051176 - Computational Techniques for Thermochemical Propulsion

Master of Science in Aeronautical Engineering

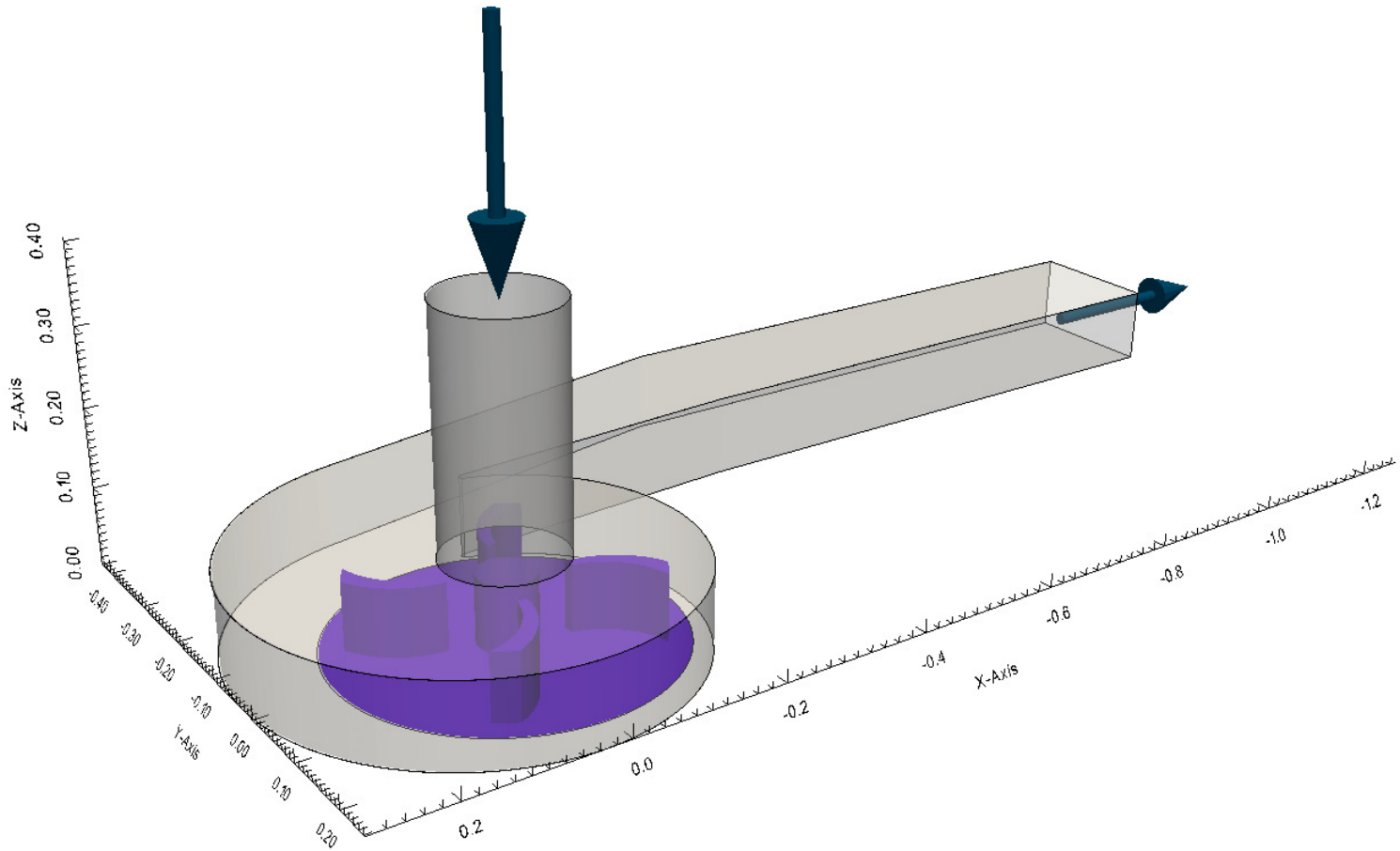
Advanced Pre-processing in OpenFOAM

Prof. **Federico Piscaglia**

Dept. of Aerospace Science and Technology (DAER)
Politecnico di Milano - Italy

federico.piscaglia@polimi.it

Case overwiew: centrifugal pump



Case Overview

Preprocessing steps

Mesh generation (snappyHexMesh) ✓
Mesh adjustments

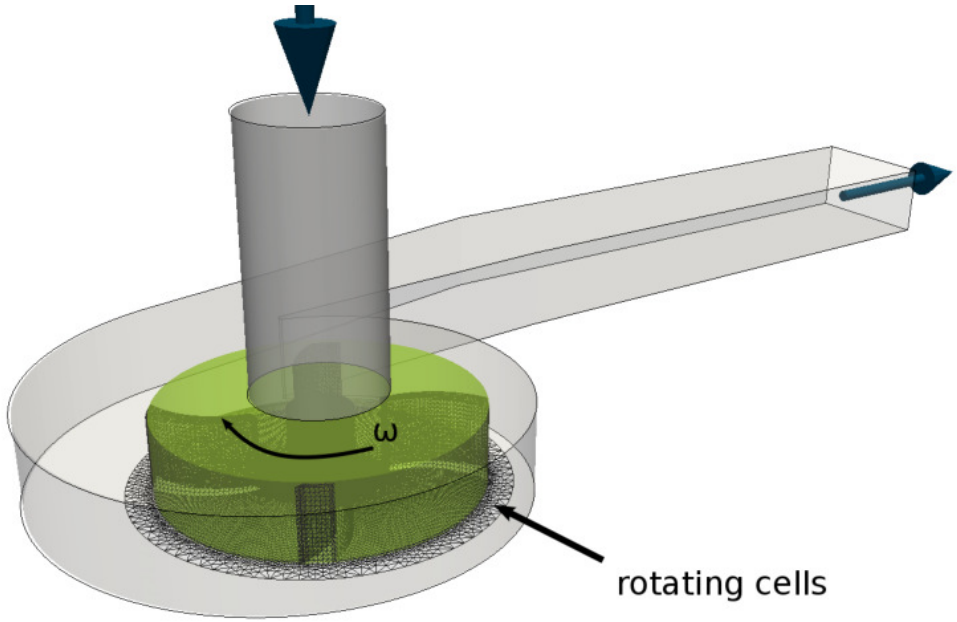
Models set up

Post-processing set up

Decompose

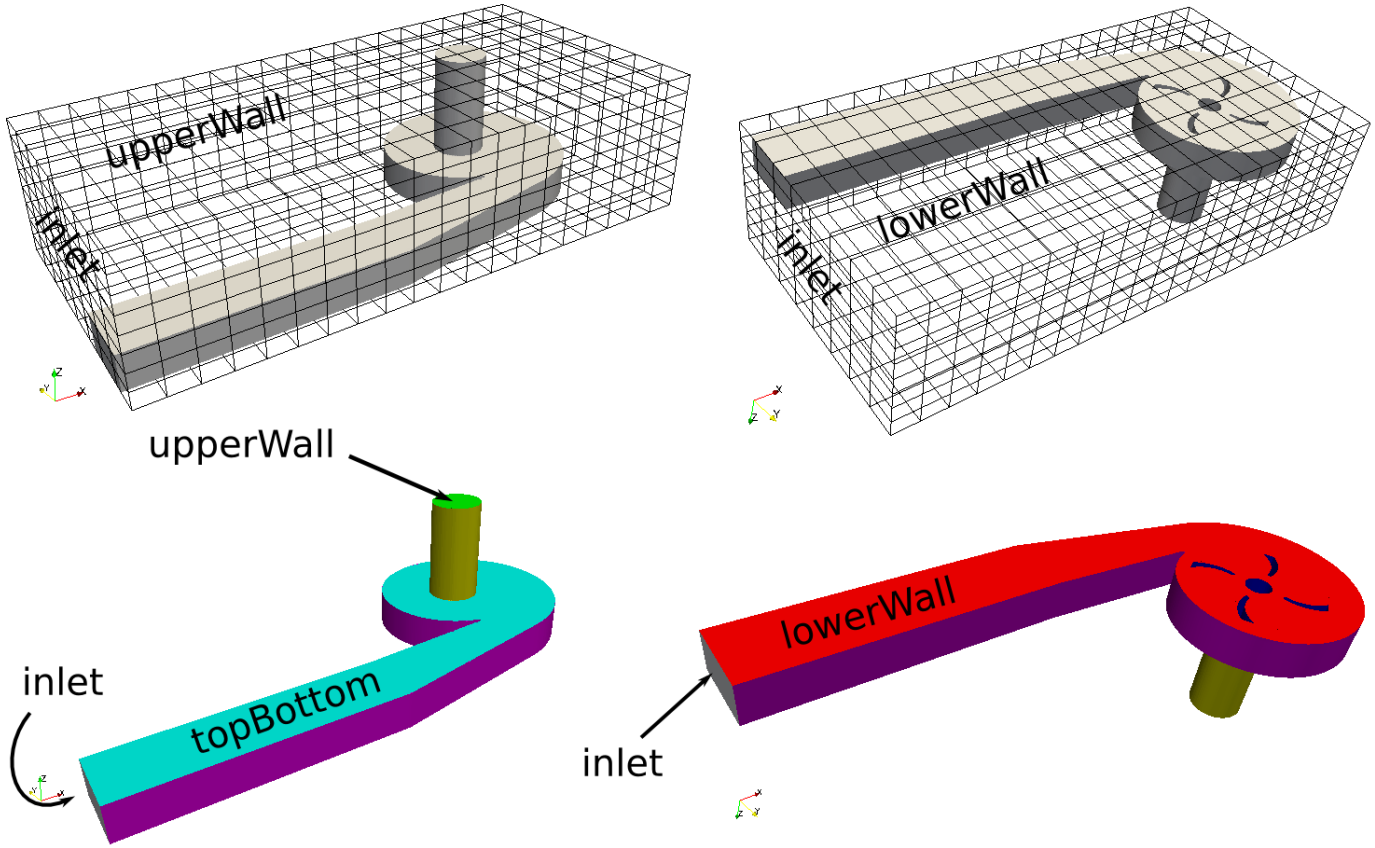
Physical modeling

- boundary conditions
- rotating cells
- turbulence modeling
- thermophysical properties
- ...



Mesh: patch names

Patch names of mesh generated by snappy are inherited from original blockMesh



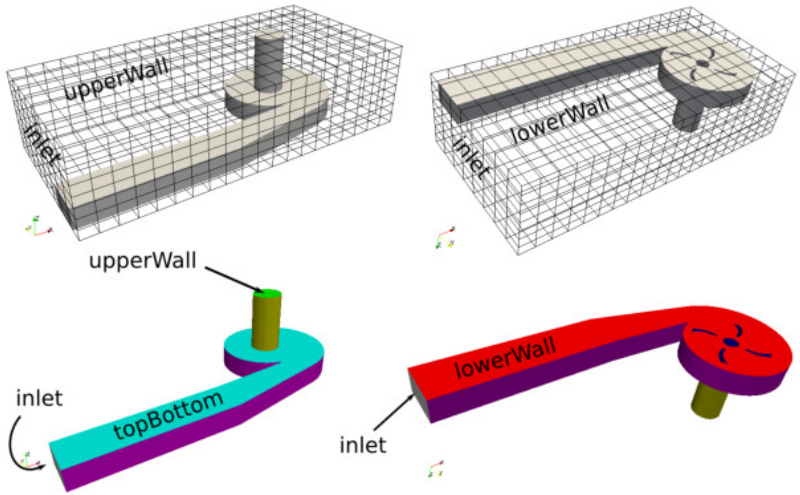
Mesh: patch names

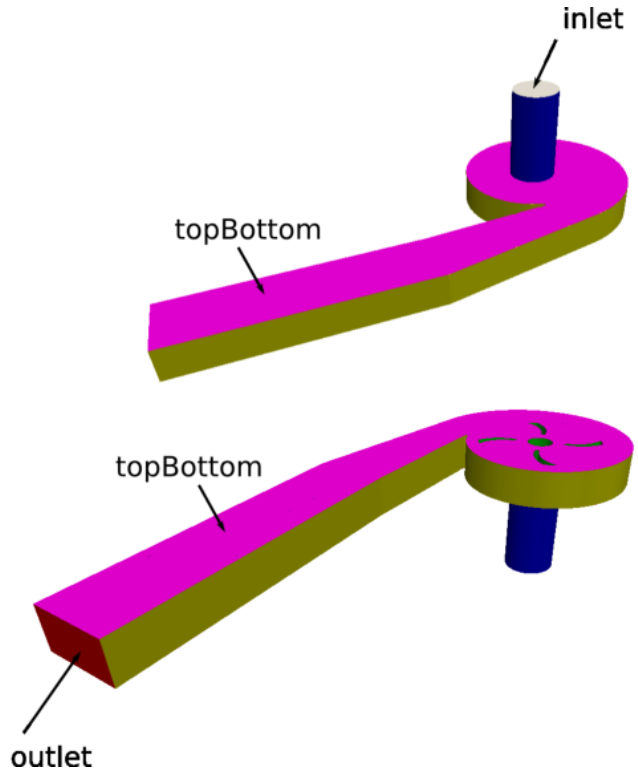
Operations to be done:

Rename “upperWall” to “inlet”

Rename “inlet” to “outlet”

Merge “lowerWall” into “topBottom”





Patch definition

- A patch is defined as a list of **consecutive** faces
- They are put **at the end** of the faces list:
  $f = \{f_0 \dots f_{ni}\} \cup \{f_{ni+1} \dots f_{nb}\}$
- Each patch has a **primitive** type:
  - `patch`
  - `wall`
  - `symmetry`, `symmetryPlane`
  - `empty`
  - `coupled` (AMI, cyclic, processor, regionCoupled)
  - `mapped`
- Internal patches are called "**baffles**"

```
leftWall
{
    type            wall;
    inGroups        1(wall);
    nFaces          50;
    startFace       4432;
}
```

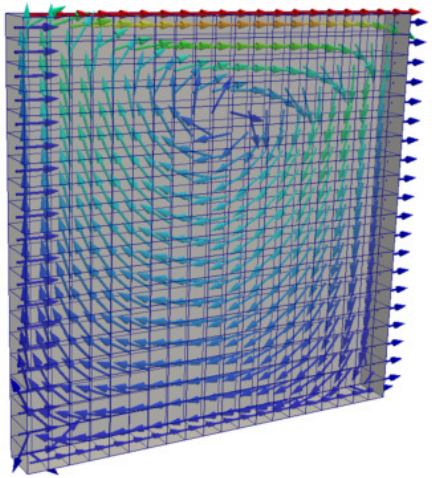
Patches are not the same of boundary conditions!

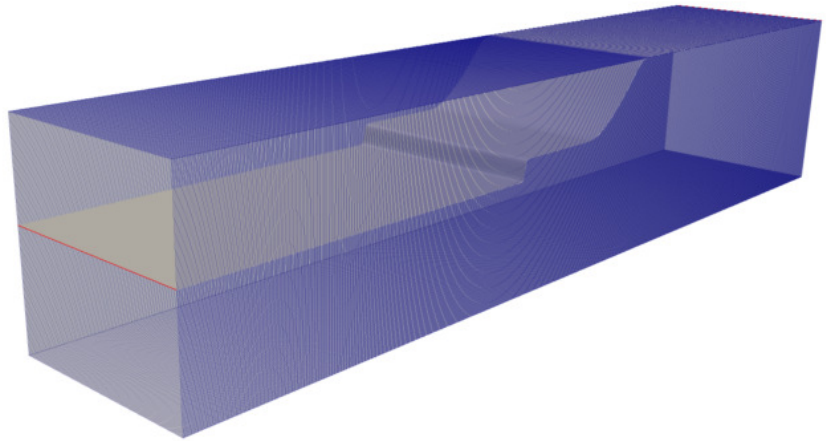
fvPatch: patch/wall

- "patch" patches are used to enforce generic BCs on permeable surfaces (e.g., inlet, outlet)
- "wall" patches are used to apply wall functions on solid surfaces.
- Each 'patch' or 'wall' boundary has also a *basic* type, that indicates the mathematical treatment of the variable:
  - **fixedValue**: Dirichlet BC
  - **zeroGradient**/**fixedGradient**: Neumann BC
  - **mixed**/**directionMixed**: Robin BC
  - **calculated**: not a real BC: values are interpolated from internal field
- from the basic BC type one can derive a **derived** type that implements a physical model. E.g.:
  - total pressure
  - wave transmissive
  - pressure jump
  - heat flux
  - etc...

fvPatch: empty

- "empty" patches are used to simulate 2D or 1D problems
- Any vector component normal to the empty patch is not solved for
- Mesh must be one-cell-thick along the patch-normal direction
- Edges must be **orthogonal** to the empty patch (if not, run `flattenMesh`)
- Additional information: none
- Fields BCs must be of type `empty`



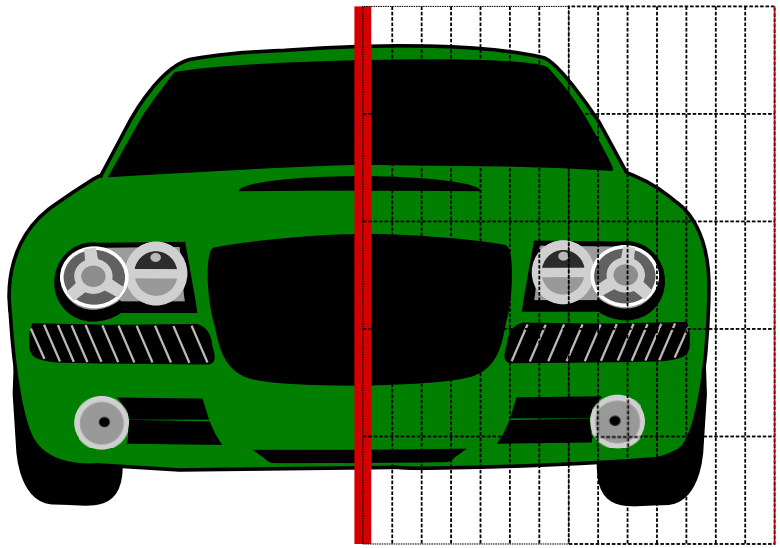


fvPatch empty


```
frontAndBack
{
  type empty;
  nFaces 100;
  startFace 200;
}
```

fvPatch: `symmetry`

- `symmetry` patches are used to enforce symmetry;
- any vector component normal to the symmetry patch is set to zero;
- scalar quantities are set as `zeroGradient`
- if the face is **perfectly planar** one can use `symmetryPlane`
- additional information: none;
- fields BCs must be of type `symmetry` (or symmetryPlane)



fvPatch: symmetry


```
midPlane
{
     type symmetry;
     // type symmetryPlane;
     nFaces 200;
     startFace 301;
}
```

Coupled patches

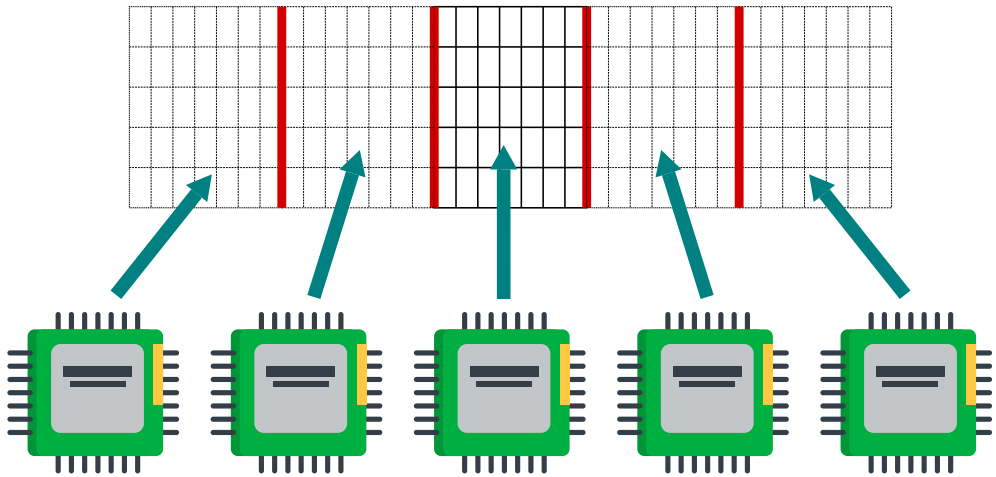
- **Coupled** patches are used to connect:
  - Opposite sides of the domain (cyclic)
  - Different subdomains of decomposed mesh (processor)
  - Different mesh regions in the same domain (AMI/ACMI)
  - Different domain regions (regionCoupled)
  - Different zones of the mesh simulating an 'active' wall (e.g. pressure jump)
- For each coupled patch, there must be **another patch to couple with**, called `neighbour`
- Additional information about the information transfer may be needed (e.g. face-to-face mapping, geometrical transform, etc.)
- Field values and matrix coefficients are exchanged through the patch and transformed if necessary

fvPatch: processor

- `processor` patches are used to exchange information between contiguous subdomain in parallel runs
- They are automatically generated by the decomposition
- Must have **perfect point match** between corresponding sides
- Specific information: face-to-face correspondence (automatically generated and written in file `faceProcAddressing`):

$$\mathcal{M}(f, f') = \{f_1 \ldots\} \mapsto \{f'_1 \ldots f'_n\}$$

- Fields BCs must be of type `processor`



fvPatch processor0

In processor0/constant/polyMesh/boundary:

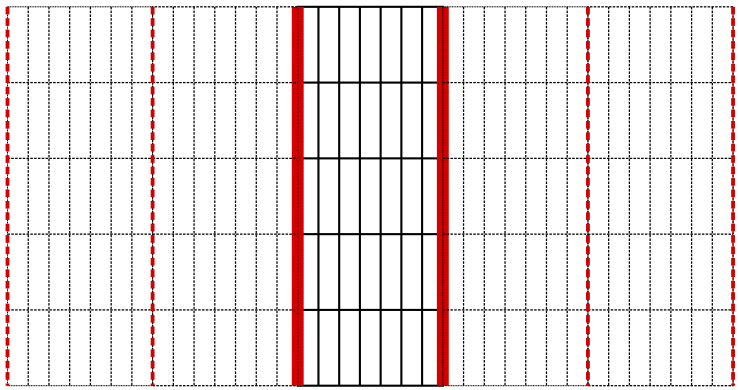
```
    procBoundary0to1
    {
        type            processor;
        inGroups        1(processor);
        nFaces          26;
        startFace       2299;
        matchTolerance  0.0001;
        transform       unknown;
        myProcNo        0;
        neighbProcNo    1;
    }
```

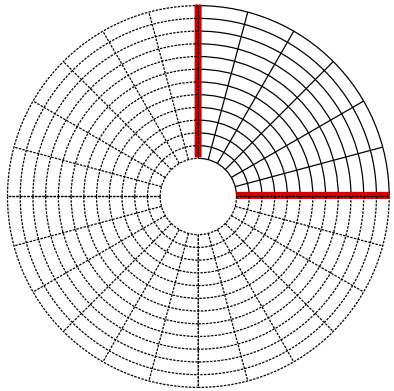
In processor1/constant/polyMesh/boundary:

```
    procBoundary1to0
    {
        type            processor;
        inGroups        1(processor);
        nFaces          26;
        startFace       2244;
        matchTolerance  0.0001;
        transform       unknown;
        myProcNo        1;
        neighbProcNo    0;
    }
```

fvPatch: cyclic

- `cyclic` patches are used to enforce periodicity
- They are generated during the case setup
- Must have **perfect point match** between corresponding sides
- Must have **the same face order**:
  $\mathcal{M}(f, f') = \mathbf{1}$
- Specific information:
  - Type of transform (translational/rotational)
  - If translational: displacement vector
  - If rotational: center and axis
- Fields BCs must be of type `cyclic`





fvPatch: cyclic

Translational cyclic:

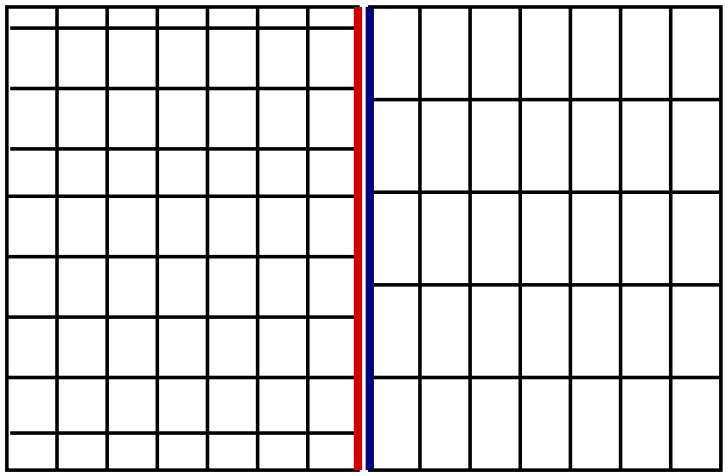
```
    cyclic_half0
    {
        type            cyclic;
        inGroups        1(cyclic);
        nFaces          1600;
        startFace       101720;
        matchTolerance  0.0001;
        transform       translationa
        neighbourPatch  cyclic_half1
        separationVector (1 0 0);
    }
    cyclic_half1
    {
        type            cyclic;
        inGroups        1(cyclic);
        nFaces          1600;
        startFace       103320;
        matchTolerance  0.0001;
        transform       translationa
        neighbourPatch  cyclic_half0
        separationVector (-1 0 0);
    }
```

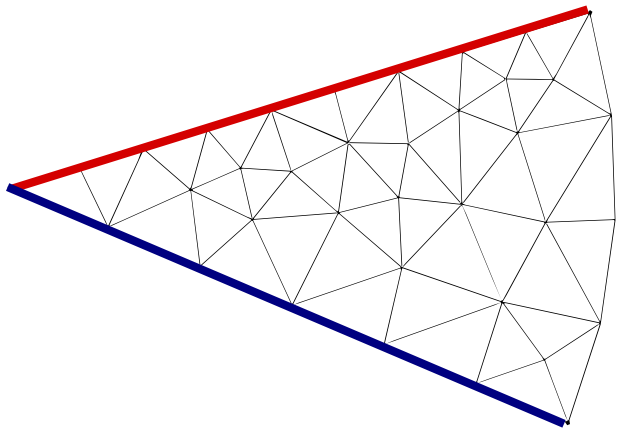
Rotational cyclic:

```
    cyclic_half0
    {
        type            cyclic;
        inGroups        1(cyclic);
        nFaces          1600;
        startFace       101720;
        matchTolerance  0.0001;
        transform       rotational;
        neighbourPatch  cyclic_half1
        rotationAxis    (0 0 1);
        rotationCentre  (0 0 0);
    }
    cyclic_half1
    {
        type            cyclic;
        inGroups        1(cyclic);
        nFaces          1600;
        startFace       103320;
        matchTolerance  0.0001;
        transform       rotational;
        neighbourPatch  cyclic_half0
        rotationAxis    (0 0 1);
        rotationCentre  (0 0 0);
    }
```

fvPatch: cyclicAMI

- Like cyclic but mesh **does not need to be conformal**
- They are generated during the case setup
- Point and face correspondence automatically calculated (transformation can be optionally specified)
- Interpolation weights are stored internally
- Fields BCs must be of type `cyclicAMI`
- AMI requires **full surface overlap**



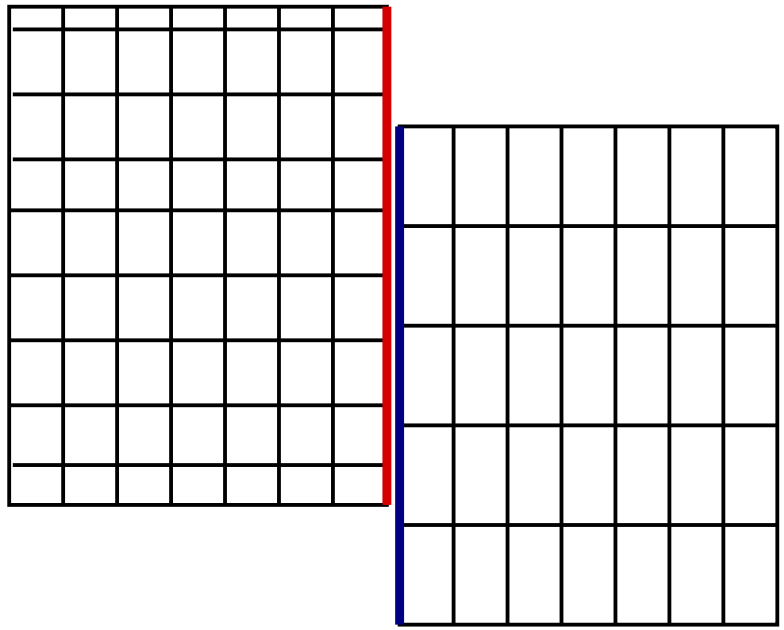


fvPatch: cyclicAMI


```
    AMI1
    {
        type            cyclicAMI;
        inGroups        1(cyclicAMI);
        nFaces          96;
        startFace       6240;
        matchTolerance  0.0001;
        transform       noOrdering;
        neighbourPatch  AMI2;
    }
    AMI2
    {
        type            cyclicAMI;
        inGroups        1(cyclicAMI);
        nFaces          96;
        startFace       6336;
        matchTolerance  0.0001;
        transform       noOrdering;
        neighbourPatch  AMI1;
    }
```

fvPatch: cyclicACMI

- Variant of AMI that can handle **partially overlapping surfaces**
- `dummy` wall patches must be inserted
  $\rightarrow$ a single `ACMI` interface requires 4 patches



fvPatch: cyclicACMI

side 1:

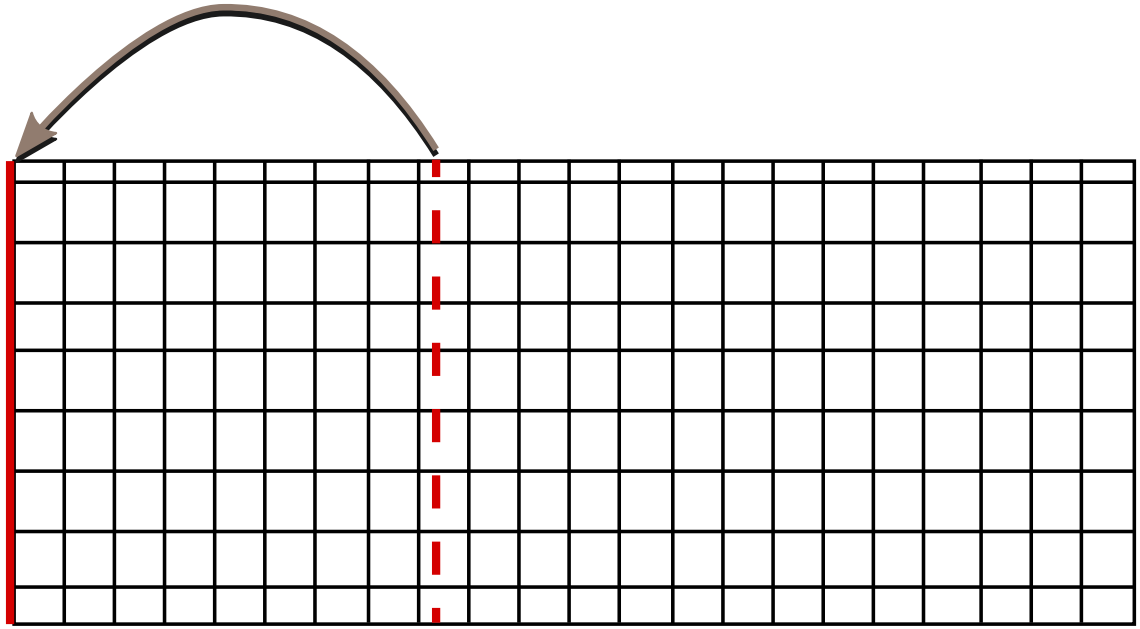
```
    ACMI1_couple
    {
        type            cyclicACMI;
        nFaces          40;
        startFace       43680;
        matchTolerance  0.0001;
        transform       noOrdering;
        neighbourPatch  ACMI2_couple
        nonOverlapPatch ACMI1_blocka
    }
    ACMI1_blockage
    {
        type            wall;
        nFaces          40;
        startFace       43720;
    }
```

side 2:

```
    ACMI2_couple
    {
        type            cyclicACMI;
        nFaces          96;
        startFace       43760;
        matchTolerance  0.0001;
        transform       noOrdering;
        neighbourPatch  ACMI1_couple
        nonOverlapPatch ACMI2_blocka
    }
    ACMI2_blockage
    {
        type            wall;
        nFaces          96;
        startFace       43856;
    }
```

fvPatch: mapped

- Reads values from internal field and set them on the boundary
- Enforces a one-way coupling
- Geometric match is not needed
- Additional information: plane offset
- Fields can (but do not have to) be of type mapped
- Optionally set average



fvPatch:mapped


```
    inlet
    {
        type            mappedPatch;
        nFaces          30;
        startFace       27238;
        sampleMode      nearestCell;
        sampleRegion    region0;
        samplePatch     none;
        offsetMode      uniform;
        offset          (0.0495 0 0);
    }
```

```
    inlet
    {
        type                mapped;
        value               uniform (10 0 0);
        interpolationScheme cell;
        setAverage          true;
        average             (10 0 0);
    }
```

Utility: createPatch

Modifies existing patch definition:

- Operates only on external faces (**not for creating internal baffles**)
- Select faces on the basis of
  - existing patches
  - faceSet
- Change patch type:
  - from `wall` to `patch`
  - from `wall` to `coupled` (AMI, periodic, . . .)
- Reorder faces if necessary (e.g. for cyclics)

```
// file: system/createPatchDict
{
  name cyc_half0; // new name

  patchInfo
  {
    type cyclic; // new type

    // other patch parameters
    neighbourPatch cyc_half1;
    transform rotational;
    rotationAxis (1 0 0);
    rotationCentre (0 0 0);
  }

  // How to construct:
  // either from 'patches' or 'set'
  constructFrom patches;

  patches (periodic1);
  set f0;
}
```

syntax: `createPatch [-overwrite] [-dict altDict]`

createPatch: outlet & inlet

Select the patch `inlet` (type: wall)
Change name to `outlet`

Change type to `patch`

no other parameters

```
{
  name outlet;
  patchInfo
  {
    type patch;
  }
  constructFrom patches;
  patches (inlet);
}
```

Select the patch `upperWall` (type: wall)
Change name to `inlet`

Change type to `patch`

no other parameters

```
{
  name inlet;
  patchInfo
  {
    type patch;
  }
  constructFrom patches;
  patches (upperWall);
}
```

createPatch: topBottom

Select all faces of faceSet “topBottomPatch” (*)
Set name to “topBottom”

Set type to “wall”

no other parameters

- faceSet “topBottomPatch” includes all faces from old “topBottom” and “lowerWall”.
- faceSet **must be generated before createPatch**

```
{
  name topBottom;

  patchInfo
  {
    type wall;
  }

  constructFrom set;

  set topBottomPatch;
}
```

Mesh sets & zones

- In OpenFOAM® it is possible to define point-, face- and cell- sets and zones.

- **sets:**
  - a "set" is a list of mesh entities stored in a separate file (`constant/polyMesh/sets/<set name>`)
  - An entity can belong to more than one set
  - Sets are not updated when mesh is changed
  - Used mainly for pre-processing
  - Generated by **topoSet** or **setSet** utilities

- **zones:**
  - a "zone" is a label assigned to individual cell entities (`constant/polyMesh/{cellZones,faceZones,pointZones}`)
  - An entity cannot belong to more than one zone
  - Zones are automatically updated when mesh is changed
  - Used mainly to set point-, face- or cell- properties in the solver (e.g. porous regions)
  - Created along with the mesh or converted from sets with **setsToZones**

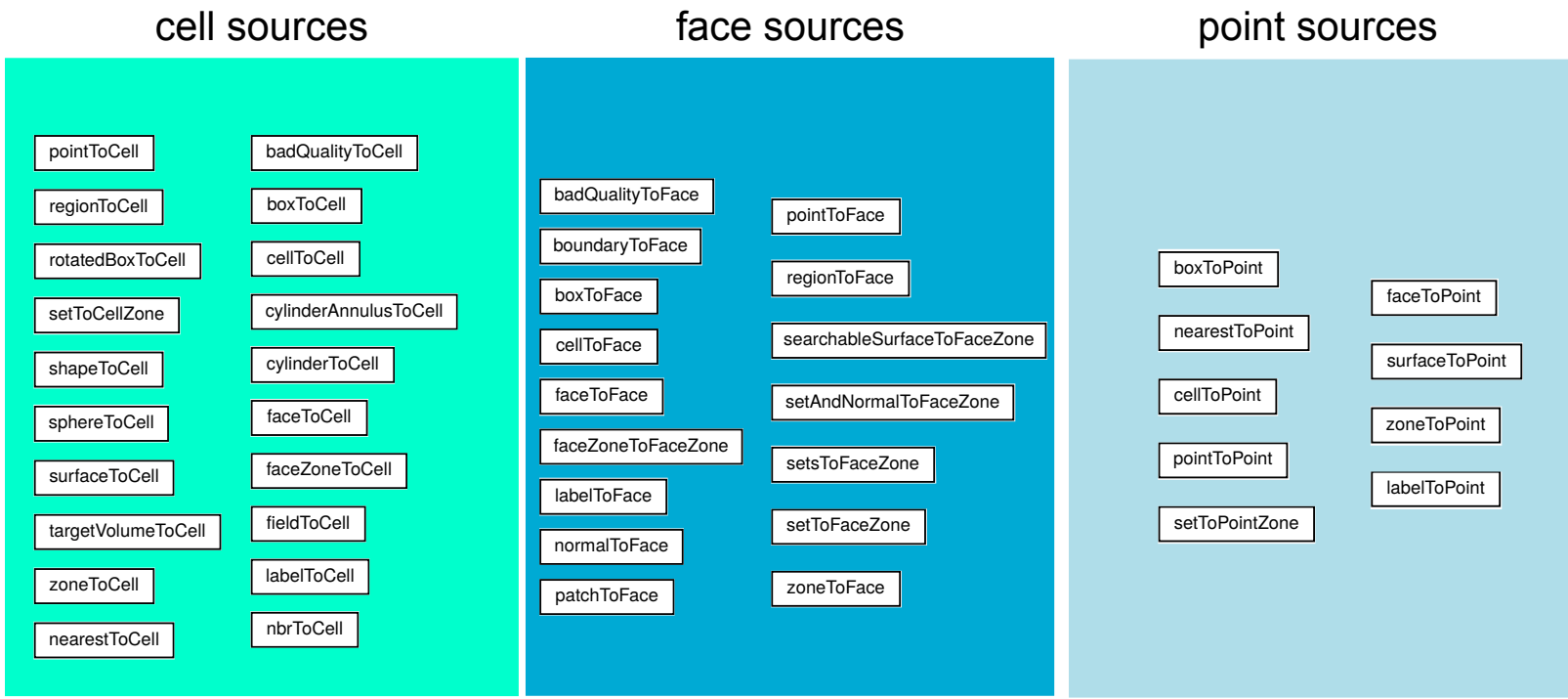
Utility: topoSet


```
// Load initial cellSet
{
  name    c0;
  type    cellSet;
  action  new;
  source labelToCell;
  sourceInfo
  {
    value (12 13 56);
  }
}
// Copy cellSet c0
{
  name    c1;
  type    cellSet;
  action  new;
  source  cellToCell;
  sourceInfo
  {
    set c0;
  }
}
```

- Type can be:
  - cellSet
  - faceSet
  - pointSet
- Action can require:
  - no Source:
    - clear (empty set)
    - invert (select all non-selected elements)
    - remove (remove set from list)
  - source:
    - new
    - add [elements to set]
    - delete [elements from set]
    - subset

set sources

All set sources derive from class `topoSetSource`



Look in `$FOAM_UTILITIES/mesh/manipulation/topoSet/topoSetDict` for some examples

utility: setSet

- **setSet** is used to interactively create/delete/modify sets
- It consists in an interactive shell one can use to run setSet commands
- Syntax:

```
readline> <setType> <setName> <action> <sourceType> <sourceParams>
```

- Example:

```
readline> cellSet c0 new labelToCell (12 13 56)
```

- Sets are written to a VTK file for immediate visualization
- Other commands:
  - list
  - time <ddd>
  - quit
  - help
- command history is saved in a hidden file **.setSet**
- Batch mode:

```
setSet -batch <file>
```

back to the tutorial...

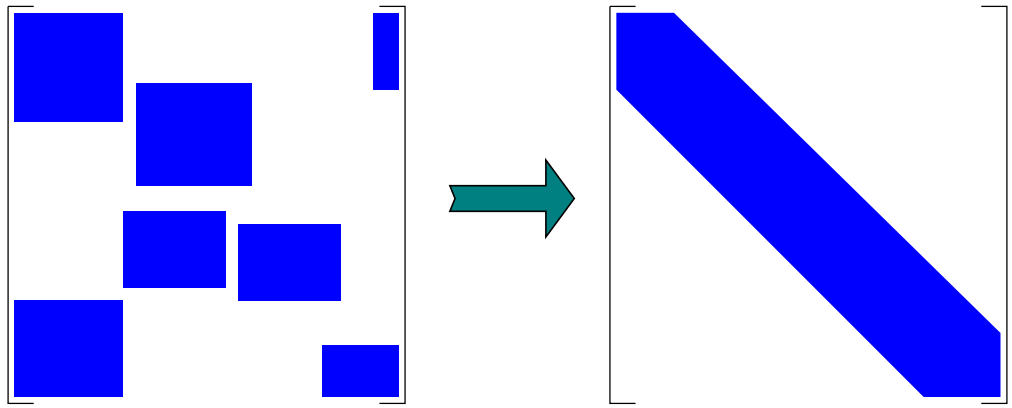
1. Create a new **faceSet** using all faces on patch "lowerWall"
2. Add all faces belonging to patch "topBottom"

```
{
    name    topBottomPatch;
    type    faceSet;
    action  new;
    source  patchToFace;
    sourceInfo
    {
        name lowerWall;
    }
}
{
    name topBottomPatch;
    type faceSet;
    action add;
    source patchToFace;
    sourceInfo
    {
        name topBottom;
    }
}
```

```
topoSet -dict system/topoSetDict.createPatch
```

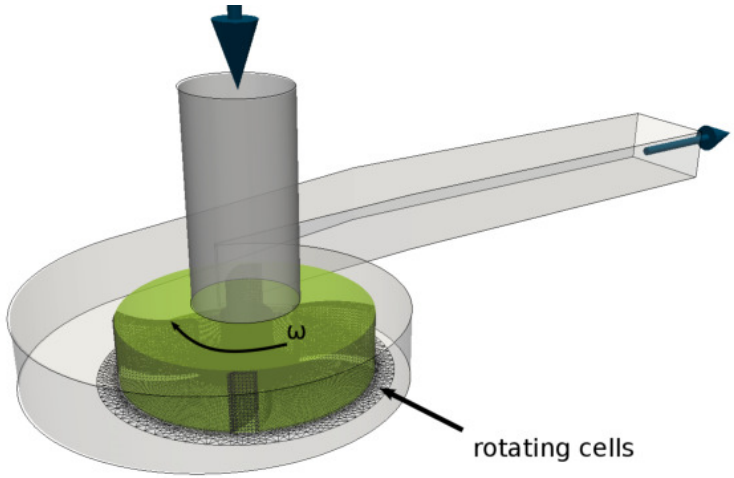
```
createPatch -overwrite
```

Renumbering the mesh



- For the numerical solver to easily converge (or converge at all), the system matrix has to be **diagonally dominant**
- Usually, real-world geometries produce strongly non-diagonal matrices
- The utility **renumberMesh** reorders cell entities to minimize the matrix sparseness

Case Overview



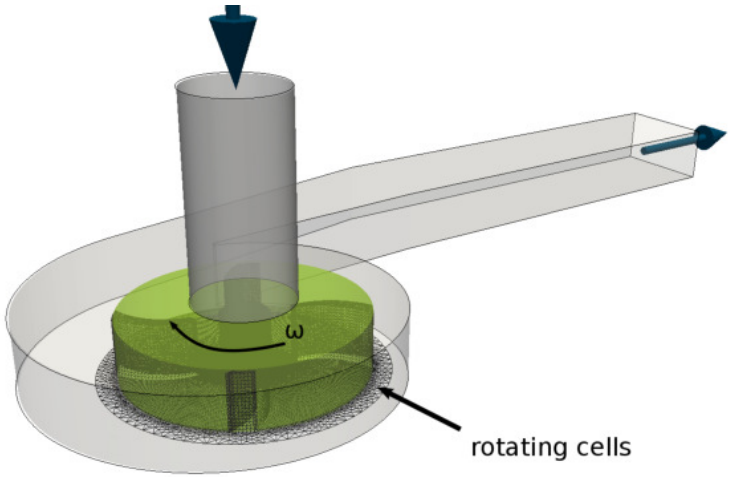
Preprocessing steps

1. Mesh generation (snappyHexMesh) ✓
2. Mesh adjustments ✓
3. Models set up
4. Post-processing set up
5. Decompose

Simulate moving cells

Dynamic mesh approach

- Cells are actually moving
- AMI patch between moving and fixed cells
- Physically very accurate
- Requires transient simulation



MRF (Moving Reference Frame)

- Cells are fixed, no interface patches
- Rotation is simulated by adding Coriolis force to momentum equation:

$$\boldsymbol{F} = -2\rho V\, \boldsymbol{\Omega} \times \boldsymbol{U}$$

- Steady state simulation (faster!)
- Can solve global quantities (H, Q) but not local effects (rotor/stator interaction)
- also known as “frozen rotor approach”;

→ We will use MRF for solving the centrifugal pump case.

MRF implementation

How is MRF inserted into the equations?

- Update patch fields:

$$\forall f \in \text{MRF zone}: \boldsymbol{S}_u = \boldsymbol{\Omega} \times \boldsymbol{U}_f$$

- In **momentum equation** an explicit source term is added:

$$\forall c \in \text{MRF zone}: \boldsymbol{S}_u = \boldsymbol{\Omega} \times \boldsymbol{U}$$

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

- In **pressure equation**, if $(\nabla p)_f$ is imposed at boundaries to account for cell-face flux φ_f, then $(\nabla p)_f$ is corrected by $(-\boldsymbol{\Omega} \times \boldsymbol{U}) \cdot \boldsymbol{S}_f$

MRF approach

- MRF are specified in file `constant/MRFProperties`
- "Rotating" cells are defined on the basis of a **cellZone**
- Parameters (right-hand-rule):
  - axis $\boldsymbol{z}$
  - angular velocity $|\omega|$,
  - origin (x_0, y_0, z_0)
- There can be any number of MRF sources in a case, as long as they do not overlap

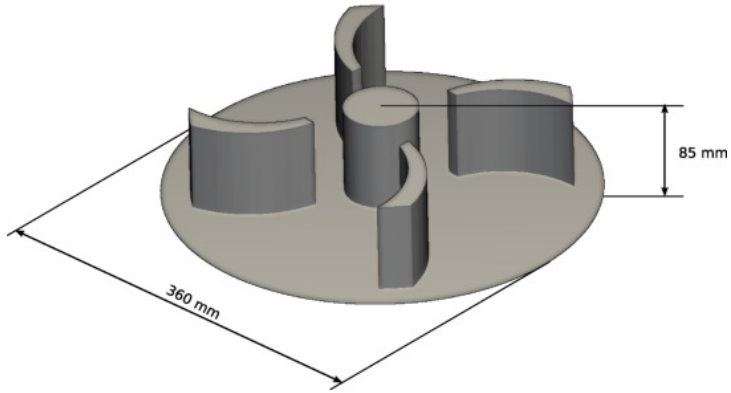
```
MRF1
{
    cellZone    movingCells;
    active      yes;

    nonRotatingPatches (topBottom);

    origin    (0 0 0);
    axis      (0 0 1);
    omega     -100;
}
```

Creating MRF zones

```
actions
(
  {
    name    movingCells;
    type    cellSet;
    action  new;
    source  cylinderToCell;
    sourceInfo
    {
      p1 ( 0 0  -1);
      p2 ( 0 0 0.085);
      radius 0.180;
    }
  }
);
```

`topoSet`

`setsToZones -noFlipMap`



Create a cellSet including all cells inside a cylinder with

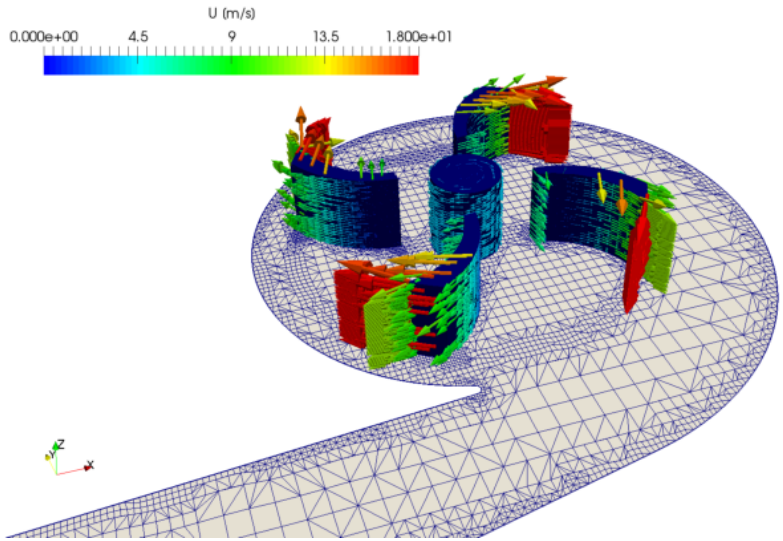
axis $\boldsymbol{z} = (0, 0, 1)$

- origin $\boldsymbol{O} = (0, 0, 0)$
- height $h = 0.085$ m

convert cellSet to cellZone

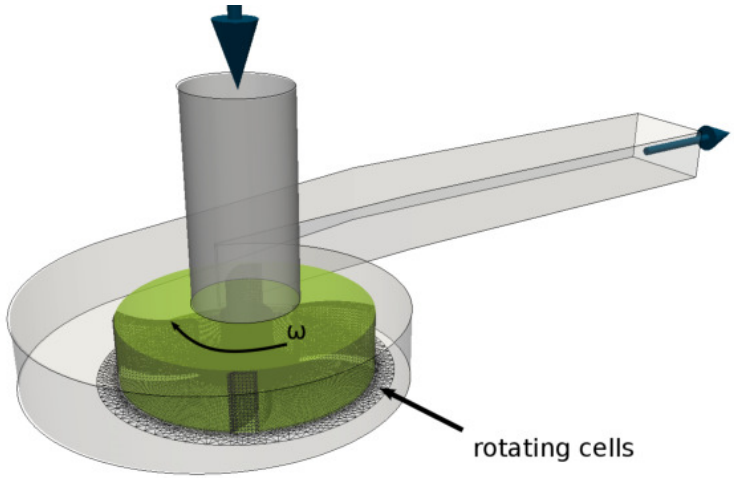
Boundary conditions

```
// file 0/U
impeller
{
  type              rotatingWallVelocity;
  origin            (0 0 0);
  axis              (0 0 1);
  omega             -100;
  value             uniform (0 0 0);
}
```



- No-slip BC on impeller blades: `rotatingWallVelocity`
  - axis $\boldsymbol{z} = (0, 0, 1)$
  - origin $\boldsymbol{O} = (0, 0, 0)$
  - speed $\omega = 100$ rad/s
- Other walls have zero velocity
- Velocity at inlet/outlet determined by pressure gradient
- Total pressure at inlet, total pressure at outlet
- Other settings (turbulence, numerics, ...) omitted here

Case Overview



Preprocessing steps

- Mesh generation (snappyHexMesh) ✓
- Mesh adjustments ✓
- Models set up ✓
- Post-processing set up
- Decompose

Running in parallel

The method of parallel computing used by OpenFOAM® is known as domain decomposition, in which the geometry and associated fields are broken into pieces and allocated to separate processors for solution.

1. The first step is to decompose the domain using the `decomposePar` utility (edit the dictionary named `decomposeParDict` which is located in the system directory of the case.)
2. The parallel running uses the public domain openMPI implementation of the standard message passing interface (MPI). Each processor runs a copy of the solver on a separate part of the domain mesh.
3. Finally the solution is reconstructed to obtain the final result.

PLEASE CHECK THE SLIDES ABOUT PARALLEL RUNNING!

Hands on – 1

Extract the mesh from tar file:

Create faceSet of “lowerWall” patch

Fix patch names:

Renumber the mesh:

Create sets for MRF

convert sets to zones:

rename `0.org` to `0`

decompose

run solver:

Hands on – 1

Extract the mesh from tar file: `tar xzf centrifugalPump-mesh.tgz`

Create faceSet of "lowerWall" patch `topoSet -dict system/topoSetDict.`

Fix patch names: `createPatch -overwrite`

Renumber the mesh: `renumberMesh -overwrite`

Create sets for MRF `topoSet`

convert sets to zones: `setsToZones -noFlipMap`

rename `0.org` to `0` `cp -r 0.org 0`

decompose `decomposePar`

run solver: `mpirun -np 4 simpleFoam -parallel > log`

Hands on – 2

Data to extract:

- p, $\boldsymbol{U}$ on a y- normal plane
- p on impeller blades
- Streamlines
- p along the outlet channel
- Torque (on impeller)
- p_2, p_1
- $\dot{V}_2$, $\dot{V}_1$
- Q-criterion, vorticity
- iso-Q surfaces
- residual history

Thank you for your attention!

contact: federico.piscaglia@polimi.it