Rotating machinery training at OFW11

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 foam-extend-3.3 (maybe numbered 4.0)

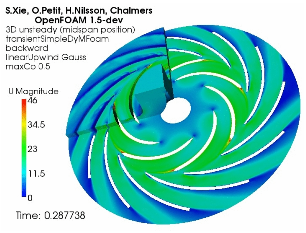
2016-06-28

What's this training about?

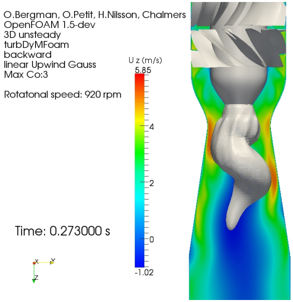
- The focus is on *rotating machinery* and functionality that is related to rotation
- We will investigate the theory and application of SRF, MRF, moving mesh, coupling interfaces, and other useful features
- We will investigate the differences between the basic solvers and the ones including rotation. The examples will use incompressible flow solvers, but the functionalities should be similar for compressible flow
- We will mainly use the tutorials distributed with foam-extend-3.3 to learn how to set up and run cases

Full cases in the Sig Turbomachinery Wiki

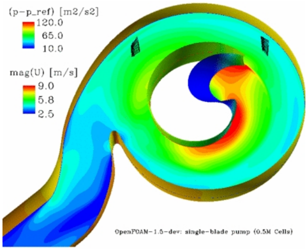
http://openfoamwiki.net/index.php/Sig_Turbomachinery



ERCOFTAC
Centrifugal Pump (ECP)



Timisoara
Swirl Generator (TSG)



Single Channel Pump
(SCP)

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
- Tutorials - how to set up and run
- Dictionaries and utilities
- Special boundary conditions

This is followed by:

- Constraint patches - cyclic, GGI of different flavours
- 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_SOLVERS/incompressible/simpleSRFFoam`
- Difference from simpleFoam (use 'kompare' with 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

The simpleSRFFoam axialTurbine tutorial

- Run tutorial:

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

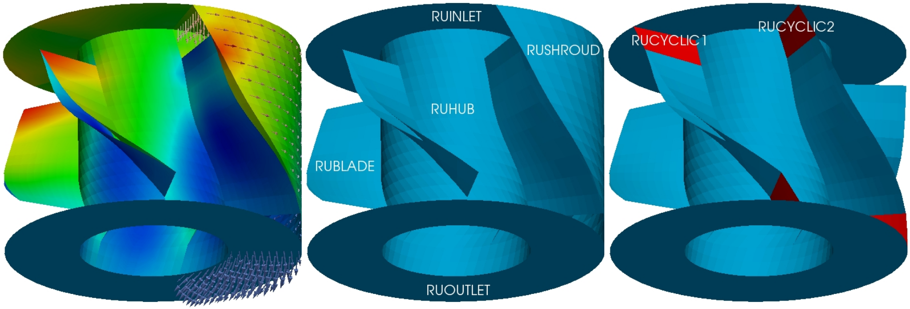
- Look at the results:

```
paraview --state=allBlades.pvsm
```

- Clean up:

```
./Allclean
```

simpleSRFFoam axialTurbine tutorial results and boundary names



RUINLET has an axial relative inlet velocity (Urel)
RUCYCLIC1 and RUCYCLIC2 are cyclic, using cyclicGgi
RUBLADE and RUHUB have zero relative velocity (Urel)
RUOUTLET has a regular zeroGradient condition

Mesh generation

- The mesh is done with `m4` and `blockMesh`
- Cylindrical coordinates are utilized (modified angle: 1/20)
- The modified angle is transformed back to radians:

```
transformPoints -scale "(1 20 1)"
```

- The coordinates are transformed to Cartesian:

```
transformPoints -cylToCart "((0 0 0) (0 0 1) (1 0 0))"
```

- GGI zones are created (see `setBatchGgi`):

```
setSet -batch setBatchGgi
setsToZones -noFlipMap
```

- In `system/decomposeParDict`:

```
globalFaceZones ( RUCYCLIC1Zone RUCYCLIC2Zone );
```

- The face zones are available for ParaView in the `VTK` directory

The SRFProperties file

The rotation is specified in `constant/SRFProperties`:

```
SRFModel  rpm;

axis (0 0 1);

rpmCoeffs
{
    rpm  -95.49; //-10 rad/s
}
```

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`:

```
RUINLET
{
    type            SRFVelocity;
    inletValue      uniform (0 0 -1);
    relative        no;  // no means that inletValue is applied as is
                         // (Urel = inletValue)
                         // yes means that rotation is subtracted from inletValue
                         // (Urel = inletValue - omega X r)
                         // and makes sure that conversion to Uabs
                         // is done correctly
    value           uniform (0 0 0); // Just for paraFoam
}
RUSHROUD
{
    type            SRFVelocity;
    inletValue      uniform (0 0 0);
    relative        yes;
    value           uniform (0 0 0);
}
```

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_);
}
```

The ggiCheck functionObject

- The flux balance at the cyclic GGI pair is checked by activating the `ggiCheck` functionobject in `system/controlDict`:

```
// Compute the flux value on each side of a GGI interface
functions
(
    ggiCheck
    {
        // Type of functionObject
        type ggiCheck;

        phi phi;

        // Where to load it from (if not already in solver)
        functionObjectLibs ("libcheckFunctionObjects.so");
    }
);
```

- Output in log file:

```
grep 'Cyclic GGI pair' log.simpleSRFFoam
```

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_SOLVERS/incompressible/MRFSimpleFoam`
- Difference from simpleFoam (use 'kompare' with 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. **Good way to check the case set-up!**

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!!!

Run the MRFSimpleFoam axialTurbine_ggi/mixingPlane tutorials

- Run the axialTurbine_ggi tutorial:

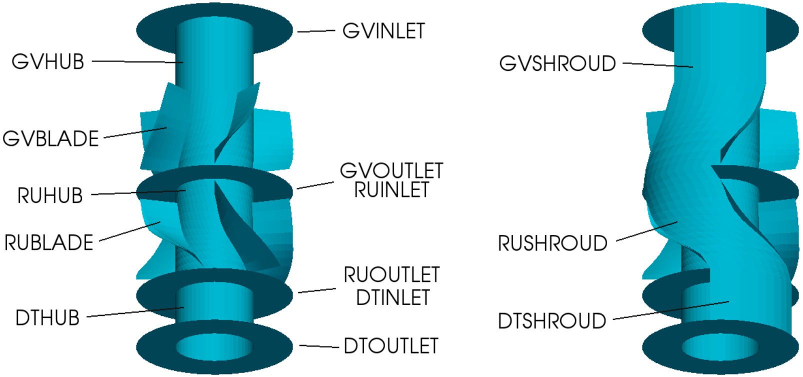
```
cp -r $FOAM_TUTORIALS/incompressible/MRFSimpleFoam/axialTurbine_ggi $FOAM_RUN
cd $FOAM_RUN/axialTurbine_ggi
./Allrun >& log_Allrun &
paraview --state=allBlades.pvsm
./Allclean
```

- Run the axialTurbine_mixingPlane tutorial:

```
tut
cp -r incompressible/MRFSimpleFoam/axialTurbine_mixingPlane $FOAM_RUN
cd $FOAM_RUN/axialTurbine_mixingPlane
./Allrun >& log_Allrun &
paraview --state=allBlades.pvsm
./Allclean
```

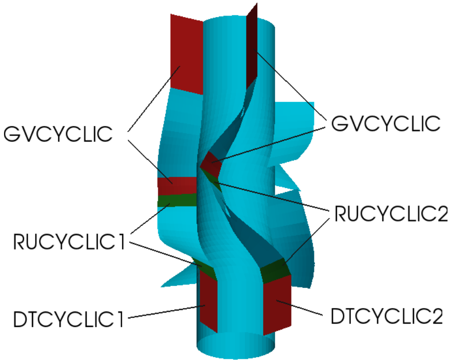
- Same mesh generation procedure as for `simpleSRFFoam/axialTurbine`

MRFSimpleFoam axialTurbine_ggi/mixingPlane tutorial boundary names



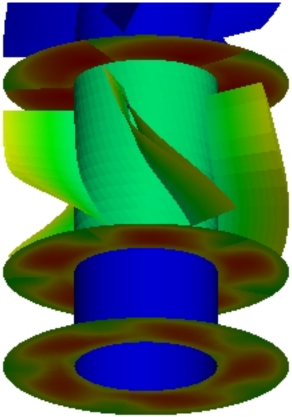
`GVOUTLET/RUINLET` and `RUOUTLET/DTINLET` are coupled using GGI/mixingPlane.

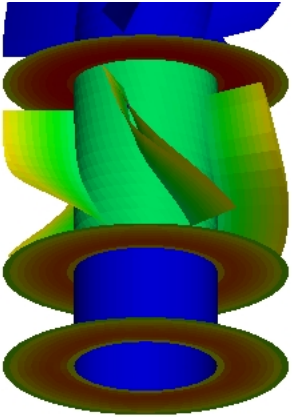
MRFSimpleFoam axialTurbine_ggi/mixingPlane tutorial boundary names



`GVCYCLIC` uses the regular cyclic boundary condition
`{RU,DT}CYCLIC{1,2}` use the cyclicGgi boundary condition

MRFSimpleFoam axialTurbine_ggi/mixingPlane tutorial results





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

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 ( RUSHROUD ); //The shroud does not rotate.
                                     //Note that RUBLADE and RUHUB
                                     //rotate although their
                                     //velocity is set to zero
                                     //in the 0-directory!

    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] -10; //In radians per second
}
```

The rotor `cellZone` is defined in `blockMeshDict.m4`. It also creates a `cellSet`.
Check which cells are marked for rotation: `foamToVTK -cellSet rotor`

Parallel set-up

- All GGI interfaces should be listed in `globalFaceZones` in `system/decomposeParDict`
- You can force the faces of a patch to be on the same processor:

```
method     patchConstrained;
patchConstrainedCoeffs
{
    method              metis;
    numberOfSubdomains    8;
    patchConstraints
    (
        (RUINLET 1)
        (GVOUTLET 1)
        (RUOUTLET 2)
        (DTINLET 2)
    );
}
```

This is currently necessary for the mixingPlane.

Special for MRF cases

- Note that the velocity, `U`, is the *absolute velocity*.
- At patches belonging to a rotational zone, that are not defined as `nonRotatingPatches`, the velocity boundary condition will be overridden and given a solid-body rotation velocity.
- The cell zones may be in multiple regions, as in the `axialTurbine` tutorials, and in a single region, as in the `mixerVessel2D` tutorial. We will get back to the coupling interfaces later.
- **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` (use 'kompare' with 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`
    `$FOAM_TUTORIALS/incompressible/icoDyMFoam/mixerGgi`
  - `$FOAM_SRC/dynamicMesh/dynamicFvMesh/turboFvMesh`
    `$FOAM_TUTORIALS/incompressible/icoDyMFoam/turboPassageRotating`
    `$FOAM_TUTORIALS/incompressible/pimpleDyMFoam/axialTurbine`
- Topological changes (morphing, not covered in the training)
  - `$FOAM_SRC/dynamicMesh/topoChangerFvMesh/mixerFvMesh`
    `$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

The differences between `pimpleDyMFoam` and `pimpleFoam` are similar.

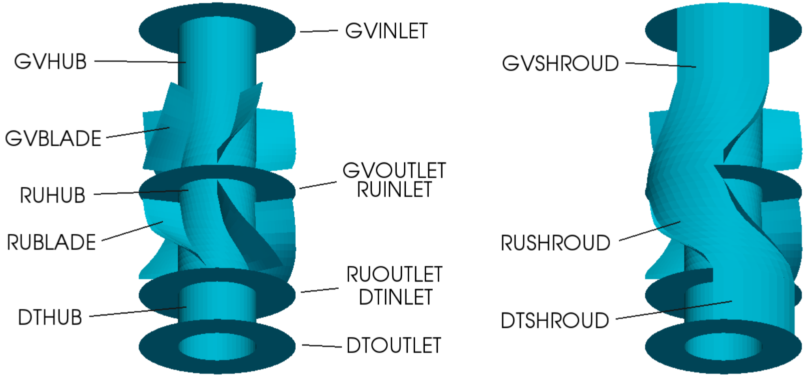
Run the pimpleDyMFoam axialTurbine tutorial

- Run tutorial:

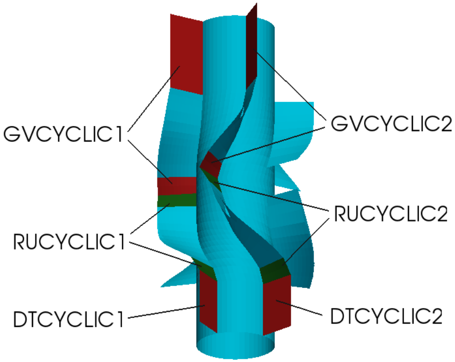
```
tut
cp -r incompressible/pimpleDyMFoam/axialTurbine \
$FOAM_RUN/axialTurbine_overlapGgi
cd $FOAM_RUN/axialTurbine_overlapGgi
./Allrun >& log_Allrun &
paraview --state=allBlades.pvsm #Click on Play!
./Allclean
```

- Mesh generation procedure as before.

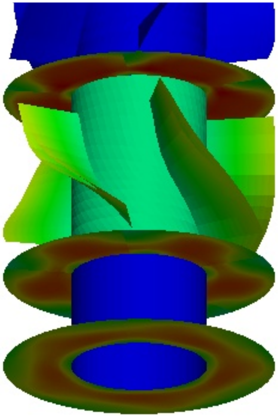
pimpleDyMFoam axialTurbine tutorial boundary names (same as MRFSimpleFoam tutorials)



pimpleDyMFoam axialTurbine tutorial boundary names (almost same as MRFSimpleFoam tutorials - GVCYCLIC differs)



pimpleDyMFoam axialTurbine tutorial results



Note that wakes are now physical!

Main differences from the MRFSimpleFoam tutorial

- Coupling uses the `overlapGgi`
- Rotating walls use `type movingWallVelocity`
- Rotation is specified in `constant/dynamicMeshDict`

The dynamicMeshDict

- In `constant/dynamicMeshDict`:

```
dynamicFvMesh        turboFvMesh;
turboFvMeshCoeffs
{
    coordinateSystem
    {
        type               cylindrical;
        origin             (0 0 0);
        axis               (0 0 1);
        direction          (1 0 0);
    }
    rpm { rotor  -95.49578; }
    slider                    //Probably not needed!
    {
        RUINLET -95.49578;
        RUOUTLET -95.49578;
        RUCYCLIC1 -95.49578;
        RUCYCLIC2 -95.49578;
    }
}
```

Constraint patches

- We have used some constraint patches:
  ```
  $FOAM_SRC/finiteVolume/fields/fvPatchFields/constraint/\
  {cyclic,cyclicGgi,ggi,mixingPlane,overlapGgi}
  ```
- We will now have a look at how they should be specified in the cases.
- We will also see how they can be analysed.

The cyclic boundary condition for planar patches

- In `constant/boundary` file (from `turboPassageRotating`):

```
stator_cyclics
{
    type           cyclic;
    nFaces         100;
    startFace      31400;
    featureCos     0.9;
}
```

- The default cyclic patches must be planar, for the automatic determination of the transformation tensor.
- The faces must be ordered in a particular way:
  First half is one side and second half is the other side.
  Face `startFace+i` couples with face `startFace+nFaces/2+i`.
  The numbering is determined by the block definition, not by the faces list in `blockMeshDict`. Just make sure that each face definition is according to the rule "clockwise when looking from inside the block".
- Check your case set-up by modifying the cyclic debug switch: `cyclic 1;`

The cyclic boundary condition for non-planar patches

- In `constant/boundary` file (from `axialTurbine_ggi`):

```
GVCYCLIC
{
    type            cyclic;
    nFaces          240;
    startFace       11940;
    featureCos      0.9;
    transform       rotational;
    rotationAxis    (0 0 1);
    rotationCentre  (0 0 0);
    rotationAngle   -72; //Degrees from second half to first half
}
```

- Can of course also be used for planar patches.
- Still same requirement on face ordering.
- Check your case set-up by modifying the cyclic debug switch: `cyclic 1;`

The GGI and its alternatives

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 geometrically
  - `overlapGgi` couples two patches that cover the same sector angle
  - `cyclicGgi` couples two translationally or rotationally cyclic patches
  - `mixingPlane` applies an averaging at the interface.
- In all cases it is necessary to create `faceZone`s 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 `MRFSimpleFoam/axialTurbine_ggi` 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 `pimpleDyMFoam/axialTurbine` 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         5;
}
patch2: vice versa
```

- `rotationAxis` defines the rotation axis
- `nCopies` specifies how many copies of the segment that fill up a full lap (360 degrees)
- The pitch must be the same on both sides of the interface!

How to use the cyclicGgi interface - the boundary file

- See examples in the `axialTurbine` tutorials
- 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    72;
    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 mixingPlane interface - boundary file

- See example in the `MRFSimpleFoam/axialTurbine_mixingPlane` tutorial
- For two patches `patch1` and `patch2` (only `mixingPlane`-specific entries):

```
patch1
{
    type            mixingPlane;
    shadowPatch     patch2;      // See ggi description
    zone            patch1Zone;  // See ggi description
    coordinateSystem
    {
        type            cylindrical;
        name            mixingCS;
        origin          (0 0 0);
        e1              (1 0 0); //direction
        e3              (0 0 1); //axis
        degrees         false;   //Use radians
    }
    ribbonPatch
    {
        sweepAxis       Theta;
        stackAxis       R;
        discretisation  bothPatches;
    }
}
patch2: vice versa
```

- No `bridgeOverlap` option for `mixingPlane`

A special note on the boundary file for GGI interfaces

- The first patch of two GGI-coupled patches will be the 'master'.
- The definitions for the coupling will only be read by the 'master'.
- If the mesh is generated with `blockMesh`, the information can be set already in the `blockMeshDict` file, and it will be transferred to the `boundary` file only for the 'master'.

- The mixingPlane information in the `boundary` file is read from:

`$FOAM_SRC/foam/meshes/polyMesh/polyPatches/constraint/mixingPlane/mixingPlanePolyPatch.C`

How to use the mixingPlane interface - fvSchemes file

- The averaging at the `mixingPlane` interface is set for each variable in the `fvSchemes` file

```
mixingPlane
{
    default          areaAveraging;
    U                areaAveraging;
    p                areaAveraging;
    k                fluxAveraging; //Transported variable
    epsilon          fluxAveraging; //Transported variable
}
```

- `areaAveraging`: A pure geometric area averaging algorithm
  `fluxAveraging`: A mass-flow averaging algorithm
  `zeroGradient`: A regular zero gradient scheme - no coupling :-(
- See: `$FOAM_SRC/finiteVolume/fields/fvPatchFields/constraint/mixingPlane/mixingPlaneFvPatchField.C`
- Other schemes are under development

How to use the mixingPlane interface - fvSolutions file

- We currently use the non-symmetric `BiCGStab` linear solver for all variables, since it is the most stable at the moment
- New testing and developments are on the way
- Please contribute with your experiences

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;`
  - `type                mixingPlane;`
- The GGI patches must be put in `faceZones`
- 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
```

- The `mixingPlane` cases must currently be decomposed with

```
method patchConstrained;
```

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 %
```

The mixingPlaneCheck functionObject

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

```
functions
(
    mixingPlaneCheck
    {
        type mixingPlaneCheck; // 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:

```
Mixing plane pair (patch1, patch2) : 0.00470072382366 -0.00470072382373
Diff = -6.73142097618e-14 or 1.43199669427e-09 %
```

The mixingPlane interface - global controlDict

- Look at the `mixingPlane` patches and ribbons in the cylindrical coordinate system by setting in the global `controlDict`:

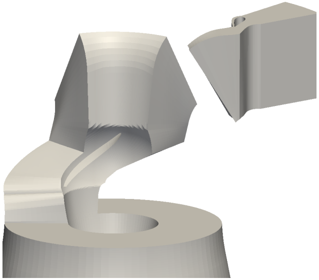
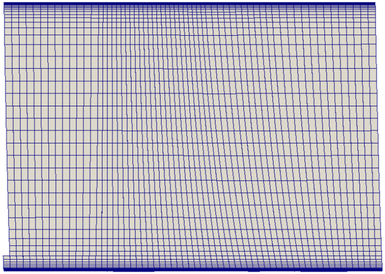
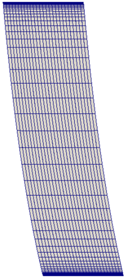
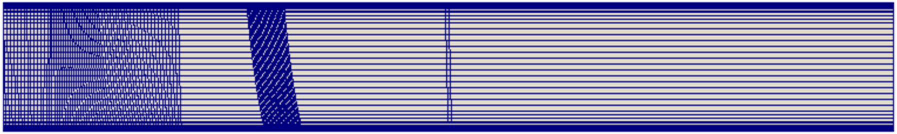
```
    mixingPlane                             2;
    MixingPlaneInterpolation                2;
```

- Creates:

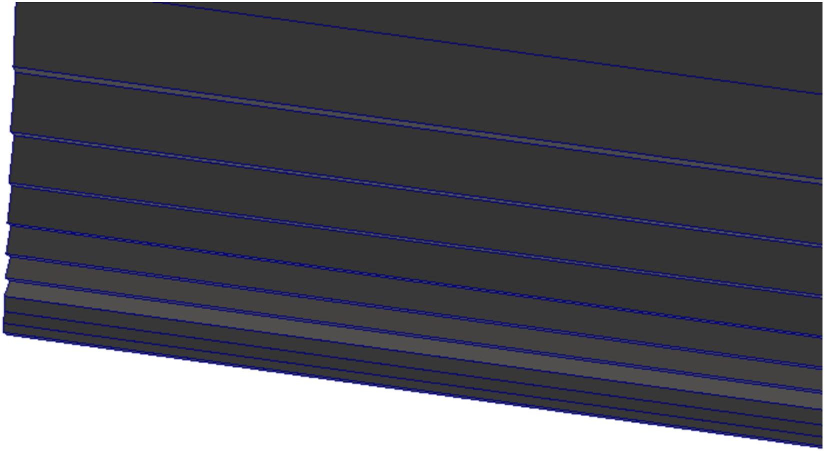
```
VTK/mixingPlaneMaster*
VTK/mixingPlaneRibbon*
VTK/mixingPlaneShadow*
```

- Load all at the same time in ParaView. You will most likely have to rescale the y-component (angle: $-\pi \leq \theta \leq \pi$ or $-180 \leq \theta \leq 180$).
- Make sure that they overlap as they should.
- Make sure that they are flat enough (no wrinkles or wavyness).
- Make sure that the ribbons resolve both sides of the interface.

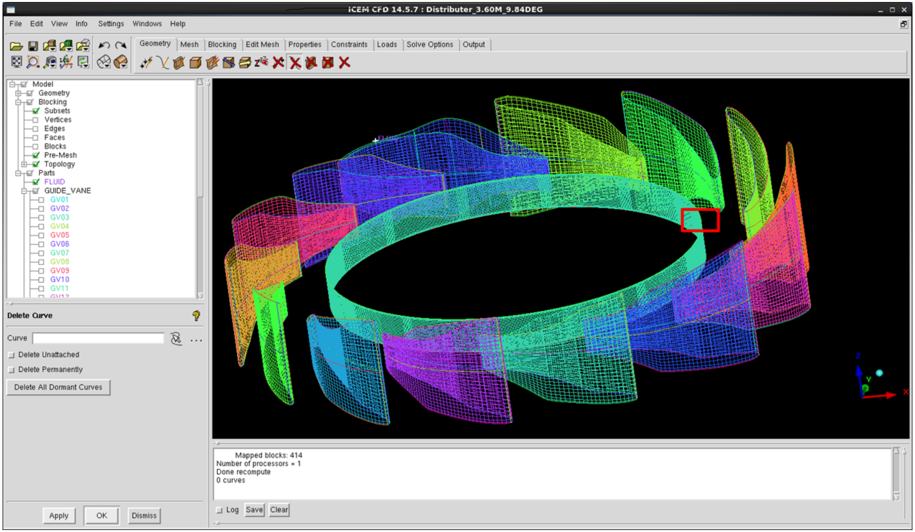
The mixingPlane interface - master, shadow, and ribbons

The mixingPlane interface - ribbon wrinkles due to mesh imperfection

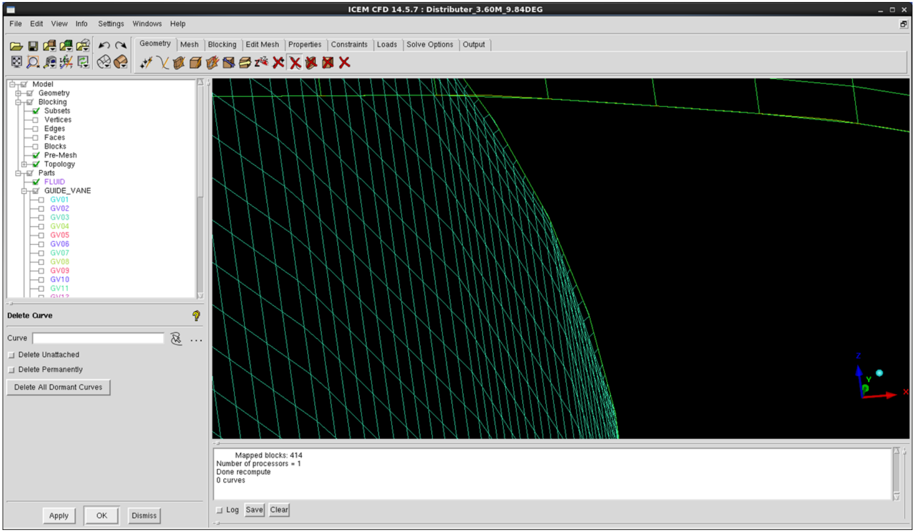


The mixingPlane interface - wavyness due to mesh imperfection



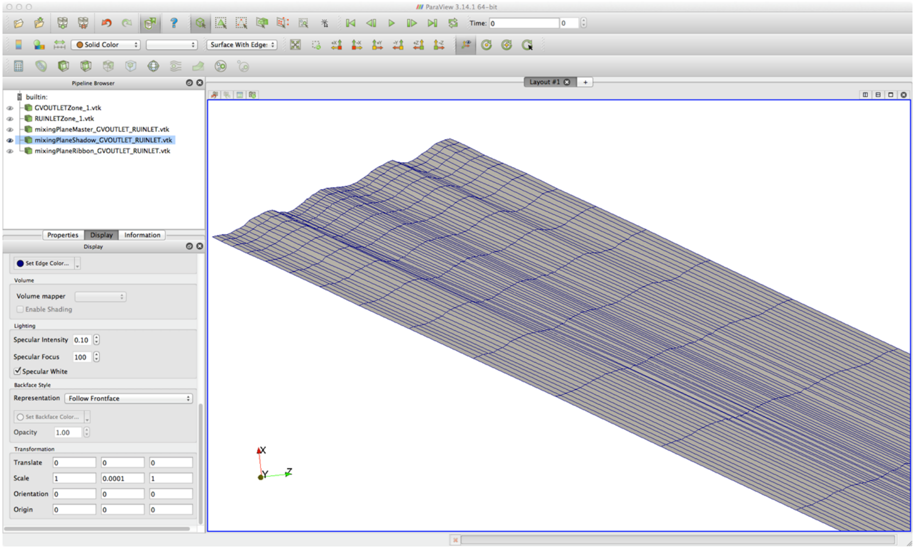
```
Mapped blocks: 414
Number of processors = 1
Done recompute
0 curves
```

The mixingPlane interface - wavyness due to mesh imperfection



```
    Mapped blocks: 414
Number of processors = 1
Done recompute
0 curves
```

The mixingPlane interface - wavyness due to mesh imperfection


ParaView 3.14.1 64-bit

Time: 0

Solid Color

Surface With Edges

Pipeline Browser

builtin:
- GVOUTLETZone_1.vtk
- RUINLETZone_1.vtk
- mixingPlaneMaster_GVOUTLET_RUINLET.vtk
- mixingPlaneShadow_GVOUTLET_RUINLET.vtk
- mixingPlaneRibbon_GVOUTLET_RUINLET.vtk

Properties | Display | Information

Display

Set Edge Color...

Volume

Volume mapper

Enable Shading

Lighting

Specular Intensity 0.10

Specular Focus 100

Specular White ✓

Backface Style

Representation: Follow Frontface

Set Backface Color...

Opacity 1.00

Transformation

Translate	0	0	0
Scale	1	0.0001	1
Orientation	0	0	0
Origin	0	0	0

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`)