# How OpenFOAM discretizes equations:

A walk through momentum equation in `icoFoam`

Open-Source CFD Course

A course at Chalmers University of Technology

Taught by Håkan Nilsson

Based on OpenFOAM v2112

Presenter:

Saeed Salehi

Division of Fluid Dynamics

Department of Mechanics and Maritime Sciences

Chalmers University of Technology

September 2022

## Constructing momentum equation

## Constructing object UEqn

- In this presentation we will have a look at how equations are discretized and assembled in OpenFOAM.
- After constructing the full linear system, it is solved. Here, we will not cover the solution procedure of the linear system.

- As an example we will have a look at the momentum equation in the `icoFoam` solver. The momentum equation for laminar Newtonian flows reads:

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

- Each term in the momentum equation is represented by an expression in the code.

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

if (piso.momentumPredictor())
{
    solve(UEqn == -fvc::grad(p));
}
```

- `fvVectorMatrix` is a *typedef* defined in `fvMatricesFwd.H` as:

```
typedef fvMatrix<scalar> fvScalarMatrix;
typedef fvMatrix<vector> fvVectorMatrix;
typedef fvMatrix<sphericalTensor> fvSphericalTensorMatrix;
typedef fvMatrix<symmTensor> fvSymmTensorMatrix;
typedef fvMatrix<tensor> fvTensorMatrix;
```

- `UEqn` is an object of the class template `fvMatrix<Type>` instantiated with type `vector`.

## fvMatrix class

- fvMatrix is a class for finite volume discretization and solution of scalar equations in OpenFOAM. In the case of vectorial equations, the solution procedure is performed in a loop for each component separately. It contains a special matrix type that uses face addressing for the matrix assembly.
- Previously, we showed that the discretized momentum equation will take the form of

$$a_P^{\mathbf{u}}\mathbf{u}_P + \sum_N a_N^{\mathbf{u}}\mathbf{u}_N = \mathbf{r} - \nabla p \qquad \text{or} \qquad \mathbf{Ax} = \mathbf{B}$$

- fvMatrix important member data:
  - ✓ An lduMatrix object (superclass, $\mathbf{A}$)
  - ✓ psi_: A reference to the volume field being discretized ($\mathbf{x}$)
  - ✓ dimensions_: The dimension set
  - ✓ source_: The source term (right-hand side of the linear system, $\mathbf{B}$)
  - ✓ internalCoeffs_: Contribution of the boundary conditions to the diagonal members of the coefficient matrix
  - ✓ boundaryCoeffs_: Contribution of the boundary conditions to the source term
  - ✓ faceFluxCorrectionPtr_: Non-orthogonal correction of the face flux field.

fvMesh Contructor

```
template<class Type>
Foam::fvMatrix<Type>::fvMatrix
(
    const GeometricField<Type,fvPatchField,volMesh>& psi,
    const dimensionSet& ds
)
:
    lduMatrix(psi.mesh()),
    psi_(psi),
    useImplicit_(false),
    lduAssemblyName_(),
    nMatrix_(0),
    dimensions_(ds),
    source_(psi.size(), Zero),
    internalCoeffs_(psi.mesh().boundary().size()),
    boundaryCoeffs_(psi.mesh().boundary().size()),
    faceFluxCorrectionPtr_(nullptr)
{
```

## `fvMatrix` compilation

- How the `fvMatrix` class is compiled and used in icoFoam? Can you find it?

## `fvMatrix` compilation

- How the `fvMatrix` class is compiled and used in icoFoam? Can you find it?
- `fvMatrix` is a class template. Uninstantiated templates can not be compiled directly and stored in a library. Can you explain why?

## fvMatrix compilation

- How the `fvMatrix` class is compiled and used in icoFoam? Can you find it?
- `fvMatrix` is a class template. Uninstantiated templates can not be compiled directly and stored in a library. Can you explain why?
- A common practice in C++ is to keep the declaration and implementation of a class template in the header file so that the compiler will have access to the full implementation wherever the header file is included.

```
#ifdef NoRepository
    #include "fvMatrix.C"
#endif
```

- Another common approach in OpenFOAM is to use macros to specialize a class template for specific type(s). Then, the specialized class can be compiled and stored in a library. An example is the compilation of the turbulence models in OpenFOAM (after OpenFOAM 3.0).
- `wmake` command defines the `NoRepository` variable by default.
- `fvMatrix.H` is included inside icoFoam. Can you track it down?
- `gcc -E` can create the output of the icoFoam.C after preprocessing step. Run the first step of `wmake` using gcc instead of g++ with `-E` flag.

## fvMatrix compilation

- How the fvMatrix class is compiled and used in icoFoam? Can you find it?
- fvMatrix is a class template. Uninstantiated templates can not be compiled directly and stored in a library. Can you explain why?
- A common practice in C++ is to keep the declaration and implementation of a class template in the header file so that the compiler will have access to the full implementation wherever the header file is included.

```
#ifdef NoRepository
    #include "fvMatrix.C"
#endif
```

- Another common approach in OpenFOAM is to use macros to specialize a class template for specific type(s). Then, the specialized class can be compiled and stored in a library. An example is the compilation of the turbulence models in OpenFOAM (after OpenFOAM 3.0).
- wmake command defines the NoRepository variable by default.
- fvMatrix.H is included inside icoFoam. Can you track it down?
- gcc -E can create the output of the icoFoam.C after preprocessing step. Run the first step of wmake using gcc instead of g++ with -E flag.
- icoFoam.C that compiler sees and works with has actually 266,191 lines of codes!

## Components of UEqn

- Each term inside the `UEqn` brackets constructs a separate instance of `fvMatrix<vector>`. That means a full linear system will be created for each term and stored in member data of `fvMatrix<vector>` class.

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

- In fact, `ddt()`, `div()`, and `lapacian()` are global functions defined in the `fvm` namespace that return a type `tmp<fvMatrix<Type>>`. We will discuss each term later in detail.
- "+" and "-" are overloaded global operators.
- Constructors of `fvMatrix` indicate that inside bracket should also return an instance of `fvMatrix`. Here the copy constructor is called.
- Constructing `UEqn` can be better explained by running `icoFoam` with activating the debug switch for `fvVectorMatrix`, i.e.,

```
icoFoam -debug-switch fvVectorMatrix=1
```

## icoFoam with debug switch for fvVectorMatrix

```
Courant Number mean: 0 max: 0
    ...
    in file lnInclude/fvMatrix.C at line 281
    Constructing fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 281
    Constructing fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 281
    Constructing fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 454
    Destroying fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 454
    Destroying fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 371
    Copying fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 454
    Destroying fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 330
    Copying fvMatrix<Type> for field U
fvMatrix<Type>::solveSegregatedOrCoupled(const dictionary& solverControls)
       : solving fvMatrix<Type>
fvMatrix<Type>::solveSegregated(const dictionary& solverControls) : solving
        fvMatrix<Type>
smoothSolver:  Solving for Ux, Initial residual = 1, Final residual =
       1.53142e-06, No Iterations 3
smoothSolver:  Solving for Uy, Initial residual = 0, Final residual = 0, No
        Iterations 0
    ...
    in file lnInclude/fvMatrix.C at line 454
    Destroying fvMatrix<Type> for field U
```

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

        if (piso.momentumPredictor())
        {
            solve(UEqn == -fvc::grad(p));
        }
```

## icoFoam with debug switch for fvVectorMatrix

```
Courant Number mean: 0 max: 0
    ...
    in file lnInclude/fvMatrix.C at line 281
    Constructing fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 281
    Constructing fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 281
    Constructing fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 454
    Destroying fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 454
    Destroying fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 371
    Copying fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 454
    Destroying fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 330
    Copying fvMatrix<Type> for field U
fvMatrix<Type>::solveSegregatedOrCoupled(const dictionary& solverControls)
       : solving fvMatrix<Type>
fvMatrix<Type>::solveSegregated(const dictionary& solverControls) : solving
        fvMatrix<Type>
smoothSolver:  Solving for Ux, Initial residual = 1, Final residual =
       1.53142e-06, No Iterations 3
smoothSolver:  Solving for Uy, Initial residual = 0, Final residual = 0, No
        Iterations 0
    ...
    in file lnInclude/fvMatrix.C at line 454
    Destroying fvMatrix<Type> for field U
```

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

if (piso.momentumPredictor())
{
    solve(UEqn == -fvc::grad(p));
}
```

- fvm::ddt(U)

## icoFoam with debug switch for fvVectorMatrix

```
Courant Number mean: 0 max: 0
    ...
    in file lnInclude/fvMatrix.C at line 281
    Constructing fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 281
    Constructing fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 281
    Constructing fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 454
    Destroying fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 454
    Destroying fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 371
    Copying fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 454
    Destroying fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 330
    Copying fvMatrix<Type> for field U
fvMatrix<Type>::solveSegregatedOrCoupled(const dictionary& solverControls)
        : solving fvMatrix<Type>
fvMatrix<Type>::solveSegregated(const dictionary& solverControls) : solving
         fvMatrix<Type>
smoothSolver:  Solving for Ux, Initial residual = 1, Final residual =
        1.53142e-06, No Iterations 3
smoothSolver:  Solving for Uy, Initial residual = 0, Final residual = 0, No
         Iterations 0
    ...
    in file lnInclude/fvMatrix.C at line 454
    Destroying fvMatrix<Type> for field U
```

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

if (piso.momentumPredictor())
{
    solve(UEqn == -fvc::grad(p));
}
```

- fvm::ddt(U)
- fvm::div(phi,U)

## icoFoam with debug switch for fvVectorMatrix

```
Courant Number mean: 0 max: 0
    ...
    in file lnInclude/fvMatrix.C at line 281
    Constructing fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 281
    Constructing fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 281
    Constructing fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 454
    Destroying fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 454
    Destroying fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 371
    Copying fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 454
    Destroying fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 330
    Copying fvMatrix<Type> for field U
fvMatrix<Type>::solveSegregatedOrCoupled(const dictionary& solverControls)
        : solving fvMatrix<Type>
fvMatrix<Type>::solveSegregated(const dictionary& solverControls) : solving
         fvMatrix<Type>
smoothSolver:  Solving for Ux, Initial residual = 1, Final residual =
        1.53142e-06, No Iterations 3
smoothSolver:  Solving for Uy, Initial residual = 0, Final residual = 0, No
         Iterations 0
    ...
    in file lnInclude/fvMatrix.C at line 454
    Destroying fvMatrix<Type> for field U
```

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

        if (piso.momentumPredictor())
        {
            solve(UEqn == -fvc::grad(p));
        }
```

- fvm::ddt(U)
- fvm::div(phi,U)
- fvm::laplacian(nu,U)

## icoFoam with debug switch for fvVectorMatrix

```
Courant Number mean: 0 max: 0
    ...
    in file lnInclude/fvMatrix.C at line 281
    Constructing fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 281
    Constructing fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 281
    Constructing fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 454
    Destroying fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 454
    Destroying fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 371
    Copying fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 454
    Destroying fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 330
    Copying fvMatrix<Type> for field U
fvMatrix<Type>::solveSegregatedOrCoupled(const dictionary& solverControls)
       : solving fvMatrix<Type>
fvMatrix<Type>::solveSegregated(const dictionary& solverControls) : solving
        fvMatrix<Type>
smoothSolver:  Solving for Ux, Initial residual = 1, Final residual =
       1.53142e-06, No Iterations 3
smoothSolver:  Solving for Uy, Initial residual = 0, Final residual = 0, No
        Iterations 0
    ...
    in file lnInclude/fvMatrix.C at line 454
    Destroying fvMatrix<Type> for field U
```

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

if (piso.momentumPredictor())
{
    solve(UEqn == -fvc::grad(p));
}
```

- fvm::ddt(U)
- fvm::div(phi,U)
- fvm::laplacian(nu,U)
- + operator

## icoFoam with debug switch for fvVectorMatrix

```
Courant Number mean: 0 max: 0
    ...
    in file lnInclude/fvMatrix.C at line 281
    Constructing fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 281
    Constructing fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 281
    Constructing fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 454
    Destroying fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 454
    Destroying fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 371
    Copying fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 454
    Destroying fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 330
    Copying fvMatrix<Type> for field U
fvMatrix<Type>::solveSegregatedOrCoupled(const dictionary& solverControls)
       : solving fvMatrix<Type>
fvMatrix<Type>::solveSegregated(const dictionary& solverControls) : solving
        fvMatrix<Type>
smoothSolver:  Solving for Ux, Initial residual = 1, Final residual =
       1.53142e-06, No Iterations 3
smoothSolver:  Solving for Uy, Initial residual = 0, Final residual = 0, No
        Iterations 0
    ...
    in file lnInclude/fvMatrix.C at line 454
    Destroying fvMatrix<Type> for field U
```

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

        if (piso.momentumPredictor())
        {
            solve(UEqn == -fvc::grad(p));
        }
```

- `fvm::ddt(U)`
- `fvm::div(phi,U)`
- `fvm::laplacian(nu,U)`
- + operator
- − operator

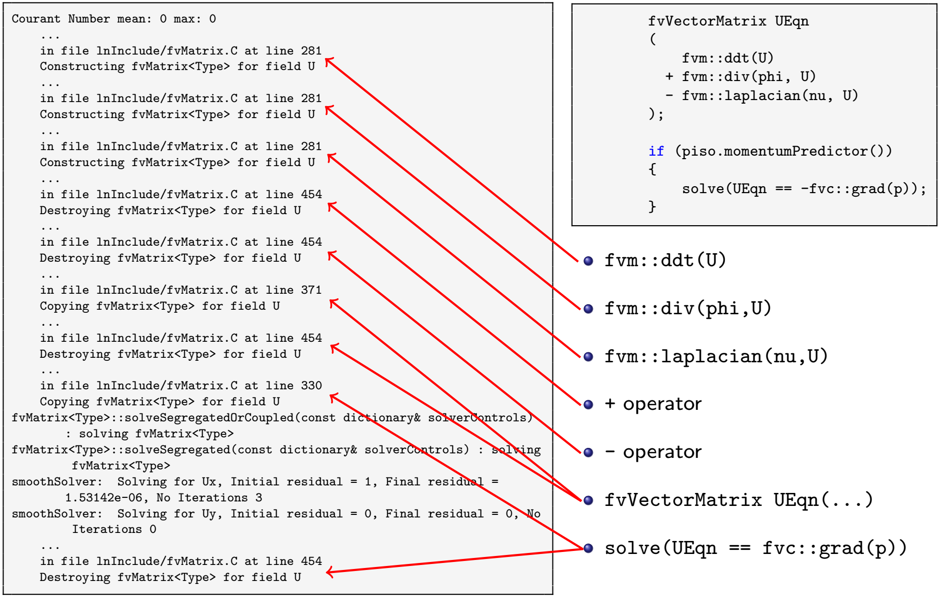
## icoFoam with debug switch for fvVectorMatrix

```
Courant Number mean: 0 max: 0
    ...
    in file lnInclude/fvMatrix.C at line 281
    Constructing fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 281
    Constructing fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 281
    Constructing fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 454
    Destroying fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 454
    Destroying fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 371
    Copying fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 454
    Destroying fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 330
    Copying fvMatrix<Type> for field U
fvMatrix<Type>::solveSegregatedOrCoupled(const dictionary& solverControls)
       : solving fvMatrix<Type>
fvMatrix<Type>::solveSegregated(const dictionary& solverControls) : solving
        fvMatrix<Type>
smoothSolver:  Solving for Ux, Initial residual = 1, Final residual =
       1.53142e-06, No Iterations 3
smoothSolver:  Solving for Uy, Initial residual = 0, Final residual = 0, No
        Iterations 0
    ...
    in file lnInclude/fvMatrix.C at line 454
    Destroying fvMatrix<Type> for field U
```

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

if (piso.momentumPredictor())
{
    solve(UEqn == -fvc::grad(p));
}
```

- fvm::ddt(U)
- fvm::div(phi,U)
- fvm::laplacian(nu,U)
- + operator
- − operator
- fvVectorMatrix UEqn(...)

# icoFoam with debug switch for fvVectorMatrix



```
Courant Number mean: 0 max: 0
    ...
    in file lnInclude/fvMatrix.C at line 281
    Constructing fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 281
    Constructing fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 281
    Constructing fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 454
    Destroying fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 454
    Destroying fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 371
    Copying fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 454
    Destroying fvMatrix<Type> for field U
    ...
    in file lnInclude/fvMatrix.C at line 330
    Copying fvMatrix<Type> for field U
fvMatrix<Type>::solveSegregatedOrCoupled(const dictionary& solverControls)
       : solving fvMatrix<Type>
fvMatrix<Type>::solveSegregated(const dictionary& solverControls) : solving
        fvMatrix<Type>
smoothSolver:  Solving for Ux, Initial residual = 1, Final residual =
       1.53142e-06, No Iterations 3
smoothSolver:  Solving for Uy, Initial residual = 0, Final residual = 0, No
        Iterations 0
    ...
    in file lnInclude/fvMatrix.C at line 454
    Destroying fvMatrix<Type> for field U
```

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

if (piso.momentumPredictor())
{
    solve(UEqn == -fvc::grad(p));
}
```

- `fvm::ddt(U)`
- `fvm::div(phi,U)`
- `fvm::laplacian(nu,U)`
- + operator
- – operator
- `fvVectorMatrix UEqn(...)`
- `solve(UEqn == fvc::grad(p))`

## Non-default copy-constructor and destructor

```
template<class Type>
Foam::fvMatrix<Type>::fvMatrix(const fvMatrix<Type>& fvm)
:
    lduMatrix(fvm),
    psi_(fvm.psi_),
    useImplicit_(fvm.useImplicit_),
    lduAssemblyName_(fvm.lduAssemblyName_),
    nMatrix_(fvm.nMatrix_),
    dimensions_(fvm.dimensions_),
    source_(fvm.source_),
    internalCoeffs_(fvm.internalCoeffs_),
    boundaryCoeffs_(fvm.boundaryCoeffs_),
    faceFluxCorrectionPtr_(nullptr)
{
    DebugInFunction
        <<"Copying fvMatrix<Type> for field "<<psi_.name()<<endl;

    if (fvm.faceFluxCorrectionPtr_)
    {
        faceFluxCorrectionPtr_ =
            new GeometricField<Type, fvsPatchField, surfaceMesh>
            (
                *(fvm.faceFluxCorrectionPtr_)
            );
    }
}
```

```
template<class Type>
Foam::fvMatrix<Type>::~fvMatrix()
{
    DebugInFunction
        << "Destroying fvMatrix<Type> for field " << psi_.name() <<
         endl;

    deleteDemandDrivenData(faceFluxCorrectionPtr_);
    subMatrices_.clear();
}
```

- Non-default copy-constructor and destructor are defined. Why?

## Non-default copy-constructor and destructor

```
template<class Type>
Foam::fvMatrix<Type>::fvMatrix(const fvMatrix<Type>& fvm)
:
    lduMatrix(fvm),
    psi_(fvm.psi_),
    useImplicit_(fvm.useImplicit_),
    lduAssemblyName_(fvm.lduAssemblyName_),
    nMatrix_(fvm.nMatrix_),
    dimensions_(fvm.dimensions_),
    source_(fvm.source_),
    internalCoeffs_(fvm.internalCoeffs_),
    boundaryCoeffs_(fvm.boundaryCoeffs_),
    faceFluxCorrectionPtr_(nullptr)
{
    DebugInFunction
        <<"Copying fvMatrix<Type> for field "<<psi_.name()<<endl;

    if (fvm.faceFluxCorrectionPtr_)
    {
        faceFluxCorrectionPtr_ =
            new GeometricField<Type, fvsPatchField, surfaceMesh>
            (
                *(fvm.faceFluxCorrectionPtr_)
            );
    }
}
```

```
template<class Type>
Foam::fvMatrix<Type>::~fvMatrix()
{
    DebugInFunction
        << "Destroying fvMatrix<Type> for field " << psi_.name() <<
         endl;

    deleteDemandDrivenData(faceFluxCorrectionPtr_);
    subMatrices_.clear();
}
```

- Non-default copy-constructor and destructor are defined. Why?
- Mainly because of memory management which should be done manually in C++ (no automatic garbage collection).
- `faceFluxCorrectionPtr_` is a pointer that points to a dynamically allocated memory on the heap (`new` command). It is responsible for the non-orthogonal correction of the face flux field and will be discussed later in the discretization of the Laplacian term.
- Copy-constructor takes care of shallow copy problem, while destructor clears the garbage.
- When `faceFluxCorrectionPtr_` goes out of scope, the pointer will be removed but its corresponding allocated memory will *not* be freed. There will be an allocated memory on the heap that nothing points to it (garbage). Therefore, we need to remove it manually in the destructor.

## Non-default copy-constructor and destructor

```
template<class Type>
Foam::fvMatrix<Type>::fvMatrix(const fvMatrix<Type>& fvm)
:
    lduMatrix(fvm),
    psi_(fvm.psi_),
    useImplicit_(fvm.useImplicit_),
    lduAssemblyName_(fvm.lduAssemblyName_),
    nMatrix_(fvm.nMatrix_),
    dimensions_(fvm.dimensions_),
    source_(fvm.source_),
    internalCoeffs_(fvm.internalCoeffs_),
    boundaryCoeffs_(fvm.boundaryCoeffs_),
    faceFluxCorrectionPtr_(nullptr)
{
    DebugInFunction
        <<"Copying fvMatrix<Type> for field "<<psi_.name()<<endl;

    if (fvm.faceFluxCorrectionPtr_)
    {
        faceFluxCorrectionPtr_ =
            new GeometricField<Type, fvsPatchField, surfaceMesh>
            (
                *(fvm.faceFluxCorrectionPtr_)
            );
    }
}
```

```
template<class Type>
Foam::fvMatrix<Type>::~fvMatrix()
{
    DebugInFunction
        << "Destroying fvMatrix<Type> for field " << psi_.name() <<
         endl;

    deleteDemandDrivenData(faceFluxCorrectionPtr_);
    subMatrices_.clear();
}
```

- Such procedures are automated using *smart pointers*, e.g., `unique_ptr`, `shared_ptr`.

## Non-default copy-constructor and destructor

```
template<class Type>
Foam::fvMatrix<Type>::fvMatrix(const fvMatrix<Type>& fvm)
:
    lduMatrix(fvm),
    psi_(fvm.psi_),
    useImplicit_(fvm.useImplicit_),
    lduAssemblyName_(fvm.lduAssemblyName_),
    nMatrix_(fvm.nMatrix_),
    dimensions_(fvm.dimensions_),
    source_(fvm.source_),
    internalCoeffs_(fvm.internalCoeffs_),
    boundaryCoeffs_(fvm.boundaryCoeffs_),
    faceFluxCorrectionPtr_(nullptr)
{
    DebugInFunction
        <<"Copying fvMatrix<Type> for field "<<psi_.name()<<endl;

    if (fvm.faceFluxCorrectionPtr_)
    {
        faceFluxCorrectionPtr_ =
            new GeometricField<Type, fvsPatchField, surfaceMesh>
            (
                *(fvm.faceFluxCorrectionPtr_)
            );
    }
}
```

```
template<class Type>
Foam::fvMatrix<Type>::~fvMatrix()
{
    DebugInFunction
        << "Destroying fvMatrix<Type> for field " << psi_.name() <<
         endl;

    deleteDemandDrivenData(faceFluxCorrectionPtr_);
    subMatrices_.clear();
}
```

- Such procedures are automated using *smart pointers*, e.g., `unique_ptr`, `shared_ptr`.
- `tmp< T >` class in OpenFOAM is reimplementation of the `shared_ptr`. Hence, a memory that is dynamically allocated on the heap using `tmp< T >` will be freed automatically when the pointer goes out of scope.
- If we are not using smart pointers (like here), a non-default copy-constructor is also required to avoid a shallow copy problem (only copying the pointer and not the allocated memory).
- Basically, for a class that allocates a dynamic memory on the heap without the usage of the smart pointers, the copy-constructor, destructor, and assignment operators have to be redefined and the allocated memory should be taken care of.

## + operator

- What does the following summation mean?

```
        fvm::ddt(U)
      + fvm::div(phi, U)
```

## + operator

- What does the following summation mean?

```
    fvm::ddt(U)
  + fvm::div(phi, U)
```

- Two objects are summed up and thus the summation operator should be overloaded.
- First, the global `operator+`, overloaded in `fvMatrix.C`, is called.
- The `checkMethod` function checks if the two sides are compatible.
- It then calls the member `operator+=` that receives a type `fvMatrix<Type>&`.
- Note that `operator+` returns a value of the same type. However, the assignment operators (e.g., `operator+=`) modify member data of the current object and thus are either `void` or `return *this`.

```
template<class Type>
Foam::tmp<Foam::fvMatrix<Type>> Foam::operator+
(
    const tmp<fvMatrix<Type>>& tA,
    const tmp<fvMatrix<Type>>& tB
)
{
    checkMethod(tA(), tB(), "+");
    tmp<fvMatrix<Type>> tC(tA.ptr());
    tC.ref() += tB();
    tB.clear();
    return tC;
}
```

## + operator

- Finally we end up with the following member operator+=.
- All the member data are summed with their corresponding values.
- Each member data calls another assignment operator+=.
- Why are dimensions summed? (Hint check its implementation)
- Implementation of the "-" operator is similar.

```
template<class Type>
void Foam::fvMatrix<Type>::operator+=(const fvMatrix<Type>& fvmv)
{
    checkMethod(*this, fvmv, "+=");

    dimensions_ += fvmv.dimensions_;
    lduMatrix::operator+=(fvmv);
    source_ += fvmv.source_;
    internalCoeffs_ += fvmv.internalCoeffs_;
    boundaryCoeffs_ += fvmv.boundaryCoeffs_;

    useImplicit_ = fvmv.useImplicit_;
    lduAssemblyName_ = fvmv.lduAssemblyName_;
    nMatrix_ = fvmv.nMatrix_;

    if (faceFluxCorrectionPtr_ && fvmv.faceFluxCorrectionPtr_)
    {
        *faceFluxCorrectionPtr_ += *fvmv.faceFluxCorrectionPtr_;
    }
    else if (fvmv.faceFluxCorrectionPtr_)
    {
        faceFluxCorrectionPtr_ = new
        GeometricField<Type, fvsPatchField, surfaceMesh>
        (
            *fvmv.faceFluxCorrectionPtr_
        );
    }
}
```

## == operator

- Now that `UEqn` is fully constructed, let's see what is happening inside

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

- Some manipulations are performed on the `UEqn` object using the == operator and the result is passed to the global function `solve` through a temporary object. The manipulation is not stored inside `UEqn` because later we need the `UEqn` without the contribution of the pressure.

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

if (piso.momentumPredictor())
{
    solve(UEqn == -fvc::grad(p));
}
```

- The temporary object is destroyed after `retun` of the `solve` function.
- The == operator is globally overloaded (two input arguments) inside `fvMatrix.C`.
- There are 14 definitions for == operator. Which one is used here?

## == operator

- Now that UEqn is fully constructed, let's see what is happening inside

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

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

if (piso.momentumPredictor())
{
    solve(UEqn == -fvc::grad(p));
}
```

- Some manipulations are performed on the UEqn object using the == operator and the result is passed to the global function solve through a temporary object. The manipulation is not stored inside UEqn because later we need the UEqn without the contribution of the pressure.
- The temporary object is destroyed after retun of the solve function.
- The == operator is globally overloaded (two input arguments) inside fvMatrix.C.
- There are 14 definitions for == operator. Which one is used here?
- To find out which one is used, we need to know the type of the left and right-hand sides of the == operator.
- UEqn is a fvMatrix<vector>.
- What about -fvc::grad(p)?

## == operator

- grad is a global function defined in the fvc namespace as

```
template<class Type>
tmp
<
    GeometricField
    <
        typename outerProduct<vector,Type>::type, fvPatchField, volMesh
    >
>
grad
(
    const GeometricField<Type, fvPatchField, volMesh>& vf
)
{
    return fvc::grad(vf, "grad(" + vf.name() + ')');
}
```

- It is only important to note that the return type is a

```
tmp
<
    GeometricField
    <
        typename outerProduct<vector,Type>::type, fvPatchField, volMesh
    >
>
```

## == operator

- Therefore, `UEqn == -fvc::grad(p)` employs the following definition of ==

```
template<class Type>
Foam::tmp<Foam::fvMatrix<Type>> Foam::operator==
(
    const fvMatrix<Type>& A,
    const tmp<GeometricField<Type, fvPatchField, volMesh>>& tsu
)
{
    checkMethod(A, tsu(), "==");
    tmp<fvMatrix<Type>> tC(new fvMatrix<Type>(A));
    tC.ref().source() += tsu().mesh().V()*tsu().primitiveField();
    tsu.clear();
    return tC;
}
```

- A new object is created using the copy-constructor.
- The right-hand side of == is a `GeometricField` which is an explicit term and only contributes to the source term of the `fvMatrix`.
- The right-hand side times mesh volume is added to the source term of the `UEqn`.

## Solution

## Tracking the solve() function

- Now that the momentum equation is fully discretized, let's see what the solve function is doing. We will only track down this function right before the linear system solution.

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

- The solve() function that receives an fvMatrix is a global function that is defined inside the fvMatrix.C file.
- Considering the input argument of the function, the following solve() function is called.

```
template<class Type>
Foam::SolverPerformance<Type> Foam::solve(fvMatrix<Type>& fvm)
{
    return fvm.solve();
}
```

- Then, a member solve() function (without any arguments) from the fvMatrix class is called on the temporary object created for the full momentum equation (not the UEqn but the UEqn == -fvc::grad(p)). It is defined in fvMatrixSolve.C.

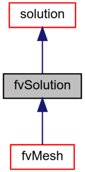
```
template<class Type>
Foam::SolverPerformance<Type> Foam::fvMatrix<Type>::solve()
{
    return this->solve(solverDict());
}
```

## Tracking the solve() function

- Thereby, another solve() function from the fvMatrix class is called but this time the function receives the solver dictionary as the input.
- solverDict() return the dictionary of the solver that is read from fvSolution.
- It calls another solverDict() function from solution class through inheritance (Can you explain it?). It returns the corresponding solver dictionary.

```
template<class Type>
const Foam::dictionary& Foam::fvMatrix<Type>::solverDict() const
{
    return psi_.mesh().solverDict
    (
        psi_.select
        (
            psi_.mesh().data::template getOrDefault<bool>
            ("finalIteration", false)
        )
    );
}
```

- Have a look at the fvMesh inheritance diagram. fvMesh *is* an fvSolution and fvSolution *is* a solution.



## Tracking the solve() function

- The `solve()` member function in `this->solve(solverDict())` receives a dictionary as an argument and is defined in `fvMatrixSolve.C` as,

```
template<class Type>
Foam::SolverPerformance<Type> Foam::fvMatrix<Type>::solve
(
    const dictionary& solverControls
)
{
    return psi_.mesh().solve(*this, solverControls);
}
```

- Another `solve()` member function is called. But this time it belongs to `fvMesh` class (why?) and receives a `fvMatrix<vector>` and a dictionary. Its definition is found in `fvMesh.C` as

```
Foam::SolverPerformance<Foam::vector> Foam::fvMesh::solve
(
    fvMatrix<vector>& m,
    const dictionary& dict
) const
{
    // Redirect to fvMatrix solver
    return m.solveSegregatedOrCoupled(dict);
}
```

- It will redirect us to the `fvMatrix` class again and call the `solveSegregatedOrCoupled()` member function that at the end calls a linear system solver.
- Now can you explain how the `solve()` function works on `pEqn`?

## LDU Addressing

## Introduction

- Now that we learned how the UEqn is constructed, we can have a look at each term in

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

- We will describe how each term is discretized in detail (i.e, time derivative, divergence, and Laplacian).
- However, In order to explain them, first, we need to understand how OpenFOAM stores the coefficients in the linear system of a discretized equation. Later, you will see that OpenFOAM mainly manipulates these coefficients during the discretization of each term.
- OpenFOAM uses a special and efficient way of storing the coefficients called Lower-Diagonal-Upper (LDU) decomposition.
- All the addressing of the coefficients is stored using the face order numbering (face addressing).

## fvMatrix class

- Recall fvMatrix member data:
  - ✓ An lduMatrix object (its superclass)
  - ✓ psi_: The volume field being discretized
  - ✓ dimensions_: The dimension set
  - ✓ source_: source term (right hand side)
  - ✓ internalCoeffs_: Contribution of the boundary conditions to the diagonal members of the coefficient matrix
  - ✓ boundaryCoeffs_: Contribution of the boundary conditions to the source term
  - ✓ faceFluxCorrectionPtr_: Non-orthogonal correction of the face flux field.

```
template<class Type>
Foam::fvMatrix<Type>::fvMatrix
(
    const GeometricField<Type,fvPatchField,volMesh>&
      psi,
    const dimensionSet& ds
)
:
    lduMatrix(psi.mesh()),
    psi_(psi),
    useImplicit_(false),
    lduAssemblyName_(),
    nMatrix_(0),
    dimensions_(ds),
    source_(psi.size(), Zero),
    internalCoeffs_(psi.mesh().boundary().size()),
    boundaryCoeffs_(psi.mesh().boundary().size()),
    faceFluxCorrectionPtr_(nullptr)
```

- The lduMatrix object is basically the coefficient matrix of the linear system (without the effects from BCs) which is stored in three components (lower, diagonal, and upper).
- Previously, we showed that the discretized momentum equation will take the form of

$$a_P^{\mathbf{u}}\mathbf{u}_P + \sum_N a_N^{\mathbf{u}}\mathbf{u}_N = \mathbf{r} - \nabla p \qquad \text{or} \qquad \mathbf{A}\mathbf{x} = \mathbf{B}$$

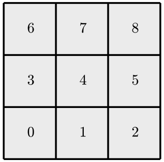
$\mathbf{A}$: coefficient matrix $\mathbf{x}$: field vector to be solved $\mathbf{B}$: source term vector

## Coefficients matrix

- Consider a uniform $3 \times 3$ cells mesh. For each cell, the discretized equation is of form

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

- They all together create a linear system ($\mathbf{Ax} = \mathbf{B}$). The coefficient matrix ($\mathbf{A}$) is $n \times n$, where $n$ is the number of cells (here $9 \times 9$).
- Cells are connected to each other through internal faces. Each face has an *owner* cell and a *neighbor* cell.
- The number of non-zero coefficients in each row and column depends on the number of internal faces of that cell index.
- Most coefficients in $\mathbf{A}$ are zero (white) except:
  O: Owner coefficients (lower triangle)
  N: Neighbor coefficients (upper triangle)
  P: Diagonal coefficients
- The number of non-zero coefficients in each triangle is equivalent to the number of internal faces (here 12).

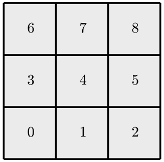


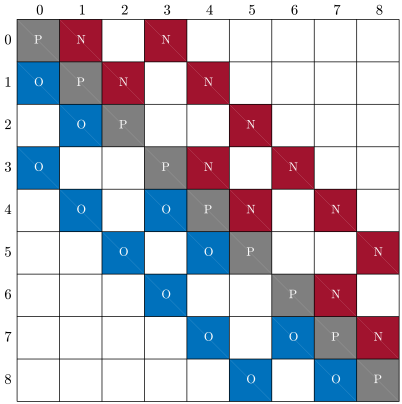
## Coefficients matrix

- Consider a uniform $3 \times 3$ cells mesh. For each cell, the discretized equation is of form

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

- They all together create a linear system ($\mathbf{Ax} = \mathbf{B}$). The coefficient matrix ($\mathbf{A}$) is $n \times n$, where $n$ is the number of cells (here $9 \times 9$).
- Cells are connected to each other through internal faces. Each face has an *owner* cell and a *neighbor* cell.
- The number of non-zero coefficients in each row and column depends on the number of internal faces of that cell index.
- Most coefficients in $\mathbf{A}$ are zero (white) except:
  O: Owner coefficients (lower triangle)
  N: Neighbor coefficients (upper triangle)
  P: Diagonal coefficients
- The number of non-zero coefficients in each triangle is equivalent to the number of internal faces (here 12).



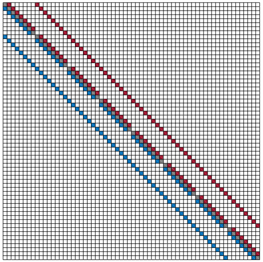


| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | P | N | | N | | | | | |
| 1 | O | P | N | | N | | | | |
| 2 | | O | P | | | N | | | |
| 3 | O | | | P | N | | N | | |
| 4 | | O | | O | P | N | | N | |
| 5 | | | O | | O | P | | | N |
| 6 | | | | O | | | P | N | |
| 7 | | | | | O | | O | P | N |
| 8 | | | | | | O | | O | P |

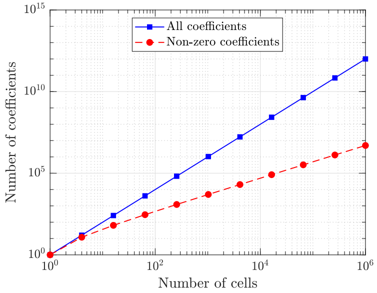
Coefficient matrix $\mathbf{A}$

## Coefficients matrix sparsity

- Storing the full matrix $\mathbf{A}$ is quite convenient for addressing and accessing the data. How about efficiency?
- Most $\mathbf{A}$ coefficients are zero and the matrix is extremely sparse.
- For a square uniform mesh, the number of coefficients grows with $n^2$ while the number of non-zero coefficients grows with $n$.
- A one million cells mesh has $10^{12}$ coefficients while the number of non-zero coefficients is less than $5 \times 10^6$ (only $0.0005\%$ of the coefficients!).
- OpenFOAM uses a special method to store only the non-zero coefficients. Only the non-zero coefficients of lower and upper triangles and diagonal coefficients are stored. Face addressing should also be stored to know to which face the non-zero coefficients belong.



$\mathbf{A}$ matrix for a $8 \times 8$ cells mesh



## Coefficients matrix addressing

- The `lduMatrix` and `lduAddressing` classes provide all the functionalities required for storing and accessing the coefficients.
- The values of the lower, upper, and diagonal coefficients are accessed through `lower()`, `upper()`, and `diag()` member function of `lduMatrix`.
- The non-zero coefficients of $\mathbf{A}$ of the $3 \times 3$ mesh is:
  `lower()` $= (a_{1,0}\ a_{3,0}\ a_{2,1}\ a_{4,1}\ a_{5,2}\ a_{4,3}\ a_{6,3}\ a_{5,4}\ a_{7,4}\ a_{8,5}\ a_{7,6}\ a_{8,7})$
  `diag()` $= (a_{0,0}\ a_{1,1}\ a_{2,2}\ a_{3,3}\ a_{4,4}\ a_{5,5}\ a_{6,6}\ a_{7,7}\ a_{8,8})$
  `upper()` $= (a_{0,1}\ a_{0,3}\ a_{1,2}\ a_{1,4}\ a_{2,5}\ a_{3,4}\ a_{3,6}\ a_{4,5}\ a_{4,7}\ a_{5,8}\ a_{6,7}\ a_{7,8})$
- The addressing is stored by the `lduAddressing` class and can be accessed through its members functions, i.e., `lowerAddr()`, `upperAddr()`, and `ownerStartAddr()`.
- The addressing of the $3 \times 3$ mesh is:
  `lowerAddr()` $= (0,\ 0,\ 1,\ 1,\ 2,\ 3,\ 3,\ 4,\ 4,\ 5,\ 6,\ 7)$
  `upperAddr()` $= (1,\ 3,\ 2,\ 4,\ 5,\ 4,\ 6,\ 5,\ 7,\ 8,\ 7,\ 8)$
  `ownerStartAddr()` $= (0,\ 2,\ 4,\ 5,\ 7,\ 9,\ 10,\ 11,\ 12,\ 12)$
- `lowerAddr()` returns list of owner cells of the internal faces while `upperAddr()` gives the list of neighbors of the internal faces. `ownerStartAddr()` specifies at which position a new column in the `lower()` or a new row in the `upper()` is started.

$$
\begin{array}{c|ccccccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\hline
0 & P & N &  & N &  &  &  &  &  \\
1 & O & P & N &  & N &  &  &  &  \\
2 &  & O & P &  &  & N &  &  &  \\
3 & O &  &  & P & N &  & N &  &  \\
4 &  & O &  & O & P & N &  & N &  \\
5 &  &  & O &  & O & P &  &  & N \\
6 &  &  &  & O &  &  & P & N &  \\
7 &  &  &  &  & O &  & O & P & N \\
8 &  &  &  &  &  & O &  & O & P
\end{array}
$$

$\mathbf{A}$ matrix for a $3 \times 3$ cells mesh

## icoLduAddressingFoam

- In order to understand and explain the LDU addressing mechanism in OpenFOAM, a special solver is created, `icoLduAddressingFoam`.
- The main intention is to calculate and store the full coefficient matrix in the linear system and solve it.
- The full momentum equation object is created `UEqnWithPressure`.
- The non-zero values of the coefficient matrix of this equation, i.e., `lower()`, `upper()`, `diag()`, and `source()` coefficients, as well as the LDU addressing, i.e., `lowerAddr()`, `upperAddr()`, and `ownerStartAddr()` are extracted and written to the output.

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

fvVectorMatrix UEqnWithPressure(UEqn == -fvc::grad(p));
if (piso.momentumPredictor())
{
    solve(UEqnWithPressure);
}

labelUList lAdd = UEqnWithPressure.lduAddr().lowerAddr();
labelUList uAdd = UEqnWithPressure.lduAddr().upperAddr();
labelUList oAdd = UEqnWithPressure.lduAddr().ownerStartAddr();
scalarField lower = UEqnWithPressure.lower();
scalarField upper = UEqnWithPressure.upper();
scalarField diag = UEqnWithPressure.diag();
vectorField source = UEqnWithPressure.source();

Info << "lowerAddr:" << lAdd << nl << endl;
Info << "upperAddr:" << uAdd << nl << endl;
Info << "ownerStartAddr:" << nl << oAdd << nl << endl;
Info << "lower:" << lower << nl << endl;
Info << "upper:" << upper << nl << endl;
Info << "diag:" << nl << diag << nl << endl;
Info << "source:" << nl << source << nl << endl;
```

## Full coefficient matrix

- The full coefficient matrix of the x-component of the momentum equation is constructed using the `simpleMatrix` class ($N \times N$, $N$ being the number of cells).
- `simpleMatrix` is a simple square matrix solver with scalar coefficients.
- All the members of the matrix and its source term (right-hand side) are initialized with zero.
- The diagonal, lower, and upper coefficients, as well as the source terms, are put in the right places.
- The diagonal, lower, and upper coefficients are computed using the face fluxes and are always scalars (`scalarField`), even for a `fvVectorMatrix`. The source term is a `vectorField` and diagonal coefficients will also change for each component by adding the contribution of boundary conditions.

```
label nCells = mesh.nCells();
simpleMatrix<scalar> CoeffMat(nCells);

// Initialization of matrix
for(label i = 0; i < nCells; i++)
{
    CoeffMat.source()[i] = 0.0;
    for(label j = 0; j < nCells; j++)
    {
        CoeffMat[i][j] = 0.0;
    }
}

// Assigning diagonal coefficients
for(label i = 0; i < nCells; ++i)
{
    CoeffMat[i][i] = diag[i];
    CoeffMat.source()[i] = source[i][0];
}

// Assigning off-diagonal coefficients
for(label faceI = 0; faceI < lAdd.size(); ++faceI)
{
    label l = lAdd[faceI];
    label u = uAdd[faceI];
    CoeffMat[l][u] = upper[faceI];
    CoeffMat[u][l] = lower[faceI];
}
```

## Boundary condition

- The effects of boundary conditions are added to the diagonals and source terms.
- In OpenFOAM,the contribution of the boundary condition to the diagonal coefficients are provided by `internalCoeffs()`, while `boundaryCoeffs()` returns the contribution of the boudanry condition to the source term.
- The impact of boundary condition is added to the linear system and now the linear system is fully constructed and ready to solve.
- The matrix and its source term are written to the output.
- The full linear system is then solved.

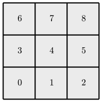
```
// Assigning contribution from BC
forAll(U.boundaryField(), patchI)
{
    const fvPatch &pp = U.boundaryField()[patchI].
      patch();
    forAll(pp, faceI)
    {
        label cellI = pp.faceCells()[faceI];
        CoeffMat[cellI][cellI] += UEqnWithPressure.
      internalCoeffs()[patchI][faceI][0];
        CoeffMat.source()[cellI] +=
      UEqnWithPressure.boundaryCoeffs()[patchI][
      faceI][0];
    }
}
```

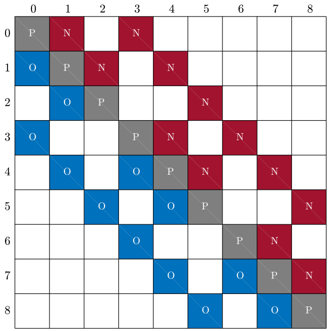
```
for(label i = 0; i < nCells; ++i)
{
    for(label j = 0; j < nCells; ++j)
    {
        Info << CoeffMat[i][j] << "\t";
    }
    Info << CoeffMat.source()[i] << endl;
}
Info << endl;

scalarField scalarFieldUx(CoeffMat.solve());
```

## icoFoam and full matrix solutions

- The full linear system for $u$ $(\mathbf{Ax} = \mathbf{B})$ is solved and the results are compared to `icoFoam`

$$\begin{pmatrix} 28.222 & -1.131 & 0 & -0.869 & 0 & 0 & 0 & 0 & 0 \\ -0.869 & 27.222 & -1.135 & 0 & -0.996 & 0 & 0 & 0 & 0 \\ 0 & -0.865 & 28.222 & 0 & 0 & -1.135 & 0 & 0 & 0 \\ -1.131 & 0 & 0 & 27.222 & -1.149 & 0 & -0.720 & 0 & 0 \\ 0 & -1.004 & 0 & -0.851 & 26.222 & -1.147 & 0 & -0.999 & 0 \\ 0 & 0 & -0.865 & 0 & -0.853 & 27.222 & 0 & 0 & -1.282 \\ 0 & 0 & 0 & -1.280 & 0 & 0 & 28.222 & -0.720 & 0 \\ 0 & 0 & 0 & 0 & -1.001 & 0 & -1.280 & 27.222 & -0.718 \\ 0 & 0 & 0 & 0 & 0 & -0.718 & 0 & -1.282 & 28.222 \end{pmatrix} \begin{pmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \\ u_7 \\ u_8 \end{pmatrix} = \begin{pmatrix} -1.155 \\ -2.440 \\ -1.118 \\ -1.199 \\ -3.075 \\ -0.762 \\ 6.333 \\ 2.406 \\ 4.575 \end{pmatrix}$$



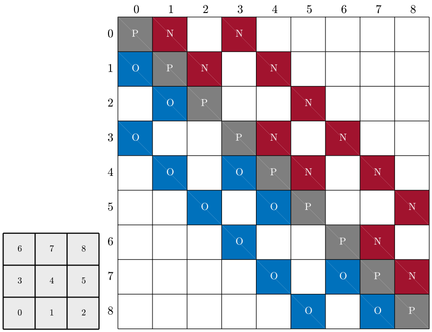


| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | P | N | | N | | | | | |
| 1 | O | P | N | | N | | | | |
| 2 | | O | P | | | N | | | |
| 3 | O | | | P | N | | N | | |
| 4 | | O | | O | P | N | | N | |
| 5 | | | O | | O | P | | | N |
| 6 | | | | O | | | P | N | |
| 7 | | | | | O | | O | P | N |
| 8 | | | | | | O | | O | P |

## icoFoam and full matrix solutions

- The full linear system for $u$ ($\mathbf{Ax} = \mathbf{B}$) is solved and the results are compared to icoFoam

$$
\begin{pmatrix}
28.222 & -1.131 & 0 & -0.869 & 0 & 0 & 0 & 0 & 0 \\
-0.869 & 27.222 & -1.135 & 0 & -0.996 & 0 & 0 & 0 & 0 \\
0 & -0.865 & 28.222 & 0 & 0 & -1.135 & 0 & 0 & 0 \\
-1.131 & 0 & 0 & 27.222 & -1.149 & 0 & -0.720 & 0 & 0 \\
0 & -1.004 & 0 & -0.851 & 26.222 & -1.147 & 0 & -0.999 & 0 \\
0 & 0 & -0.865 & 0 & -0.853 & 27.222 & 0 & 0 & -1.282 \\
0 & 0 & 0 & -1.280 & 0 & 0 & 28.222 & -0.720 & 0 \\
0 & 0 & 0 & 0 & -1.001 & 0 & -1.280 & 27.222 & -0.718 \\
0 & 0 & 0 & 0 & 0 & -0.718 & 0 & -1.282 & 28.222
\end{pmatrix}
\begin{pmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \\ u_7 \\ u_8 \end{pmatrix}
=
\begin{pmatrix} -1.155 \\ -2.440 \\ -1.118 \\ -1.199 \\ -3.075 \\ -0.762 \\ 6.333 \\ 2.406 \\ 4.575 \end{pmatrix}
$$



icoFoam, full matrix, and MATLAB results for $u$

| Cell No. | OpenFOAM (icoFoam) | OpenFOAM (full matrix) | MATLAB ($\mathbf{A}^{-1}\mathbf{B}$) |
|---|---|---|---|
| 0 | $-0.0462047$ | $-0.0462047$ | $-0.0462047$ |
| 1 | $-0.0972005$ | $-0.0972005$ | $-0.0972006$ |
| 2 | $-0.0434501$ | $-0.0434501$ | $-0.0434503$ |
| 3 | $-0.0449111$ | $-0.0449111$ | $-0.0449111$ |
| 4 | $-0.1167420$ | $-0.1167420$ | $-0.1167420$ |
| 5 | $-0.0216980$ | $-0.0216980$ | $-0.0216980$ |
| 6 | 0.2267990 | 0.2267990 | 0.2267992 |
| 7 | 0.1745530 | 0.1745530 | 0.1745531 |
| 8 | 0.2403400 | 0.2403400 | 0.2403402 |

## `ddt(U)`

## ddt(U)

- The ddt(U) in

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

is a global function defined in the namespace fvm inside the fvmDdt.C as:

```
template<class Type>
tmp<fvMatrix<Type>>
ddt
(
    const GeometricField<Type, fvPatchField, volMesh>& vf
)
{
    return fv::ddtScheme<Type>::New
    (
        vf.mesh(),
        vf.mesh().ddtScheme("ddt(" + vf.name() + ')')
    ).ref().fvmDdt(vf);
}
```

- The function receives a GeometricField (here U) and returns a type tmp<fvMatrix<Type>>.
- The New function is a selector that returns a type tmp<ddtScheme<Type>> which is the abstract base class for the all the time discretization schemes (ddt).

## Theory and implementation

- The `fvmDdt(vf)` function is called which is a pure virtual function declared in the abstract base class `ddtScheme` and overridden in the subclasses (e.g., `EulerDdtScheme`, `backwardDdtScheme`, etc.).
- Here we will look at the `fvmDdt(vf)` implementation in `EulerDdtScheme`.
- The Euler first-order implicit method discretize the time derivative term as:

$$\int_V \frac{\partial \phi}{\partial t}\,\mathrm{d}V = \frac{\phi^{\text{new}} - \phi^{\text{old}}}{\Delta t} V_P = \underbrace{\phi^{\text{new}} \frac{V_P}{\Delta t}}_{\text{diagonal}} - \underbrace{\phi^{\text{old}} \frac{V_P}{\Delta t}}_{\text{source}}$$

- In the case of dynamic mesh, the old cell volume is used in the source term.
- The contribution of the time derivative term to the diagonal members is inversely proportional to the time step size. That is why reducing time step enhances the matrix diagonal dominance and thereby stabilizes the simulation.

```
{
    tmp<fvMatrix<Type>> tfvm
    (
        new fvMatrix<Type>
        (
            vf,
            vf.dimensions()*dimVol/dimTime
        )
    );

    fvMatrix<Type>& fvm = tfvm.ref();

    scalar rDeltaT = 1.0/mesh().time().
      deltaTValue();

    fvm.diag() = rDeltaT*mesh().Vsc();

    if (mesh().moving())
    {
        fvm.source() = rDeltaT*vf.oldTime().
      primitiveField()*mesh().Vsc0();
    }
    else
    {
        fvm.source() = rDeltaT*vf.oldTime().
      primitiveField()*mesh().Vsc();
    }

    return tfvm;
}
```

## laplacian(nu,U)

## Theory

- The Laplacian term is discretized using the Gauss' theorem, as

$$\int_V \nabla \cdot (\gamma \nabla \phi)\,\mathrm{d}V = \sum_f \gamma_f\, \mathbf{s}_f \cdot (\nabla\phi)_f.$$

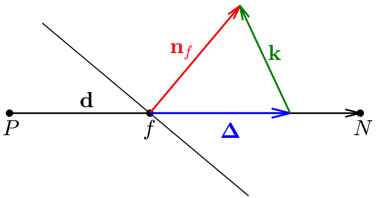
- The diffusion coefficient is mostly interpolated on the faces using the `linear` scheme. Therefore, it is only a matter of calculating surface normal gradients, $(\nabla\phi)_f$. The Laplacian scheme in the `fvSchemes` specifies:

```
laplacian(gamma,phi)    Gauss <interpolation scheme> <snGrad scheme>
```

- The surface normal vector $\mathbf{s}_f$ is defined as $\mathbf{s}_f = \mathbf{n}_f|\mathbf{s}_f|$, where $\mathbf{n}_f$ is the unit surface normal vector and $|\mathbf{s}_f|$ is the surface area.

- surface normal gradient can be decomposed into *orthogonal* and *non-orthogonal* parts, as $\mathbf{n}_f = \mathbf{\Delta} + \mathbf{k}$:

$$\begin{aligned}\mathbf{n}_f \cdot (\nabla\phi)_f &= \mathbf{\Delta}\cdot(\nabla\phi)_f + \mathbf{k}\cdot(\nabla\phi)_f \\ &= \underbrace{|\mathbf{\Delta}|\frac{\phi_N - \phi_P}{|\mathbf{d}|}}_{\text{orthogonal}} + \underbrace{\mathbf{k}\cdot(\nabla\phi)_f}_{\text{non-orthogonal}}\end{aligned}$$



- The orthogonal and non-orthogonal parts are treated implicitly and explicitly.

## Theory

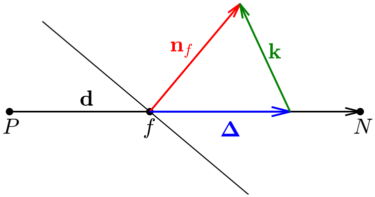
$$\sum_f \gamma_f \, \mathbf{s}_f \cdot (\nabla\phi)_f = \sum_f \gamma_f |\mathbf{s}_f| \, \mathbf{n}_f \cdot (\nabla\phi)_f .$$

- Substituting the decomposed surface normal gradient into the discretized Laplacian term theorem yields

$$\sum_f \gamma_f |\mathbf{s}_f| \, \mathbf{n}_f \cdot (\nabla\phi)_f = \sum_f \gamma_f |\mathbf{s}_f| \left( |\mathbf{\Delta}| \frac{\phi_N - \phi_P}{|\mathbf{d}|} + \mathbf{k} \cdot (\nabla\phi)_f \right)$$

- There are different ways to choose and compute $\mathbf{\Delta}$ and $\mathbf{k}$. For more information, see prof. Jasak's PhD thesis. The over-relaxed approach states that $\mathbf{k}$ should be orthogonal to $\mathbf{n}_f$. Hence,

$$\mathbf{\Delta} = \frac{\mathbf{d}}{\mathbf{d} \cdot \mathbf{n}_f}, \quad \mathbf{k} = \mathbf{n}_f - \mathbf{\Delta}$$



- So, the discretized Laplacian term becomes

$$\sum_f \gamma_f |\mathbf{s}_f| \left( \frac{\phi_N - \phi_P}{\mathbf{d} \cdot \mathbf{n}_f} + \mathbf{k} \cdot (\nabla\phi)_f \right) = \underbrace{\sum_f \gamma_f |\mathbf{s}_f| \left( \frac{\phi_N - \phi_P}{\mathbf{d} \cdot \mathbf{n}_f} \right)}_{\text{Implicit orthogonal}} + \underbrace{\sum_f \gamma_f |\mathbf{s}_f| \left( \mathbf{k} \cdot (\nabla\phi)_f \right)}_{\text{Explicit non-orthogonal correction}}$$

## laplacian(nu,U)

- Similar to ddt(U), laplacian(nu,U) in

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

  is a global function defined in the namespace fvm inside the fvmLaplacian.C. There are several definitions for the laplacian. In order to find out which is used here, we need to assess the input arguments.

- In the createFields.H of icoFoam, nu is created as a dimensionedScalar.

```
dimensionedScalar nu
(
    "nu",
    dimViscosity,
    transportProperties
);
```

- dimensionedScalar is a typedef:

```
typedef dimensioned<scalar> dimensionedScalar;
```

- Therefore, we should look for a definition of laplacian that receives two input arguments with type dimensioned and GeometricField.

## laplacian function

- The corresponding definition reads:

```
template<class Type, class GType>
tmp<fvMatrix<Type>>
laplacian
(
    const dimensioned<GType>& gamma,
    const GeometricField<Type, fvPatchField, volMesh>& vf
)
{
    const GeometricField<GType, fvsPatchField, surfaceMesh> Gamma
    (
        IOobject
        (
            gamma.name(),
            vf.instance(),
            vf.mesh(),
            IOobject::NO_READ
        ),
        vf.mesh(),
        gamma
    );

    return fvm::laplacian(Gamma, vf);
}
```

- The template function converts the `dimensionedScalar` into a `surfaceScalarField` and calls another definition of `laplacian`. The return type is `tmp<fvMatrix<Type>>`.

## laplacian function

- Now the following function is called

```
template<class Type, class GType>
tmp<fvMatrix<Type>>
laplacian
(
    const GeometricField<GType, fvsPatchField, surfaceMesh>& gamma,
    const GeometricField<Type, fvPatchField, volMesh>& vf
)
{
    return fvm::laplacian
    (
        gamma,
        vf,
        "laplacian(" + gamma.name() + ',' + vf.name() + ')'
    );
}
```

- Here again another definition of the `laplacian` function is called. The return type is `tmp<fvMatrix<Type>>`.
- The third argument is a type `word` (similar to `string`). In this case in returns:

```
"laplacian(nu,U)"
```

- `word` is a subclass of `string` which is a subclass of `std::string`. `+` is an operator overloaded inside `std::string`.

## gaussLaplacianScheme

- Finally the following laplacian function is executed

```
template<class Type, class GType>
tmp<fvMatrix<Type>>
laplacian
(
    const GeometricField<GType, fvsPatchField, surfaceMesh>& gamma,
    const GeometricField<Type, fvPatchField, volMesh>& vf,
    const word& name
)
{
    return fv::laplacianScheme<Type, GType>::New
    (
        vf.mesh(),
        vf.mesh().laplacianScheme(name)
    ).ref().fvmLaplacian(gamma, vf);
}
```

- Here again, a selector function New is used that is connected to the OpenFOAM runtime selection mechanism. It returns an object to the employed scheme class. Therefore, the correct fvmLaplacian member function will be called.
- The available schemes for discretization are Gauss and relaxedNonOrthoGauss. In other words, the abstract base class laplacianScheme with pure virtual fvmLaplacian member function has two subclasses.
- Using Gauss, the overridden fvmLaplacian member function of gaussLaplacianScheme is called. Now let's find the implementation of fvmLaplacian.

## fvmLaplacian

- The template `fvmLaplacian` function is declared and defined inside `gaussLaplacianScheme.H` and `gaussLaplacianScheme.C`, respectively.

```
tmp<fvMatrix<Type>> fvmLaplacian
(
    const GeometricField<GType, fvsPatchField, surfaceMesh>&,
    const GeometricField<Type, fvPatchField, volMesh>&
);
```

```
template<class Type, class GType>
tmp<fvMatrix<Type>>
gaussLaplacianScheme<Type, GType>::fvmLaplacian
(
    const GeometricField<GType, fvsPatchField, surfaceMesh>& gamma,
    const GeometricField<Type, fvPatchField, volMesh>& vf
)
{
    const fvMesh& mesh = this->mesh();

    const surfaceVectorField Sn(mesh.Sf()/mesh.magSf());

    //Continued ...
}
```

## fvmLaplacian

- The template `fvmLaplacian` function is declared and defined inside `gaussLaplacianScheme.H` and `gaussLaplacianScheme.C`, respectively.

```
tmp<fvMatrix<Type>> fvmLaplacian
(
    const GeometricField<GType, fvsPatchField, surfaceMesh>&,
    const GeometricField<Type, fvPatchField, volMesh>&
);
```

```
template<class Type, class GType>
tmp<fvMatrix<Type>>
gaussLaplacianScheme<Type, GType>::fvmLaplacian
(
    const GeometricField<GType, fvsPatchField, surfaceMesh>& gamma,
    const GeometricField<Type, fvPatchField, volMesh>& vf
)
{
    const fvMesh& mesh = this->mesh();

    const surfaceVectorField Sn(mesh.Sf()/mesh.magSf());

    //Continued ...
}
```

- This is a general definition for any types of `fvMatrix` and `GType`. However, the compiler picks up a totally different definition for scalar diffusion coefficient. In this presentation, we will only focus on the scalar diffusion coefficient and leave out the general tensor coefficient for you to study.

## Partial-specialisation for scalar diffusion coefficient

- The calculations of the Laplacian term for a scalar diffusion parameter (`GType=scalar`) is carried out using *partial-specialisation*. Macros are used to enable this functionality.
- After declaration of class in `gaussLaplacianScheme.H`, the macro function `defineFvmLaplacianScalarGamma(Type)` is defined as the declaration of the `fvmLalapcian` and `fvcLalapcian` functions for different types of `fvMatrix` while keeping `GType scalar`. Subsequently, it is called for all possible `fvMatrix` types. Note the escape character \ which escapes the new line.
- The macro is performed as a pre-processing step (before compilation). The current macro is simply a text replacement that repeats the same function declaration for different `fvMatrix` types.
- After the pre-processing step, all the macros are carried out and the compiler only sees several function declarations for different `fvMatrix` types.
- Each function declaration needs a definition. The definitions are carried out inside `gaussLaplacianSchemes.C` using other macros.

```
#define defineFvmLaplacianScalarGamma(Type)                                \
                                                                           \
template<>                                                                 \
tmp<fvMatrix<Type>> gaussLaplacianScheme<Type, scalar>::fvmLaplacian \
(                                                                          \
    const GeometricField<scalar, fvsPatchField, surfaceMesh>&,             \
    const GeometricField<Type, fvPatchField, volMesh>&                     \
);                                                                         \
                                                                           \
template<>                                                                 \
tmp<GeometricField<Type, fvPatchField, volMesh>>                           \
gaussLaplacianScheme<Type, scalar>::fvcLaplacian                           \
(                                                                          \
    const GeometricField<scalar, fvsPatchField, surfaceMesh>&,             \
    const GeometricField<Type, fvPatchField, volMesh>&                     \
);

defineFvmLaplacianScalarGamma(scalar);
defineFvmLaplacianScalarGamma(vector);
defineFvmLaplacianScalarGamma(sphericalTensor);
defineFvmLaplacianScalarGamma(symmTensor);
defineFvmLaplacianScalarGamma(tensor);
```

## fvmLaplacian for scalar diffusion coefficient

- The gaussLaplacianSchemes.C mainly looks like:

```
makeFvLaplacianScheme(gaussLaplacianScheme)


#define declareFvmLaplacianScalarGamma(Type)                                   \
                                                                               \
template<>                                                                     \
Foam::tmp<Foam::fvMatrix<Foam::Type>>                                          \
Foam::fv::gaussLaplacianScheme<Foam::Type, Foam::scalar>::fvmLaplacian         \
(                                                                              \
    const GeometricField<scalar, fvsPatchField, surfaceMesh>& gamma,           \
    const GeometricField<Type, fvPatchField, volMesh>& vf                      \
)                                                                              \
{                                                                              \
    const fvMesh& mesh = this->mesh();                                         \

    //Continued ...
}

declareFvmLaplacianScalarGamma(scalar);
declareFvmLaplacianScalarGamma(vector);
declareFvmLaplacianScalarGamma(sphericalTensor);
declareFvmLaplacianScalarGamma(symmTensor);
declareFvmLaplacianScalarGamma(tensor);
```

- The macro function declareFvmLaplacianScalarGamma(Type) is defined as the definition of the fvmLalapcian and fvcLalapcian functions where type of fvMatrix is an input variable. The naming of the macro function is misleading! It is then called for all possible fvMatrix types.

## fvmLaplacian for scalar diffusion coefficient

- Let's recall the theory

$$\int_V \nabla \cdot (\gamma \nabla \phi)\, \mathrm{d}V = \underbrace{\sum_f \gamma_f |\mathbf{s}_f| \left( \frac{\phi_N - \phi_P}{\mathbf{d} \cdot \mathbf{n}_f} \right)}_{\text{Implicit orthogonal}} + \underbrace{\sum_f \gamma_f |\mathbf{s}_f| \left( \mathbf{k} \cdot (\nabla \phi)_f \right)}_{\text{Explicit non-orthogonal correction}}$$

- The `mesh` object is created.
- `gammaMagSf` object is created as a `surfaceSclaraField` which is diffusion coefficient time surface area magnitude ($\gamma_f |\mathbf{s}_f|$).
- `magSf()` is a member function of `fvMesh` class that return the mesh face areas.
- Then, `fmvLaplacianUncorrected` function is called that receives three arguments. It is responsible for the implicit orthogonal part of the Laplacian discretized equation.

```
template<>                                                                    \
Foam::tmp<Foam::fvMatrix<Foam::Type>>                                         \
Foam::fv::gaussLaplacianScheme<Foam::Type, Foam::scalar>::fvmLaplacian \
(                                                                             \
    const GeometricField<scalar, fvsPatchField, surfaceMesh>& gamma,         \
    const GeometricField<Type, fvPatchField, volMesh>& vf                    \
)                                                                             \
{                                                                             \
    const fvMesh& mesh = this->mesh();                                        \
                                                                              \
    GeometricField<scalar, fvsPatchField, surfaceMesh> gammaMagSf            \
    (                                                                         \
        gamma*mesh.magSf()                                                    \
    );                                                                        \
                                                                              \
    tmp<fvMatrix<Type>> tfvm = fvmLaplacianUncorrected                        \
    (                                                                         \
        gammaMagSf,                                                           \
        this->tsnGradScheme_().deltaCoeffs(vf),                               \
        vf                                                                    \
    );                                                                        \
    fvMatrix<Type>& fvm = tfvm.ref();                                         \
```

- `deltaCoeffs` function returns the term $\frac{1}{\mathbf{d} \cdot \mathbf{n}_f}$ in the Laplacian equation. It is slightly different as it is a stabilized form for very bad meshes. We will show it in Doxygen.

## fmvLaplacianUncorrected (implicit orthogonal)

$$\sum_f \gamma_f |\mathbf{s}_f| \left( \frac{\phi_N - \phi_P}{\mathbf{d}\cdot\mathbf{n}_f} \right) = \sum_f \frac{\gamma_f |\mathbf{s}_f|}{\mathbf{d}\cdot\mathbf{n}_f}\phi_N - \sum_f \frac{\gamma_f |\mathbf{s}_f|}{\mathbf{d}\cdot\mathbf{n}_f}\phi_P$$

- `tvfm` object is created as a `tmp<fvMatrix<Type>>`, using `new` operator with correct dimensions. Then the `ref()` function is called to get a non-const reference.
- The first term in the above equation is the contribution of the neighbors which will fill the off-diagonal members of the `fvMtarix`. Therefore, the `upper()` values are set to the first term. The `lower()` is not set here, since Laplacian creates a symmetric matrix.
- The second term is the contribution of the owners which will change the diagonal members of the coefficient matrix.
- If one loops all the faces of one cell, $\phi_P$ does not change and the second term becomes $-\phi_P \sum_f \frac{\gamma_f |\mathbf{s}_f|}{\mathbf{d}\cdot\mathbf{n}_f}$

```
template<class Type, class GType>
tmp<fvMatrix<Type>>
gaussLaplacianScheme<Type, GType>::fvmLaplacianUncorrected
(
    const surfaceScalarField& gammaMagSf,
    const surfaceScalarField& deltaCoeffs,
    const GeometricField<Type, fvPatchField, volMesh>& vf
)
{
    tmp<fvMatrix<Type>> tfvm
    (
        new fvMatrix<Type>
        (
            vf,
            deltaCoeffs.dimensions()*gammaMagSf.dimensions
        ()*vf.dimensions()
        )
    );
    fvMatrix<Type>& fvm = tfvm.ref();

    fvm.upper() = deltaCoeffs.primitiveField()*gammaMagSf.
       primitiveField();
    fvm.negSumDiag();
```

- Therefore, for each cell, one can sum up all the off-diagonal members and negate the value to calculate the diagonal coefficient. This is exactly done inside `negSumDiag()`.

## fmvLaplacianUncorrected, boundary conditions

- In OpenFOAM, all boundary conditions are a mix of `fixedValue` and `fixedGradient` conditions. Here we will only present the `fixedValue` condition and leave the `fixedGradient` as a practice for you.
- For a cell containing both internal and boundary faces (not coupled) the discretized equation gives:

$$\underbrace{\sum_{f_i} \gamma_{f_i} |\mathbf{s}_{f_i}| \left( \frac{\phi_N - \phi_P}{\mathbf{d}_i \cdot \mathbf{n}_{f_i}} \right)}_{\text{Internal faces}} + \underbrace{\sum_{f_b} \gamma_{f_b} |\mathbf{s}_{f_b}| \left( \frac{\phi_b - \phi_P}{\mathbf{d}_b \cdot \mathbf{n}_{f_b}} \right)}_{\text{Boundary faces}}$$

- The second term of the above equation can be rewritten as

$$\sum_{f_b} \frac{\gamma_{f_b} |\mathbf{s}_{f_b}|}{\mathbf{d}_b \cdot \mathbf{n}_{f_b}} \phi_b - \sum_{f_b} \frac{\gamma_{f_b} |\mathbf{s}_{f_b}|}{\mathbf{d}_b \cdot \mathbf{n}_{f_b}} \phi_P$$

- For a `fixedValue` type BC, The first term is a constant value and thus contributes to the source term (right-hand side of the linear system). The second term has a contribution from owners and affects the diagonal coefficients.
- Generally, the boundary conditions only affect the diagonal coefficients and/or the source terms. In OpenFOAM, the contribution of BCs to the diagonal coefficients is stored via `internalCoeffs` while `boundaryCoeffs` is responsible for contributions of BCs to the source terms.

## fmvLaplacianUncorrected, boundary conditions

- A forAll loop is performed on the boundary patches.
- The diagonal contribution is not directly added to the coefficient matrix. They are just stored and later will be added to the diagonal coefficients (just before obtaining the solution).
- pGamma is value of gammaMagSf on the boundary patch, i.e., $\gamma_{f_b}|\mathbf{s}_{f_b}|$.

```
forAll(vf.boundaryField(), patchi)
{
    const fvPatchField<Type>& pvf = vf.boundaryField()[patchi];
    const fvsPatchScalarField& pGamma = gammaMagSf.boundaryField()[patchi];
    const fvsPatchScalarField& pDeltaCoeffs =
        deltaCoeffs.boundaryField()[patchi];

    if (pvf.coupled())
    {
        fvm.internalCoeffs()[patchi] =
            pGamma*pvf.gradientInternalCoeffs(pDeltaCoeffs);
        fvm.boundaryCoeffs()[patchi] =
           -pGamma*pvf.gradientBoundaryCoeffs(pDeltaCoeffs);
    }
    else
    {
        fvm.internalCoeffs()[patchi] = pGamma*pvf.gradientInternalCoeffs();
        fvm.boundaryCoeffs()[patchi] = -pGamma*pvf.gradientBoundaryCoeffs();
    }
}
```

- The rest of equation is implemented inside the functions gradientInternalCoeffs and gradientBoundaryCoeffs. These are virtual functions that are overridden for each boundary condition.
- The minus sign of the boundaryCoeffs is because it is moved to the right hand side.

$$\sum_{f_b} \frac{\gamma_{f_b}|\mathbf{s}_{f_b}|}{\mathbf{d}_b \cdot \mathbf{n}_{f_b}} \phi_b - \sum_{f_b} \frac{\gamma_{f_b}|\mathbf{s}_{f_b}|}{\mathbf{d}_b \cdot \mathbf{n}_{f_b}} \phi_P$$

## fixedValue boundary conditions

- The functions `gradientInternalCoeffs` and `gradientBoundaryCoeffs` are implemenetd inside the basic boundary conditions (e.g., `fixedValue`, `fixedGradient`, `mixed`, etc.). Derived boundary conditions are subclasses of the basics and picks up their implementation.
- Here we will have a look at the implementation of these functions inside `fixedValueFvPatchField`. Later, you can assess other boundary conditions.
- One can see the consistency of theory and implementation.

```
template<class Type>
Foam::tmp<Foam::Field<Type>>
Foam::fixedValueFvPatchField<Type>::
        gradientInternalCoeffs() const
{
    return -pTraits<Type>::one*this->patch().
        deltaCoeffs();
}


template<class Type>
Foam::tmp<Foam::Field<Type>>
Foam::fixedValueFvPatchField<Type>::
        gradientBoundaryCoeffs() const
{
    return this->patch().deltaCoeffs()*(*this);
}
```

$$\sum_{f_b}\frac{\gamma_{f_b}|\mathbf{s}_{f_b}|}{\mathbf{d}_b\cdot\mathbf{n}_{f_b}}\phi_b-\sum_{f_b}\frac{\gamma_{f_b}|\mathbf{s}_{f_b}|}{\mathbf{d}_b\cdot\mathbf{n}_{f_b}}\phi_P=\sum_{f_b}\underbrace{\gamma_{f_b}|\mathbf{s}_{f_b}|}_{\texttt{pGamma}}\underbrace{\frac{\phi_b}{\mathbf{d}_b\cdot\mathbf{n}_{f_b}}}_{\texttt{gradientBoundaryCoeffs}}-\sum_{f_b}\underbrace{\gamma_{f_b}|\mathbf{s}_{f_b}|}_{\texttt{pGamma}}\underbrace{\frac{1}{\mathbf{d}_b\cdot\mathbf{n}_{f_b}}}_{\texttt{gradientInternalCoeffs}}\phi_P$$

## Explicit non-orthogonal correction

- Now that the implicit orthogonal (i.e., uncorrected) part of the Laplacian is fully calculated, let's calculate the explicit non-orthogonal correction term. Recall

$$\int_V \nabla \cdot (\gamma \nabla \phi)\, \mathrm{d}V = \underbrace{\sum_f \gamma_f |\mathbf{s}_f| \left( \frac{\phi_N - \phi_P}{\mathbf{d} \cdot \mathbf{n}_f} \right)}_{\text{Implicit orthogonal}} + \underbrace{\sum_f \gamma_f |\mathbf{s}_f| \left( \mathbf{k} \cdot (\nabla \phi)_f \right)}_{\text{Explicit non-orthogonal}}$$

- The `corrected()` function returns a boolean that specifies whether non-orthogonal correction is required. For instance, it returns `true` for `corrected` scheme.

- After that, we have another `if` statement that checks the `fluxRequired` flag in the `fvSchemes`. It specifies whether the non-orthogonal correction should be performed for the `flux()` function.
- The `flux()` method of `fvMatrix` class is used to correct the face fluxes. In the pressure velocity coupling algorithm for incomrepssible flows, the `flux()` method is called on the p:

```
phi = phiHbyA - pEqn.flux();
```

```
if (this->tsnGradScheme_().corrected())                          \
{                                                                \
    if (mesh.fluxRequired(vf.name()))                            \
    {                                                            \
        fvm.faceFluxCorrectionPtr() = new                        \
        GeometricField<Type, fvsPatchField, surfaceMesh>         \
        (                                                        \
            gammaMagSf*this->tsnGradScheme_().correction(vf)     \
        );                                                       \
                                                                 \
        fvm.source() -=                                          \
            mesh.V()*                                            \
            fvc::div                                             \
            (                                                    \
                *fvm.faceFluxCorrectionPtr()                     \
            )().primitiveField();                                \
    }                                                            \
```

## Face flux correction

- `fluxRequired` flag is set in the `fvSchemes`. For example:

```
fluxRequired
{
    default         no;
    p               ;
}
```

- The above dictionary specifies that if we are discretizing a Laplacian term for `p` and we need to perform the non-orthogonal correction, we should include this correction for the `flux()` function of `p`.
- Usually solvers overrides the `fluxRequired` flag to `true` for pressure field. For instance, the end of `createFileds.H` of `icoFoam` reads:

```
mesh.setFluxRequired(p.name());
```

- Finally, when the `faceFluxCorrectionPtr_` is appropriately constructed, it is used to correct the `fieldFlux` in `flux()` method of `fvMatrix` as

```
    if (faceFluxCorrectionPtr_)
    {
        fieldFlux += *faceFluxCorrectionPtr_;
    }
```

## Explicit non-orthogonal correction

- We derived the Non-orthogonal correction term as $\sum_f \gamma_f |\mathbf{s}_f| \left(\mathbf{k} \cdot (\nabla\phi)_f\right)$.
- The `faceFluxCorrectionPtr()` is dynamically allocated as a `GeometricField` and set to

$$\underbrace{\texttt{gammaMagSf}}_{\gamma_f|\mathbf{s}_f|} * \underbrace{\texttt{this->tsnGradScheme\_().correction(vf)}}_{\mathbf{k}\cdot(\nabla\phi)_f}$$

- One can see that the correction term is constructed accurately. Here, we will not show the details of the `correction(vf)` function.

```
if (this->tsnGradScheme_().corrected())                                \
{                                                                      \
    if (mesh.fluxRequired(vf.name()))                                  \
    {                                                                  \
        fvm.faceFluxCorrectionPtr() = new                              \
        GeometricField<Type, fvsPatchField, surfaceMesh>               \
        (                                                              \
            gammaMagSf*this->tsnGradScheme_().correction(vf)           \
        );                                                             \
                                                                       \
        fvm.source() -=                                                \
            mesh.V()*                                                  \
            fvc::div                                                   \
            (                                                          \
                *fvm.faceFluxCorrectionPtr()                           \
            )().primitiveField();                                      \
    }                                                                  \
```

- The divergence of the correction term is calculated explicitly and multiplied by the cell volumes (because of volume integration) and subtracted from the discretized equation source term (moved to the right-hand side).
- As you can see the non-orthogonal correction is an *explicit* term that contributes to the right-hand side. Therefore, applying non-orthogonal correction makes the whole linear system more explicit and so it needs more loops to converge.

## Explicit non-orthogonal correction

- If the fluxRequired flag is inactive (as it is in fvm::laplacian(nu,U)), the else statement activates.

```
        else                                                                   \
        {                                                                      \
            fvm.source() -=                                                    \
                mesh.V()*                                                      \
                fvc::div                                                       \
                (                                                              \
                    gammaMagSf*this->tsnGradScheme_().correction(vf)           \
                )().primitiveField();                                          \
        }                                                                      \
    }                                                                          \
                                                                               \
    return tfvm;                                                               \
```

- It is basically similar calculations without creating and storing faceFluxCorrectionPtr().