DUST User Manual

Davide Montagnani
Matteo Tugnoli
Federico Fonte
Alberto Savino
Alessandro Cocco
Andrea Colli

July 5, 2022

Disclaimer

The present document refers to the state of the code developed by Politecnico di Milano for the collaboration with A3 Airbus at the present date. All the specifications herein contained are subject to partial, substantial or complete modifications in the near future. The aim of the present document is to provide an early reference for the code usage and testing during the development phase and should not be considered in any form an official document provided by Politecnico di Milano.

Contents

Acronyms

List of the acronyms employed in this document:

fmm Fast Multipol Method

SP Surface (3D) Panel

VL Vortex Lattice

LL Lifting Line

VP Vortex Particle

VPM Vortex Particle Method

Chapter 1

Introduction

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

It 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.

1.1 Workflow

The most typical workflow with DUST is illustrated in figure 1.1. 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.

1.2 Input 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.

An example of text input file is presented in file 1.1. 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:

- Keywords use the snake_case naming convention
- All the parameters assignments are written with an equal sign, i.e. keyword = value, for any kind of

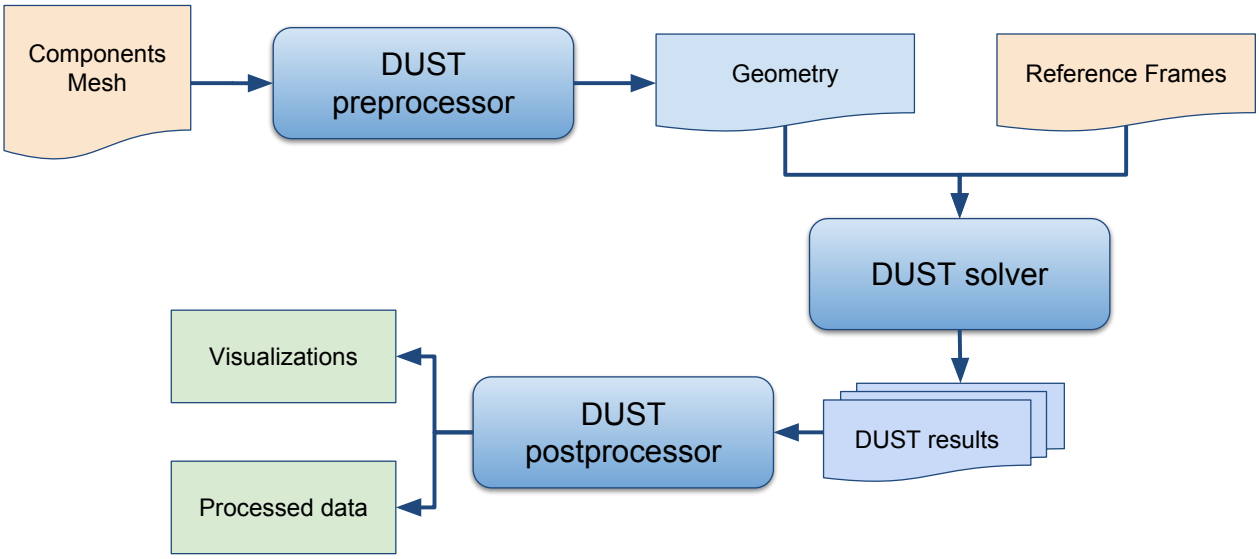

Figure 1.1: Description of the worflow with DUST

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.

input file 1.1: example input file

```
! comments are introduced with an exclamation mark as in Fortran
example_string = a_string
example_integer = 59
example_real = 67.84 !all comments even inline are ignored
example_logical = T !as well as all the extra spaces.

!empty lines are ignored
example_array = (/ 0.5, 2.3e-3, 5.67329/)

example_multiple = 1.3
example_multiple = 7.2 !some keywords can have multiple values

!some keyword can require to be grouped inside another grouping keyword
example_grouping = {
       grouped_var_int = 5 !indenting can be helpful, but is not required
   grouped_var_logical = F
```

```
}
```

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.

All the details on the parameters of the single files will be provided in the description of each input file in the following chapters.

1.3 Internal Binary Files format

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.

1.4 Output Files format

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.

Chapter 2

Building DUST

The building mechanism for DUST is currently under evolution in order to become more automated and reliable, however it is not in final configuration.

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

Chapter 3

DUST Preprocessor

The DUST preprocessor is used to generate the geometrical components required to model the surfaces of the analysed body. It gathers the meshes of all the components required for the complete model, process them when necessary and generates all the parametrically specified components.

The preprocessor is executed simply invoking the executable `dust_pre` in the desired folder. The input file containing all the required informations for the execution of the preprocessor must be passed as argument to the command call. If not provided explicitly the preprocessor automatically tries to read the default input file `dust_pre.in`.

```
dust_pre input_file_name.in
```

Command 3.1: Preprocessor command looking for input file `input_file_name.in`

```
dust_pre
```

Command 3.2: Preprocessor command looking for default input file `dust_pre.in`

3.1 Input file

The input file of the preprocessor specifies the geometrical components required for the model, their name and the reference system to which they will be attached.

The format is the same as all the other input files, as already specified in section1.2.

input file 3.1: dust_pre.in

```
comp_name = rotor
geo_file = blade.in
ref_tag = Hub01

comp_name = wing
geo_file = wing.in
ref_tag = Root01

tol_se_wing = 0.001
inner_product_te = -0.5
file_name = ./geo_input.h5
```

An example of the preprocessor input file is presented in file 3.1, while the detailed parameters are:

- `comp_name`: *required:* at least one. *multiple:* yes. *type* string.

  This is the name assigned to the geometrical component. Will be mainly used by the user afterwards to specify postprocess analyses.

- `geo_file`: *required:* at least one. *multiple:* yes. *position:* must be after `comp_name`. *type:* string.

  Indicates the auxiliary input file which must be provided with the details on the mesh of the component. It is a string with the path to the location of the file. In case of relative path the path is relative to the location in which the preprocessor was called.

- `ref_tag`: *required:* at least one. *multiple:* yes. *position:* must be after `geo_file`. *type:* string

  Provides a string tag which indicates the reference frame to which the geometrical component is attached. Must correspond to one of the reference frames that will be provided to the solver.

- `tol_se_wing`: *required:* no. *multiple:* no. *default:* 0.001 *type:* real.

  Define the global tolerance at which the mesh node are merged to identify the open trailing edges. More details in section 3.5.

- `inner_product_te`: *required:* no. *multiple:* no. *default:* -0.5 *type:* real.

  Define the global tolerance for the identification of trailing edges using the inner product of the normals.

- `file_name`: *required:* yes. *multiple:* no. *type:* string.

  Define the name of the binary file which contains the geometry, to be used by the solver.

As discussed in section 1.3 the file, being a DUST internal file is in binary hdf5 format. The use of the .h5 extension is not compulsory in a unix environment, but is recommended to distinguish internal binary files from input/output files in different formats.

3.2 Geometry file

The geometry file defines the parameters required to generate the geometry mesh of a component. Different components, with different component names and attached to different reference frames, can have the same geometry and hence use the same geometry file. A geometry file can be also use to define a parametric geometry, but details on that case will be described in section 3.3. file 3.2 is an example of a geometry file for a non parametric geometry.

input file 3.2: geo_file.in

```
mesh_file = component-mesh.cgns
mesh_file_type = cgns
el_type = p

mesh_symmetry = F
symmetry_point = (/0.0, 0.0, 0.0/)
symmetry_normal = (/0.0, 1.0, 0.0/)

mesh_mirror = F
mirror_point = (/0.0, 0.0, 0.0/)
mirror_normal = (/1.0, 0.0, 0.0/)

tol_se_wing = 0.001
inner_product_te = -0.5

proj_te = T
proj_te_dir = parallel
proj_te_vector = (/1.0, 0.0, 0.0/)
```

```
suppress_te = F

section_name = cgns_comp_1
section_name = cgns_comp_2

offset = (/0.0, 0.0, 0.0/)
scaling_factor = 1.0
```

The detailed parameters of the geometry file are:

- `mesh_file`: *required:* yes (if not parametric). *multiple:* no. *type:* string.

  name of the file containing the mesh.

- `mesh_file_type`: *required:* yes. *multiple:* no. *type:* string

  type of the mesh. Valid options at the moment are `cgns` for cgns, `parametric` for parametrically generated meshes and `basic` for ascii input meshes. The last is only for development purposes.

- `el_type`: *required:* yes. *multiple:* no. *type:* character.

  type of the elements of the mesh. `p` stands for surface panels to model solid bodies, `v` stands for vortex lattice elements used to model flat surfaces, `l` stands for lifting lines used to produce a 1D model of a lifting surface (only for parametric input) and finally `a` stands for actuator disk, to produce a simple model of a rotor (only for parametric input).

- `mesh_symmetry` *required:* no. *multiple:* no. *default:* false. *type:* logical.

  Choose to reflect the mesh around a point and a direction. Useful to produce full meshes out of symmetrical half models. Keeps both the original and the symmetrical part.

- `symmetry_point`: *required:* only if `mesh_symmetry` is true. *multiple:* no. *default:* (0.0, 0.0, 0.0). *type:* real array, length 3.

  point around which to reflect the mesh.

- `symmetry_normal`: *required:* only if `mesh_symmetry` is true. *multiple:* no. *default:* (0.0, 1.0, 0.0). *type:* real array, length 3.

  Direction in which to reflect the mesh.

- `mesh_mirror` *required:* no. *multiple:* no. *default:* false. *type:* logical.

  Choose to mirror the mesh around a point and a direction. Same as `mesh_symmetry` but does not keep both the original, i.e. the mesh is not doubled.

- `mirror_point`: *required:* only if `mesh_mirror` is true. *multiple:* no. *default:* (0.0, 0.0, 0.0). *type:* real array, length 3.

  point around which to mirror the mesh.

- `mirror_normal`: *required:* only if `mesh_mirror` is true. *multiple:* no. *default:* (0.0, 1.0, 0.0). *type:* real array, length 3.

  Direction in which to mirror the mesh.

- `tol_se_wing`: *required:* no. *multiple:* no. *default:* 0.001 *type:* real.

  Tolerance in trailing edge merging of nodes. Override for the single component the value defined (or the default value) in the main input file to the preprocessor. Warning: the default value is not employed if the same parameter is defined (and not left default) in file 3.1.

- `inner_product_te`: *required:* no. *multiple:* no. *default:* -0.5 *type:* real Tolerance for the identification of trailing edges using the inner product of the normals. Override for the single component the value defined (or the default value) in the main input file to the preprocessor. Warning: the default value is not employed if the same parameter is defined (and not left default) in file 3.1.

- `proj_te`: *required:* no. *multiple:* no. *default:* false *type:* logical.

  Force the projection of the trailing edge in a specific direction

- `proj_te_dir`: *required:* if `proj_te` is true. *multiple:* no. *type:* string.

  Choose in which way to project the trailing edge. If it is `parallel` the trailing edge direction will be forced in the direction given in `proj_te_vector`, while if it is `normal` the trailing edge will be projected in a plane normal to `proj_te_vector`.

- `proj_te_vector`: *required:* if `proj_te` is true. *multiple:* no. *type:* real array, length 3.

  vector to specify the direction declared in `proj_te_dir`

- `suppress_te`: *required:* no. *multiple:* no. *default:* false *type:* logical.

  Suppress the trailing edge from the component: even if a trailing edge is found, it is suppressed and the component will not release a wake from a trailing edge during the simulation.

- `section_name` *required:* no. *multiple:* yes. *type:* string

  To be used only with cgns meshes: specify only a subset of all the sections (geometrical components) available in the cgns file to be loaded and employed as geometry. If no `section_name` is specified, all the sections of the cgns file will be employed.

- `offset` *required:* no. *multiple:* no. *default:* (0.0, 0.0, 0.0) *type:* real array, length 3.

  offset to apply to the loaded points. Allows to move the coordinates of the loaded points of the vector specified.

- `scaling_factor` *required:* no. *multiple:* no. *default:* 1.0 *type:* real.

  Scaling factor to apply to the loaded points. Allows to scale the coordinates of the loaded points of the specified factor.

  offset and scaling are applied in the following order:

$$\mathbf{r} = Scaling(\mathbf{r}_{loaded} + offset)$$

3.2.1 Basic Mesh

The basic mesh input is an extremely simplified way to provide a surface mesh providing just the the position of points and point-element connectivity.

When employing the basic way of input, the preprocessor expects two ascii files containing the points location and connectivity. If the parameter `mesh_file` was set to `/path/to/mesh/` the preprocessor expects the point locations to be in `/path/to/mesh/rr.dat` and the connectivity in `/path/to/mesh/ee.dat` [1]

The points coordinates must be provided in the `rr.dat` file as three floating point numbers each row, for a total of n_p rows, where n_p is the number of points in the mesh. The three numbers represent the x, y and z coordinates of each point (in the local reference frame).

The connectivity must be provided in the `ee.dat` file as four integer numbers each row, for a total of n_e rows, where n_e is the total number of elements. Each row represents an element and the integers are the indices of the points forming the element, starting from one, in the order in which they were provided in the `rr.dat` file.

Elements can be both quadrangular or triangular, in case of triangular elements use a 0 as the fourth index. The order in which the points are listed in each element defines it normal, according to the right-handed screw rule. Neighbouring elements must not have opposing normals, and in three dimensional surface panels the normals direction should be outward from the body.

[1] Note that the preprocessor just add the suffix `ee.dat` and `rr.dat` to the basename, one can have different basic mesh files in a folder by giving them different prefixes and providing the path with the prefix to the input file, e.g `mesh_file = /path/to/mesh/file_name_` will look for `/path/to/mesh/file_name_rr.dat` and `/path/to/mesh/file_name_ee.dat`

Two examples of basic mesh generation via scripting. The MATLAB/OCTAVE scripts to generate the rectangular wing and the cylindrical ellipsoidal tank shown in figure 3.1 are shown in this paragraph as an example of basic mesh generation along with the `rr.dat`, `ee.dat` files generated by the scripts to be read by DUST as input files.

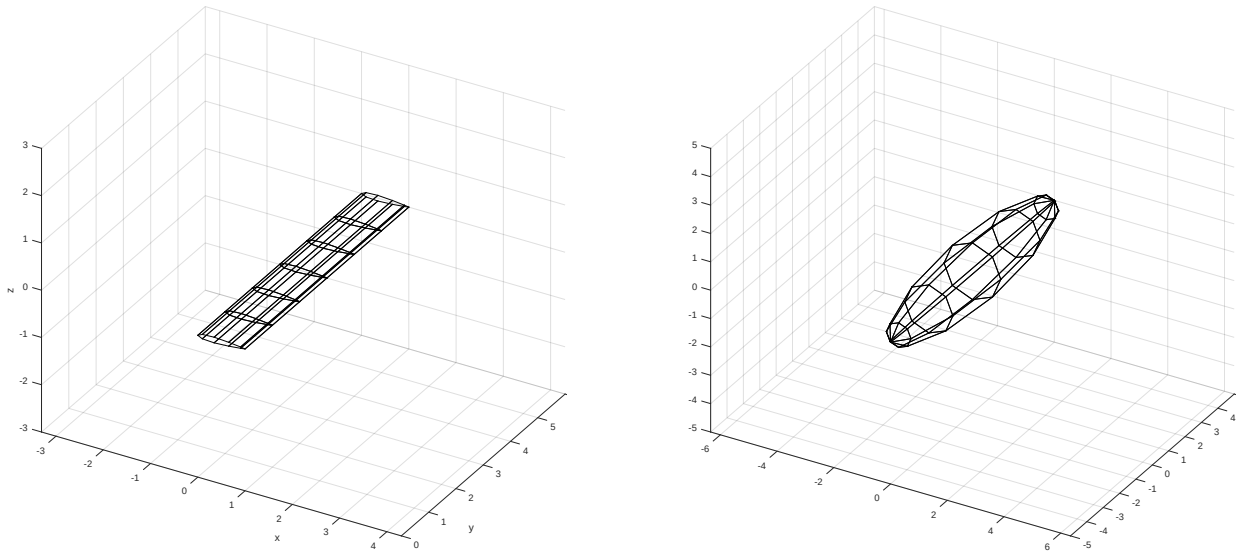

Figure 3.1: Components defined by means of basic mesh generations in the examples: wing and tank.

The following script 3.3 relies on the function `setAirfoil4()` to define the two-dimensional airfoils and builds the basic input files for DUST of a rectangular wing with open tips.

input file 3.3: `wing.m`

```
% ========================================================================== %
% build the geometry and connectivity for a rectangular wing
% ( y-axis identifies the spanwise direction )
%
% rr(3,n_points): array of the coordinates of the points of the surface
% ee(4,n_elems ): array of the note-to-elem connectivity
% ========================================================================== %
clear all ; clc ; close all

% === params ===
chord = 1.0 ;
span = 6.0 ;
n_span_el = 10 ;
n_chord_el = 5 ; % n_chord_el = n. elems on the chord

%> the tot. number of elems for a 3dpanel component ('p' in dust) is
n_elems = (n_span_el)*(2*n_chord_el) ;

%> the tot. number of points for a 3dpanel component ('p' in dust) is
n_points_per_sec = 2*n_chord_el+1 ;
```

```
n_points = (n_span_el+1) * n_points_per_sec ;

%> vector of spanwise coord.s of the airfoil sections: uniform spacing, here
y_sec = linspace(0,span,(n_span_el+1)) ;

% === rr array ===
rr = zeros(n_points,3) ;

for i_sec = 1 : n_span_el+1

  %> define the 2d airfoil, NACA 4-digit airfoils, NACA-MPSS
  % setAirfoil ( M , P , SS , chord , n_chord_el , other parameters ... )
  [ x , z ] = setAirfoil4( 0 , 0 , 12 , chord , n_chord_el , ...
                        0.0 , 0.0 , 0.0 , 0.0 , 0 ) ;
  %> update rr array
  rr(1+(i_sec-1)*n_points_per_sec:i_sec*n_points_per_sec,:) = ...
                              [ x ; y_sec(i_sec)*ones(size(x)) ; z ]' ;
end

% === ee array ===
ee = zeros(n_elems,4) ; ie = 0 ;

for i_sec = 1 : n_span_el
  for i_p = 1 : n_points_per_sec-1

    ie = ie + 1 ;
    ee(ie,:) = [ i_sec *n_points_per_sec+i_p , ...
                 i_sec *n_points_per_sec+i_p+1 , ...
                (i_sec-1)*n_points_per_sec+i_p+1 , ...
                (i_sec-1)*n_points_per_sec+i_p ] ;
  end
end

% === check the connectivity with the Patch plot routine ===
% patch has rr,ee arrays as inputs --> easy way to check the rr,ee arrays
figure ; grid on ; axis equal
patch('Vertices',rr,'Faces',ee,'FaceColor','none')

% === save to .dat file in ascii format ===
%save('wing_rr.dat','rr','-ascii')
%save('wing_ee.dat','ee','-ascii')
dlmwrite('wing_rr.dat',rr, 'delimiter','\t')
dlmwrite('wing_ee.dat',ee, 'delimiter','\t')
```

The files 3.4 and 3.5 are respectively the `rr.dat` and `ee.dat` files produced by the `wing.m` script.

input file 3.4: `rr.dat` created by `wing.m`

```
   1.0000 0 -0.0013
   0.9045 0 -0.0139
   0.6545 0 -0.0409
   0.3455 0 -0.0596
   0.0955 0 -0.0460
       0 0 0
   0.0955 0 0.0460
   0.3455 0 0.0596
   0.6545 0 0.0409
   0.9045 0 0.0139
   1.0000 0 0.0013
   1.0000 1.0000 -0.0013
   0.9045 1.0000 -0.0139
   0.6545 1.0000 -0.0409
   0.3455 1.0000 -0.0596
   0.0955 1.0000 -0.0460
       0 1.0000 0
   0.0955 1.0000 0.0460
   0.3455 1.0000 0.0596
   0.6545 1.0000 0.0409
   0.9045 1.0000 0.0139
   1.0000 1.0000 0.0013
   1.0000 2.0000 -0.0013
   0.9045 2.0000 -0.0139
   0.6545 2.0000 -0.0409
   0.3455 2.0000 -0.0596
   0.0955 2.0000 -0.0460
       0 2.0000 0
   0.0955 2.0000 0.0460
   0.3455 2.0000 0.0596
   0.6545 2.0000 0.0409
   0.9045 2.0000 0.0139
   1.0000 2.0000 0.0013
   1.0000 3.0000 -0.0013
   0.9045 3.0000 -0.0139
  ...
   0.9045 6.0000 0.0139
   1.0000 6.0000 0.0013
```

input file 3.5: `ee.dat` file created by `wing.m`

```
   12 13 2 1
   13 14 3 2
   14 15 4 3
   15 16 5 4
   16 17 6 5
   17 18 7 6
   18 19 8 7
   19 20 9 8
   20 21 10 9
   21 22 11 10
   23 24 13 12
   24 25 14 13
   25 26 15 14
   26 27 16 15
   27 28 17 16
   28 29 18 17
   29 30 19 18
   30 31 20 19
   31 32 21 20
   32 33 22 21
   34 35 24 23
   35 36 25 24
   36 37 26 25
   37 38 27 26
   38 39 28 27
   39 40 29 28
   40 41 30 29
   41 42 31 30
   42 43 32 31
  ...
   69 70 59 58
   70 71 60 59
   71 72 61 60
   72 73 62 61
   73 74 63 62
   74 75 64 63
   75 76 65 64
   76 77 66 65
```

The following script 3.6 builds the basic input files for DUST of a ellipsoidal tank with cylindrical sections.

input file 3.6: `tank.m`

```
% ========================================================================== %
% build the geometry and connectivity of a tank with ellipsoidal shape
% ( y-axis identifies the axial direction )
%
% rr(3,n_points): array of the coordinates of the points of the surface
% ee(4,n_elems ): array of the note-to-elem connectivity
%
% /o----o--------o----o\
% /_o----o--------o----o_\
% o< -- . -- . -- . -- . >o -- . -- axis
% \ o----o--------o----o /
% \o----o--------o----o/
```

```
% sec.1 .2 .3 .4
% ========================================================================== %
clear all ; clc ; close all

% === params ===
a = 5.0 ; b = 1.0 ; % major and minor semi-axis of the cylindrical ellipsoid
n_axial_sec = 5 ; % n. cylindrical sections, excluding the tank extreme points
n_radial_el = 8 ; % n_chord_el = n. elems on the chord

%> the tot. number of elems for a 3dpanel component ('p' in dust) is
n_elems = n_radial_el * (n_axial_sec+1) ;

%> the tot. number of points for a 3dpanel component ('p' in dust) is
n_points_per_sec = n_radial_el ;
n_points = ...
  n_points_per_sec*n_axial_sec + 2 ; % + 2: for the extreme points

%> vector of spanwise coord. and radius of the tank sections
% here, ellipsoid with Chebychev spacing on the major axis
dth = pi/(n_axial_sec+2) ;
th_vec = linspace( pi-dth , dth , n_axial_sec ) ;
y_sec = a * cos(th_vec) ;
r_sec = b * ( 1.0 - y_sec.^2 / a^2 ).^0.5 ;

%> extreme points y coord
y01 = - a ; y02 = a ;

% === rr array ===
rr = zeros(n_points,3) ; ir = 0 ;

for i_sec = 1 : n_axial_sec
  for i_p = 1 : n_points_per_sec

    %> update counter and find the th angle describing the polar coord
    ir = ir + 1 ;
    th = 2*pi*(i_p-1) / n_points_per_sec ;

    rr(ir,:) = [ r_sec(i_sec)*cos(th) , y_sec(i_sec) , r_sec(i_sec)*sin(th) ] ;

  end
end
%> update rr with the first and last point
rr = [ [ 0.0, -a, 0.0 ] ; rr(1:ir,:) ; [ 0.0, a, 0.0 ] ] ;

% === ee array ===
ee = zeros(n_elems,4) ; ie = 0 ; % Initialisation to zero

%> first TRIA sector (last col remains equal to 0)
for i_p = 1 : n_points_per_sec

  ie = ie + 1 ;
  ee(ie,1:3) = [ 1 , 1+i_p , 1+mod(i_p,n_points_per_sec)+1 ] ;

end

%> inner QUAD sectors
```

```
for i_sec = 1 : n_axial_sec-1
  for i_p = 1 : n_points_per_sec

    ie = ie + 1 ;
    ee(ie,:) = int32([ 1+(i_sec-1)*n_points_per_sec+ i_p , ...
                1+ i_sec *n_points_per_sec+ i_p , ...
                1+ i_sec *n_points_per_sec+ mod(i_p,n_points_per_sec)+1 , ...
                1+(i_sec-1)*n_points_per_sec+ mod(i_p,n_points_per_sec)+1 ]);
  end
end

%> first TRIA sector (last col remains equal to 0)
for i_p = 1 : n_points_per_sec

  ie = ie + 1 ;
  ee(ie,1:3) = int32([1+(n_axial_sec-1)*n_points_per_sec+i_p , ...
               n_points , ...
               1+(n_axial_sec-1)*n_points_per_sec+mod(i_p,n_points_per_sec)+1]);

end

% === save to .dat file in ascii format (OCTAVE) ===
save('tank_rr.dat','rr','-ascii')
save('tank_ee.dat','ee','-ascii')
% === save to .dat file in ascii format (MATLAB) ===
dlmwrite('tank_rr.dat',rr, 'delimiter','\t')
dlmwrite('tank_ee.dat',ee, 'delimiter','\t')


% === check the connectivity with the Patch plot routine ===
% patch has rr,ee arrays as inputs --> easy way to check the rr,ee arrays
%> patch does not allow 0 indices in ee -> substitute 4th zero el with 3rd el
for ie = 1 : n_elems
 if ( ee(ie,4) == 0 ) ; ee(ie,4) = ee(ie,3) ; end
end

figure ; grid on ; axis equal
patch('Vertices',rr,'Faces',ee,'FaceColor','none')
```

The files 3.7 and 3.8 are respectively the `rr.dat` and `ee.dat` files of the ellipsoidal tank, generated by the `tank.m` script 3.6.

input file 3.7: `rr.dat` file created by `tank.m`

```
     0 -5.0000 0
 0.4339 -4.5048 0
 0.3068 -4.5048 0.3068
 0.0000 -4.5048 0.4339
-0.3068 -4.5048 0.3068
-0.4339 -4.5048 0.0000
-0.3068 -4.5048 -0.3068
-0.0000 -4.5048 -0.4339
 0.3068 -4.5048 -0.3068
 0.8467 -2.6602 0
 0.5987 -2.6602 0.5987
 0.0000 -2.6602 0.8467
-0.5987 -2.6602 0.5987
-0.8467 -2.6602 0.0000
-0.5987 -2.6602 -0.5987
-0.0000 -2.6602 -0.8467
 0.5987 -2.6602 -0.5987
 1.0000 0.0000 0
 0.7071 0.0000 0.7071
 0.0000 0.0000 1.0000
-0.7071 0.0000 0.7071
-1.0000 0.0000 0.0000
-0.7071 0.0000 -0.7071
-0.0000 0.0000 -1.0000
 0.7071 0.0000 -0.7071
 0.8467 2.6602 0
 0.5987 2.6602 0.5987
 0.0000 2.6602 0.8467
-0.5987 2.6602 0.5987
-0.8467 2.6602 0.0000
-0.5987 2.6602 -0.5987
-0.0000 2.6602 -0.8467
 0.5987 2.6602 -0.5987
 0.4339 4.5048 0
 0.3068 4.5048 0.3068
 0.0000 4.5048 0.4339
-0.3068 4.5048 0.3068
-0.4339 4.5048 0.0000
-0.3068 4.5048 -0.3068
-0.0000 4.5048 -0.4339
 0.3068 4.5048 -0.3068
     0 5.0000 0
```

input file 3.8: `ee.dat` file created by `tank.m`

```
 1 2 3 0
 1 3 4 0
 1 4 5 0
 1 5 6 0
 1 6 7 0
 1 7 8 0
 1 8 9 0
 1 9 2 0
 2 10 11 3
 3 11 12 4
 4 12 13 5
 5 13 14 6
 6 14 15 7
 7 15 16 8
 8 16 17 9
 9 17 10 2
10 18 19 11
11 19 20 12
12 20 21 13
13 21 22 14
14 22 23 15
15 23 24 16
16 24 25 17
17 25 18 10
18 26 27 19
19 27 28 20
20 28 29 21
21 29 30 22
22 30 31 23
23 31 32 24
24 32 33 25
25 33 26 18
26 34 35 27
27 35 36 28
28 36 37 29
29 37 38 30
30 38 39 31
31 39 40 32
32 40 41 33
33 41 34 26
34 42 35 0
35 42 36 0
36 42 37 0
37 42 38 0
38 42 39 0
39 42 40 0
40 42 41 0
41 42 34 0
```

3.3 Parametric mesh generation

The parametric mesh generation has the aim of allowing the generation of slender bodies, mainly wings, to be generated parametrically in the preprocessor with few directives, without resorting to an external mesh generator.

To generate the parametric geometry a geometry input file to be referenced in the preprocessor input file 3.1 is employed. The input file is similar to the geometry files for standard meshes described in file 3.2

3.3.1 Surface panels and vortex lattices

input file 3.9: paramtetric_geo_file.in

```
mesh_file_type = parametric
el_type = v

scaling_factor = 1.0
offset = (/0.0, 0.0, 0.0/)

airfoil_table_correction = T

mesh_symmetry = F
symmetry_point = (/0.0, 0.0, 0.0/)
symmetry_normal = (/0.0, 1.0, 0.0/)

mesh_mirror = F
mirror_point = (/0.0, 0.0, 0.0/)
mirror_normal = (/1.0, 0.0, 0.0/)

twist_linear_interpolation = F

starting_point = (/0.0,0.0865,0.0/)
reference_chord_fraction = 0.0

!mesh_flat = T

nelem_chord = 10
type_chord = cosineLE

! First section
chord = 1.0
twist = -1.0
airfoil_table = ./airfoils/naca0012.c81
airfoil = NACA0012

! First region
span = 0.5
sweep = 0.0
dihed = 0.0
nelem_span = 3
type_span = uniform

! Second section
chord = 1.0
twist = 0.0
airfoil = interp
```

```
! Second region
span = 4.0
sweep = 10.0
dihed = 5.0
nelem_span = 10
type_span = uniform

! Third section
chord = 0.5
twist = 0.0
airfoil_table = ./airfoils/naca0012.c81
airfoil = NACA0012
```

The parametric geometry is obtained by connecting with a surface different planar sections. With the planar sections the user defines the shape of the section, the chord and the twist of the section. Then sections are connected by regions, with which the number of span panels, span, dihedral and sweep are defined. The user can define an arbitrary number n_s of sections, which must then be connected with $n_s - 1$ regions. A representation of sections and regions employed for the generation of a wing is presented in figure 3.2. The body is generated from the first section, starting from the specified starting point, and it is extruded along the y axis, with the x axis pointing from the leading edge towards the trailing edge and the z axis normal to the x-y plane. The type of geometry generated depends on the type of elements required. If surface panels are required the full three dimensional geometry will be generated, if vortex lattice elements are required only the flat, mean line surface will be generated and finally if lifting line elements are required only a one dimensional line will be generated. Finally the mirroring of the geometry introduced for user generated meshes is available also for parametric geometries.

All the detailed parameters of the geometry input file for parametric geometry are:

- `mesh_file_type`: *required:* yes. *multiple:* no. *type:* string.

  Use `parametric` for parametric geometry.

- `el_type`: *required:* yes. *multiple:* no. *type:* character.

  type of the elements of the mesh. `p` stands for surface panels to model solid bodies, `v` stands for vortex lattice elements used to model flat surfaces, `l` stands for lifting lines used to produce a 1D model of a lifting surface.

- `offset` *required:* no. *multiple:* no. *default:* (0.0, 0.0, 0.0) *type:* real array, length 3.

  offset to apply to the loaded points. Allows to move the coordinates of the loaded points of the vector specified.

- `scaling_factor` *required:* no. *multiple:* no. *default:* 1.0 *type:* real.

  Scaling factor to apply to the loaded points. Allows to scale the coordinates of the loaded points of the specified factor.

  offset and scaling are applied in the following order:

$$\mathbf{r} = Scaling(\mathbf{r}_{loaded} + offset)$$

- `airfoil_table_correction`: *required:* yes. *multiple:* no. *type:* logical.

  Require the viscous correction for `v` elements by introducing the c81 aerodynamic tables. (Still experimental)

- `mesh_symmetry` *required:* no. *multiple:* no. *default:* false. *type:* logical.

  Choose to reflect the mesh around a point and a direction. Useful to produce full meshes out of symmetrical half models. Keeps both the original and the symmetrical part.

- `symmetry_point`: *required:* only if `mesh_reflection` is true. *multiple:* no. *default:* (0.0, 0.0, 0.0). *type:* real array, length 3.

  point around which to reflect the mesh.

- `symmetry_normal`: *required:* only if `mesh_reflection` is true. *multiple:* no. *default:* (0.0, 1.0, 0.0). *type:* real array, length 3.

  Direction in which to reflect the mesh.

- `mesh_mirror` *required:* no. *multiple:* no. *default:* false. *type:* logical.

  Choose to mirror the mesh around a point and a direction. Same as `mesh_symmetry`, but does not keep both the original, i.e. the mesh is not doubled.

- `mirror_point`: *required:* only if `mesh_reflection` is true. *multiple:* no. *default:* (0.0, 0.0, 0.0). *type:* real array, length 3.

  point around which to mirror the mesh.

- `mirror_normal`: *required:* only if `mesh_reflection` is true. *multiple:* no. *default:* (0.0, 1.0, 0.0). *type:* real array, length 3.

  Direction in which to mirror the mesh.

- `twist_linear_interpolation` *required:* no. *multiple:* no. *default:* false. *type:* logical.

  Choose to apply linear interpolation to the twist angle, instead of interpolating the point coordinates between two neighboring sections defined in the input file.

- `starting_point` *required:* no. *multiple:* no. *default:* (/ 0.0, 0.0, 0.0 /) *type:* real array, length 3.

  point in the local reference frame in which to start extruding the parametric geometry.

- `reference_chod_fraction` *required:* no. *multiple:* no. *default:* 0.0 *type:* real

  Fraction of the chord at which to place the axis which will be rotated of the sweep and dihedral angles, and around which airfoils are twisted.

- `mesh_flat` *required:* no. *multiple:* no. *default:* true *type:* logical

  Used only in case of lifting lines (`l`) elements. If enabled instead of generating lifting lies actually twisted according to the input twist, but rather a flat surface with only the normal vectors twisted according to the input twist.

- `nelem_chord` *required:* yes, if parametric. *multiple:* no. *type:* integer.

  Number of elements in chord direction. Note: if the elements are vortex lattice $nelem\_chord$ elements will be generated, while in case of surface panels $2nelem\_chord$ elements will be produced, $nelem\_chord$ on the lower and $nelem\_chord$ on the upper side. In case of lifting lines this parameter is ignored.

- `type_chord` *required:* no. *multiple:* no. *default:* uniform *type:* string.

  Type of subdivision in the chord-wise direction. Can be `uniform` for a uniform distribution (suggested for vortex lattices), `cosine` for a cosine distribution, refined both on the leading and trailing edge, `cosineLE` for a half cosine refined only on leading edge (suggested for surface panels) or `cosineTE` for a + half cosine refined only on the trailing edge.

- `chord` *required:* at least 2. *multiple:* yes. *type:* real.

  Defines a section, by its chord length

- `twist` *required* in same number as `chord`. *multiple* yes. *type:* real.

  Angle of twist, in degrees, of the airfoil section.

- `airfoil`/`airfoil_table` *required:* in same number as `chord`. *multiple:* yes. *type:* string.

  Airfoil of the section. There are different ways to define an airfoil. It can be defined by the user as a series of points, and so `airfoil` must be the path to a .dat file (the extension is mandatory) containing the two dimensional coordinates of an airfoil; the file must have in the first line an integer representing the number of points provided, followed by the coordinates of each point on separate lines. It can also be an analytical NACA profile, and so must be provided as a NACAXXXX string (at the moment only 4 digits and some 5 digits are implemented).

  In case of lifting lines or when `airfoil_table_correction` is true on the vortex lattice, to mark the difference the `airfoil_table` must be employed, and it refers to the path to the corresponding c81 lookup table.

  Optionally in the `airfoil` or `airfoil_table` parameter it is possible to specify the option `interp`. In this case the airfoil (airfoil section, mean line, section defined from file or c81 lookup table according to the type of elements) in the section will be linearly interpolated among the pair of explicitly specified airfoils in which the section is contained.

- `span` *required:* at least one, in same number as `chord`-1 *multiple:* yes. *type:* real.

  Define the span length of the region between two sections.

- `sweep` *required:* in same number as `span` *multiple:* yes. *type:* real.

  Angle of sweep of the region, positive if swept backwards (towards positive x).

- `dihed` *required:* in same number as `span` *multiple:* yes. *type:* real.

  Angle of dihedral of the region, positive if rotated upwards (towards positive z).

- `nelem_span` *required:* in same number as `span` *multiple:* yes. *type:* integer.

  Number of elements in the spanwise direction, in the single region.

- `type_span` *required:* in same number as `span` *multiple:* yes. *type:* string.

  type of refinement of the elements in the spanwise direction. As for the chordwise direction, the options are `uniform` for a uniform distribution, `cosine` for a cosine refinement both inboard and outboard, and `cosine_ib` and `cosine_ob` for half cosine refinement only inboard or outboard.

The concept of sections and regions is illustrated in figure 3.2. Furthermore an example to understand the geometrical role of the parameters in the input file is presented in file 3.10 and figure 3.3.

When generating the parametric geometry, the slender body is generated starting from the `starting_point` and then develops in the y direction (in the local reference frame of the component), with deviations induced by the various angles used as input.

From the `starting_point` a segmented reference line is created, in blue in figure 3.3. Each segment, which creates a region as in figure 3.2, is created from the previous node, is long `span` and is angled from the local component y direction of `dihed` dihedral angle with respect to the horizontal plane (rotating around the local x axis, positive upward, red in the figure) and of `sweep` angle with respect to the vertical plane (rotating around the local z axis, positive backward towards x, green in figure).

Then on each node the airfoil section is applied, 2 dimensional for surface panels (as in figure) mono dimensional for vortex lattices (lifting lines will be discussed afterwards). The section is collocated so that the reference line passes through a certain `reference_chord_fraction` of the airfoil chord. Then the section is rotated around such point of a `twist` angle (around the component local y axis, positive when creating a positive pitch to the airfoils, in magenta in the figure).

Finally all the sections are connected with the appropriate number of elements, as shown in figure 3.2.

input file 3.10: Parametric geometry for figure 3.3

```
mesh_file_type = parametric
el_type = p
```

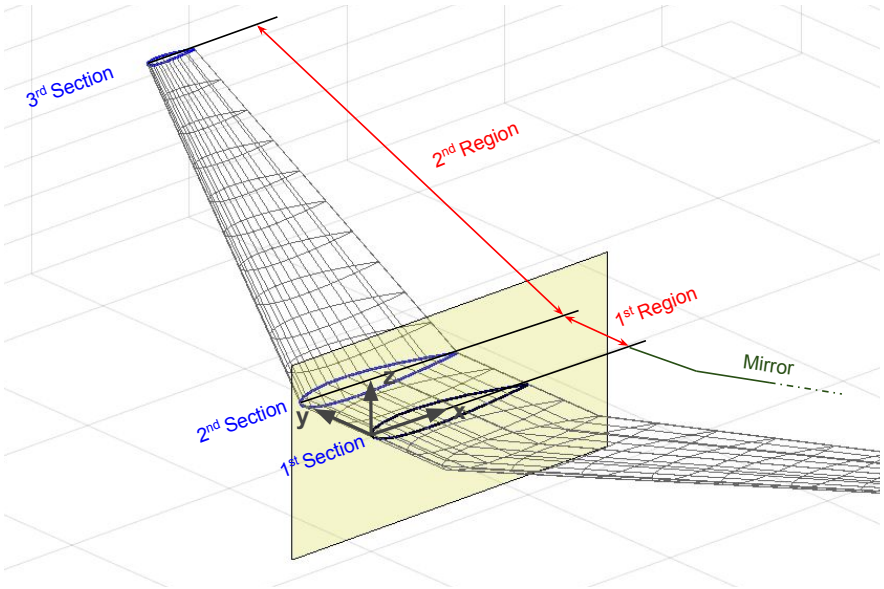

Figure 3.2: Sections and regions in parametric wing generation

```
nelem_chord = 10
type_chord = uniform ! uniform cosineLE cosineTE

scaling_factor = 1.0
offset = (/0.0, 0.0, 0.0/)

starting_point = (/0.0,0.0,0.0/)
reference_chord_fraction = 0.25

! section 1
chord = 2.0
twist = 20.00
airfoil = NACA0012

! region 1
span = 1.5
sweep = 0.0
dihed = 0.0
nelem_span = 15
type_span = uniform

! section 2
chord = 1.5
twist = 15.0
airfoil = NACA0012

! region 2
span = 5.0
sweep = 0.0
dihed = 5.0
nelem_span = 50
type_span = uniform

! section 3
```

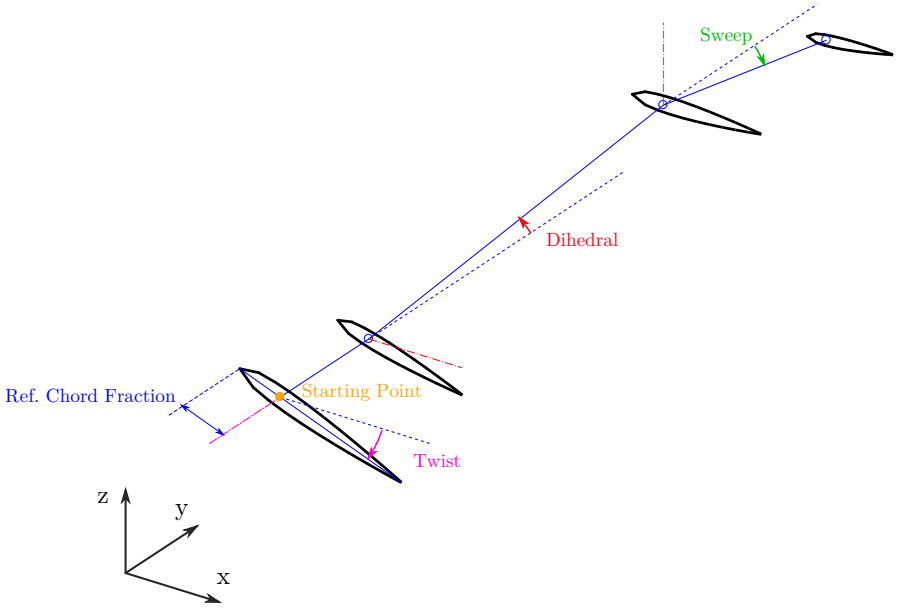

Figure 3.3: Generation logic of the geometry of parametric elements, generated from input file 3.10

```
chord = 1.5
twist = 0.0
airfoil = NACA0012

! region 3
span = 2.0
sweep = 15.0
dihed = 0.0
nelem_span = 20
type_span = uniform

! section 4
chord = 1.0
twist = -5
airfoil = NACA0012
```

3.3.2 Airfoils sections geometry

The airfoils sections, introduced with the keyword `airfoil`, as previously discussed, can be introduced either by specifying a NACA airfoil shape, or by providing a two dimensional geometry using an ascii file. In both cases the resulting number of points in the section is the one defined in `nelem_chord`, with the distribution specified in `type_chord`.

In case of NACA airfoils, only 4 digits NACA airfoils and some 5 digits ones are implemented. When surface panels are employed the full shape is discretized, while when employing vortex lattices only the camber line is used to discretize the surface.

In case of user provided geometry, the geometry must be provided as an ascii file with the coordinates used to describe the shape to be discretized. To mark the use of the user-defined shape the file name must have the extension `.dat`. The first line of the file must contain a single integer, which represents the number of points used to describe the shape, and thus the number of following lines. The next lines must contain two real numbers on each line, representing the x and y coordinates of each point. The horizontal, streamwise x axis points towards the trailing edge, and the vertical y axis points upwards.

The order in which the points are provided defines the direction of the curve that defines the shape of the airfoil. The curve must start at the trailing edge, pass from the lower side of the airfoil, the leading edge, the upper side and end again at the trailing edge. The first and last point can be not coinciding, to generate an open trailing edge. In case of vortex lattices, the mean line is automatically computed.

The input is assumed to be normalized to have an unit chord, the final chord of the geometry will be generated by multiplying by the `chord` parameter the coordinates (thus if the geometry provided is not normalized to have a unit chord, the actual chord obtained will be different from the value specified in `chord`).

An example of user prescribed airfoil shape is provided in file 3.11.

input file 3.11: Example of user specified airfoil shape (the middle lines have been suppressed for brevity)

```
72
1.0 0.0
0.996103 -1.3540067E-4
0.984179 -3.5814368E-4
0.964464 -1.693276E-4
0.937582 -5.391884E-4
0.904605 -0.0017722012
0.866738 -0.0038649566
...
...
...
0.948453 0.013156898
0.96962 0.0077201095
0.985778 0.00345874
0.996349 8.9363643E-4
1.0 0.0
```

3.3.3 Lifting lines

The geometrical logic for the generation of lifting lines is similar to the one for surface panels or vortex lattices, and differs in only few details. An example of the same geometry defined in file 3.10 and depicted in figure 3.3 but generated as lifting lines is presented in figure 3.4. When employing lifting lines the reference line is generated starting from the `starting_point` exactly in the same way as in the other cases, however in this case this line is the one that will become the lifting line. From the lifting line a single panel is generated to represent the object surface, and implicitly the first wake panel. The panel is long 75% of the indicated chord, and it is angled according to the `twist` angle set in the input. Therefore note that in case of lifting lines the lifting line is essentially always placed at 25% of the ideal airfoil it should represent, and that the parameter `reference_chord_fraction` is ignored.

Finally it is important to stress that the reference frames introduced in the present section concerning the generation of parametric components are local to those component, and are required only to assign coordinates to the points generated parametrically, just as a mesh generated from CAD files and an external mesher will generate points in a certain reference frame. The actual position of the component in space during the simulation depends on the reference frame in which the component will be introduced. Reference frames employed during the simulation are discussed in section 5.4.

3.3.4 Hinged Surfaces

It is possible to introduce one or more movable surfaces in the parametric definition of a wing. As outlined by the scheme in Fig. 3.5 left, in a two-dimensional problem the control surface can be defined in the local reference frame of the component, by means of the hinge axis position H, the chordwise direction ξ and a blending region $[-u, u]$ (defined by `hinge_Offset` parameter)introduced to avoid irregular behavior of the mesh along with the rotation angle θ. Three regions are defined using the coordinates defined through this

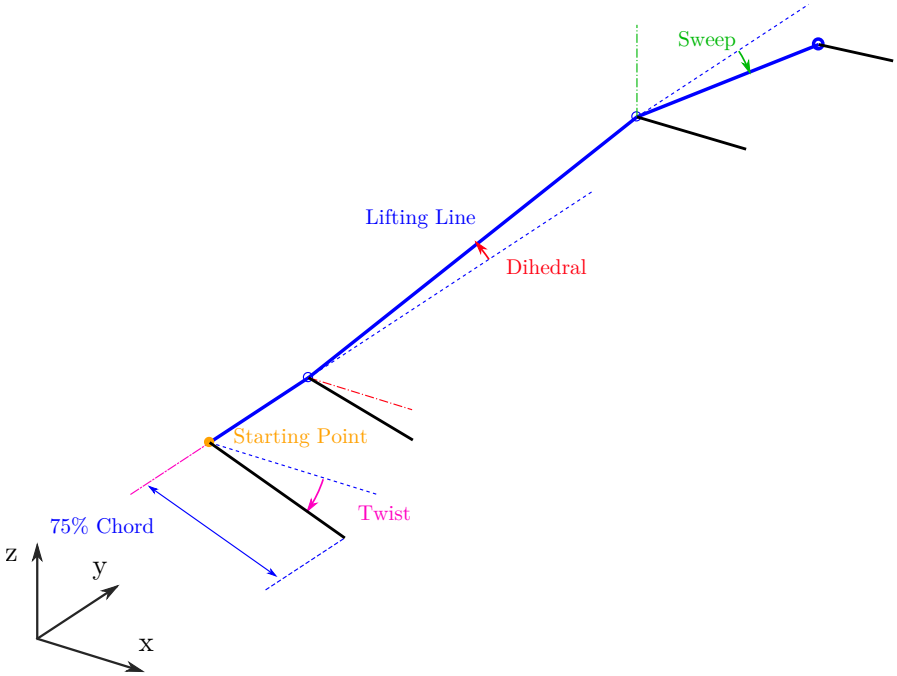

Figure 3.4: Generation logic of the geometry of parametric lifting lines.

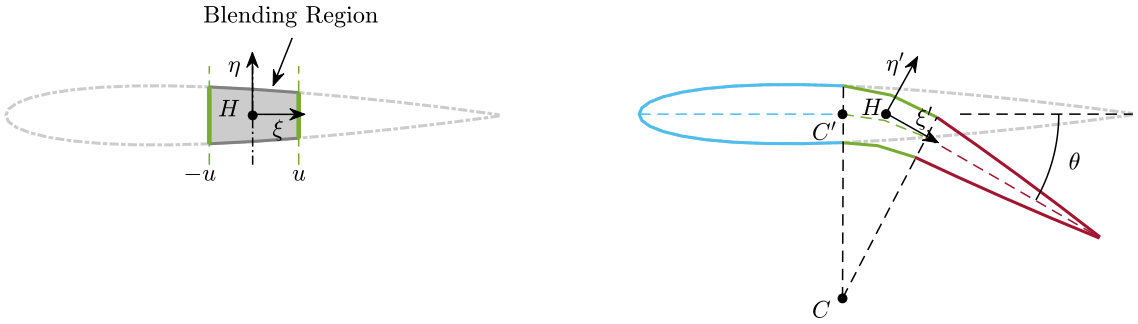

Figure 3.5: Scheme of the two-dimensional hinged surface configuration

reference frame:

1. $\xi \leq -u$: no influence of the aileron rotation;
2. $\xi \geq u$: rigid rotation around the hinge;
3. $-u \leq \xi \leq u$: blending region for avoiding irregular behavior defined as an arc of a circle whose center is locate on point C;

In a three-dimensional problem, the reference configuration of a control surface, as an aileron, is defined in wind axis reference frame of the component.

The aerodynamic sections that are involved in the control surface are then the ones that satisfy the condition $y(A) < y(P) < y(B)$, where $y(P)$ is the ordinate of the $P_i - th$ aerodynamic mesh point expressed in the wind reference system. As in the 2D case, we can then define the three regions for each stripe identified at the previous point. The y coordinate of the origin of the sectional reference frame is determined by linear interpolation between points A and B. Points A and B corresponds to the user input `node_1` and `node_2`, respectively. The rotation axis is defined by $\boldsymbol{h} = (B - A)$.

input file 3.12: Parametric geometry for flapped wing

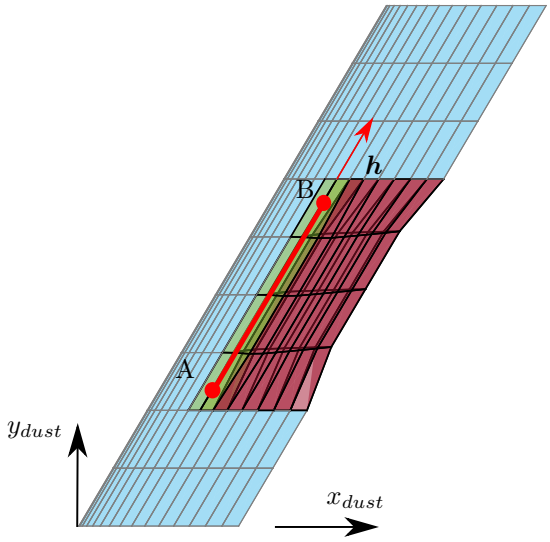

Figure 3.6: hinge reference system for a swept wing

```
mesh_file_type = parametric
el_type = p
scale_factor = 1.0
offset = (/0.0 , 0.0, 0.0/)

mesh_symmetry = T
symmetry_point = (/0.0 , 0.0, 0.0/)
symmetry_normal = (/0.0 , 1.0, 0.0/)

nelem_chord = 15
type_chord = cosineLE ! uniform cosineLE cosineTE
starting_point = (/0.0,0.0,0.0/)
reference_chord_fraction = 0.25

n_hinges = 2
hinge = {
  hinge_tag = aileron_right
  hinge_nodes_input = parametric ! or from_file
  node_1 = (/ 1.4178, 1.4557, 0.3473/) ! In the local ref.frame
  node_2 = (/ 2.4026 , 3.1615 , 0.6946/)
  n_nodes = 2
  ! }
  ! hinge_nodes_input_from_file = {
  ! node_file = hinge_node.dat
  ! }
  hinge_ref_dir = (/ 1.0, 0.0 , 0.0 /)
  hinge_offset = 0.1
  hinge_spanwise_blending = 0.01
  hinge_rotation_input = function:sin
  hinge_rotation_function = {
    amplitude = 30.0 ! deg
    omega = 12.5 ! rad/sec
```

```
    phase = 0.0 ! deg
  }

hinge = {
  hinge_tag = aileron_left
  hinge_nodes_input = parametric
  node_2 = (/ 1.4178, -1.4557, 0.3473/)
  node_1 = (/ 2.4026 , -3.1615 , 0.6946/)
  n_nodes = 2
  ! }
  ! hinge_nodes_input_from_file = {
  ! node_file = hinge_node.dat
  ! }
  hinge_ref_dir = (/ 1.0, 0.0 , 0.0 /)
  hinge_offset = 0.1
  hinge_merge_tol = 0.01
  hinge_spanwise_blending = 0.01
  hinge_rotation_input = function:sin
  hinge_rotation_function = {
    amplitude = -30.0 ! deg
    omega = 12.5 ! rad/sec
    phase = 0.0 ! deg
  }


! First section
chord = 2
twist = 0.0
airfoil = NACA0012

! First region
span = 5.0
sweep = 30.0
dihed = 10.0
nelem_span = 20
type_span = uniform

! Second section
chord = 2
twist = 0.0
airfoil = NACA0012
```

All the detailed parameters of the Hinged surfaces input file are:

- `n_hinge`: *required:* no. *multiple:* no. *default:* 0. *type:* integer
  Number of hinges and rotating parts (e. g. aileron) of the component.
- `hinge_Tag`: *required:* no. *multiple:* same number as `n_hinge` *type:* string
  name of the control surface.
- `hinge_Nodes_Input`: *required:* yes. *multiple:* no. *type:* string
  type of hinge nodes input: `parametric` or `from_file`. (TODO add details)
- `node_1` *required:* yes. *multiple:* no. *type:* real array, length 3.
  First node of the hinge. Components in the local reference frame of the component.

- `node_2` *required:* yes. *multiple:* no. *type:* real array, length 3.

  Second node of the hinge. Components in the local reference frame of the component.

- `hinge_ref_dir` *required:* yes. *multiple:* no. *type:* real array, length 3.

  Reference direction of the hinges, indicating zero-deflection direction in the local reference frame of the component.

- `hinge_merge_tol` *required:* no. *multiple:* no. *type:* real. *default:* 0.01

  Tolerance for adaptive hinge mesh in chord adimensional length.

- `hinge_offset` *required:* no. *multiple:* no. *type:* real. *default:* 0.

  offset in the `hinge_ref_dir` needed for avoiding irregular behavior of the surface for large deflections.

- `hinge_spanwise_blending` *required:* no. *multiple:* no. *type:* real. *default:* 0.

  Blending in the spanwise direction needed for avoiding irregular behavior of the surface for large deflections.

- `hinge_rotation_input` *required:* yes. *type:* string.

  input type of the rotation: `function`, `from_file` or `coupling`. If the chosen option is `coupling`, then the component must be coupled with preCICE (`coupling T`).

- `hinge_rotation_function`: *required:* no. *type:* string.

  Parser for hinge input with simple functions; the supported functions are: `function:const`, `function:sin`, `function:cos`.

  - `amplitude` *required:* yes. *type:* real.
    amplitude of the rotation in degrees for constant, cosine and sine functions.
  - `omega` *required:* yes. *type:* real. *default:* 0.0
    Angular velocity of the rotation function in deg/s, for constant?, cosine and sine functions.
  - `phase` *required:* yes. *type:* real. *default:* 0.0
    phase angle of the rotation function in deg, for constant?, cosine and sine functions.

- `hinge_Rotation_File` *required:* no. *type:* string.

  Parser for hinge input from file

  - `file_name` *required:* yes. *type:* string.
    name of the file containing the input of the hinge rotation

- `hinge_rotation_coupling` *required:* no. *type:* string.

  Parser for hinge input from coupling (see 4.3.1)

  - `coupling_node_subset` *required:* no. *type:* string.
    Define a subset of structural nodes to evaluate coupling: `range` or `from_file`
  - `coupling_node_first` *required:* no. *type:* integer.
    If node subset is defined through `range` input: first ID of the nodes
  - `coupling_node_last` *required:* no. *type:* integer.
    If node subset is defined through `range` input: last ID of the nodes
  - `coupling_node_filename` *required:* no. *type:* string.
    file collecting the IDs of the coupling nodes for hinge coupling

3.4 Pointwise mesh generation

The `pointwise` mesh definition extends the capabilities of the `parametric` input. First, the reference line of the component is defined as a list of points connected with straight lines or Hermitian splines. The sections of a component are defined at each input points by means of their plane coordinates, their dimensions and rotation around an axis perpendicular to their own plane. The input file is mainly composed of three sections: a generic section for the types of the aerodynamic elements and the symmetry and mirroring options, a section for the `point` list and a section for the `Line`. The parameters of the first section are very similar to the ones used in the `parametric` description, except for the `starting_point`, that is not needed here, since the point with `Id`= 1 is meant to be the first point of the reference line.
An example of input file for `pointwise`-defined component is provided in file 3.13. The description of this file and all the parameters follows.

input file 3.13: "Pointwise" geometry definition

```
mesh_file_type = pointwise
el_type = p

mesh_symmetry = T
symmetry_point = (/0.0,0.0,0.0/)
symmetry_normal = (/0.0,1.0,0.0/)

mesh_mirror = F
mirror_point = (/0.0,0.0,0.0/)
mirror_normal = (/0.0,1.0,0.0/)

reference_chord_fraction = 0.0

mesh_flat = F

nelem_chord = 20
type_chord = cosineLE

! === Points ===
point = {
  id = 1
  coordinates = (/ 0.0 , 0.0 , 0.0 /)
  airfoil = NACA2412
  chord = 1.0
  twist = 5.0
  section_normal = yAxis
}
point = {
  id = 2
  coordinates = (/ 1.5 , 3.0 , 0.3 /)
  airfoil = NACA2412
  chord = 0.4
  twist = 0.0
  section_normal = yAxis
}
point = {
  id = 3
  coordinates = (/ 1.8 , 3.5 , 0.7 /)
  airfoil = NACA2412
  chord = 0.3
  twist = 0.0
  section_normal = reference_line ! (default)
```

```
}
point = {
  Id = 4
  Coordinates = (/ 2.1 , 3.5 , 1.1 /)
  airfoil = NACA2412
  chord = 0.3
  twist = 0.0
  section_normal = reference_line ! (default)
}
point = {
  id = 5
  coordinates = (/ 2.4 , 3.0 , 1.4 /)
  airfoil = NACA2412
  chord = 0.4
  twist = 0.0
  section_normal = y_axis_neg
  flip_section = T
}
point = {
  id = 6
  coordinates = (/ 3.5 , 0.0 , 1.5 /)
  airfoil = NACA2412
  chord = 1.0
  twist = 5.0
  section_normal = y_axis_neg
  flip_section = T
}

! === Lines ===
line = {
  type = straight
  end_points = (/ 1 , 2 /)
  nelems = 5
}
line = {
  type = spline
  end_points = (/ 2 , 5 /)
  nelems = 10
  ! see documentation for optional inputs
}
line = {
  type = straight
  end_points = (/ 5 , 6 /)
  nelems = 5
}
```

- `mesh_file_type`: *required:* yes. *multiple:* no. *type:* string.

  Use `pointwise` for pointwise geometry.

- `el_type`: *required:* yes. *multiple:* no. *type:* character.

  type of the elements of the mesh. `p` stands for surface panels to model solid bodies, `v` stands for vortex lattice elements used to model flat surfaces, `l` stands for lifting lines used to produce a 1D model of a lifting surface.

- `mesh_symmetry` *required:* no. *multiple:* no. *default:* false. *type:* logical.

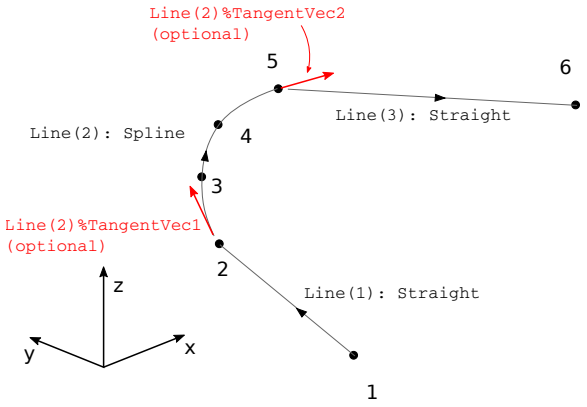

Figure 3.7: Pointwise definition of a component: reference line by `points` and `Lines` connecting them.

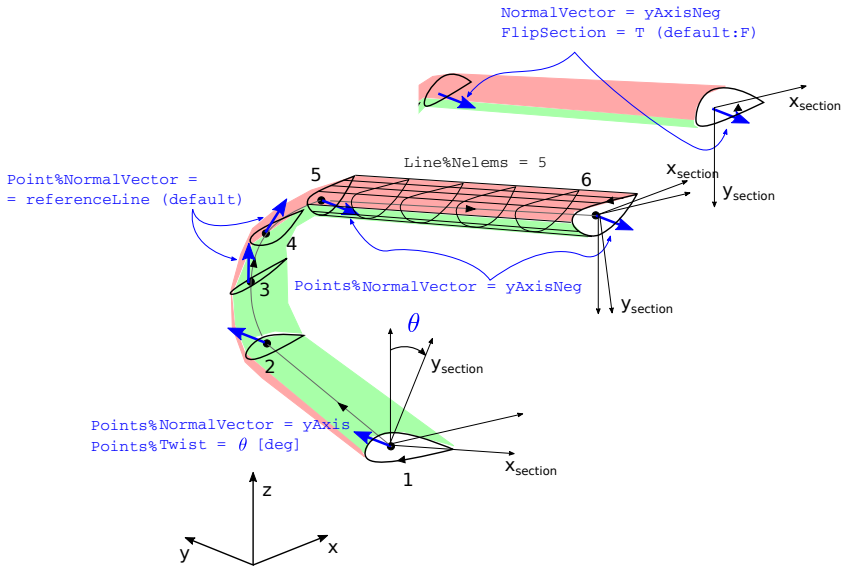

Figure 3.8: Pointwise definition of a component: section parameters (`airfoil`, `chord`, `twist`, `section_-normal`) at `points` of the reference line and the number `nelems` of elements along the reference line for each `line`. The parameter `flip_section` can be set to `.true.` in order to flip the y-coordinate of the airfoil, in the section reference frame: the comparison between the goemetry with `flip_section` equal to `F` or `T` at `point` with `id` $= 5, 6$ is shown in the picture for a non-symmetrical airfoil.

Choose to reflect the mesh around a point and a direction. Useful to produce full meshes out of symmetrical half models. Keeps both the original and the symmetrical part.

- `symmetry_point`: *required:* only if `mesh_reflection` is true. *multiple:* no. *default:* (0.0, 0.0, 0.0). *type:* real array, length 3.

  point around which to reflect the mesh.

- `symmetry_normal`: *required:* only if `mesh_reflection` is true. *multiple:* no. *default:* (0.0, 1.0, 0.0). *type:* real array, length 3.

  Direction in which to reflect the mesh.

- `mesh_mirror` *required:* no. *multiple:* no. *default:* false. *type:* logical.

Choose to mirror the mesh around a point and a direction. Same as `mesh_symmetry`, but does not keep both the original, i.e. the mesh is not doubled.

- `mirror_point`: *required:* only if `mesh_reflection` is true. *multiple:* no. *default:* (0.0, 0.0, 0.0). *type:* real array, length 3.

  point around which to mirror the mesh.

- `mirror_normal`: *required:* only if `mesh_reflection` is true. *multiple:* no. *default:* (0.0, 1.0, 0.0). *type:* real array, length 3.

  Direction in which to mirror the mesh.

- `reference_chod_fraction` *required:* no. *multiple:* no. *default:* 0.0 *type:* real

  Fraction of the chord at which to place the axis which will be rotated of the sweep and dihedral angles, and around which airfoils are twisted.

- `mesh_flat` *required:* no. *multiple:* no. *default:* true *type:* logical

  Used only in case of lifting lines (`l`) elements. If enabled instead of generating lifting lies actually twisted according to the input twist, but rather a flat surface with only the normal vectors twisted according to the input twist.

- `nelem_chord` *required:* yes, if parametric. *multiple:* no. *type:* integer.

  Number of elements in chord direction. Note: if the elements are vortex lattice $nelem\_chord$ elements will be generated, while in case of surface panels $2nelem\_chord$ elements will be produced, $nelem\_chord$ on the lower and $nelem\_chord$ on the upper side. In case of lifting lines this parameter is ignored.

- `type_chord` *required:* no. *multiple:* no. *default:* `uniform` *type:* string.

  type of subdivision in the chord-wise direction. Can be `uniform` for a uniform distribution, `cosine` for a cosine distribution, refined both on the leading and trailing edge, `cosineLE` for a half cosine refined only on leading edge or `cosineTE` for a half cosine refined only on the trailing edge.

A list of `point` groups follows. These points are used to describe the reference line of the component. The parameters of a `point` element are:

- `id` *required*: yes, *multiple*: no, *type*: integer.

  Id. number of the `point`, used in the point-to-line connectivity defined in `line` groups.

- `coordinates` *required*: yes, *multiple*: no, *type:* real array, length 3.

  Coordinates of the point in the local reference frame of the component.

- `airfoil` *required*: yes, *multiple*: no, *type*: string.

  Same as in the parametric definition of a component. To be used for vortex lattices or surface panels

- `airfoil_table` *required*: yes, *multiple*: no, *type*: string.

  Same as in the parametric definition of a component. To be used for lifting lines only in place of `airfoil`

- `chord` *required*: yes, *multiple*: no, *type*: real.

  Define section dimensions, by its chord length.

- `twist` *required*: yes, *multiple*: no, *type*: real.

  Angle of twist, in degrees, of the airfoil section. rotation around the vector that is normal to the plane of the section.

- `section_normal` *required*: no, *multiple*: no , *default*: `reference_line` , *type*: string.

  String to define the plane of the section. It can be: `reference_line`, for sections perpendicular to the reference line; `y_axis`, `y_axis_neg` for sections perpendicular to the y-axis or $-y$-axis of the component local reference frame; `vector`, to define a generic vector.

- `section_normal_vector` *required*: only if `section_normal = vector`, *multiple*: no, *type*: real array, length 3.

  Components of the normal vector of the section, if `section_normal = vector`.

- `flip_section` *required*: no, *multiple*: no, *default*: `F` , *type*: logical.

  Flip the y coordinates in the section reference frame. Meant to help the pointwise definition of close wing configurations.

A list of `line` groups follows. These lines are used to build the reference line of the component connecting the points defined above. The parameters of a `line` element are:

- `type` *required*: yes, *multiple*: no, *type*: string.

  Line type. It can be: `straight` for straight lines, `spline` for Hermitian splines.

- `end_points` *required*: yes, *multiple*: no, *type*: integer array, lenght 2.

  `id` numbers of the beginning and ending `point`s of the line. For `straight` lines, this two numbers must be consecutive. For `spline`s the spline is built using these ones as the first and last points and all the points with `Id` in between as interior points, so that they must exist in the point list.

- `nelems` *required*: yes, *multiple*: no, *type*: integer.

  Number of elements in the direction of the reference line belonging to this region.

- `type_span` *required*: no, *multiple*: no, *default*: `uniform`, *type*: string.

  type of refinement of the elements in the spanwise direction. As for the chordwise direction, the options are `uniform` for a uniform distribution, `cosine` for a cosine refinement both inboard and outboard, and `cosineIB` and `cosineOB` for half cosine refinement only inboard or outboard.

- `tangent_vec1` *required*: if `Line%type = spline` and `Line%end_points(1) = 1`, *multiple*: no, *type*: real array, length 3.

  Tangent vector at the first point of a `spline`. This is an optional input for `spline`. If this field is not present, the spline inherits the tangent vector from the neighboring line (that must be a `straight` line).

- `tangent_vec2` *required*: if `Line%type = spline` and `Line%EndPoints(2)` is the last point of the reference line, *multiple*: no, *type*: real array, length 3.

  Tangent vector at the last point of a `spline`. This is an optional input for `spline`. If this field is not present, the spline inherits the tangent vector from the neighboring line (that must be a `straight` line).

- `tension` *required*: no, *multiple*: no, *default*: 0.0 , *type*: real.

  `tension` parameter of the Hermitian `spline`.

- `bias` *required*: no, *multiple*: no, *default*: 0.0 , *type*: real.

  `bias` parameter of the Hermitian `spline`.

3.4.1 Hermitian splines

The Hermitian splines by the following expression,

$$\mathbf{r}_i(t) = \mathbf{r}_{i-1}h_0(t) + \mathbf{r}_i h_1(t) + \mathbf{d}_{i-1}h_2(t) + \mathbf{d}_i h_3(t) \ , \tag{3.1}$$

where $h_i(t),\ i = 0:3$ are the Hermitian functions, the points $\mathbf{r}_i$ are the points to be interpolated, $\mathbf{d}_i$ the approximation of the derivatives at the interpolation points. For the interior points of a spline,

$$\begin{aligned} \mathbf{d}_i = {} & 0.5(\mathbf{r}_{i+1} - \mathbf{r}_i)\ (1 - \texttt{bias})(1 - \texttt{tension}) \\ & + 0.5(\mathbf{r}_i - \mathbf{r}_{i-1})\ (1 + \texttt{bias})(1 - \texttt{tension}) \ . \end{aligned} \tag{3.2}$$

The `bias` parameter is a weight on the finite difference computed using forward and backward differences, while `tension` acts like a tensile action on the spline, see figure 3.9.

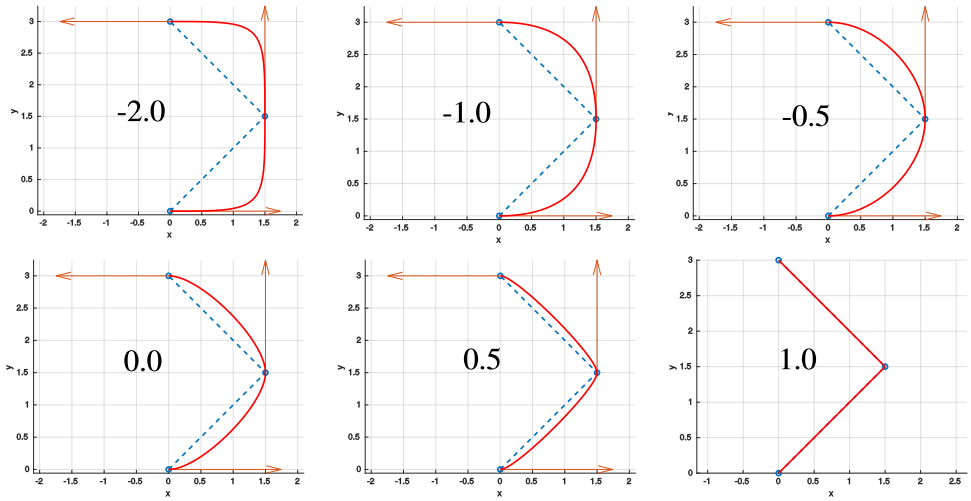

Figure 3.9: Influence of the `tension` parameter on the shape of the spline interpolating points $(0,0)$, $(1.5,1.5)$ $(0,3)$, with prescribed horizontal derivatives at the end points.

3.4.2 Limitations and errors in the pointwise definition of a component

So far, the code has the following limitations:

- if `mesh_symmetry = T`, the last section of the component can be joined only if the first section is joined. A workaround is to flip the order of the points and reverse the direction of the reference line, in order to have the first section corresponding to the previously defined last section and viceversa, so that the desired sections are joined. No limitation exists if both the first and last sections belong to the symmetry plane and are joined.
- it is not possible to join two consecutive `splines` lines. So far, `splines` are meant to join `straight` lines or to be the first or last line. However a single spline can be used to join an arbitrary number of points.
- it is not possible to define the first or last line as a `spline`, without providing the tangent vector at the free end, i.e. the `tangent_vec1` must be provided to the first line, `tangent_vec2` to the last line. spline routines have the capability to treat free ends, with second derivative equal to zero, but some modifications in the pointwise definition of a component are still needed to allow for free ends.

3.4.3 Bodies of Revolution

A simple parametric mesh generation to create bodies of revolution is available. With this type of input is possible to generate bodies by revolving an user provided profile or cylindrical bodies with smooth tapered ends only with parametric inputs. An example of input file for this kind of parametric input is available in file

input file 3.14: Boies of revolution geometry definition

```
mesh_file_type = revolution
el_type = p

mesh_symmetry = T
symmetry_point = (/0.0,0.0,0.0/)
symmetry_normal = (/0.0,1.0,0.0/)

mesh_mirror = F
mirror_point = (/0.0,0.0,0.0/)
mirror_normal = (/0.0,1.0,0.0/)
```

```
!mesh_file = rev_profile.dat

rev_length = 5.0
rev_nose_radius = 1.0
rev_radius = 0.7
rev_nelem_long = 30
rev_nelem_rev = 10
```

- `mesh_file_type`: *required:* yes. *multiple:* no. *type:* string.

  Use `revolution` for bodies of revolution

- `el_type`: *required:* yes. *multiple:* no. *type:* character.

  Only `p` for surface panels is supported for bodies of revolution.

- `mesh_symmetry` *required:* no. *multiple:* no. *default:* false. *type:* logical.

  Choose to reflect the mesh around a point and a direction. Useful to produce full meshes out of symmetrical half models. Keeps both the original and the symmetrical part.

- `symmetry_point`: *required:* only if `mesh_reflection` is true. *multiple:* no. *default:* (0.0, 0.0, 0.0). *type:* real array, length 3.

  point around which to reflect the mesh.

- `symmetry_normal`: *required:* only if `mesh_reflection` is true. *multiple:* no. *default:* (0.0, 1.0, 0.0). *type:* real array, length 3.

  Direction in which to reflect the mesh.

- `mesh_mirror` *required:* no. *multiple:* no. *default:* false. *type:* logical.

  Choose to mirror the mesh around a point and a direction. Same as `mesh_symmetry` but does not keep both the original, i.e. the mesh is not doubled.

- `mirror_point`: *required:* only if `mesh_reflection` is true. *multiple:* no. *default:* (0.0, 0.0, 0.0). *type:* real array, length 3.

  point around which to mirror the mesh.

- `mirror_normal`: *required:* only if `mesh_reflection` is true. *multiple:* no. *default:* (0.0, 1.0, 0.0). *type:* real array, length 3.

  Direction in which to mirror the mesh.

- `mesh_file`: *required:* no. *multiple:* no. *type:* string.

  name of an ascii file containing the point list describing the curve or profile to revolve, which will be revolved around the local x axis. If the input is present, the rest of the inputs are neglected and the body is built revolving the provided profile, otherwise if `mesh_file` is not present, the parametric input is employed

- `rev_length`: *required:* yes if `mesh_file` is not present. *multiple:* no. *type:* real.

  Length of the parametric body of revolution, from tip to tip.

- `rev_nose_radius`: *required:* yes if `mesh_file` is not present. *multiple:* no. *type:* real.

  radius of the smooth circular tapering at the ends of the cylindrical body of revolution

- `rev_radius`: *required:* yes if `mesh_file` is not present. *multiple:* no. *type:* real.

  radius of the central cylindrical section of the body of revolution

- `rev_nelem_long`: *required:* yes if `mesh_file` is not present. *multiple:* no. *type:* integer.

  Number of elements in the longitudinal direction of the body of revolution

- `rev_nelem_rev`: *required:* yes. *multiple:* no. *type:* integer.

  Number of elements in the revolution direction.

3.5 Trailing Edge

The trailing edge is the line from which the wake is shed from the geometry elements. It is meant to represent the location where the thin vortical layer leaves a lifting body when the boundary layer is still attached. In the DUST preprocessor the trailing edge is identified geometrically, possibly according to a set of parameters.

In case of paranetrically generated elements, which represent extruded airfoils, the trailing edge is simply generated connecting all the trailing edge of the two (or mono) dimensional sections that compose the parametric geometry.

In case of unstructured surface meshes, the preprocessor proceeds by trying to geometrically identify the trailing edge in sharp corners between elements.

Considering the (rather common) possibility that the trailing edges are left open, i.e. with element edges not geometrically connected and/or not logically connected in the connectivity, first the DUSTpreprocessor merges the close nodes and generates an alternative connectivity taking into account the new connections. This merging is just functional to the definition of the trailing edge, and will be discarded after the trailing edge had been identified. To control the merging, the parameter `tol_se_wing` can be declared in the preprocessor input file, to be applied globally, or in each (or some) single geometry input file to be applied to a single component. Each pair of nodes separated by a distance lower than `tol_se_wing` will be merged in this phase.

After merging the mesh the new connectivity will be employed to identify the edges belonging to the trailing edge. The edges between two elements whose normal vectors are *sufficiently* opposed are marked as trailing edge. In particular, the edge connecting the elements i and j is a trailing edge if

$$\boldsymbol{n}_i \cdot \boldsymbol{n}_j < \texttt{inner\_product\_te}, \tag{3.3}$$

where $\boldsymbol{n}_i, \boldsymbol{n}_j$ are the normal unit vectors of the respective elements.

Moreover during the preprocessing also the direction of the first wake panel is decided. During the simulation the whole wake is advected according to the velocity generated by the singularities of the bodies and the wake, however the first wake panel is geometrically pre-determined and its intensity is implicitly solved alongside the rest of the singularities of the body surfaces. The first panel starts from the trailing adge and its nodes, and its length is decided during the simulation, however the direction alongside it is stretched from the trailing edge is determined in the preprocessor.

If no specific indication is given, the direction of the first implicit wake panel is the average of the two edges directions from the upper and lower elements connected at the trailing edge node. It is however possible to alter this behaviour by setting `proj_te` to true. Then the user must provide a direction vector with `proj_te_vector`, which identifies the direction in which to project the first panel direction if `proj_te_dir` is `parallel`, or is the normal to the plane in which to project the first panel direction if `proj_te_dir` is `normal`.

3.6 Actuator Disks

Actuator disks are built employing a parametric input file just as file 3.9 but with different parameters, shown in file 3.15

input file 3.15: actdisk_geo_file.in

```
mesh_file_type = parametric
el_type = a

radius = 2.5
nstep = 20
```

```
axis = 3
traction = 10.0
```

All the detailed parameters of the geometry input file for parametric actuator disks are:

- `mesh_file_type`: *required:* yes. *multiple:* no. *type:* string.
  Use `parametric` for parametric geometry.
- `el_type`: *required:* yes. *multiple:* no. *type:* character.
  type of the elements of the mesh. For actuator disks must be `a`
- `radius`: *required:* yes. *multiple:* no. *type:* real.
  radius of the actuator disk.
- `nstep`: *required:* yes. *multiple:* no. *type:* integer.
  Number of straight segments used to discretize the circle of the actuator disk.
- `axis`: *required:* yes. *multiple:* no. *type:* integer.
  Which of the three axis of the reference frame to use as axis of the rotor. The reference frame is the one specified for the component in the preprocessor input, file 3.1
- `traction`: *required:* yes. *multiple:* no. *type:* real.
  traction of the actuator disk.

Chapter 4

Coupling with a structural software through preCICE

4.1 Introduction

The communication between DUST and MBDyn is managed by preCICE (Precise Code Interaction Coupling Environment), a coupling library for partitioned multi-physics simulations, originally developed for fluid-structure interaction and conjugate heat transfer simulations. preCICE offers methods for transient equation coupling, communication means, and data mapping schemes. It is written in C++ and offers additional bindings for C, Fortran, Matlab, and Python. preCICE (`https://github.com/precice/`) is an open-source software released under the LGPL3 license.

4.2 Compilation

TODO

4.3 input Files

TODO

4.3.1 DUST

The DUST component input card is enriched of the following parameters:

- `coupled` *required:* no. *default:* F. *type:* logical.

  component of a coupled simulation with respect to a structural solver.

- `coupling_type` *required:* yes. *type:* string. type of the coupling:

  - `rigid`: rigid component and node
  - `rbf`: Generic DUST component and node coupling

  Although three different types of are implemented, the most general one is the `rbf` since it can manage all DUST components and can be coupled with both rigid and flexible elements.

- `coupling_node` *required:* no. *type:* real array, length 3. *default:* (0.0, 0.0, 0.0)

  Node for `rigid` coupling in the reference configuration (x, y, z).

- `coupling_node_file` *required:* yes. *type:* string.

  File containing the nodes for FSI (fluid structure interaction). It is required for `rbf` coupling.

- `coupling_node_orientation` *required:* no. *type:* real array, length 3. *default:* (/1.,0.,0., 0.,1.,0., 0.,0.,1./)

  Orientation of the node for rigid coupling. This array contains the local components (in the local reference frame of the geometrical component) of the unit vectors of the coupling node reference frame. In the `rbf` frame indicates the rotation matrix from the structural component to the DUST component reference frame.

In the DUST solver input file, a new keyword is added:

- `precice_config` *required:* no. *default:* ./../precice-config.xml. *type:* string.

  Path of the preCICE XML configuration file.

4.3.2 preCICE XML

preCICE needs to be configured at runtime via an `xml` file, typically named `precice-config.xml`. Here, you specify which solvers participate in the coupled simulation, which coupling data values they exchange, which fixed-point acceleration and many other things.

First of all, the fields exposed to communication between the two solvers are declared.

```
<?xml version="1.0"?>

<precice-configuration>

  <solver-interface dimensions="3">

    <!-- === Data ========================================== -->

    <data:vector name="Position" />
    <data:vector name="rotation" />
    <data:vector name="Velocity" />
    <data:vector name="AngularVelocity" />
    <data:vector name="Force" />
    <data:vector name="Moment" />
```

Here follows the definition of the two meshes, for each of which it is declared which data among the declared before are used. In this case, both DUST and MBDyn use all the exposed fields.

```
    <!-- === Mesh ========================================== -->
    <mesh name="MBDynNodes">
      <use-data name="Position" />
      <use-data name="rotation" />
      <use-data name="Velocity" />
      <use-data name="AngularVelocity" />
      <use-data name="Force" />
      <use-data name="Moment" />
    </mesh>

    <mesh name="dust_mesh">
      <use-data name="Position" />
```

```
    <use-data name="rotation" />
    <use-data name="Velocity" />
    <use-data name="AngularVelocity" />
    <use-data name="Force" />
    <use-data name="Moment" />
  </mesh>
```

In this part are declared the two participants, MBDyn and DUST. For each participant it is indicated which field is received (`read-data`) and which is sent (`write-data`). Considering DUST, this receives the kinematic variables and sends the loads.

```
  <!-- === Participants ================================== -->
  <participant name="MBDyn">
    <use-mesh name="MBDynNodes" provide="yes"/>
    <write-data name="Position" mesh="MBDynNodes" />
    <write-data name="rotation" mesh="MBDynNodes" />
    <write-data name="Velocity" mesh="MBDynNodes" />
    <write-data name="AngularVelocity" mesh="MBDynNodes" />
    <read-data name="Force" mesh="MBDynNodes" />
    <read-data name="Moment" mesh="MBDynNodes" />
  </participant>

  <participant name="dust">
    <use-mesh name="dust_mesh" provide="yes" />
    <use-mesh name="MBDynNodes" from="MBDyn" />
    <write-data name="Force" mesh="dust_mesh" />
    <write-data name="Moment" mesh="dust_mesh" />
    <read-data name="Position" mesh="dust_mesh" />
    <read-data name="rotation" mesh="dust_mesh" />
    <read-data name="Velocity" mesh="dust_mesh" />
    <read-data name="AngularVelocity" mesh="dust_mesh" />
    <mapping:nearest-neighbor direction="read" from="MBDynNodes" to="dust_mesh"
      constraint="consistent" />
    <mapping:nearest-neighbor direction="write" from="dust_mesh" to="MBDynNodes"
      constraint="conservative" />
  </participant>
```

For each two participants that should exchange data, you have to define an m2n communication. This establishes an m2n (i.e. parallel, from the M processes of the one participant to the N processes of the other) communication channel based on TCP/IP sockets between MBDyn and DUST.

```
  <!-- === Communication ================================= -->
  <m2n:sockets exchange-directory="./../" from="MBDyn" to="dust"/>
```

A coupling scheme can be either serial or parallel and either explicit or implicit. Serial refers to the staggered execution of one participant after the other where the first participant is computed before the second one. With an explicit scheme, both participants are only executed once per time window. With an implicit scheme, the participants are executed multiple times until convergence.

The `max-time value` field indicates the maximum time (end time) for the coupled simulation (NOTE: actually the final time is the shorter between this and the final time set in DUST and in the MBDyn input).

With `time-window-size value`, you can define the coupling time window (=coupling time step) size. If a participant uses a smaller one, it will subcycle until this window size is reached. Setting it equal to -1, it is set according to a specific `method`, here taking the value form the first participant MBDyn.

To control the number of sub-iterations within an implicit coupling loop, you can specify the maximum number of iterations, `max-iterations` and you can specify one or several convergence measures:

- `relative-convergence-measure` for a relative criterion

- `absolute-convergence-measure` for an absolute criterion
- `min-iteration-convergence-measure` to require a minimum of iterations

```
  <!-- === Coupling scheme ================================ -->
  <coupling-scheme:serial-implicit>
    <participants first="MBDyn" second="dust" />
    <max-time value="100.0" />
    <time-window-size value="-1" valid-digits="10" method="first-participant" />
    <exchange data="Position" from="MBDyn" mesh="MBDynNodes" to="dust" />
    <exchange data="rotation" from="MBDyn" mesh="MBDynNodes" to="dust" />
    <exchange data="Velocity" from="MBDyn" mesh="MBDynNodes" to="dust" />
    <exchange data="AngularVelocity" from="MBDyn" mesh="MBDynNodes" to="dust" />
    <exchange data="Force" from="dust" mesh="MBDynNodes" to="MBDyn" />
    <exchange data="Moment" from="dust" mesh="MBDynNodes" to="MBDyn" />
    <max-iterations value="60"/>
    <absolute-convergence-measure limit="1.0e-4" data="Position" mesh="MBDynNodes" />
    <absolute-convergence-measure limit="1.0e-3" data="rotation" mesh="MBDynNodes" />
    <absolute-convergence-measure limit="1.0e-3" data="Velocity" mesh="MBDynNodes" />
    <absolute-convergence-measure limit="1.0e-3" data="AngularVelocity" mesh="MBDynNodes" />
```

Mathematically, implicit coupling schemes lead to fixed-point equations at the coupling interface. A pure implicit coupling without acceleration corresponds to a simple fixed-point iteration, which still has the same stability issues as an explicit coupling. We need acceleration techniques to stabilize and accelerate the fixed-point iteration. In preCICE, three different types of acceleration can be configured: constant (`constant` under-relaxation), `aitken` (adaptive under-relaxation), and various quasi-Newton variants (`IQN-ILS` aka. Anderson acceleration, `IQN-IMVJ` aka. generalized Broyden).

```
    <acceleration:aitken>
      <data name="Force" mesh="MBDynNodes"/>
      <initial-relaxation value="0.1"/>
    </acceleration:aitken>
  </coupling-scheme:serial-implicit>

 </solver-interface>

</precice-configuration>
```

For more details, see `https://precice.org`.

4.3.3 MBDyn external force

The aerodynamic loads computed by DUST are introduced in the MBDyn model as an external structural force acting on some nodes.

4.4 Coupling workflow

In this section the workflow to build a coupled model will be deeply explained considering the cases of a generic aircraft with a fuselage (modelled as rigid body) and a flexible wing, and the case of flexible three bladed rotor. Both examples will use the `rbf` coupling type.

4.4.1 Wing with Control Surfaces

Let consider the case of a symmetric wing. According to MBDyn manual, it is easier to define the beam axis oriented with the local x. Let define the angles λ, δ, θ as the sweep, dihedral and twist angle. In order to pass

from the global or wind axis frame of reference to the local beam axis the following rotation matrix are used:

- rotation matrix from the DUST reference frame to the MBDyn reference frame:

$$\mathbf{R}_b = \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4.1}$$

- twist rotation of the MBDyn node around the beam axis x.

$$\mathbf{R}_\theta = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix} \tag{4.2}$$

- sweep rotation of the MBDyn node. The sweep is taken positive for a rotation about $-z$.

$$\mathbf{R}_\lambda = \begin{bmatrix} \cos(\lambda) & -\sin(\lambda) & 0 \\ \sin(\lambda) & \cos(\lambda) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4.3}$$

- Dihedral rotation of the MBDyn node.

$$\mathbf{R}_\delta = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\delta) & \sin(\delta) \\ 0 & -\sin(\delta) & \cos(\delta) \end{bmatrix} \tag{4.4}$$

Generic rotation matrix for a right wing:

$$\begin{aligned} \mathbf{R}_{\text{right}} &= \mathbf{R}_b\mathbf{R}_\theta\mathbf{R}_\lambda\mathbf{R}_\delta = \\ &= \begin{bmatrix} \sin(\lambda) & \cos(\delta)\cos(\lambda) & \cos(\lambda)\sin(\delta) \\ -\cos(\lambda)\cos(\theta) & \sin(\delta)\sin(\theta)+\cos(\delta)\cos(\theta)\sin(\lambda) & \sin(\delta)\cos(\theta)\sin(\lambda)-\cos(\delta)\sin(\theta) \\ -\cos(\lambda)\sin(\theta) & \cos(\delta)\sin(\lambda)\sin(\theta)-\sin(\delta)\cos(\theta) & \cos(\delta)\cos(\theta)+\sin(\delta)\sin(\lambda)\sin(\theta) \end{bmatrix} \end{aligned}$$

Generic rotation matrix for a left wing:

$$\begin{aligned} \mathbf{R}_{\text{left}} &= \mathbf{R}_b\mathbf{R}_\theta\mathbf{R}_\lambda^T\mathbf{R}_\delta^T = \\ &= \begin{bmatrix} -\sin(\lambda) & \cos(\delta)\cos(\lambda) & -\cos(\lambda)\sin(\delta) \\ -\cos(\lambda)\cos(\theta) & -\sin(\delta)\sin(\theta)-\cos(\delta)\cos(\theta)\sin(\lambda) & \sin(\delta)\cos(\theta)\sin(\lambda)-\cos(\delta)\sin(\theta) \\ -\cos(\lambda)\sin(\theta) & \sin(\delta)\cos(\theta)-\cos(\delta)\sin(\lambda)\sin(\theta) & \cos(\delta)\cos(\theta)+\sin(\delta)\sin(\lambda)\sin(\theta) \end{bmatrix} \end{aligned}$$

All these rotations are intended to rotate back the structural component in the DUST component reference frame. When this procedure is performed, then the aerodynamic mesh can be constructed as a normal DUST parametric mesh. TODO

Chapter 5

DUST solver

The DUST solver is the main executable of DUST, and its aim is to run the actual simulation and obtain the required full solution to the problem.

As the other executable it is run by simply invoking the executable `dust` with the correct input file as command argument.

```
dust input_file_name.in
```

Command 5.3: Solver command looking for input file `input_file_name.in`

If no command argument is provided the solver looks for a default input file, `dust.in`.

```
dust
```

Command 5.4: Solver command looking for default input file `dust.in`

The solver requires, among other inputs, the geometry file result of the preprocessor. If such file is not found the solver attempts to run the preprocessor by executing the preprocessor with default input file (command 3.2).

5.1 Input file

The input file which should be provided to the solver is used to set all t he parameters required for the execution of the simulation, from execution options, time parameters to the wake and model settings.

An example of input file for the solver is presented in file 5.1

input file 5.1: dust.in

```
! --- execution parameters ---
basename = sim_01
basename_debug = sim_01_debug
debug_level = 3

! --- time parameters ---
tstart = 0.0
tend = 10.0
timesteps = 20
!dt = 0.5
dt_out = 2.0
```

```
dt_debug_out = 1.0
output_start = F
output_detailed_geo = T

! --- restart parameters ---
restart_from_file = F
restart_file = prev_solution_0021.h5
reset_time = F

! --- input files ---
reference_file = References.in
geometry_file = geo_file.h5

! --- stream parameters ---
u_inf = (/1.0, 0.0, 0.0/)
u_ref = 1.0
P_inf = 1.0
rho_inf = 1.0
a_inf = 340.0
mu_inf = 0.00001

! --- wake parameters ---
n_wake_panels = 4
n_wake_particles = 10000
particles_box_min = (/-10.0, -10.0, -10.0/)
particles_box_max = (/10.0, 10.0, 10.0/)
implicit_panel_scale = 0.3
implicit_panel_min_vel = 1.0e-8
rigid_wake = F
rigid_wake_vel = (/1.0, 0.0, 0.0/)
join_te = F
join_te_factor = 1.0

! --- model parameters ---
far_field_ratio_doublet = 10.0
far_field_ratio_source = 10.0
doublet_threshold = 1.0e-6
rankine_rad = 0.1
vortex_rad = 0.1
k_vortex_rad = 1.0
cutoff_rad = 0.001

! --- additional models to use ---
vortstretch = T
vortstretch_from_elems = F
diffusion = T
turbulent_viscosity = F
penetration_avoidance = F
penetration_avoidance_check_radius = 5.0
penetration_avoidance_element_radius = 1.5
divergence_filtering = T
filter_time_scale = 40.0
fmm = T
fmm_panels = F
viscosity_effects = F
particles_redistribution = F
```

```
particles_redistribution_ratio = 3.0
octree_level_solid = 4
turbulent_viscosity = F
HCAS = F
HCAS_time = 4.0
HCAS_velocity = (/2.0, 0.0, 0.0/)

! --- Fast multipole parameters ---
box_length = 10.0
n_box = (/2, 2, 2/)
octree_origin = (/-10.0, -10.0, -10.0/)
n_octree_levels = 6
min_octree_part = 10
multipole_degree = 2
dyn_layers = F
nmax_octree_levels = 7
leaves_time_ratio = 3.0

! --- Lifting Lines parameters ---
ll_solver = GammaMethod
ll_reynolds_corrections = F
ll_reynolds_corrections_nfact = 0.2
ll_max_iter = 100
ll_tol = 1.0e-6
ll_damp = 5.0
ll_stall_regularisation = T
ll_stall_regularisation_nelems = 1
ll_stall_regularizations_niters = 1
ll_stall_regularization_alpha_stall = 15
ll_artificial_viscosity = 0.0
ll_artificial_viscosity_adaptive = F
ll_artificial_viscosity_adaptive_alpha = 15
ll_artificial_viscosity_adaptive_dalpha = 3
ll_loads_avl = F

! --- Vortex Lattice Correction ---
vl_tol = 1.0e-4
vl_relax = 0.3
vl_maxiter = 100
vl_start_step = 0
vl_dynstall = F
aitken_relaxation = T
```

The details of each parameter are:

- `basename`: *required:* yes. *multiple:* no. *type:* string.

  The base name of the simulation and will be the prefix of the output files. Can be a complete path leading to the location in which to store the results.

- `basename_debug`: *required:* no. *multiple:* no. *default:* `basename` *type:* string.

  The base name of the debug output. It is especially useful to direct the debug output to a different location than the standard output. The amount of (if any) debug output is controlled by `debug_level`.

- `debug_level`: *required:* no. *multiple:* no. *default:* 1*type:* integer.

  A parameter controlling the verbosity (both in terms of standard output and files output) of the simulation. A smaller value leads to fewer output, with 0 being a silent simulation, while values higher of 10

lead to debug files to be produced. A value of 3 is a balanced value to obtain a reasonable amount of informations on the simulation.

WARNING: the present feature tends to evolve rapidly during the development. Values higher than 10 may lead to unexpected behaviours.

- `tstart`: *required:* yes. *multiple:* no. *type:* real.

  The start time of the simulation.

- `tend`: *required:* yes. *multiple:* no. *type:* real.

  The end time of the simulation.

- `timesteps`: *required:* yes if `dt` not set, forbidden if set. *multiple:* no. *type:* integer.

  The number of timesteps for the simulation. Note that during the simulation a number of timesteps equal to `timesteps`+1 will be displayed, because the starting step is counted too.

- `dt`: *required:* yes if `timesteps` not set, forbidden if set. *multiple:* no. *type:* real.

  The time step length.

- `dt_out`: *required:* yes. *multiple:* no. *type:* real.

  Time interval of solution output.

- `dt_debug_out`: *required:* no. *multiple:* no. *default:* `dt_out`. *type:* real.

  Time interval of debug output, if `debug_level` is set high enough to write debug files.

- `output_start`: *required:* no. *multiple:* no. *default* False. *type:* logical.

  Output the first iteration. Since it is an implicit solver, the first calculation happens at t_0 before any time interval has passed, if set to True output the first result.

- `output_detailed_geo`: *required:* no. *multiple:* no. *default* False. *type:* logical.

  Output additional geometry information in the result files. Makes the result files easier to interpret by third party software, however makes the result files slightly bigger and output slightly slower.

- `restart_from_file`: *required:* no. *multiple:* no. *default:* False. *type:* logical.

  Restart the simulation from a previous result.

- `restart_file`: *required:* yes if `restart_from_file` is true. *multiple:* no. *type:* string.

  name of the solution file from which to restart the simulation.

  Note: when reloading a certain result, the solver will attempt to load also the relative geometry file in the same location, as discussed in 5.9. If such geometry file is not provided the solver, as in the case of standard starting, will attempt to generate such file with the default preprocessor input. Beware that in this case, if the geometry of the results from which the solver is starting was generated from another geometry input this could lead to unexpected behaviour.

- `reset_time`: *required:* no. *multiple:* no. *default:* False. *type:* logical.

  If set to true, when restarting from a previous result, the time will be set as the `tstart` from the input file. If set to false, the time will be set as the one from the result file loaded. This is delicate since the movement of the reference frames is based on the current time in the simulation, and if reset the time could lead to a position of the geometry different from the one in the loaded result file.

- `reference_file`: *required:* yes. *multiple:* no. *type:* string.

  name of the file containing the definition of the reference frames employed in the simulation.

- `geometry_file`: *required:* yes. *multiple:* no. *type:* string.

  name of the file containing the geometry definition, generated by the preprocessor

- `u_inf`: *required:* no. *multiple:* no. *default:* (/1.0, 0.0, 0.0/) *type:* real array of length 3.

  Free stream velocity vector.

- `u_ref`: *required:* no. *multiple:* no. *default:* |`u_inf`| *type:* real.

  Reference velocity. Usually is safe not to set anything and use the norm of the free stream velocity, however in hover flight conditions it is useful to set a non zero reference velocity (e.g. the blade tip velocity)

- `gust`: *required:* no. *multiple:* no. *default:* F *type:* logical.

  - `gust_type`: *required:* no. *multiple:* no. *default:* AMC *type:* string.
    Select the gust type: `ACM` or `linear`.

$$\boldsymbol{s} = -\sum \left[\boldsymbol{x} - \left(\boldsymbol{O} + U_{des}\boldsymbol{v}(t-\tau)\right)\right]\boldsymbol{v} \tag{5.1}$$

$$\boldsymbol{U} = \boldsymbol{U} + \frac{U_{des}}{2}\left(1 - \cos\left(\frac{\pi s}{H}\right)\right) \tag{5.2}$$

  - `gust_origin`: *required:* yes. *multiple:* no. *type:* real array of length 3.
    0: Position of the point whose airstream velocity is being computed.
  - `gust_front_direction`: *required:* yes. *multiple:* no. *type:* real array of length 3.
    $\boldsymbol{v}$: Unit vector that defines the direction of propagation of the front.
  - `gust_front_speed`: *required:* yes. *multiple:* no. *type:* real.
    Velocity of propagation of the front in direction `gust_front_direction`
  - `gust_u_des`: *required:* yes. *multiple:* no. *type:* real.
    U_{des}: gust amplitude.
  - `gust_perturbation_direction`: *required:* no. *multiple:* no. *default:* (/0.0, 0.0, 1.0/) *type:* real array of length 3.
    Unit vector that defines the direction of the velocity perturbation
  - `gust_gradient`: *required:* no. *multiple:* yes. *type:* real.
    H: gust wave length.
  - `gust_start_time`: *required:* no. *multiple:* no. *default:* 0.0 *type:* real.
    τ: Time at which the gust starts.

- `P_inf`: *required:* no. *multiple:* no. *default:* 101325.0 *type:* real.

  Free stream pressure.

- `a_inf`: *required:* no. *multiple:* no. *default:* 340.0 *type:* real.

  Free stream sound speed. For lifting lines and compressibility corrections.

- `mu_inf`: *required:* no. *multiple:* no. *default:* 0.000018 *type:* real.

  Free stream dynamical viscosity. For lifting lines and viscosity corrections.

- `rho_inf`: *required:* no. *multiple:* no. *default:* 1.225 *type:* real.

  Free stream density.

- `n_wake_panels`: *required:* no. *multiple:* no. *default:* 1 *type:* integer.

  Number of panel wake rows before converting the wake to vortex particles.

- `n_wake_particles`: *required:* no. *multiple:* no. *default:* 10000 *type:* integer.

  Number of maximum vortex particles in the wake. If the limit is reached the simulation is stopped, but can be restarted from last save with a higher value of the parameter.

- `particles_box_min`: *required:* no. *multiple:* no. *default:* (/-10.0, -10.0, -10.0/) *type:* real array of length 3.

  Minimum coordinates (in the base reference frame) of the box containing the vortex particles. If the particles get outside the box are cancelled.

- `particles_box_max`: *required:* no. *multiple:* no. *default:* (/10.0, 10.0, 10.0/) *type:* real array of length 3.

  Maximum coordinates (in the base reference frame) of the box containing the vortex particles. If the particles get outside the box are cancelled.

- `implicit_panel_scale`: *required:* no. *multiple:* no. *default:* 0.3 *type:* real.

  Scaling of the first panel of the wake, the implicit one which enforces the Kutta condition. The first panel geometry is not dictated by the advection of its vertices as for the other ones, but the first row of points come from the geometry, while the second row is calculated by local velocity and timestep, scaled by the `implicit_panel_scale` parameter.

- `implicit_panel_min_vel`: *required:* no. *multiple:* no. *default:* 1.0e-8 *type:* real.

  When the free stream velocity and body motions do not create a sufficiently high local velocity at the trailing edge, the first panel might turn out of zero length. To avoid this, when the trailing edge local velocity is below `implicit_panel_min_vel` a fixed length of the panel is imposed.

- `rigid_wake`: *required:* no. *multiple:* no. *default:* False *type:* logical.

  Impose a wake which is evolved rigidly according to a prescribed global velocity rather than the local velocity field. Works for both panels and particles.

- `rigid_wake_vel`: *required:* yes if `rigid_wake` is True. *multiple:* no. *type:* real array of length 3.

  Velocity to impose to the rigid wake.

- `join_te`: *required:* no. *multiple:* no. *default:* False. *type:* logical.

  Employ trailing edge joining for close trailing edges.

- `join_te_factor`: *required:* no. *multiple:* no. *default:* 1.0. *type:* real.

  If employing joining of trailing edges, the trailing edges which are closer than `join_te_factor` times the length of the adjacent trailing edge elements will be joined.

- `far_field_ratio_doublet`: *required:* no. *multiple:* no. *default:* 10.0 *type:* real.

  Ratio with respect to element length to set the thresholds for far field approximation. When evaluating the influence of the doublets of an element, if the evaluation point is distant more than `far_field_ratio_doublet` times the characteristic length of the element, simplified cheaper far field approximated formulae are employed instead of the standard ones. The characteristic length of the element is taken as the maximum length of all the element edges.

- `far_field_ratio_source`: *required:* no. *multiple:* no. *default:* 10.0 *type:* real.

  As for `far_field_ratio_doublet` determines the threshold after which far field approximations are employed, just for sources. Applies only to three dimensional surface panels.

- `doublet_threshold`: *required:* no. *multiple:* no. *default:* 1.0e-6 *type:* real.

  Parameter which sets the distance threshold under which the evaluation point, with respect to a panel, is considered inside the plane of the panel.

- `rankine_rad`: *required:* no. *multiple:* no. *default:* 0.1 *type:* real.

  Parameter which sets the radius under which the Rankine approximation of vortexes cores is employed. Used for aerodynamic elements and panels (i.e. everything except vortex particles)

- `vortex_rad`: *required:* no. *multiple:* no. *default:* 0.1 *type:* real.

  Parameter which uniformly sets the radius of the vortex particles, only used if `k_vortex_rad` is disabled.

- `k_vortex_rad`: *required:* no. *multiple:* no. *default:* 1.0 *type:* real.

  Coefficient for the automatic computation of the radius of each vortex particle; set a negative number to disable the feature and revert to uniform vortex radius.

- `cutoff_rad`: *required:* no. *multiple:* no. *default:* 0.001 *type:* real.

  Parameter which sets the radius under which the vortexes interaction is completely set to zero.

- `vortstretch`: *required:* no. *multiple:* no. *default:* T *type:* logical.

  Calculate the evolution of vorticity of the particles considering the vortex stretching

- `vortstretch_from_elems`: *required:* no. *multiple:* no. *default:* F *type:* logical.

  Compute also the contribution to the vortex stretching of the particles due to the model elements.

- `diffusion`: *required:* no. *multiple:* no. *default:* T *type:* logical.

  Calculate the evolution of vorticity of the particles considering the vorticity diffusion

- `turbulent_viscosity`: *required:* no. *multiple:* no. *default:* F *type:* logical.

  Introduce additional turbulent viscosity (Smagorinsky style) to the vorticity diffusion term. Working only when `fmm` is turned on. Still experimental feature.

- `penetration_avoidance`: *required:* no. *multiple:* no. *default:* F *type:* logical.

  Apply the penetration avoidance algorithm to avoid the penetration of particles inside the solid bodies.

- `penetration_avoidance_check_radius`: *required:* no. *multiple:* no. *default:* 5.0 *type:* real.

  radius multiplication factor of the check radius. All the particles within a distance $d \leq P_r U_{ref} \Delta t$ from each element are checked for potential penetration, where P_r is the multiplication factor. A bigger factor minimizes the risk of penetration of extremely fast particles, but affects the performance.

- `penetration_avoidance_element_radius`: *required:* no. *multiple:* no. *default:* 1.5 *type:* real.

  Surface correction element radius multiplication factor. The velocity of the particles is corrected to avoid penetration through a particular element if the particle hits the surface within a distance (tangent to the surface) $d_t \leq P_e \max\{l_i\} \frac{\sqrt{2}}{2}$ from the element centre, where l_i are the lenght of the element sides and P_e is the factor. A bigger factor leads to a potentially higher level of correction (concurrent correction from different neighbouring elements) while a smaller factor may let some particles slip through the surface.

- `divergence_filtering`: *required:* no. *multiple:* no. *default:* T *type:* logical.

  Employ the divergence filtering to keep the vorticity field divergence-free.

- `filter_time_scale`: *required:* no. *multiple:* no. *default:* 40.0 *type:* real.

  Timescale of the time filter to filter the divergence. The input is not the actual timescale but the number of simulation timesteps of which the timescale is consisting.

- `fmm`: *required:* no. *multiple:* no. *default:* T *type:* logical.

  Use the fast multipole method for particles evolution.

- `fmm_panels`: *required:* no. *multiple:* no. *default:* F *type:* logical.

  Use the fast multipole method also to compute interactions of the particles on the solid bodies panels.

- `viscosity_effects`: *required:* no. *multiple:* no. *default:* F *type:* logical.

  Take into account viscosity effects on the geometry surface: enables the release of vortex particles from different points on the geometry surface. Experimental feature, still in development.

- `particles_redistribution`: *required:* no. *multiple:* no. *default:* F *type:* logical.

  Redistribute the particles having a small intensity to the neighbouring ones. Active only if `fmm` is active, otherwise ignored. Redistributed particles are then ereased to reduce the total number of particles.

- `particles_redistribution_ratio`: *required:* no. *multiple:* no. *default:* 3.0 *type:* real.

  When employing `particles_redistribution`, each particle of intensity $|\boldsymbol{\alpha}_i|$ will be redistributed to the other particles contained in the octree leaf cell if $|\boldsymbol{\alpha}_i| r < \alpha_{ave}$ where r is `particles_redistribution_ratio` and α_{ave} is the average intensity of the particles inside the cell.

- `octree_level_solid`: *required:* yes, if `particles_redistribution` is true. *multiple:* no. *type:* integer.

  When employing `particles_redistribution`, the particles too near to a solid boundary are not redistributed to reduce the error induced by the redistribution on the solution on solid surfaces. The octree cells at level `octree_level_solid` containing a panel (SP, VL or LL) are marked as containing a solid boudary, as well as all their neighbour at that level. All the children of cells containing a solid boundary are marked as containing a solid boundary. In the cells containing a solid boundary the redistribution is not performed.

  For this reason, a small `octree_level_solid` approaching 1 will lead to a wide zone around solid bodies in which redistribution does not occur, while a `octree_level_solid` approaching `n_octree_levels` leads to redistribution to be performed in areas up to close to the solid bodies.

- `turbulent_viscosity`: *required:* no. *multiple:* no. *default:* F *type:* logical.

  Employ an additional turbulent viscosity, using a Smagorinsky-like model to take into account the dissipation of energy in turbulent conditions towards small, not resolved turbulent scales. It works only in an octree environment, so when `fmm` is active.

- `HCAS`: *required:* no. *multiple:* no. *default:* F *type:* logical.

  Hover Convergence Augmentation System. An additional velocity is applied to the particles in the wake to let the particles generated in the transient condition at the start to be pushed faster outside the domain, and reach stationary conditions faster. The velocity defined by `HCAS_velocity` is applied full only at the beginning of the simulation, and is then decreased linearly in `HCAS_time`. After `HCAS_time` after the simulation start the additional velocity is not applied anymore and the simulation carries on without tampering.

- `HCAS_time`: *required:* yes if `HCAS` is true. *multiple:* no. *type:* real.

  Duration of HCAS application

- `HCAS_velocity`: *required:* yes if `HCAS` is true. *multiple:* no. *type:* real array of length 3.

  Velocity to apply to the particles during the HCAS use.

- `box_length`: *required:* yes if `fmm` is true. *multiple:* no. *type:* real.

  Length of the level 1 cubic boxes composing the octree. See figure 5.1 for a graphic explanation.

- `n_box`: *required:* yes if `fmm` is true. *multiple:* no. *type:* integer array of length 3.

  Number of base level 1 cubic boxes in each spatial direction. See figure 5.1 for a graphic explanation.

- `octree_origin`: *required:* yes if `fmm` is true. *multiple:* no. *type:* real array of length 3.

  origin of the octree. Starting from the origin, the octree mesh extends in each direction of `n_box` times `box_length`. See figure 5.1 for a graphic explanation.

- `n_octree_levels`: *required:* yes if `fmm` is true. *multiple:* no. *type:* integer.

  Number of levels in which the base boxes are divided. At each level the upper level boxes are divided into eight half sized boxes.

- `min_octree_part`: *required:* yes if `fmm` is true. *multiple:* no. *type:* integer.

  Minimum number of particles contained in an octree box in order to consider it a leaf (lowest level box). If not enough particles are contained in a box, the box is not considered and the particles are gathered at the higher level parent box.

- `multipole_degree`: *required:* yes if `fmm` is true. *multiple:* no. *type:* integer.

  Degree of the expansions in the multipole method.

- `dynamic_layers`: *required:* no. *multiple:* no. *default:* False. *type:* logical.

  Use dynamic octree layers, i.e. a further division layer in the octree is added every time the time spent in the particle to particle calculations is greater than the one spent in the fast multipole part.

- `nmax_octree_levels`: *required:* yes if `dynamic_layers` is True. *multiple:* no. *type:* integer.

  Maximum number of divisions allowed during dynamic layers. The number of starting layers is still `n_octree_levels`, which however might increase during the simulation, but are always kept under a maximum number.

  WARNING: it is advised not to let the number of octree layers to be greater than 7. The number of cells in the octree increases really rapidly at each subdivision, and after 7 layers it is easy to fill the complete memory just with the pointers necessary to keep the track of the connectivity.

- `leaves_time_ratio`: *required:* yes if `dynamic_layers` is True. *multiple:* no. *type:* real.

  Ratio between the time spent in the particle to particle computations in the leaves of the octree with respect to the rest of the fast multipole computations that triggers the increase of the octree levels.

- `ll_solver`: *required:* no. *multiple:* no. *default:* `GammaMethod`. *type:* string.

  Choose the type of lifting line solver to be employed. `gammma_method` performs a fixed point iteration method with the lifting line circulation as the unknown variable. It is the default method.

  `alpha_method` performs a similar fixed point method but with the angle of attack of the lifting line section as variable. It allows for the use of regularization in case of problematic partially stalled configurations.

- `ll_reynolds_corrections`: *required:* no. *multiple:* no. *default:* False. *type:* logical.

  Employ a Reynolds number correction to obtain an extrapolation of the lifting lines tables at the simulation conditions Reynolds number if different than the one(s) provided in the lookup tables

- `ll_reynolds_corrections_nfact`: *required:* no. *multiple:* no. *default:* 0.2 *type:* real.

  The Reynolds correction extrapolation is based on a power law: $Cf = Cf_T \left(\frac{Re}{Re_T}\right)^N$, the parameter provides the value of the power law. Usually it can vary from 0.125 to 0.2.

- `ll_max_iter`: *required:* no. *multiple:* no. *default:* 100 *type:* integer.

  Number of iterations of the fixed point non-linear solver used to obtain the lifting lines solution.

- `ll_tol`: *required:* no. *multiple:* no. *default:* 1.0e-6 *type:* real.

  Relative tolerance at which the fixed point lifting lines solver stops

- `ll_damp`: *required:* no. *multiple:* no. *default:* 25.0 *type:* real.

  Value of the damping (relaxation) coefficient employed during fixed point iterations, to suppress possible oscillations.

- `ll_stall_regularisation`: *required:* no. *multiple:* no. *default:* True. *type:* logical.

  During the first timesteps of simulations employing lifting lines, usually in challenging configurations such as hovering rotors, it is possible that a non perfect convergence of the non-linear solver leads to some few elements of the lifting line (typically one) to converge on a stalled configuration among a series of completely non stalled elements. This is unphysical, and usually it is solved during the evolution of the simulation. It is however possible to identify this behaviour and use the previous solution for these sections to improve convergence.

- `ll_stall_regularisation_nelems`: *required:* no. *multiple:* no. *default:* 1. *type:* integer.

  Number of lifting line elements to correct in case of isolated stall among non stalled elements. At the moment cannot be higher than 1.

- `ll_stall_regularisation_niters`: *required:* no. *multiple:* no. *default:* 1. *type:* integer.

  Number of LL iterations between two regularisation processes.

- `ll_stall_regularisation_alpha_stall` *required:* no. *multiple:* no. *default:* 15.0. *type:* real.

  Stall angle [deg] used as a threshold for regularisation process.

- `ll_loads_avl`: *required:* no. *multiple:* no. *default:* F. *type:* logical.

  Use AVL expression for the inviscid load contributions of LL elements, in the same way of load computation used for VL elements.

- `ll_artificial_viscosity`: *required:* no. *multiple:* no. *default:* 0.0. *type:* real.

  Artificial viscosity used to spatially regularize the solution with a gaussian kernel, to be used in case of `ll_solver` = `alpha_method` to regularize post-stall situations. The default value of 0 leads to no regularization, while values greater than 0 lead to wider gaussian regularization. The value is uniform in the domain if `ll_artificial_viscosity_adaptive` is false, while if true represents the maximum introduced value.

- `ll_artificial_viscosity_adaptive`: *required:* no. *multiple:* no. *default:* False. *type:* logical.

  Use an adaptive strategy to introduce artificial viscosity for regularization, in order to regularize post stall configuration while not influencing non stalled configurations.

- `ll_artificial_viscosity_adaptive_alpha`: *required:* yes if `ll_artificial_viscosity_adaptive` is true. *multiple:* no. *type:* real.

  Angle of attack after which the full artificial viscosity is introduced, thus after which the maximum regularization is operated. Should be set around or over the stall angle

- `ll_artificial_viscosity_adaptive_dAlpha`: *required:* yes if `ll_artificial_viscosity_adaptive` is true. *multiple:* no. *type:* real.

  Angle of attack range before `ll_artificial_viscosity_adaptive_alpha` where the artificial viscosity is gradually introduced from zero to the maximum value set in `ll_artificial_viscosity`.

- `vl_tol`: *required:* no. *multiple:* no. *default:* 1.0e-4 *type:* real. Tolerance for the absolute error on lift coefficient in fixed point iteration for corrected vortex lattice.

- `vl_relax`: *required:* no. *multiple:* no. *default:* 0.3 *type:* real. Constant relaxation factor for rhs update

- `aitken_relaxation`: *required:* no. *multiple:* no. *default:* T *type:* logical. Activate the Aitken acceleration and stabilization method involved during the vortex lattice fixed point iteration. The initial relaxation is the one specified in `vl_relax`.

- `vl_maxiter`: *required:* no. *multiple:* no. *default:* 100 *type:* integer. Max number of iteration for correction of vortex lattice

- `vl_start_step`: *required:* no. *multiple:* no. *default:* 1 *type:* integer. Time step to start correcting the vortex lattice element

- `vl_dynstall`: *required:* no. *multiple:* no. *default:* F *type:* logical. Activate Boeing[1] dynamic stall on the corrected `v` element. (Still experimental)

- `vl_average`: *required:* no. *multiple:* no. *default:* F *type:* logical. Average the solution between the iteration: this may further stabilize the solution in stalled conditions

- `vl_average_iter`: *required:* no. *multiple:* no. *default:* 10 *type:* integer. Number of iteration on which the average is computed.

[1] `https://apps.dtic.mil/sti/pdfs/AD0767240.pdf`

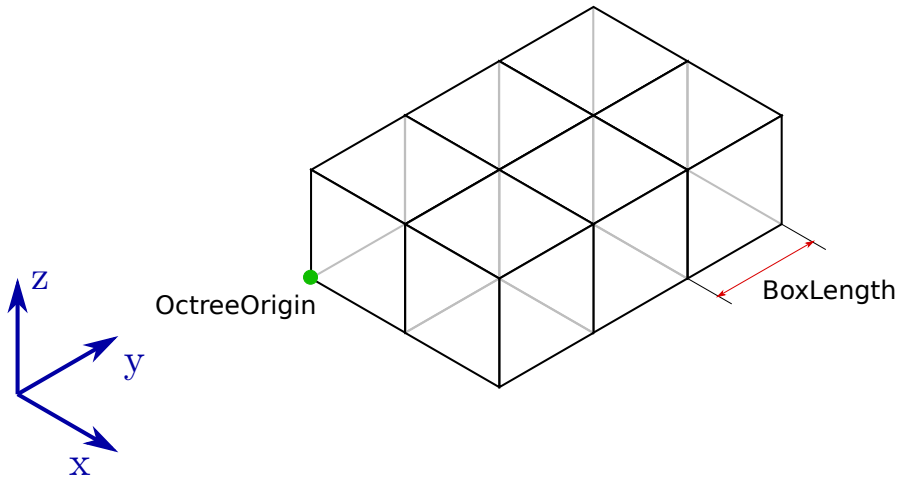

Figure 5.1: System of cube boxes to compose the first level of the octree. In green is highlighted the origin of the octree, from which two rows of boxes in x, three in y and one in z starts, corresponding to `n_box` = (/2, 3, 1/). In red is highlighted the length of each cube side, `box_length`

5.2 Time Stepping

The simulation is started from parameter `tstart` (unless restarting from a previous solution and not resetting time) and ends at `tend` after a certain number of timesteps. The user must provide the parameters `tstart`, `tend` and alternatively the number of timesteps (`timesteps`) between them or the size of the timesteps (`dt`). The non specified parameter is computed during the execution.

When specifying the number of timesteps, the interval between the starting time (or re-starting time) and the end time is divided in a number `timesteps` of equal intervals, and then `timesteps`+1 steps are executed, counting also the starting one.

When specifying the timestep length `dt`, the smallest number of steps to get to `tend` is executed. If the time interval cannot be divided into an integer number of timesteps, all the timesteps will be executed anyway with a timestep length of `dt`, and the last timestep will be shorter to arrive precisely at `tend`. It must however be noted that DUST is not designed at the moment to handle variable timestep lengths, for this reason in few particular unsteady configurations, this shorter timestep could lead to small fluctuations of the loads just in the last timestep. To avoid this issue use a `dt` which lead to a precise number of timesteps or prescribe the number of timesteps instead of the timestep length (or discard the results of the last step).

5.3 Models Parameters Choice

In the previous section 5.1 all the available parameters for the DUST solver have been listed. While the brief description in most of the cases is enough to describe the simple functioning of the parameters, for some model parameters is necessary a more comprehensive discussion.

Vortex Models Parameters In DUST all the description of both the surface elements and the wake relies (also) on vortex models. The vortex particles represent a small vortex unit, lifting lines are represented by a single vortex, panel wakes, vortex lattices and surface panels all have a uniform distribution of doublets, which is equivalent to a vortex ring along the sides of such panels.

The velocity induced by vortexes decreases with a certain power of the distance from the vortex ($1/r$ in two dimensions, $1/r^2$ in three), which means that while at higher distances the induced velocity becomes small, for distances approaching zero the velocity becomes very high and eventually singular. This model is completely irrotational, with all the vorticity confined in a point/line, but is clearly not physical, and for this reason it is

necessary to regularize the induced velocity in the proximity of the vortex. If in the case of the panels this can be seen merely as a regularization, in the case of the vortex particles the use of a regularized core is necessary to have a rotational volume for each particle which is able to represent the vorticity field generated by the wake.

Starting precisely from this consideration, it is advisable to set the parameter `vortex_rad`, which is the radius of the Rosenhead regularized core employed for the vortex particles, to a value which allow the generated particles to represent the whole generated vorticity field, by slightly overlapping one another. This can be done for example by setting `VotexRad` equal to the (average) length of the trailing edge element sides, where the particles are first generated and spaced.

Since the generation of the particles from the trailing edges sets somehow the spacing, and the resolution, of the vortical phenomena, the radius of the vortex model of the panels, `rankine_rad` should be chosen at least in the same order of magnitude of `vortex_rad`. The vortex regularization performed on the panels is the classical Rankine one. Since it is expected that the vortical phenomena on surfaces are more confined than in the wake the user might want to take a little smaller radius for the surface panels than the particles one, however remaining in the same order of magnitude. Too small radii might lead to not physical, too strong interactions with close particles or panels.

Eventually for the panels the parameter `cutoff_rad` sets a radius underneath which the interaction is set to zero, to avoid any kind of modelling when a point is essentially coinciding with the vortex. This parameter should be set *much smaller* than `rankine_rad`.

5.4 Reference frames

The reference frames are the basis for both the placement of the geometry components in the space and for the definition of their movement. Reference frames are handled by the solver, which reads them from a separate input file, indicated in the solver input file, file 5.1. Before detailing the inputs for the reference frames file it is important to understand how reference frames work inside DUST.

Reference frames are defined hierarchically from a base reference frame. The base reference frame is called "0" and cannot be defined, is a standard right-nanded Cartesian reference frame and is defined inside DUST. Starting from the base reference frame all the necessary reference frames can be defined.

It is important to stress that the base reference frame it is not defined upon any other implied reference frame and thus has no implied orientation with respect to anything else. Instead it is just the definition of the three axis x-y-z upon which all the other reference frames, geometrical components, parameters are defined. The user can give any meaning to the three axis, e.g. x horizontal towards rear, y horizontal towards right and z vertical upward, but also x front, y up and z right. The user must be then consistent with the inputs, both in terms of geometry and for example free stream velocity, but there is no implied orientation in the base reference.

All the following reference frames are defined upon another reference frame, called "parent" reference frames. Obviously the parent reference frame must always be defined, so that all the branches of the tree defined by the multiple reference frame can be traversed back to the base reference frame. The relative positioning of two reference frames with respect to the base one is depicted in figure 5.2. The use of multiple reference frames allows to position a geometrical entity, with its own local coordinates, in any position of the domain, and in any relative position with respect to the other components, in a logical way. An example of references file, for static references is presented in file 5.2.

input file 5.2: references_static.in

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
reference_tag = root
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 = F
```

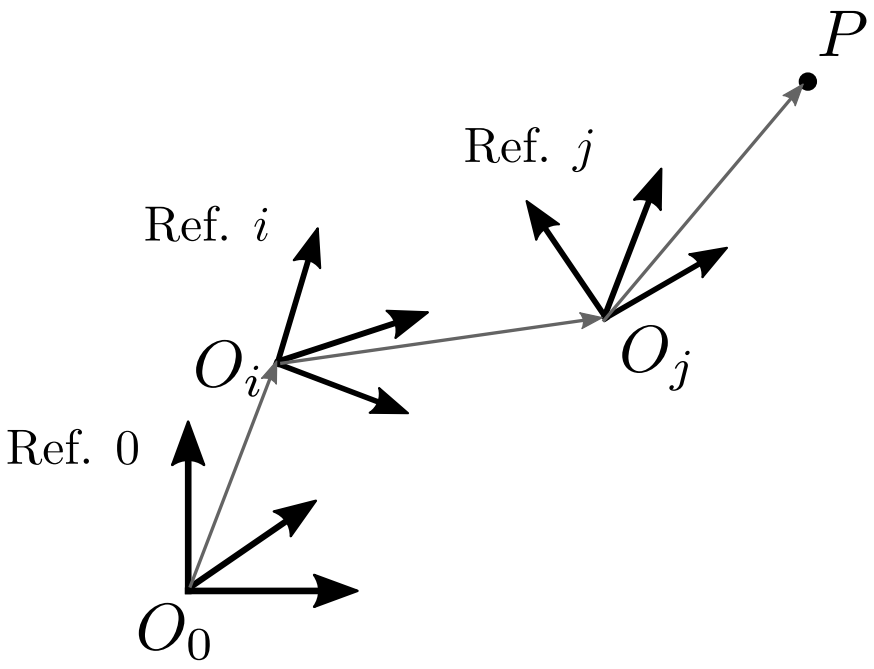

Figure 5.2: Positioning of relative reference frame

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
reference_tag = first
parent_tag = Root
origin = (/10.0, 2.0, 4.0/)
orientation = (/0.0,1.0,0.0, 0.0,0.0,1.0, 1.0,0.0,0.0/)
multiple = F
moving = F

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
reference_tag = second
parent_tag = Root
origin = (/0.0, 0.0, 0.0/)
orientation = (/0.999941,0.0,-0.010859, 0.0,1.0,0.0, +0.010859,0.0,0.999941/)
multiple = F
moving = F
```

The details of the parameters are:

- `reference_tag`: *required:* yes. *multiple:* yes. *type:* string.

  The tag is a unique identifier of the reference frame, it is a string and is referenced by the geometry component and other reference frames. All tags are valid except for "0" which is reserved by the base reference frame.

  The number of references tags declares the number of reference frames defined.

- `parent_tag`: *required:* yes, in same number as `reference_tag`. *multiple:* yes. *type:* string.

  Declares the reference frame upon which the current reference frame is defined. The valid values are all the other reference frames defined in the input file (also later in the file) and "0".

- `origin`: *required:* yes, in same number as `reference_tag`. *multiple:* yes. *type:* real array of length 3.

  origin of the current reference frame, in coordinates of the parent reference frame (not the global ones).

- `orientation`: *required:* yes, in same number as `reference_tag`. *multiple:* yes. *position:* must follow `origin` *type:* real array of length 9.

  orientation of the current reference frame with respect to the previous one. Considering a Fortran filling order (fastest cycling index is the row one), the 9 component vector forms a matrix with the vectors of the current base in the components of the parent base, by columns. In other words the first three coefficients represent the components of the x axis of the current frame in the parent frame, and so on for y and z axis.

- `multiple`: *required:* yes, in same number as `reference_tag`. *multiple:* yes. *type:* logical. Is the reference frame multiple?

- `moving`: *required:* yes, in same number as `reference_tag`. *multiple:* yes. *type:* logical. Is the reference frame moving with respect to the parent?

5.5 Moving reference frames

Special input is required when a reference frame is moving or is multiple, in which case a special grouping keyword is required. As shown in file 5.3 if the parameter `moving` is set to true, it must be followed by the grouping parameter `motion` in which the definition of the motion is provided.

All the motions in DUST are provided as a translation of a pole and a rotation around an axis passing from the pole, as depicted in figure 5.3. The motion of the pole can be defined in terms of position or velocity. Both can be described with some simple functions or from a time history from a data file. In the same way the rotation around an axis can be defined in terms of angle or rotation rate, and with simple functions or data input.

The simple functions are either a constant value, or a sinusoidal function of time defined as

$$f(t) = A \sin(\omega t + \phi) + r \tag{5.3}$$

where A is the amplitude, ω the pulsation, ϕ the phase and r an offset.

input file 5.3: references_moving.in

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
reference_tag = moving
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
    input_type = simple_function
    function = (/ 0 , 0 , 0 /)
    !file = xxx.dat
    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 = ...
  }
  rotation = {
    input = position
    input_type = simple_function
```

```
    !file = ...
    function = 1
    Axis = (/ 0.0 , 1.0 , 0.0 /)
    amplitude = 0.1
    omega = 1.2566
    phase = 0.0
    offset = 0.0
    psi_0 = 0.0
  }
}
```

The details of the parameters inside the `motion` group are:

- `pole`: *required:* yes. *multiple:* not inside each `motion` group.
  Grouping keyword containing the specification of the pole motion, the pole motion parameters are
  - `input`: *required:* yes. *multiple:* not inside each `pole` group. *type:* string.
    In which way motion is imposed, can be `position` or `velocity`
  - `input_type`: *required:* yes. *multiple:* not inside each `pole` group. *type:* string.
    type of input, can be `simple_function` or `from_file`
  - `function`: *required:* yes if `input_type` is `simple_function`. *multiple:* not inside each `pole` group. *type:* integer array of length 3.
    type of function to be imposed, at the moment `0` is for a constant function and `1` is for a sinusoidal function. An array of three components is expected, one for each direction of the pole motion.
  - `file`: *required:* yes if `input_type` is `from_file`. *multiple:* not inside each `pole` group. *type:* string.
    name of the file where the time history is of the motion (position or velocity) is stored. The file is supposed to have 4 columns, the first containing a series of time values adequate for the simulation time, and the following three the three components of the pole motion.
  - `amplitude`: *required:* no, used only if `input_type` is `simple_function`. *multiple:* not inside each `pole` group. *default:* 1.0 *type:* real.
    Collective amplitude of the motion, applied to all the three components of the motion
  - `vector`: *required:* no, used only if `input_type` is `simple_function`. *multiple:* not inside each `pole` group. *default:* (/1.0, 1.0, 1.0/) *type:* real array of length 3.
    Relative amplitude of the motion for each component
  - `omega`: *required:* no, used only if `input_type` is `simple_function`. *multiple:* not inside each `pole` group. *default:* (/1.0, 1.0, 1.0/) *type:* real array of length 3
    Pulsation of each sinusoidal motion. Considered only if input is `simple_function` and only for the components with sinusoidal functions.
  - `phase`: *required:* no, used only if `input_type` is `simple_function`. *multiple:* not inside each `pole` group. *default:* (/0.0, 0.0, 0.0/) *type:* real array of length 3.
    phase of each sinusoidal motion. Considered only if input is `simple_function` and only for the components with sinusoidal functions.
  - `offset`: *required:* no, used only if `input_type` is `simple_function`. *multiple:* not inside each `pole` group. *default:* (/0.0, 0.0, 0.0/) *type:* real array of length 3.
    Constant offset (position/velocity) for each component of the pole motion
  - `Position_0`: *required:* no, used only if `input_type` is `simple_function`. *multiple:* not inside each `pole` group. *default:* (/0.0, 0.0, 0.0/) *type:* real array of length 3
    Starting position of the pole in the three component, considered only if `input` is `velocity`
- `rotation`: *required:* yes. *multiple:* not inside each `motion` group.
  Grouping keyword containing the specification of the rotation, which parameters are:

– input: *required:* yes. *multiple:* not inside each rotation group. *type:* string.
  In which way the rotation is imposed, can be position for imposing an angle or velocity for imposing a rotation rate
– input_type: *required:* yes. *multiple:* not inside each rotation group. *type:* string.
  type of input, can be simple_function or from_file
– function: *required:* yes if input_type is simple_function. *multiple:* not inside each rotation group. *type:* integer.
  type of function to be imposed, at the moment 0 is for a constant function and 1 is for a sinusoidal function.
– file: *required:* yes if input_type is from_file. *multiple:* not inside each rotation group. *type:* string.
  name of the file where the time history is of the rotation (angle or rotation rate) is stored. The file is supposed to have 2 columns, the first containing a series of time values adequate for the simulation time, and the following the rotation around the axis.
– Axis: *required:* yes. *multiple:* not inside each rotation group. *type:* real array of length 3.
  orientation of the rotation axis, in the parent reference frame. The rotation axis keeps a constant orientation in the parent reference frame and always passes through the pole during its motion
– amplitude: *required:* no, used only if input_type is simple_function. *multiple:* not inside each rotation group. *default:* 1.0 *type:* real.
  Collective amplitude of the rotation
– omega: *required:* no, used only if input_type is simple_function. *multiple:* not inside each rotation group. *default:* 1.0 *type:* real.
  Pulsation of the sinusoidal motion. Considered only if input is simple_function and input_type is a sinusoidal function.
– phase: *required:* no, used only if input_type is simple_function. *multiple:* not inside each rotation group. *default:* 0.0 *type:* real.
  phase of the sinusoidal motion. Considered only if input is simple_function and input_type is a sinusoidal function.
– offset: *required:* no, used only if input_type is simple_function. *multiple:* not inside each rotation group. *default:* 0.0 *type:* real.
  Constant offset of rotation or rotation rate
– Psi_0: *required:* no, used only if input_type is simple_function. *multiple:* not inside each rotation group. *default:* 0.0 *type:* real.
  Starting angle of the rotation, considered only if input is velocity

5.6 Multiple reference frames

To ease the setup of rotors in terms of reference frames and motions, a special set of instructions have been developed for the reference frames. Declaring a reference frame multiple, a single component is replicated n times in a special set of multiple, automatically generated reference frames. While there is possibility to expansion towards different types of multiple reference frames, at the moment multiplicity is employed only for rotors: an example input file with a reference frame for a rotor is given in file 5.4.

input file 5.4: references_rotor.in

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
reference_tag = Rotor01
parent_tag = Root
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/)
```

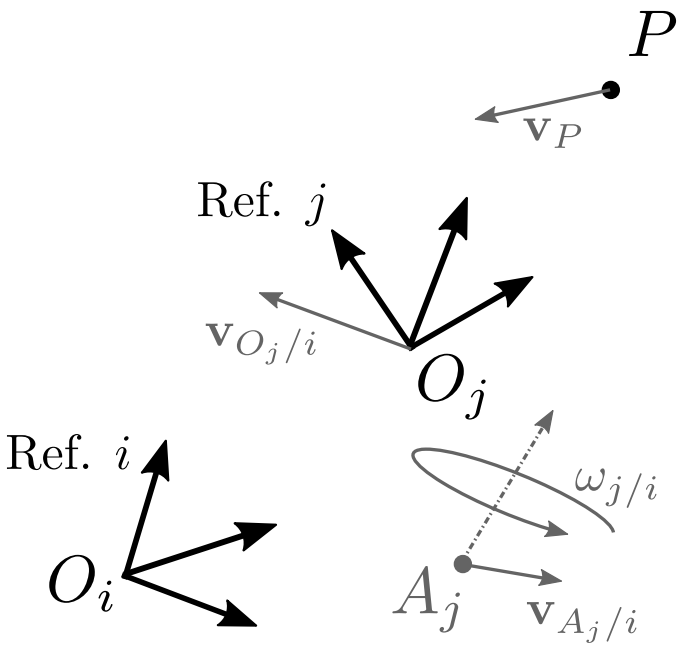

Figure 5.3: Velocity of relative reference frame

```
moving = F
multiple = T
multiplicity = {
  mult_type = rotor
  n_blades = 4
  rot_axis = (/0.0, 0.0, 1.0/)
  rot_rate = 6.28318530717959 !2*pi, T=1
  psi_0 = 0.0
  hub_offset = 0.0

  n_dofs = 3
  dof = {
    hinge_type = Flap
    hinge_offset = (/ 0.0 , 0.032432 , 0.0 /)
    Collective = 3.0 ! deg
    cyclic_ampl = 0.0 ! deg
    cyclic_phas = 0.0 ! deg
  }
  dof = {
    hinge_type = Lag
    hinge_offset = (/ 0.0 , 0.021622 , 0.0 /)
    Collective = -10.0
    cyclic_ampl = 0.0
    cyclic_phas = 0.0
  }
  dof = {
    hinge_type = Pitch
    hinge_offset = (/ 0.0 , 0.086486 , 0.0 /)
    Collective = 12.0
    cyclic_ampl = 0.0
    cyclic_phas = 0.0
```

```
  }

}
```

Note that the rotor reference frame in file 5.4 is set as not moving. This means that the starting reference frame is stationary (with respect to its parent) however all the other reference frames that are automatically generated move in order to represent the different movements of a rotor.

If in the reference frame specification the parameter `multiple` is set to true, it must be followed by a grouping keyword, `multiplicity`, which must contain the details of the multiplicity. The detailed parameters required in the multiplicity are:

- `mult_type`: *required:* yes. *multiple:* not inside each `multiplicity` group. *type:* string.

  type of multiplicity. At the moment only `rotor` is enabled.

- `N_Blades`: *required:* yes if `mult_type` is `rotor`. *multiple:* not inside each `multiplicity` group. *type:* integer.

  Number of blades of the rotor. It is also the number of time the geometrical component associated with the reference frame will be multiplied in the domain.

- `Rot_Axis`: *required:* yes if `mult_type` is `rotor`. *multiple:* not inside each `multiplicity` group. *type:* real array of length 3.

  Axis of rotation of the rotor, with respect to the current reference frame

- `Rot_Rate`: *required:* yes if `mult_type` is `rotor`. *multiple:* not inside each `multiplicity` group. *type:* real.

  rotation rate of the blades around the rotation axis. The rotation of the rotor is kept constant.

- `Psi_0`: *required:* yes if `mult_type` is `rotor`. *multiple:* not inside each `multiplicity` group. *type:* real.

  Starting angle of the rotor at the beginning of the simulation

- `Hub_Offset`: *required:* yes if `mult_type` is `rotor`. *multiple:* not inside each `multiplicity` group. *type:* real.

  offset from the rotation pole (the origin of the multiple reference frame) of the beginning of the chain of reference frames for each blade. It is constant and does not imply any secondary motion, can be used to represent the central part of the rotor axis assembly.

- `N_dofs`: *required:* yes if `mult_type` is `rotor`. *multiple:* not inside each `multiplicity` group. *type:* integer.

  Number of additional degrees of freedom of each blade, generally representing a movement around one of the blades hinges

- `dof`: *required:* yes if `N_dofs` is greater than zero. *multiple:* yes, in the number defined by `N_dofs`.

  Grouping keyword containing the details about geometry and movement of one additional degree of freedom. The degrees of freedom are connected in a chain according to the order in which are defined. A representation of the chain of degrees of freedom is presented in figure 5.4.

  - `hinge_Type`: *required:* yes. *multiple:* not inside each `dof` group. *type:* string.
    type of hinge considered, affecting the axis of rotation of the hinge movement with respect to the rotor axis, can be `Flap`,`Lag`, or `Pitch`.
  - `hinge_Offset`: *required:* yes. *multiple:* not inside each `dof` group. *type:* real array of length 3.
    offset of the hinge rotation axis with respect to the previous hinge or the rotor hub (axis + hub offset).
  - `Collective`: *required:* yes. *multiple:* not inside each `dof` group. *type:* real.
    Collective (constant during rotation) angle of rotation (in degrees) of the degree of freedom.

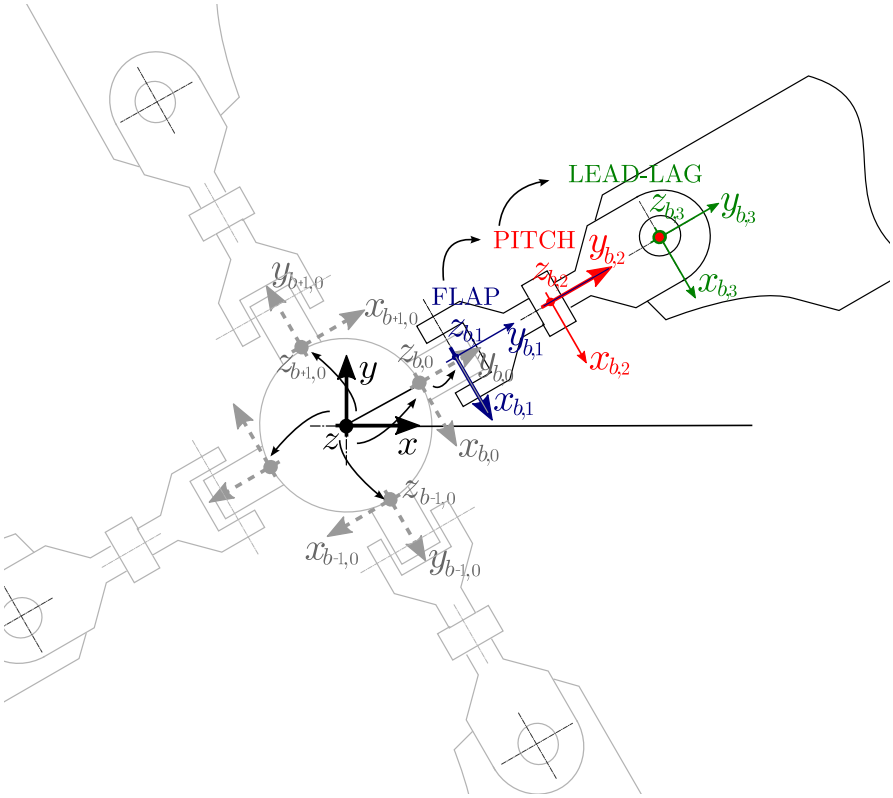

Figure 5.4: multiple reference frames in a rotor

– `Cyclic_Ampl`: *required:* yes. *multiple:* not inside each `dof` group. *type:* real.
Cyclic (sinusoidal during rotation) angle of rotation (in degrees) of the degree of freedom.
– `Cyclic_Phas`: *required:* yes. *multiple:* not inside each `dof` group. *type:* real.
phase of the cyclic movement, in degrees, with respect to the rotor rotation.

5.7 Dimensional Units

In the present section the dimensional units, normalization (or the lack of it) in DUST will be discussed.

All the equations, models, inputs, outputs etc. in the code are **dimensional**. This does not mean that they are required to be in a specific dimensional unit, but only that **no sort of non-dimensionalization is performed in the code**. The user is fully resposible for the scaling (or not) of his inputs, and as a consequence of the outputs.

More in detail the core of DUST solves the potential equations, which are a simplified version of the incompressible Euler equations. Such equations do not depend on a scaling parameter (such as Mach or Reynolds for compressible Navier-Stokes) so their (scaled) results are independent on the scaling employed. Three possible approaches at the scaling of the variables and results are given in examples 1,2 and 3.

Example 1 (Potential case, fully dimensional)
Consider a isolated wing with span $b = 20m$ *and chord* $c = 2m$ *flying at* $U = 60m/s$ *in air with density* $\rho = 1.225kg/m^3$.

The user can take a completely dimensional approach and insert (or build) the geometry in the correct size in meters, and impose a free stream velocity and density corresponding to the real flight conditions. In this case the results, for example in terms of loads, will be in the same units set employed in the inputs, $L[N]$.

To obtain the loads coefficient then the user must non-dimensionalize the results employing the appropriate reference quantities used in the computations:

$$C_L = \frac{L}{1/2\rho U^2 bc} = \frac{L}{1/2 \cdot 1.225 \cdot 60^2 \cdot 20 \cdot 2}$$

Example 2 (Potential case, fully scaled)

Starting from the conditions of example 1, a user might want to re-scale all the quantities with respect to scaling units. For example, the geometry might be scaled by the span, giving $b' = b/b = 1$, $c' = c/b = 0.1$ and introduced scaled from the mesh generator, scaled in the pre-processor or built parametrically already scaled. The velocity might be scaled on the free stream one, specifying directly $U' = U/U = 1$, as well as the density $\rho' = \rho/\rho = 1$.

With these parameters as input, the output would be scaled accordingly, leding for example to the scaled loads L'. These outputs, being scaled, are not numerically equal to the ones obtained in example 1, however they can be on the one hand scaled back to the real ones employing the inverse scaling used for the inputs, or on the other hand if only the non-dimensional coefficients are sought they can be obtained using the scaled reference quantities:

$$C_L = \frac{L'}{1/2\rho' U'^2 b' c'} = \frac{L'}{1/2 \cdot 1 \cdot 1^2 \cdot 1 \cdot 0.1}$$

When non-dimensionalized on the appropriate reference quantities the non-dimensional coefficients have always the same values.

Example 3 (Potential case, partially scaled)

While in example 2 all the quantities have been scaled with respect to some scaling quantity, the user can choose to use different quantities, or to scale only some of the quantities. For example, the user might want to keep the geometry from the CAD in meters, $b = 20m$, $c = 2m$ and scale the flight conditions for ease of use, $U' = U/U = 1$, $\rho' = \rho/\rho = 1$.

The results L'' are going to be again scaled, in a different way than in example 2, and the non-dimensional coefficient can be retrieved using the reference quantities employed in the computation:

$$C_L = \frac{L''}{1/2\rho' U'^2 bc} = \frac{L''}{1/2 \cdot 1 \cdot 1^2 \cdot 20 \cdot 2}$$

It must however be stressed that when talking about scaling of the inputs and non-dimensionalization of the results, these operations are left to the user to be carried out independently before and after the execution of DUST. To maintain complete usage flexibility, it is not possible to introduce any explicit scaling or non-dimensionalizing unit. When using certain features of the code (lifting lines, vorticity diffusion, separations etc.) the effects of viscosity and compressibility are simulated in different manners. To account for the right flow conditions it is necessary to specify the parameters governing the viscous and compressibility phenomena, i.e. the dynamic viscosity μ and the speed of sound c in the input file. In continuity with the rest of the code, the dimensional physical properties of the flow must be provided, rather than the non dimensional numbers Reynolds and Mach. It is responsibility of the user to insert them correctly and eventually scale them to obtain similarity of Reynolds and Mach with respect to the target conditions. Examples on how to achieve that in different conditions are presented in example 4.

Example 4 (Ways to determine physical properties)

Depending on the cases, the physical properties of the fluid to be given as input might be:

- *Already known and inserted as they are, if all the other reference quantities have not be scaled*
- *Obtained from the original ones, keeping Reynolds and Mach number equal, if some of the reference quantities were scaled:*

$$\frac{\rho U L}{\mu} = \frac{\rho' U' L'}{\mu'} \quad \rightarrow \quad \mu' = \mu \frac{\rho' U' L'}{\rho U L}$$

$$\frac{U}{c} = \frac{U'}{c'} \quad \rightarrow \quad c' = c \frac{U'}{U}$$

Debug Level	Effects
<0	Almost no screen output, useful just for batch runs
1	Minimal screen output
3	Standard screen output information
5	Verbose screen output (fmm data etc.)
7	Extra warnings and diagnostics, can lead to false negative warnings
8	Extra verbose output (lifting lines data etc.)
15	Output minimal geometry in ascii files
15	Output extended geometry in ascii files
20	Output the solution in ascii files
50	Output the full linear system in ascii files

Table 5.1: `debug_level` levels and their effects

- *Obtained from the non-dimensional parameters, if those are known rather than the properties of the fluid:*

$$Re = \frac{\rho' U' L'}{\mu'} \quad \rightarrow \quad \mu' = \frac{\rho' U' L'}{Re}$$

$$Ma = \frac{U'}{c'} \quad \rightarrow \quad c' = \frac{U'}{M}$$

Finally, the reference pressure given as input is used as reference, free stream pressure, and does not affect the results except for a constant offset in the pressure field output. Loads are computed integrating on a closed surface and so are not affected by a constant offset on pressure. Knowing the reference pressure used as input, it is possible to retrieve the local coefficient of pressure:

$$C_p(\boldsymbol{x}) = \frac{P(\boldsymbol{x}) - P_{ref}}{1/2 \rho U^2}$$

where $P(\boldsymbol{x})$ is the local pressure from the results, $P(\boldsymbol{x})$ is the reference pressure given as input and ρ, U are the reference quantities employed during the simulation.

5.8 Debug Levels

As discussed in section 5.1 in the solver input file it is possible to set the parameter `debug_level` which selects the verbosity of the output of the code. The level is selected choosing an integer number, and it is cumulative: selecting a certain value the user gets the additional output given at that value *and* all the outputs at the lower levels. As a rule of thumb the debug levels up to 10 generate increasing levels of verbosity in the screen outputs, while increasing levels over 10 generate additional outputs saved in files, in the appropriate `basename_debug` path. The current effects of the debug level are presented in table 5.1. Be aware that the debug file output is mainly targeted for development work, and its content is susceptible to sudden and undocumented changes.

5.9 Output Files

When running the solver, a certain number of hdf5 binary files are generated, in the location specified by `basename`. Two main type of files are generated:

- `basename`_geo.h5: a file containing the geometry of the simulation, which is similar to the input given to the solver, with some additions related to the single simulation.
- `basename`_res_XXXX.h5: a number of result files obtained during the simulation, printed at the frequency specified in the input file.

Chapter 6

DUST Postprocessor

The DUST postprocessor is used to generate meaningful data from the binary results generated during the execution of the solver. While as discussed in 1.4 it is possible to look at the content of the hdf5 results, these being based on the singularities intensities on the surface provide little insight on the solution of the solver.

The postprocessor takes the specified results and use them to obtain a variety of different processed data, from visualizations to loads.

The preprocessor is executed simply invoking the executable `dust_post` in the desired folder. The input file containing all the required informations for the execution of the postprocessor must be passed as argument to the command call. If not provided explicitly the preprocessor automatically tries to read the default input file `dust_post.in`.

```
dust_post input_file_name.in
```

Command 6.5: Postprocessor command looking for input file `input_file_name.in`

```
dust_post
```

Command 6.6: Postprocessor command looking for default input file `dust_post.in`

The different possible analysis that can be performed on the results are:

- **Integral loads**: history of loads acting on the geometry or parts of the geometry
- **Visualizations**: visualization of the surface solution on the geometry and of the wake
- **Probes**: Time history of certain variables probed in a set of specified points
- **Flow fields**: Visualization of the flow field in a structured block domain
- **Sectional loads**: distribution of the loads along a direction on long aspect ratio components (i.e. wings or blades)

6.1 Input file

The input file of the postprocessor contains first all the information required for retrieving the correct results and to process the data (model parameters etc.) and then in separate sections all the analyses that are requested. An arbitrary number of analyses can be requested in a single input file, however it is also possible to group the analyses in different input files then invoked multiple times as different inputs for the postprocessor.

The format is the same as all the other input files, as already specified in section1.2.

A generic input file describing the main input is:

input file 6.1: dust_post.in

```
!--- Data Names ---
data_basename = ./Output/sim_results
basename = ./Postpro/postpro_output

!--- Model Parameters (as in solver) ---
far_field_ratio_doublet = 10.0
far_field_ratio_source = 10.0
doublet_threshold = 1.0e-6
rankine_rad = 0.1
vortex_rad = 0.1
cutoff_rad = 0.001

analysis = {

type = viz
name = vis01
start_res = 1
end_res = 100
step_res = 1
format = vtk
wake = T
variable = vorticity
}

analysis = {

type = integral_loads
name = load01
start_res = 1
end_res = 100
step_res = 1
format = dat
average = F
component = all
reference_tag = RotorHub
}
```

- `data_basename`: *required:* yes. *multiple:* no.

  Base name (with path) of the data which must be analysed.

- `basename`: *required:* yes. *multiple:* no.

  Base name (with path) of the postprocessing results

- `far_field_ratio_doublet`: *required:* no. *multiple:* no. *default:* 10

  Ratio with respect to element length to set the thresholds for far field approximation. Same as in solver input file 5.1.

- `far_field_ratio_source`: *required:* no. *multiple:* no. *default:* 10

  As for `far_field_ratio_doublet` determines the threshold after which far field approximations are employed, just for sources. Same as in solver input file 5.1.

- `doublet_threshold`: *required:* no. *multiple:* no. *default:* 1.0e-6

  Parameter which sets the distance threshold under which the evaluation point, with respect to a panel, is considered inside the plane of the panel. Same as in solver input file 5.1.

- `rankine_rad`: *required:* no. *multiple:* no. *default:* 0.1

  Parameter which sets the radius under which the Rankine approximation of vortexes cores is employed. Used for aerodynamic elements and panels (i.e. everything except vortex particles). Same as in solver input file 5.1.

- `vortex_rad`: *required:* no. *multiple:* no. *default:* 0.1

  Parameter which sets the radius of the vortex particles. Same as in solver input file 5.1.

- `cutoff_rad`: *required:* no. *multiple:* no. *default:* 0.001

  Parameter which sets the radius under which the vortexes interaction is completely set to zero. Same as in solver input file 5.1.

- `analysis`: *required:* at least one. *multiple:* yes

  Grouping keyword containing the information for a single postprocessing analysis.

The grouping keyword `analysis` specifies a single analysis, and the parameters contained in the group depend on the type of the analysis. A series of `analysis` groups can be contained in a single input file.

6.1.1 Visualizations

Visualizations are the main form of assessment of the results, they allow to see the movement of geometry and wake and the intensity of the solution on the surfaces and on the wake.

An example `analysis` group for a visualization is:

input file 6.2: dust_post.in for visualization

```
!--- Data Names ---
data_basename = ./Output/sim_results
basename = ./Postpro/postpro_output

analysis = {

type = viz
name = vis01
start_res = 1
end_res = 100
step_res = 1
format = vtk
wake = T
average = F
variable = vorticity
component = all

}
```

- `type`: *required:* one for each `analysis`. *multiple:* no

  type of the analysis, `viz` for surface visualizations

- `name`: *required:* one for each `analysis`. *multiple:* no

  name of the analysis, will be appended as a suffix to `basename`

- `start_res`: *required:* one for each `analysis`. *multiple:* no

  First result in the time series of the solver results to analyse.

- `end_res`: *required:* one for each `analysis`. *multiple:* no

  Last result in the time series of the solver results to analyse.

- `step_res`: *required:* one for each `analysis`. *multiple:* no

  Stride to employ when loading the time series of the solver results.

- `format`: *required:* one for each `analysis`. *multiple:* no

  format of the processed results, can be `vtk` for vtk output or `tecplot` for tecplot `plt` files

- `wake`: *required:* no. *multiple:* no. *default:* True

  Output the wake in the postprocessing.

- `separate_wake`: *required:* no. *multiple:* no. *default:* False

  Since in vtk output is difficult to separate in the visualization process the different pieces of the solution, it is possible to output the wake in separate files with respect to the surface solution. It affects only the vtk output and only if `wake` is set to true.

- `average`: *required:* no. *multiple:* no. *default:* False

  average the results in the given time span, and output just one averaged result. When used in visualizations, `wake` must be False.

- `variable`: *required:* at least one for each `analysis`. *multiple:* yes.

  variable to output in the processed result. More than one variable can be generated in the same analysis. At the moment are implemented the following variables are implemented:

  - `vorticity`
  - `vorticity_vector`
  - `velocity`
  - `surface_velocity`
  - `pressure`
  - `cp`
  - `turbulent_viscosity`
  - `vortex_rad`

  Note that that even if the keyword `vorticity` is employed, the printed result is labelled `singularity_intensity` since it represent the intensity of the surface/point solution on the different elements. The keyword is likely to be changed also in the future releases.

- `component`: *required:* no. *multiple:* yes. *default:* all.

  Geometrical component to include in the visualization. More component can be included. If not declared, or if declared `all` all the components are loaded and processed.

6.1.2 Integral loads

Integral loads allow to obtain the time history of the loads acting on one or more components, in one of the reference frames defined in section 5.4.

An example `analysis` group for integral loads is:

input file 6.3: dust_post.in for integral loads

```
!--- Data Names ---
data_basename = ./Output/sim_results
basename = ./Postpro/postpro_output
```

```
analysis = {

type = integral_loads
name = load01
start_res = 1
end_res = 100
step_res = 1
format = dat
average = F
component = all
reference_tag = RotorHub

}
```

- `type`: *required:* one for each `analysis`. *multiple:* no
  type of the analysis, `integral_loads` for integral loads
- `name`: *required:* one for each `analysis`. *multiple:* no
  name of the analysis, will be appended as a suffix to `basename`
- `start_res`: *required:* one for each `analysis`. *multiple:* no
  First result in the time series of the solver results to analyse.
- `end_res`: *required:* one for each `analysis`. *multiple:* no
  Last result in the time series of the solver results to analyse.
- `step_res`: *required:* one for each `analysis`. *multiple:* no
  Stride to employ when loading the time series of the solver results.
- `format`: *required:* one for each `analysis`. *multiple:* no
  format of the processed results, can be `dat` for formatted ascii files output or `tecplot` for tecplot plt files
- `average`: *required:* no. *multiple:* no. *default:* False
  average the results in the given time span.
- `component`: *required:* no. *multiple:* yes. *default:* all.
  Geometrical component to include in the loads computation. More component can be included. If not declared, or if declared `all` all the components are loaded and processed.
- `Reference_Tag`: *required:* yes. *multiple:* no.
  The tag of the reference frame in which the loads should be referred to.

Output .dat file

The output file containing integral loads measurements in `.dat` format has the following structure. The first 4 lines of the file constitute the header of the file. The first three lines contains the number of the geometrical components `n_comp` whose loads are integrated, the tag of reference system in which the components of the integral loads are expressed, the tags of the `n_comp` components analyzed. As an example,

```
1: # Integral loads: N.components: 2
2: # Ref.sys : Wing1_Reference
3: # Components : Wing1 , Wing2
```

The fourth line is the header of the following lines, containing the time, the three components of force and moment loads, the 9 elements of the (unrolled) rotation matrix and the 3 global coordinates of the origin of the local reference system, required to compute the components of the loads in the global reference frame, given those in the local reference frame.

```
  4: # t , Fx , Fy , Fz , Mx , My , Mz , ref_mat(9) , ref_off(3)
```

The last lines of the file contain the actual data. As an example, for a 100-timestep analysis with the local reference frame `Wing1_Reference` aligned with the global reference frame (the rotation matrix is equal to the identity) and the origin in $(0.0, -1.0, 0.0)$, these lines read

```
  5: 0.000000E+000 0.260753E+001 0.000000E+000 0.115993E+003
     0.150792E+003 -0.434173E+001 -0.338980E+001 0.100000E+001
     0.000000E+000 0.000000E+000 0.000000E+000 0.100000E+001
     0.000000E+000 0.000000E+000 0.000000E+000 0.100000E+001
     0.000000E+000 -0.100000E+001 0.000000E+000
...:
104: 0.858240E-002 0.545038E+001 0.000000E+000 0.830829E+002
     0.108007E+003 -0.312156E+001 -0.708549E+001 0.100000E+001
     0.000000E+000 0.000000E+000 0.000000E+000 0.100000E+001
     0.000000E+000 0.000000E+000 0.000000E+000 0.100000E+001
     0.000000E+000 -0.100000E+001 0.000000E+000
```

6.1.3 Hinge Loads

Hinge loads allows to obtain the time history of the load acting to one control surface along the hinge axis reference frame as defined in figure 3.6.

An example `analysis` group for hinge loads is:

```
analysis = {
  type = hinge_loads
  name = hm

  start_res = 1
  end_res = 20
  step_res = 1

  format = dat
  average = F
  component = Wing
  hinge_tag = Aileron
}
```

- `type`: *required:* one for each `analysis`. *multiple:* no
  type of the analysis, `hinge_loads` for hinge loads
- `name`: *required:* one for each `analysis`. *multiple:* no
  name of the analysis, will be appended as a suffix to `basename`
- `start_res`: *required:* one for each `analysis`. *multiple:* no
  First result in the time series of the solver results to analyse.
- `end_res`: *required:* one for each `analysis`. *multiple:* no
  Last result in the time series of the solver results to analyse.

- `step_res`: *required:* one for each `analysis`. *multiple:* no

  Stride to employ when loading the time series of the solver results.

- `format`: *required:* one for each `analysis`. *multiple:* no

  format of the processed results, can be `dat` for formatted ascii files output or `tecplot` for tecplot `plt` files

- `average`: *required:* no. *multiple:* no. *default:* False

  average the results in the given time span.

- `component`: *required:* no. *multiple:* yes. *default:* all.

  Geometrical component to include in the loads computation. More component can be included. If not declared, or if declared `all` all the components are loaded and processed.

- `hinge_tag`: *required:* yes. *multiple:* no.

  The tag of the control surface of the component in which the loads should be referred to.

Output .dat file

The output file containing hinge loads measurements in `.dat` format has the following structure. The first 4 lines of the file constitute the header of the file. The first three lines contains the number of the geometrical components `n_comp` whose loads are integrated, the tag of reference system in which the components of the integral loads are expressed, the tags of the `n_comp` components analyzed. As an example,

```
1: # hinge Moment:
2: # Components : Wing
3: # hinge : Aileron
```

The fourth line is the header of the following lines, containing the time, the three components of force and moment loads, the 9 elements of the (unrolled) rotation matrix and the 3 global coordinates of the origin of the hinge reference system, required to compute the components of the loads in the global reference frame, given those in the hinge axis reference frame. The hinge moment is located always under the column `Mh`.

```
4: # t , Fv , Fh , Fn , Mv , Mh , Mn , axis_mat(9) , node_hinge(3)
```

The last lines of the file contain the actual data. As an example, for a 20-timestep analysis with the hinge reference frame `Wing1_Aileron` aligned with the global reference frame (the rotation matrix is equal to the identity) and the origin of the hinge is $(0.25, -0.1, 0.0)$, these lines read

```
 5: 0.000000E+000 0.212095E+004 -0.34102E-015 0.11784E+005
    0.295794E+006 -0.28342E+004 -0.53235E+005 0.10000E+001
    0.000000E+000 0.000000E+000 0.000000E+000 0.10000E+001
    0.000000E+000 0.000000E+000 0.000000E+000 0.10000E+001
    0.250000E+000 -0.10000E+000 0.000000E+000

...:
 25: 0.190000E+000 0.121453E+003 -0.75517E-017 0.643457E+003
     0.161506E+005 -0.81645E+002 -0.30484E+004 0.100000E+001
     0.000000E+000 0.000000E+000 0.00000E+000 0.100000E+001
     0.000000E+000 0.000000E+000 0.00000E+000 0.100000E+001
     0.25000E+000 -0.10000E+000 0.00000E+000
```

6.1.4 Probes

Probes allow to obtain a time history of some variables in some points by sampling the solution at such points.

An example `analysis` group for probes is:

input file 6.4: dust_post.in for probes

```
!--- Data Names ---
data_basename = ./Output/sim_results
basename = ./Postpro/postpro_output

analysis = {

type = probes
name = prb01
start_res = 1
end_res = 100
step_res = 1
format = dat

variable = Velocity

input_type = point_list
point = (/0.0, 1.0, 2.0/)
point = (/1.0, 0.0, -2.0/)

!input_type = from_file
!file = point_list.dat

}
```

- `type`: *required:* one for each `analysis`. *multiple:* no
  type of the analysis, `probes` for probes
- `name`: *required:* one for each `analysis`. *multiple:* no
  name of the analysis, will be appended as a suffix to `basename`
- `start_res`: *required:* one for each `analysis`. *multiple:* no
  First result in the time series of the solver results to analyse.
- `end_res`: *required:* one for each `analysis`. *multiple:* no
  Last result in the time series of the solver results to analyse.
- `step_res`: *required:* one for each `analysis`. *multiple:* no
  Stride to employ when loading the time series of the solver results.
- `format`: *required:* one for each `analysis`. *multiple:* no
  format of the processed results, can be `dat` for formatted ascii files output or `tecplot` for tecplot `plt` files
- `variable`: *required:* at least one for each `analysis`. *multiple:* yes.
  variable to output in the processed result. More than one variable can be generated in the same analysis. At the moment `Velocity`, `Pressure` and `cp` are implemented.
- `input_type`: *required:* one for each `analysis`. *multiple:* no
  The way the probing points are specified, `point_list` for a series of points specified directly in the input file, `from_file` to load the list of points from a formatted ascii file.

- `point`: *required:* at least one if `input_type` is `point_list` *multiple:* yes.

  Position of the probing point.

- `file`: *required:* yes if `input_type` is `from_file` *multiple:* no.

  Location and name of the file containing the list of probing points.

Output .dat file

The output file containing point probe measurements in `.dat` format has the following structure. The first line is a header containing the number of probes

```
1: # N. of point probes: 2
```

and the following 3 lines contains the 3 global coordinates of the point probes. As an example, if the `n_probes` = 2 probes have global cartesian coordinates $(0.0, 1.0, 2.0)$ and $(1.0, 0.0, -2.0)$, these lines read

```
2: 0.000000E+000 0.100000E+001
3: 0.100000E+001 0.000000E+000
4: 0.200000E+001 -0.200000E+001
```

The fifth line is a header line, containing the description of the next lines. The first field is the time,

```
5: # t 2( ux uy uz )
```

while `n_probes` $\times$ 3 columns follow, containing the global components of the velocity vectors, measured by the point probes. The following lines contains the fields described for each time step required for the analysis. As an example, for a 100-timestep analysis these lines read

```
  6: 0.000000E+000 0.465000E+002 -0.899269E-004 -0.469068E-004 0.465000E+002
     0.633261E-004 -0.980265E-004
...:
105: 0.858240E-002 0.465003E+002 -0.238562E-002 -0.119496E-002 0.465023E+002
     0.216758E-002 -0.357483E-002
```

6.1.5 Flow Field

Flow fields allow to probe systematically the domain obtaining the solution probed in a structured series of points in 1-2 or 3 dimensions, allowing the visualization of the flow field induced by the solution of the singular elements.

An example `analysis` group for flow fields is:

input file 6.5: dust_post.in for flow fields

```
!--- Data Names ---
data_basename = ./Output/sim_results
basename = ./Postpro/postpro_output

analysis = {

type = flow_field
name = ff01
start_res = 1
end_res = 100
step_res = 1
format = vtk
```

```
average = F

variable = Velocity

n_xyz = (/ 10, 10, 1/)
min_xyz = (/ -2.0, -2.0, 1.0 /)
max_xyz = (/ 2.0, 2.0, 1.0 /)
}
```

- `type`: *required:* one for each `analysis`. *multiple:* no
  type of the analysis, `flow_field` for flow fields
- `name`: *required:* one for each `analysis`. *multiple:* no
  name of the analysis, will be appended as a suffix to `basename`
- `start_res`: *required:* one for each `analysis`. *multiple:* no
  First result in the time series of the solver results to analyse.
- `end_res`: *required:* one for each `analysis`. *multiple:* no
  Last result in the time series of the solver results to analyse.
- `step_res`: *required:* one for each `analysis`. *multiple:* no
  Stride to employ when loading the time series of the solver results.
- `format`: *required:* one for each `analysis`. *multiple:* no
  format of the processed results, can be `vtk` for binary vtk files output or `tecplot` for tecplot plt files
- `average`: *required:* no. *multiple:* no. *default:* False
  average the results in the given time span, only one averaged result will be printed.
- `variable`: *required:* at least one for each `analysis`. *multiple:* yes.
  variable to output in the processed result. More than one variable can be generated in the same analysis. At the moment `Velocity`, `Pressure` and `cp` are implemented.
- `n_xyz`: *required:* one for each `analysis`. *multiple:* no.
  Number of points in each direction of the sampling box. To have a 2D plane or a 1D line insert 1 point as number of points in the relevant direction
- `min_xyz`: *required:* one for each `analysis`. *multiple:* no.
  Minimum of the coordinates of the box containing the structured sampling points.
- `max_xyz`: *required:* one for each `analysis`. *multiple:* no.
  Maximum of the coordinates of the box containing the structured sampling points.

6.1.6 Sectional loads

Sectional loads allow to obtain the distribution of the forces along one direction of a slender body, e.g. a wing or a blade.

The way the sectional loads are retrieved is different when employing a parametrically generated element or a generic element (i.e. an unstructured mesh generated from a mesh generator).

In the first case is sufficient to provide the reference line for the moments calculation, and the subdivision is already implied in the structure of the parametric element.

An example `analysis` group for sectional loads on parametric components:

input file 6.6: dust_post.in for sectional load on parametric components

```
!--- Data Names ---
data_basename = ./Output/sim_results
basename = ./Postpro/postpro_output

analysis = {

type = sectional_loads
name = sl01
start_res = 1
end_res = 100
step_res = 1
format = dat
average = F

component = wing

axis_nod = (/ 0.0, 0.0, 0.0 /)
axis_dir = (/ 0.0, 0.1, 0.0 /)

lifting_line_data = F
vortex_lattice_data = F
}
```

- `type`: *required:* one for each `analysis`. *multiple:* no
  type of the analysis, `sectional_loads` for sectional loads
- `name`: *required:* one for each `analysis`. *multiple:* no
  name of the analysis, will be appended as a suffix to `basename`
- `start_res`: *required:* one for each `analysis`. *multiple:* no
  First result in the time series of the solver results to analyse.
- `end_res`: *required:* one for each `analysis`. *multiple:* no
  Last result in the time series of the solver results to analyse.
- `step_res`: *required:* one for each `analysis`. *multiple:* no
  Stride to employ when loading the time series of the solver results.
- `format`: *required:* one for each `analysis`. *multiple:* no
  format of the processed results, can be `dat` for formatted ascii files output or `tecplot` for tecplot plt files
- `average`: *required:* no. *multiple:* no. *default:* False
  average the results in the given time span.
- `component`: *required:* one for each `analysis`. *multiple:* no.
  component (parametrically generated) to analyse. Only one component can be enabled for each analysis.
- `axis_nod`: *required:* one for each `analysis`. *multiple:* no.
  Coordinates of the node from which to start the line around which to calculate the distribution of moment along the span.
- `axis_dir`: *required:* one for each `analysis`. *multiple:* no.
  Direction of the line (starting from `axis_nod`) around which to calculate the distribution of moment along the span.

- `lifting_line_data`: *required:* no. *multiple:* no. *Default:* False
- `vortex_lattice_data`: *required:* no. *multiple:* no. *Default:* False

  Output additional informations regarding span distribution of several quantities of interest for the lifting lines. It works only for parametric elements which are lifting lines.

On generic components it is necessary to define an ordered slender box with orientation to prescribe the interpolation along a series of sections of the surface data of the components to obtain the distribution of loads along the prescribed direction.

An example `analysis` group for sectional loads on generic components:

input file 6.7: dust_post.in for sectional load on generic components

```
!--- Data Names ---
data_basename = ./Output/sim_results
basename = ./Postpro/postpro_output

analysis = {

type = sectional_loads
name = sl01
start_res = 1
end_res = 100
step_res = 1
format = dat
average = F

component = wing

box_sect = {
 ref_node = (/ -0.5 ,-0.5 , -0.3 /)
 face_vec = (/ 1.0 , 0.0 , 0.0 /)
 face_bas = (/ 2.0 , 1.0 /)
 face_hei = (/ 1.0 , 1.0 /)
 span_vec = (/ 0.0 , 1.0 , 0.0 /)
 span_len = 3.0
 num_sect = 10
 reshape_box = T
}


}
```

- `type`: *required:* one for each `analysis`. *multiple:* no

  type of the analysis, `sectional_loads` for sectional loads
- `name`: *required:* one for each `analysis`. *multiple:* no

  name of the analysis, will be appended as a suffix to `basename`
- `start_res`: *required:* one for each `analysis`. *multiple:* no

  First result in the time series of the solver results to analyse.
- `end_res`: *required:* one for each `analysis`. *multiple:* no

  Last result in the time series of the solver results to analyse.

- `step_res`: *required:* one for each `analysis`. *multiple:* no

  Stride to employ when loading the time series of the solver results.

- `format`: *required:* one for each `analysis`. *multiple:* no

  format of the processed results, can be `dat` for formatted ascii files output or `tecplot` for tecplot plt files

- `average`: *required:* no. *multiple:* no. *default:* False

  average the results in the given time span.

- `component`: *required:* one for each `analysis`. *multiple:* no.

  component to analyse. Only one component can be enabled for each analysis.

- `box_sect`: *required:* one for each `analysis` if component not parametrically generated. *multiple:* no.

  Grouping keyword containing the information about the box generated to interpolate the sectional loads

- `ref_node`: *required:* one for each `box_sect`. *multiple:* no.

  Coordinates of the point from which to build the bounding box for the sectional loads.

- `face_vec`: *required:* one for each `box_sect`. *multiple:* no.

  Direction of the base faces of the bounding box (roughly the chord direction). The bounding box is extruded between the two faces.

- `face_bas`: *required:* one for each `box_sect`. *multiple:* no.

  Lenght of the base faces in the `face_vec` direction. It must have two components, one for the first beginning face, one for the last end face.

- `face_hei`: *required:* one for each `box_sect`. *multiple:* no.

  Height of the base faces, normal to the `face_vec` direction. It must have two components, one for the first beginning face, one for the last end face.

- `span_vec`: *required:* one for each `box_sect`. *multiple:* no.

  Direction of the span for the sectional loads, indicates the direction between the beginning and end faces of the bounding box.

- `span_len`: *required:* one for each `box_sect`. *multiple:* no.

  Length of the span for the sectional loads, indicates the distance between the beginning and end faces of the bounding box.

- `num_sect`: *required:* one for each `box_sect`. *multiple:* no.

  Number of sections, i.e. subdivisions in the span direction of the box. The subdivision must be coarse enough to avoid having one single surface panel appearing in three different sections. This means that for an equally spaced mesh and a tightly fitted box that there should be a number of sections lower or equal to the number of surface elements in the span direction.

- `reshape_box`: *required:* one for each `box_sect`. *multiple:* no.

  Enables the possibility to reshape a too loose sections box to closely fit the component. Automatically reduces the size of the span to fit the component, keeping the same number of subdivision (thus generating a finer subdivision than requested)

Output .dat file

The output file containing sectional loads in `.dat` format has the following structure. The first line is a header specifying the structural component analyzed and the force/moment component collected in the file. As an example,

```
1: # Sectional load Fz of component: Wing1
```

The second line contains the number of sections `n_sec` in which the structural component is divided and the number of time steps `n_time` of the analysis. As an example, the output file for a 10-section blade reads

```
2: # n_sec : 10 ; n_time : 100. Next lines: y_cen , y_span, chord
```

The second line also introduce the content of lines 3, 4 and 5. Line 3 contains the spanwise coordinate `y_cen` of the centre of the sections, line 4 contains the spanwise dimension `y_span` of the section, and line 5 contains the chord dimension `chord` of the section. As an example, if the the blade length is 0.6 and it is divided in 10 uniform sections with chord 0.05, the output file reads

```
3: 0.30000E-001 0.90000E-001 0.15000E+000 0.21000E+000 ...
4: 0.60000E-001 0.60000E-001 0.60000E-001 0.60000E-001 ...
5: 0.50000E-001 0.50000E-001 0.50000E-001 0.50000E-001 ...
```

The sixth line is another header file containing the fields and their dimension collected in the next `n_time` lines.

```
6: # t , sec(n_sec) , ref_mat(9) , ref_off(3)
```

The first field is the time. The sectional loads of the `n_sec` sections follow. The last 12 lines contains the (first-column) unrolled 3x3 rotation matrix the global components of the origin of the local reference frame, required to obtain the components of the loads in the global reference frame from those expressed in the local reference frame,

$$\underline{f}^G = \underline{\underline{R}}\,\underline{f}^L \ . \tag{6.1}$$

As an example, the lines collecting the fields described above of the first and the last of the 100 timesteps of the analysis reads

```
  7: 0.000000E+000 0.966615E+002 0.966615E+002 0.966615E+002 0.966615E+002
     0.966615E+002 0.966615E+002 0.966615E+002 0.966615E+002 0.966615E+002
     0.966615E+002 0.100000E+001 0.000000E+000 0.000000E+000 0.000000E+000
     0.100000E+001 0.000000E+000 0.000000E+000 0.000000E+000 0.100000E+001
     0.000000E+000 -0.100000E+001 0.000000E+000
...:
105: 0.858240E-002 0.571236E+002 0.701379E+002 0.755018E+002 0.780314E+002
     0.790800E+002 0.790800E+002 0.780314E+002 0.755017E+002 0.701378E+002
     0.571235E+002 0.100000E+001 0.000000E+000 0.000000E+000 0.000000E+000
     0.100000E+001 0.000000E+000 0.000000E+000 0.000000E+000 0.100000E+001
     0.000000E+000 -0.100000E+001 0.000000E+000
```

The geometrical component of the example is attached to a steady reference frame, whose origin has global components $(0.0, -1.0, 0.0)$ and whose axes are aligned with those of the global reference frame, so that the rotation matrix $\underline{\underline{R}}$ is equal to the identity matrix.

The extra files produced if `lifting_line_data` or if `vortex_lattice_data` are set to `T` do not contain the 12 columns of the rotation matrix and the global components of the origin of the local reference frame.

The files generated by sectional loads contain, as sectional quantity:

- `Fx`: force acting on the section, `x` direction of the component local reference frame

- `Fy`: force acting on the section, `y` direction of the component local reference frame
- `Fz`: force acting on the section, `z` direction of the component local reference frame
- `Mo`: moment acting on the section, with respect to the axis specified in the definition of the sectional load with `axis_nod` and `axis_dir`

additionally, when `lifting_line_data` or `vortex_lattice_data` are set to `T` the following quantities are printed:

- `Cl`: two dimensional lift coefficient of the section, as retrieved from the lookup tables
- `Cd`: two dimensional drag coefficient of the section, as retrieved from the lookup tables
- `Cm`: two dimensional moment coefficient of the section, as retrieved from the lookup tables
- `vel_2d`: magnitude of the projection of the relative velocity (as the sum of the free stream velocity, the opposite of the body velocity and the influence of all the singularities, $\mathbf{v}_{rel} = \mathbf{v}_\infty - \mathbf{v}_b + \mathbf{v}_{ind} = \mathbf{v}_{rel,free} + \mathbf{v}_{ind}$) in the plane identified by the normal and tangential unit vectors $\hat{\mathbf{n}}$, $\hat{\mathbf{t}}$, as represented in figure 6.1, $v_{2d} = |\mathbf{v}_{rel} \cdot \hat{\mathbf{b}}|$.
- `vel_outplane`: magnitude of the out-of-plane velocity, $\mathbf{v}_{rel} \cdot \hat{\mathbf{b}}$, where $\hat{\mathbf{b}} = \hat{\mathbf{n}} \times \hat{\mathbf{t}}$ is the unit vector "in the crossflow direction"
- `alpha`: angle of attack of the in plane relative velocity at the control point, corrected with the 2D influence needed by Piszkin and Lewinski formulation of LL elements, in order to get the right angle of attack to be used in the aerodynamic tables
- `vel_2d_isolated`: same as `vel_2d` but for the relative free stream velocity $\mathbf{v}_{rel,free}$ only, being the influence of the body and wake singularities neglected;
- `vel_outplane_isolated`: same as `vel_outplane` but for the relative free stream velocity $\mathbf{v}_{rel,free}$ only, being the influence of the body and wake singularities neglected;
- `alpha_isolated`: angle of attack of the in plane relative free stream velocity at the control point, equal to $\alpha_{is} = \text{atan2}(\mathbf{v}_{rel,free} \cdot \hat{\mathbf{n}},\ \mathbf{v}_{rel,free} \cdot \hat{\mathbf{t}})$.

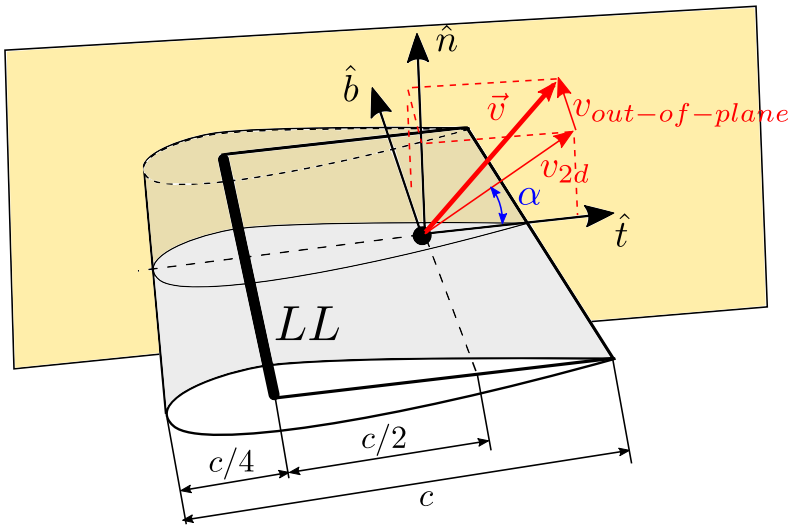

Figure 6.1: Lifting line data.

6.1.7 Chordwise Loads

Chordwise loads allow to obtain the distribution of the forces, pressure and the geometrical quantities along the chord at a specific spanwise section of a parametric component.

Warning: at the moment, the chordwise load card is implemented only for parametric components.

An example `analysis` group for chordwise loads on parametric components:

input file 6.8: dust_post.in for chordwise load on parametric components

```
!--- Data Names ---
data_basename = ./Output/sim_results
basename = ./Postpro/postpro_output

analysis = {

  type = chordwise_loads
  name = cl01
  start_res = 1
  end_res = 100
  step_res = 1
  format = dat
  average = F

  component = wing
  n_station = 2
  span_station = (/1., 1.5/)
  axis_nod = (/ 0.0, 0.0, 0.0 /)
  axis_dir = (/ 0.0, 0.1, 0.0 /)

}
```

- `type`: *required:* one for each `analysis`. *multiple:* no
  type of the analysis, `chordwise_loads` for chordwise loads
- `name`: *required:* one for each `analysis`. *multiple:* no
  name of the analysis, will be appended as a suffix to `basename`
- `start_res`: *required:* one for each `analysis`. *multiple:* no
  First result in the time series of the solver results to analyse.
- `end_res`: *required:* one for each `analysis`. *multiple:* no
  Last result in the time series of the solver results to analyse.
- `step_res`: *required:* one for each `analysis`. *multiple:* no
  Stride to employ when loading the time series of the solver results.
- `format`: *required:* one for each `analysis`. *multiple:* no
  format of the processed results, can be `dat` for formatted ascii files output or `tecplot` for tecplot plt files
- `average`: *required:* no. *multiple:* no. *default:* False
  average the results in the given time span.
- `component`: *required:* one for each `analysis`. *multiple:* no.
  component (parametrically generated) to analyse. Only one component can be enabled for each analysis.

- `axis_nod`: *required:* one for each `analysis`. *multiple:* no.

  Coordinates of the node from which to start the line around which to calculate the distribution of moment along the span.

- `axis_dir`: *required:* one for each `analysis`. *multiple:* no.

  Direction of the line (starting from `axis_nod`) around which to calculate the distribution of moment along the span.

- `n_station`: *required:* no. *multiple:* no. *Default:* 1

  Number of stations where the loads are extracted

- `span_station`: *required:* no. *multiple:* no. *Default:* (/0./)

  Spanwise coordinates in the component reference frame (in wind axis) where they are extracted The vector dimension must be the same as `n_station`

Output .dat file

The output file containing chordwise loads in `.dat` format has the following structure. The first line is a header specifying the structural component. As an example,

```
1: # Chordwise load of component: wing
```

The second line contains the spanwise location and the chord length evaluated at the first requested time step `start_res`.

```
2: # spanwise_location: 1.00; chord_length: 0.40
```

The third line contains the number of chordwise section `n_chord` in which the structural component is divided and the number of time steps `n_time` of the analysis. As an example, the output file for a 10-section blade reads

```
3: # n_sec : 10 ; n_time : 100. Next lines: x_chord , z_chord
```

The third line also introduce the content of lines 4 and 5. Line 4 contains the chordwise coordinate `x_chord` of the centre of the panels, line 4 contains the thickness dimension `z_chord` of the profile, taken at `start_res`. As an example, if the the chord length is 0.4 and it is divided in 10 uniform sections with chord 0.05, the output file reads

```
3: 0.130E+000 0.900E-001 0.500E-001 ...
4: -0.257E-002 -0.711E-002 -0.103E-001 ...
```

The fifth line is another header file containing the fields and their dimension collected in the next `n_time` lines.

```
6: # t, n\_chord
```

The first field is the time. The chordwise loads of the `n_chord` sections follow.

The files generated by sectional loads contain, as sectional quantity:

- `Pres`: Panel pressure for 3D panels, or delta pressure for vortex lattice or lifting line.
- `Cp`: Panel pressure coefficient for 3D panels, or delta pressure coefficent for vortex lattice or lifting line. The implemented formulation is

$$C_p = \frac{P - P_\infty}{\frac{1}{2}\rho_\infty U_\infty^2} \tag{6.2}$$

  Where P_∞ is equal to `P_inf`, ρ_∞ is equal to `rho_inf` and and U_∞ is equal to the norm of `u_inf` or `u_ref` if `u_inf` is zero.

- `dFx`: Panel force per unit length in chordwise direction
- `dFz`: Panel force per unit length in flapwise direction
- `dNx`: Panel local normal in chordwise direction
- `dNz`: Panel local normal in flapwise direction
- `dTx`: Panel local tangent in chordwise direction
- `dTz`: Panel local tangent in flapwise direction
- `x_cen`: Panel center chordwise coordinate
- `z_cen`: Panel center flapwise coordinate

6.1.8 Aeroacoustics

The aeroacoustics postprocessing is a specific analysis used to extract from results all the various data required to perform an aeroacoustics analysis (with an external software) on the analysed results. It is available only in .dat ascii format and it is intended rather than to be directly plotted/visualized to act as input for another software.

input file 6.9: dust_post.in for aeroacoustics

```
!--- Data Names ---
data_basename = ./Output/sim_results
basename = ./Postpro/postpro_output

analysis = {

type = aeroacoustics
name = aer01
start_res = 1
end_res = 100
step_res = 1
format = dat

component = wing
component = tail

}
```

- `type`: *required:* one for each `analysis`. *multiple:* no
  type of the analysis, `aeroacoustics` for aeroacoustics data
- `name`: *required:* one for each `analysis`. *multiple:* no
  name of the analysis, will be appended as a suffix to `basename`
- `start_res`: *required:* one for each `analysis`. *multiple:* no
  First result in the time series of the solver results to analyse.
- `end_res`: *required:* one for each `analysis`. *multiple:* no
  Last result in the time series of the solver results to analyse.
- `step_res`: *required:* one for each `analysis`. *multiple:* no
  Stride to employ when loading the time series of the solver results.

- `format`: *required:* one for each `analysis`. *multiple:* no

  format of the processed results, can only be `dat` for formatted ascii files output in case of aeroacoustics data analysis

- `component`: *required:* no. *multiple:* yes. *default:* all components.

  Components to include in the aeroacoustics data. Only the data of the selected components will be included in the output file.

Output .dat file

For each DUST component is generated a corresponding `.dat` file. The output file containing aeroacustics data in `.dat` format has the following structure. The first line is a header specifying the analysis.

```
1: # Aeroacoustic Data
```

The second and third lines contain the current simulation time.

```
2: # Time
3: 0.000000000000000E+000
```

The fourth line is a header file containing the fields and their dimension collected in the nextlines.

```
4: # Element Center (3), Element Normal (3), Element Center Velocity (3),
    Element Area (1), Force Acting on Element (3)
```

In the case of a rotor/propeller component, each blade is labelled with Comp_ _ _ . As an example, with a 3 blade propeller/rotor, reads:

```
5: Comp 000
  ...
..: Comp 001
  ...
```