Immersed Boundary Method in FOAM

Theory, Implementation and Use

Hrvoje Jasak and Željko Tuković

Chalmers University, Gothenburg

Faculty of Mechanical Engineering and Naval Architecture, Zagreb

Outline

Objective

- Describe the implementation of the Immersed Boundary Method in OpenFOAM
- Demonstrate application of the immersed boundary method on tutorial cases

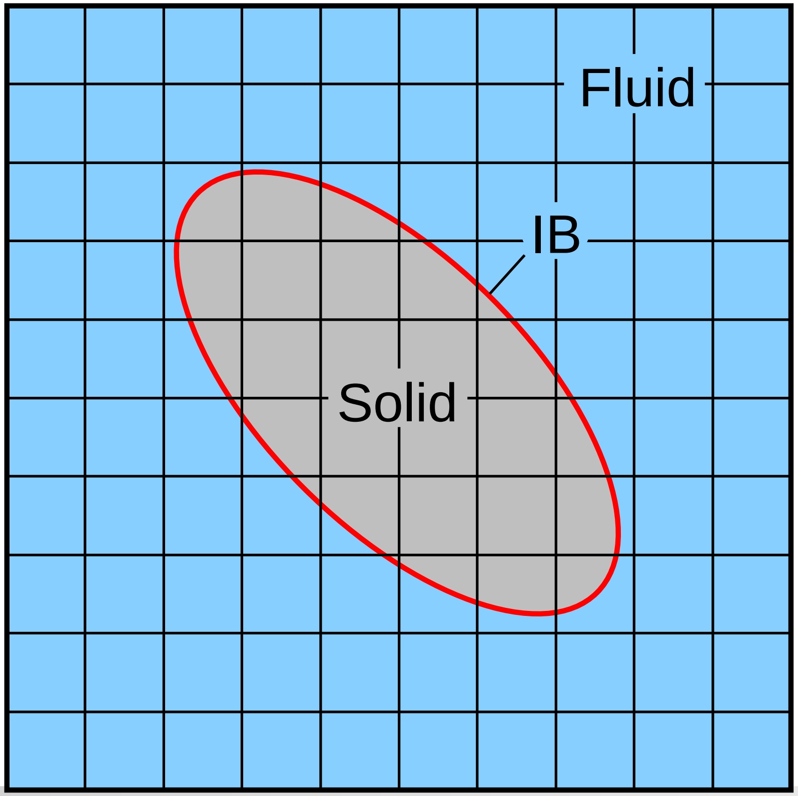
Topics

- General framework of the Immersed Boundary Method (IBM)
- Selected IBM approach
- Imposition of Dirichlet and Neumann boundary conditions
- Treatment of the pressure equation
- Implementation details: Class layout
- Tutorial cases and settings
- Fitting functions and high-Re flows
- Wall function implementation in body-fitted meshes
- Wall functions on immersed boundary patch
- Turbulent flow tutorial case

Immersed Boundary Method (IBM)

Immersed Boundary Method: Non-Conformal Boundary Surfaces

- Simulation of the flow around immersed boundary is carried out on a grid (usually Cartesian) which does not conform to the boundary shape
- Immersed boundary (IB) is represented by surface grid
- IB boundary conditions modify the equations in cells which interact with the immersed boundary



IBM: Pros and Cons

Advantages of IBM Over Body-Fitted Mesh Methods

- Substantially simplified grid generation for complex geometry
- Inclusion of body motion is relatively simple due to the use of stationary, non-deforming background grids

Disadvantages of IBM Over Body-Fitted Mesh Methods

- Imposition of boundary conditions at IB is not straightforward: special techniques are developed and implemented
- Problem with grid resolution control in boundary layers: effective near-wall mesh size is approx 50% larger than in equivalent body-fitted mesh
- Limited to low and moderate Reynolds number flows: this is resolved using the Immersed Boundary wall function implementation

Wish List

1. IBM solution MUST mimic the equivalent body-fitted mesh solution
2. Minimal interaction in top-level code: flow solvers and auxiliary models to be used without coding changes
3. Remove limitations on background mesh structure: must work with polyhedra
4. Automate mesh refinement under the IB surface

Boundary Conditions in IBM

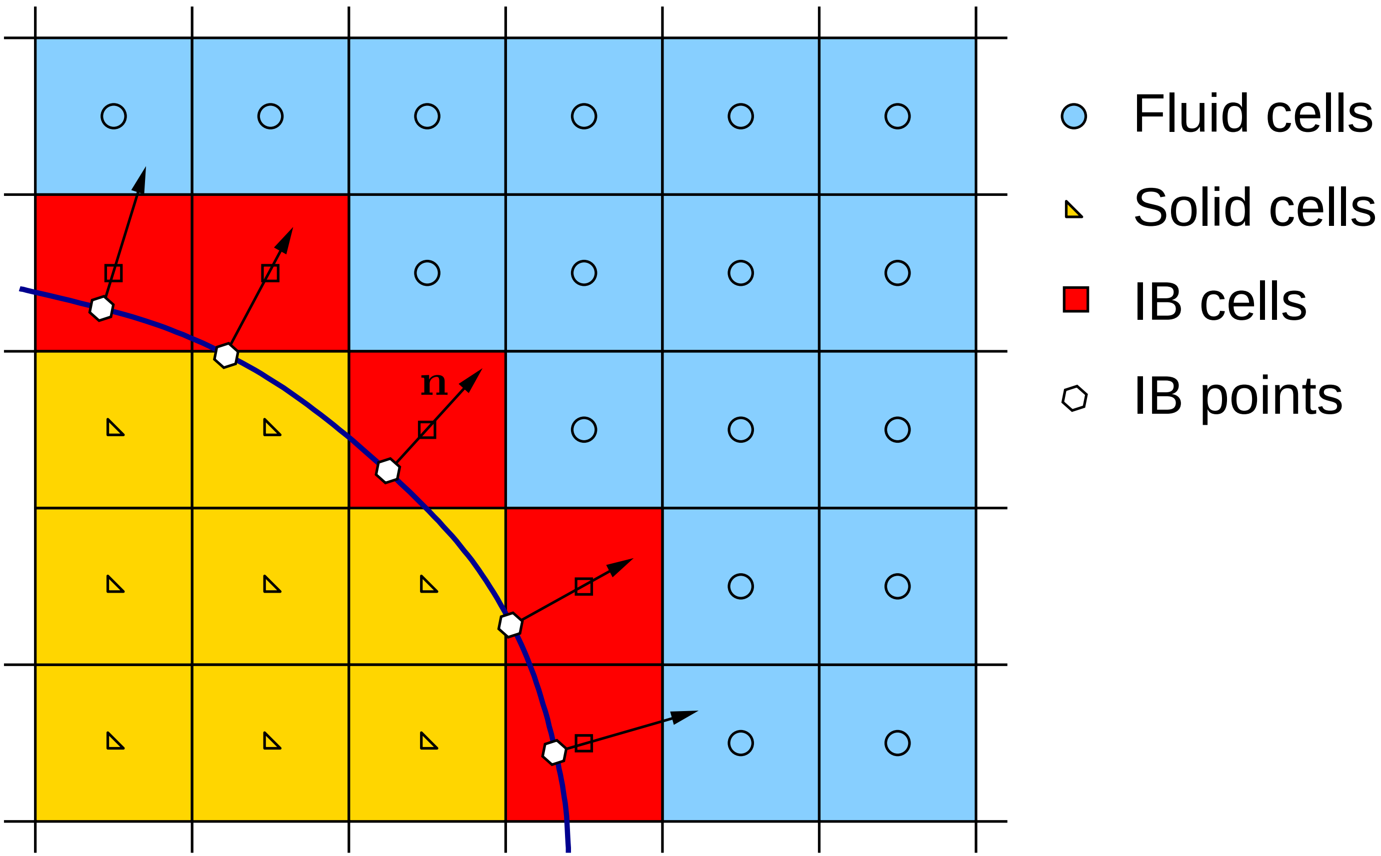
Imposition of Boundary Conditions at the IB Distinguishes one IB Method from Another

- **Continuous forcing approach**
  - Effect of the IB is imposed by the source (force) term in governing equations
  - Continuous forcing function is spread over a band of cells near IB
  - Independent of the spatial discretisation procedure
  - Smeared boundary description leads to accuracy and stability problems
  - Requires solution of governing equations inside the IB
- **Discrete forcing approach**
  - **Indirect imposition of BC**
    - Forcing term is introduced into discretised equation
    - Forcing function still spread over the band of cells
  - **Direct imposition of BC**
    - Modification of discretised equations near the IB to directly impose the BC on cells that touch IB.
    - Sharpness of the IB is preserved
    - Best accuracy and highest Reynolds number flows (without modification)

IBM In OpenFOAM: Selected Approach

Implementation of IBM In OpenFOAM

- Discrete forcing approach with direct imposition of boundary conditions
- Basic principle: Value of dependent variable in the IB cell centres is calculated by interpolation using neighbouring cells values and boundary condition at the corresponding IB point

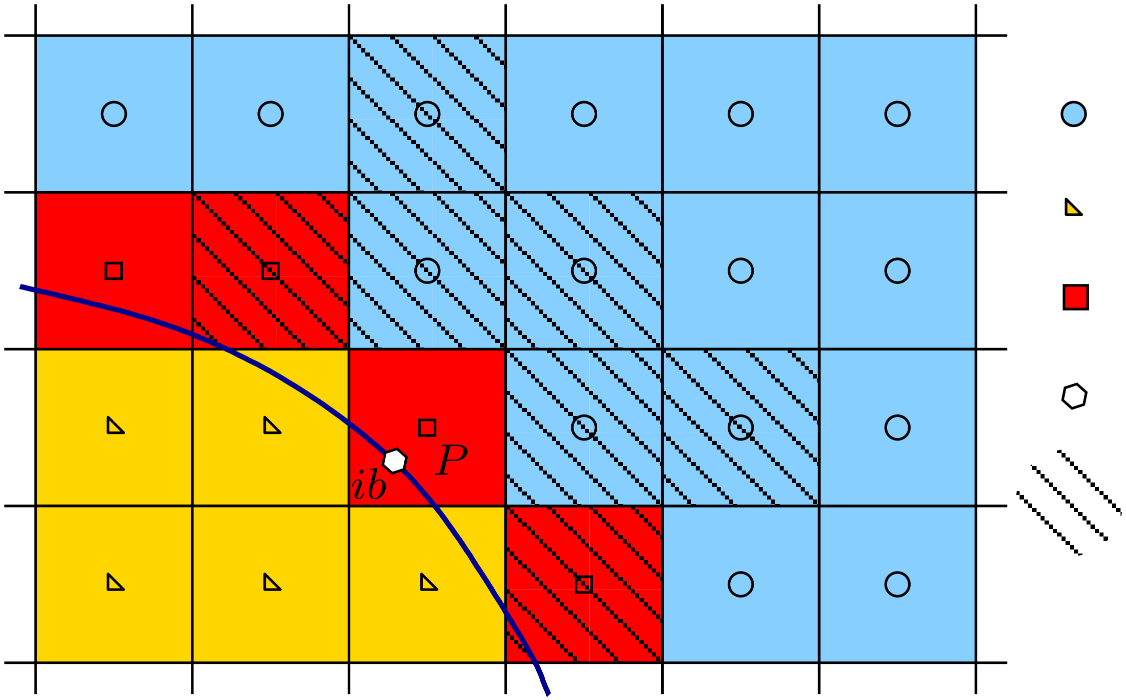


IB BC with Quadratic Interpolation

Dirichlet Immersed Boundary Condition

$$\phi_P = \phi_{ib} + C_0(x_P - x_{ib}) + C_1(y_P - y_{ib})$$
$$+ C_2(x_P - x_{ib})(y_P - y_{ib}) + C_3(x_P - x_{ib})^2 + C_4(y_P - y_{ib})^2$$

Unknown coefficients of quadratic polynomial determined using weighted least square method on extended stencil.

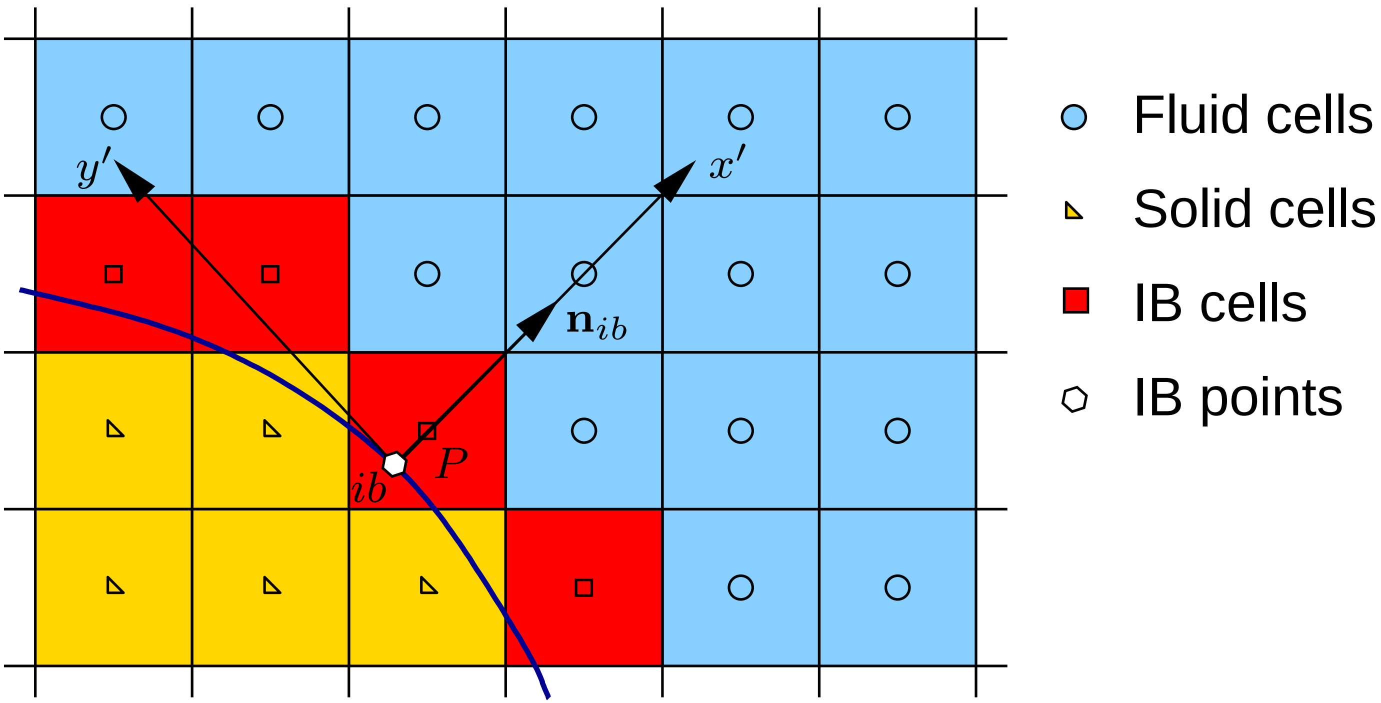


IB BC with Quadratic Interpolation

Neumann Immersed Boundary Condition Interpolation is performed in local coordinate

system $x'y'$ where x'-axis coincides with the normal to the immersed boundary at the point ib:

$$\phi_P = C_0 + [\mathbf{n}_{ib}\bullet(\nabla\phi)_{ib}]\, x'_P + C_1 y'_P + C_2 x'_P y'_P + C_3 (x'_P)^2 + C_4 (y'_P)^2$$

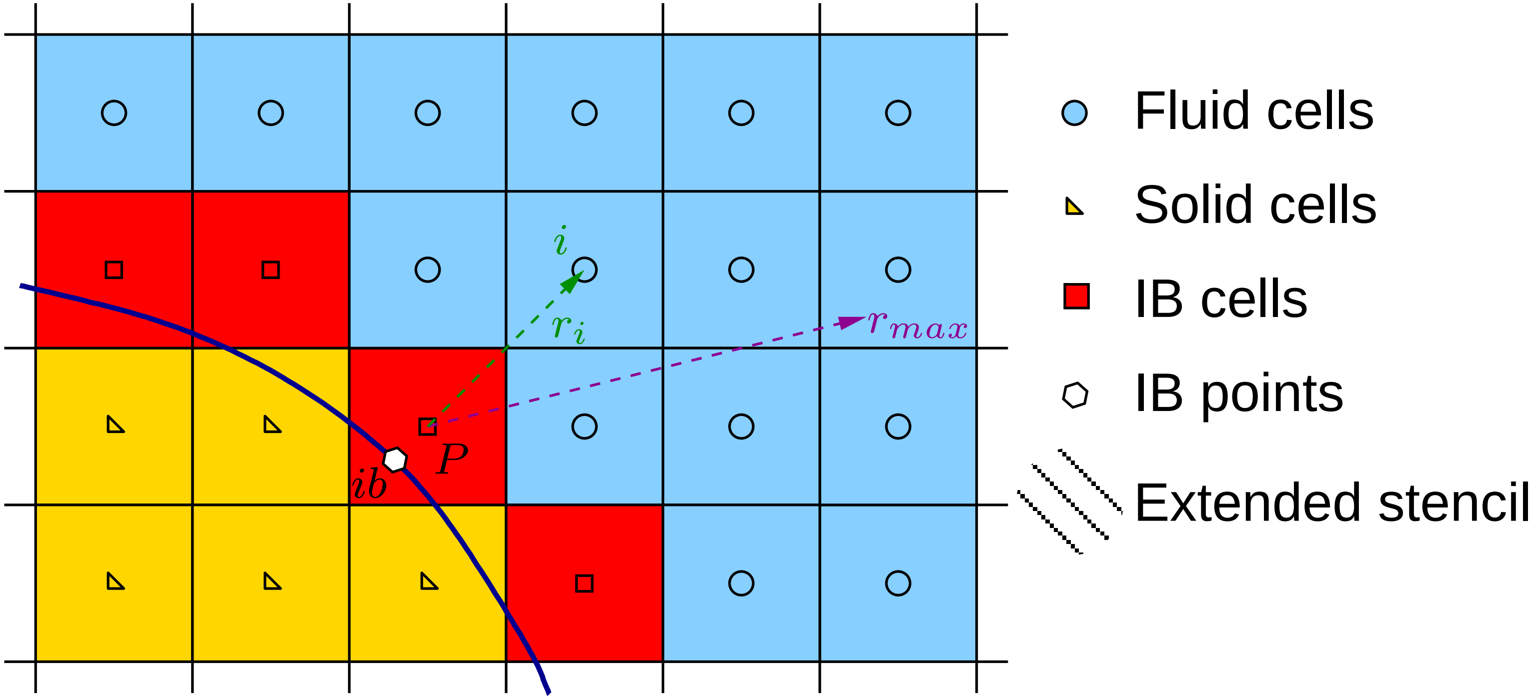


Least Squares Weighting Factors

Two options are considered:

- Inverse quadratic distance weight function: $w_i = \frac{1}{r_i^2}$
- Cosine weight function

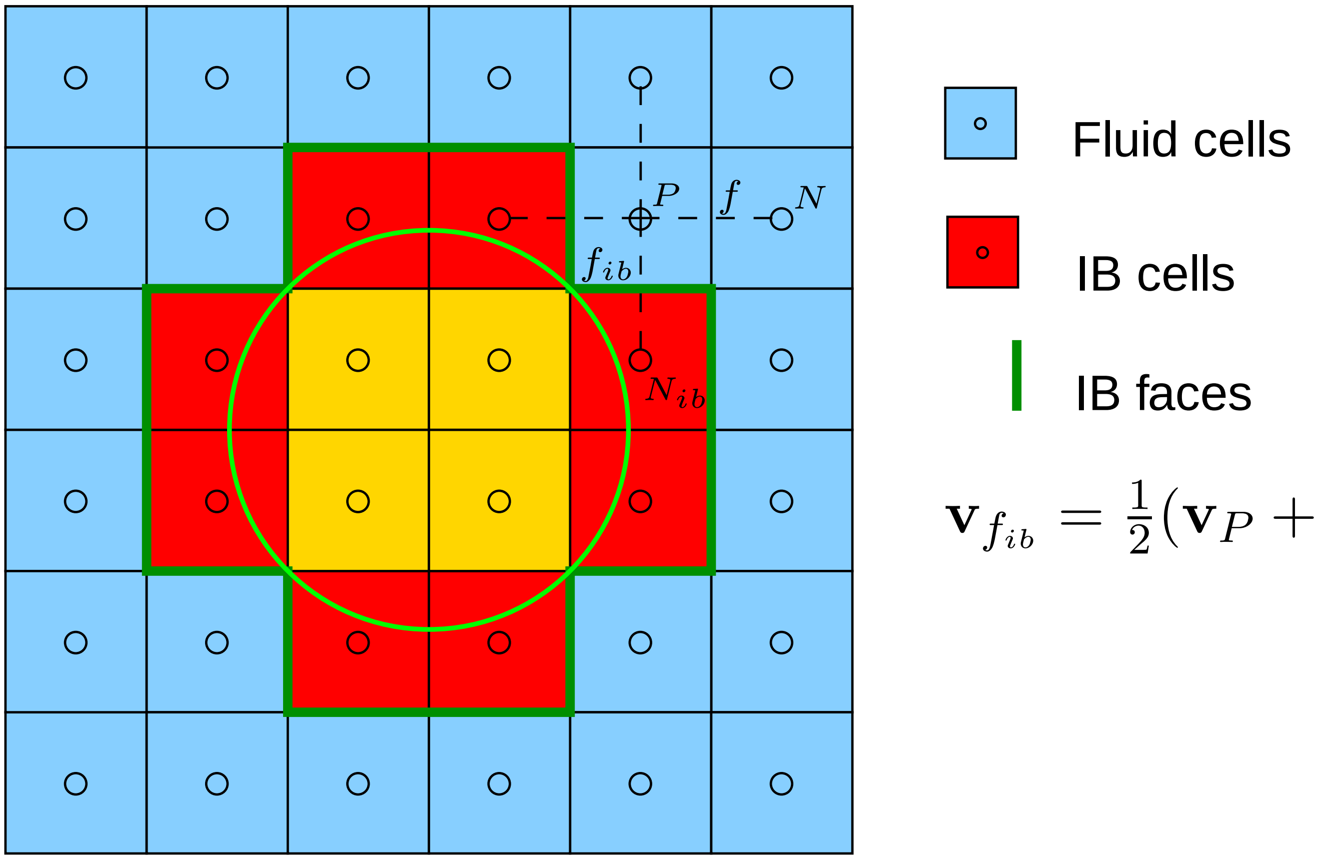
$$w_i = \frac{1}{2}\left[1 + \cos\left(\pi \frac{r_i}{S r_{max}}\right)\right]$$



Pressure Equation BC at IB

Pressure Equation for a Fluid Cell P Next to IB Cells

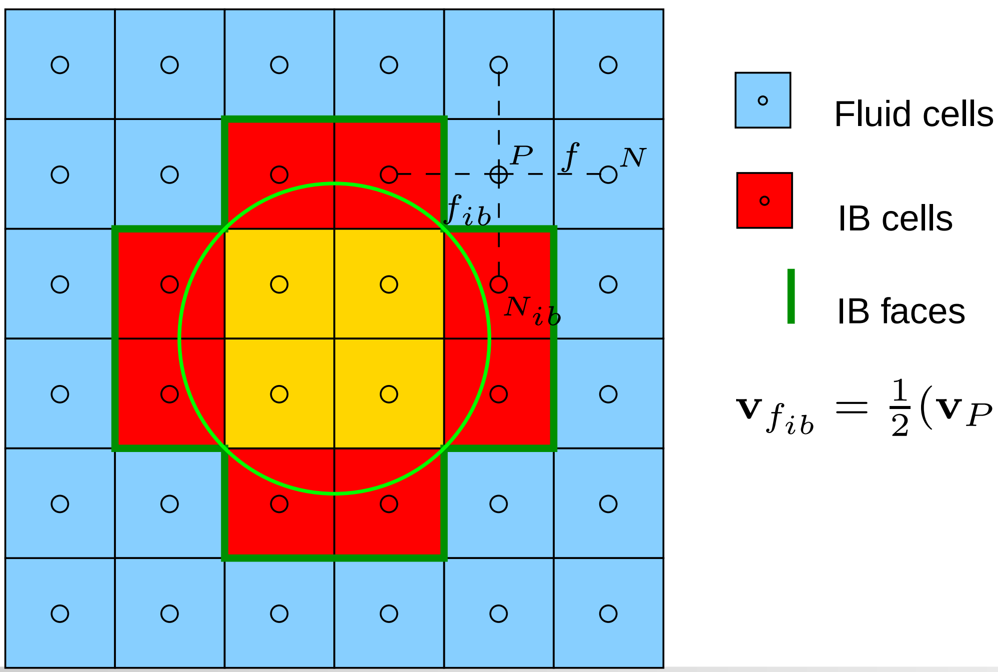
$$\sum_f \left(\frac{1}{a_P}\right)_f \mathbf{n}_f \bullet (\nabla p)_f S_f = \sum_f \mathbf{n}_f \bullet \left(\frac{\mathbf{H}_P}{a_P}\right)_f S_f + \sum_{f_{ib}} \mathbf{n}_{f_{ib}} \bullet \mathbf{v}_{f_{ib}} S_{f_{ib}}$$



$$\mathbf{v}_{f_{ib}} = \frac{1}{2}(\mathbf{v}_P + \mathbf{v}_{N_{ib}})$$

Pressure Equation BC at IB

- Boundary condition for pressure is not needed for solution of pressure equation since velocity at “IB faces” is treated as specified; . . . but pressure at IB faces ($p_{f_{ib}}$) is needed for the momentum equation!
- Pressure for IB faces and IB cells is calculated after solution of pressure equation by applying procedure for Neumann boundary condition imposition using quadratic interpolation
- Interpolated velocity at IB faces ($\mathbf{v}_{f_{ib}}$) must be scaled in such a way to impose zero net mass flux through the closed cage of IB faces around immersed boundary



$$\mathbf{v}_{f_{ib}} = \frac{1}{2}(\mathbf{v}_P + \mathbf{v}_{N_{ib}})$$

IBM: Implementation Details

Immersed Boundary Implementation in Three Classes:

- `class immersedBoundaryPolyPatch`: Basic mesh support, IB mesh
- `class immersedBoundaryFvPatch`: FV support, with derived Fv properties
  - Cell and face mask fields, live and dead cells indication
  - Calculation of intersection points, normals and distances
  - Calculation of interpolation matrices used in imposition of boundary conditions
  - Parallel communications framework and layout
- `class immersedBoundaryFvPatchField`: field support and evaluation of boundary conditions
  - Patch field evaluation for the IB patch
  - Calculation and interpolation of field data at mesh intersection, fixed value and fixed gradient conditions etc.
  - Handling of boundary updates

IBM: Implementation Details

```
class immersedBoundaryPolyPatch
:
    public polyPatch
{
    ...

    // Member Functions

        // Access

            //- Return immersed boundary surface mesh
            const triSurfaceMesh& ibMesh() const
            {
                return ibMesh_;
            }

            //- Return true if solving for flow inside the IB
            bool internalFlow() const
            {
                return internalFlow_;
            }

            //- Return triSurface search object
            const triSurfaceSearch& triSurfSearch() const;
};
```

IBM: Implementation Details

Including `immersedBoundaryPolyPatch` into boundary mesh of a `polyMesh` by modifying `constant/polyMesh/boundary` dictionary

```
6
(
    ibCylinder // constant/triSurface/ibCylinder.ftr
    {
        type                immersedBoundary;
        nFaces              0;
        startFace           3650;

        internalFlow        no;
    }
    in
    {
        type                patch;
        nFaces              25;
        startFace           3650;
    }

    ...
)
```

IBM: Implementation Details

```
class immersedBoundaryFvPatch
:
    public fvPatch
{
    // Private data

        //- Reference to processor patch
        const immersedBoundaryPolyPatch& ibPolyPatch_;

        //- Finite volume mesh reference
        const fvMesh& mesh_;

    // Member Functions

        //- Get fluid cells indicator, marking only live fluid cells
        const volScalarField& gamma() const;

        //- Return list of fluid cells next to immersed boundary (IB cells
        const labelList& ibCells() const;

        //- Return list of faces for which one neighbour is an IB cell
        //  and another neighbour is a live fluid cell (IB faces)
        const labelList& ibFaces() const;
};
```

IBM: Implementation Details

```
class immersedBoundaryFvPatch
:
    public fvPatch
{
        ...

        //- Return IB points
        const vectorField& ibPoints() const;

        //- Return IB cell extended stencil
        const labelListList& ibCellCells() const;

        //- Return dead cells
        const labelList& deadCells() const;

        //- Return live cells
        const labelList& liveCells() const;

        //- Get inverse Dirichlet interpolation matrix
        const PtrList<scalarRectangularMatrix>&
        invDirichletMatrices() const;

        //- Get inverse Neumann interpolation matrix
        const PtrList<scalarRectangularMatrix>&
        invNeumannMatrices() const;
};
```

IBM: Implementation Details

```
template<class Type>
class immersedBoundaryFvPatchField
:
    public fvPatchField<Type>
{
    // Private data

        //- Local reference cast into the processor patch
        const immersedBoundaryFvPatch& ibPatch_;

        //- Local reference to fvMesh
        const fvMesh& mesh_;

        //- Defining value field
        Field<Type> refValue_;

        //- Defining normal gradient field
        Field<Type> refGrad_;

        //- Does the boundary condition fix the value
        Switch fixesValue_;

        ...
};
```

Implementation details

```
template<class Type>
class immersedBoundaryFvPatchField
:
    public fvPatchField<Type>
{
    ...

    //- Impose Dirichlet BC at IB cells and return corrected cells values
    //  Calculate value and gradient on IB intersection points
    tmp<Field<Type> > imposeDirichletCondition() const;

    //- Impose Neumann BC at IB cells and return corrected cells values
    //  Calculate value and gradient on IB intersection points
    tmp<Field<Type> > imposeNeumannCondition() const;

    ...
};
```

IBM: Implementation Details

```
template<class Type>
class immersedBoundaryFvPatchField
:
    public fvPatchField<Type>
{
    ...

        //- Update the coefficients associated with the patch field
        void updateCoeffs();

        //- Evaluate the patch field
        virtual void evaluate
        (
            const Pstream::commsTypes commsType = Pstream::blocking
        );

        //- Manipulate matrix
        virtual void manipulateMatrix(fvMatrix<Type>& matrix);

    ...
};
```

IBM: Implementation Details

Including `immersedBoundaryFvPatchField` into boundary of a `volVectorField`

```
boundaryField
{
    ibCylinder
    {
        type immersedBoundary;
        refValue uniform (0 0 0);
        refGradient  uniform (0 0 0);
        fixesValue yes;

        setDeadCellValue   yes;
        deadCellValue      (0 0 0);

        value uniform (0 0 0);
    }
    movingWall
    {
        type parabolicVelocity;
        maxValue 0.375;
        n (1 0 0);
        y (1 0 0);
        value uniform (0.375 0 0);
    }
    ...
}
```

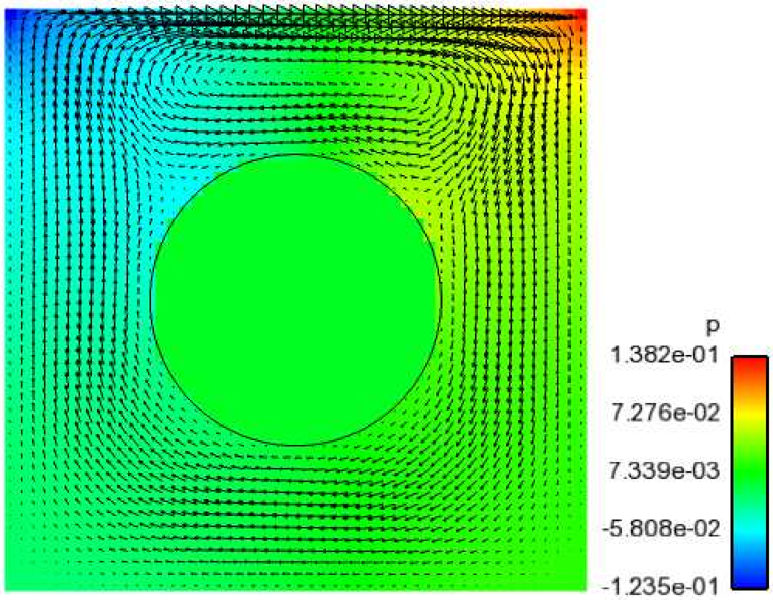
IBM: Implementation Details

Example of solver source code - `icoIbFoam`

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

    // Pressure-velocity corrector
    int oCorr = 0;
    do
    {
        fvVectorMatrix UEqn
        (
            fvm::ddt(U)
          + fvm::div(phi, U)
          - fvm::laplacian(nu, U)
        );

        UEqn.boundaryManipulate(U.boundaryField());
        solve(UEqn == -cellIbMask*fvc::grad(p));

        ...
    }

    ...
}
```

Tutorial: Cylinder in a Cavity

Tutorial Case: `cavity`



- Laminar flow around cylinder driven by motion of cavity top wall
- Case setup data
  - Cavity dimension: 1×1 m
  - Moving wall velocity: 0.375 m/s
  - Cylinder diameter: 0.5 m
  - Reynolds number: 37.5

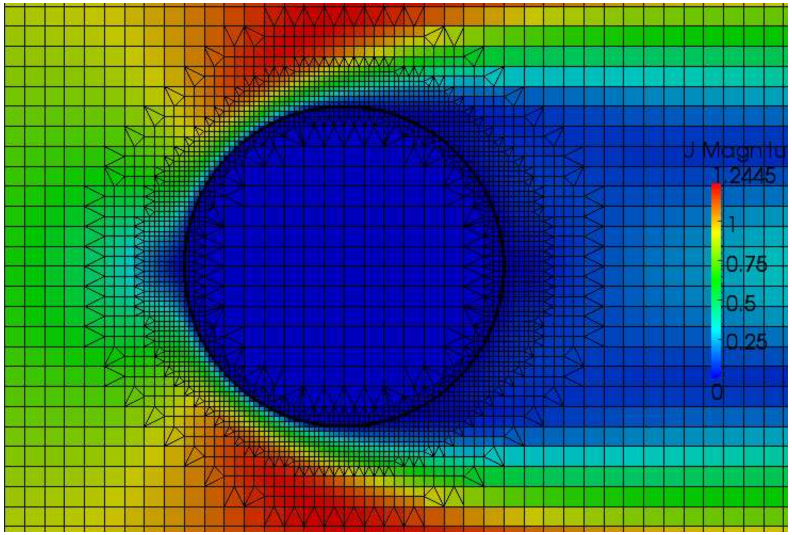
Tutorial: Cylinder in a Cavity

Steps of Case Setup and Simulation

- Define volume mesh in `constant/polyMesh/blockMeshDict` dictionary
- Create `polyMesh` using `blockMesh`
- Copy immersed boundary mesh (`ibCylinder.{ftr,stl}`) into `constant/triSurface` folder
- Include `immersedBoundaryPolyPatch` into `polyMesh` boundary
- Include `immersedBoundaryFvPatchField` into boundary field of pressure and velocity fields
- Set discretisation schemes in `./system/faSchemes` dictionary
- Set solution controls in `./system/faSolution` dictionary
- Set time step size `./system/controlDict`
- Run the case using `icoIbFoam`
- Post-process the case using `paraFoam`

Tutorial: Flow Around a Cylinder

Tutorial Case: `flowOverCylinder`



- Laminar flow around a circular cylinder in open space
- Case setup data
  - Open space dimensions: $90 \times 90 \text{ m}$
  - Inlet velocity: 1 m/s
  - Cylinder diameter: 1 m
  - Reynolds number: 100

Tutorial: Flow Around a Cylinder

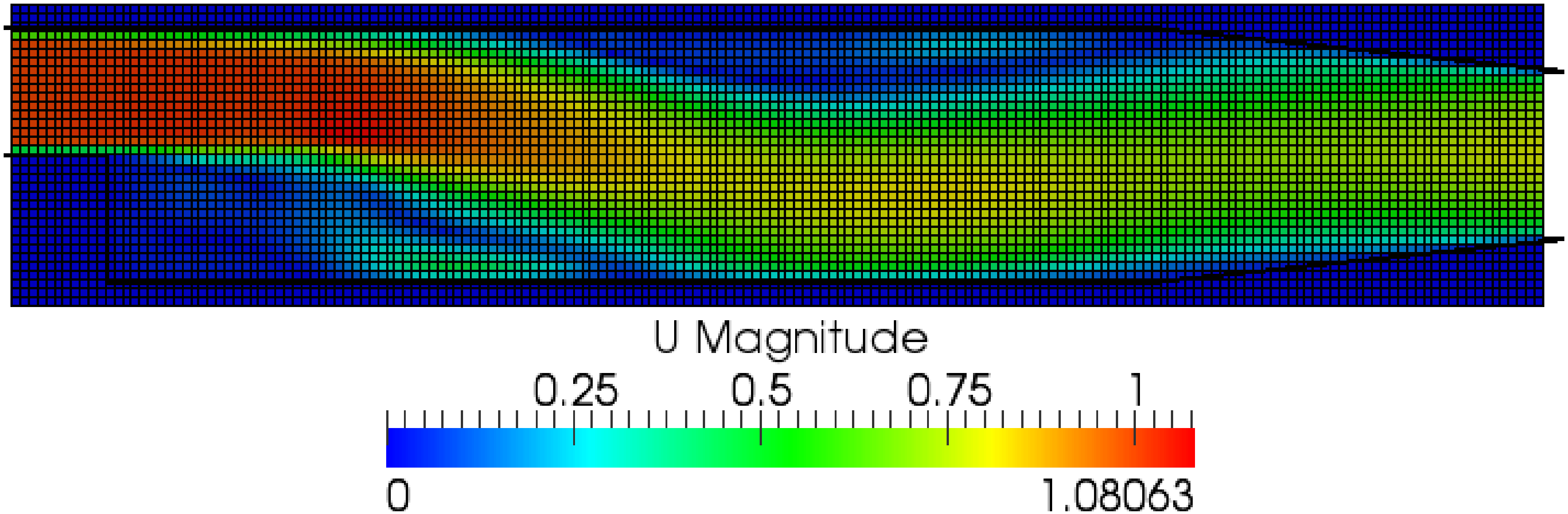
Steps of Case Setup and Simulation

- Define volume mesh in `constant/polyMesh/blockMeshDict` dictionary
- Create `polyMesh` using `blockMesh`
- Copy immersed boundary mesh (`ibCylinder.{ftr,stl}`) into `constant/triSurface` folder
- Include `immersedBoundaryPolyPatch` into `polyMesh` boundary
- Include `immersedBoundaryFvPatchField` into boundary field for the pressure and velocity fields
- Refine volume mesh using `refineCylinderMesh` application which must be compiled before
- Refine volume mesh using `refineImmersedBoundaryMesh` application
- Set discretisation schemes in `./system/faSchemes` dictionary
- Set solution controls in `./system/faSolution` dictionary
- Set time step size `./system/controlDict`
- Run the case using `icoIbFoam`
- Post-process the case using `paraFoam`

Tutorial: Backward-Facing Step by Pitz and Daily

Tutorial Case: `pitzDailyLaminar`

- Laminar flow over a backward-facing step by Pitz and Daily
- Case setup data:
  - Inlet velocity: 1 m/s
  - Reynolds number: 2500



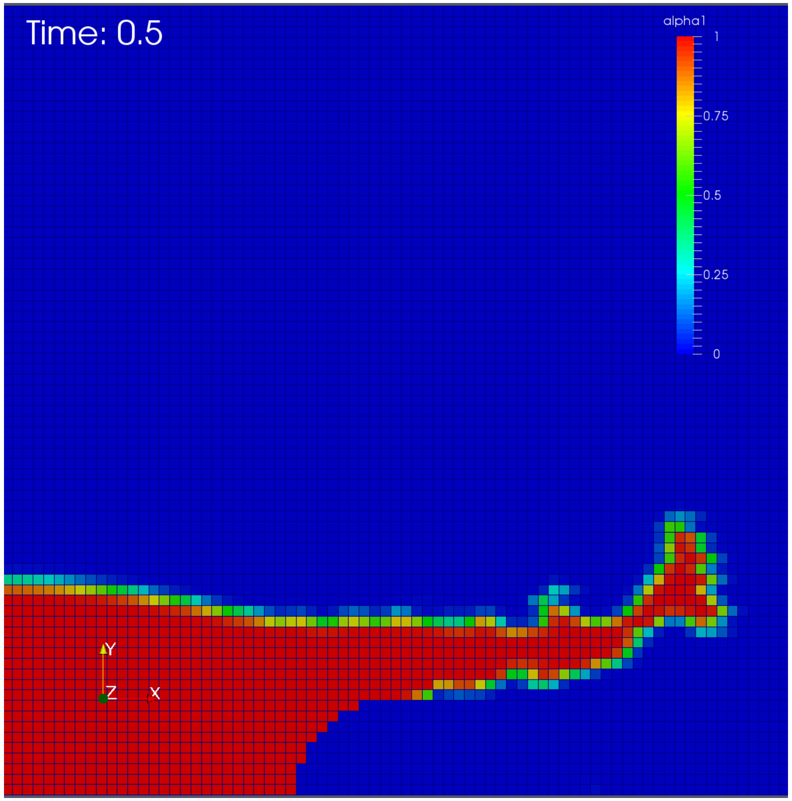
Tutorial: Backward-Facing Step by Pitz and Daily

Steps of Case Setup and Simulation

- Define volume mesh in `constant/polyMesh/blockMeshDict` dictionary
- Create `polyMesh` using `blockMesh`
- Copy immersed boundary mesh (`pitzDailyIB.{ftr,stl}`) into `constant/triSurface` folder
- Include `immersedBoundaryPolyPatch` into `polyMesh` boundary
- Include `immersedBoundaryFvPatchField` into boundary field of pressure and velocity fields
- Set discretisation schemes in `./system/faSchemes` dictionary
- Set solution controls in `./system/faSolution` dictionary
- Set time step size `./system/controlDict`
- Run the case using `icoIbFoam`
- Post-process the case using `paraFoam`

Tutorial: VOF Dam Break Over a Bump

Tutorial Case: `damBreakWithCylinder`



- Dam break VOF interface capturing simulation with circular bump at the bottom boundary
- Cylinder represented by STL surface `ibCylinder.stl`
- VOF solver uses implicit volume fraction equation and p-U system with variable density/viscosity
- Case setup data:
  - Domain dimension: $2 \times 2\ \mathrm{m}$
  - Bump diameter: $0.5\ \mathrm{m}$
  - Water-air multi-phase system

Tutorial: VOF Dam Break Over a Bump

Steps of Case Setup and Simulation

- Define volume mesh in `constant/polyMesh/blockMeshDict` dictionary
- Create `polyMesh` using `blockMesh`
- Copy immersed boundary mesh (`ibCylinder.{ftr,stl}`) into `constant/triSurface` folder
- Include `immersedBoundaryPolyPatch` into `polyMesh` boundary
- Include `immersedBoundaryFvPatchField` into boundary field of pressure and velocity fields
- Set discretisation schemes in `./system/faSchemes` dictionary
- Set solution controls in `./system/faSolution` dictionary
- Set time step size `./system/controlDict`
- Run the case using `interIbFoam`
- Post-process the case using `paraFoam`

Immersed Wall Functions

Dirichlet Condition - Implications

- A functional form in the Dirichlet condition specifies that the near-wall profile of a variable will be approximately quadratic
- This is appropriate for most cases and consistent with second-order discretisation: feed-back from functional fit adjusts local variable distribution
- For velocity in high-Re flows, quadratic fit is inappropriate: modification is required
- Equivalent modification appears in body-fitted meshes: **wall functions**
- Other implementations of IB wall functions are reported in literature, but rely on Cartesian background mesh
- New polyhedral implementation will be derived, based on the equivalence with the body-fitted wall functions

Standard Wall Functions

Standard Wall Functions on a Body-Fitted Mesh

- Wall functions modify the wall drag and turbulence variables, eg. for $k - epsilon$ model
  1. Collect k and near-wall distance y for near-wall cell
  2. Calculate y^* based on laminar viscosity ν_l at the wall

$$y^* = \frac{C_\mu^{0.25} \sqrt{k}\, y}{\nu_l}$$

  3. If y^* indicates log-law region, calculate turbulence generation and dissipation and account for wall shear by modifying viscosity in the near-wall cell

$$G = \frac{\nu_{eff}\, \mathbf{n} \bullet (\nabla \mathbf{u})_w}{C_\mu^{0.25}\, \kappa\, y}$$

$$\epsilon = \frac{C_\mu^{0.75}\, k^{1.5}}{\kappa\, y}$$

$$\nu_w = C_\mu^{0.25} \frac{k\, y}{\nu_l} \rightarrow \tau_w = \nu_w\, \mathbf{n} \bullet (\nabla \mathbf{u})_w$$

Standard Wall Functions

Standard Wall Functions on a Body-Fitted Mesh: Analysis

- In the near-wall cell, $\mathbf{u}$ and k are calculated. y^* is a function of k and $\mathbf{u}$ responds to the change in y^* to match the log-law profile
- Introduction of ν_w is a stable implicit mechanism to add momentum sink: responds to near-wall velocity gradient without division
- It is crucial to allow k to respond to the velocity gradient (via G) and vice-versa (via τ_w)

Immersed Boundary Wall Function: Issues

- Velocity solution in near wall cell must be decomposed into the normal and tangential component: wall functions act on tangential component only
- The near-wall point is **fitted for all variables**: implementing wall functions on the near-wall IB point will not work
- Data for active k and $\mathbf{n}\bullet(\nabla\mathbf{u})_w$ must be sampled from “live” flow cells

Immersed Boundary Wall Function

Immersed Boundary Wall Function: Algorithm

1. For each immersed boundary point, introduce the “sampling point”, 150% further away from the wall
2. At the sampling point, perform a least-square fit of fields through the interpolation stencil **excluding other immersed boundary point**
3. Based on least-square fit, evaluate near-wall tangential velocity, turbulence kinetic energy and laminar viscosity
4. Calculate y^* based on the sampling point near-wall distance and k
5. If y^* indicates log-law region for the sampling point, a log-law fit can be established to the IB point, otherwise, U will be fitted quadratically, $\nu_{eff} = \nu_l$ and G and ϵ are set to zero
6. Since all parameters of the least square fit are known, log-law fit for the IB point can be established:
   - Modify G, ϵ and ν_{eff} in the IB point (they are not used in actual immersed boundary wall function calculation, but only as a post-processing result)
   - Log-law fit the tangential velocity; wall-normal velocity is fitted quadratically, as in low-Re flows
   - Fitted log-law velocity appears in force balance for active cells and modified near-wall velocity field.

Immersed Boundary Wall Function

Immersed Boundary Wall Function: Consequences

- Log-law fit correctly captures near-wall velocity profile: drag is identical to body-fitted meshes
- Effective near-wall distance y used with the Immersed Boundary method is 150% of the distance to the first active cell centre
- Increase in effective near-wall y can be counteracted by refining the background mesh next to the IB boundary: `refineImmersedBoundaryMesh` utility
- By necessity, smoothness of y and y^* adjacent to the IB patch is lower than in body-fitted meshes
- Since the k transport equation is not solved in the IB cell, value of k follows from local equilibrium

Implementation

Implementation of the Immersed Boundary Wall Function

- Basic IB wall function class, `immersedBoundaryWallFunctionFvPatchField`
  - Class storing point-based IB data, with variation in the wall value (`wallValue`) and wall mask (`wallMask`) fields: point-wise switching in behaviour of the IB patch field
  - Additional function `ibSamplingPointValue`, extracting the data at the "shifted" sampling point in the wall-normal direction from the live stencil point data
  - `setIbCellValues`: function imposing IB value onto the internal field, based on wall values and mask
- Velocity IB wall function class, `immersedBoundaryVelocityWallFunctionFvPatchVectorField`, performing velocity decomposition into normal and tangential component only, and separately fitting each component
- ϵ or ω IB wall function class, performing IB wall function calculation and setting `wallValue` and `wallMask` for G, k, ϵ and $\mathbf{u}$
- IB wall function class for nu_t at IB is not required: wall drag accounted for directly in velocity fit

Summary

Implementation of Wall Functions on Immersed Boundary Patch

- Implementation uses flow data in the sampling point within th flow solution, located above each IB cell, in the wall-normal direction
- Log-law analysis is performed using sampled data. Based on this, the near-wall log-law profile is established
  - For turbulence variables, G and ϵ are calculated in the standard way
  - IB cell velocity vector is decomposed into the normal and tangential component
    - U_n is fitted using standard quadratic interpolation, consistent with the Dirichlet boundary condition
    - U_t is fitted based on the log-law profile between the solid wall and the sampling point
  - Since the k transport equation is not solved in the IB cell, k is calculated from the local equilibrium condition