Turbomachinery training at OFW8

Håkan Nilsson

Applied Mechanics/Fluid Dynamics,
Chalmers University of Technology,
Gothenburg, Sweden

Contributions from:
Maryse Page and Martin Beaudoin, IREQ, Hydro Quebec
Hrvoje Jasak, Wikki Ltd.

Using OpenFOAM-1.6-ext

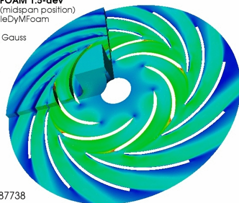
2013-06-11

What's this training about?

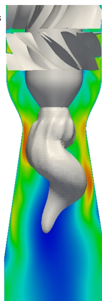
- *Turbo* means *spin*, or *whirl*
- Our focus is thus on *rotating machinery* and functionality that is related to rotation
- We will investigate the theory and application of SRF, MRF, moving mesh, coupling interfaces, and other useful features
- We will investigate the differences between the basic solvers and the ones including rotation. The examples will use incompressible flow solvers, but the functionalities should be similar for compressible flow
- We will mainly use the tutorials distributed with OpenFOAM-1.6-ext to learn how to set up and run cases

Full cases in the Sig Turbomachinery Wiki

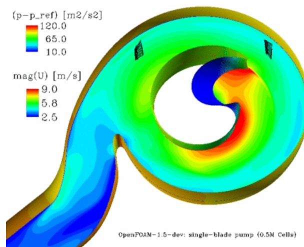
http://openfoamwiki.net/index.php/Sig_Turbomachinery



ERCOFTAC
Centrifugal Pump (ECP)



Timisoara
Swirl Generator (TSG)



Single Channel Pump

Prerequisites

You know how to ...

- use Linux commands
- run the basic OpenFOAM tutorials
- use the OpenFOAM environment
- compile parts of OpenFOAM
- read the implementation of `simpleFoam` and `icoFoam`
- read C/C++ code

Learning outcomes

You will know ...

- the underlying theory of SRF, MRF and moving mesh
- how to find applications and libraries for rotating machinery
- how to figure out what those applications and libraries do
- how a basic solver can be modified for rotation
- how to set up cases for rotating machinery

Fundamental features for CFD in rotating machinery

Necessary:

- Utilities for special mesh/case preparation
- Solvers that include the effect of rotation of (part(s) of) the domain
- Libraries for mesh rotation, or source terms for the rotation
- Coupling of rotating and steady parts of the mesh

Useful:

- Specialized boundary conditions for rotation and axi-symmetry
- A cylindrical coordinate system class
- Tailored data extraction and post-processing

Training organization

The rotation approaches (SRF, MRF, moving mesh) are presented as:

- Theory
- Solver, compared to basic solver
- Classes, called by additions to basic solver
- Summary of difference from basic solver
- Tutorial - how to set up and run
- Dictionaries and utilities
- Special boundary conditions

This is followed by:

- Coupling interfaces - GGI
- Other useful information

Single rotating frame of reference (SRF), theory

- Compute in the rotating frame of reference, with velocity and fluxes relative to the rotating reference frame, using Cartesian components.
- Coriolis and centrifugal source terms in the momentum equations (laminar version):

$$\nabla \cdot (\vec{u}_R \otimes \vec{u}_R) + \underbrace{2\vec{\Omega} \times \vec{u}_R}_{Coriolis} + \underbrace{\vec{\Omega} \times (\vec{\Omega} \times \vec{r})}_{centrifugal} = -\nabla(p/\rho) + \nu\nabla \cdot \nabla(\vec{u}_R)$$

$$\nabla \cdot \vec{u}_R = 0$$

where $\vec{u}_R = \vec{u}_I - \vec{\Omega} \times \vec{r}$

- See derivation at:

`http://openfoamwiki.net/index.php/See_the_MRF_development`

The simpleSRFFoam solver

- Code:
  `$FOAM_TUTORIALS/incompressible/simpleSRFFoam/simpleSRFFoam`
- Not compiled by default
- Difference from simpleFoam:
  - `Urel` instead of `U`
  - In header of `simpleSRFFoam.C`: `#include "SRFModel.H"`
  - In `createFields.H`:
    ```
    Info<< "Creating SRF model\n" << endl;
    autoPtr<SRF::SRFModel> SRF
    (
        SRF::SRFModel::New(Urel)
    );
    ```
  - In `UrelEqn` of `simpleSRFFoam.C`: `+ SRF->Su()`
  - At end of `simpleSRFFoam.C`, calculate and write also the absolute velocity: `Urel + SRF->U()`

What is then implemented in the `SRFModel` class?

The SRFModel class

- Code:
  `$FOAM_SRC/finiteVolume/cfdTools/general/SRF/SRFModel/SRFModel`
- Reads `constant/SRFProperties` to set: `axis_` and `omega_`
- Computes `Su` as `Fcoriolis() + Fcentrifugal()`
  where `Fcoriolis()` is `2.0*omega_ ^ Urel_`
  and `Fcentrifugal()` is `omega_ ^ (omega_ ^ mesh_.C())`
- Computes `U` as `omega_ ^ (mesh_.C() - axis_*(axis_ & mesh_.C()))`
- ... and e.g. a `velocity` member function (`positions` as argument):

```
return omega_.value() ^ (positions - axis_*(axis_ & positions));
```

Summary of difference between simpleSRFFoam and simpleFoam

The `simpleSRFFoam` solver is derived from the `simpleFoam` solver by

- adding to `UEqn` (LHS): `2.0*omega ^ U + omega ^ (omega ^ mesh.C())`
- specifying the `omega` vector
- defining the velocity as the relative velocity

Compile simpleSRFFoam and run the mixer tutorial

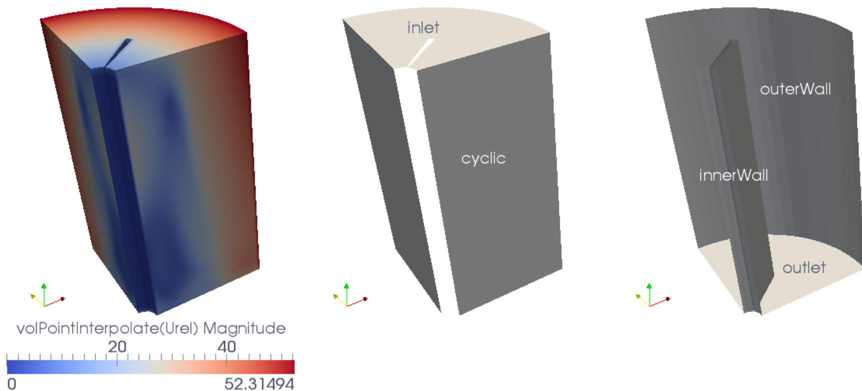
- Compile solver

```
cd $FOAM_RUN/../applications
cp -r $FOAM_TUTORIALS/incompressible/simpleSRFFoam/simpleSRFFoam .
wmake simpleSRFFoam
```

- Run tutorial

```
cp -r $FOAM_TUTORIALS/incompressible/simpleSRFFoam/mixer $FOAM_RUN
cd $FOAM_RUN/mixer
./Allrun >& log_Allrun &
```

simpleSRFFoam mixer tutorial results and boundary names



The SRFProperties file

The rotation is specified in `constant/SRFProperties`:

```
SRFModel  rpm;

axis (0 0 1);

rpmCoeffs
{
    rpm  5000.0;
}
```

Currently, the rotational speed can only be specified in rpm, but can easily be extended starting from:

```
$FOAM_SRC/finiteVolume/cfdTools/general/SRF/SRFModel/rpm
```

Boundary condition, special for SRF

Boundary condition for `Urel`:

```
inlet
{
    type             SRFVelocity;
    inletValue       uniform (0 0 -10); // Absolute CARTESIAN velocity
    relative         yes; // yes means that rotation is subtracted from inletValue
                          // (Urel = inletValue - omega X r)
                          // and makes sure that conversion to Uabs
                          // is done correctly
                          // no means that inletValue is applied as is
                          // (Urel = inletValue)
    value            uniform (0 0 0); // Just for paraFoam
}
```

Next slide shows the implementation...

The SRFVelocity boundary condition

- Code:

  `$FOAM_SRC/finiteVolume/cfdTools/general/SRF/\`

  `derivedFvPatchFields/SRFVelocityFvPatchVectorField`

- In `updateCoeffs`:

```
// If relative, include the effect of the SRF
if (relative_)
{
    // Get reference to the SRF model
    const SRF::SRFModel& srf =
        db().lookupObject<SRF::SRFModel>("SRFProperties");

    // Determine patch velocity due to SRF
    const vectorField SRFVelocity = srf.velocity(patch().Cf());

    operator==(-SRFVelocity + inletValue_);
}
else // If absolute, simply supply the inlet value as a fixed value
{
    operator==(inletValue_);
}
```

Multiple frames of reference (MRF), theory

- Compute the absolute Cartesian velocity components, using the flux relative to the rotation of the local frame of reference (rotating or non-rotating)
- Development of the SRF equation, with convected velocity in the inertial reference frame (laminar version):

$$\nabla \cdot (\vec{u}_R \otimes \vec{u}_I) + \vec{\Omega} \times \vec{u}_I = -\nabla(p/\rho) + \nu \nabla \cdot \nabla(\vec{u}_I)$$

$$\nabla \cdot \vec{u}_I = 0$$

- The same equations apply in all regions, with different Ω.
  If $\vec{\Omega} = \vec{0}$, $\vec{u}_R = \vec{u}_I$
- See derivation at:
  `http://openfoamwiki.net/index.php/See_the_MRF_development`

The MRFSimpleFoam solver

- Code:
  `$FOAM_TUTORIALS/incompressible/MRFSimpleFoam/MRFSimpleFoam`
- Not compiled by default
- Difference from simpleFoam:
  - In header of `MRFSimpleFoam.C`:
    ```
    #include "MRFZones.H"
    ```
  - In `createFields.H`:
    ```
    MRFZones mrfZones(mesh);
    mrfZones.correctBoundaryVelocity(U);
    ```
  - Modify `UEqn` in `MRFSimpleFoam.C`:
    ```
    mrfZones.addCoriolis(UEqn());
    ```
  - Calculate the relative flux in the rotating regions:
    ```
    phi = fvc::interpolate(U, "interpolate(HbyA)") & mesh.Sf();
    mrfZones.relativeFlux(phi);
    ```
  - Thus, the *relative* flux is used in `fvm::div(phi, U)` and `fvc::div(phi)`

What is then implemented in the `MRFZones` class?

The MRFZones class (1/5) – Constructor

- Code:
  `$FOAM_SRC/finiteVolume/cfdTools/general/MRF/MRFZone.C`
- Reads `constant/MRFZones` to:
  - Get the names of the rotating MRF zones.
  - Get for each MRF zone:
    - `nonRotatingPatches` (`excludedPatchNames_` internally)
    - origin (`origin_` internally)
    - axis (`axis_` internally)
    - omega (`omega_` internally, and creates vector `Omega_`)
- Calls setMRFFaces()...

The MRFZones class (2/5) – Constructor: setMRFFaces()

- Arranges faces in each MRF zone according to
  - `internalFaces_`
    where the *relative flux* is computed from interpolated absolute velocity minus solid-body rotation.
  - `includedFaces_` (default, overridden by `nonRotatingPatches`)
    where solid-body rotation **absolute velocity vectors are fixed** and **zero relative flux is imposed**, i.e. those patches are set to rotate with the MRF zone. (The velocity boundary condition is overridden!!!)
  - `excludedFaces_` (coupled patches and `nonRotatingPatches`)
    where the *relative flux* is computed from the (interpolated) absolute velocity minus solid-body rotation, i.e. those patches are treated as `internalFaces_`. Stationary walls should have zero absolute velocity.

  - Those can be visualized as `faceSets` if `debug` is activated for `MRFZone` in the global `controlDict` file.

The MRFZones class (3/5) – Foam::MRFZone::correctBoundaryVelocity

For each MRF zone, set the rotating solid body *velocity*, $\vec{\Omega} \times \vec{r}$, on *included* boundary faces:

```
void Foam::MRFZone::correctBoundaryVelocity(volVectorField& U) const
{
    const vector& origin = origin_.value();
    const vector& Omega = Omega_.value();
    // Included patches
    forAll(includedFaces_, patchi)
    {
        const vectorField& patchC = mesh_.Cf().boundaryField()[patchi];
        vectorField pfld(U.boundaryField()[patchi]);
        forAll(includedFaces_[patchi], i)
        {
            label patchFacei = includedFaces_[patchi][i];
            pfld[patchFacei] = (Omega ^ (patchC[patchFacei] - origin));
        }
        U.boundaryField()[patchi] == pfld;
    }
}
```

The MRFZones class (4/5) – Foam::MRFZone::addCoriolis

For each MRF zone, add $\vec{\Omega} \times \vec{U}$ as a source term in `UEqn` (minus on the RHS)

```
void Foam::MRFZone::addCoriolis(fvVectorMatrix& UEqn) const
{
    if (cellZoneID_ == -1)
    {
        return;
    }

    const labelList& cells = mesh_.cellZones()[cellZoneID_];
    const scalarField& V = mesh_.V();
    vectorField& Usource = UEqn.source();
    const vectorField& U = UEqn.psi();
    const vector& Omega = Omega_.value();

    forAll(cells, i)
    {
        label celli = cells[i];
        Usource[celli] -= V[celli]*(Omega ^ U[celli]);
    }
}
```

The MRFZones class (5/5) – Foam::MRFZone::relativeFlux

For each MRF zone, make the given absolute mass/vol flux relative. Calls `Foam::MRFZone::relativeRhoFlux` in `MRFZoneTemplates.C`. I.e., on internal and excluded faces $\phi_{rel} = \phi_{abs} - (\vec{\Omega} \times \vec{r}) \cdot \vec{A}$. On included faces: $\phi_{rel} = 0$

```
template<class RhoFieldType>
void Foam::MRFZone::relativeRhoFlux
(
    const RhoFieldType& rho,
    surfaceScalarField& phi
) const
{
const surfaceVectorField& Cf = mesh_.Cf();
const surfaceVectorField& Sf = mesh_.Sf();
const vector& origin = origin_.value();
const vector& Omega = Omega_.value();
// Internal faces
forAll(internalFaces_, i)
{
    label facei = internalFaces_[i];
    phi[facei] -= rho[facei]*
      (Omega ^ (Cf[facei] - origin)) & Sf[facei];
}
```

```
// Included patches
forAll(includedFaces_, patchi)
{
    forAll(includedFaces_[patchi], i)
    {
        label patchFacei = includedFaces_[patchi][i];
        phi.boundaryField()[patchi][patchFacei] = 0.0;
    }
}
// Excluded patches
forAll(excludedFaces_, patchi)
{
    forAll(excludedFaces_[patchi], i)
    {
        label patchFacei = excludedFaces_[patchi][i];
        phi.boundaryField()[patchi][patchFacei] -=
            rho.boundaryField()[patchi][patchFacei]
           *(Omega ^
            (Cf.boundaryField()[patchi][patchFacei]
             - origin))
          & Sf.boundaryField()[patchi][patchFacei];
}}}
```

Summary of difference between MRFSimpleFoam and simpleFoam

The `MRFSimpleFoam` solver is derived from the `simpleFoam` solver by

- defining regions and setting the `Omega` vector in each region
- setting a solid-body rotation velocity at included patch faces
- adding `-V[celli]*(Omega ^ U[celli])` to `UEqn.source()`
- setting a relative face flux for use in `fvm::div(phi, U)` and `fvc::div(phi)` (explicitly set to zero for included patch faces, as it should be)

Note that setting a relative face flux at a face between two regions with different rotational speed requires that the face normal has no component in the tangential direction! I.e. the interface between those regions must be axi-symmetric!!!

Compile MRFSimpleFoam and run the mixerVessel2D tutorial

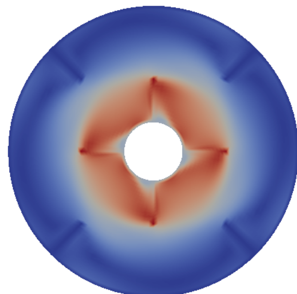
- Compile solver

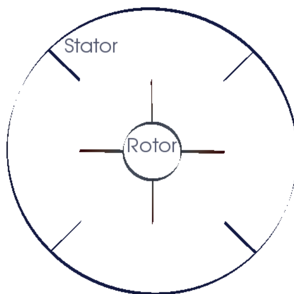
```
cd $FOAM_RUN/../applications
cp -r $FOAM_TUTORIALS/incompressible/MRFSimpleFoam/MRFSimpleFoam .
wmake MRFSimpleFoam
```

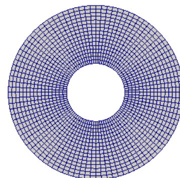
- Run tutorial (don't care about the wmake error message)

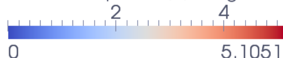
```
cp -r $FOAM_TUTORIALS/incompressible/MRFSimpleFoam/mixerVessel2D $FOAM_RUN
cd $FOAM_RUN/mixerVessel2D
./Allrun >& log_Allrun &
```

MRFSimpleFoam mixerVessel2D tutorial results, boundary names, and rotor cellZone









The `rotor cellzone` is used to define where to apply the additional term

Note that the solution resembles a snap-shot of a specific rotor orientation. Wakes will become unphysical!

Mesh generation and modification

- The `makeMesh` file:

```
m4 < constant/polyMesh/blockMeshDict.m4 > constant/polyMesh/blockMeshDict
blockMesh
cellSet #Actually not needed in this case!!!
setsToZones -noFlipMap #Actually not needed in this case!!!
```

- The `blockMeshDict` tells `blockMesh` to create the `rotor cellZone` and to write that zone as a `cellSet` (the `cellSet` is not used by `MRFSimpleFoam`)
- If some other tool than `blockmesh` is used, the `rotor cellZone` must be created some way. We'll come back to that...

Descriptions of `cellSet` and `setsToZones`:

- `cellSet` reads the `cellSetDict` and in this case uses the `rotor cellZone` to write that zone as a `cellSet`. This was already done by `blockMesh`, so in fact it doesn't have to be done again.
- `setsToZones -noFlipMap` uses the `rotor` cellSet to create the same `cellZone` as we started with, so that is also not needed.

The MRFZones file

For each zone in `cellZones`:

```
rotor // Name of MRF zone
{
    //patches   (rotor); //OBSOLETE, IGNORED! See next two lines
    // Fixed patches (by default they 'move' with the MRF zone)
    nonRotatingPatches (); // I.e. the rotor patch will rotate

    origin    origin [0 1 0 0 0 0 0]  (0 0 0);
    axis      axis   [0 0 0 0 0 0 0]  (0 0 1);
    omega     omega  [0 0 -1 0 0 0 0] 104.72; //In radians per second
}
```

There is a `dynamicMeshDict` file, but it is not used

Special for MRF cases

- Note that the velocity, `U`, is the absolute velocity.
- At patches not defined as `nonRotatingPatches`, the velocity boundary condition will be overridden and given a solid-body rotation velocity.
- In the `mixerVessel2D` tutorial we use a single mesh region, but quite often two regions are coupled. We will get back to that...
- **Always make sure that the interfaces between the zones are perfectly axi-symmetric**. Although the solver will probably run also if the mesh surface between the static and MRF zones is not perfectly symmetric about the axis, it will not make sense. Further, if a GGI is used at such an interface, continuity will not be fulfilled.

Moving meshes, theory

- We will limit ourselves to non-deforming meshes with a fixed topology and a known rotating mesh motion
- Since the coordinate system remains fixed, and the Cartesian velocity components are used, the only change is the appearance of the relative velocity in convective terms. In cont. and mom. eqs.:

$$\int_S \rho\vec{v}\cdot\vec{n}dS \longrightarrow \int_S \rho(\vec{v}-\vec{v}_b)\cdot\vec{n}dS$$

$$\int_S \rho u_i\vec{v}\cdot\vec{n}dS \longrightarrow \int_S \rho u_i(\vec{v}-\vec{v}_b)\cdot\vec{n}dS$$

where $\vec{v}_b$ is the integration boundary (face) velocity

- See derivation in:
  Ferziger and Perić, Computational Methods for Fluid Dynamics

The icoDyMFoam solver

- Code:
  `$FOAM_SOLVERS/incompressible/icoDyMFoam`
- Important differences from `icoFoam`, for non-morphing meshes (`mixerGgiFvMesh` and `turboFvMesh`, we'll get back...):
  - In header of `icoDyMFoam.C`: `#include "dynamicFvMesh.H"`
  - At start of main function in `icoDyMFoam.C`:
    ```
    # include "createDynamicFvMesh.H" //instead of createMesh.H
    ```
  - Before `# include UEqn.H`:
    ```
    bool meshChanged = mesh.update(); //Returns false in the present cases
    ```
  - After calculating and correcting the new absolute fluxes:
    ```
    // Make the fluxes relative to the mesh motion
    fvc::makeRelative(phi, U);
    ```
- I.e. the relative flux is used everywhere except in the pressure-correction equation, which is not affected by the mesh motion for incompressible flow (Ferziger&Perić)

We will now have a look at the `dynamicFvMesh` classes and the functions used above...

dynamicMesh classes

- The `dynamicMesh` classes are located in:
  `$FOAM_SRC/dynamicMesh`
  There are two major branches, bases on how the coupling is done:
- GGI (no mesh modifications, i.e. non morphing)
  - `$FOAM_SRC/dynamicMesh/dynamicFvMesh/mixerGgiFvMesh`
    Tutorial: `$FOAM_TUTORIALS/incompressible/icoDyMFoam/mixerGgi`
  - `$FOAM_SRC/dynamicMesh/dynamicFvMesh/turboFvMesh`
    Tutorial: `$FOAM_TUTORIALS/incompressible/icoDyMFoam/turboPassageRotating`
    (3D, 2D case supplied)
- Topological changes (morphing, not covered in the training)
  - `$FOAM_SRC/dynamicMesh/topoChangerFvMesh/mixerFvMesh`
    Tutorial: `$FOAM_TUTORIALS/incompressible/icoDyMFoam/mixer2D`
  - `$FOAM_SRC/dynamicMesh/topoChangerFvMesh/multiMixerFvMesh`
    No tutorial

We focus on `turboFvMesh` ...

In $FOAM_SRC/dynamicMesh/dynamicFvMesh/turboFvMesh

```
bool Foam::turboFvMesh::update()
{
    movePoints
    (
        csPtr_->globalPosition
        (
            csPtr_->localPosition(allPoints())
          + movingPoints()*time().deltaT().value()
        )
    );

    // The mesh is not morphing
    return false;
}
```

Member data `csPtr_` is the coordinate system read from the `dynamicMeshDict` dictionary. Member function `movingPoints()` uses the `rpm` for each rotating `cellZone`, specified in the `dynamicMeshDict` dictionary, and applies it as an angular rotation in the cylindrical coordinate system.

In $FOAM_SRC/finiteVolume/finiteVolume/fvc/fvcMeshPhi.C

```
void Foam::fvc::makeRelative
(
    surfaceScalarField& phi,
    const volVectorField& U
)
{
    if (phi.mesh().moving())
    {
        phi -= fvc::meshPhi(U);
    }
}
```

I.e. the mesh flux is subtracted from `phi`.

- In the general dynamic mesh case, moving/deforming cells may cause the conservation equation not to be satisfied (Ferziger&Perić).
- Mass conservation can be enforced using a space conservation law, which will depend on which time discretization is used. An example is provided, but the details are left for another training...

In $FOAM_SRC/finiteVolume/finiteVolume/fvc/fvcMeshPhi.C

```
Foam::tmp<Foam::surfaceScalarField> Foam::fvc::meshPhi
(
    const volVectorField& vf
)
{
    return fv::ddtScheme<vector>::New
    (
        vf.mesh(),
        vf.mesh().ddtScheme("ddt(" + vf.name() + ')')
    )().meshPhi(vf);
}
```

E.g. `$FOAM_SRC/finiteVolume/finiteVolume/ddtSchemes/EulerDdtScheme/EulerDdtScheme.C:`

```
template<class Type>
tmp<surfaceScalarField> EulerDdtScheme<Type>::meshPhi
(
    const GeometricField<Type, fvPatchField, volMesh>&
)
{
    return mesh().phi(); // See $FOAM_SRC/finiteVolume/fvMesh/fvMeshGeometry.C
}
```

Summary of difference between icoDyMFoam and icoFoam

- Move the mesh before the momentum predictor
- Make the fluxes relative after the pressure-correction equation
- The relative flux is used everywhere except in the pressure-correction equation

Run the TurboPassageRotating2D tutorial

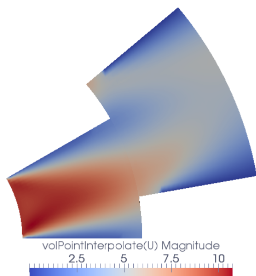
- 2D on OFW8 stick:

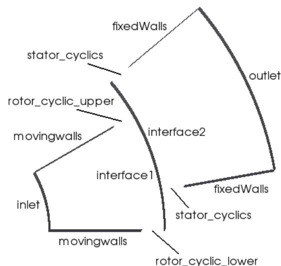
```
cp -r /home/openfoam/training_materials/Track3-1/turboPassageRotating2D $FOAM_RUN
cd $FOAM_RUN/turboPassageRotating2D
./Allrun >& log_Allrun &
```

- 3D in OpenFOAM-1.6-ext distribution:

```
cp -r $FOAM_TUTORIALS/incompressible/icoDyMFoam/turboPassageRotating $FOAM_RUN
cd $FOAM_RUN/turboPassageRotating
./Allrun >& log_Allrun &
```

icoDyMFoam turboPassageRotating tutorial results, boundary names, and cellZones





The `cellRegion0` `cellZone` is used to set mesh rotation

The Allrun script

- `blockMesh`: Creates the mesh with two regions (no zones)
- `cp constant/polyMesh/boundary.org constant/polyMesh/boundary`: Some information needed by GGI interfaces is not generated by `blockMesh`, so it has been prepared (we'll get back to this later...)
- `setSet -batch setBatch`: Create interface faceSets
- `regionCellSets`: Create one cellSet per mesh region
- `setsToZones -noFlipMap`: Transform sets to zones, without modifying face normals
- `icoDyMFoam`: Run simulation (done in parallel in script)

I.e. we need a cellZone for the rotating region(s), to specify the `rpm`s in `dynamicMeshDict` and faceZones for the GGI interfaces (fix for parallel simulations). The cellZone could have been generated directly by `blockMesh`.

The dynamicMeshDict dictionary

```
dynamicFvMesh       turboFvMesh;     // Use the turboFvMesh class
turboFvMeshCoeffs
{
    coordinateSystem                 // Specify the rotation axis
    {
        type            cylindrical;
        origin          (0 0 0);
        axis            (0 0 1);
        direction       (1 0 0);
    }
    rpm                              // Set the cell rotational speed(s)
    {
       cellRegion0  60;
    }
    slider                           // Set the coupled face rotational speed(s)
    {
        interface1_faces 60;
        rotor_cyclic_upper_faces 60;
        rotor_cyclic_lower_faces 60;
    }
}
```

Special boundary conditions

- For inlet velocity:

```
inlet
{
    type            surfaceNormalFixedValue;
    refValue        uniform -10;
    value           uniform (9.659258262890629 2.588190451025207 0);
}
```

Entry `value` is just for paraFoam

- For moving wall velocity:

```
movingwalls
{
    type            movingWallVelocity;
    value           uniform (0 0 0);
}
```

I.e. the velocity is the same as the moving mesh.

- For coupled patches: `ggi`, `overlapGgi`, `cyclicGgi`. We'll get back...

Coupling interfaces - GGI

We will have a quick look at the GGI (General Grid Interface), without going into theory and implementation (see training OFW6)

- GGI interfaces make it possible to connect two patches with non-conformal meshes.
- The GGI implementations are located here:
  `$FOAM_SRC/finiteVolume/fields/fvPatchFields/constraint/`
- `ggi` couples two patches that typically match
- `overlapGgi` couples two patches that cover the same sector angle
- `cyclicGgi` couples two translationally or rotationally cyclic patches
- In all cases it is necessary to create faceZones of the faces on the patches. This is the way parallelism is treated, but it is a must also when running sequentially.

How to use the ggi interface - the boundary file

- See example in the `icoDyMFoam/mixerGgi` tutorial
- For two patches `patch1` and `patch2` (only `ggi`-specific entries):

```
patch1
{
    type            ggi;
    shadowPatch     patch2;
    zone            patch1Zone;
    bridgeOverlap   false;
}
patch2: vice versa
```

- `patch1Zone` and `patch2Zone` are created by `setSet -batch setBatch`, with the `setBatch` file:

```
faceSet patch1Zone new patchToFace patch1
faceSet patch2Zone new patchToFace patch2
quit
```

- Setting `bridgeOverlap false` disallows partially or completely uncovered faces, where `true` sets a slip wall boundary condition.

How to use the overlapGgi interface - the boundary file

- See example in the `icoDyMFoam/turboPassageRotating` tutorial
- For two patches `patch1` and `patch2` (only `overlapGgi`-specific entries):

```
patch1
{
    type            overlapGgi;
    shadowPatch     patch2;      // See ggi description
    zone            patch1Zone;  // See ggi description
    rotationAxis    (0 0 1);
    nCopies         12;
}
patch2: vice versa
```

- `rotationAxis` defines the rotation axis
- `nCopies` specifies how many copies of the segment that fill up a full lap (360 degrees)

How to use the cyclicGgi interface - the boundary file

- See example in the `icoDyMFoam/turboPassageRotating` tutorial
- For two patches `patch1` and `patch2` (only `cyclicGgi`-specific entries):

```
patch1
{
    type             cyclicGgi;
    shadowPatch      patch2;       // See ggi description
    zone             patch1Zone;   // See ggi description
    bridgeOverlap    false;        // See ggi description
    rotationAxis     (0 0 1);
    rotationAngle    -30;
    separationOffset (0 0 0);
}
patch2: vice versa, with different rotationAxis/Angle combination
```

- `rotationAxis` defines the rotation axis of the `rotationAngle`
- `rotationAngle` specifies how many degrees the patch should be rotated about its rotation axis to match the `shadowPatch`
- `separationOffset` is used for translationally cyclic patches

How to use the GGI interfaces - time directories and decomposePar

- The type definition in the `boundary` file must also be set in the time directory variable files:
  - `type                ggi;`
  - `type                overlapGgi;`
  - `type                cyclicGgi;`
- The faceZones must be made global, for parallel simulations, with a new entry in `decomposeParDict`:

```
globalFaceZones
(
    patch1zone
    patch2Zone
);  // Those are the names of the face zones created previously
```

Info from ggi

- At the beginning of the simulation, or every time the `ggi` needs to be re-evaluated (moving meshes), weighting factor corrections are reported:

```
Evaluation of GGI weighting factors:
  Largest slave weighting factor correction : 0.00012549019 average: 3.0892875e-05
  Largest master weighting factor correction: 3.2105724e-08 average: 4.4063979e-10
```

- The values should be small!

The ggiCheck functionObject

- Prints out the flux through `ggi/cyclicGgi` interface pairs
- Entry in the `system/controlDict` file:

```
functions
(
    ggiCheck
    {
        type ggiCheck; // Type of functionObject
        phi phi;       // The name of the flux variable
        // Where to load it from (if not already in solver):
        functionObjectLibs ("libcheckFunctionObjects.so");
    }
);
```

- Output example:

```
Cyclic GGI pair (patch1, patch2) : 0.0006962669457 0.0006962669754
Diff = 8.879008314e-12 or 1.27523048e-06 %
```

Mesh generation and cellZones

We need cellZones, which can be created e.g.

- directly in `blockMesh`
- from a multi-region mesh using `regionCellSets` and `setsToZones -noFlipMap`
- using the `cellSet` utility, the `cylinderToCell` cellSource, and `setsToZones -noFlipMap`
- in a third-party mesh generator, and converted using `fluent3DMeshToFoam`

You can/should check your zones in paraFoam (`Include Zones`, or `foamToVTK`)

Use perfectly axi-symmetric interfaces between the zones!

Boundary conditions that may be of interest

In `$FOAM_SRC/finiteVolume/fields/fvPatchFields/derived`:

- `movingWallVelocity` Only normal component, moving mesh!
- `rotatingWallVelocity` Only tangential component, spec. axis/omega!
- `movingRotatingWallVelocity` Combines normal component, moving mesh, and tangential component, spec. axis/rpm
- `flowRateInletVelocity` Normal velocity from flow rate
- `surfaceNormalFixedValue` Normal velocity from scalar
- `rotatingPressureInletOutletVelocity` C.f. `pressureInletOutletVelocity`
- `rotatingTotalPressure` C.f. `totalPressure`

At `http://openfoamwiki.net/index.php/Sig_Turbomachinery_Library_OpenFoamTurbo`, e.g.:

- `profile1DfixedValue` Set 1D profile at axi-symmetric (about Z) patch

Utilities and functionObjects

At http://openfoamwiki.net/index.php/Sig_Turbomachinery

- The `convertToCylindrical` utility
  Converts `U` to `Urel`. Note that Z-axis must be the center axis of rotation, but you can easily make it general with the `cylindricalCS` class in `$FOAM_SRC/OpenFOAM/coordinateSystems`
- The `turboPerformance` functionobject
  Computes head, power (from walls and inlet/outlet), efficiency, force (pressure, viscous), moment (pressure, viscous)
  Outputs in log file and forces, fluidPower and turboPerformance directories.
  Example entry for `controlDict` (change `rotor` to `movingwalls` to run with `turboPassageRotating`)

Questions?

Further information

- http://openfoamwiki.net/index.php/Sig_Turbomachinery
- http://www.extend-project.de/user-groups/11/viewgroup/groups
- http://www.tfd.chalmers.se/~hani/kurser/OS_CFD
  (if you want to link, please add the year as e.g.: OS_CFD_2012)