PostPy

This is a code to read the *Fortran77* unformatted outputs 'grid_out' and 'flow_out' from *Multall-Open-20.9.exe.* Then, this data is post-processed and written as several multiblock *TecPlot* inputs called 'ParaView_TecPlotInterpreter_blades.dat' and 'ParaView_TecPlotInterpreter_passages.dat', which can be easily read by the open source code *ParraView.* The code should work with any axial machine. It has not been validated with the *mix[ed flow]* option of *Multall*, but if the outputs are compatible it should work or at least be easy to adjust.

Getting started: How to initiate a session

The Python code is organized in a directory containing '*main.py*' (Which currently does not contain anything, but can be used to automate the postprocessing process), and the folder 'Components'. Components contains:

- __init__.py
- _001_turbomachine_analysis.py
- Full_machine_ploter.py
- _002_blade_row.py
- _003_multiblock_solution.py
- _004_grid.py
- _005_flow_field.py

Each python file contains one class, correspondingly:

- Turbomachine
- PlotMachine
- BladeRow
- BlockCFD
- Coordinates
- FlowField

From the user point of view, the only class you directly interact with is **Turbomachine.** As said before, it is possible to automate processes through '*main.py*', but the tool has been conceived as an interactive one, so it is highly recommended to use it through direct user inputs in the 'Python Console'. To do so, just load the package and create your object:

```
In [3]: from Components import *
Backend TkAgg is interactive backend. Turning interactive mode on.
In [4]: machine = Turbomachine()
Reading Fortran binary raw data from "grid_out"...
Done, "grid_out" closed.
Reading Fortran binary raw data from "flow_out"...
Done, "flow_out" closed.
```

Note that calling the class '*Turbomachine()*' will automatically read the *Multall* outputs '*grid_out*' and '*flow_out*' with their default names. However, if you want to store several results in the same folder you can specify which files to use:

```
In [3]: machine_2 = Turbomachine('ThisIsAGrid','ThisIsTheFlow')
Reading Fortran binary raw data from "ThisIsAGrid"...
Done, "ThisIsAGrid" closed.
Reading Fortran binary raw data from "ThisIsTheFlow"...
Done, "ThisIsTheFlow" closed.
```

This command creates an object with all the things you will need to postprocess the simulation. When you call the class it automatically reads the input data, extracts the number of rows of the machine, and generates an attribute list called '*rows : list*' containing the row objects[1]. When these *row* objects are created with the class *BladeRow*, they automatically scan the row to extract the passage domain and the isolated blade geometry. These are stored in the attribute objects '*passage*_original' and '*blade*_original', both instances of the class *BlockCFD.* When the class *BlockCFD* is called it automatically loads the attribute objects '*grid*' (an instance of *Coordinates*) and '*flow_field*' (an instance of *FlowField*). These objects themselves compute the (x,y,z) coordinates of every point of the grid and a series of flowfield variables in every node of the grid. All of these processes are automatic and happen in less than a second! To sum up, when *Turbomachine* is called, it returns an object such that:

- machine = (*Turbomachine*)
  - plot = (*PlotMachine*)
  - rows = list
    - row[i] = (*BladeRow*)
      - original_passage = (*BlockCFD*)
        - grid = (*Coordinates*)
        - flow_field = (*FlowField*)
      - original_blade = (*BlockCFD*)
        - grid = (*Coordinates*)
        - flow_field = (*FlowField*)

The attribute object '*plot*' has not been introduced yet but it is convenient to show it here. This is only a basic description of the structure PostPy works with, each object has its own peculiarities.

Quick set up: Generate *ParaView* input

Create default ParaView input

Let's assume you only want to see 3D plots of the flowfield. In this case you just have to load *Turbomachine* and directly ask it to generate your .dat file calling the method

```
machine.gen_ParaView_input()
```

[1] Note that a stage usually has two rows, the code cannot distinguish between stages, only static or moving rows.

```
In [4]: machine.gen_ParaView_input()
Generating extra passage geometry and printing to .dat file, this might take couple minutes...
  2 blade row(s) found.
Updating geometry in row 0, this might take some time...
Updating geometry in row 1, this might take some time...
  Blade row 0: printing 2 passage(s).
   writing passage 1/2...
   writing passage 2/2...
  Blade row 1: printing 2 passage(s).
   writing passage 1/2...
   writing passage 2/2...
Done! your file is "ParaView_TecPlotInterpreter_passages.dat"

Generating extra blade geometry and printing to .dat file, this might take couple minutes...
  2 blade row(s) found.
  Blade row 0: printing 3 blades.
   writing blade 1/3...
   writing blade 2/3...
   writing blade 3/3...
  Blade row 1: printing 3 blades.
   writing blade 1/3...
   writing blade 2/3...
   writing blade 3/3...
Done! your file is "ParaView_TecPlotInterpreter_blades.dat"
```

Note that several things have happened now. First of all, 2 consecutive processes were triggered: the computation of the passage, and the computation of the blade. If you only want to generate one of those *ParaView* inputs you can directly call:

```
machine.gen_blades_ParaView()
machine.gen_passages_ParaView()
```

Note that blade generation is notably faster due to the smaller number of points (~50 times less than the passage). Another important process going on here is that the tool is generating *extra geometry*. This is not always done, as the code checks whether that geometry already exists or not[2]. This process checks each blade row (from the attribute *machine.rows[j_row]*) and creates several instances of the geometry and flow field. Note that 2 passages and 3 blades have been saved. These are the default settings and can be changed as follows.

Changing the number of instances

Each object row = (*BladeRow*) has an attribute ***N_instances***. This can be independently changed for every row of the machine. This same object also has the auto-computed attribute ***N_blades*** that informs the user about the number of blades of the real machine. This last attribute shouldn't be changed. For instance, to generate an output with 3 passages in the first row and 5 in the second one this is done:

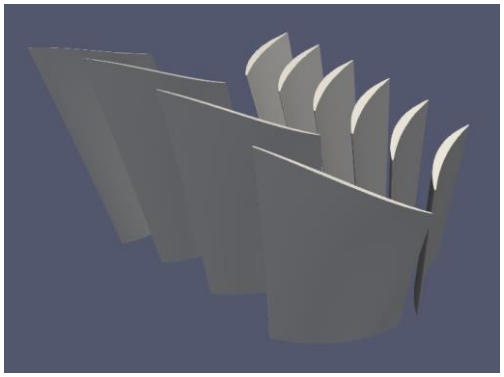
```
In [5]: machine.rows[0].N_instances = 3
In [6]: machine.rows[1].N_instances = 5
In [7]: machine.gen_ParaView_input()
```

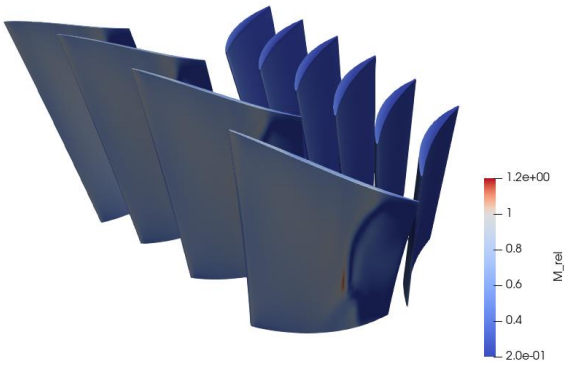
[2] See that when writing the blades no geometry updates were done!

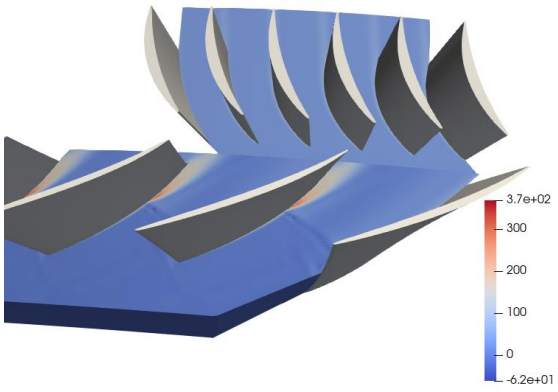
The geometry is only generated when required. This is an example of a script to plot a quarter of the annulus of the machine:

```
from Components import *

machine = Turbomachine()
N_rows = machine.N_rows

factor = 0.25
for i in range(N_rows):
    machine.rows[i].N_instances = int(np.ceil(machine.rows[i].N_blades
* factor))

machine.gen_ParaView_input()
```

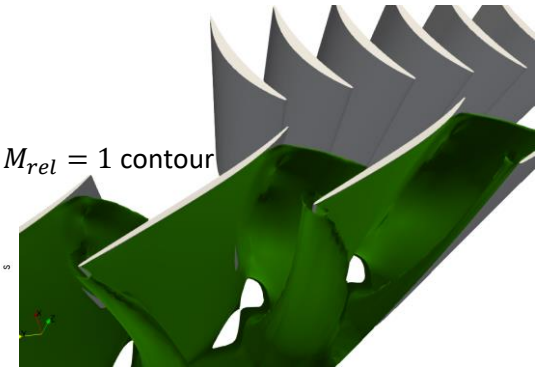
Conveniently, *Turbomachine* also offers the attribute *N_rows.* These are examples of the outputs in *ParaView*. Note that the blades also contain flow information.







$M_{rel} = 1$ contour



Using the tool to learn about a case

PostPy offers more than a direct translation to *ParaView*. All the flow field is stored in Python variables, so it is possible to do a fair amount of postprocessing within the tool itself.

Full machine plots

The more general analysis tools are in the attribute object *machine.plot.* This is an instance of the class *PlotMachine* with the methods:

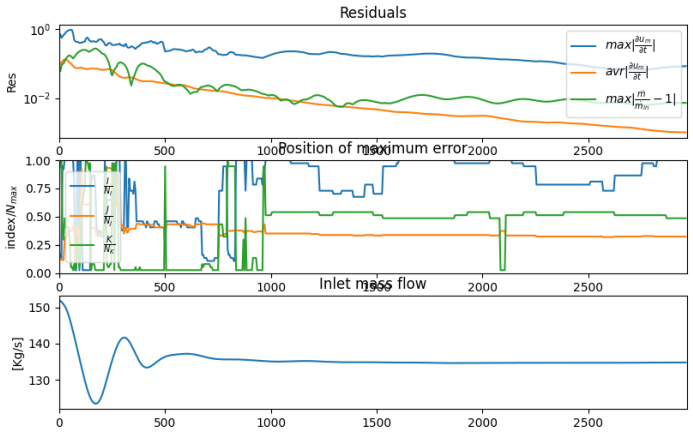
- convergence_history(file : str (optional input))
- variable_evolution_1D(variable : str, avr_type : str (optional))
- variable_evolution_2D(variable : str, avr_type : str (optional))
- variable_B2B(variable : str, level : float, levels : list)
- blades_contour(normal : str, level : float)
- blades_grid(normal : str, level : float)
- passage_contour(normal : str, level : float)
- passage_grid(normal : str, level : float)
- linear_cascade_blades(level : float)
- linear_cascade_contour(level : float)
- linear_cascade_grid(level : float)

Convergence history

It is possible to obtain the residual evolution of a run calling:

```
In [3]: machine = Turbomachine()
Reading Fortran binary raw data from "grid_out"...
Done, "grid_out" closed.
Reading Fortran binary raw data from "flow_out"...
Done, "flow_out" closed.
In [4]: machine.plot.convergence_history()
```

By default this function will look for the *Multall* file ‘stage.log’, but if the residuals are stored with another name it can be passed as an input to the method. This generates the plot:



The first subplot shows the residuals as evaluated by *Multall:* Maximum change of meridional velocity per numerical time-step, root mean square of this quantity (the actual convergence criteria) and the so-called continuity error (mass flow residual).

The second subplot shows the grid node where the maximum change in u_m was found. This is: the most problematic node in the domain. In this case we see it is towards the suction side (high

I index in a compressor, check *Multall* documentation), always in the inter-row space (medium J index, this should be correlated to the grid of the specific case), and initially at the root but then at mid-span (k index).

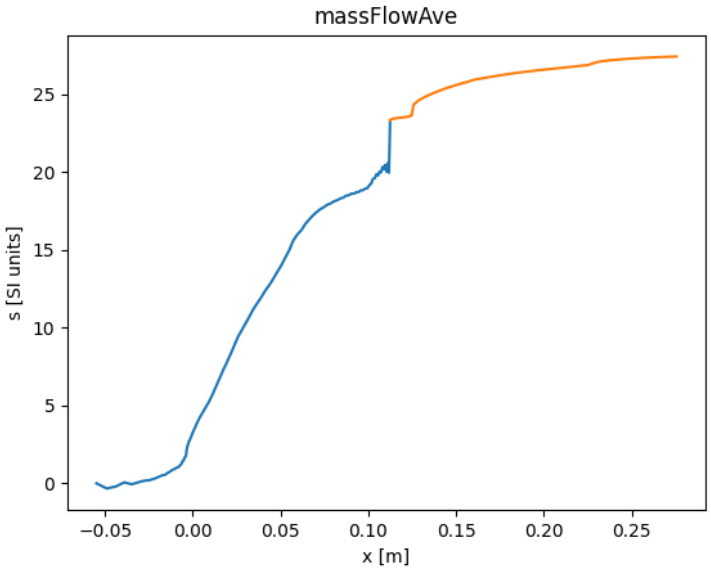
The last plot simply shows the evolution of the mass flow at the inlet of the computational domain.

1D fluid property plots

Calling *machine.plot.variable_evolution_1D('variable')* takes the variable specified[3] and makes the average of it at every grid plane $J = constant$. These surfaces are close to planes perpendicular to the meridional line of the machine, but they are not exact in general. The x coordinate is taken from the mid-span. By default the average is mass flow average, but there is an optional input that can take the values:

- 'massFlowAve'
- 'areaAve'

The implementation of these averages is discussed below. This is an example of a 1D plot (Note that each blade row is ploted in a different color!):



2D fluid property pitch-wise averaged plots

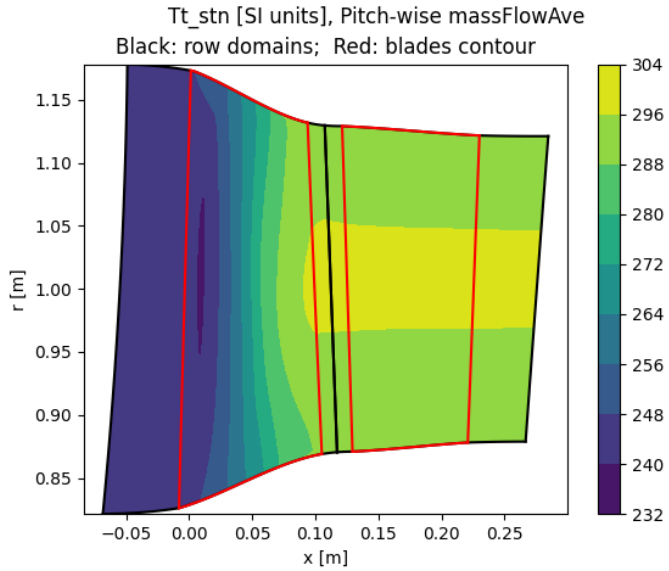
This function is very similar to the previous one. Calling *machine.plot.variable_evolution_2D('variable')* takes the variable specified[4] and makes the *pitch-wise* average of it at every grid plane $J = constant$. By default the average is mass flow average, but there is an optional input that can take the values:

- 'massFlowAve'
- 'areaAve'

The implementation of these averages is discussed below. This is an example of a 2D plot:

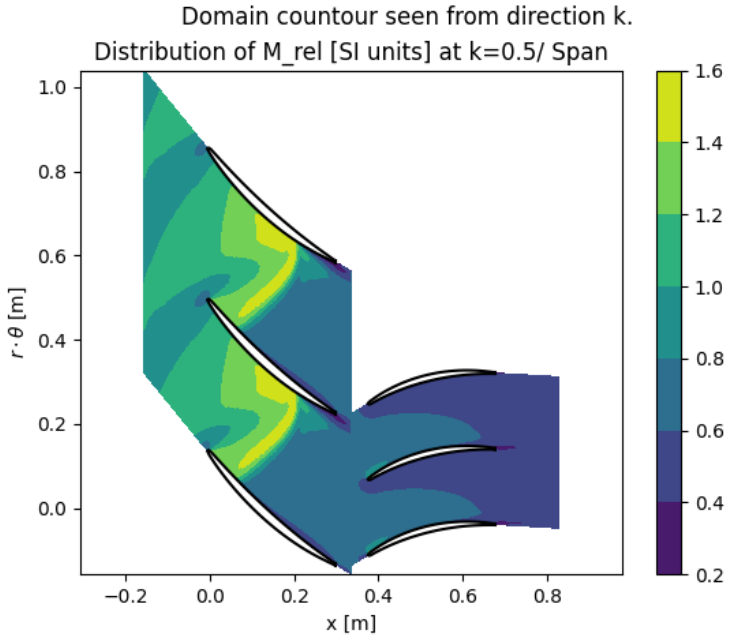
[3] Available varibales are stored in a dictionary and they are accessible by: *machine.rows[0].passage_original.flow_field.variables.keys().* It is also possible to read them (or even add more!) is the source code of **_005_flow_field.py**

[4] IBIDEM



2D blade to blade plane plots

These plots can be obtained by using the method *object.plot.variable_B2B()* with the compulsory inputs 'variable' (as before, a key in the variable dictionary), 'k' (The spanwise position given as a fraction of the local span) and 'levels', which is a list with the levels from the variable to plot. Python can be tricked to plot a single contour line at level l by maxing $levels = [l - \delta,\ l + \delta]$ with $\delta \ll l$. This method is building upon *BlockCFD. plot_B2B_process* method, and plots of a single passage can be done from this level with the method *BlockCFD.plot_B2B()* taking the same arguments. In this last cases 'levels' is an optional input. This is an example of a blade to blade plot:

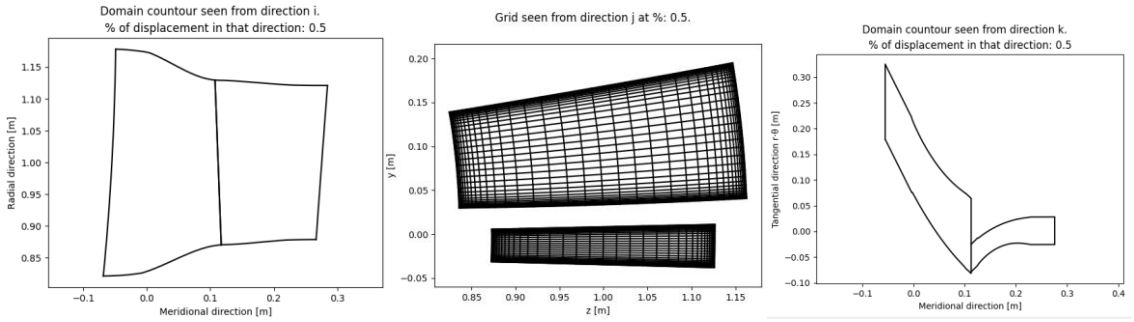


Geometry plots: Choosing 'normal' and 'level'

These methods are those reading two inputs, the string *normal* and the float *level.* The *normal* input is the name of the axis perpendicular to the plot, so that 'i' is looking to the flow-path, 'j' is looking to the annulus and 'k' is looking through the radial direction. *level* is a float input between 0 and 1. If it is 0 it plots the first grid surface *normal*=1, and if it is 1 it plots *normal=N_n*.

The method computes where the grid plane is and plots the closest to the level required, no grid interpolations are done.

There are 4 methods like this, two of them plot the blade object and the other two plot the passage. Contour simply sows the boundary of the objects while grid also includes the automatically generated grid by *Multall.* All of these methods only plot 1 passage, so they are quicker than the following ones.

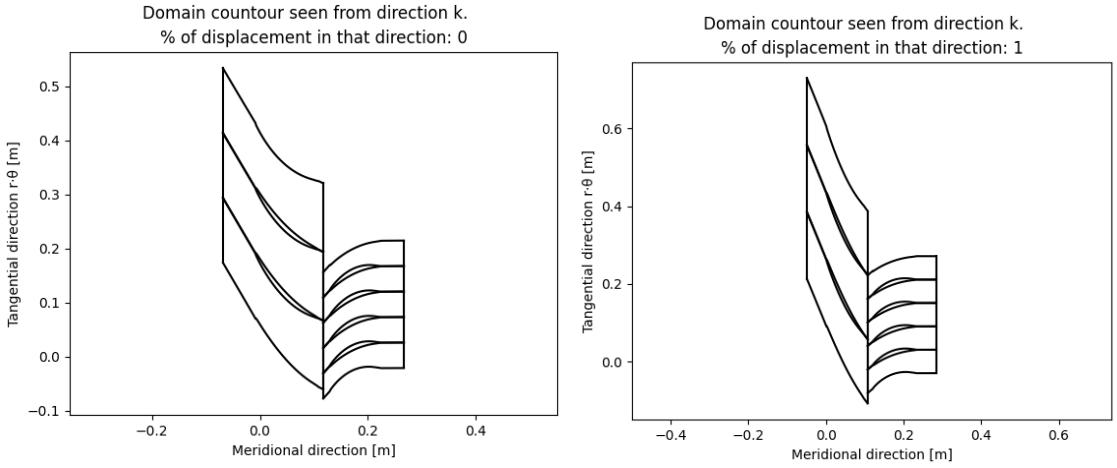


Geometry plots: Only choosing 'level'

These methods are intended to provide insight into the geometry of the full cascade. There are 3 of them:

- linear_cascade_blades(*level*)
- linear_cascade_contour(*level*)
- linear_cascade_grid(*level*)

They are plots of grid surfaces $k = contant$ at a height determined by *level*. Be aware that when the grid surface is far from $r = contant$ the geometry is highly distorted. However, it is fair to assume that these grid surfaces are close to stream surfaces so this distorted view is representative of the effective geometry the flow is going through. Be aware that the horizontal coordinate is the axial distance and not the distance along the grid surface.



Detail plots and numerical values

It is also possible to get a detailed view of the flow field in a precise axial (j_{grid}) position, or even the numerical value of an averaged property in a region of the grid plane. All of these actions are performed by methods living in the class *BlockCFD*, and they can be accessed by:

`Machine.rows[`*intNumber*`].passage_original.`*method*

Or

`Machine.rows[`*intNumber*`].blade_original.`*method*

The averages shown previously in this document are also performed by methods in this class and will be discussed in the following section.

Pitch-wise averaged plots

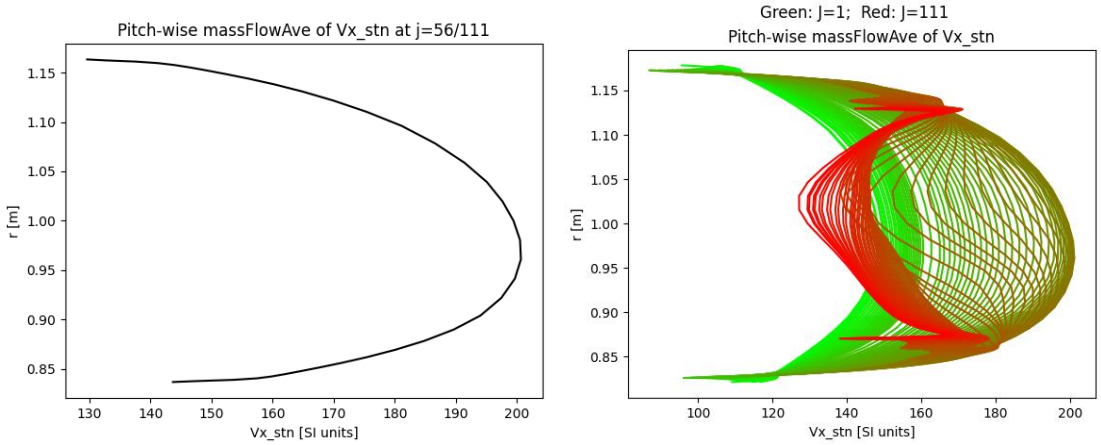
Here we are talking about two methods:

- plot_pitch_average(variable : str, j : float, avr_type : str (optional))
- plot_pitch_average_evolution(variable : str, avr_type : str (optional))

variable is a string calling one of the keys of the variable dictionary stored in *self.flow_field.variables*, and *avr_type* is an optional input specifying the averaging procedure to use. The default is set to mass flow average, and the possible methods are:

- 'massFlowAve'
- 'areaAve'
- 'mixedOutAve'

Note that the last method is only available at this level and not at full scale. These two methods plot the spanwise distribution of the variable selected. The first one requires an additional input (*j,* float between 0 and 1) specifying the axial position of the averaging plane within the row (0 is the upstream mixing plane, 1 is the downstream mixing plane). The second method simply plots all the $j = constant$ surfaces overlapped. A continuation a compressor rotor is shown:



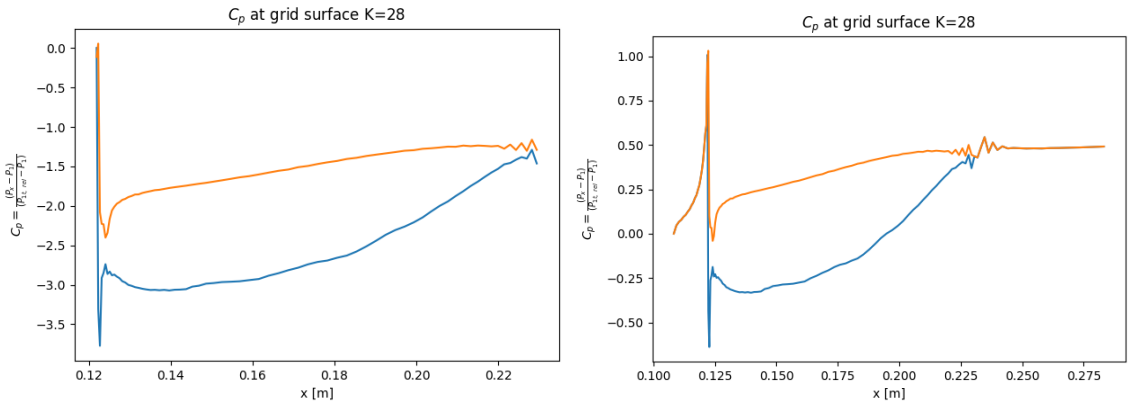
C_p plots and similar

There are another two methods called:

- plot_Cp(k : float)
- plot_variable_on_contour(variable : str, k : float)

That work in a very similar manner. As before, variable is a key for the variables dictionary and *k* is a float between 0 and 1 indicating the span-wise grid surface. What these methods do is taking a variable (in the case of the first one $C_p = \frac{P[i,\ :,\ k] - P[i,\ 0,\ k]}{P_t[i,\ 0,\ k] - P[i,\ 0,\ k]}$ is computed, so C_p with respect to the beginning of the domain, $j = 0$) and plotting it around the contour of the domain (so $i = 0$ and $i = N_i - 1$). Note that if this method is triggered in the blade object (*machine.rows[].blade_original.method()*) it will plot C_p with respect to the leading edge of the

blade, but if these methods are called in a passage (*machine.rows[].passage_original.method()*) it will contain the evolution upstream and downstream of the blade. Plots of compressor stator blade and passage C_p at 75% span[5]:



Obtaining numerical results

Finally, all of these plotting methods are based on the averaging methods themselves. There are three of them:

- get_area_average(*variable* : str, *level* : float, *i_lim* : list, *k_lim* : list)
- get_mass_flow_average(*variable* : str, *level* : float, *i_lim* : list, *k_lim* : list)
- get_mixed_out_average(*level* : float, *i_lim* : list, *k_lim* : list)

When called, all of these methods return a single float value. They only take averages in planes $j = constant$ (perpendicular to the meridional velocity), and the inputs are as follow:

- variable: A string with the key of the variable to average.
- level: a float between 0 and 1 giving the axial position of the averaging plane.
- i_lim: a 2 component list with floats between 0 and 1 indicating the fraction of pitch-wise plane that is taken. For pitch-wise averages take i_lim = [0, 1].
- J_lim: a 2 component list with floats between 0 and 1 indicating the fraction of span-wise plane that is taken for the average.

Note that these function allow to take an average of only a fraction of the passage.

Averaging methods

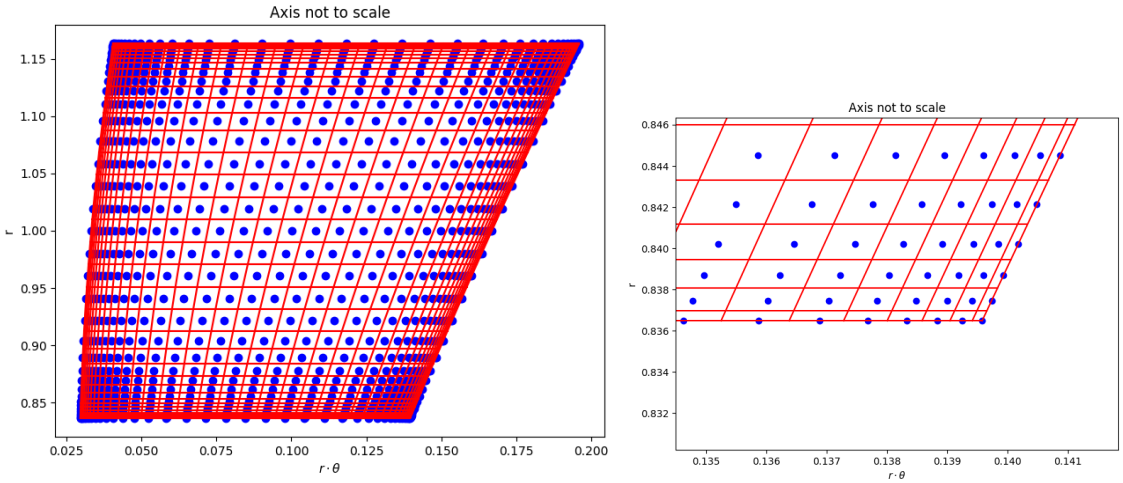
This subsection introduces the implementation of the averaging procedures. All of the three averaging methods (see above) are based on the method:

- Pre_average(*variable* : list, *level* : float, *i_lim* : list, *k_lim* : list)

This method takes as input a **list** of strings with variable names and the same inputs as the other averaging methods. Then it returns a **list** of *numpy.ndarray* containing the value of the selected variables in the grid nodes of the averaging plane defined by [i_lim, level, k_lim], AND the **numpy.ndarray** *dA*. This last output is basic to every averaging method as it is the "differential" area associated with each grid node in the averaging surface.

[5] The plots for the Blades will never close at the trailing edge. This is also seen in the ParaView files: *PostPy* is reading the blades between the coordinates *Multall* provides, which are the real ones, but to deal with TE separation *Multall* computes and adds a "fairing" at the trailing edge, so the "blade" in the CFD grid seems longer than the physical one.

dA is computed with a dual grid obtained with the mid-points between nodes of the original grid. The image shows the original nodes in blue and the computed dual grid in red. Each original node is associated to a dual grid face:



Then the area of each dual grid cell is computed as $(AverageVerticalEdges) \cdot (AverageHorizontalEdges)$ and it is asociated to the original node.

Area and mass averages

Each average procedure approximates the integrals with these areas. For area average:

$$\bar{\psi} = \frac{1}{A}\int \psi dA \approx \frac{1}{\sum_j dA_j}\sum_j (\psi_j dA_j)$$

And for mass flow average:

$$\bar{\psi} = \frac{1}{\dot{m}}\int \psi \rho u_x dA \approx \frac{1}{\sum_j (\rho_j u_{x,j} dA_j)}\sum_j (\psi_j \rho_j u_{x,j} dA_j)$$

Where j iterates over all the nodes in the averaging surface. Note that this is assuming that u_x is exactly perpendicular to the $j = constant$ grid plane, OR, that $j = constant$ is indeed a plane with constant angle γ with the longitudinal axis. For axial machines and the grids generated by *Multall* this a good approximation.

Mixed out state average

To get a mixed out state average the first step is to define a mixed out state. In this case, the state chosen is that one that a viscid fluid will achieve after evolving through a constant section duct with inviscid walls. In the mixed out state it is true that:

- There are no internal stresses in the fluid: $u_r = 0$, $u_m = u_{m0}$, $u_\theta = B \cdot r$ (solid body rotation).
- There is no heat transfer between streamlines ($T = T_0$)
- Pressure distribution is that needed for radial equilibrium ($\frac{\partial P}{\partial r} = \rho \frac{u_\theta^2}{r}$)

The last condition is integrated to achieve the pressure distribution[6]:

$$\frac{dP}{dr} = \rho\frac{u_\theta^2}{r} = \rho B^2 r = \frac{P}{R_g T_0}B^2 r \rightarrow \frac{dP}{P} = \frac{B^2}{R_g T_0} r dr \rightarrow \ln\left(\frac{P}{P_0}\right) = \frac{B^2}{R_g T_0}\frac{r^2 - r_0^2}{2}$$

$$P(r) = P_0 \exp\left[\frac{B^2}{R_g T_0}\frac{r^2 - r_0^2}{2}\right]$$

This is enough to obtain all the fluid properties, including total ones by using $T_t = T + \frac{u^2}{2C_p}$. The mixed out state is defined by 4 variables: $u_{m0},\ B,\ T_0$ and P_0. These are obtained by imposing that the mixed out state and the real flow have some integral values in common:

- Mass flow: $\int \rho u_m dA$ is conserved
- Total energy: $\frac{1}{\dot{m}}\int T_t \rho u_m dA$ is conserved
- Axial momentum: $\int (P + \rho u_m^2) dA$ is conserved
- Angular momentum: $\int (\rho u_m u_\theta r) dA$ is conserved[7]

These are 4 equations that are solved simultaneously to compute the mixed out state. The mixed out average method does not let the user select which variable to average: it always returns a list of floats containing $[P,\ P_t,\ T,\ T_t,\ u_m,\ u_\theta,\ s]$ in international system units. If the variable is not uniform in the mixed out state, the mass average is returned.

Available variables

This last section shows where extra flow variables can be computed as well as the definition of the current ones.

All the variables are contained in a dictionary in an object instance of the class *FlowField*. This object is called "flow_field" and it is an attribute of the class *BlockCFD.* When the class *Flowfield* is instanciated, it automatically triggers its method *gen_variables* and there is where the attribute *variables = (Dictionary)* is created. This is in the source code *_005_flow_field.py*.

More variables can be added by the user and *PostPy* will take care of passing them to *ParaView*[8], but bare in mind that more variables will increase the writing time as well as the size of the final file, and simple computations are easily done in *ParaView* interface.

Multall output provides the following variables:

- ρ
- ρu_x
- ρu_r
- ρu_θ
- $\rho e = \rho\left(C_v T + \frac{u^2}{2}\right)$

[6] This derivation is valid for every Mach number. Operating with B and r to get the circumferential Mach number it is possible to reach the limit for $M^2 \ll 1$ of $\frac{P}{P_0} = 1 + \frac{B^2}{R_g T_0}\frac{r^2 - r_0^2}{2}$, which is the solution for $\rho = \rho_0 = \frac{P_0}{R_g T_0} = constant$.

[7] Al lof the previous conservation equations are based on Greitzers' Internal Flow book, the last one is my addition to retain the angular momentum: there is no external torque applied during the mixing process.

[8] Avoid the names i, j, k, x, y ,z both in small and capital letters.

- $\omega\ [rad\ s^{-1}]$
- C_p and γ

The velocities are given in the stationary frame. It is trivial to obtain the velocities dividing by ρ. The velocities in the y and z direction are computed as $u_y = u_r \sin(\theta) + u_\theta \cos(\theta)$ and $u_z = u_r \sin(\theta) - u_\theta \cos(\theta)$ where θ is a coordinate from the grid (the angle of the radial vector measured from the vertical axis z). The velocities in the relative frame are obtained by $u_{\theta,rel} = u_{\theta,stn} - r \cdot \omega$, $u_{r,rel} = u_{r,stn}$ and $u_{x,rel} = u_{x,stn}$ and then applying the same transformation. An additional velocity (meridional) is computed as $u_m = \sqrt{u_r^2 + u_x^2}$, and with this the *pitch* angle of the flow is obtained as $\cos(\gamma) = \frac{u_x}{u_m}$. The *yaw* angle is computed both in stationary and relative frame as $\tan(\sigma) = \frac{u_\theta}{u_m}$. Lastly, the absolute ($\alpha$) and relative ($\beta$) flow angles are computed as $\tan(\alpha) = \frac{u_{\theta,stn}}{u_x}$ and $\tan(\beta) = \frac{u_{\theta,rel}}{u_x}$.

Thermodynamic properties are obtained from the internal energy: $T = \frac{1}{C_v}(e - E_k)$ with $E_k = \frac{1}{2}\left(u_{\theta,stn}^2 + u_r + u_x\right)$ the kinetic energy in the stationary frame of reference. Then the static pressure is computed as $P = \rho R_g T$ and total temperature in both stationary and relative frames of reference as $T_t = T + \frac{E_k}{C_p}$ with E_k being the kinetic energy in the correspondent reference system. The rest of total properties are obtained from $\left(\frac{\rho_t}{\rho}\right)^{\gamma-1} = \left(\frac{P_t}{P}\right)^{\frac{\gamma-1}{\gamma}} = \frac{T_t}{T}$. Mach number is computed based on the speed of sound $a = \sqrt{\gamma R_g T}$. Note that the static properties are independent of the reference system.

The entropy is computed as $s = C_p \ln\left(\frac{T}{T_{t,ref}}\right) - R_g \ln\left(\frac{P}{P_{t,ref}}\right)$, where the reference values are taken from the total conditions at the inlet of the machine (they are uniform).

Lastly, the total to total ($\beta_{tt} = \frac{P_t}{P_{t,ref}}$) and total to static ($\beta_{ts} = \frac{P}{P_{t,ref}}$) pressure ratios are also computed. Some geometrical variables, as r, θ and the grid surfaces (i, j, k) are also saved here because they might be useful while using *ParaView*.