051176 - Computational Techniques for Thermochemical Propulsion

Master of Science in Aeronautical Engineering

Linear Equation Solvers

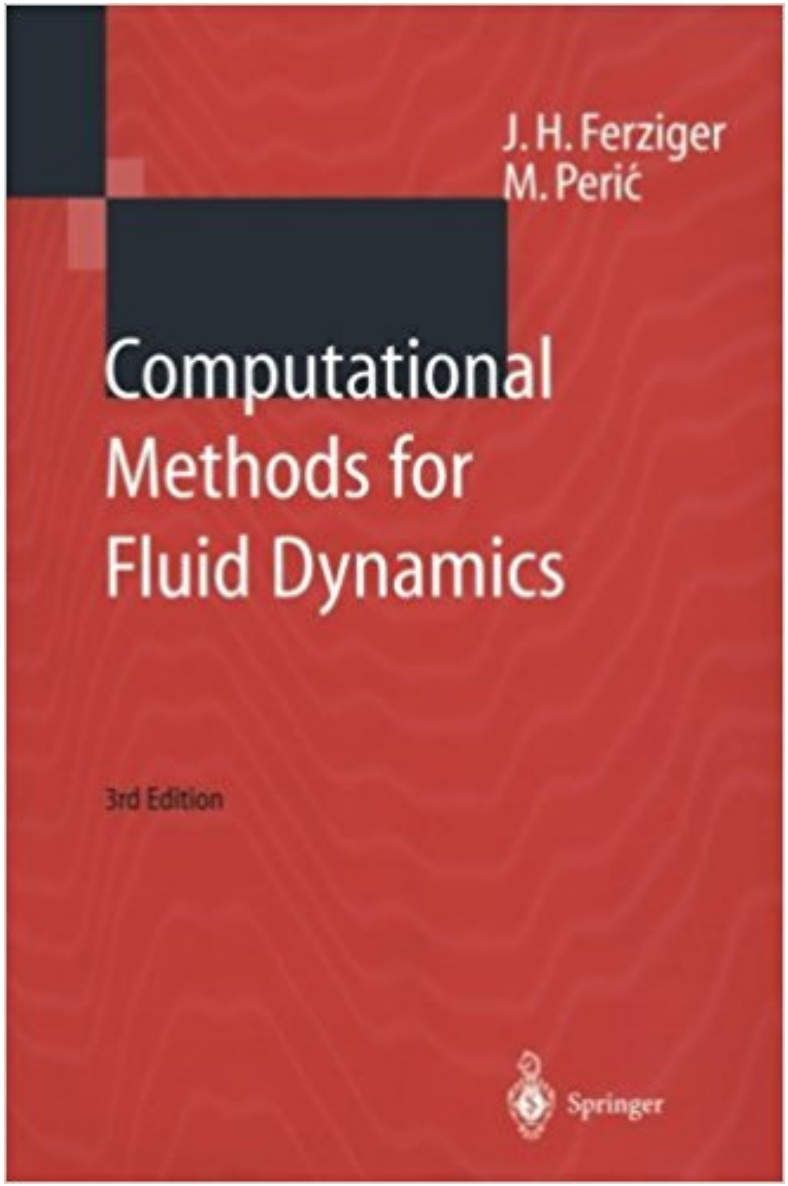
Prof. **Federico Piscaglia**

Dept. of Aerospace Science and Technology (DAER)

POLITECNICO DI MILANO, Italy

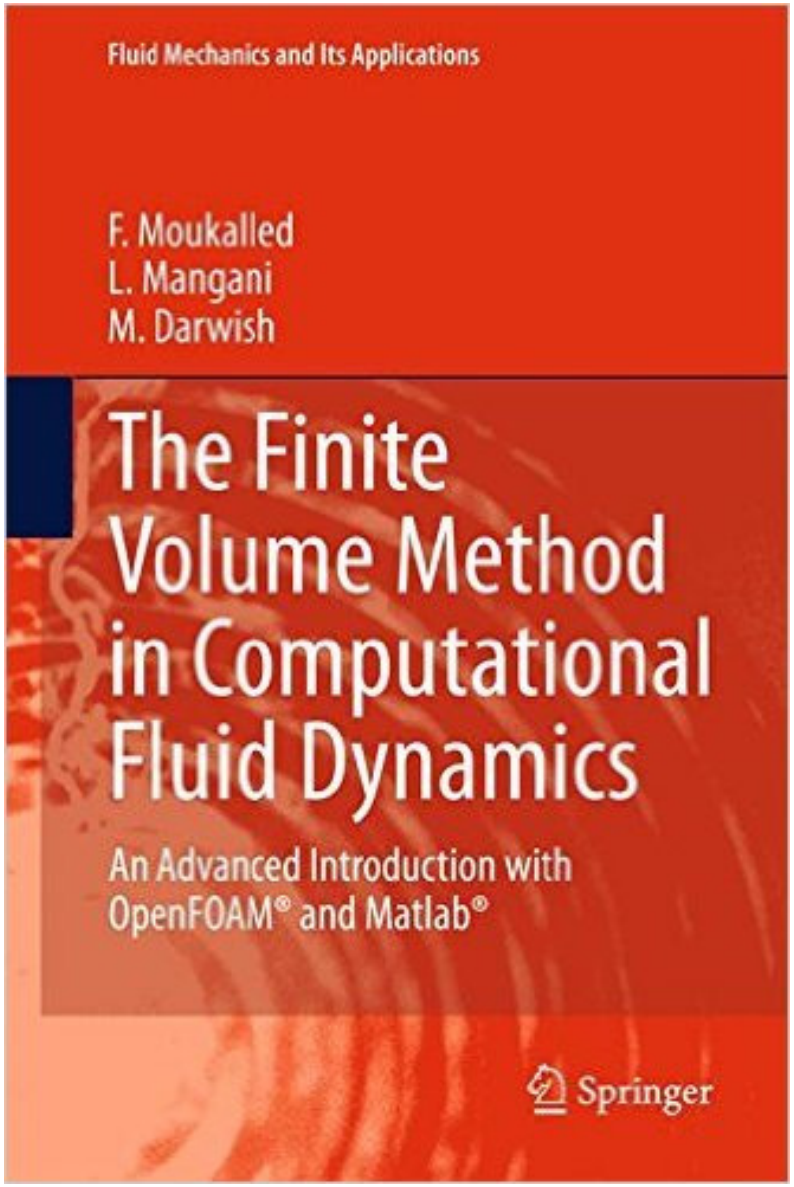
federico.piscaglia@polimi.it

Bibliography



Ferziger, Joel H., Peric, Milovan. "**Computational Methods for Fluid Dynamics**", Third Edition, Springer 2002.

Bibliography

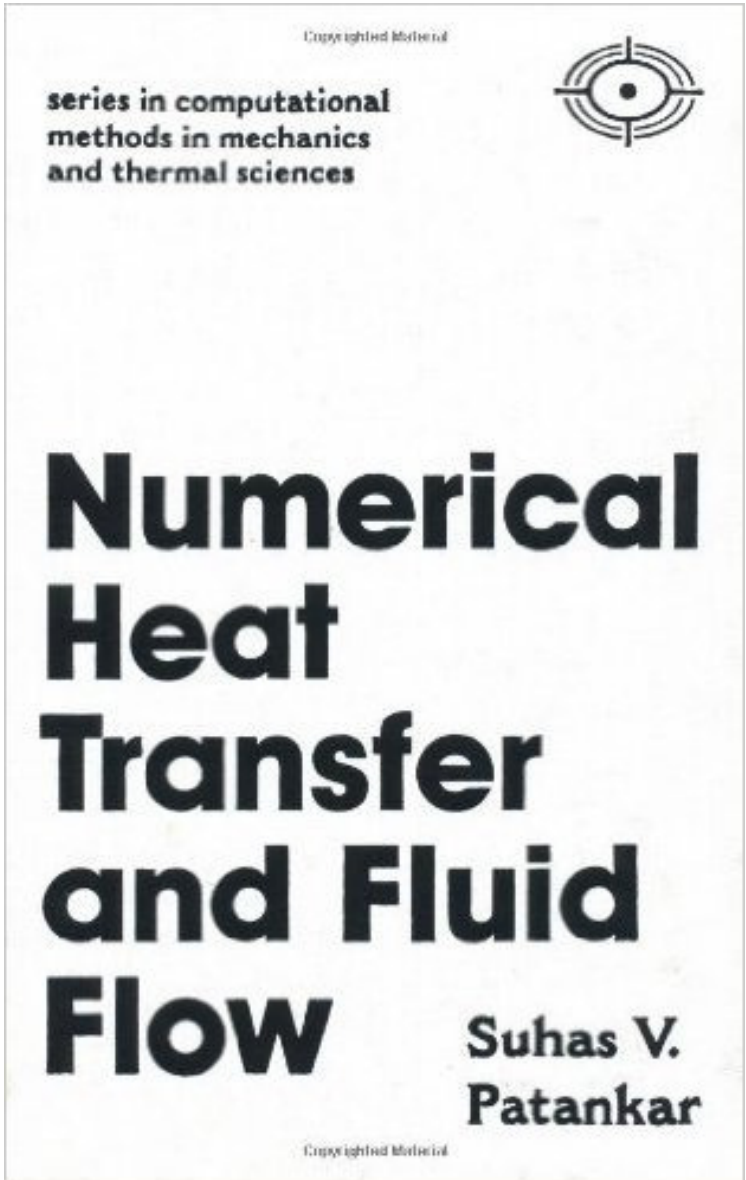


Some figures in this slides are taken from the reference text book:

F. Moukalled, L. Mangani, M. Darwish. "**The Finite Volume Method in Computational Fluid Dynamics**", Springer International Publishing Switzerland 2016.

Prof. Marwan Darwish is greatly acknowledged for sharing the images from his book and for allowing to include them in this course's material.

Bibliography



Suhas Patankar. "**Numerical Heat Transfer and Fluid Flow**", CRC Press, 1980.

The discretization process

The distribution of ϕ is discretized (**discretization process**); specific **discretization methods** are then available to discretize equations from the continuum form.

Change of ϕ over Δt in the Control Volume (CV)	+	Surface flux of ϕ over time Δt across the control volume	=	Source/sink terms over Δt in the control volume

- The discrete values of ϕ are typically computed by solving a set of algebraic equations relating the values at neighboring grid elements to each other; these discretized algebraic equations are derived from the conservation equation governing ϕ:

$$\frac{\partial\left(\rho\phi\right)}{\partial t}+\nabla\cdot\left(\rho\boldsymbol{v}\phi\right)=\nabla\cdot\left(\Gamma^{\phi}\nabla\phi\right)+Q^{\phi}$$

Once the values of ϕ are computed, the data is processed to extract any needed information.

Transport equations in OpenFOAM

Conservation of scalar quantities: the integral form of the equation describing conservation of a scalar quantity Φ (**TRANSPORT EQUATION**) can be expressed in its general form as:

$$\underbrace{\underbrace{\frac{\partial}{\partial t}\int_\Omega \rho\phi d\Omega}_{\texttt{temporal derivative}} + \underbrace{\int_\Omega \rho\phi(\vec{u}-\vec{u_b})\cdot\vec{n}dS}_{\texttt{convection term}}}_{\frac{D\Phi}{Dt}} = \underbrace{\int_\Omega \Gamma\nabla\Phi\cdot\vec{n}dS}_{\texttt{diffusion term}} + \underbrace{\sum f_\Phi}_{\texttt{source/sink terms}}$$

where f_Φ represents the transport of Φ by mechanisms other than convection and any sources or sinks of the scalar.

Example in OpenFOAM:

```
tmp<fvVectorMatrix> tUEqn
(
    fvm::ddt(rho, U) + fvm::div(phi, U)
  ==
  - turbulence->divDevRhoReff(U)
    fvOptions(rho, U)
);

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

Integration of the equations

The integration of the equations over each element is referred to as local assembly while the construction of the overall system of equations from these contributions is referred to as global assembly.

Thus, while the discretization of the equations is derived in terms of neighbor elements, **the assembly of the equations in the global matrix accounts for the actual indices of the elements.**

The class `fvMatrix` in OpenFOAM

How is domain discretization (e.g. mesh) connected to the solution of the governing equations?

In OpenFOAM the matrix of coefficients is stored in a class named `lduMatrix` and the specialized `fvMatrix`.

The chosen storage format of the coefficients is based on the *ldu sparse format*, in which all coefficients are stored in three main arrays:

- diagonal
- upper
- lower

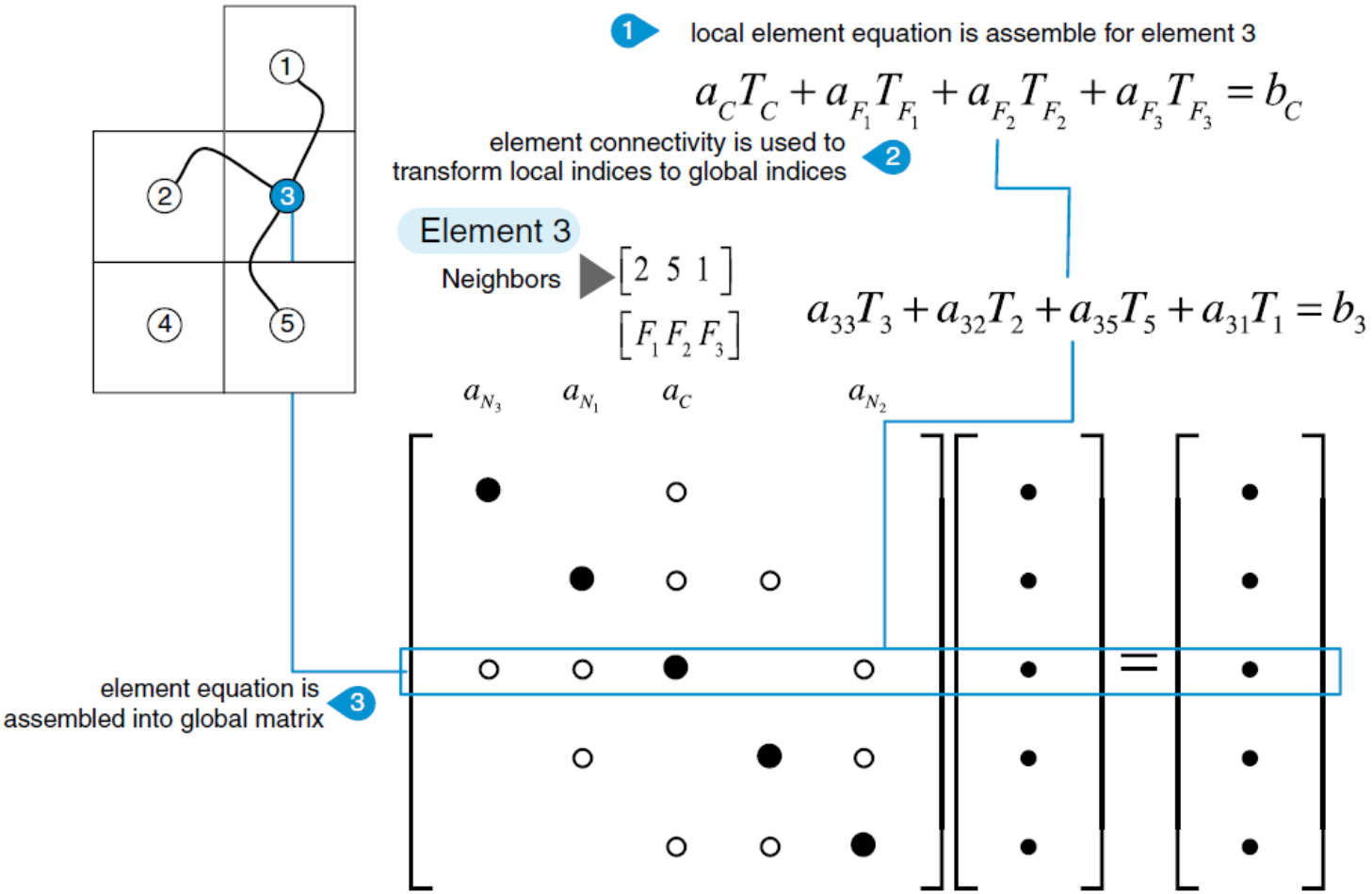
Face-addressing scheme in OpenFOAM

The storage of coefficients in OpenFOAM is based on the **face addressing scheme**: coefficients are stored following the interior face ordering, with access to elements (cells) based on the owner/neighbor indices associated with interior faces.

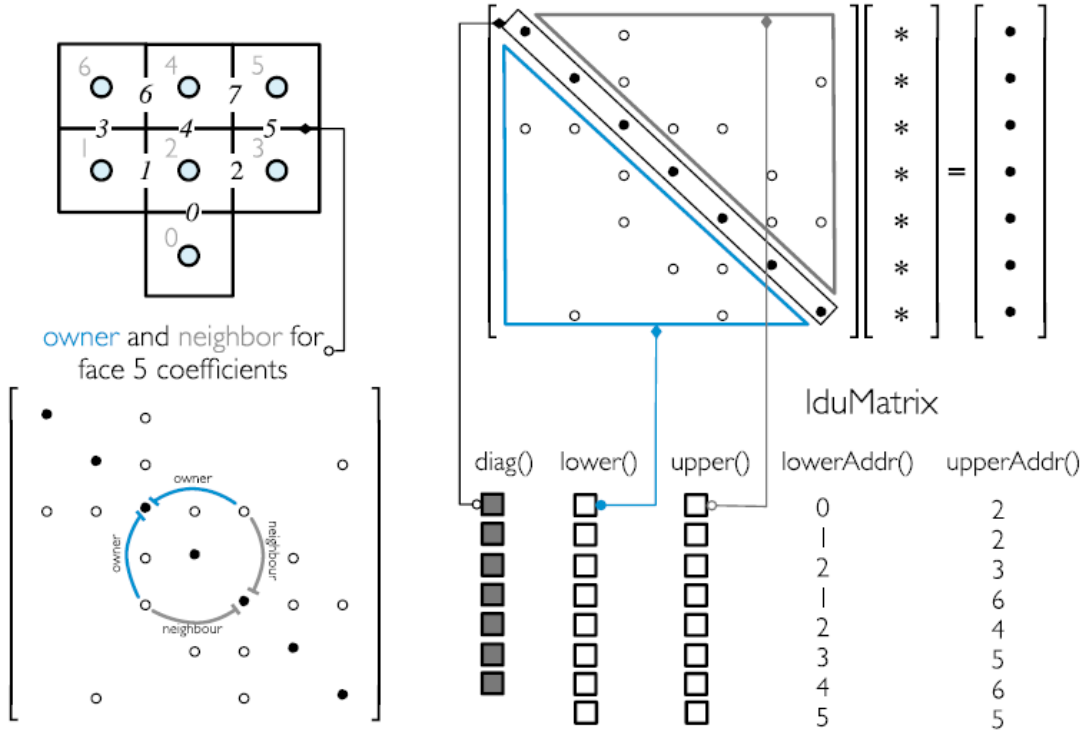
SOME NOTES:

- the cell with the lower index is the owner while the neighbor is the cell with the higher index;
- for boundary faces the owner is always the cell to which the face is attached while no neighbor is defined by setting the neighbour index to -1.
- The list of owner or neighbor indices thus define the order in which the cell-to-cell coefficients are assembled for the various integral operators.

Solving a Linear System in the FV framework



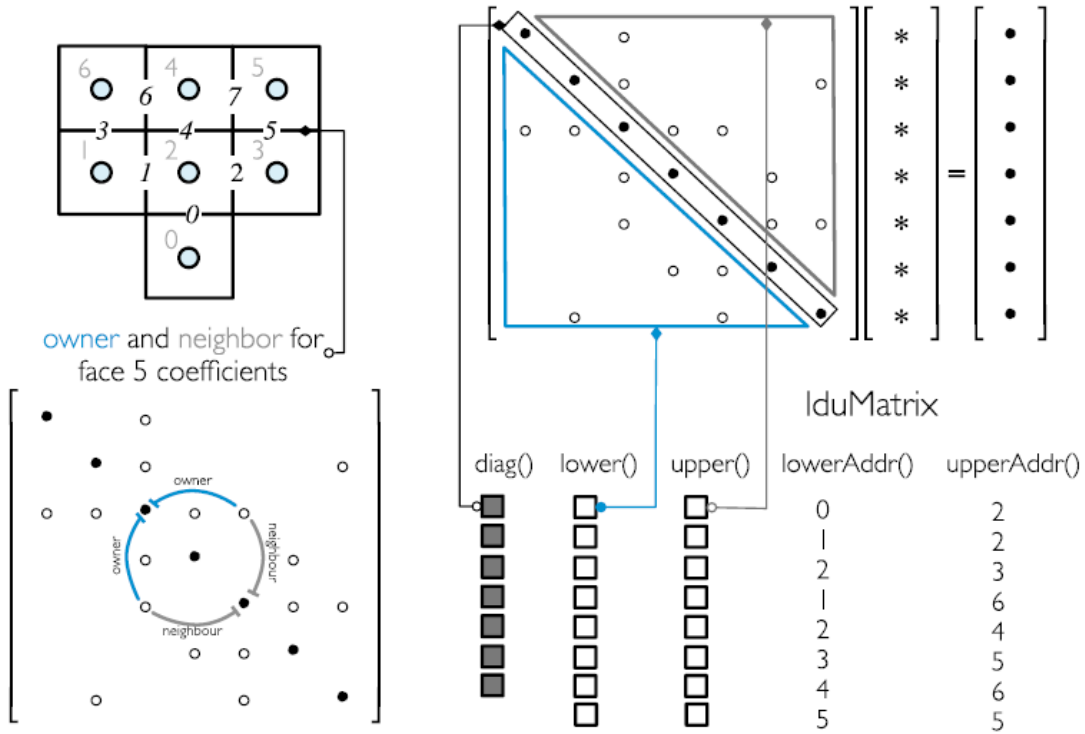
Face-addressing scheme in OpenFOAM



OpenFOAM makes use of:

- face addressing in its discretization loops and coefficients storage.
- arbitrary polyhedral elements in its meshes: coefficients are stored following the interior face ordering, with access to elements (cells) and their coefficients, based on the owner/neighbor indices associated with interior faces.

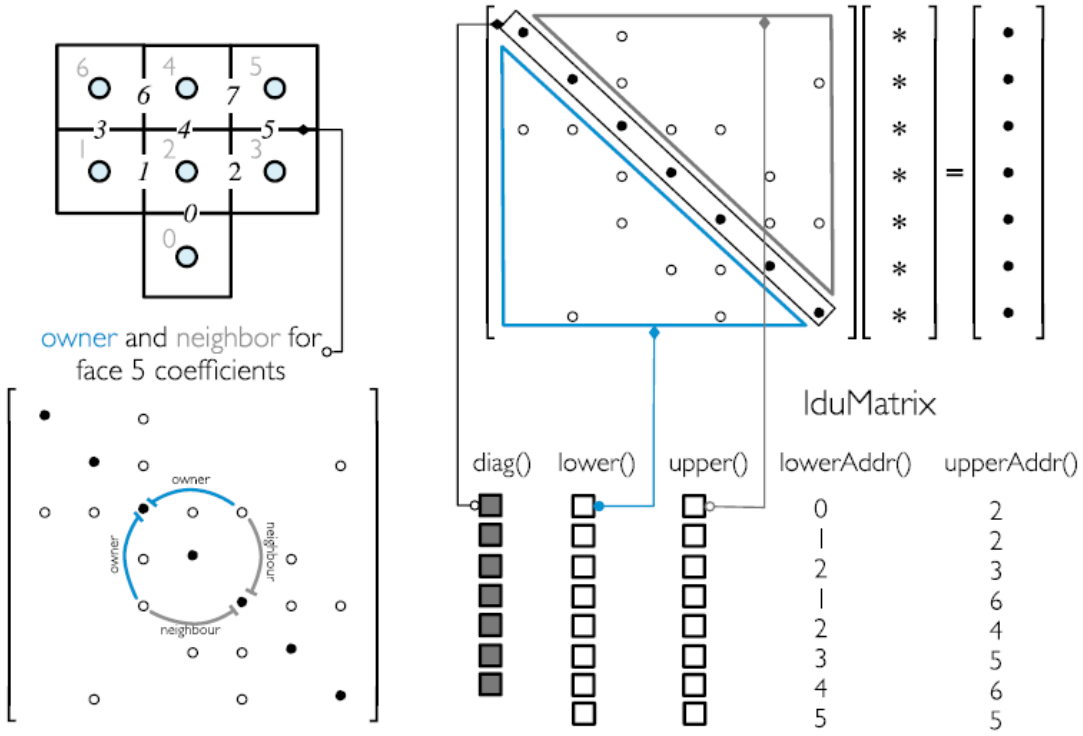
`lduAddressing` in OpenFOAM



lduAddressing is implemented in the `lduMatrix` class that includes 5 arrays representing:

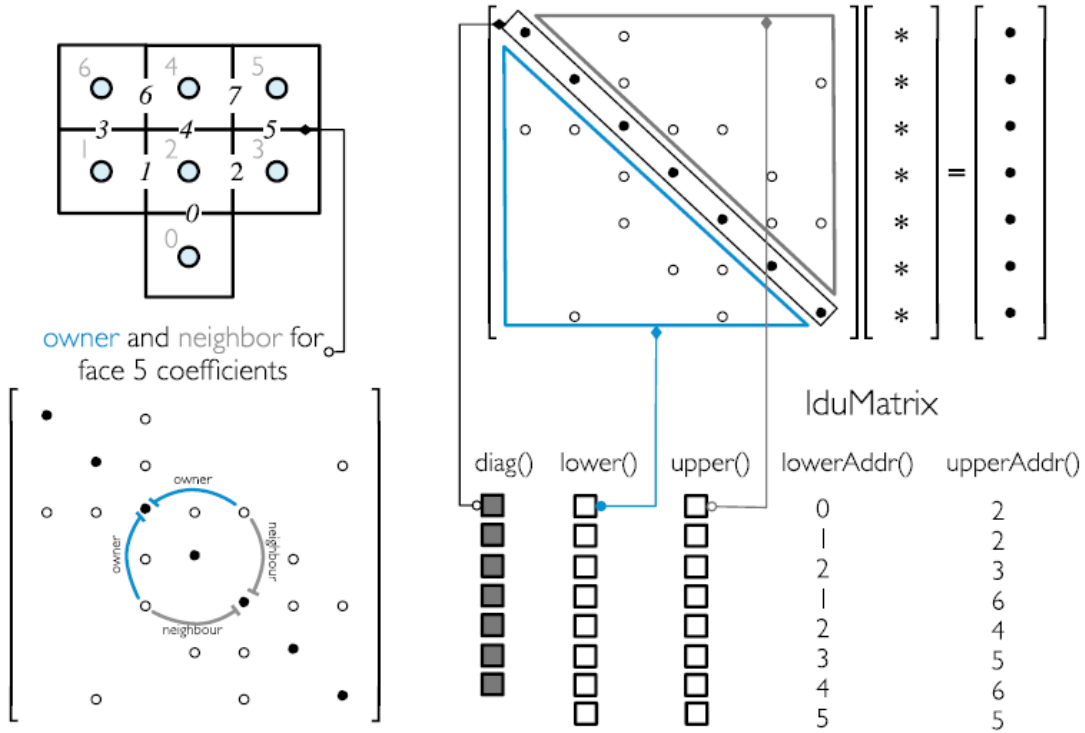
- diagonal, upper and lower coefficients
- lower and upper indices of the face owner and neighbor

The class `lduMatrix` in OpenFOAM



- `owners` → lower triangular part of the matrix (lower addressing)
- `neighbors` → to the upper triangular part (upper addressing)

The class `lduMatrix` in OpenFOAM



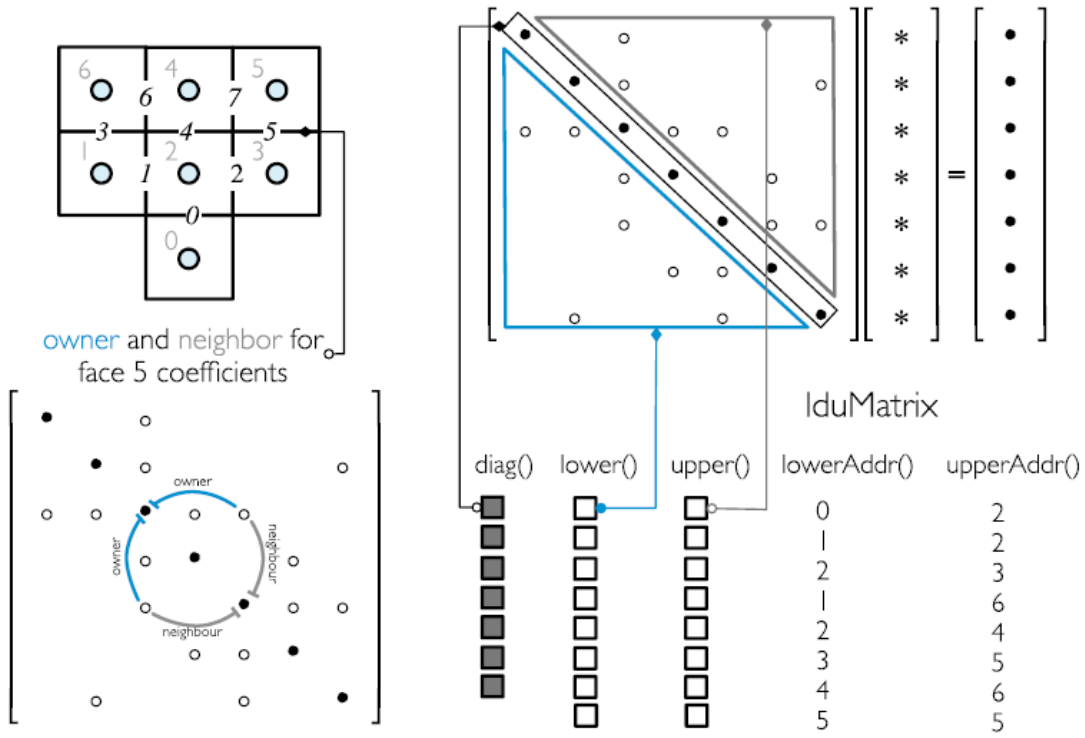
For a given a face for an **OWNER** cell:

- lower addressing → COLUMN where face flux is stored in the matrix
- upper addressing → ROW where face flux is stored in the matrix

For a given a face for an **NEIGHBOUR** cell:

- lower addressing → ROW where face flux is stored in the matrix
- upper addressing → COLUMN where face flux is stored in the matrix

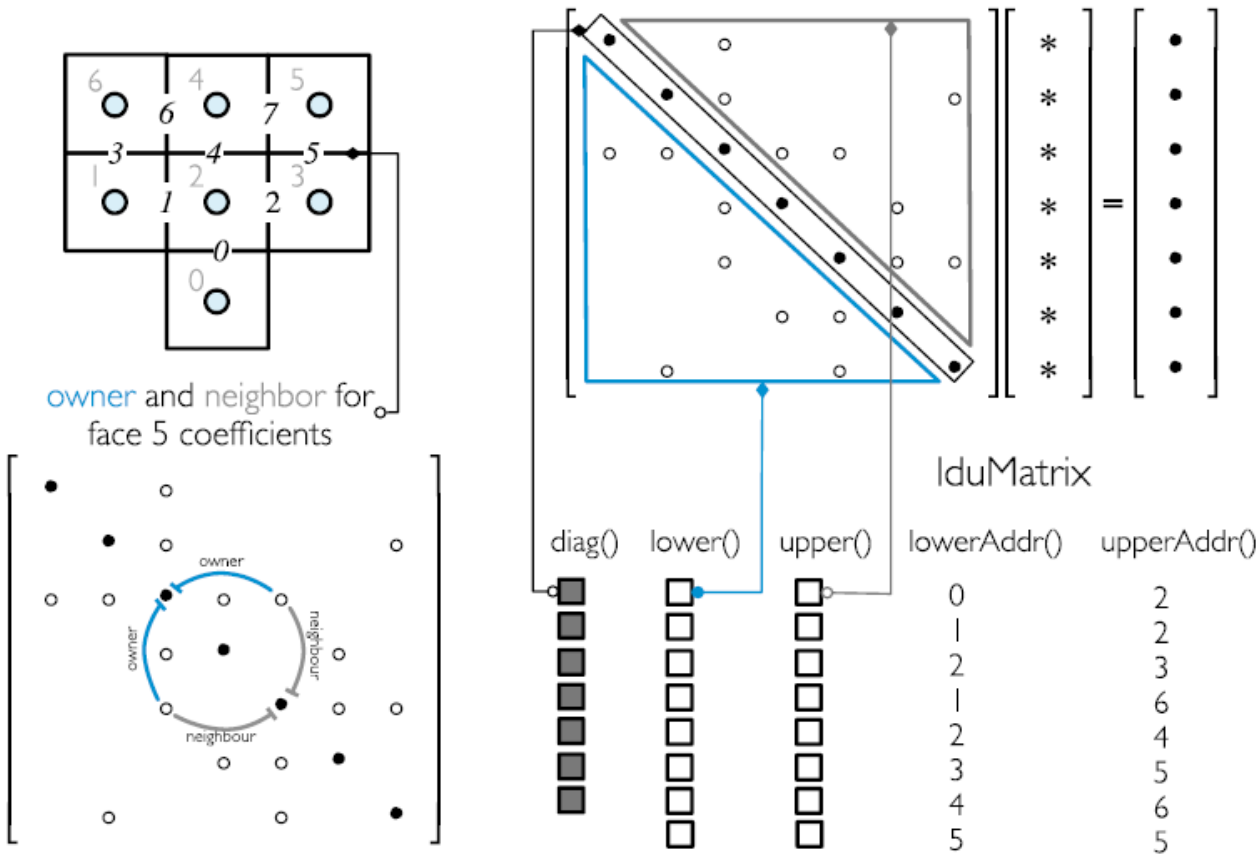
The class `lduMatrix` in OpenFOAM



In `lowerAddr()` and `upperAddr()`, the face number is the position of the owner or neighbor in the array plus one (since numbering starts with 0). The owner and neighbor of each face are displayed in the `lowerAddr()` and `upperAddr()` array, respectively.

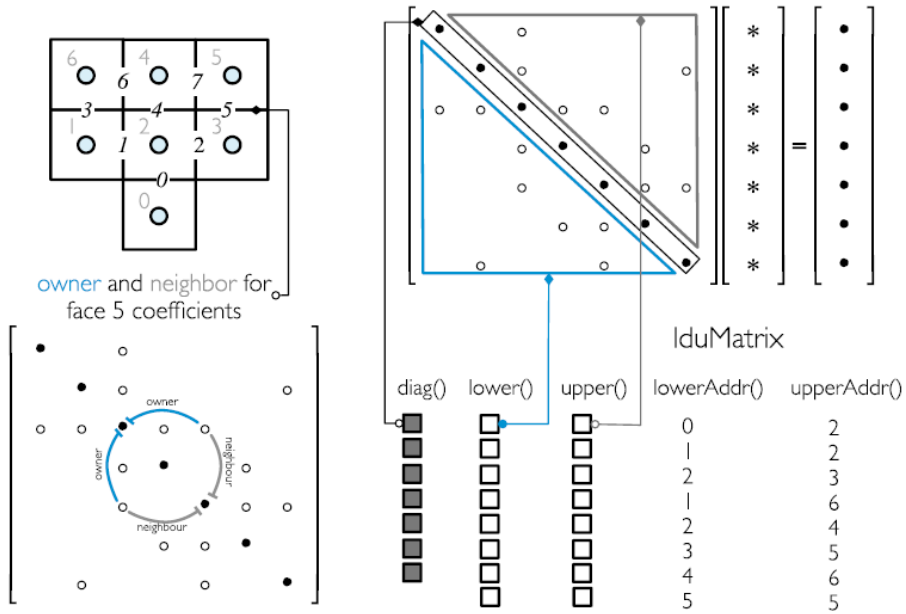
Example

lduAddressing



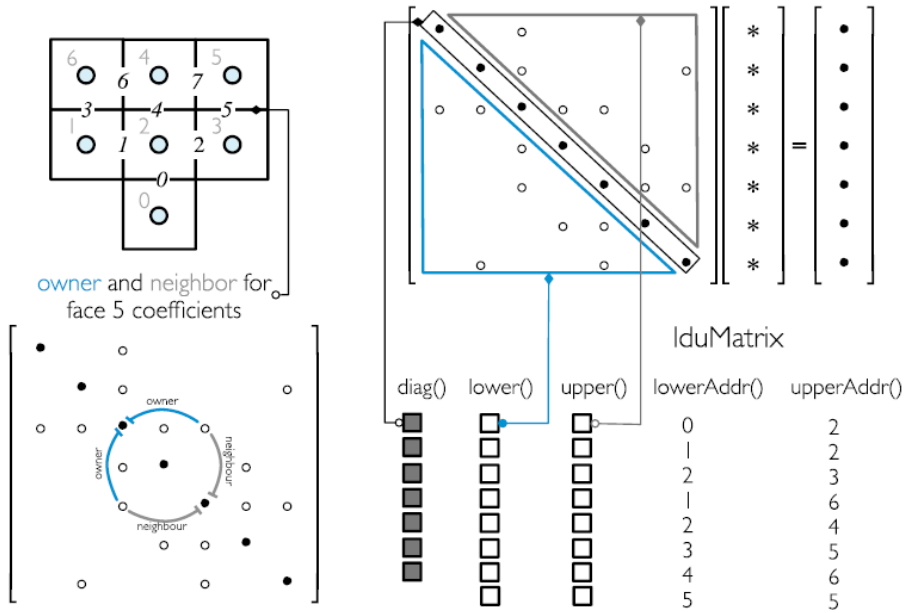
The lower, diagonal and upper coefficients are all scalarFields (basically a list), but they are indexed differently. The diagonal coefficients are indexed by cell index; whereas the upper and lower triangles are indexed by face index. In order to get the indices to match, an addressing array is provided for the lower and upper triangles to translate their face index into a corresponding cell index. This addressing array is called an lduAddressing array.

`lduAddressing`: example



- Considering **internal face number 4**, for example, its related information is stored in the *fifth row* (in C++ indexing starts from 0) of the `lower()`, `upper()`, `lowerAddr()`, and `upperAddr()` array, respectively.

`lduAddressing`: example



Data stored indicate that for face n. 4:

- its owner is cell number 2 ($\rightarrow$`lowerAddr()` array)
- its neighbor is cell number 4 ($\rightarrow$`upperAddr()` array)
- the coefficient multiplying ϕ_4 in the algebraic equation for cell number 2 is stored in the fifth column of the array `upper()`
- the coefficient multiplying ϕ_2 in the algebraic equation for cell number 4 is stored in the fifth row of the array `lower()`.

lduAddressing: IMPORTANT NOTE!

lduAdressing provides information about the addresses of the off diagonal coefficients in relation to the faces to which they are related.

$\rightarrow$ This means that while the computational efficiency for various operations on the matrix is high when they are mainly based on loops over all faces of the mesh, **direct access to a specific row-column matrix element is difficult and inefficient**.

One example is the summation of the off-diagonal coefficients for each row given by

$$a_C = \sum_{n \approx nb(C)} a_n$$

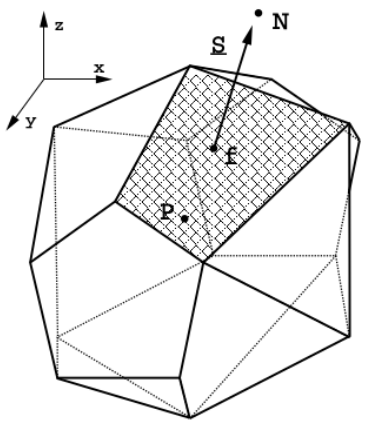
In this case the use of face addressing does not allow direct looping over the off-diagonal elements of each row and performing such operation requires looping over all elements because it can only be done, following a face based approach:

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

The Finite Volume (FV) method

The FV method uses the integral form of the conservation equations as its starting point

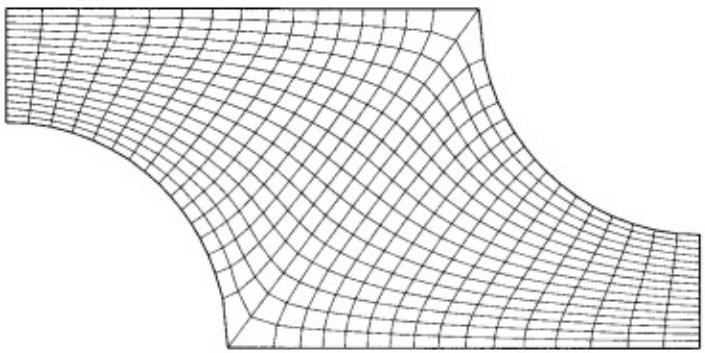
- the solution domain is divided into a finite number of continuous control volumes, where conservation equations are applied.
- variable values are calculated at the centroid of each CV
- FV method can accommodate any type of grid;
- the method is conservative by construction, as long as surface integrals (representing convective and diffusive fluxes) are the same for the CVs sharing the boundary.

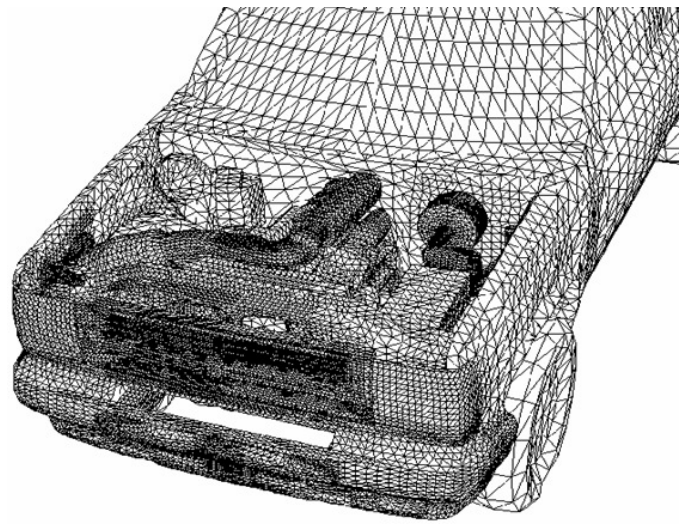


DISADVANTAGES: methods of order higher than 2^{nd} are difficult to implement in a FV framework.

The discretization process

The numerical solution of a partial differential equation consists of finding the values of the dependent variable ϕ at specified points called grid elements, or grid nodes, resulting from the discretization of original geometry into a set of non overlapping discrete elements; this last process is known as *meshing*.





In OpenFOAM, the resulting nodes or variables are generally positioned at cell centroids (**collocated arrangement of the variables**).

The discretization process

The distribution of ϕ is discretized (**discretization process**); specific **discretization methods** are then available to discretize equations from the continuum form.

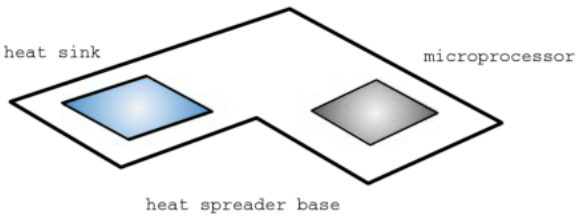
Change of ϕ over Δt in the Control Volume (CV)	+	Surface flux of ϕ over time Δt across the control volume	=	Source/sink terms over Δt in the control volume

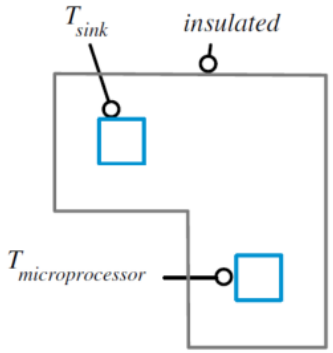
- The discrete values of ϕ are typically computed by solving a set of algebraic equations relating the values at neighboring grid elements to each other; these discretized algebraic equations are derived from the conservation equation governing ϕ:

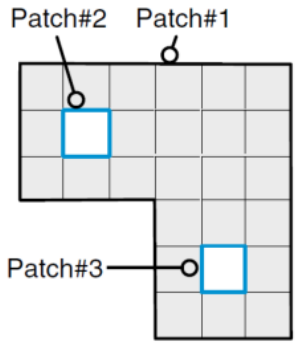
$$\frac{\partial\left(\rho\phi\right)}{\partial t}+\nabla\cdot\left(\rho\boldsymbol{v}\phi\right)=\nabla\cdot\left(\Gamma^{\phi}\nabla\phi\right)+Q^{\phi}$$

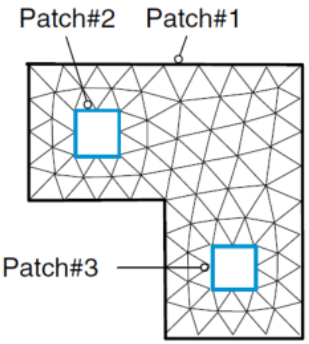
Once the values of ϕ are computed, the data is processed to extract any needed information.

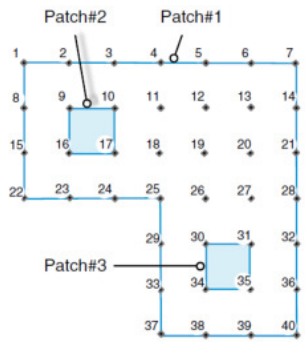
Domain Discretization

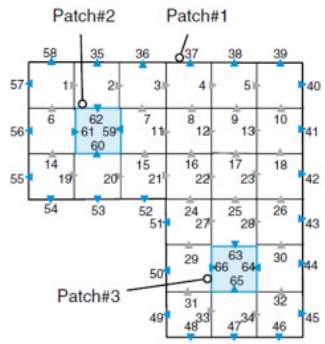


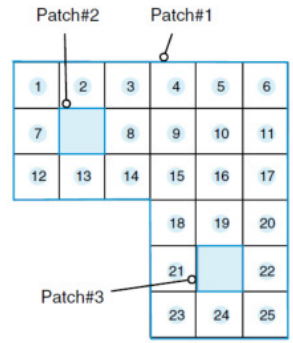






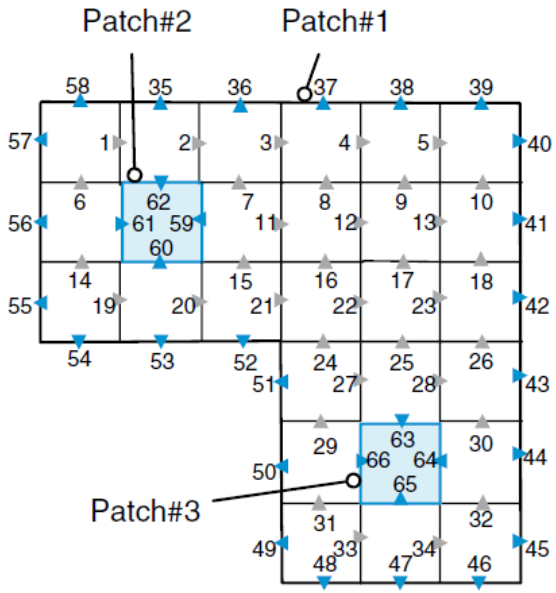


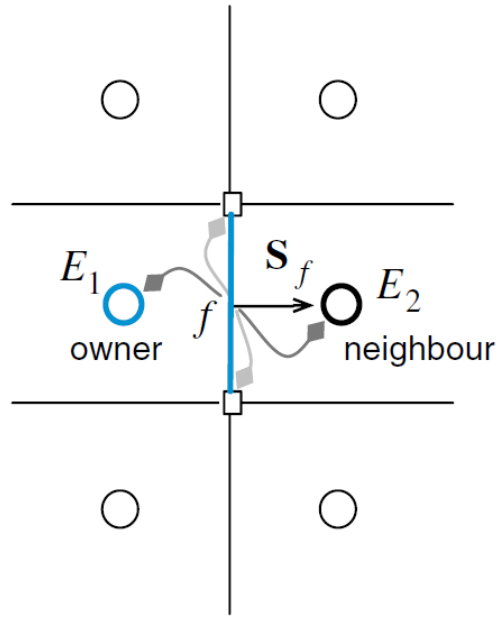




- The geometric domain is subdivided into discrete non-overlapping cells or elements (mesh)
- The mesh is composed of *discrete elements* (`cells` in OpenFOAM) defined by a set of *vertices* (`points`) and bounded by *faces* (`faces`)

Domain Discretization: faces





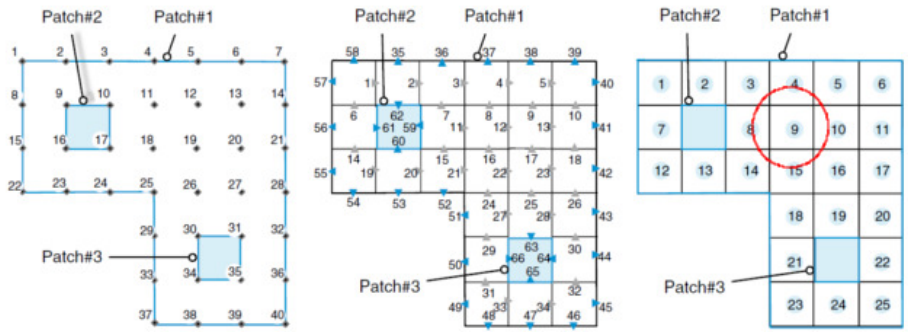
The cell faces in the mesh can be divided into two groups:

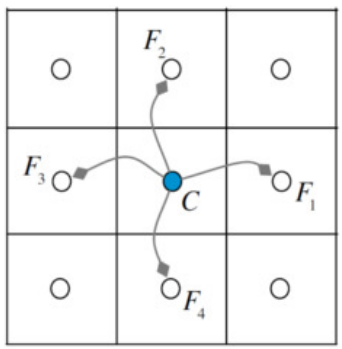
- **internal faces** (between two control volumes)
- **boundary faces**, which coincide with the boundaries of the domain.

Mesh Topology in OpenFOAM

Cell connectivity

During discretization, the partial differential equations are integrated over each element in the mesh resulting in a set of algebraic equations with each one linking the value of the variable at an element to the values at its neighbors.





Element 9 Connectivity

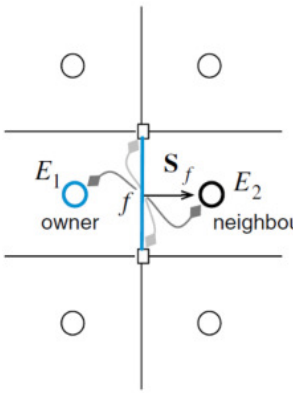
Neighbours [10 4 8 15]
Faces [12 8 11 16]
Vertices [19 11 12 18]

- **the algebraic equations are then assembled** into global matrices and vectors;
- the assembly of the equations in the global matrix accounts for the actual indices of the elements: to do this, **topological information about elements, faces, and vertices** (that are represented in terms of *connectivity lists*) are needed.

The integration of the equations over each element is referred to as **local assembly** while the construction of the overall system of equations from these contributions is referred to as **global assembly**. Thus, while the discretization of the equations is derived in terms of neighbor elements, the assembly of the equations in the global matrix accounts for the actual indices of the elements.

Mesh Topology in OpenFOAM

Face connectivity

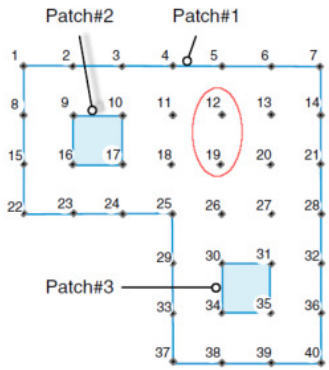


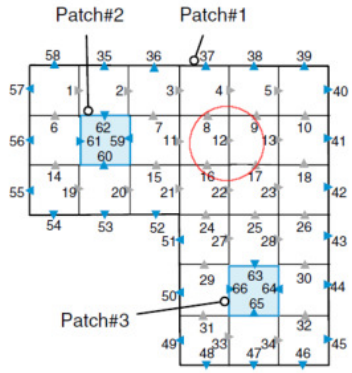
Face 12 Connectivity

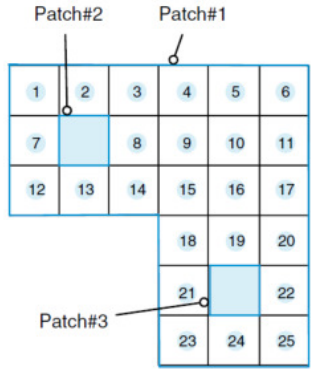
Element1 9

Element2 10

Vertices [19 12]





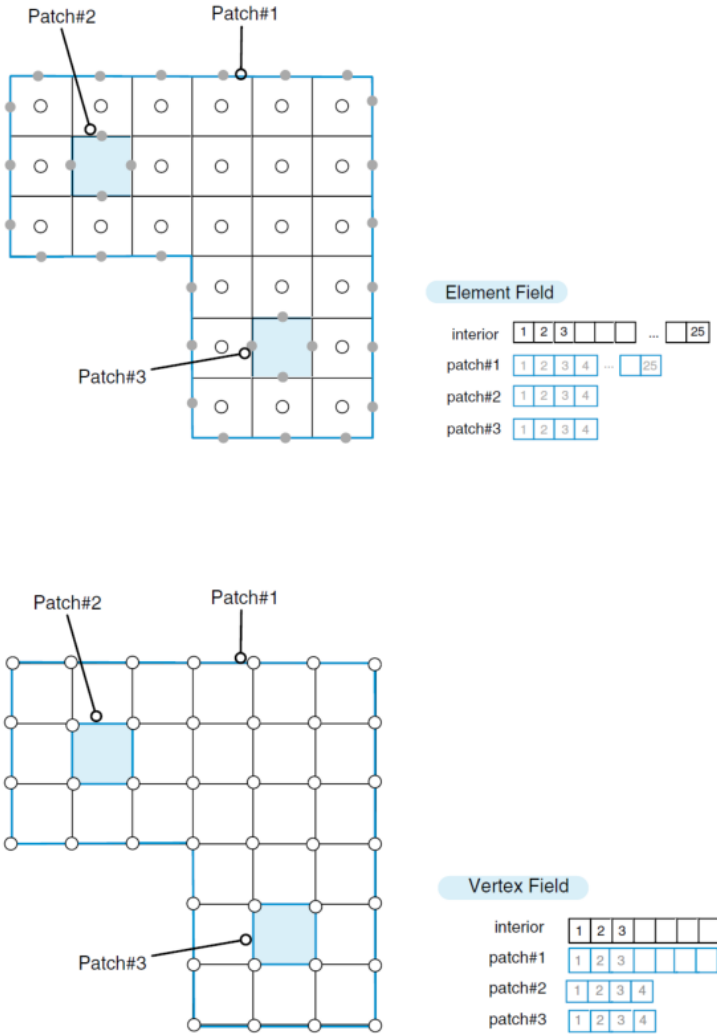


IMPORTANT: in OpenFOAM the face area vector $\mathbf{S}_f$ is constructed for each face in such a way that it points outwards from the cell with the lower label, is normal to the face and has the magnitude equal to the area of the face.

- the cell with the lower label is called the **OWNER** of the face its label is stored in the "owner" array;
- the label of the other cell (**NEIGHBOUR**) is stored in the "neighbour" array.

Boundary face area vectors point outwards from the computational domain boundary faces are "owned" by the adjacent cells.

Connectivity Lists in OpenFOAM

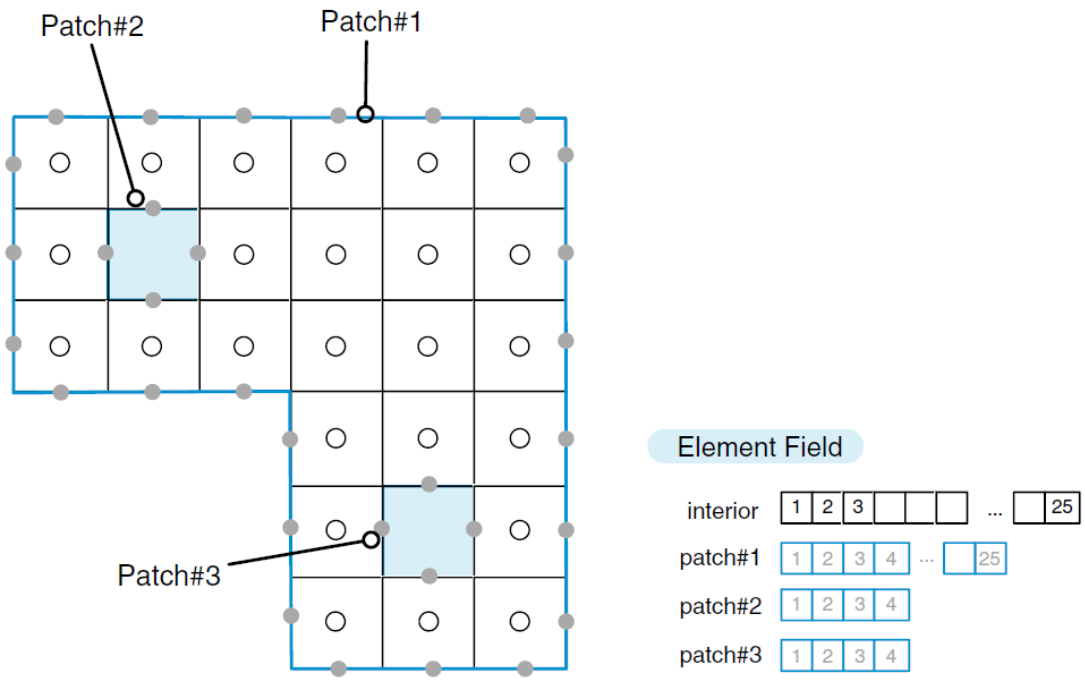


Equation Discretization

The governing partial differential equations, are transformed into a set of algebraic equations, one for each element in the computational domain. These algebraic equations are then assembled into a global matrix and vectors that can be expressed in the form

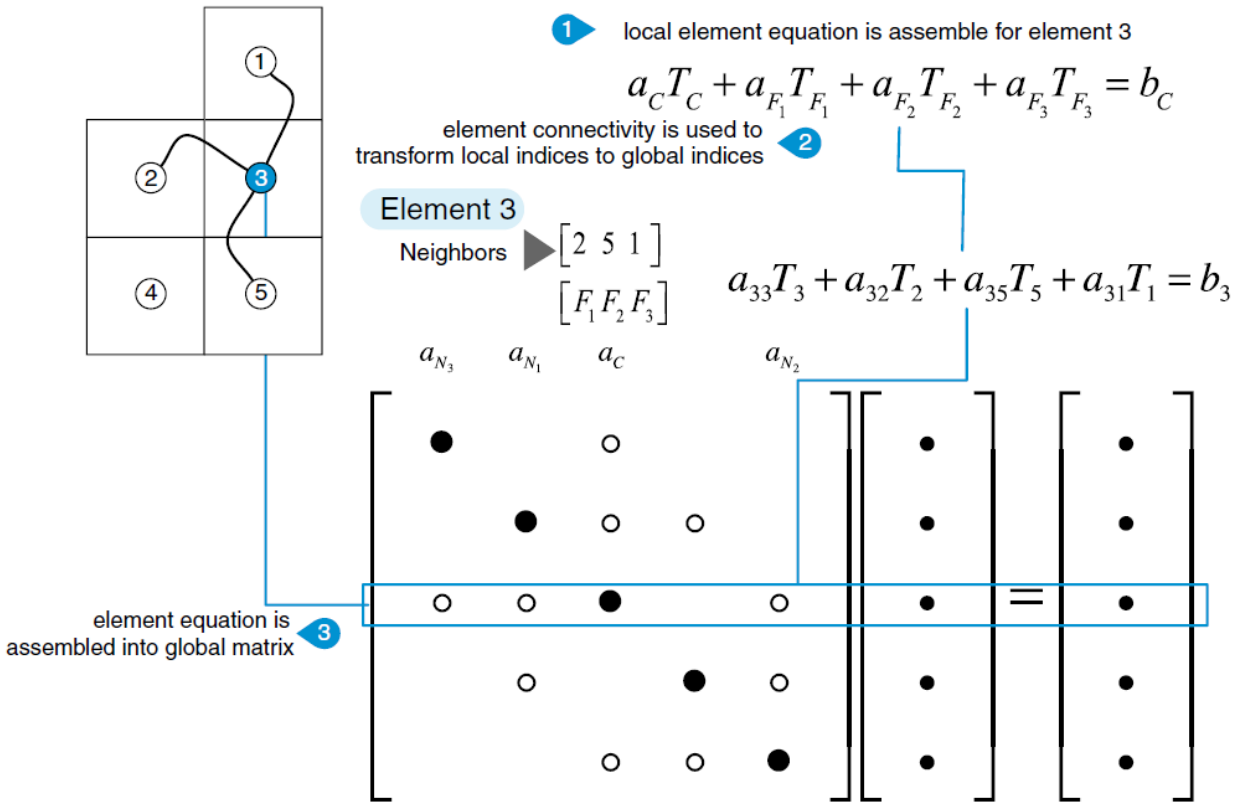
$$\boldsymbol{A}\,[T] = \boldsymbol{b}$$

where the *unknown variable T* is defined at each control volume CV.



Building the Coefficient Matrix

As already mentioned, the integration of the equations over each element is referred to as **local assembly**, while the construction of the overall system of equations from these contributions is referred to as **global assembly**.



Thus, while the discretization of the equations is derived in terms of neighbor elements, the assembly of the equations in the global matrix accounts for the actual indices of the elements.

Solution of the Linear System

Direct Methods

In a direct method the solution to the system of equations is obtained by

$$[T] = \boldsymbol{A}^{-1}\boldsymbol{b}$$

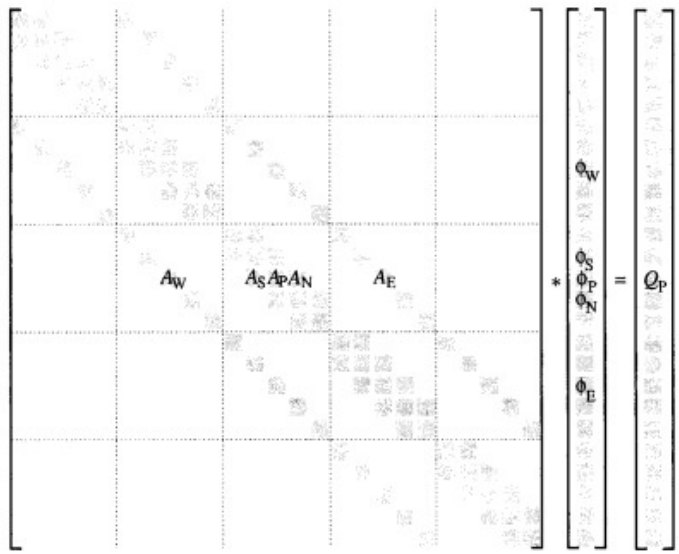
Therefore a solution for [T] is guaranteed if A^{-1} can be found.

$$\begin{bmatrix} \phi_1 \\ \phi_2 \\ \vdots \\ \vdots \\ \phi_n \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \dots & a_{nn} \end{bmatrix}^{-1} \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ \vdots \\ b_n \end{bmatrix}$$

However, the operation count for the inversion of an N×N matrix is O(N^3), which is computationally expensive. Consequently, inversion is almost never employed in practical problems.

FV methods: Linear Equation Systems

The result of the discretization process is a system of algebraic equations, which are linear or non linear according to the nature of the partial differential equation(s) from which they are derived.



- linear system must be solved implictly $\rightarrow$ inverting the coefficient matrix is too expensive!
- preconditioning is needed
- linear solvers (multigrid, Conjugate Gradient for symmetric matrices; bi-conjugate gradients for non-symmetric matrices) lead to different performance

FV methods: Linear Equation Systems

The resulting coefficient matrix A is a sparse matrix:

i	-1-	-2-	-3-	-4-	-5-	-6-	-7-	-8-	-9-
-1-	X	N				N			
-2-	O	X	N		N				
-3-		O	X	N					
-4-			O	X	N				N
-5-		O		O	X	N		N	
-6-	O				O	X	N		
-7-						O	X	N	
-8-					O		O	X	N
-9-				O				O	X

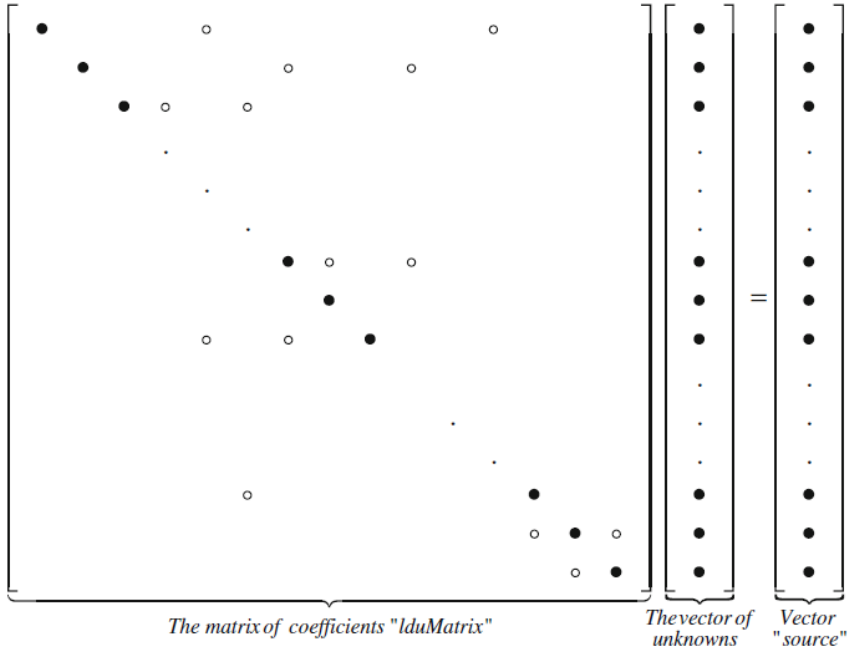
Owner, neighbour and diagonal coefficients are defined as:

$$A_{ij} = \begin{cases} \text{owner}, & i > j \\ \text{diagonal}, & i = j \\ \text{neighbour}, & i < j \end{cases}$$

Where:

- the coefficient matrix A must be **diagonal dominant**: boundary conditions influences off-diagonal terms;
- with this definition, the lower triangle has the owner coefficients; and the upper triangle has the neighbour coefficients.

Building the Linear Matrix in OpenFOAM®...



In OpenFOAM, the class `fvMatrix` builds the linear matrix; this happens anytime you write something like:

```
tmp<fvVectorMatrix> tUEqn
(
    fvm::ddt(rho, U) + fvm::div(phi, U)
  + MRF.DDt(rho, U)
  + turbulence->divDevRhoReff(U)
 ==
    fvOptions(rho, U)
);
```

Implicit methods

Solution of the Discretized Equations

In a linear system AX=B:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \dots & a_{nn} \end{bmatrix} \begin{bmatrix} \phi_1 \\ \phi_2 \\ \vdots \\ \vdots \\ \phi_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ \vdots \\ b_n \end{bmatrix}$$

- each row represents an equation defined over one cell of the computational domain, and the non-zero coefficients are those related to the neighbors of that element;
- the coefficient a_{ij} measures the strength of the link between the value of ϕ_i at the centroid of the control volume and its neighbors.

Iterative Methods: basic concepts

Iterative Methods

Any system of equations can be solved by Gauss elimination or LU decomposition. Unfortunately:

- the triangular factors of sparse matrices are not sparse, so the cost of these methods is quite high;
- the discretization error is usually much larger than the accuracy of the computer arithmetic so there is no reason to solve the system that accurately;
- solution to somewhat more accuracy than that of the discretization scheme suffices.

This leaves an opening for iterative methods. In an iterative method, one guesses a solution, and uses the equation to systematically improve it. **If each iteration is cheap and the number of iterations is small, an iterative solver may cost less than a direct method.** On the other hand:

- iterative methods show the advantage to require **lower computational cost per iteration** and **lower memory** if compared to direct methods.
- **ITERATIVE methods for linear systems are more effective**, in particular considering that for the discretization methods of interest here the **coefficient matrix A** is:
  - **sparse** for unstructured grids
  - **banded** for structured meshes

Iterative Methods

To unify the presentation of iterative methods, the coefficient matrix will be written in the following form:

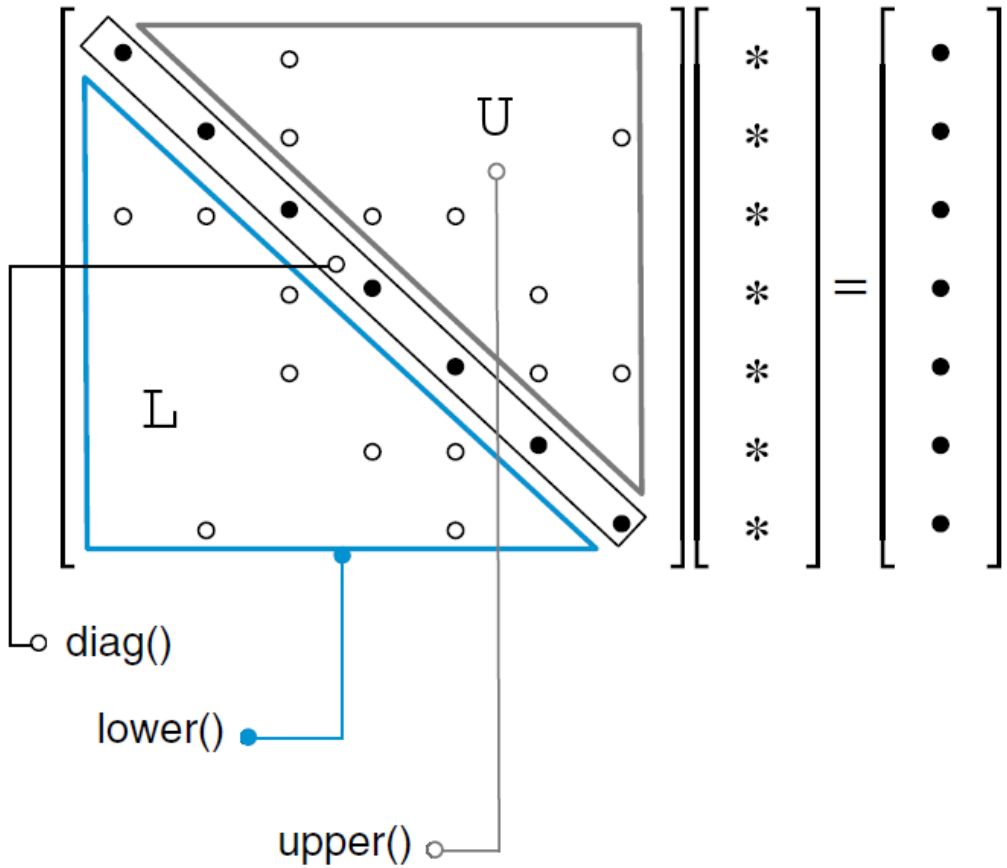
$$\boldsymbol{A} = \underbrace{\begin{bmatrix} a_{11} & 0 & 0 & \dots & 0 \\ 0 & a_{22} & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & a_{nn} \end{bmatrix}}_{\textbf{D (diagonal)}} + \underbrace{\begin{bmatrix} 0 & 0 & 0 & \dots & 0 \\ a_{21} & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & a_{n,n-1} & 0 \end{bmatrix}}_{\textbf{L (lower)}} + \underbrace{\begin{bmatrix} 0 & a_{12} & a_{13} & \dots & a_{1n} \\ 0 & 0 & a_{23} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 0 \end{bmatrix}}_{\textbf{U (upper)}}$$

$$\boldsymbol{A} = \boldsymbol{D} + \boldsymbol{L} + \boldsymbol{U}$$

being:

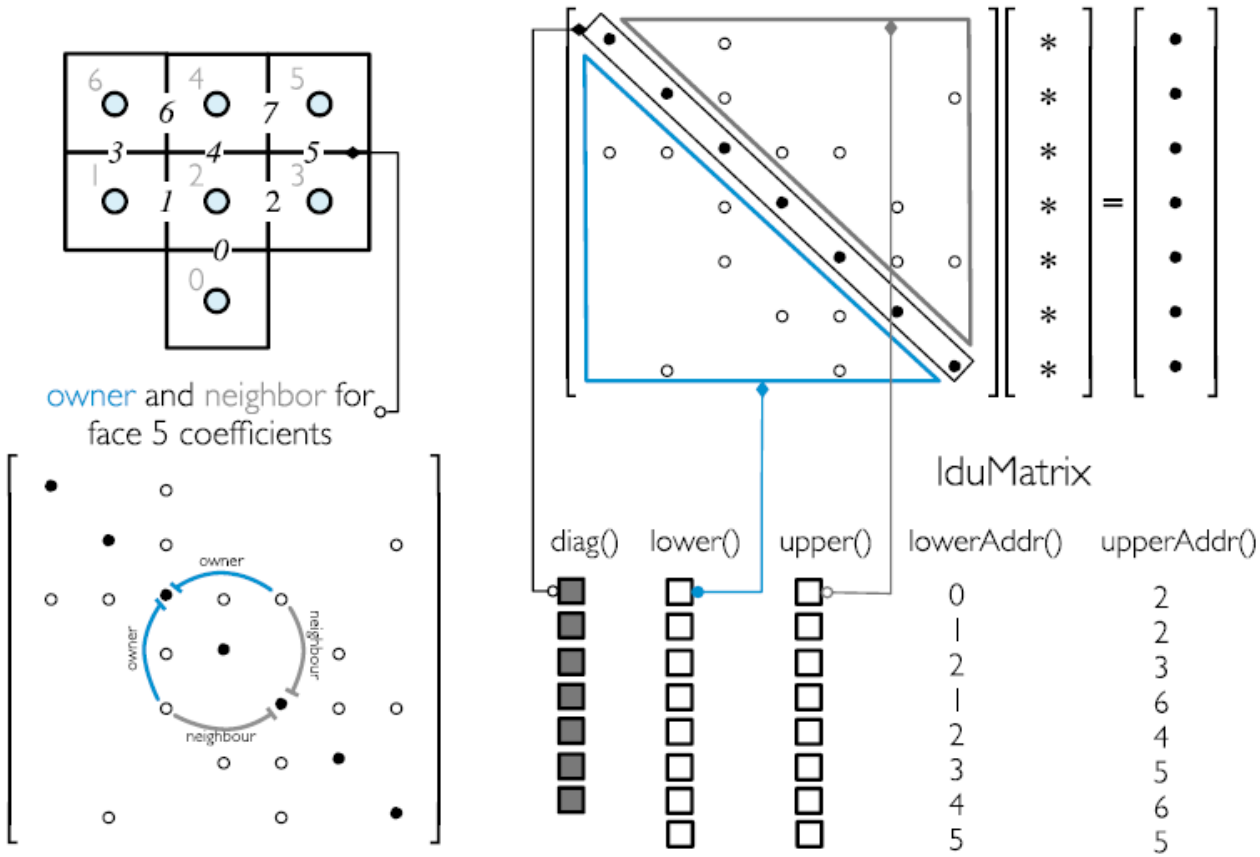
- $\boldsymbol{D}$ = diagonal matrix
- $\boldsymbol{L}$ = strictly lower matrix
- $\boldsymbol{U}$ = strictly upper matrix

Iterative Methods



$$\boldsymbol{A} = \boldsymbol{D} + \boldsymbol{L} + \boldsymbol{U}$$

lduAddressing



The lower, diagonal and upper coefficients are all scalarFields (basically a list), but they are indexed differently. The diagonal coefficients are indexed by cell index; whereas the upper and lower triangles are indexed by face index. In order to get the indices to match, an addressing array is provided for the lower and upper triangles to translate their face index into a corresponding cell index. This addressing array is called an lduAddressing array.

Iterative Methods

Iterative methods for solving a linear system of the type $\boldsymbol{A}\phi = \boldsymbol{b}$ compute a series of solutions of $\phi^{(n)}$ that, if certain conditions are satisfied, converge to the exact solution ϕ.

Thus, for the solution, a starting point (**initial conditions**) ϕ^0 is chosen, it follows:

$$\phi^0 \rightarrow \phi^{(n-1)} \rightarrow \phi^{(n)}$$

How can this be done? If the coefficient matrix A is decomposed:

$$\boldsymbol{A} = \boldsymbol{M} - \boldsymbol{N}$$

$$(\boldsymbol{M} - \boldsymbol{N})\phi = \boldsymbol{b} \quad \rightarrow \quad \boldsymbol{M}\boldsymbol{\phi}^{(n)} = \boldsymbol{N}\boldsymbol{\phi}^{(n-1)} + \boldsymbol{b}$$

hence:

$$\boldsymbol{\phi}^{(n)} = \boldsymbol{M}^{-1}\boldsymbol{N}\boldsymbol{\phi}^{(n-1)} + \boldsymbol{M}^{-1}\boldsymbol{b}$$

which can be written as:

$$\boxed{\boldsymbol{\phi}^{(n)} = \boldsymbol{B}\boldsymbol{\phi}^{(n-1)} + \boldsymbol{C}\boldsymbol{b}}$$

Different choices of these matrices define different iterative methods!

Iterative Methods: properties

To guarantee convergence, an iterative method of the form:

$$\boldsymbol{\phi}^{(n)} = \boldsymbol{B}\boldsymbol{\phi}^{(n-1)} + \boldsymbol{C}\boldsymbol{b}$$

should possess the following characteristics:

1) at convergence $\boldsymbol{\phi} = \boldsymbol{B}\boldsymbol{\phi} + \boldsymbol{C}\boldsymbol{b}$, which leads to:

$$\boldsymbol{C}^{-1}(\boldsymbol{I} - \boldsymbol{B})\boldsymbol{\phi} = \boldsymbol{b} \quad \rightarrow \quad \boldsymbol{A} = \boldsymbol{C}^{-1}(\boldsymbol{I} - \boldsymbol{B}) \quad \rightarrow \quad \boldsymbol{B} + \boldsymbol{C}\boldsymbol{A} = \boldsymbol{I}$$

2) **CONVERGENCE:** starting from some guess $\boldsymbol{\phi}^0 \neq \boldsymbol{\phi}$, the method should guarantee that $\boldsymbol{\phi}^n$ will converge to $\boldsymbol{\phi}$ as n increases.

Iterative Methods: stopping criterion

3) it must include some type of a **stopping criterion**. Most of used criteria consider the variation of the norm of the residual error:

$$\boldsymbol{r}^n = \boldsymbol{A}\boldsymbol{\phi}^{(n)} - \boldsymbol{b}$$

Possible convergence criteria are:

a) the *maximum residual over the domian* is lower that an threshold ε:

$$\max_{i=1}^{n} \left| b_i - \sum_{j=1}^{N} a_{i,j}\phi_j^{(n)} \right| \leq \varepsilon$$

b) the *root mean residual* be smaller than ε:

$$\sum_{i=1}^{n} \left(b_i - \sum_{j=1}^{N} a_{ij}\phi_j^{(n)} \right)^2 \leq \varepsilon$$

c) the *maximum normalized difference between two consecutive iterations* is lower that an threshold ε:

$$\max_{i=1}^{n} \left| \frac{\phi_i^{(n)} - \phi_i^{(n-1)}}{\phi_i^{(n)}} \right| \cdot 100 \leq \varepsilon$$

Iterative Methods: common features

Considering the system of equations described by:

$$\boldsymbol{\phi}^{(n)} = \boldsymbol{B}\boldsymbol{\phi}^{(n-1)} + \boldsymbol{C}\boldsymbol{b}$$

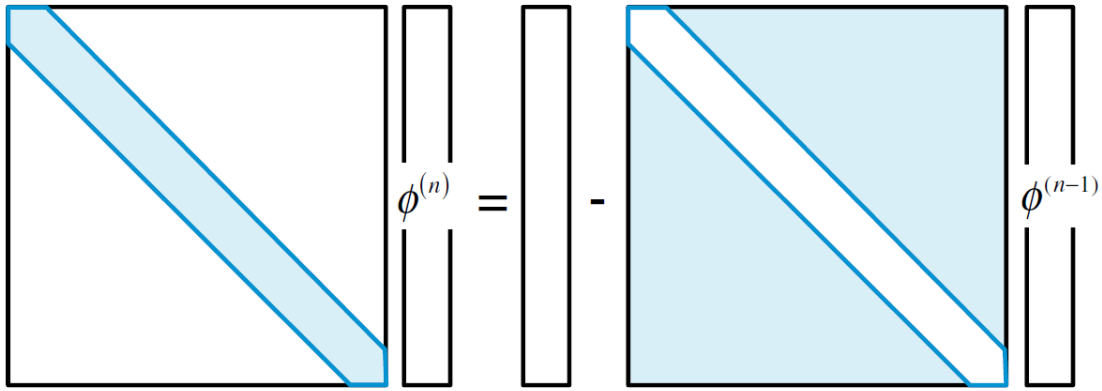
An iterative solution is achieved as follows:

- the solution process starts by assigning guessed values to the unknown vector $\boldsymbol{\phi}$; guessed values are used to calculate new estimates and computations proceed until a new estimate is computed;
- Iterations continue until the changes in the predictions between two consecutive iterations drop below a vanishing value or until a preset convergence criterion is satisfied;
- once this happens the final solution is reached.

The rate of convergence of iterative methods depends on the spectral properties of the iteration matrix $\boldsymbol{B}$. Please remember this when we will talk about **preconditioning**.

Solution of Linear Systems

Jacobi Method



If the diagonal elements of the coefficient matrix A are nonzero, then each equation i (i.e. row) can be used to solve ϕ_i:

$$\phi_j^{(n)} = \frac{1}{a_{ij}} \left(b_i - \sum_{j=1}^{N} a_{ij} \phi_j^{(n-1)} \right) \quad j \neq i, \quad i = 1 \ldots N$$

which can be written as

$$\phi_j^{(n)} = -\boldsymbol{D}^{-1} \left(\boldsymbol{L} + \boldsymbol{U} \right) \phi_j^{(n-1)} + \boldsymbol{D}^{-1} \boldsymbol{b}$$

The Jacobi method converges as long as $\rho\left(-\boldsymbol{D}^{-1}\left(\boldsymbol{L}+\boldsymbol{U}\right)\right) < 1$. This condition is satisfied for a large class of matrices (including *diagonally dominant*) for which:

$$\sum_{j=1}^{N} |a_{ij}| < |a_{ii}| \quad i \neq j$$

Preconditioning

Given the system of equations described by:

$$\boldsymbol{\phi}^{(n)} = \boldsymbol{B}\boldsymbol{\phi}^{(n-1)} + \boldsymbol{C}\boldsymbol{b}$$

- Since the rate of convergence of iterative methods depends on the spectral properties of the iteration matrix $\boldsymbol{B}$, a transformation of the system of equations into an equivalent one that has the same solution, but better spectral properties, favors a faster convergence to the final iterative solution than in the original system.

- **A preconditioner is defined as a matrix that effects such a transformation**. A preconditioning matrix $\boldsymbol{P}$ is defined such that the system:

  $$\boldsymbol{P}^{-1}\boldsymbol{A}\boldsymbol{\phi} = \boldsymbol{P}^{-1}\boldsymbol{b}$$

  has the same solution as the original system $\boldsymbol{A}\boldsymbol{\phi} = \boldsymbol{b}$, but the spectral properties of its coefficient matrix $\boldsymbol{P}^{-1}\boldsymbol{A}$ are more conducive.

- In defining the preconditioner P, the difficulty is to find a matrix that approximates A^{-1} and is easy to invert (i.e., to find P^{-1}) at a reasonable cost.

Preconditioning and Iterative Methods

The system $\boldsymbol{AX} = \boldsymbol{b}$ is modified into:

$$\boldsymbol{P}^{-1}\boldsymbol{A}\boldsymbol{\phi} = \boldsymbol{P}^{-1}\boldsymbol{b}$$

so it follows:

$$\begin{aligned}\boldsymbol{\phi}^{(n)} &= \boldsymbol{B}\boldsymbol{\phi}^{(n-1)} + \boldsymbol{C}\boldsymbol{b} \\ &= \boldsymbol{P}^{-1}(\boldsymbol{P} - \boldsymbol{A})\boldsymbol{\phi}^{(n-1)} + \boldsymbol{P}^{-1}\boldsymbol{b} \\ &= (\boldsymbol{I} - \boldsymbol{P}^{-1}\boldsymbol{A})\boldsymbol{\phi}^{(n-1)} + \boldsymbol{P}^{-1}\boldsymbol{b}\end{aligned}$$

which in residual form can be written as:

$$\begin{aligned}\boldsymbol{\phi}^{(n)} &= (\boldsymbol{I} - \boldsymbol{P}^{-1}\boldsymbol{A})\boldsymbol{\phi}^{(n-1)} + \boldsymbol{P}^{-1}\boldsymbol{b} \\ &= \boldsymbol{\phi}^{(n-1)} + \boldsymbol{P}^{-1}\left(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{\phi}^{(n-1)}\right) \\ &= \boldsymbol{\phi}^{(n-1)} + \boldsymbol{P}^{-1}\boldsymbol{r}^{(n-1)}\end{aligned} \tag{1}$$

From both equations it is now clear that the iterative procedure is just a fixed-point iteration on a preconditioned system associated with the decomposition, where spectral properties are:

$$\rho\left(\boldsymbol{I} - \boldsymbol{P}^{-1}\boldsymbol{A}\right) < 1$$

```
Time = 0.01

Courant Number mean:  0.0976825 max:  0.585607
smoothSolver:  Solving for Ux, Initial residual = 0.160686, Final residual = 6.83031e-06, No Iterations 19
smoothSolver:  Solving for Uy, Initial residual = 0.260828, Final residual = 9.65939e-06, No Iterations 18
DICPCG: Solving for p, Initial residual = 0.428925, Final residual = 0.0103739, No Iterations 22
time step continuity errors :  sum local = 0.000110788, global = 3.77194e-19, cumulative = -6.72498e-20
DICPCG: Solving for p, Initial residual = 0.30209, Final residual = 5.26569e-07, No Iterations 33
time step continuity errors :  sum local = 6.61987e-09, global = -2.74872e-19, cumulative = -3.42122e-19
ExecutionTime = 0.01 s ClockTime = 0 s
```

Solution algorithm in pisoFOAM

Linear Solver Control in OpenFOAM

Linear algebraic solvers in OpenFOAM are grouped under three main classes denoted by:

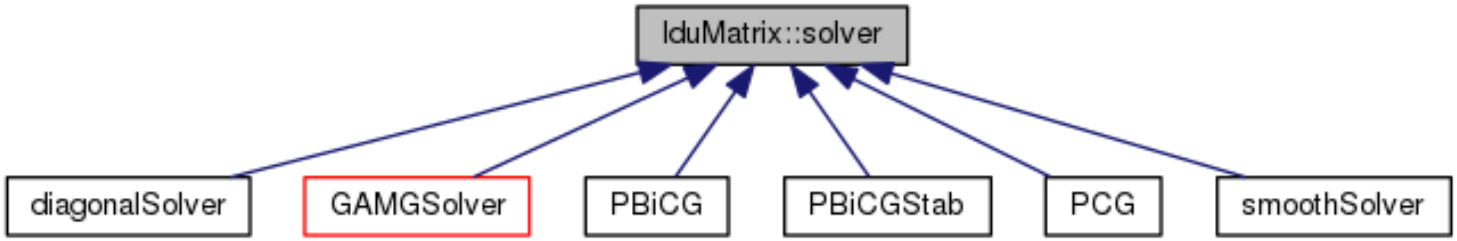
- `solvers`: include the implementation of conjugate gradient and multigrid algorithms
- `preconditioners`: implement the product $\boldsymbol{P}^{-1}\boldsymbol{r}$ (see Eq. 1)
- `smoothers`: advance the solution

The source codes of the linear algebraic solvers can be found at:

```
$FOAM_SRC/OpenFOAM/matrices/lduMatrix/
```

Smoothers and *preconditioners* are differentiated by relating to smoothers the fixed point relation and embedding them within the preconditioners framework.

lduMatrix::solvers



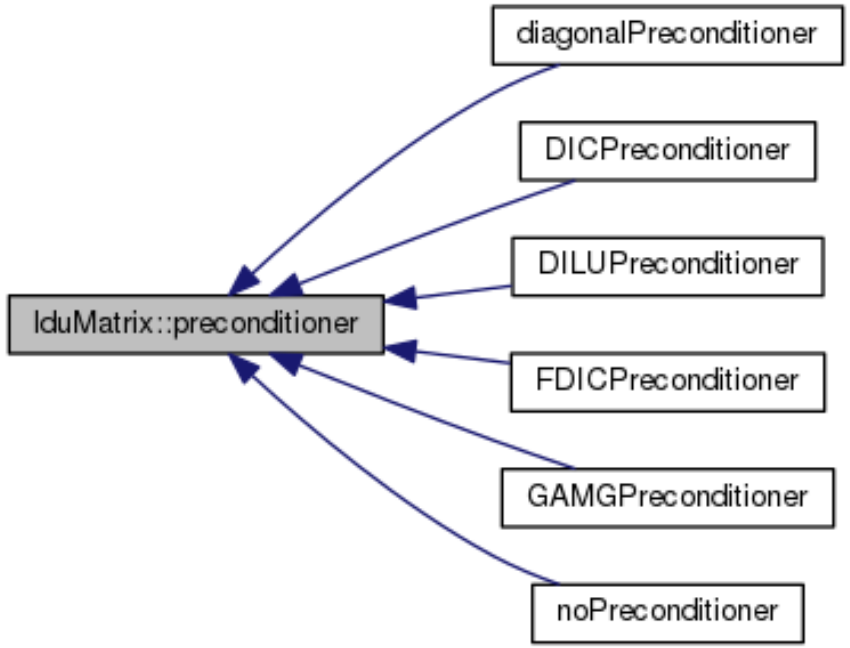
Doxygen structure of the class `lduMatrix::solver` in OpenFOAM.

The class `solver` (which is a folder also) includes:

- `PCG/PBiCGStab`: Stabilised preconditioned (bi-)conjugate gradient, for both symmetric and asymmetric matrices.
- `PCG/PBiCG`: preconditioned (bi-)conjugate gradient, with PCG for symmetric matrices, PBiCG for asymmetric matrices.
- `smoothSolver`: solver that uses a smoother.
- `GAMG`: generalised geometric-algebraic multi-grid.
- `diagonal`: diagonal solver for explicit systems.

The solvers distinguish between symmetric matrices and asymmetric matrices. The symmetry of the matrix depends on the terms of the equation being solved, e.g. time derivatives and Laplacian terms form coefficients of a symmetric matrix, whereas an advective derivative introduces asymmetry.

lduMatrix::preconditioner



The class `preconditioner` contains various implementations of the diagonal ILU denoted by

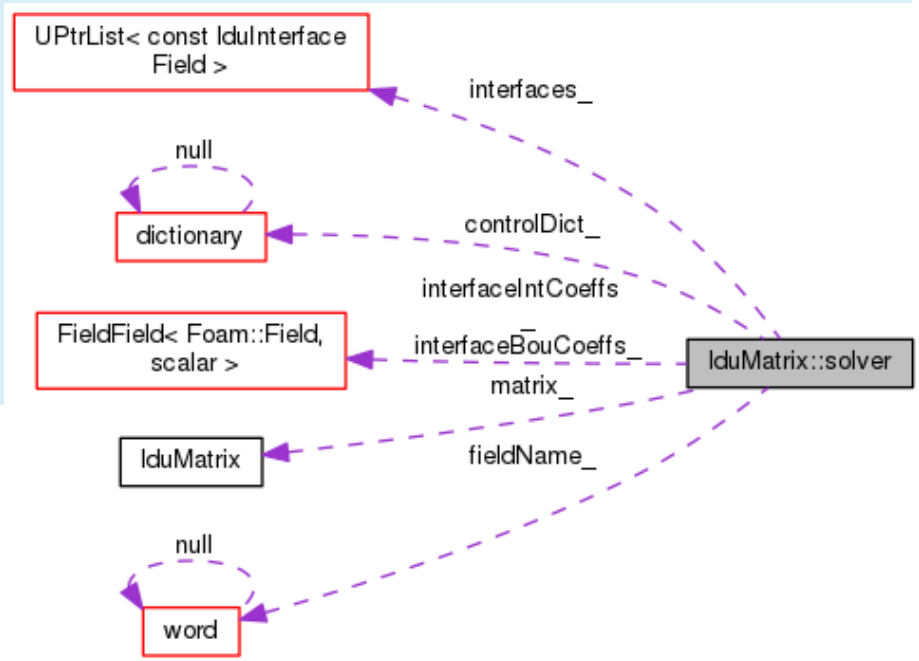
- `diagonalPreconditioner`: diagonal preconditioner;
- `DICPreconditioner, DILUPreconditioner`: diagonal Incomplete Cholesky preconditioner for symmetric and asymmetric matrices;
- `FDICPreconditioner`: faster version of the DICPreconditioners diagonalbased incomplete Cholesky preconditioner for symmetric matrices. Reciprocal of the preconditioned diagonal and the upper coefficients divided by the diagonal are calculated and stored.

$FOAM_CASE/system/fvSolution

How to set the linear solver in your simulation? The equation solvers, tolerances and algorithms are controlled from the `fvSolution` dictionary in the `system` directory of the simulation folder:

```
solvers
{
  p
  {
        solver           PCG;
        preconditioner   DIC;
        tolerance        1e-07;
        relTol           0.01;
  }

    ...
}
```



The abstract class `lduMatrix::solver` uses the class `dictionary` to access the dictionary `solvers` in the file `fvSolution`.

$FOAM_CASE/system/fvSolution

In `system/fvSolution::solvers`, for each solver, you will find the following keywords:

```
solvers
{
  p
  {
        solver           PCG;
        preconditioner   DIC;
        tolerance        1e-07;
        relTol           0.01;
  }

    ...
}
```

For each variable (p, U, etc) the user is asked to define:

- `solver` type;
- `preconditioner` or `smoother` type;
- `tolerance`: maximum allowable value of the absolute residual for the linear solver to stop iterating;
- `relTol`: ratio between the initial residual and the actual residual for the linear solver to stop iterating.

OpenFOAM: Output Monitor

```
Time = 0.01

Courant Number mean:  0.0976825 max:  0.585607
smoothSolver:  Solving for Ux, Initial residual = 0.160686, Final residual = 6.83031e-06, No Iterations 19
smoothSolver:  Solving for Uy, Initial residual = 0.260828, Final residual = 9.65939e-06, No Iterations 18
DICPCG: Solving for p, Initial residual = 0.428925, Final residual = 0.0103739, No Iterations 22
time step continuity errors :  sum local = 0.000110788, global = 3.77194e-19, cumulative = -6.72498e-20
DICPCG: Solving for p, Initial residual = 0.30209, Final residual = 5.26569e-07, No Iterations 33
time step continuity errors :  sum local = 6.61987e-09, global = -2.74872e-19, cumulative = -3.42122e-19
ExecutionTime = 0.01 s ClockTime = 0 s
```

Solution algorithm in pisoFOAM

Thank you for your attention!

contact: federico.piscaglia@polimi.it

Appendix A

Preconditioning of the coefficient matrix

Gradient Methods

Gradient Methods include:

- `Steepest Descent`
- `Conjugate Gradient` → very commonly used in OpenFOAM

Gradient methods were initially developed for cases where the coefficient matrix A is symmetric positive definite (SPD) to reformulate the problem as a minimization problem for the quadratic vector function $\boldsymbol{Q}(\boldsymbol{\phi})$:

$$\boldsymbol{Q}(\boldsymbol{\phi}) = \frac{1}{2}\boldsymbol{\phi}^{\boldsymbol{T}}\boldsymbol{A}\boldsymbol{\phi} - \boldsymbol{b}^{\boldsymbol{T}}\boldsymbol{\phi} + \boldsymbol{c}$$

being $\boldsymbol{c}$ a vector of scalars.

How do gradient methods work?

The gradient of the quadratic vector function $\boldsymbol{Q}(\boldsymbol{\phi})$ is calculated:

$$\boldsymbol{Q'}(\boldsymbol{\phi}) = \frac{1}{2}\boldsymbol{A^T}\boldsymbol{\phi} + \frac{1}{2}\boldsymbol{A}\boldsymbol{\phi} - \boldsymbol{b}$$

Only if A is symmetric ($\boldsymbol{A} = \boldsymbol{A}^T$), it follows:

$$\boldsymbol{Q'}(\boldsymbol{\phi}) = \boldsymbol{A}\boldsymbol{\phi} - \boldsymbol{b}$$

and

$$\boldsymbol{Q'}(\boldsymbol{\phi}) = 0 \quad \Longrightarrow \quad \boldsymbol{A}\boldsymbol{\phi} = \boldsymbol{b}$$

Therefore minimizing $\boldsymbol{Q'}(\boldsymbol{\phi})$ is equivalent to solving the algebraic system and the solution of the minimization problem yields the solution of the system of linear equations.

How do gradient methods work?

Now for the function $\boldsymbol{Q'}(\boldsymbol{\phi})$ to have a global minimum it is necessary for the coefficient matrix A to be positive definite

$$\boldsymbol{\phi}^T \boldsymbol{A}\boldsymbol{\phi} > 0, \forall \boldsymbol{\phi} \neq 0$$

This requirement can be established by considering the relationship between the exact solution $\boldsymbol{\phi}$ and its current estimate $\boldsymbol{\phi}^n$. Being:

$$\boldsymbol{e} = \boldsymbol{\phi} - \boldsymbol{\phi}^{(n)}$$

it follows:

$$\begin{aligned}
\boldsymbol{Q}(\boldsymbol{\phi}+\boldsymbol{e}) &= \frac{1}{2}(\boldsymbol{\phi}+\boldsymbol{e})^T \boldsymbol{A}\,(\boldsymbol{\phi}+\boldsymbol{e}) - \boldsymbol{b}^T(\boldsymbol{\phi}+\boldsymbol{e}) + \boldsymbol{c} \\
&= \frac{1}{2}(\boldsymbol{\phi})^T \boldsymbol{A}\,(\boldsymbol{\phi}) + \frac{1}{2}(\boldsymbol{e})^T \boldsymbol{A}\,(\boldsymbol{\phi}) + \frac{1}{2}(\boldsymbol{\phi})^T \boldsymbol{A}\,(\boldsymbol{e}) + \frac{1}{2}(\boldsymbol{e})^T \boldsymbol{A}\,(\boldsymbol{e}) - \boldsymbol{b}^T(\boldsymbol{\phi}) - \boldsymbol{b}^T(\boldsymbol{e}) + \boldsymbol{c} \\
&= \underbrace{\frac{1}{2}(\boldsymbol{\phi})^T \boldsymbol{A}\,(\boldsymbol{\phi}) - \boldsymbol{b}^T(\boldsymbol{\phi}) + \boldsymbol{c}}_{\boldsymbol{Q}(\boldsymbol{\phi})} + \frac{1}{2}\left(\underbrace{\boldsymbol{e^T A \phi}}_{\boldsymbol{e^T b = b^T w}} + \underbrace{\phi^T A\, e}_{\boldsymbol{b^T e}} \right) - \boldsymbol{b}^T\boldsymbol{e} + \frac{1}{2}\boldsymbol{e}^T\boldsymbol{A}\,\boldsymbol{e} \\
&= \boldsymbol{Q}(\boldsymbol{\phi}) + \frac{1}{2}\boldsymbol{e}^T\boldsymbol{A}\,\boldsymbol{e}
\end{aligned}$$

How do gradient methods work?

Being:

$$\boldsymbol{Q}(\boldsymbol{\phi}+\boldsymbol{e}) = \boldsymbol{Q}(\boldsymbol{\phi}) + \frac{1}{2}\boldsymbol{e}^T \boldsymbol{A}\, \boldsymbol{e}$$

it follows that:

1) if A is positive definite, the second term will be always positive except when $\boldsymbol{e} = 0$, in which case the required solution would have been obtained;
2) when $\boldsymbol{A}$ is positive definite, all its eigenvalues are positive and the function $\boldsymbol{Q}(\boldsymbol{\phi})$ has a unique minimum.

Thus with a **symmetric** and **positive definite** matrix a converging series of $\boldsymbol{\phi}^{(n)}$ can be derived such that

$$\boldsymbol{\phi}^{(n+1)} = \boldsymbol{\phi}^{(n)} + \alpha^{(n)} \left(\boldsymbol{\delta}\, \boldsymbol{\phi}^{(n)} \right)$$

where $\alpha^{(n)}$ is some relaxation factor, and $\boldsymbol{\delta}\, \boldsymbol{\phi}^{(n)}$ is related to the correction needed to minimize the function $\boldsymbol{\phi}^{(n)}$ at each iteration. **This can be accomplished in a variety of ways leading to different methods.**

How do gradient methods work?

The Conjugate Gradient Method tries to minimize the gradient of the quadratic vector function $\boldsymbol{Q}(\boldsymbol{\phi})$:

$$\boldsymbol{Q'}(\boldsymbol{\phi}) = \frac{1}{2}\boldsymbol{A^T}\boldsymbol{\phi} + \frac{1}{2}\boldsymbol{A}\boldsymbol{\phi} - \boldsymbol{b}$$

where $\boldsymbol{Q}(\boldsymbol{\phi})$ is a **paraboloid**. The solution is then iteratively found starting from an initial position $\boldsymbol{\phi}^{(0)}$ and moving down the paraboloid until the minimum is reached. For a quick convergence, the sequence of steps $\boldsymbol{\phi}^{(0)}$, $\boldsymbol{\phi}^{(1)}$, $\boldsymbol{\phi}^{(2)}$, ... should be selected such that the fastest rate of descent occurs in the the direction $-\boldsymbol{Q'}(\boldsymbol{\phi})$, being:

$$-\boldsymbol{Q'}(\boldsymbol{\phi}) = -\boldsymbol{A}\boldsymbol{\phi} + \boldsymbol{b}$$

The exact solution being $\boldsymbol{\phi}$, the error and residual at any step n, denoted respectively by $\boldsymbol{e}^{(n)}$ and $\boldsymbol{r}^{(n)}$, are computed as:

$$\begin{cases} \boldsymbol{e}^{(n)} = \boldsymbol{\phi}^{(n)} - \boldsymbol{\phi} \\ \boldsymbol{r}^{(n)} = -\boldsymbol{A}\boldsymbol{\phi} + \boldsymbol{b} = -\boldsymbol{Q'}(\boldsymbol{\phi}) \end{cases}$$

hence:

$$\boxed{\boldsymbol{r}^{(n)} = -\boldsymbol{A}\boldsymbol{e}^{(n)}}$$

The Conjugate Gradient Method

In the Conjugate Gradient Method, a set of search directions $\boldsymbol{d}^{(0)}$,$\boldsymbol{d}^{(1)}$,$\boldsymbol{d}^{(2)}$,...,$\boldsymbol{d}^{(N-1)}$ is selected, that must satisfy the **A-orthogonal condition**:

$$\left(\boldsymbol{d}^{(n)}\right)^T \boldsymbol{A}\boldsymbol{d}^{(m)} = 0$$

If in each search direction the right step size is taken, the solution will be found after N steps. Step $n+1$ is chosen such that

$$\boldsymbol{\phi}^{(n+1)} = \boldsymbol{\phi}^{(n)} + \boldsymbol{\alpha}^{(n)}\boldsymbol{d}^{(n)}$$

Subtracting $\boldsymbol{\phi}$ from both sides of the above equation, an equation for the error is obtained as

$$\boldsymbol{e}^{(n+1)} = \boldsymbol{e}^{(n)} + \boldsymbol{\alpha}^{(n)}\boldsymbol{d}^{(n)}$$

so:

$$\begin{aligned}\boldsymbol{r}^{(n+1)} &= -\boldsymbol{A}\boldsymbol{e}^{(n+1)} \\ &= -\boldsymbol{A}\left(\boldsymbol{e}^{(n)} + \boldsymbol{\alpha}^{(n)}\boldsymbol{d}^{(n)}\right) \\ &= \boldsymbol{r}^{(n)} - \boldsymbol{\alpha}^{(n)}\boldsymbol{A}\,\boldsymbol{d}^{(n)}\end{aligned}$$

which shows how each new residual $\boldsymbol{r}^{(n+1)}$ is just a linear combination of the previous residual and $\boldsymbol{A}\,\boldsymbol{d}^{(n)}$.

The Conjugate Gradient Method

It is further required that $\boldsymbol{e}^{(n+1)}$ be A-orthogonal to $\boldsymbol{d}^{(n)}$. This new condition is equivalent to finding the minimum point along the search direction $\boldsymbol{d}^{(n)}$. Using this A-orthogonality condition between $\boldsymbol{e}^{(n+1)}$ and $\boldsymbol{d}^{(n)}$:

$$\left(\boldsymbol{d}^{(n)}\right)^T \boldsymbol{A}\boldsymbol{e}^{(n+1)} = 0 \quad \Longrightarrow \quad \left(\boldsymbol{d}^{(n)}\right)^T \boldsymbol{A}\left(\boldsymbol{e}^{(n)} + \boldsymbol{\alpha}^{(n)}\boldsymbol{d}^{(n)}\right) = 0$$

hence:

$$\boldsymbol{\alpha}^{(n)} = \frac{\left(\boldsymbol{d}^{(n)}\right)^T \boldsymbol{r}^{(n+1)}}{\left(\boldsymbol{d}^{(n)}\right)^T \boldsymbol{A}\,\boldsymbol{d}^{(n)}}$$

To derive the search direction, it is assumed to be governed by an equation of the form

$$\boldsymbol{d}^{(n+1)} = \boldsymbol{r}^{(n+1)} + \boldsymbol{\beta}^{(n)}\boldsymbol{d}^{(n)}$$

The A-orthogonality requirement of the d vectors implies that

$$\left(\boldsymbol{d}^{(n)}\right)^T \boldsymbol{A}\boldsymbol{d}^{(n)} = 0$$

so:

$$\boldsymbol{\beta}^{(n)} = -\frac{\left(\boldsymbol{r}^{(n+1)}\right)^T \boldsymbol{A}\,\boldsymbol{d}^{(n)}}{\left(\boldsymbol{d}^{(n)}\right)^T \boldsymbol{A}\,\boldsymbol{d}^{(n)}}$$

The Conjugate Gradient Method

Being:

$$\boldsymbol{r}^{(n+1)} = \boldsymbol{r}^{(n)} - \boldsymbol{\alpha}^{(n)} \boldsymbol{A}\, \boldsymbol{d}^{(n)}$$

it follows:

$$\boldsymbol{A}\, \boldsymbol{d}^{(n)} = \frac{1}{\boldsymbol{\alpha}^{(n)}} \left(\boldsymbol{r}^{(n+1)} - \boldsymbol{r}^{(n)} \right)$$

and:

$$\boldsymbol{\beta}^{(n)} = \frac{\left(\boldsymbol{r}^{(n+1)} \right)^T \left(\boldsymbol{r}^{(n+1)} - \boldsymbol{r}^{(n)} \right)}{\left(\boldsymbol{d}^{(n)} \right)^T \boldsymbol{r}^{(n)}}$$

$$= \frac{\left(\boldsymbol{r}^{(n+1)} \right)^T \boldsymbol{r}^{(n+1)} - \underbrace{\left(\boldsymbol{r}^{(n+1)} \right)^T \boldsymbol{r}^{(n)}}_{=0}}{\left(\boldsymbol{d}^{(n)} \right)^T \boldsymbol{r}^{(n)}} = \frac{\left(\boldsymbol{r}^{(n+1)} \right)^T \boldsymbol{r}^{(n+1)}}{\left(\boldsymbol{d}^{(n)} \right)^T \boldsymbol{r}^{(n)}}$$

The Conjugate Gradient Method

The denominator of the above equation can be expressed as:

$$\begin{aligned}\left(\boldsymbol{d}^{(n)}\right)^T \boldsymbol{r}^{(n)} &= \left(\boldsymbol{r}^{(n)} + \beta^{(n-1)}\boldsymbol{d}^{(n-1)}\right)^T \boldsymbol{r}^{(n)} \\ &= \left(\boldsymbol{r}^{(n)}\right)^T \boldsymbol{r}^{(n)} + \beta^{(n-1)} \underbrace{\left(\boldsymbol{d}^{(n-1)}\right)^T \boldsymbol{r}^{(n)}}_{=0} \\ &= \left(\boldsymbol{r}^{(n)}\right)^T \boldsymbol{r}^{(n)}\end{aligned}$$

and finally:

$$\boldsymbol{\beta}^{(n)} = \frac{\left(\boldsymbol{r}^{(n+1)}\right)^T \boldsymbol{r}^{(n+1)}}{\left(\boldsymbol{r}^{(n)}\right)^T \boldsymbol{r}^{(n)}}$$

The Conjugate Gradient Method

The Conjugate Gradient algorithm performs the following steps:

$\boldsymbol{d}^{(0)} = \boldsymbol{r}^{(0)} = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{\phi}^{(0)}$ (choose residual as starting direction)

iterate starting at until convergence

$\boldsymbol{\alpha}^{(n)} = \dfrac{\left(\boldsymbol{d}^{(n)}\right)^T \boldsymbol{r}^{(n)}}{\left(\boldsymbol{d}^{(n)}\right)^T \boldsymbol{A}\,\boldsymbol{d}^{(n)}}$ (Choose factor in direction)

$\boldsymbol{\phi}^{(n+1)} = \boldsymbol{\phi}^{(n)} + \alpha^{(n)}\boldsymbol{d}^{(n)}$ (Obtain new $\boldsymbol{\phi}$)

$\boldsymbol{r}^{(n+1)} = \boldsymbol{r}^{(n)} - \boldsymbol{\alpha}^{(n)}\boldsymbol{A}\,\boldsymbol{d}^{(n)}$ (calculate new residual)

$\boldsymbol{\beta}^{(n)} = \dfrac{\left(\boldsymbol{r}^{(n+1)}\right)^T \boldsymbol{r}^{(n+1)}}{\left(\boldsymbol{r}^{(n)}\right)^T \boldsymbol{r}^{(n)}}$ (Calculate coefficient to conjugate residual)

$\boldsymbol{d}^{(n+1)} = \boldsymbol{r}^{(n+1)} + \boldsymbol{\beta}^{(n)}\boldsymbol{d}^{(n)}$ (obtain new conjugated search direction)

Preconditioned Conjugate Gradient

The convergence rate of the CG method may be increased by preconditioning. This can be done by multiplying the original system of equations by the inverse of the preconditioned matrix $\boldsymbol{P^{-1}}$, where $\boldsymbol{P}$ is a symmetric positive-definite matrix, to yeld:

$$\boldsymbol{P^{-1}A\phi = P^{-1}b}$$

The problem is that $\boldsymbol{P^{-1}A}$ is not necessarily symmetric even if $\boldsymbol{P}$ and $\boldsymbol{A}$ are symmetric. To circumvent this problem the Cholesky decomposition is used to write P in the form

$$\boldsymbol{P = LL^T}$$

To guarantee symmetry, the system of equations is written as

$$\boldsymbol{L^{-1}AL^{-T}L^T\phi = L^{-1}b}$$

where $\boldsymbol{L^{-1}AL^{-T}}$ is symmetric and positive-definite. The CG method can be used to solve for $\boldsymbol{L^T\phi}$, from which $\boldsymbol{\phi}$ is found. However, by variable substitutions, $\boldsymbol{L}$ can be eliminated from the equations without disturbing symmetry or affecting the validity of the method. Performing this step and adopting the terminology used with the CG method, the various steps in the preconditioned CG method are obtained.

Preconditioned Conjugate Gradient

$\boldsymbol{r}^{(0)} = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{\phi}^{(0)}$ and $\boldsymbol{d}^{(0)} = \boldsymbol{P^{-1}}\boldsymbol{r}^{(0)}$ (choose starting direction)

iterate starting at (n) until convergence

$\boldsymbol{\alpha}^{(n)} = \frac{\left(\boldsymbol{r}^{(n)}\right)^T \boldsymbol{P^{-1}}\boldsymbol{r}^{(n)}}{\left(\boldsymbol{d}^{(n)}\right)^T \boldsymbol{A}\,\boldsymbol{d}^{(n)}}$ (Choose factor in $\boldsymbol{d}$ direction)

$\boldsymbol{\phi}^{(n+1)} = \boldsymbol{\phi}^{(n)} + \alpha^{(n)}\boldsymbol{d}^{(n)}$ (Obtain new $\boldsymbol{\phi}$)

$\boldsymbol{r}^{(n+1)} = \boldsymbol{r}^{(n)} - \boldsymbol{\alpha}^{(n)}\boldsymbol{A}\,\boldsymbol{d}^{(n)}$ (calculate new residual)

$\boldsymbol{\beta}^{(n)} = \frac{\left(\boldsymbol{r}^{(n+1)}\right)^T \boldsymbol{P^{-1}}\boldsymbol{r}^{(n+1)}}{\left(\boldsymbol{r}^{(n)}\right)^T \boldsymbol{P^{-1}}\boldsymbol{r}^{(n)}}$ (Calculate coefficient to conjugate residual)

$\boldsymbol{d}^{(n+1)} = \boldsymbol{P^{-1}}\boldsymbol{r}^{(n+1)} + \boldsymbol{\beta}^{(n)}\boldsymbol{d}^{(n)}$ (obtain new conjugated search direction)

Preconditioned Conjugate Gradient

There are a range of options for preconditioning of matrices in the conjugate gradient solvers, represented by the preconditioner keyword in the solver dictionary, listed below. Note that the DIC/DILU preconditioners are exclusively specified in the tutorials in OpenFOAM.

- `DIC/DILU`: diagonal incomplete-Cholesky (symmetric) and incomplete-LU (asymmetric)
- `FDIC`: faster diagonal incomplete-Cholesky (DIC with caching, symmetric)
- `diagonal`: diagonal preconditioning.
- `GAMG`: geometric-algebraic multi-grid.
- `none`: no preconditioning.

Multigrid Method (GAMG)

Geometric-algebraic multi-grid solvers

```
    p
    {
        solver          GAMG;
        tolerance       1e-7;
        relTol          0.01;
        smoother        GaussSeidel;
    }
```

The generalised method of geometric-algebraic multi-grid (GAMG) uses the principle of:

- generating a quick solution on a mesh with a small number of cells; mapping this solution onto a finer mesh;
- using it as an initial guess to obtain an accurate solution on the fine mesh. GAMG is faster than standard methods when the increase in speed by solving first on coarser meshes outweighs the additional costs of mesh refinement and mapping of field data.

In practice, GAMG starts with the mesh specified by the user and coarsens/refines the mesh in stages. The user is only required to specify an approximate mesh size at the most coarse level in terms of the number of cells