DUST

Quick Start Guide

M. Tugnoli
D. Montagnani
www.dust-project.org

Contents

1 Introduction

DUST is a flexible solution to simulate the aerodynamics of complex, non-conventional aircraft configuration.

This quick start guide is meant to give new DUST users a quick way to start using it without excessive details on the implementation and on all the parameters, focusing on practical examples.

1.1 Workflow

DUST consists of three executables which are meant to perform preprocessing of the geometry, run the simulation in the desired condition and then postprocess the results obtained to gather the required meaningful data.

The most typical workflow with DUST is illustrated in figure 1. The geometry of the solid bodies, in form of cgns mesh or parametric directive, must be provided to the preprocessor, which performs preliminary operations and generates a binary geometry file.

Such file is provided, alongside the parameters for the simulation and the reference frames, to the solver, which executes the simulation and produces the complete results inside binary files.

The produced results contain the complete solution obtained during the simulation, in terms of distribution of singularities on body surfaces and wake. However it is difficult to obtain condensed, meaningful data from such results.

For this reason it is possible to specify a variety of different analyses to be performed by the postprocessor, which employs the global results to generate a series of different meaningful results, from visualization and flow fields to loads and loads distribution, in different formats.

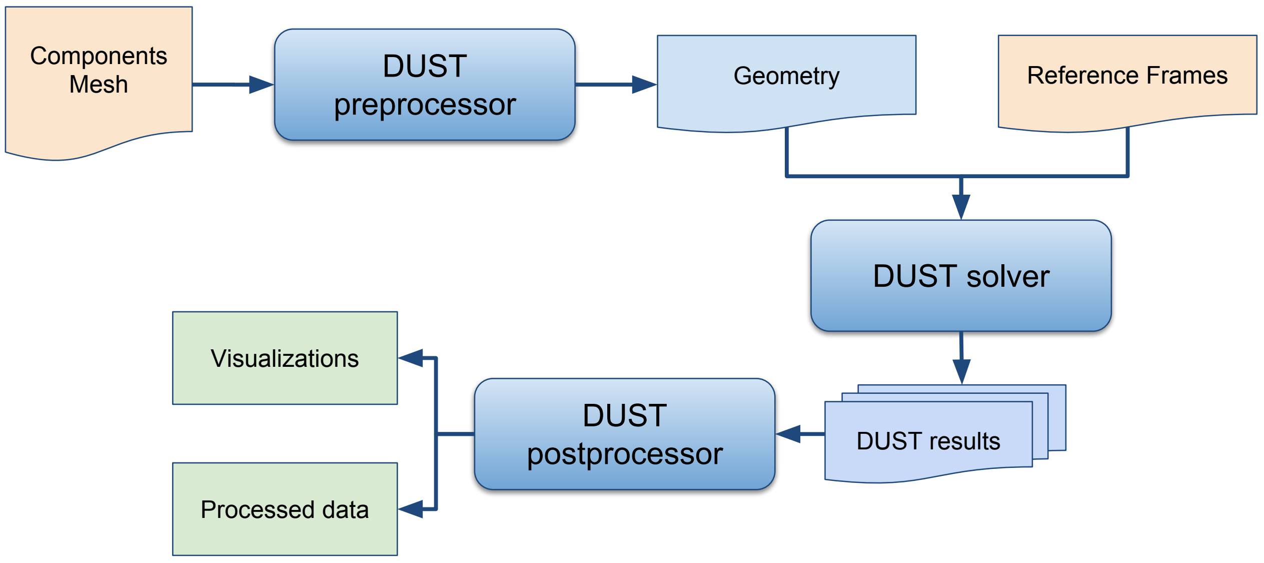

Figure 1: Description of the worflow with DUST

1.2 Files Format

All the input files, which are used to define parameters to the DUST executables, share the same flexible, free format text files structure.

The files are simple text files, without any requirement on the extension, however the extension `*.in` is recommended to distinguish the input files inside the case folders. The different executables automatically look for the input file `exe_name.in` where they are invoked.

Input files employ loosely a Fortran syntax, and comprise of a series of keyword assignments. The required keyword can generally be inserted in any order inside the input file, and should be written according to the following rules:

- All the parameters assignments are written with an equal sign, i.e. keyword = value, for any kind of value.
- Extra spaces and blank lines are ignored
- Comments are introduced with an exclamation mark "!", and can be introduced at the beginning of a line, or after a valid keyword assignment.
- Strings are introduced *as they are*, without quotes, and will be automatically stripped of leading and trailing spaces
- Integer and real numbers are introduced as usual, also with exponential notation
- Logicals are introduced as "T" for true and "F" for false
- Arrays can be introduced with the Fortran notation: contained between brackets and slashes and with elements separated by comas.
- Some keywords might be required to be contained inside a grouping keyword. It is a keyword which contains another set of keywords inside curly brackets.

Regarding the number, position and compulsoriness of the keywords, depending on the keyword:

- Some keyword can be required and compulsory, some can be optional, in the sense that a default value is employed if the keyword is not defined, and also can be required only in specific cases, according choices made in other parameters (e.g. if the user requires a simulation to be restarted from file, should provide the file name, otherwise the file name keyword can be neglected)
- Some keyword can be multiple: if the user should provide a list of a series of entities a keyword can be repeated several times
- Some keyword, contrary to all the other, must be placed in precise relative position (before/after) another keyword, due to the strong correlation of the parameters.

Meaningful errors are reported in case of wrong input.

All the files that contain data, input geometry and results, which are not meant to be interpreted by the user but are only for internal use, are written in binary hdf5 format.

The hdf5 format is a common opensource binary format which allows to store efficiently both in terms of size and I/O speed a variety of data in a single file.

The standard extension for hdf5 files is `.h5` and, even if it is not compulsory, should be added to the binary filenames in the parameter when specified.

As previously discussed in section 1.1 the binary files results are not meant to be interpreted by the user (even if the use of applications as hdfView can give a brief insight on some results). The postprocessor can interpret those results and output processed data readily employable by the user.

Most of the postprocessed results can be obtained in Tecplot binary format, which is compact and convenient. However Tecplot is a proprietary software available only under licence. For this reason all the results can also be produced in the standard vtk format for visualizations (that can be opened by a variety of programs, e.g. paraview or visit) and formatted ascii files for line plots.

Ascii files are also employed for analyses which output a single set of values (which are not meant to be plotted) and are convenient to be read inside automated execution loops.

1.3 Installation

For the current installation procedure refer to the file `INSTALL` in the root directory of the latest version of `DUST`.

2 Simple Wings Example

The first example considered is the simple wings one, which can be found in `examples/simple_wings` in the `DUST` distribution.

While it is not a particularly meaningful example, comprising just two squared wings, one oscillating and one stationary, it is useful to start introducing the key concepts in the `DUST` execution.

First it is recommended to copy the example folder in another location to be able to experiment without modifying the original files. Then is again recommended to copy or link the executables previously built in the example folder.

The first step is to execute the command

```
$ ./dust_pre
```

to generate the geometry defined in the default preprocessor input file `dust_pre.in`.

The input file contains:

Input file 1: dust_pre.in in simple wings example

```
CompName = right_wing
GeoFile = wing.in
RefTag = Right

CompName = left_wing
GeoFile = wing.in
RefTag = Left

FileName = geo_input.h5
```

which simply generates two different components, two equal wings, from the same geometry file, and assign them to two reference frames. Finally the name of the geometry file which will be generated is defined.

The wing is generated parametrically, from the file:

Input file 2: wing.in in simple wings example

```
MeshFileType = parametric
ElType = v

nelem_chord = 5
type_chord = uniform   ! uniform  cosineLE  cosineTE
starting_point = (/0.0,0.0,0.0/)
reference_chord_fraction = 0.25

! First section
chord = 0.4
twist = 8.0
airfoil = NACA0012

! First region
span = 5.0
sweep = 0.0
dihed = 0.0
nelem_span = 20
type_span = uniform

! Second section
```

```
chord = 0.4
twist = 8.0
airfoil = NACA0012
```

The generated geometry will consist of vortex lattice flat elements, since **ElType** is set to `v` (`p` for surface panels and `l` for lifting lines). The parametric generation works by defining a set of sections, with their chord, twist and shape, connected by regions with a certain span, sweep and dihedral. The generated geometry is particularly simple comprising only of a single region which creates a rectangular wing.

After running successfully the preprocessor it is possible to run the solver with:

```
$ ./dust
```

which again looks for input in the default file *dust.in*

Input file 3: dust.in in simple wings example

```
! Simulation names
basename       = ./Output/wings

! Timings ---------------------
tstart = 0.0
tend = 30.0
dt = 0.50
dt_out = 0.5
output_start = T

! reference values ---------------------
u_inf = (/0.1, 0.0, 0.0/)

! geometry input -----------------
GeometryFile = geo_input.h5
ReferenceFile = References.in

! Model parameters -------------------
FMM = F
Vortstretch = F
Diffusion = F

! wake parameters  -------------
n_wake_panels = 65
```

In the file is defined the name of the output, the details about the solution times, the reference value of the free stream velocity, where to read the (just generated) geometry and reference frames, some additional models are set to false for this simple case and finally the number of wake panels rows is set larger to the number of timesteps to ensure a full panel wake.

The reference frames file contains the information regarding the two reference frames at which the wings are attached:

Input file 4: References.in in simple wings example

```

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
Reference_Tag = Left
Parent_Tag = 0
Origin = (/0.0, -6.0, 0.0/)
Orientation = (/1.0,0.0,0.0, 0.0,1.0,0.0, 0.0,0.0,1.0/)
Multiple = F
```

```
Moving = F

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
Reference_Tag = Right
Parent_Tag = 0
Origin = (/0.0, 1.0, 0.0/)
Orientation = (/1.0,0.0,0.0, 0.0,1.0,0.0, 0.0,0.0,1.0/)
Multiple = F
Moving = T
Motion = {
 Pole = {
 Input       = position                ! position , velocity
 Input_Type  = simple_function         ! from_file , simple_function
 Function    = (/  0  ,  0  ,  0  /)   ! 0:uniform,1:sin
 !File        = xxx.dat                 ! File = xxx.dat (nt,4)
 Amplitude   = 0.00
 Vector      = (/ 0.0 , 0.0 , 0.0 /)
 Omega       = (/ 0.0 , 0.0 , 0.0 /)
 Phase       = (/ 0.0 , 0.0 , 0.0 /)
 Offset      = (/ 0.0 , 0.0 , 0.0 /)
!Position_0 = ...       ! Initial position: needed if the input is velocity
 }
 Rotation    = {
 Input       = position
 Input_Type  = simple_function         !
 Function    =  1                      ! 0:uniform,1:sin
 !File        = rot.in                 ! File = xxx.dat (nt,2)
 Axis        = (/ 0.0 , 1.0 , 0.0 /)
 Amplitude   = 0.1
 Omega       = 0.5
 Phase       = 0.0
 Offset      = 0.0
 !Psi_0       = 0.0      ! Initial position: needed if the input is velocity
 }
}
```

The reference frames are defined hierarchically starting from the reference frame `0`, from which all the others are defined. Each reference frame must have a parent frame on which is defined, any number of references can be generated and the whole tree of reference frames must be connected to the `0` one, which in turn cannot be re-defined.

Each reference frame is defined upon the parent by its origin offset in the parent reference frame and by its orientation, defined with the 9-component rotation matrix.

The definition of reference frames is also the way to impose a motion to the components connected to the reference frame. Each reference frame can perform an arbitrary roto-translation around a point and an axis, and a cascade of reference frame can describe a complex movement.

In this case a simple oscillation is defined.

Once the execution is terminated the solver should have generated a series of output files in the Output folder containing the results of the simulation, in terms of intensity of all the singularities of which the model in composed. These results should be further processed to obtain meaningful insights on the simulation.

By running

```
$ ./dust_post
```

the postprocessor executes the analyses described in the postprocessor input file

Input file 5: dust_post.in in simple wings example

```
basename = Postprocessing/post
data_basename = Output/wings

Analysis = {
  Name = v1
  Type = Viz
  Format = vtk !tecplot
  StartRes = 1
  EndRes = 61
  StepRes = 1
  Wake = T
  Variable = Vorticity
  Variable = Pressure
}

Analysis = {
Type = integral_loads
Name = loads_left
StartRes = 1
EndRes   = 61
StepRes  = 1
Format   = dat
Component = left_wing
Reference_Tag = 0
}

Analysis = {
Type = integral_loads
Name = loads_right
StartRes = 1
EndRes   = 61
StepRes  = 1
Format   = dat
Component = right_wing
Reference_Tag = 0
}

Analysis = {
Type = integral_loads
Name = loads_full
StartRes = 30
EndRes   = 61
StepRes  = 1
Format   = dat
Average = T
Component = left_wing
Component = right_wing
Reference_Tag = 0
}
```

Four different analyses are performed: first the results are transformed for visualization in vtk format, which can be opened by different visualization tools.

Then the loads are computed and outputted in formatted ascii files, first for the two separate wings, and then for the two wings together, averaged.

The loads are computed on the indicated components and projected on the reference frame indicated by the reference tag.

3 Flapping Wing Example

The second example concerns a slightly more complex case, showcasing a more complex parametrical geometry, a complex composite motion and more types of postprocessing.

The running procedure is identical to the previous example: run preprocessor, solver and postprocessor in this order.

The input file for the wing geometry contains

Input file 6: ParamWing.in in flapping wing example

```
MeshFileType = parametric
ElType = p

starting_point = (/0.0,0.0,0.0/) !Beginning of the extruded geometry
reference_chord_fraction = 0.0   !fraction of chord at which the extrusion line
                                 !is placed

nelem_chord = 8                  !number of elements in chord direction
                                 !(must be one for all the component)

type_chord = cosineLE            !type of element distribution in chord
                                 ! uniform , cosine , coisneLE , cosineTE

! First section (ideally a wing rib)
chord = 1.0           !chord length
twist = 10            !chord twist (deg)
airfoil = NACA3018    !Airfoil profile (only 4 digits naca supported at the
                      ! moment, dufferent profiles could be loaded from file)

! First region (ideally a wing bay)
span = 2.5            !span of the region
sweep = -10.0         !sweep angle (deg)
dihed = 10.0          !dihedral angle (deg)
nelem_span = 20       !number of elements in span region
type_span = uniform   !distribution of elements in span

! Second section
chord = 0.6
twist = 0.0
airfoil = NACA2012

! Second region
span = 2.0
sweep = 5.0
dihed = -3.0
nelem_span = 16
type_span = uniform

! Third section
chord = 0.2
twist = -2.0
airfoil = NACA0012
```

The wing is generated in a very similar way with respect to the previous example, however surface panels are employed (`ElType=p`) which means that the whole shape of the airfoil is used to create a three dimensional geometry rather just the camber line for vortex lattices. In this case more sections and regions are employed to generate a more complex shape.

On the execution parameters again there is no particular differences with respect to the previous example, however the reference frame system is much more complicated, comprising of a chain of moving and not moving reference frames with the wing attached only at the end of this chain, in order to generate a complex motion. The references are:

Input file 7: References.in in flapping wing example

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
Reference_Tag = Plunge
Parent_Tag = 0
Origin = (/0.0, 0.0, 0.0/)
Orientation = (/1.0,0.0,0.0, 0.0,1.0,0.0, 0.0,0.0,1.0/)
Multiple = F
Moving = T
Motion = {
 Pole = {
  Input      = position
  Input_Type = simple_function
  Function   = (/  0   ,  0   ,  1   /)
  Amplitude  = 0.5
  Vector     = (/ 0.0 , 0.0 , 1.0 /)
  Omega      = (/ 1.2 , 0.62832 , 0.62832 /)
  Phase      = (/ -1.5708 , 0.0 , 0.0 /)
  Offset     = (/ 0.0 , 0.0 , 0.0 /)
 }
 Rotation   = {
  Input      = position
  Input_Type = simple_function
  Function   =  0
  Axis       = (/ 0.0 , 1.0 , 0.0 /)
  Amplitude  = 0.0
  Omega      = 2.0
  Phase      = 0.0
  Offset     = 0.0
  Psi_0      = 0.0
 }
}

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
Reference_Tag = Offset1
Parent_Tag = Plunge
Origin = (/0.0, 0.3, 0.0/)
Orientation = (/1.0,0.0,0.0, 0.0,1.0,0.0, 0.0,0.0,1.0/)
Multiple = F
Moving = F

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
Reference_Tag = Flap
Parent_Tag = Offset1
Origin = (/0.0, 0.0, 0.0/)
Orientation = (/1.0,0.0,0.0, 0.0,1.0,0.0, 0.0,0.0,1.0/)
Multiple = F
Moving = T
Motion = {
 Pole = {
  Input      = position
  Input_Type = simple_function
  Function   = (/  0   ,  0   ,  0  /)
  Amplitude  = 1.0
  Vector     = (/ 0.0 , 0.0 , 0.0 /)
  Omega      = (/ 1.2 , 0.62832 , 0.62832 /)
  Phase      = (/ -1.5708 , 0.0 , 0.0 /)
  Offset     = (/ 0.0 , 0.0 , 0.0 /)
 }
 Rotation   = {
  Input      = position
  Input_Type = simple_function
  Function   =  1
  !File       = Omega_1.dat
  Axis       = (/ 1.0 , 0.0 , 0.0 /)
  Amplitude  = 0.349
  Omega      = 0.62832
  Phase      = 3.14
  Offset     = 0.0
  Psi_0      = 0.0
 }
}

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
Reference_Tag = Pitch
Parent_Tag = Flap
Origin = (/0.0, 0.0, 0.0/)
Orientation = (/1.0,0.0,0.0, 0.0,1.0,0.0, 0.0,0.0,1.0/)
Multiple = F
Moving = T
Motion = {
 Pole = {
  Input      = position
  Input_Type = simple_function
  Function   = (/  0   ,  0   ,  0  /)
  Amplitude  = 1.0
  Vector     = (/ 0.0 , 0.0 , 0.0 /)
  Omega      = (/ 1.2 , 0.62832 , 0.62832 /)
  Phase      = (/ -1.5708 , 0.0 , 0.0 /)
  Offset     = (/ 0.0 , 0.0 , 0.0 /)
 }
 Rotation   = {
  Input      = position
  Input_Type = simple_function
  Function   =  1
  !File       = Omega_1.dat
  Axis       = (/ 0.0 , 1.0 , 0.0 /)
  Amplitude  = 0.17453
  Omega      = 0.62832
  Phase      = -1.5708
  Offset     = 0.0
  Psi_0      = 0.0
 }
}
```

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
Reference_Tag = Wing
Parent_Tag = Pitch
Origin = (/0.2, 0.0, 0.0/)
Orientation = (/1.0,0.0,0.0, 0.0,1.0,0.0, 0.0,0.0,1.0/)
Multiple = F
Moving = F
```

The other novel inputs are contained in the postprocessor directives:

Input file 8: dust_post.in in flapping wing example

```
basename = Postprocessing/post
data_basename = Output/flapwing

Analysis = {
  Name = v1
  Type = Viz

  Format = vtk

  StartRes = 1
  EndRes = 61
  StepRes = 1
  Wake = T

  Variable = Vorticity

}

Analysis = {
  Type = integral_loads
  Name = loads1

  StartRes = 1
  EndRes = 61
  StepRes = 1

  Format = dat

  CompName = wing
  Reference_Tag = 0
}

Analysis = {
  Type = integral_loads
  Name = loads2

  StartRes = 1
  EndRes = 61
  StepRes = 1

  Format = dat

  CompName = wing
  Reference_Tag = Wing
}
```

```
Analysis = {
  Name = flow1
  Type = flow_field

  StartRes = 1
  EndRes = 61
  StepRes = 1

  Format = vtk !tecplot, vtk

  Variable = Velocity

  Nxyz = (/30,1,30/)
  Minxyz = (/-2.0, 3.5, -3.0/)
  Maxxyz = (/5.0, 3.5, 3.0/)

}

Analysis = {
  Name = flow2
  Type = flow_field

  StartRes = 1
  EndRes = 61
  StepRes = 1

  Format = vtk !tecplot, vtk

  Variable = Velocity

  Nxyz = (/1,30,30/)
  Minxyz = (/2.0, 0.0, -3.0/)
  Maxxyz = (/2.0, 5.0, 3.0/)

}
```

The standard visualization of the geometry surface and wake is produced, alogside the integral loads in the absolute and wing refrence frame. Finally flow fields are generated: the velocity is computed on a grid of equally spaced points in order to visualize the flow fields. In this case two planes are employed, one cutting across the wing and one cutting across the wake. If a number of points different from 1 is employed in all the directions a volume is generated.

Pay attention that computing the influence of all the surface and wake elements on a great number of points might result in a computational effort greater of the one needed to run the simulation, since the main advantage of a panel based method is exactly not to need to compute quantities on the field but only on surfaces.

4 Robin Example

This example is more complete and complex than the previous ones, and many more features of DUST are employed in this case.

This case represent a robin standardized helicopter fuselage under a rotor in forward flight. The surface of the fuselage comes from a CGNS mesh (available in the example folder) and it is modelled with surface panels, while the blades of the rotor are modelled with lifting lines, which require tabulated airfoil data, provided also in this case.

The geometry generation is similar to the previous cases, the lifting lines blades are generated in the same way as the parametric elements already discussed, while the surface panels for the fuselage are simply loaded from the pre-existing mesh.

The geometry generation should also not be necessary for the purpose of the present run since it is meant to be restarted from a previous provided state and the geometry is also loaded. However all the required data to generate the geometry from scratch are provided.

The input file for the solver is:

Input file 9: dust.in in robin example

```
basename       = ./Output/robin

! free-stream conditions ---------------
u_inf   = (/ 27.0 , 0.0 , 0.0 /)

! Time ----------------------------------
tstart = 0.0
tend = 0.630063
dt = 0.000751577
dt_out = 0.000751577
output_start = T

! geometry -----------------------------
GeometryFile = geo_input.h5

! restart ------------------------------
restart_from_file= T
restart_file = ./Output/starting_res.h5
reset_time = F

! reference frames ---------------------
ReferenceFile = ./References.in

! Model parameters ---------------------
VortexRad = 0.03
FMM = T
Vortstretch = T
Diffusion = T
PenetrationAvoidance = T

! wake parameters ----------------------
n_wake_panels = 1
n_wake_particles = 100000
particles_box_min = (/ -2.0, -2.0, -7.0/)
particles_box_max = (/  6.0,  2.0,  1.0/)

! octree parameters--------------------
```

```
BoxLength = 4
NBox = (/2,1,2/)
OctreeOrigin = (/-2.0, -2.0, -7.0/)
NOctreeLevels = 6
MinOctreePart = 5
MultipoleDegree = 2

! lifting lines solver-----------------
LLtol = 1.0e-4
LLdamp = 5.0
```

While the first inputs are similar to the previous examples, the first new feature introduced is the restart from a previous result, with its keywords: the logical swithch to enable the restart, the result file from which to restart the simulation and whether to restart the time from **`tstart`** or keep on with the time of the loaded result. In this way it is possible to restart a previous simulation for a longer time, ore use a simulation result as the starting point for another simulation. Keep in mind that the geometry motion is still linked to the movement of the reference frames, which is defined starting from **`tstart`**.

Next the parameter **`VortexRad`** is introduced, which sets the radius of the particles in the wake. Since in this simulation the number of panels in the wake is set to one (cannot be less than one) all the panels in the wake after the first row are converted into vortex particles. The advantages of vortex particles are mainly related to the higher stability when wakes interact with bodies and other wakes. Following, always related to the particles wake, the Fast Multipole Method (FMM) is enabled. The fast multipole allows, on the base of an underlying octree grid, to approximate the interaction between distant clusters of particles through polynomial expansions, which reduces the computational burden of the calculation of N particles interaction from $\mathcal{O}(N^2)$ to $\mathcal{O}(N)$.

Then some other particles related models are enabled, to activate the different terms of the vorticity equation which regulates the evolution of the particles vorticity: vortex stretching and diffusion. The penetration avoidance is an additional model to ensure the non penetration of the particles into solid bodies.

In the wake parameters, as previously discussed, only one wake panel in imposed, while is set the number of maximum vortex particles. This is done to allocate the correct amount of memory for all the particles, if the number of particles exceed the imposed maximum, the simulation is stopped (however it is possible to re-start the simulation with an increased maximum number of particles).

The particles box is a parallelepiped aligned with the base reference frame defined by its two minimum and maximum vertices, and is used to define the area of interest in which the particles exist. I the particles evolve outside this box are considered too far to introduce a significant contribution, and are discarded. A too big box leads to the need of tracking a larger amount of particles, while a box too close to the body might lead to errors due to the fact that particles are cancelled in an area in which they still exert a significant interaction with the body itself.

The octree parameters are the parameter which affect how the FMM is executed: the octree must be defined in it largest layer, composed of a series of cubic boxes of a certain lenght in each direction, starting from a certain vertex. The possibility to define a certain number of boxes in each direction is left only to allow the definition of non-cubic overall octree boxes, however the number of boxes at the highest level should be kept minimum, the refinement of the grid is automatically performed in the splitting of each octree box in another 8 cubes, and then another 8 cubes and so on for **`NOctreeLevels`** levels of splitting. A higher number of levels will reduce the close interactions between particles (which are expensive) by having smaller cells, but will increase the computational overhead of the fast multipole expansion computations, and vice versa. A sweet spot for each case should be found, but the number of levels should not exceed 7 since higher subdivisions will most likely fill up the memory even of the most capable computers. The minimum number of particles for each cell defines the threshold at which in one octree cells short range interactions are computed, if less particles are present in the cell the upper level (bigger) cell is considered. This number should be small but high enough to ensure a decent computation of the multipole expansions in the cell.

Finally some parameters are introduced to control the solution of lifting lines. While the equations for all the other elements is linear, and so their solution is obtained by a linear system solver , the lifting lines, due to their dependence on lookup tables are subject to a non linear equation, which must be solved iterratively with a fixed-point iterations method. The parameter define the tolerance for the convergence of the method and the damping for the under-relaxation of the iterations. A higher damping leads to slower but more robust convergence, while a smaller damping leads to faster convergence but might lead to oscillation. The value of the parameter should be assessed case by case.