Instructional workshop on OpenFOAM programming
LECTURE # 3

Pavanakumar Mohanamuraly

April 26, 2014

Outline

Recap of Week 1

1*d* Heat Equation

Finite difference

lduMatrix and *fvMatrix*

FOAM Finite volume - Operators (*laplacian*)

Recap of Week 1

- Code development compile/running
- Basic FOAM data-structures
- FOAM polyMesh and fvMesh
- FOAM fields

$1d$ Heat equation

$$\frac{\partial \phi}{\partial t} - \kappa \frac{\partial^2 \phi}{\partial x^2} = f(x,t) \quad 0 \leq x \leq L, \quad t \geq 0 \tag{1}$$

with initial conditions,

$$\phi(x,0) = g_0(x)$$

and boundary conditions,

- Dirichlet type

$$\phi(0,t) = \phi_0 \ \ and \ \ \phi(L,t) = \phi_L$$

- Neumann type

$$\frac{\partial \phi}{\partial x}(0,t) = \phi_0' \ \ and \ \ \frac{\partial \phi}{\partial x}(L,t) = \phi_L'$$

Robins BC discussed on Day 2 (user defined BCs)

Finite difference (FD) basics [1]

$$\begin{array}{c}\text{Continuous}\\\text{PDE for}\\\phi(x,t)\end{array} \xrightarrow{\begin{array}{c}\text{Finite}\\\text{difference}\end{array}} \begin{array}{c}\text{Discrete}\\\text{Difference}\\\text{Equation}\end{array} \xrightarrow{\begin{array}{c}\text{Solution}\\\text{method}\end{array}} \begin{array}{c}\phi_i^m \text{ approxi-}\\\text{mation to } \phi(x,t)\end{array}$$

- x discretized as uniformly spaced interval $0 \leq x \leq L$ such that

$$x_i = (i-1)\Delta x \quad i = 1, 2, \ldots N \;\; where, \;\; \Delta x = \frac{L}{N-1}$$

- Similarly, discretize t uniformly in $0 \leq t \leq T$

$$t_m = (m-1)\Delta t \quad i = 1, 2, \ldots M \;\; where, \;\; \Delta t = \frac{T}{M-1}$$

$$\phi(x,t) \rightarrow \phi(x_i, t_m) \rightarrow \phi_i^m$$

[1] Source: `http://www.nada.kth.se/~jjalap/numme/FDheat.pdf`

2^{nd} order central difference

Taylor series expansion

- Forward

$$\phi_{i+1} = \phi_i + \Delta x \left.\frac{\partial \phi}{\partial x}\right|_{x_i} + \frac{\Delta x^2}{2} \left.\frac{\partial^2 \phi}{\partial x^2}\right|_{x_i} + \dots \tag{2}$$

- Backward

$$\phi_{i-1} = \phi_i - \Delta x \left.\frac{\partial \phi}{\partial x}\right|_{x_i} + \frac{\Delta x^2}{2} \left.\frac{\partial^2 \phi}{\partial x^2}\right|_{x_i} - \dots \tag{3}$$

Adding (2) and (3) we get

$$\phi_{i+1} + \phi_{i-1} = 2\phi_i + \Delta x^2 \left.\frac{\partial^2 \phi}{\partial x^2}\right|_{x_i} + O(\Delta x)^4 \tag{4}$$

$$\left.\frac{\partial^2 \phi}{\partial x^2}\right|_{x_i} = \frac{\phi_{i+1} - 2\phi_i + \phi_{i-1}}{\Delta x^2} + O(\Delta x)^2 \tag{5}$$

2^{nd} order central difference

Subtracting (2) and (3) we get

$$\phi_{i+1} - \phi_{i-1} = 2\Delta x \left.\frac{\partial \phi}{\partial x}\right|_{x_i} + (\Delta x)^3 \frac{\partial^3 \phi}{\partial x^3} + \ldots \tag{6}$$

$$\left.\frac{\partial \phi}{\partial x}\right|_{x_i} = \frac{\phi_{i+1} - \phi_{i-1}}{2\Delta x} + O(\Delta x)^2 \tag{7}$$

Alternatively, using equation (2),

$$\left.\frac{\partial \phi}{\partial x}\right|_{x_i} = \frac{\phi_{i+1} - \phi_i}{\Delta x} + O(\Delta x) \tag{8}$$

and equation (3),

$$\left.\frac{\partial \phi}{\partial x}\right|_{x_i} = \frac{\phi_i - \phi_{i-1}}{\Delta x} + O(\Delta x) \tag{9}$$

Boundary condition

- Dirichlet type

$$\phi_{i=1} = \phi_0 \quad \phi_{i=N} = \phi_L$$

- Neumann type (1^{st} order)

$$\phi_1 = \phi_2 - \phi_0' \Delta x \quad and \quad \phi_N = \phi_{N-1} + \phi_L' \Delta x$$

- Neumann type (2^{nd} order)

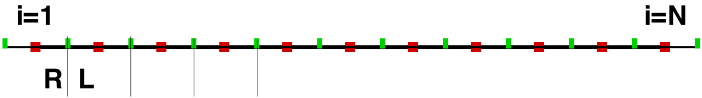
$$\phi_{i=0} = \phi_2 - 2\phi_0' \Delta x \quad and \quad \phi_{N+1} = \phi_N - 2\phi_L' \Delta x$$

  - Halo nodes $i = 0$ and $i = N + 1$

Make FD look like FV - *owner/neighbour*

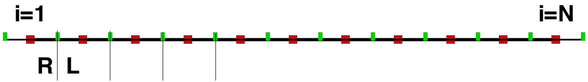
- Need *cell-to-cell* connectivity for FOAM FD
- Avoid this by modifying FD implementation
- Original domain

- Modified domain (half points - faces)



Make FD look like FV - *owner/neighbour*

- Modified domain



- Split into face contribution (interior faces)

$$\left.\frac{\partial^2\phi}{\partial x^2}\right|_{x_i} = \frac{\phi_{i+1} - 2\phi_i + \phi_{i-1}}{\Delta x^2} = \frac{1}{\Delta x}\left(\underbrace{\frac{\phi_{i+1} - \phi_i}{\Delta x}}_{\mathbf{L}} - \underbrace{\frac{\phi_i - \phi_{i-1}}{\Delta x}}_{\mathbf{R}}\right) \tag{10}$$

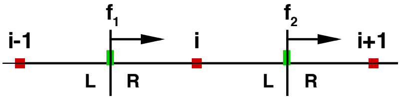
$$\frac{\phi_{i+1} - \phi_i}{\Delta x} = \frac{1}{\Delta x}\left(\underbrace{\phi_{i+1}}_{\mathbf{L}} - \underbrace{\phi_i}_{\mathbf{R}}\right) \tag{11}$$

$$\frac{\phi_i - \phi_{i-1}}{\Delta x} = \frac{1}{\Delta x}\left(\underbrace{\phi_i}_{\mathbf{L}} - \underbrace{\phi_{i-1}}_{\mathbf{R}}\right) \tag{12}$$

Make FD look like FV - *owner/neighbour*

Sign consistency examples (ignore Δx for clarity)

- Case (i)



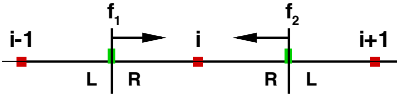
$$\mathbf{f_1} = \phi_{\mathbf{L}} - \phi_{\mathbf{R}} = \phi_{i-1} - \phi_i \qquad (13)$$

$$\mathbf{f_2} = \phi_{\mathbf{L}} - \phi_{\mathbf{R}} = \phi_i - \phi_{i+1} \qquad (14)$$

$$\underbrace{-\mathbf{f_1}}_{\mathbf{R}} + \underbrace{\mathbf{f_2}}_{\mathbf{L}} = -(\phi_{i+1} - 2\phi_i + \phi_{i-1}) = -\left.\frac{\partial^2 \phi}{\partial x^2}\right|_{x_i} \qquad (15)$$

Make FD look like FV - *owner/neighbour*

- Case (ii)



$$\mathbf{f_1} = \phi_\mathbf{L} - \phi_\mathbf{R} = \phi_{i-1} - \phi_i \qquad (16)$$

$$\mathbf{f_2} = \phi_\mathbf{L} - \phi_\mathbf{R} = \phi_{i+1} - \phi_i \qquad (17)$$

$$\underbrace{-\mathbf{f_1}}_{\mathbf{R}} - \underbrace{\mathbf{f_2}}_{\mathbf{R}} = -(\phi_{i+1} - 2\phi_i + \phi_{i-1}) = -\left.\frac{\partial^2 \phi}{\partial x^2}\right|_{x_i} \qquad (18)$$

- As long as L/R is consistently defined we get the correct expression

Make FD look like FV - *owner/neighbour*

Neumann boundary conditions

- Remember the fact that

$$\phi'_0 = \frac{\phi_1 - \phi_0}{\Delta x} = \phi^f_1 \; and \; \phi'_L = \frac{\phi_{N+1} - \phi_N}{\Delta x} = \phi^f_N \qquad (19)$$

  where, f denotes the face value
- Simply enforce values ϕ'_0 and ϕ'_L for the Neumann patch face

Make FD look like FV - *owner/neighbour*

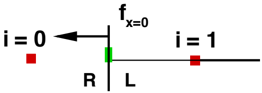
Dirichlet boundary conditions

- Patch face value set to enforces correct values (weak)
- Introduce halo nodes at both ends
  - At $i = N$
$$\frac{\phi_{N+1} - \phi_N}{\Delta x} = \frac{1}{\Delta x}\left(\phi_L - \phi_N\right) \tag{20}$$
  - At $i = 1$
$$\frac{\phi_{i=1} - \phi_{i=0}}{\Delta x} = \frac{1}{\Delta x}\left(\phi_1 - \phi_0\right) \tag{21}$$
- Remember that if dy and dz is $= 1$ then $\Delta x = \Delta V$, where ΔV is cell volume

Make FD look like FV - *owner/neighbour*

Sign consistency for boundary (ignore Δx for clarity)

- Case (i) At $x = 0$



- Dirichlet type

$$\mathbf{f_{x=0}} = \frac{\phi_{\mathbf{L}} - \phi_{\mathbf{R}}}{\Delta x} = \frac{\phi_1 - \phi_{x=0}}{\Delta x} \tag{22}$$

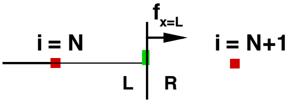
- Neumann type

$$\mathbf{f_{x=0}} = \phi'_0 \tag{23}$$

Make FD look like FV - *owner/neighbour*

Sign consistency for boundary (ignore Δx for clarity)

- Case (i) At $x = 0$



- Dirichlet type

$$\mathbf{f_{x=L}} = \frac{\phi_{\mathbf{L}} - \phi_{\mathbf{R}}}{\Delta x} = \frac{\phi_N - \phi_{N+1}}{\Delta x} \tag{24}$$

- Neumann type

$$\mathbf{f_{x=L}} = \phi'_L \tag{25}$$

How to access field boundary values ?

- *fixedValue* patch type

```
const tmp<scalarField> sf;
scalarField some( x.boundaryField()[ipatch].
    valueBoundaryCoeffs(sf) );
```

- *fixedGradient* patch type

```
scalarField some( x.boundaryField()[ipatch].
    gradientBoundaryCoeffs() );
```

Hands on - FD solver I

- For convenience set $dx = dy = dz = 1$ in the $1d$ grid
- Ignore time term for now and solve for steady-state
- Create a *volScalarField* phi and read from input file
- Create a *surfaceScalarField* named *flux*
- Loop over all internal faces of *flux* and set face value

$$\phi_f = \phi_{owner} - \phi_{neighbour} \tag{26}$$

- For Neumann patches set face value to the *gradient*

$$\phi_f = \phi'(x = 0 \ or \ x = L) \tag{27}$$

- For Dirichlet patches set face value using

$$\phi_f = \phi_{owner} - \phi(x = 0 \ or \ x = L) \tag{28}$$

Hands on - FD solver I

- Create a *volScalarField* named *residue* and initialize to zero
- Loop over all faces of *flux*
- Add the face value to the *owner* cell and subtract from *neighbour* cell of *residue*
- Do the same for boundary faces (no *neighbour* cell)
- Now residue contains the laplacian

Matrix forms

- Possible to construct matrix version of the discrete operator

Matrix form of discrete system of Poisson Equations

$$
\begin{array}{cc}
 & \begin{array}{cccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \end{array} &
\begin{pmatrix}
-1 & 1 & \times & \times & \times & \times & \times & \times & \times & \times \\
1 & -2 & 1 & \times & \times & \times & \times & \times & \times & \times \\
\times & 1 & -2 & 1 & \times & \times & \times & \times & \times & \times \\
\times & \times & 1 & -2 & 1 & \times & \times & \times & \times & \times \\
\times & \times & \times & 1 & -2 & 1 & \times & \times & \times & \times \\
\times & \times & \times & \times & 1 & -2 & 1 & \times & \times & \times \\
\times & \times & \times & \times & \times & 1 & -2 & 1 & \times & \times \\
\times & \times & \times & \times & \times & \times & 1 & -2 & 1 & \times \\
\times & \times & \times & \times & \times & \times & \times & 1 & -2 & 1 \\
\times & \times & \times & \times & \times & \times & \times & \times & 1 & -1
\end{pmatrix}
\end{array}
\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \end{pmatrix}
=
\begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ f_6 \\ f_7 \\ f_8 \\ f_9 \\ f_{10} \end{pmatrix}
$$

- BC enforced separately using *boundaryCoeff* and *internalCoeff*
- For *zeroGradient* BC this matrix becomes the full system
- Most of the entries of A are zeros $\implies$ A is *sparse*

Sparse Matrix Storage in FOAM

- FOAM uses the *LDU* sparse matrix storage
- Uses the *owner*/*neighbour* data for addressing non-zero L/U entries
- Optimized for symmetric matrices also handles asymmetric matrix
- Key limitation *"cannot store entries beyond first level of cell neighbours"*
  - Big issue for hybrid and highly skewed meshes
  - Higher order FVM ($> 2^{nd}$ order)

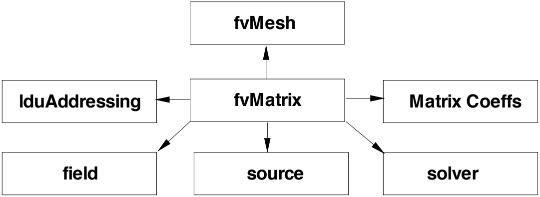
lduMatrix

- Abstract base class implementing the *ludMatix* and solvers
- Uses the *lduMatrix* to construct the *owner/neighbour* addressing scheme

```
/// Construct given an LDU addressed mesh.
lduMatrix (const lduMesh &)
/// Construct given an LDU addressed mesh and an
    Istream.
lduMatrix (const lduMesh &, Istream &)
```

- *lduMatrix* is not usable, need non-abstract class *fvMatrix*
- FOAM operators like *grad*, *div*, etc, return *fvMatrix*

fvMatrix



- Solves the system of equations

$$\mathbf{Ax} = \mathbf{b} \tag{29}$$

where, $\mathbf{x}$ is the *field*, $\mathbf{b}$ is the *source* and sparse matrix A entries are *Matrix Coeffs*.

- Subject to BC specified on the *field* using a specified *solver*

fvm and *fvc* namespace

- Two possible ways to construct discrete operators
  - Matrix-free scoped in *fvc* namespace
  - Matrix-based scoped in *fvm* namespace
- Explicit matrix construction impossible for non-linear operations like limiting
- Operators defined in *fvc* return field types
- Operators defined in *fvm* return *fvMatrix* types

Krylov solvers [2]

- For large sparse systems it only possible to perform *matrix-vector* products ($\mathbf{Ax}$)
- Even explicit construction of matrix $\mathbf{A}$ is impossible
- But it is possible to construct a *Krylov sequence* $\{\mathbf{x}, \mathbf{Ax}, \mathbf{A}^2\mathbf{x}, \mathbf{A}^3\mathbf{x}, \ldots\}$
- *Krylov sub-space* methods build up on the sequence and look for
  - Eigenvectors and
  - Invariant sub-spaces

  within the *Krylov sub-space*
- A large class of LA algorithms like CG, BICG, GMRES are based on *Krylov sub-spaces*
- Preconditioning is normally performed to speed up convergence

[2] Source: "The Matrix Eigenvalue Problem" by David S. Watkins

Krylov sub-space solvers in *fvMatrix*

- Krylov sub-space solvers available for *fvMatrix*
  - Symmetric matrix
    - GAMG - Geometric/Algebraic Multi-Grid solver
    - ICCG - Incomplete Cholesky Conjugate Gradient
    - PCG - Preconditioned Conjugate Gradient solver
  - Asymmetric matrix
    - BICCG - Bi-Conjugate Gradient
    - GAMG - Geometric/Algebraic Multi-Grid solver
    - PBiCG - Preconditioned Bi-Conjugate Gradient
  - Matrix preconditioners
    - DIC - Diagonal Incomplete Cholesky
    - DILU - Diagonal Incomplete LU
    - GAMG - Geometric/Algebraic Multi-Grid solver

Multigrid and Krylov solver beyond the scope of the workshop

How to use *fvMatrix* ?

- Will look at the specific constructor using *field* variable

```
fvScalarMatrix A( x, x.dimensions() );
```

- Access non-zero upper diagonal entries using $A.upper()$
- Access diagonal entries using $A.diag()$
- Access non-zero lower diagonal entries using $A.lower()$
- First access to $A.upper()$ after construction makes A symmetric
  - $lowerPtr\_ = upperPtr\_;$
- Since *fvMatrix* entries are based on $mesh.owner()/neighbour()$ following holds true
  - Size of $diag().size()$ = number of cells
  - Size of $upper()/lower().size()$ = number of internal faces

negDiagSum() function

- Negated row-wise sum of non-zero off diagonal coeffs

```
void Foam::lduMatrix::negSumDiag()
{
  const scalarField& Lower
    = const_cast<const lduMatrix &>(*this).lower();
  const scalarField& Upper
    = const_cast<const lduMatrix &>(*this).upper();
  scalarField& Diag = diag();

  const labelUList& l = lduAddr().lowerAddr();
  const labelUList& u = lduAddr().upperAddr();

  for (register label face=0; face<l.size(); face++)
  {
    Diag[l[face]] -= Lower[face];
    Diag[u[face]] -= Upper[face];
  }
}
```

Hands on - Create FD matrix A from example

- Ignore BC patches for now
- Access $A.upper()$ and assign that to 1
- Calculate the diagonal term using the $negSumDiag()$ function
- Now check if the matrix is symmetric using $A.symmetric()$
- It should return *true* or *false* ?

Hints

- $A.upper()/diag()$ is a *scalarField* so *forAll* works
- Lazy yet efficient option is to use assignment operator $=$

Specifying the BC

The BC is specified using two fields

A.boundaryCoeffs()

- Source term that goes to the *RHS*
- For Neumann condition the $-$*fixedGradient* value is specified
- For Dirichlet condition the $-$*fixedValue* divided by Δx

A.internalCoeffs()

- Term that goes to the *LHS* matrix coefficient
- For Neumann condition it is *zero*
- For Dirichlet condition it is -1 divided by Δx

Show on black-board how this calculated

Access boundary values (*fixedValue*)

- Access field connected to A using $A.psi()$

```
forAll( A.psi().boundaryField(), ipatch ) {
  if
  (
    A.psi().boundaryField()[ipatch].type()
    == "fixedValue"
  ){
    const tmp<scalarField> sf;
    scalarField value
    (
      A.psi().boundaryField()[ipatch].
          valueBoundaryCoeffs(sf)
    );
    A.boundaryCoeffs()[ipatch] = -1.0 * value;
    A.internalCoeffs()[ipatch] = -1.0;
  }
```

Access boundary values (*fixedGradient*)

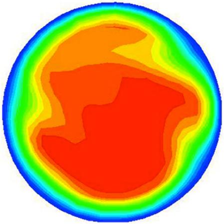
```
    if
    (
      A.psi().boundaryField()[ipatch].type()
      == "fixedGradient"
    )
    {
      Info << ipatch << " fixedGradient ";
      scalarField gradient
      (
        A.psi().boundaryField()[ipatch].
            gradientBoundaryCoeffs()
      );
      A.boundaryCoeffs()[ipatch] = -1.0 * gradient;
      Info << " gradient = " << gradient << "\n";
    }
```

Access boundary values (*zeroGradient*)

```
if
(
  A.psi().boundaryField()[ipatch].type()
  == "zeroGradient"
)
{
  Info << "Patch " << ipatch << ": is zeroGradient\n
      ";
}
```

How to access field boundary values ?

- Number of coefficients equal the number of faces in the patch
- Consequence of the fact that it is possible to have different values for each face in the patch
- Achieved using the *nonuniform* key word in the field dictionary
- Inlet velocity profiles for pipe flows



Hands on: Print boundary field type and value

- Loop over all patches of the field *x*
- Print the type of boundary and boundary value
- Set correct values for *A.boundaryCoeffs()/internalCoeffs()*

Solving the LA problem

- So far we have only discussed setting up $\mathbf{Ax}$
- What about $\mathbf{b}$?

Solving the LA problem

- So far we have only discussed setting up $\mathbf{Ax}$
- What about $\mathbf{b}$?
- It is accessed using $A.source()$
- If everything is set then we are ready to solve the LA problem

$$\mathbf{Ax} = \mathbf{b} \tag{30}$$

The *solution* dictionary

system/*fvSolution* dictionary

```
solvers
{
  x // The field variable name
  {
    solver            PCG;
    preconditioner    none;
    tolerance         1e-06;
    relTol            0;
  }
}
```

Hands on - Post-process 1*d* data

- 1*d* case is actually 3*d* with once cell thickness
- Print the result to a file *profile.dat*

```
#include <fstream>
....
std::ofstream fout("initial_sol.dat");
forAll( mesh.C(), i )
  fout << mesh.C()[i][0] << "  " << x[i] << "\n";
```

- Visualize using *gnuplot*

```
gnuplot> plot "initial_sol.dat" using 1:2 with lines
```

Hands on - Using fvMatrix for solution

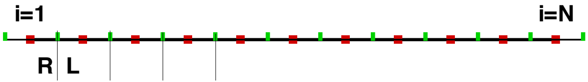
- Solve the $1d$ FD problem without time derivative using *fvMatrix*
- Proper boundary condition and value should be set
- Choose appropriate solver (write solver dictionary)
- Set the *A.source*() to *zero*
- Essentially solving the equation

$$\frac{\partial^2 \phi}{\partial x^2} = 0 \tag{31}$$

- Should give a straight line profile if Dirichlet BC is specified on both ends (use the previous hand-on to visualize results in gnuplot)

Finite Volume (FV) discretization of $1d$ heat equation

- FV domain



- Split into face contribution (interior faces)

$$\left.\frac{\partial^2 \phi}{\partial x^2}\right|_{x_i} = \frac{\phi_{i+1} - 2\phi_i + \phi_{i-1}}{\Delta x^2} = \frac{1}{\Delta x}\left(\underbrace{\frac{\phi_{i+1} - \phi_i}{\Delta x}}_{\mathbf{L}} - \underbrace{\frac{\phi_i - \phi_{i-1}}{\Delta x}}_{\mathbf{R}}\right) \tag{32}$$

$$\frac{\phi_{i+1} - \phi_i}{\Delta x} = \frac{1}{\Delta x}\left(\underbrace{\phi_{i+1}}_{\mathbf{L}} - \underbrace{\phi_i}_{\mathbf{R}}\right) \tag{33}$$

$$\frac{\phi_i - \phi_{i-1}}{\Delta x} = \frac{1}{\Delta x}\left(\underbrace{\phi_i}_{\mathbf{L}} - \underbrace{\phi_{i-1}}_{\mathbf{R}}\right) \tag{34}$$

Finite Volume (FV) discretization of 1d heat equation

- It is not a surprise that we end up with the same set of equations
- FV and FD give the same discretization for equally spaced grids in this case
- But this time we make use of FOAM operators to make the job simple

FOAM operators - *laplacianScheme*

- *fvm* :: *laplacian* is an in-built operator which yields correct *fvMatrix*
- Need to define the *laplacianScheme* in the *fvSchemes* dictionary

```
laplacianSchemes
{
  /// Green-gauss type with
  /// non-orthogonality correction
  default Gauss linear corrected;
}
```

- Constructor requires only the field as argument

```
fvScalarMatrix A( fvm::laplacian(x) );
A.source() = 0.0;
A.solve();
```

Hands on - FOAM'ish solver

- Implement the FV solver with similar inputs and equations as the FD hands on
- Plot 1d solution using *gnuplot*

End of Week 2 Day 1