omitted → default to `NO_READ` and `NO_WRITE`

- Set `D` values inside the porous zone using `forAll(...)`
- Array operator accesses value → need to assign (=) a value
- ...*i.e.* a `scalar`, not a `dimensionedScalar`; create the value

```
76 // Darcy constant hard-coded to 1000
77 const scalar dNu = 1000*nu.value();
```

accesses the value from `dimensionedScalar`

- Write the `forAll(...)` loop to set `D` to `dNu` in the porous zone
- Compile
- Testing: add `Info` statement to write `D` to terminal; recompile
- Copy the `porousBlockage` tutorial (pisoFoam) to `run` directory

```
>> run
>> cp -r $FOAM_TUTORIALS/incompressible/pisoFoam/laminar/porousBlockage .
```

- Generate mesh, and porous zone with (with topoSet), then run the new solver

```
>> blockMesh
>> topoSet
```

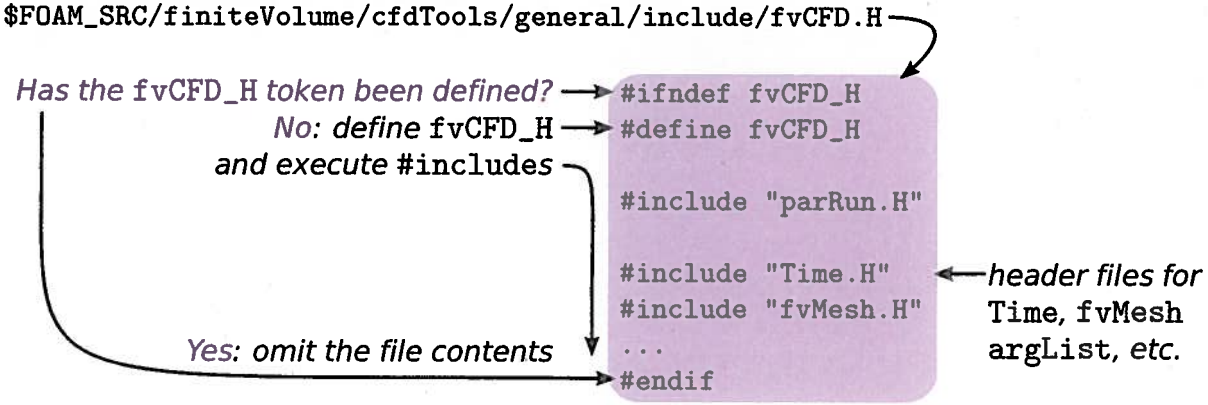
- Run the new solver and check `D` field

```
>> porousIcoFoam | less
```

- `porousIcoFoam.C` → ∼30 statements (lines ending ';'), 5 controls (`if`/`while`)
- Understand the (short) code → modify with confidence

- Starts with (line 32): `#include "fvCFD.H"` → included single file
- ...containing `#include` statements for `.H` files of commonly used classes

```
>> find $FOAM_SRC -name fvCFD.H -type f
$FOAM_SRC/finiteVolume/cfdTools/general/include/fvCFD.H
```



- `#ifndef...#define...#endif` used in `.H` files to prevent multiple inclusions
- `fvCFD.H` → makes solver code tidy; convenient, code reuse

- Inside the `main()` function lines 37-51:
- ... standard `#include` statements to read `Time` data, mesh,
- ... fields (`createFields.H`), controls (`pisoControl`, `initContinuityErrors.H`)
- Lines 52-119: solution code, with iterative loops

```
52  while (runTime.loop())                    <- increments time (runTime++)
53  {                                            stops if end time exceeded
67      if (piso.momentumPredictor())
68      {                                     <- user-controlled switch
69          solve(UEqn == -fvc::grad(p));
70      }
73      while (piso.correct())                <- loops nCorrectors iterations
74      {
89          while (piso.correctNonOrthogonal())   <- nNonOrthogonalCcorrectors
90          {                                        iterations
100             pEqn.solve(mesh.solver(p.select(piso.finalInnerIter())));
106         }
110         U = HbyA - rAU*fvc::grad(p);
112     }
119 }
```

- `loop()` defined in `Time.C`
- PISO controls (`correct()`, *etc.*) in `solutionControl.C`, `pimpleControl.C`

- In the momentum equation, add and subtract source term $A\mathbf{U}$

$$\underbrace{\frac{\partial \mathbf{U}}{\partial t} + \nabla \bullet (\mathbf{U}\mathbf{U}) - \nu \nabla^2 \mathbf{U} - A\mathbf{U}}_{-\mathbf{H}(\mathbf{U}),\ \text{contains all terms except } \nabla p} + A\mathbf{U} = -\nabla p \qquad (*)$$

- Divide by A, take divergence and apply $\nabla \bullet \mathbf{U} = 0$

$$-\nabla \bullet \left(\frac{\mathbf{H}(\mathbf{U})}{A} \right) + \cancelto{=0}{\nabla \bullet \mathbf{U}} = -\nabla \bullet \frac{1}{A} \nabla p$$

- Pressure equation (the actual one — $1/A$ makes it much more convergent)

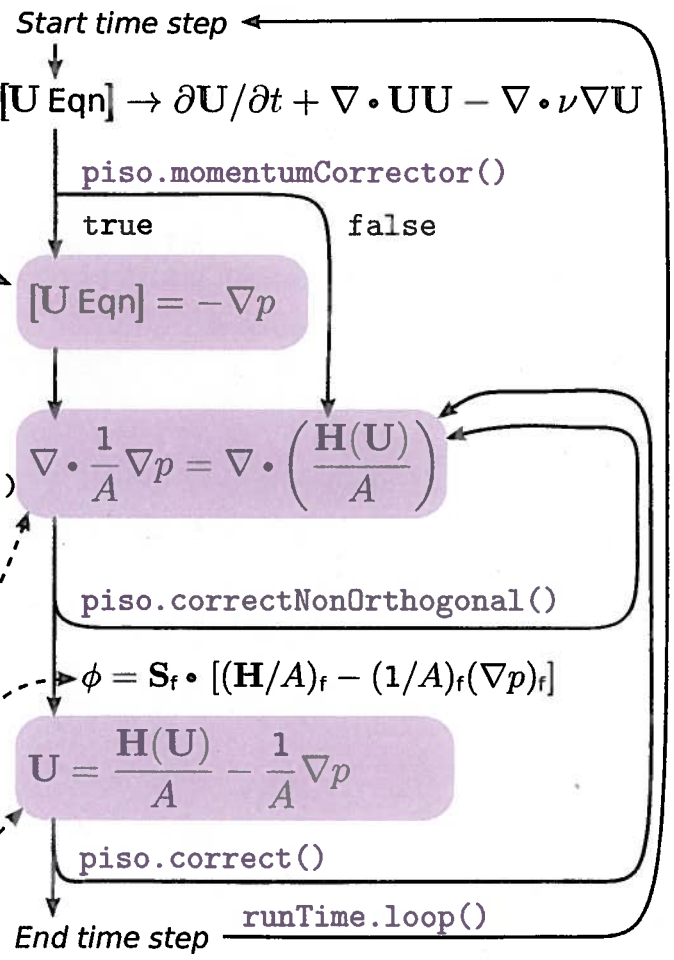
$$\nabla \bullet \frac{1}{A} \nabla p = \nabla \bullet \left(\frac{\mathbf{H}(\mathbf{U})}{A} \right)$$

- Momentum corrector, (*) divided by A

$$\mathbf{U} = \frac{\mathbf{H}(\mathbf{U})}{A} - \frac{1}{A} \nabla p$$

- A = diagonal coefficients of momentum matrix equation (PISO, SIMPLE, PIMPLE)

```
60 fvVectorMatrix UEqn
61 (
62     fvm::ddt(U)
63   + fvm::div(phi, U)
64   - fvm::laplacian(nu, U)
65 );

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

75 volScalarField rAU(1.0/UEqn.A());
77 volVectorField HbyA("HbyA", U);
78 HbyA = rAU*UEqn.H();

79 surfaceScalarField phiHbyA
82 (fvc::interpolate(HbyA) & mesh.Sf())

93 fvScalarMatrix pEqn
94 (
95     fvm::laplacian(rAU, p) ==
96     fvc::div(phiHbyA)
97 );

100 pEqn.solve(...);

104 phi = phiHbyA - pEqn.flux();

110 U = HbyA - rAU*fvc::grad(p);
```



- Darcy force $\rightarrow -D\mathbf{U}$ on right had side (RHS) of '=' in momentum equation
- ... or $+D\mathbf{U}$ on LHS
- Add the term to the `fvMatrix<Type>` class $\rightarrow$ represents matrix equation

`fvm::` *functions return* `fvMatrix` $\rightarrow$ *create* M *and* $\mathbf{B}$

```
61 fvVectorMatrix UEqn
62 (
63     fvm::ddt(U)
64   + fvm::div(phi, U)
65   - fvm::laplacian(nu, U)
66   + D*U
67 );
```

$$\begin{bmatrix} M_{11} & M_{12} & \dots & M_{1N} \\ M_{21} & M_{22} & \dots & M_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ M_{N1} & M_{N2} & \dots & M_{NN} \end{bmatrix} \begin{bmatrix} \mathbf{U}_1 \\ \mathbf{U}_2 \\ \vdots \\ \mathbf{U}_N \end{bmatrix} = \begin{bmatrix} \mathbf{B}_1 \\ \mathbf{B}_2 \\ \vdots \\ \mathbf{B}_N \end{bmatrix}$$

$$[M] \qquad \mathbf{U} = \mathbf{B}$$

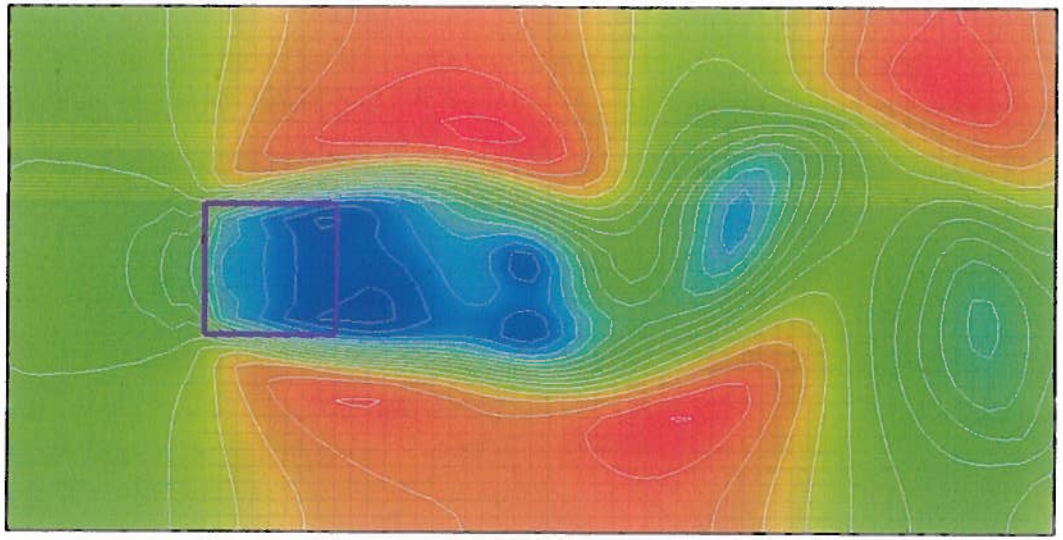
or, alternatively

```
66 ==
67   - D*U
```

`GeometricField (volVectorField)` *adds contribution to* $\mathbf{B}$

- `fvMatrix<Type>` stores:
  Matrix coefficients $[M]$ $\rightarrow$ `PtrList<Field<Type>>`
  Solution variable $\mathbf{U}$ $\rightarrow$ `GeometricField<Type, ...>&` *reference*
  Source $\mathbf{B}$ $\rightarrow$ `Field<Type>`
- Compile the solver

- Run the `porousBlockage` case with `porousIcoFoam`
- View results in ParaView; plot contours $0 < |\mathbb{U}| < 1.7$ m/s, 0.1 m/s interval



- Porosity creates blockage → flows around blockage, shedding vortices

- Calculations involve derivatives, *e.g.* $\nabla \bullet$, ∇, ∇^2, $\nabla \times$, $\partial/\partial t$
- ...represented by functions, *e.g.* `div`, `grad`, `laplacian`, `curl`, `ddt`

- `fvm::` (*e.g.* `fvm::div`) → matrix coefficients / source (`fvMatrix<Type>`)
- `fvc::` (*e.g.* `fvc::grad`) → calculates new field (`vol<Type>Field`)
- Fields can provide source to `fvMatrix`, *e.g.* `D*U`, `fvc::grad(p)`

- Segregated → matrix equation (`fvMatrix`) solves for one variable
- Decoupled → vectors *etc.*, *e.g.* $\mathbf{U}$, solves for each component, U_x, U_y, U_z
- ⇒ 4 `fvm::` terms → `ddt`, `laplacian`, `div(phi,...)` (advection), `Sp` (source)

- List of derivatives for arbitrary field $\mathbf{Q}$

Description	Expression	Function
Time derivative†	$\partial\rho\mathbf{Q}/\partial t$	`fvm::ddt(rho, Q)`
Advection‡	$\nabla\bullet(\rho\mathbf{U}\mathbf{Q})$	`fvm::div(phi, Q)`
Laplacian†	$\nabla\bullet\Gamma\nabla\mathbf{Q}$	`fvm::laplacian(Gamma, Q)`
Implicit source	$\rho\mathbf{Q}$	`fvm::Sp(rho, Q)`
Divergence	$\nabla\bullet\mathbf{Q}$	`fvc::div(Q)`
Gradient	$\nabla\mathbf{Q}$	`fvc::grad(Q)`
Curl	$\nabla\times\mathbf{Q}$	`fvc::curl(Q)`
Explicit source	$\mathbf{Q}$	`Q`

† Equivalent functions exist for $\partial\mathbf{Q}/\partial t$ and $\nabla^2\mathbf{Q}$ (without ρ, Γ)
‡ Advection: `fvm::div` function with advective flux field `phi` as the 1st argument

- Equation with time derivative and source

$$\frac{\partial \mathbf{Q}}{\partial t} = a\mathbf{Q}$$

- Discretized $\rightarrow \Delta t$ = time step; 'o' = old time step values

$$(\mathbf{Q} - \mathbf{Q}^{o})/\Delta t = a\mathbf{Q}$$

- For $a = 1/\Delta t$, if source is explicit, *i.e.* uses known values $\mathbf{Q}^{o}$

$$\mathbf{Q} - \mathbf{Q}^{o} = \mathbf{Q}^{o} \quad \Rightarrow \quad \mathbf{Q} = 2\mathbf{Q}^{o}$$

- ... but if source is implicit, *i.e.* the solution variable $\mathbf{Q}$

$$\mathbf{Q} - \mathbf{Q}^{o} = \mathbf{Q} \quad \Rightarrow \quad \mathbf{0} = \mathbf{Q}^{o} \quad !?$$

- $\Rightarrow$ Positive sources (right hand side of =) must be explicit
- $\Rightarrow$ Negative sources can be implicit $\rightarrow$ preferred, more stable
- Task: implement Darcy term as implicit source, recompile, test

SuSp does it for you.

- Isotropic porosity model $\rightarrow \mathbf{f} = -\nu d \mathbf{U}$ with scalar d = 1000 m^{-2}
- Anisotropic porosity model $\rightarrow \mathbf{f} = -\mu \mathbf{D} \bullet \mathbf{U}$ with tensor $\mathbf{D}$
- Task: implement anisotropic model with d_x, d_z = 1000, d_y = 1 (low resistance in y)

```
63 volTensorField::Internal D
64 (
65     IOobject
66     (
67         "D",
68         runTime.timeName(),
69         mesh
70     ),
71     mesh,
72     dimensionedTensor("D", dimViscosity/dimArea, Zero)
73 );
```

- Change dNu to `tensor`

```
76 const tensor dNu =
77     tensor(1000, 0, 0, 0, 1, 0, 0, 0, 1000)*nu.value();
```

if anisotropic must be tensor

- List of algebraic operations and code functions

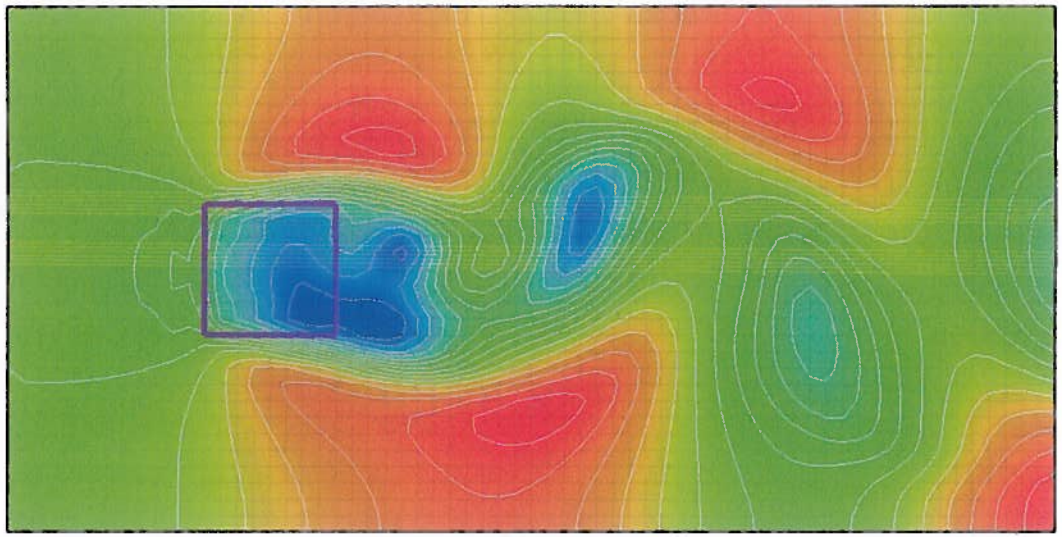
Operator	Ranks	Expression	Code
Inner product	≥ 1	$\mathbf{a} \cdot \mathbf{b}$	`a & b`
Double inner product	2	$\mathbf{a} : \mathbf{b}$	`a && b`
Cross product	1	$\mathbf{a} \times \mathbf{b}$	`a ^ b`
Outer product		$\mathbf{ab}\dagger$	`a * b`
Square		$\mathbf{a}^2 \equiv \mathbf{aa}$	`sqr(a)`
Magnitude squared		$\lvert\mathbf{a}\rvert^2 \equiv \mathbf{a} \overset{R}{:} \mathbf{a}$	`magSqr(a)`
Magnitude		$\lvert\mathbf{a}\rvert \equiv \sqrt{\mathbf{a} \overset{R}{:} \mathbf{a}}$	`mag(a)`
Transpose	2	$\mathbf{a}^{\mathrm{T}}$	`a.T()`
Trace	2	$\mathrm{tr}\,\mathbf{a} = \mathbf{I} : \mathbf{a}$	`tr(a)`
Symmetric	2	$\mathrm{symm}\,\mathbf{a} = (\mathbf{a} + \mathbf{a}^{\mathrm{T}})/2$	`symm(a)`
Skew	2	$\mathrm{skew}\,\mathbf{a} = (\mathbf{a} - \mathbf{a}^{\mathrm{T}})/2$	`skew(a)`
Deviatoric	2	$\mathrm{dev}\,\mathbf{a} = \mathbf{a} - (\mathrm{tr}\,\mathbf{a})\mathbf{I}/3$	`dev(a)`
Deviatoric (II)	2	$\mathrm{dev}_{\mathrm{II}}\,\mathbf{a} = \mathbf{a} - 2(\mathrm{tr}\,\mathbf{a})\mathbf{I}/3$	`dev2(a)`

$\dagger \mathbf{a} \otimes \mathbf{b}$ 'Rank' R: 0 = scalar; 1 = vector; 2 = tensor

- Modify `porousIcoFoam.C` for tensor D; compile and test

```
66 + (D & U)
```

- View results in ParaView; plot contours $0 < |\mathbf{U}| < 1.7$ m/s, 0.1 m/s interval



- Challenge: how can the model be implemented in a more implicit way?
- Hint: $\mathbf{D} \bullet \mathbf{U} \equiv [(1/3)\,\mathrm{tr}\,\mathbf{D}]\,\mathbf{U} + \mathrm{dev}(\mathbf{D}) \bullet \mathbf{U}$

- Functionality specified by the user through `functions` in `controlDict` file
- Generally post-processing, calculating time-data and writing to a single file
- ...or larger data sets, *e.g.* field data, images, written to individual files

- Task: write a function object that writes time-data to file for kinetic energy, K

$$K = \frac{1}{2}m|\mathbf{U}|^2 = \frac{1}{2}\sum_{i=0}^{N-1}\rho_i V_i|\mathbf{U}_i|^2$$

where N = number of cells, V = cell volumes

- Assume initially for incompressible solver, with constant density ρ

post Process - Cldt.

- The foamNew... script collection includes a script for a new function object
- Create a new function object named energy

```
>> run
>> foamNewFunctionObject energy
```

- Go into the directory and run a test compilation

```
>> cd energy ; wmake
```

- Parameters of the model → private data

$$K = \frac{1}{2}\rho \sum V|\mathbf{U}|^2$$

scalar — accessed from object registry, identified by name (word)

- Modify private data in energy.H

```
81 //- Name of velocity field on object registry
82 word UName_;
83
84 //- Density
85 scalar rho_;
86
87 //- label
88 label labelData_;
```

- Modify private data at construction in `energy.C`

```
53 UName_(dict.lookupOrDefault<word>("UName", "U")),
54 rho_(readScalar(dict.lookup("rho"))),
55 labelData_(readLabel(dict.lookup("labelData")))
```

- Modify `read()` function in `energy.C`

```
70 dict.readIfPresent("UName", UName_);
71 dict.lookup("rho") >> rho_;
72 dict.lookup("labelData") >> labelData_;
```

- Function objects are called indirectly by `Time::loop()` function in solver

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

Time.C:

```
bool running = run();
functionObjects_.execute();
```

FunctionObjectList.C:

```
forAll(*this, objectI)        // loops over function objects
{
    ok = operator[](objectI).execute() && ok;
    ok = operator[](objectI).write() && ok;
}
```

calls `execute()` *and* `write()`

← hold value in execute e.g average.

writing directly.

- Calculate kinetic energy in the `write()` function
- First, access `U` field on the object registry with the `lookupObject()` function

```
91 void Foam::energy::write()
92 {
93     const volVectorField& U = obr_.lookupObject<volVectorField>(UName_);
```

template function.

base class

by default → `"U"`

- Include `volFields.H` so the compiler recognises `"volVectorField"`
- How do we calculate KE — a single value — for the whole domain?
  `0.5*rho_*magSqr(U)` *calculates one value per cell*
- Clue: look at the "finite volume calculate" (`fvc`) functionality
  see `$FOAM_SRC/finiteVolume/finiteVolume/fvc`

```
94     Info<< "Kinetic Energy = "
95         << 0.5*rho_*fvc::...chosen function...(magSqr(U)) << endl;
```

- Include `fvc.H` header file and compile

U · U
magSqr

Volume Integrate

- Test on the `pitzDaily` case; copy the case to the `run` directory
- Create a file in `system` named `energy`

```
1 // Description
2 //    Writes kinetic energy of entire domain
3
4 type    energy;
5 libs    ("libenergyFunctionObject.so");
6 rho     1.2;
```

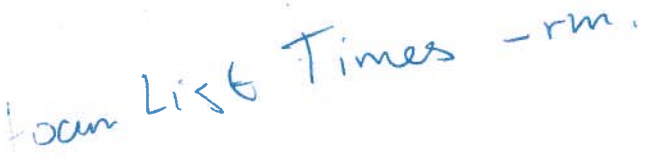
- Include the function inside `functions` in `controlDict`

```
48 functions
49 {
50     #includeFunc    energy;
51     ...
```

- Run the `pitzDaily` case; monitor the function object output, *e.g.*

```
>> simpleFoam | grep "Kinetic"
```

- Output includes a "name" and dimensions. Why?
- Task: modify the code to remove the name and dimensions



- Document the new function object in the `.H` file header, *e.g.*

```
29 Description
30     This function object calculates kinetic energy over the entire domain
31     and writes data to a file
32
33     Example of function object specification:
34     \verbatim
35     energy1
36     {
37         type            energy;
38         functionObjectLibs ("libenergyFunctionObject.so");
39         ...
40         UName           U;      // optional
41         rho             1.205;
42     }
43     \endverbatim
44
45     \heading Function object usage
46     \table
47         Property     | Description             | Required | Default value
48         type         | type name: energy       | yes      |
49         UName        | Name of U field         | no       | U
50         rho          | Density                 | yes      |
51     \endtable
```

- Task: modify the function object to write to file, *e.g.* for graph plotting
- The output to file class, `OFstream`, provides convenient functionality
- ...supported by the `fileName` class
- To set up writing to file, construct an `OFstream` from a `fileName`
- ⇒ create an `OFstream` named `file_` as private data; in `energy.H`

```
87 //- Output file stream
88 OFstream file_;
```

- In constructor in `energy.C`, initialize `os_` from `fileName`
- ...where the file name is `<functionObjectName>.dat` in the case directory

```
57 rho_(readScalar(dict.lookup("rho"))),
58 file_(obr_.time().path()/name + ".dat")
```

case directory *file name* *extension*

- Compile, including any required header files

- Data needs to be written as a time and value on each line
- ⇒ stream to `file_` instead of `Info` in `write()` function

```
96 file_<< obr_.time().  ...time access function...  << " "
97       << 0.5*rho_*fvc::domainIntegrate(magSqr(U)).value() << endl;
```

- Add a file header comment in the constructor function

```
59 file_ << "# time, kineticEnergy" << endl;
```

- Compile and test
- Column width varies in output file; use TAB spaces

  see punctuation tokens in `$FOAM_SRC/OpenFOAM/lnInclude/token.H`

```
59 file_ << "# time" << token::TAB << "kineticEnergy" << endl;
```

```
96 file_<< obr_.time().timeName() << token::TAB
```

- Compile and check the output is formatted neatly

- Decompose the test case into 2 domains and run in parallel
- 2 files are written → `write()` function is executed by each process
- Task: ensure this function object just writes one file in parallel
- Static function `PStream::master()` → `true` if process runs on master node
- ⇒ wrap file streaming in `if` statement with `PStream::master()`

```
59 if (Pstream::master())
60 {
61     file_ << "# time" << token::TAB << "kineticEnergy" << endl;
62 }
```

- Repeat for the file streaming in `write()` function. Note that...
- `fvc::domainIntegrate` cannot be executed only on master node. Why?
- Compile and test in parallel → file written to `processor0` dir
- ⇒ change `fileName` in `OFstream` initialization

```
57 file_(obr_.time().rootPath()/obr_.time().globalCaseName()/name + ".dat")
```

root path *case directory name*

- energy function object applicable to incompressible solvers only
- Task: extend applicability to compressible cases
- Compressible cases store varying density field — usually named `rho`
- ⇒ in `write()` function, set up logic based on `rho` field on database

```
100 scalar KE = 0.0;
101
102 if (obr_.foundObject<volScalarField>("rho"))
103 {
104     const volScalarField& rho = obr_.lookupObject<volScalarField>("rho");
105     KE = ...evaluation based on rho field... ;
106 }
107 else
108 {
109     KE = ...evaluation based on rho_ scalar... ;
110 }
111
112 if (Pstream::master())
113 {
114     file_ << obr_.time().timeName() << token::TAB << KE << endl;
115 }
```

- rho input is now redundant for compressible case
- ...assumed name of density field for compressible case
- Task: default rho_ = 1; add rhoName_ entry, defaulting to "rho"
- Add private data rhoName_ to energy.H
- Add "defaulting" initialization to rhoName_ and rho_ in energy.C

```
56 rhoName_(dict.lookupOrDefault<word>("rhoName", "rho")),
57 rho_(dict.lookupOrDefault<scalar>("rho", 1.0)),
```

- Modify read() function in energy.C

```
78 dict.readIfPresent("rhoName", rhoName_);
79 dict.readIfPresent("rho", rho_);
```

- Replace "rho" with rhoName_ in object registry lookups
- Compile and test on a compressible flow case, *e.g.* squareBend
  $FOAM_TUTORIALS/compressible/rhoSimpleFoam/squareBend

foamMonitor.