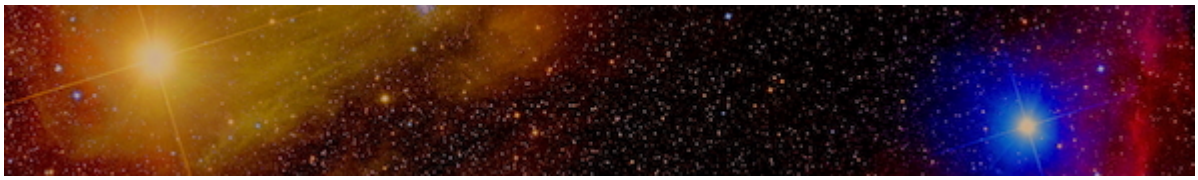


Antares Documentation

Release 2.2.0

Antares Development Team

Apr 15, 2024

CONTENTS

Antares is a Python Data Processing library mainly intended for Computational Fluid Dynamics. Antares provides a fully documented cross-platform high-level python application programming interface. Antares helps you develop data processing applications. It can be used on operating systems OS X, Windows, or Linux. It defines a set of routines and tools for building application software. The software components communicate through a specified data structure.

To get a didactic introduction, please visit the *introduction* (page 3) section and the *tutorial* (page 349) pages. Visit the *installation page* (page 5) to get started with it. You can browse the *API reference* (page 11) to find out what you can do with antares, and then check out the *tutorials* (page 349).

To report an issue, please go to ReportIssues. To consult user-dedicated information, please visit the redmine server[1].

[1] https://ant.cerfacs.fr

CHAPTER

ONE

INTRODUCTION

We often collect data through numerical simulation, observation, or experimentation, but the data is often not readily usable as is. This raw data has to be analyzed to enhance understanding of the underlying physical phenomena. Antares helps making this data processing by delivering a set of features.

Antares can be inserted in your python computation process at the preprocessing or postprocessing stages, but can also be used in a co-processing workflow.

You can get another insight of Antares capabilities by visiting the *tutorial pages* (page 349).

1.1 Features

1.2 Going further

If you are convinced that antares can help you process your data, then you might visit the *installation page* (page 5) to see how you can get the package and get started with it. You may also want to look at the *tutorial* (page 349) and *API reference* (page 11) pages.

CHAPTER

TWO

INSTALLATION

2.1 Dependencies

Recommended versions are:

- python[2] 3 (>= 3.7) is supported. If you experience any trouble, please use the bug tracker[3] or go to Reporting Issues[4]
- NumPy[5] >= 1.9.0
- (optional) VTK[6] >= 6.3.0 (supported from 5.6.1 but with less treatments)
- (optional) SciPy[7] >= 0.12.1
- (optional) h5py[8] >= 2.7.0
- (optional) tqdm[9] (progress bar)

For parallel processing:

- mpi4py >= 2.0.0
- h5py >= 2.7.1 parallel
- METIS >= 5.1.0
- HDF5 >= 1.8.15 parallel

[2] https://www.python.org/
[3] https://ant.cerfacs.fr/projects/antares/issues/new
[4] https://cerfacs.fr/antares/issues.html
[5] https://www.numpy.org/
[6] https://www.vtk.org/
[7] https://www.scipy.org/
[8] https://www.h5py.org/
[9] https://github.com/tqdm/tqdm

2.2 Installation

You have two different ways to use Antares depending on what you want to do with it.

It is very easy to setup with both ways, assuming that python is properly installed.

2.2.1 Straightforward Use

If you want to use directly Antares, **the easiest way** is to source the file antares.env located at the Antares root directory:

```
source PATH_TO_ANTARES/antares.env      # in bash
source PATH_TO_ANTARES/antares_csh.env  # in csh
```

You're done !

Alternatively, you can manually set the environment:

- Set an environment variable to the Antares root directory

```
export ANTARES='PATH_TO_ANTARES/'  # in bash
setenv ANTARES 'PATH_TO_ANTARES/'  # in csh
```

- Add Antares to your `PYTHONPATH`

```
export PYTHONPATH=$PYTHONPATH:$ANTARES    # in bash
setenv PYTHONPATH ${PYTHONPATH}:$ANTARES  # in csh
```

- Add the bin/ folder to your `PATH` (if you want to use the bin)

```
export PATH=$PATH:$ANTARES/bin  # in bash
set path=($ANTARES/bin $path)   # in csh
```

You're all set !

However, some specific modules will be disabled (see below).

2.2.2 Installing with extensions

Some Antares treatments are written in `C` langage to get CPU performance. So the installation needs to go through a compilation phase to use them.

If you want to use Antares with these specific compiled modules, then you have to install Antares the following way. You may need an internet connection, and you need to install the setuptools[10] package first.

Then, get the Antares source package, unpack it, and go to the Antares root directory.

By default, Antares is installed as other python packages into Python's main site-packages directory if you do:

```
python setup.py install
```

Very often though, you want to install python packages in an alternate location (see Alternate Installation[11]).

[10] https://pypi.python.org/pypi/setuptools
[11] https://docs.python.org/2/install/#alternate-installation

Alternate Installation Schemes

Alternate Installation: the User Scheme[12]

You can then either do,

```
python setup.py install --user
```

This will install Antares in your own local directory. To know the name of this directory, type:

```
python -c'import site; print(site.USER_BASE)'
```

or simply:

```
env PYTHONUSERBASE=/somewhere_in_the_system/local python setup.py install --user
```

Next, you just need to add the package directory to the `PYTHONPATH` and the binaries to the `PATH`

```
export PYTHONPATH=$PYTHONPATH:/somewhere_in_the_system/local/lib/python3.7/site-packages
export PATH=$PATH:/somewhere_in_the_system/local/bin
```

You may also install python packages in a custom location (see Custom Installation[13]).

Custom Installation

For a custom installation, add the installation directory to the `PYTHONPATH`

```
export PYTHONPATH=$PYTHONPATH:/somewhere_in_the_system
```

then install Antares with:

```
python setup.py install --install-lib=/somewhere_in_the_system --install-scripts=/
↪somewhere_in_the_system/bin
```

If you have no or limited network access, then look at the section *Dependencies* (page 5).

2.3 Extensions

Extensions are libraries written in C that can be called in python. You have to compile them if you want to use them in Antares.

If you want to change compiler options, then you can pass distutils-known variables as follows:

```
env CC="gcc" CFLAGS="-O3" OPT="" python setup.py build_ext
```

[12] https://docs.python.org/2/install/#inst-alt-install
[13] https://docs.python.org/2/install/#custom-installation

2.3.1 *Boundary Layer Module*

To install the BL treatment, you can type for example,

```
python setup.py install --user --bl
```

OpenMP

The BL module can be compiled with OpenMP. The compiler option for openmp should be provided. Here an example on linux gcc:

```
python setup.py install --user --bl --bl-openmp-flags -fopenmp
```

Optimization

The BL module can be compiled with higher optimization options than the setup default ones. By default, the setup uses the same option used for the python compilation itself.

Use *–bl-optim-flags* to define your own optimization level (depends on the compiler used). If defined, the tool will set also the following: -UDEBUG -DNDEBUG Here is an example with linux gcc:

```
python setup.py install --user --bl --bl-optim-flags -O3
```

2.3.2 *Transformation of face-based connectivity to element-based connectivity*

To install this extension, you can type for example,

```
python setup.py install --user --ngon
```

2.3.3 *Process High-Order Solutions from the Jaguar solver*

To install this extension, you can type for example,

```
python setup.py install --user --jag
```

2.3.4 *Polyhedral Mesh Support*

To install this extension, you can type for example,

```
python setup.py install --user --poly
```

Orientation of 2d mesh normals

To install this functionality used in TreatmentCellNormal, you can type for example,

```
python setup.py install --user --orient
```

2.3.5 *FBC (Face-based Connectivity)*

Warning: This treatment is not available in Antares from version 1.10.0.

The FBC treatment depends on the metis[14] library.

You need to set the environnement variables `METIS_DIR`, the root directory of the metis[15] library.

Then, you can type for example,

```
python setup.py install --user --fbc
```

Warning: This treatment has not been tested on windows platforms.

Parallel Support

The metis[16] library is required when reading an HDF-CGNS file with unstructured zones.

You need to set the environnement variable `METIS_LIB_DIR` where the shared library lies.

2.3.6 *C++ Mesh Cutter*

To install the C++ Mesh Cutter module used in the "ccut" Treatment, you can type for example,

```
python setup.py install --user --mshcppcutter
```

Compiler Options

The C++ Mesh Cutter module can be compiled with OpenMP. The usage is the same as with the BL module, as you need to pass in the right flag for the compiler. An example with gcc on linux :

```
python setup.py install --user --mshcppcutter --mshcppcutter-openmp-flags -fopenmp
```

---

[14] https://glaros.dtc.umn.edu/gkhome/views/metis
[15] https://glaros.dtc.umn.edu/gkhome/views/metis
[16] https://glaros.dtc.umn.edu/gkhome/views/metis

2.4 Documentation Generation

The python package **sphinx** is required. The documentation of antares is generated with Sphinx[17] from docstrings inserted in the code using the reStructuredText[18] (reST) syntax.

The documentation is available at https://www.cerfacs.fr/antares.

However, if you want to install it locally, then you may need to generate it from the source files.

The documentation is in `PATH_TO_ANTARES/doc`. Go to this directory.

Type

```
make html
```

Then open doc/_build/html/index.html

Type

```
make latexpdf
```

Then open doc/_build/latex/antares_doc.pdf

2.5 Data to run examples

Data are available at Example Data[19].

[17] https://www.sphinx-doc.org/en/stable/rest.html
[18] https://docutils.sourceforge.net/rst.html
[19] https://cerfacs.fr/antares/downloads/data.tgz

CHAPTER

THREE

APPLICATION PROGRAMMING INTERFACE

For a didactic introduction to the API objects, please visit the API tutorial[20] pages.

The API of antares relies on a hierarchical data structure. Let's see the ABC.

3.1 Antares Basic Classes

The root node of this four-level hierarchical structure is an instance of the *Base* class. The child nodes of the root node are instances of the *Zone* class. An instance of the *Base* class (named a *Base* in the following) is essentially a container of *Zone* instances. Each *Zone* is basically a collection of *Instant* instances. Each *Instant* is a container of variables.

Click on any of the object to access its detailed documentation.

3.2 Antares Topological Classes

3.2.1 Boundary

Class for Boundary Condition.

[20] https://cerfacs.fr/antares/src/tutorial/base.html

class antares.api.Boundary.**Boundary**(*bnd=None*, *inherit_computer=None*)

Boundary class inherits from *Window* (page 22) class.

Note: The attribute *name* (page 19) is the name of the Boundary object. The dictionary `Zone.boundaries` maps a (key) name to a Boundary object. Both names could be different for a given Boundary object even if it would be weird to do so.

Methods

add_computer_function (page 13)(new_func)	Set a new function.
clear (page 13)()	
compute (page 13)(var_name[, location, reset, store])	Compute a given variable on the whole zone.
compute_bounding_box (page 13)(coordinates)	Compute the bounding box of the zone with respect to the coordinates.
compute_coordinate_system (page 13)([ttype, ...])	Compute a new coordinate system in the Datasets.
copy (page 14)()	
delete_variables (page 14)(list_vars[, location])	Delete variables in the dataset.
deserialized (page 14)(pickable_boundary)	Build a pickable representation of the boundary.
dimension (page 14)()	Dimension of the Dataset.
duplicate_variables (page 14)(list_vars, list_newvars)	Duplicate variables in the dataset.
fromkeys (page 15)(iterable[, value])	
get (page 15)(k[,d])	
get_ghost_cells (page 15)()	Return the ghost_cells table.
get_location (page 15)(location, new_dataset)	Return a copy of the Datasets containing only the variables located in the original Datasets at the specified location.
get_shape (page 15)()	Get the shape of the dataset.
is_structured (page 15)()	Tell whether the zone is a structured mesh or not.
items (page 15)()	
keys (page 16)()	Return keys as a list and not KeysView nor dict_keys.
orient_flat_range_slicing (page 16)(base)	Create the slicing to reorient data from boundary to orientation 1.
orientation_slicing (page 16)(base)	Create the slicing to reorient the boundary of 3D structured mesh to orientation 1.
pop (page 16)(k[,d])	If key is not found, d is returned if given, otherwise KeyError is raised.
popitem (page 16)()	as a 2-tuple; but raise KeyError if D is empty.
rel_to_abs (page 16)([coordinates, conservative_vars, ...])	Transform conservative variables from relative frame to absolute frame by looping on all instants using the `Instant.rel_to_abs()` method.

continues on next page

Table 1 – continued from previous page

rename_variables (page 17)(list_vars, list_newvars[, ...])	Rename variables in the dataset.
serialized (page 17)([data])	Build a pickable representation of the boundary.
set_computer_model (page 17)(modeling[, ...])	Set a computer modeling for the zone.
set_formula (page 17)(formula)	Set a formula for the dataset (Zone or Boundary).
set_formula_from_attrs (page 17)(name)	Set a formula from a name in the dataset attribute.
setdefault (page 17)(k[,d])	
slicing_orientation (page 17)()	Give the orientation of a 3D slicing.
update (page 17)([E,]**F)	If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v
values (page 18)()	

Attributes

add_computer_function(*new_func*)

Set a new function.

The computer will receive a new function associated to its current modeling.

clear() → None. Remove all items from D.

compute(*var_name*, *location=None*, *reset=False*, *store=True*)

Compute a given variable on the whole zone.

This variable is computed on all instants of the zone.

Use the `Instant.compute()` method.

Parameters

- **var_name** (*str*) – The name of the variable to compute.
- **location** (str in *LOCATIONS* (page 369)) – The location of the variable. If `None`, the default location is assumed.
- **reset** (*bool*) – Remove temporary fields stored in the equation manager.
- **store** (*bool*) – Store temporary fields in the equation manager.

compute_bounding_box(*coordinates*)

Compute the bounding box of the zone with respect to the coordinates.

Parameters
coordinates (*list(str)*) – list of variable names

Returns
the bounding box

Return type
dictionary with key: variable names, value: list with min and max values

compute_coordinate_system(*ttype='cartesian2cylindrical', remove_current=False, current_coord_sys=['x', 'y', 'z'], new_coord_sys=['x', 'r', 'theta'], origin=[0.0, 0.0, 0.0]*)

Compute a new coordinate system in the Datasets.

Parameters

- **ttype** (*str in ['cartesian2cylindrical', 'cylindrical2cartesian']*) – type of transformation
- **remove_current** (*bool*) – remove current coordinate system after transformation
- **current_coord_sys** (*list of 3 str*) – names of the current coordinates
- **new_coord_sys** (*list of 3 str*) – names of the new coordinates
- **origin** (*list of 3 float*) – position of the origin

> **Warning:** 'cylindrical2cartesian' not implemented
>
> for 'ttype'='cartesian2cylindrical', in 'new_coord_sys', the first coordinate is the axial direction, the second the radial one, and the third the azimuthal one (by default (x, r, θ))
>
> The first coordinate name in 'new_coord_sys' must also be into 'current_coord_sys'.

copy()

delete_variables(*list_vars, location=None*)

Delete variables in the dataset.

Parameters

- **list_vars** (*list(str)*) – list of variables to delete
- **location** (str in *LOCATIONS* (page 369) or 'None') – if None, delete the variables at the all locations

equivalent to del zone[:, :, list_vars] which uses `del` with zone slicing instead.

classmethod deserialized(*pickable_boundary*)

Build a pickable representation of the boundary.

dimension()

Dimension of the Dataset.

Returns
dimension of the Datasets

Return type
int

duplicate_variables(*list_vars, list_newvars, location=None*)

Duplicate variables in the dataset.

Parameters

- **list_vars** (*list(str)*) – list of variables to duplicate
- **list_newvars** (*list(str)*) – list of new variable names
- **location** (str in *LOCATIONS* (page 369)) – if different from None, change only the variables at the location specified

Duplication is performed element-wise.

classmethod fromkeys(*iterable*, *value=None*)

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

get_ghost_cells()

Return the ghost_cells table.

Returns
a list containing a dictionary for each index with two keys: 'min' and 'max'.

Each key corresponds to the boundary min or max of that index. As we are in Windows (or in a Boundary), return a default result as if this was a zone without any boundaries defined

The values are lists containing as many elements as the number of boundaries. For each boundary a list of two elements is given:

- the first is the slicing of the present block node array corresponding to this boundary
- the second is:
  - if the boundary is a join: (donor zone name, node array slicing of the donor boundary, trirac)
  - else: None

get_location(*location*, *new_dataset*)

Return a copy of the Datasets containing only the variables located in the original Datasets at the specified location.

Parameters

- **location** (string in *LOCATIONS* (page 369)) – location to extract
- **new_dataset** (`Datasets`) – The dataset that will contain the variables located at the given location.

Returns
the Datasets with only specified location variables

Return type
`Datasets` or `None`

get_shape()

Get the shape of the dataset.

The shape is the shape of the node values, either taken from the shared instant, or taken from the first instant.

Returns
the shape

Return type
tuple

is_structured()

Tell whether the zone is a structured mesh or not.

Note: all instants are supposed to be of the same kind.

Return type
bool

items() → a set-like object providing a view on D's items

keys()

Return keys as a list and not KeysView nor dict_keys.

orient_flat_range_slicing(*base*)

Create the slicing to reorient data from boundary to orientation 1.

If the orientation of the boundary is -1, the boundary is reoriented. The rule is to invert the first axis of the boundary that has a range greater then 1. If the orientation of the boundary is 1 or undefined, it does nothing.

Main difference with *Boundary.orientation_slicing()* (page 16): the range of the slicing is given by the boundary data range. Then, the start and stop are None. So, only the step is changed.

Parameters

base (Base) – the parent Base of the Boundary.

Returns

the slicing to apply to reorient boundary data.

Return type

tuple(slice)

orientation_slicing(*base*)

Create the slicing to reorient the boundary of 3D structured mesh to orientation 1.

If the orientation of the boundary is -1, the boundary is reoriented. The rule is to invert the first axis of the boundary that has a range greater then 1. If the orientation of the boundary is 1 or undefined, it does nothing.

Each slice of the slicing is assumed to have a None "step".

Parameters

base (Base) – the parent Base of the Boundary.

Returns

the slicing to apply to reorient the boundary.

Return type

tuple(slice)

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.

If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), remove and return some (key, value) pair

as a 2-tuple; but raise KeyError if D is empty.

rel_to_abs(*coordinates=None*, *conservative_vars=None*, *omega='in_attr'*, *angle='in_attr'*)

Transform conservative variables from relative frame to absolute frame by looping on all instants using the `Instant.rel_to_abs()` method.

Parameters

- **coordinates** (*list(str)*) – list of coordinates names
- **conservative_vars** (*list(str)*) – list of conservative variables names in the following order: density, momentum along the x-axis; momentum along the y-axis, momentum along the z-axis and total energy per unit of volume
- **omega** (*float*) – angular speed of the current base. If `in_attr` use the omega stored in the attrs, necessary if different angular speeds in the base (for example one angular speed per superblock)

- **angle** (*float*) – angular deviation of the current base. If `in_attr` use the angle stored in the attrs, necessary if different angular deviations in the base (for example one angular deviation per superblock and per instant)

Note: may be moved elsewhere in future releases

Warning: the angular speed must be perpendicular to the x-axis

rename_variables(*list_vars*, *list_newvars*, *location=None*)

Rename variables in the dataset.

Parameters

- **list_vars** (*list(str)*) – list of variables to rename
- **list_newvars** (*list(str)*) – list of new variable names
- **location** (str in *LOCATIONS* (page 369)) – if different from `None`, change only the variables at the location specified

Replacement is performed element-wise.

serialized(*data=True*)

Build a pickable representation of the boundary.

set_computer_model(*modeling*, *species_database=None*, *addons=None*)

Set a computer modeling for the zone.

See `antares.api.Instant.Instant.set_computer_model()`.

set_formula(*formula*)

Set a formula for the dataset (Zone or Boundary).

See `antares.api.Instant.Instant.set_formula()`

set_formula_from_attrs(*name*)

Set a formula from a name in the dataset attribute.

The computer will receive a new formula associated to its current modeling. This formula is included in the zone attribute with the key *name*.

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

slicing_orientation()

Give the orientation of a 3D slicing.

Considering (i, j, k) as a right-handed basis, a 3D slicing has orientation 1 when the slicing gives a orientation 1 face, i.e. the resulting (i', j') basis can be completed with a k' vector such that (i', j', k') is a right-handed basis AND k' direction is outward the interval of k values.

Returns
1 if right-handed oriented, -1 if left-handed oriented, 0 if undefined

Return type
int

update([*E*], ***F*) → None. Update D from mapping/iterable E and F.

If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

values() → an object providing a view on D's values

property attrs

Dictionary `antares.core.AttrsManagement.AttrsManagement` containing the attributes of the Datasets.

bndphys

Type
, optional

container

Attribute (of type `CustomDict`) containing all kind of data associated to the Window, and not constrained to the shape. These data are ignored during a family slicing. optional.

donor_bnd_name

Name of the donor boundary.

Type
str, optional. Valid for 'type'='grid_connectivity' with 'gc_type'='abutting_1to1'

donor_zone_name

Name of the donor zone.

Type
str, optional. Valid for 'type'='grid_connectivity' with 'gc_type'='abutting_1to1'

elsA

Dictionary of elsA options.

keys: + bnd_dir: directory of boundary files + omega_file: filename for rotation velocity on walls + axis_ang_1, axis_ang_2: + xrot_angle, yrot_angle, zrot_angle: rotation angle for periodicity + xtran, ytran, ztran: translation for periodicity

Type
dict, optional

family_name

Name of the family associated to the boundary condition.

Type
str, optional

family_number

Number of the family associated to the boundary condition.

Type
int, optional

gc_type

Type of the grid connectivity boundary.

Type
str in ['abutting_1to1', 'abutting'], compulsory if 'type'='grid_connectivity', else optional

glob_border_cur_name

The boundary belongs to this set of boundaries.

Type
str, optional. Valid for 'type'='grid_connectivity' with 'gc_type'='abutting'

glob_border_opp_name

The set of boundaries opposite to the set of boundaries which the boundary belongs to.

Type
str, optional. Valid for 'type'='grid_connectivity' with 'gc_type'='abutting'

name

Name of the boundary.

Type
str, compulsory

num_type

For gmsh.

Type
str, compulsory if 'type'='boundary'

pangle

Type
, optional

periodicity

namedtuple PERIODIC_T. None if no periodicity Otherwise, provide the rotation center and angle and the translation (all np.array(float of nbdim size) keys: rotationcenter rotationangle translation

property shared

Attribute (of type `Instant`) containing variables shared for all the Instants contained in the Datasets.

property slicing

Get slicing attribute.

slicing_donor

Contains the interface patch subrange of indices for the adjacent zone.

Type
tuple or list, compulsory, for structured grids: tuple of 3 slice objects (six integers), topological information

transform

List of X integers, X being the dimension of the space, valid for 'type'='grid_connectivity' with 'gc_type'='abutting_1to1'

Type
list(int), optional

type

Type of the boundary.

Type
str in ['boundary', 'grid_connectivity'], compulsory

zone_name

Name of the Zone in which the slicing should be applied.

Type

str, compulsory

3.2.2 Family

A Family object is a collection of objects of type Zone, *Boundary* (page 11), and *Family* (page 20).

class antares.api.Family.**Family**(**args*, ***kwargs*)

Family class.

Methods

add_attr (page 21)(item, value[, deep])	Add an attribute to all the elements of the family.
build_reduced_copy (page 21)()	Build a copy of the Family with only Family subtrees and attrs.
clear()	
copy()	
deserialized (page 21)(pickable_family, parent)	Build a Family from its representation.
fromkeys(iterable[, value])	
get(k[,d])	
get_extractor (page 21)([name, extractor_keys])	Create an extractor for the CFD computation based on the family definition.
items()	
keys()	Return keys as a list and not KeysView nor dict_keys.
pop(k[,d])	If key is not found, d is returned if given, otherwise KeyError is raised.
popitem()	as a 2-tuple; but raise KeyError if D is empty.
serialized (page 21)([data])	Build a pickable representation of the family.
set_superblock (page 21)(zone_name)	Set a family to the list of zones that are all connected by joins to the zone given.
setdefault(k[,d])	
update([E,]**F)	If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v
update_reduced_copy (page 22)(reduced_family, data, ...)	Updates a copy of the Family to add reduced data.
values()	

Attributes

add_attr(*item*, *value*, *deep=False*)

Add an attribute to all the elements of the family.

Parameters

- **item** (*str*) – attribute name
- **value** – attribute to be added

build_reduced_copy()

Build a copy of the Family with only Family subtrees and attrs.

If *self* is:

```
Family -+-> 'f1': Family
        +-> 'f2': Family
        +-> 'z1': Zone
        +-> 'z2': Zone
```

Then this operator gives:

```
Family -+-> 'f1': Family
        +-> 'f2': Family
```

This operator is useful combined with *update_reduced_copy*, to rebuild families after reduction treatments (e.g. cut).

Returns
a copy of the Family with only Family subtrees and attrs

Return type
Family (page 20)

classmethod deserialized(*pickable_family*, *parent*)

Build a Family from its representation.

get_extractor(*name='extractor_antares'*, *extractor_keys=None*)

Create an extractor for the CFD computation based on the family definition.

Parameters

- **name** (*str*) – name to give to the extractor
- **extractor_keys** (*dict*) – keys to be setted to the extractor

Returns
the extractor

Return type
str

serialized(*data=True*)

Build a pickable representation of the family.

set_superblock(*zone_name*)

Set a family to the list of zones that are all connected by joins to the zone given.

Parameters

zone_name (*str*) – name of the starting zone

update_reduced_copy(*reduced_family*, *data*, *reduced_data*)

Updates a copy of the Family to add reduced data.

If *self* is:

```
Family -+-> 'f1': Family
        +-> 'f2': Family
        +-> 'z1': Zone (`data`)
        +-> 'z2': Zone
```

Then this operator updates the reduced_family as shown here:

```
Family -+-> 'f1': Family
        +-> 'f2': Family
        +-> 'z1': Zone (`reduced_data`)
```

This operator is useful when combined with *build_reduced_copy*, to rebuild families after reduction treatments (e.g. cut).

Parameters

- **reduced_family** (*Family* (page 20)) – the reduced Family copy to update. This Family must have the same Family subtrees than *self*.
- **data** – the data that has been reduced as *reduced_data*.
- **reduced_data** – *data* after reduction operation.

property attrs

Get the attributes of the Family.

Return type

Dictionary `antares.core.AttrsManagement.AttrsManagement`

3.2.3 Window

Window Class.

class antares.api.Window.**Window**(*window=None*, *inherit_computer=None*)

Window class.

Methods

add_computer_function (page 24)(new_func)	Set a new function.
clear (page 24)()	
compute (page 24)(var_name[, location, reset, store])	Compute a given variable on the whole zone.
compute_bounding_box (page 24)(coordinates)	Compute the bounding box of the zone with respect to the coordinates.
compute_coordinate_system (page 24)([ttype, ...])	Compute a new coordinate system in the Datasets.
copy (page 25)()	
delete_variables (page 25)(list_vars[, location])	Delete variables in the dataset.
deserialized (page 25)(pickable_window)	Build a pickable representation of the window.
dimension (page 25)()	Dimension of the Dataset.
duplicate_variables (page 25)(list_vars, list_newvars)	Duplicate variables in the dataset.
fromkeys (page 25)(iterable[, value])	
get (page 25)(k[,d])	
get_ghost_cells (page 25)()	Return the ghost_cells table.
get_location (page 26)(location, new_dataset)	Return a copy of the Datasets containing only the variables located in the original Datasets at the specified location.
get_shape (page 26)()	Get the shape of the dataset.
is_structured (page 26)()	Tell whether the zone is a structured mesh or not.
items (page 26)()	
keys (page 26)()	Return keys as a list and not KeysView nor dict_keys.
pop (page 26)(k[,d])	If key is not found, d is returned if given, otherwise KeyError is raised.
popitem (page 26)()	as a 2-tuple; but raise KeyError if D is empty.
rel_to_abs (page 26)([coordinates, conservative_vars, ...])	Transform conservative variables from relative frame to absolute frame by looping on all instants using the `Instant.rel_to_abs()` method.
rename_variables (page 27)(list_vars, list_newvars[, ...])	Rename variables in the dataset.
serialized (page 27)([data])	Build a pickable representation of the window.
set_computer_model (page 27)(modeling[, ...])	Set a computer modeling for the zone.
set_formula (page 27)(formula)	Set a formula for the dataset (Zone or Boundary).
set_formula_from_attrs (page 27)(name)	Set a formula from a name in the dataset attribute.
setdefault (page 27)(k[,d])	
update (page 27)([E,]**F)	If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v
values (page 27)()	

Attributes

add_computer_function(*new_func*)

Set a new function.

The computer will receive a new function associated to its current modeling.

clear() → None. Remove all items from D.

compute(*var_name*, *location=None*, *reset=False*, *store=True*)

Compute a given variable on the whole zone.

This variable is computed on all instants of the zone.

Use the `Instant.compute()` method.

Parameters

- **var_name** (*str*) – The name of the variable to compute.
- **location** (str in *LOCATIONS* (page 369)) – The location of the variable. If `None`, the default location is assumed.
- **reset** (*bool*) – Remove temporary fields stored in the equation manager.
- **store** (*bool*) – Store temporary fields in the equation manager.

compute_bounding_box(*coordinates*)

Compute the bounding box of the zone with respect to the coordinates.

Parameters
coordinates (*list(str)*) – list of variable names

Returns
the bounding box

Return type
dictionary with key: variable names, value: list with min and max values

compute_coordinate_system(*ttype='cartesian2cylindrical'*, *remove_current=False*, *current_coord_sys=['x', 'y', 'z']*, *new_coord_sys=['x', 'r', 'theta']*, *origin=[0.0, 0.0, 0.0]*)

Compute a new coordinate system in the Datasets.

Parameters

- **ttype** (*str in ['cartesian2cylindrical', 'cylindrical2cartesian']*) – type of transformation
- **remove_current** (*bool*) – remove current coordinate system after transformation
- **current_coord_sys** (*list of 3 str*) – names of the current coordinates
- **new_coord_sys** (*list of 3 str*) – names of the new coordinates
- **origin** (*list of 3 float*) – position of the origin

> **Warning:** 'cylindrical2cartesian' not implemented
>
> for 'ttype'='cartesian2cylindrical', in 'new_coord_sys', the first coordinate is the axial direction, the second the radial one, and the third the azimuthal one (by default (x, r, θ))
>
> The first coordinate name in 'new_coord_sys' must also be into 'current_coord_sys'.

copy()

delete_variables(*list_vars*, *location=None*)

Delete variables in the dataset.

Parameters

- **list_vars** (*list(str)*) – list of variables to delete
- **location** (str in *LOCATIONS* (page 369) or 'None') – if None, delete the variables at the all locations

equivalent to del zone[:, :, list_vars] which uses **del** with zone slicing instead.

classmethod deserialized(*pickable_window*)

Build a pickable representation of the window.

dimension()

Dimension of the Dataset.

Returns
dimension of the Datasets

Return type
int

duplicate_variables(*list_vars*, *list_newvars*, *location=None*)

Duplicate variables in the dataset.

Parameters

- **list_vars** (*list(str)*) – list of variables to duplicate
- **list_newvars** (*list(str)*) – list of new variable names
- **location** (str in *LOCATIONS* (page 369)) – if different from None, change only the variables at the location specified

Duplication is performed element-wise.

classmethod fromkeys(*iterable*, *value=None*)

get(k[, d]) → D[k] if k in D, else d. d defaults to None.

get_ghost_cells()

Return the ghost_cells table.

Returns
a list containing a dictionary for each index with two keys: 'min' and 'max'.

Each key corresponds to the boundary min or max of that index. As we are in Windows (or in a Boundary), return a default result as if this was a zone without any boundaries defined

The values are lists containing as many elements as the number of boundaries. For each boundary a list of two elements is given:

- the first is the slicing of the present block node array corresponding to this boundary
- the second is:
  - if the boundary is a join: (donor zone name, node array slicing of the donor boundary, trirac)
  - else: None

get_location(*location*, *new_dataset*)

Return a copy of the Datasets containing only the variables located in the original Datasets at the specified location.

Parameters

- **location** (string in *LOCATIONS* (page 369)) – location to extract
- **new_dataset** (`Datasets`) – The dataset that will contain the variables located at the given location.

Returns
the Datasets with only specified location variables

Return type
`Datasets` or `None`

get_shape()

Get the shape of the dataset.

The shape is the shape of the node values, either taken from the shared instant, or taken from the first instant.

Returns
the shape

Return type
tuple

is_structured()

Tell whether the zone is a structured mesh or not.

Note: all instants are supposed to be of the same kind.

Return type
bool

items() → a set-like object providing a view on D's items

keys()

Return keys as a list and not KeysView nor dict_keys.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.

If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), remove and return some (key, value) pair

as a 2-tuple; but raise KeyError if D is empty.

rel_to_abs(*coordinates=None*, *conservative_vars=None*, *omega='in_attr'*, *angle='in_attr'*)

Transform conservative variables from relative frame to absolute frame by looping on all instants using the `Instant.rel_to_abs()` method.

Parameters

- **coordinates** (*list(str)*) – list of coordinates names
- **conservative_vars** (*list(str)*) – list of conservative variables names in the following order: density, momentum along the x-axis; momentum along the y-axis, momentum along the z-axis and total energy per unit of volume
- **omega** (*float*) – angular speed of the current base. If `in_attr` use the omega stored in the attrs, necessary if different angular speeds in the base (for example one angular speed per superblock)
- **angle** (*float*) – angular deviation of the current base. If `in_attr` use the angle stored in the attrs, necessary if different angular deviations in the base (for example one angular deviation per superblock and per instant)

Note: may be moved elsewhere in future releases

Warning: the angular speed must be perpendicular to the x-axis

rename_variables(*list_vars*, *list_newvars*, *location=None*)

Rename variables in the dataset.

Parameters

- **list_vars** (*list(str)*) – list of variables to rename
- **list_newvars** (*list(str)*) – list of new variable names
- **location** (str in *LOCATIONS* (page 369)) – if different from `None`, change only the variables at the location specified

Replacement is performed element-wise.

serialized(*data=True*)

Build a pickable representation of the window.

set_computer_model(*modeling*, *species_database=None*, *addons=None*)

Set a computer modeling for the zone.

See `antares.api.Instant.Instant.set_computer_model()`.

set_formula(*formula*)

Set a formula for the dataset (Zone or Boundary).

See `antares.api.Instant.Instant.set_formula()`

set_formula_from_attrs(*name*)

Set a formula from a name in the dataset attribute.

The computer will receive a new formula associated to its current modeling. This formula is included in the zone attribute with the key *name*.

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

update([*E*], ***F*) → None. Update D from mapping/iterable E and F.

If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

values() → an object providing a view on D's values

property attrs

Dictionary `antares.core.AttrsManagement.AttrsManagement` containing the attributes of the Datasets.

container

Attribute (of type `CustomDict`) containing all kind of data associated to the Window, and not constrained to the shape. These data are ignored during a family slicing. optional.

property shared

Attribute (of type `Instant`) containing variables shared for all the Instants contained in the Datasets.

property slicing

Slicing gives the window location in the block.

This attribute is used to slice numpy arrays. Then, it has the following types depending on the mesh structures.

Type
tuple(slice) or list or ndarray, compulsory

- for unstructured grids: list of indices that gives the points (node values) that belong to this boundary
- for structured grids: tuple of slice objects. The slices (start, stop, step) accept only None for the step value. It contains the topological information (six integers in 3D, four in 2D).

zone_name

Name of the Zone in which the slicing should be applied.

Type
str, compulsory

3.3 Antares I/O Classes

3.3.1 Reader

Proxy Class to read files.

Parameters

- **base: Base**
  Base in which the data read should be added. If this key is not used, the reader will create a new Base object, else the input base will be modified.

  example: If no key **base**:

```
reader = antares.Reader(<format>)
...
base = reader.read()
```

  base will contain the data read from the file.

  example: If **base** is not None:

```
reader = antares.Reader(<format>)
reader['base'] = input_base
...
reader.read()
```

input_base will be complemented with the data read from the file.

- **filename:** ***str***
  The name of the file(s). To match several files, two tags can be used:
  - `<zone>`, to match the names of the zones and,
  - `<instant>`, to match the names of the instants. This tag can appear more than once.

  example: If the files are named:

```
Visu_4000_0000_t00, Visu_4000_0001_t00,
Visu_4000_0000_t01, Visu_4000_0001_t01
```

  then the key **filename** could be `Visu_4000_<zone>_<instant>`.

- **shared:** ***bool*****, default=** ***False***
  if True, the data read are added to the shared Instant. The tag `<instant>` can not be used in the **filename** key if this key is True.

- **topology_file:** ***str***
  If one wants to attach topological information to the base, a file giving the topology must be read.

  > **Warning:** The name of the zone read must match the name of the **topology_file** blocks. This means that one might need to set the attribute **zone_prefix**. Moreover, the shape of the zones on the **topology_file** must match the shape of the numpy arrays of each zone/instant located at nodes (if any)

- **topology_format:** ***str*****, default=** ***'elsa'***
  The format of the topology file.

- **bndphys_file:** ***str***
  To be used in combination with **topology_file**

- **zone_prefix:** ***str*****, default=** ***''***
  The prefix to add in front of the zone name.

- **zone_suffix:** ***str*****, default=** ***''***
  The suffix to add back of the zone name.

- **zone_regex:** ***str*****, default=** ***'.*'***
  To interpret the zone name defined by the tag `<zone>` of the attribute **filename**, Antares uses regular expressions. Take a look at the regex[21] module to properly change this argument.

  example: to match a 4-digit number.

```
reader['zone_regex'] = '\\\\d{4}'
```

- **instant_regex:** ***str*** **or** ***tuple(int, int)*** **or** ***tuple(int, int, int)*****, default=** ***'.*'***
  Same as **zone_regex**, but for `<instant>` tag. Only for python >= 2.7, if tuple, then the regex matches the interval between the first and the second integer. If given, the third integer corresponds to the number of digits, and may generate leading zeroes.')

[21] https://docs.python.org/library/re.html?__doc__+='highlight=regex#regular-expression-syntax

example: (0, 3, 1)

- **instant_step:** ***int*****, default=** ***1***
  Take every **instant_step** file into account. This only works when tag `<instant>` or **instant_regex** represent data that can be ordered
- **location:** ***str*** **in** ***LOCATIONS*** **(page 369), default=** ***'node'***
  Location of the variables.

Main functions

class `antares.io.Reader.`**`Reader`**(*file_format=None*, ***kwargs*)

Proxy class for file readers.

`__init__`(*file_format=None*, ***kwargs*)

Create the Reader object.

Parameters
`file_format` (*str*) – the format of the file(s).

`read`()

Create the base from the given information.

Note: Post-execution of `set_coordinate_names()` to set the names of coordinates.

Returns
the antares Base filled with the data from file.

Return type
`Base`

For each format below, the associated keys supplement the above shared settable keys.

Binary Tecplot

Description

Read files with tecplot binary format (v75, v112).

Parameters

- **n_hbt:** ***int*****, default=** ***None***
  If one wants to read HBT/TSM files, put here the number of harmonics.
- **shared_mesh:** ***bool*****, default=** ***False***
  True if the connectivity and the coordinates of nodes change from an instant to another.

Preconditions

Postconditions

The Tecplot **ZoneTitle** is used for the Antares zone name. Therefore, for an unstructured mesh, if two tecplot zones have the same ZoneTitle and the same StrandID, but different element types, this will lead to a single Antares zone in the output base.

The Tecplot title of the header section is set in `Base.attrs` with the key **TecplotHeaderTitle**.

The tecplot parameter **FileType** is set in `Base.attrs` with the key **TecplotFileType**. (see also *antares.io.writer.WriterTecplotBinary.WriterTecplotBinary* (page 45))

The entry **Time** is set in `Instant.attrs`. The value of this solution time is a 64-bit floating-point number.

Example

The following example shows how to read one file.

```
import antares
myr = antares.Reader('bin_tp')
myr['filename'] = 'file.plt'
base = myr.read()
```

The following example shows how to read one file per zone with the tag '<zone>'.

```
import antares
myr = antares.Reader('bin_tp')
myr['filename'] = 'file_<zone>.plt'
base = myr.read()
```

The following example shows how to read one file per zone per instant with the tags '<zone>' and '<instant>'.

```
import antares
myr = antares.Reader('bin_tp')
myr['filename'] = 'file_<zone>_<instant>.plt'
base = myr.read()
```

ASCII Tecplot

Description

Read files with Tecplot ASCII format.

The lazy loading pattern is not available for this format.

Parameters

see Common Parameters from *Reader Proxy* (page 28).

Preconditions

This reader handles structured and unstructured mesh.

The DATA PACKING can be POINT or BLOCK.

Postconditions

Example

The following example shows how to read one file.

```
import antares
myr = antares.Reader('fmt_tp')
myr['filename'] = 'file.dat'
base = myr.read()
```

HDF CGNS

Description

Read files with HDF CGNS format.

This reader 'hdf_cgns' supports MPI parallelism.

Parameters

- **bnd_filename:** ***str***
  The name of the boundary file.
- **base_name:** ***str*****, default=** ***None***
  Name of the node that has the CGNSBase_t label.
- **rename_vars:** ***bool*****, default=** ***True***
  Rename CGNS variables with Antares names.
- **split:** ***str in ['always', 'never'] or None*****, default=** ***None***
  Tell whether to split unstructured zones or not. `None` let the reader decide automatically. Parallel Only.
- **distribution:** ***list(str) or None*****, default=** ***None***
  In parallel, this parameter allows the user to impose the distribution of zones as a list of CGNS 'blockname'. Each process should receive a different subset of names. All zones must be distributed. If no zones are given for one process, the value of this parameter should be an empty list '[]'. When this parameter is used, 'split' is not taken into account (unstructured zones are not splitted).
- **instant_names:** ***list(str) or None*****, default=** ***None***
  List of FlowSolution_t names that will be stored in different instants. Setting this option prevents from reading the BaseIterativeData_t node.

  - example: **instant_names** = ['FlowSolution#Init'] to read the FlowSolution_t data structure named 'FlowSolution#Init' in the file.

- **polyhedra_support:** ***bool*****, default=** ***False***
  If True, Elements_t nodes of ElementType_t NGON_n and NFACE_n are stored as face-based connectivity. If False, those elements are stored as standard elements with vertex-based connectivity if allowed by the elements.

- **follow_links:** ***bool*****, default=** ***False***
  If enabled, all CGNS links are followed, and corresponding CGNS nodes are read. Othewise, no CGNS links are taken into account.

- **force_serial:** ***bool*****, default=** ***False***
  If enabled during a parallel execution, each compute core will read the **filename** independently as for a serial process.

- **base_subregion:** `Base`**, default=** ***None***
  Base where to collect zone subregions. If *None*, subregions are appended in the output base as standard zones.

Preconditions

To use **polyhedra_support**, the 'poly' extension of Antares must be installed. The implementation of 'polyhedra_support' is limited to CGNS version < 3.4.0. ParentElements nodes of NGON_n and NFACE_n nodes are ignored.

The library ngon2elt may have to be generated during the antares installation process for reading face-based connectivity. Refer to *installation of ngon2elt* (page 8).

Postconditions

Reading the BaseIterativeData_t data structure: The identifiers IterationValues and TimeValues are read. This is only available when the option **instant_names** is not used. The identifier IterationValues is preferentially used for the instant names. If the identifier TimeValues is available, then the `Instant` will hold the key 'Time' in its `Instant.attrs`. If the identifier IterationValues is available, then the `Instant` will hold the key 'Iteration' in its `Instant.attrs`.

Reading the ZoneIterativeData_t data structure: The identifier FlowSolutionPointers is read to get the solution field for each recorded time value or iteration. There is an implied one-to-one correspondence between each pointer and the associated TimeValues and/or IterationValues under BaseIterativeData_t.

Only one CGNS GridCoordinates_t data structure is read, and only cartesian coordinates.

Reading the FlowSolution_t data structure: If the symbol "#" is in the label name, what is next is called the suffix. If the suffix only contains digits, then the instant name is the suffix, else it is the global variable *instant_name* (page 370). If the label name only contains digits, then the instant name is this figure. Otherwise, the instant name is the global variable *instant_name* (page 370). If **instant_names** is set, then this sets the instant names.

When **polyhedra_support** is enabled, zone grid connectivities are not handled.

Reading the ZoneSubRegion_t data structure: It is stored in a `Zone` just as the Zone_t data structure. Each antares zone coming from a ZoneSubRegion is associated with a family name. It is then possible to separate ZSR zones from standard zones afterwards. The name of the ZSR and the family is defined by a prefix 'ZSR', the name of the CGNS ZoneSubRegion_t node, and a suffix that is the name of the zone including this ZSR, joined by the character '_'. Data from CGNS DataArray_t are stored in an `Instant` named '0' of the antares zone.

Example

The following example shows how to read one file.

```
import antares
myr = antares.Reader('hdf_cgns')
myr['filename'] = 'file.cgns'
base = myr.read()
```

antares.io.reader.ReaderHdfCgns.**get_base_node_names**(*filename*)

returns the list of Base_t from a file.

Parameters

filename (*str*) – HDF CGNS filename

Reading the list of Base_t from a file

```
import os
from antares.io.reader.ReaderHdfCgns import get_base_node_names
names = get_base_node_names('all_cut.cgns')
print(names)
```

HDF Antares

Description

Read files with antares HDF5 format.

Parameters

- **n_hbt: *int*, default= *None***
  If one wants to read HBT/TSM files, put here the number of harmonics.
- **format: `str` in ['2015', '2022'], default= *'2015'***
  The file format version.
  - format 2015: The zone/instant/variable 'list_keys' are read from h5 attrs.
  - format 2022: The zone/instant/variable 'list_keys' are read from h5 datasets.

Preconditions

Postconditions

Example

The following example shows how to read one file.

```
import antares
myr = antares.Reader('hdf_antares')
myr['filename'] = 'file.hdf'
base = myr.read()
```

HDF LaBS

Description

Read files with HDF Labs format.

Format from Lattice-Boltzmann solver LaBs or ProLB[22]

Parameters

- **velocity_components:** *list(str)*, **default=** *['Vx', 'Vy', 'Vz']*
  Velocity components stored in the instant.
- **velocityRMS_components:** *list(str)*, **default=** *['VxRMS', 'VyRMS', 'VzRMS']*
  RMS components of the velocity vector stored in the instant.
- **velocity_squared_components:** *list(str)*, **default=** *['Vx2', 'Vy2', 'Vz2']*
  Squared velocity components of the velocity vector stored in the instant.
- **shared_mesh:** *bool*, **default=** *False*
  If True, then the connectivity and the coordinates of nodes are shared between all instants. The mesh is fixed during the simulation.
- **Unknown_components:** *list(str)*, **default=** *['UnknownX', 'UnknownY', 'UnknownZ']*
  Unknown components found in the file. Writer need to be adapted.
- **version:** *str*, **default=** *'v2'*
  ProLB version used, either 'v2' or 'v3'

Preconditions

Allowed elements: triangles and hexahedra.

Postconditions

Example

The following example shows how to read one file.

```
import antares
myr = antares.Reader('hdf_labs')
myr['filename'] = 'file.hdf'
base = myr.read()
```

[22] http://www.prolb-cfd.com

VTK

VTK format.

Warning: dependency on VTK[23]

Parameters

- **coordinates:** ***list(str)***
  Names of variables that define the mesh when it is not specified in the file.
- **single_type:** ***bool*****, default=** ***False***
  Tell if the file contains only a unique type of elements for an unstructured grid.

CSV

Description

Read files with CSV (comma-separated values) format.

Parameters

- **separator:** ***str*****, default=** ***';'***
  Separator between numbers.
- **cantera_avbp:** ***bool*****, default=** ***False***
  Specific for cantera_avbp format.

Preconditions

If any, the lines of the header must contain the name of variables separated by the separator string. Columns may have different sizes. Rows are composed of numbers (integer or floating-point) separated by the separator string.

Postconditions

Example

The following example shows how to read one file.

```
import antares
myr = antares.Reader('csv')
myr['filename'] = 'file.dat'
myr['separator'] = ','
base = myr.read()
```

[23] https://www.vtk.org/

Fluent

Description

Read Fluent files with Binary data.

The implementation is based on *ANSYS Fluent Meshing User's Guide Release 15.0 November 2013* and ANSYS Fluent Meshing User's Guide Release 12.0 January 2009:[24]

Parameters

- **filename:** ***str***
  The mesh or solution file name. If **base** is not given, then **filename** must be the mesh file. If **base** is given, then the **filename** must be the solution file. The mesh file must have the extension '.cas' and the solution file must have the extension '.dat'.
- **base:** ***str*****, default=** ***None***
  Input base used only if the mesh file has already been read.
- **xfile:** ***str*****, default=** ***None***
  Path name of the file xfile.h. It is compulsory if a solution **filename** is given. This file can be found in the user's Fluent installation.

Precondition

The reader does not support multi-instant data and ASCII data.

The reader only supports face-based connectivity.

The library ngon2elt has to be generated during the antares installation process. Refer to *installation of ngon2elt* (page 8).

Precondition

A grid connectivity is stored in a Boundary object as a regular physical boundary condition.

Examples

The following example shows how to only read the mesh file.

```
import antares

# read mesh only
myr = antares.Reader('fluent')
myr['filename'] = 'file.cas'
base = myr.read()
```

The following example shows how to read both mesh and solution files.

[24] https://www.afs.enea.it/project/neptunius/docs/fluent/html/ug/node1464.htm

```
import antares

# read mesh
myr = antares.Reader('fluent')
myr['filename'] = 'mesh_path/mesh.cas'
base = myr.read()
# read solution
myr['base'] = base
myr['filename'] = 'solution_path/solution.dat'
myr['xfile'] = 'xfile_path/xfile.h'
base = myr.read()
```

PYCGNS

Description

Read in-memory pyCGNS objects.

https://pycgns.github.io/

https://github.com/pyCGNS/pyCGNS

Parameters

- **object: `object`**
  Source object to be read.
- **polyhedra_support: *bool*, default= *False***
  If enabled, Elements_t nodes of ElementType_t NGON_n and NFACE_n are stored as polyhedra. Otherwise, for cells allowing it, those elements are stored as canonical elements.
- **base_subregion: `Base`, default= *None***
  Base where to collect zone subregions. If *None*, subregions are appended in the output base as standard zones.
- **skip_solution: *bool*, default= *False***
  If enabled, do not read FlowSolution data, read only geometric information.

Preconditions

Zones must be structured, or unstructured with face-based connectivities. Element-based connectivities are not read for unstructured zones.

To use **polyhedra_support**, the 'poly' extension of Antares must be installed. The implementation of 'polyhedra_support' is limited to CGNS version < 3.4.0. ParentElements nodes of NGON_n and NFACE_n nodes are ignored.

The library ngon2elt may have to be generated during the antares installation process for reading face-based connectivity. Refer to *installation of ngon2elt* (page 8).

Postconditions

Zone grids are stored in the shared instant. Zone solutions and BC data are stored in a unique instant named '0000'. BC data shall have the same geometric slicing as its parent boundary. Small CGNS parameter nodes (low memory usage) that are unknown to antares but that may be required for future applications are stored as is under antares attrs attribute of the bound object.

Reading the ZoneSubRegion_t data structure: It is stored in a `Zone` just as the Zone_t data structure if **base_subregion** is None. Each antares zone coming from a ZoneSubRegion is associated with a family name. It is then possible to separate ZSR zones from standard zones afterwards. The name of the ZSR and the family is defined by a prefix 'ZSR', the name of the CGNS ZoneSubRegion_t node, and a suffix that is the name of the zone including this ZSR, joined by the character '_'. Data from CGNS DataArray_t are stored in an `Instant` named '0' of the antares zone.

Example

The following example shows how to read one object.

```
import antares
import CGNS.PAT.cgnskeywords as _ck
import CGNS.PAT.cgnsutils as _cu
import CGNS.MAP

tree, _, _ = CGNS.MAP.load('file.cgns')
cgns_bases = _cu.hasChildType(tree, _ck.CGNSBase_ts)

sp_reader = Reader('pycgns')
sp_reader['object'] = cgns_bases[0]
base = sp_reader.read()
```

HDF5 AVBP

Description

Read files from the AVBP[25] code

More information on the format can be found here[26].

The root node of the hierarchical HDF5 data structure may contain groups, and groups may contain datasets.

Parameters

see Common Parameters from *Reader Proxy* (page 28).

- **filename:**
  It is not possible to read a mesh file and at least one solution file in one shot (see example below).
- **location:**
  Not used.
- **groups_vars:** ***list or tuple***
  Select subsets of groups and variables from the solution file.

[25] https://www.cerfacs.fr/avbp7x
[26] https://ant.cerfacs.fr/projects/antares_safran/wiki/HDF5AVBP

- example to read only some variables possibly stored under different groups:

```
groups_vars = ['rhou', 'rhov']
```

- example to read 'rhou' and 'rhov' from group1, 'pressure' from group2, and all variables from group3:

```
groups_vars = (('group1', ['rhou', 'rhov']),
               ('group2', ['pressure']),
                'group3')
```

Preconditions

If the group 'Coordinates' exists, then the file is considered to be an AVBP mesh file. If one group among ['Additionals', 'GaseousPhase', 'RhoSpecies', 'Reactions', 'LiquidPhase', 'FictiveSpecies', 'Sparkignition', 'RealGas', 'LiquidPhase_ptcl', 'Parameters_ptcl'] exists, then the file is considered to be an AVBP solution file.

Postconditions

Mesh File:

If the option **shared** is set to True, then the instant is the shared instant of the zone, else the instant is a regular instant.

Coordinates x, y and z in 3-D only, are lazy loaded.

All variables in the group 'Parameters' are read, and set in the `Zone.attrs`.

Volume connectivities, `Instant.connectivity`, are eager loaded.

All variables from the group 'VertexData' are lazy loaded. The name of variables are stored in the `Instant.attrs` to be able to separate mesh data from solution data in the AVBP writer for example.

If the group 'Boundary' exists, then some *`Boundary`* (page 11) objects are created. The number of objects is the size of the dataset 'bnode_lidx'. The name of boundaries are given by the dataset 'PatchLabels', otherwise they are named 'Patch<number>'. Some datasets of the group 'Boundary' or groups at the root node are not used by antares, but they are still stored in the *`Boundary.container`* (page 18) (for use in the AVBP writer for example). These groups are ['Periodicity', 'Patch'] and these datasets are ['PatchGeoType', 'Patch->area', 'bnode->normal', 'bnd_<elt_type>->face', 'bnd_<elt_type>->elem']. All boundaries are gathered in a *`Family`* (page 20) named 'Patches'.

Solution File:

If the group 'Average' exists, then the key 'status' is set in the `Instant.attrs` with the value 'Average' to be able to separate average solution from instantaneous solution in the AVBP writer for example.

All datasets in all groups are read. If the shape of a dataset is equal to the shape of the instant, then the dataset is stored as a variable in the instant, else the dataset is stored in the `Instant.attrs`.

Example

The following example shows how to read one mesh file in the shared instant, and to read a whole solution file.

```
import antares
myr = antares.Reader('hdf_avbp')
myr['filename'] = 'file.mesh.h5'
myr['shared'] = True
base = myr.read()

myr = antares.Reader('hdf_avbp')
myr['filename'] = 'sol_ite0001000.h5'
myr['base'] = base
base = myr.read()
```

The following example shows how to read some specific groups and datasets from a solution file.

```
myr = antares.Reader('hdf_avbp')
myr['filename'] = 'sol_ite0001000.h5'
myr['groups_vars'] = ('GaseousPhase', ('Additionals', ['pressure']),
                      'RhoSpecies')
myr['base'] = base
base = myr.read()
```

3.3.2 Lazy Loading Pattern

The Lazy Loading Design Pattern is applied to antares Readers to defer the storage of variables in numpy arrays until the point at which they are required. Therefore, this can save reading time and memory if all the variables do not have to be read from the file. The opposite of lazy loading is eager loading where all variables of a file are read at once.

If all variables have to be read, then the lazy loading may imply a reading time overhead. In that case, you can activate the *File Cache System* (page 371).

3.3.3 Writer

Proxy Class to write files.

Parameters

- **base: `Base`**
  Base to dump in a file.
- **filename:** ***str***
  The name of the output file. Two tags may be used to tell Antares which part of the filename will be variable, the `<zone>` tag and the `<instant>` tag.

  For example, if one has a base made of two zones (e.g. rotor and stator) and two instants per rows (e.g. t00 and t01), if the filename is set as `<zone>_instant_<instant>.dat`, then the written files will be:

  ```
  rotor_instant_t00.dat
  rotor_instant_t01.dat
  stator_instant_t00.dat
  stator_instant_t01.dat
  ```

- **zone_format:** ***str*, default=** ***'%s'***
  To interpret the zone name defined by the tag `<zone>` of **filename**, Antares uses string formatting. Only two regular expressions are available by now, `%s` and `%0[0-9]*d`.
- **instant_format:** ***str*, default=** ***'%s'***
  To interpret the instant name defined by the tag `<instant>` of **filename**, Antares uses string formatting. Only two regular expressions are available by now, `%s` and `%0[0-9]*d`.
- **topology_format:** ***str***
  The format of the topology file. If not given, the topology file will not be written.
- **scripts_format:** ***str***
  The format of the cfd scripts. If not given, the cfd scripts will not be written.
- **bnd_dir:** ***str*, default=** ***'./'***
  Location of the boundary data directory. Useful when **topology_format** is given.
- **scripts_dir:** ***str*, default=** ***'./'***
  Location of the cfd scripts directory. Useful when **scripts_format** is given.
- **coordinates:** ***list(str)***
  The mesh coordinates (used for some format like vtk).
- **dtype:** ***str*, default=** ***'float64'***
  The data type used to dump data.
- **memory_mode:** ***bool*, default=** ***False***
  If memory mode is True, the base is deleted on the fly to limit memory usage.

Main functions

class antares.io.Writer.**Writer**(*file_format=None*, ***kwargs*)

Proxy class for file writers.

__init__(*file_format=None*, ***kwargs*)

Initialize the Writer instance.

Parameters
file_format (*str*) – the format of the file(s).

dump()

Dump all Zones and all Instants of a Base to files.

For each format below, the associated keys supplement the above shared settable keys.

HDF5 CGNS

Description

Write files with the CGNS[27] format.

This writer 'hdf_cgns' supports MPI parallelism.

The root node of the hierarchical HDF5 data structure may contain groups, and groups may contain datasets.

[27] https://cgns.github.io

Parameters

see Common Parameters from *Writer Proxy* (page 41).

- **link:** ***str*****, default=** ***'single'***
  Can take the value:
  - 'zone': 1 file per zone + 1 master file
  - 'proc': 1 file per processor + 1 master file
  - 'single': 1 file, different possible strategies
  - 'merged': write 1 temporary file per processor then merge them into a single file
- **strategy:** ***str*****, default=** ***'one_pass'***
  Only for **link** = 'single'. Can take the value:
  - 'one_pass': the file is collectively created and filled in one pass (encouraged in sequential)
  - 'two_passes': the file structure is created during a first pass by one processor and it is filled individually in parallel (encouraged in parallel)
  - 'update_copy': the file is copied by one processor and its solution is updates individually in parallel (encouraged for a base having a large number of zones but a few data)
- **source_filename:** ***str*****, default=** ***None***
  Name of the CGNS file to copy and update solution. It must have strictly the same structure (zones and shape of zones) as the **base**. Only for **link** = 'single' and **strategy** = 'update_copy'.
- **append:** ***bool*****, default=** ***False***
  If True, append the **base** in the existing CGNS file.
- **base_name:** ***str*****, default=** ***BASE_NAME***
  Name of the CGNSBase_t node.
- **rename_vars:** ***bool*****, default=** ***True***
  Rename Antares variables with CGNS names.
- **mixed_conn:** ***bool*****, default=** ***False***
  Write all elements in a single CGNS node as mixed elements.

Preconditions

The input base could contain structured and unstructured zones.

Postconditions

The output only contain one time solution.

Example

The following example shows how to write a file 'case.cgns'.

```
import antares
writer = Writer('hdf_cgns')
writer['filename'] = 'case.cgns'
writer['base'] = base
writer.dump()
```

Tecplot (binary)

Description

Write files with tecplot binary format (v75, v112).

https://cerfacs.fr/antares/doc/_downloads/bin_tp_V112_data_format.dat

Parameters

-

Preconditions

The Tecplot title of the header section is searched in `Base.attrs` with the key **TecplotHeaderTitle**.

The Tecplot parameter **FileType** is searched in `Base.attrs` with the key **TecplotFileType**.

The attrs **Time** is searched in `Instant.attrs`. If the values of `Instant.attrs['Time']` are the same for some `Instant`, then the corresponding instants will be considered identical, and only one of them will be considered in the output file. Therefore, the output file will not contain all the `Instant` of the **base**.

Postconditions

An instant may contain ('rho', 'cell') and ('rho', 'node') for example, then the tecplot variable names would be 'rho_cell' and 'rho'.

Variables and connectivities that are shared are not written as such in the tecplot file. So if you read the written file back in Antares, it will take more memory.

If there are only variables located at cell in the base, data are written as node values.

Example

The following example shows how to write one file.

```
import antares
myw = antares.Writer('bin_tp')
myw['filename'] = 'file.plt'
myw['base'] = base
myw.dump()
```

The following example shows how to write one file per zone with the tag '<zone>'.

```
import antares
myw = antares.Writer('bin_tp')
myw['filename'] = 'file_<zone>.plt'
myw['base'] = base
myw.dump()
```

The following example shows how to write one file per zone per instant with the tags '<zone>' and '<instant>'.

```
import antares
myw = antares.Writer('bin_tp')
myw['filename'] = 'file_<zone>_<instant>.plt'
myw['base'] = base
myw.dump()
```

class antares.io.writer.WriterTecplotBinary.**WriterTecplotBinary**

Writer to write bin_tp format (structured and unstructured).

https://ant.cerfacs.fr/projects/antares/wiki/DataFileFormats

> **Warning:** Variables and connectivities that are shared are not written as such in the tecplot file. So if you read the written file back in Antares, it will take more memory

The Tecplot title of the header section is searched in the base.attrs with the key **TecplotHeaderTitle**.

The Tecplot parameter **FileType** is searched in the base.attrs with the key **TecplotFileType**.

The attrs **Time** is searched in the Instants.

An instant may contain: ('rho', 'cell') and ('rho', 'node'), then the tecplot variable names would be 'rho_cell' and 'rho'.

CSV

Comma-separated values CSV format.

Parameters

- **separator:** ***str*****, default=** ***;***
  Separator character between numbers.
- **with_col_num:** ***bool*****, default=** ***True***
  Add column number to variable names if enabled: 'a' -> 'a(0)'.

class antares.io.writer.WriterCSV.**WriterCSV**

Writer for CSV (Comma-separated values) format.

The first line contain the name of variables separated by the separator string. Columns may have different sizes. Rows are composed of numbers separated by the separator string.

This is an Instant writer. So, if many zones are in a base, then it will write many files if the tag '<zone>' is in the 'filename'.

This is an Instant writer (It only writes an Instant in a file). So, if many zones are in a base, then it will write many files if the tag '<zone>' is in the 'filename'. Otherwise, the zone name is appended at the end of the 'filename'.

Binary VTK

VTK format.

Parameters

- **append:** ***bool*****, default=** ***False***
  If True, append the base in the existing '_xmlbase.pvd' file. This allows the user to write zones and instants on-the-fly.

class antares.io.writer.WriterVtk.**WriterVtk**

HDF5 Jaguar Restart

Writer HDF JAGUAR RESTART

Parameters

- **strategy:** ***str*****, default=** ***'h5py_ll'***
  Can be one of:
  - 'monoproc': the file is created by one proc. This must be used for initial solution creation only.
  - 'h5py_hl': Parallel write with h5py high level API. Simple but slow because writes multiple times to reorder cells.
  - 'h5py_ll': Parallel write with h5py low level API. Use hyperslabs to perform reordering with only one write.
  - 'fortran': Parallel write with fortran HDF5 write. Most efficient way to write but need compilation.

class antares.io.writer.WriterHdfJaguarRestart.**WriterHdfJaguarRestart**

Writer to write bases for high order JAGUAR solver.

Write only one time solution.

Seq: Write only initial solution. Parallel: Reorder and write high order solution from Jaguar coprocessing.

Several parallel write modes have been tested:

- h5py_sorted_highAPI: 'h5py_hl'
- h5py_sorted_lowAPI: 'h5py_ll'
- fortran_sorted_opt2: 'fortran'

In sorted results, the efficiency is from best to worst: fortran > h5py_ll > h5py hl

Here, only h5py low level API (restart_mode=h5py_ll) is used because it has an efficiency near of the pure fortran, but it is much easier to modify and debug. For final best performance, a pure fortran code can be derived quite easily based of existing one from the python h5py_ll code.

HDF5 AVBP

Description

Write files for the AVBP[28] code

More information on the format can be found here[29].

The root node of the hierarchical HDF5 data structure may contain groups, and groups may contain datasets.

Parameters

see Common Parameters from *Writer Proxy* (page 41).

- **filename:** ***str***
  Tags <zone> and <instant> are not used.
- **groups_vars:** ***list or tuple***
  Give the location of solution variables in AVBP groups for the solution file.
  - example to write 'rhou' and 'rhov' in group1 and 'pressure' in group2:

    ```
    groups_vars = (('group1', ['rhou', 'rhov']),
                   ('group2', ['pressure']))
    ```

  - example to write all variables in group1:

    ```
    groups_vars = 'group1'
    ```

Warning: you may end up with a file that does not respect the AVBP format.

[28] https://www.cerfacs.fr/avbp7x
[29] https://ant.cerfacs.fr/projects/antares_safran/wiki/HDF5AVBP

Preconditions

The input base must have a single zone and a single instant (or a shared instant).

The instant may contain the group names as keys in its `Instant.attrs`. The values are lists of avbp variable names. e.g.:

instant.attrs['GaseousPhase'] = ['rhou', 'rho']

instant.attrs['Additionals'] = ['tau_turb_x', 'tau_turb_y', 'tau_turb_z']

This information is used to write variables in the right groups.

Postconditions

The outputs are the mesh file named **filename**.mesh.h5 and the solution file named **filename**.sol.h5. The file named **filename**.asciiBound.key is output if the zone has boundary conditions.

Mesh File:

If some instant variables are in *`antares.core.Constants.KNOWN_COORDINATES`* (page 369), then the mesh file is created. If the zone contains boundary conditions in `Zone.boundaries`, then the file asciiBound.key is created.

If the coordinates of the **base** are not ['x', 'y', 'z'], then they are renamed in-place.

The group 'Parameters' is created. All entries of `Zone.attrs` are written to datasets in this group.

The group 'Coordinates' and its datasets ['x', 'y', 'z'] are created.

The group 'Connectivity' and its datasets '<elt_type>->node' are created.

If the zone contains boundary conditions in `Zone.boundaries`, then the group 'Boundary' is created. All entries in *`Boundary.container`* (page 18) compatible with the AVBP format will be considered, the groups ['Periodicity', 'Patch'] and the datasets ['PatchGeoType', 'Patch->area', 'bnode->normal', 'bnd_<elt_type>->face', 'bnd_<elt_type>->elem'].

All variables from the entry 'VertexData' of `Instant.attrs` and the variable 'volume' will be created as datasets in the group 'VertexData'. If no such variables are in the instant, then the group is not created.

Solution File:

If some variables are different from ['x', 'y', 'z'] and the variables contained in the entry 'VertexData' of `Instant.attrs`, then the solution file is created.

The group 'Parameters' is created. All entries of `Zone.attrs` in ['dtsum', 'nit_av', 'niter', 'nnode', 't_av', 'versionstring', 'gitid'] are written to datasets in this group. By default, ['dtsum', 'niter', 'nnode', 'versionstring'] are set to [0, 0, <instant.shape>, 'AVBP Version V7.X'].

All variables are written in the following groups ['GaseousPhase', 'Reactions', 'RhoSpecies', 'LiquidPhase', 'FictiveSpecies', 'Additionals', 'Sparkignition', 'RealGas', 'Average'] or new user-defined groups following the option **groups_vars** or the AVBP file format.

Example

The following example shows how to write a mesh file 'case.mesh.h5', and a solution file 'case.sol.h5', and a file 'case.asciiBound.key'.

```
import antares
writer = Writer('hdf_avbp')
writer['filename'] = 'case'
writer['base'] = base
writer.dump()
```

The following example shows the same thing, but with some user modification for the placement of some variables in groups.

```
import antares
writer = Writer('hdf_avbp')
writer['filename'] = 'case'
writer['base'] = base
writer['groups_vars'] = (('Additionals', ['rho', 'rhou']),
                         ('Reactions', ['pressure']))
writer.dump()
```

HDF5 antares

Description

Write files with antares HDF5 format.

Parameters

see Common Parameters from *Writer Proxy* (page 41).

- **format: `str` in ['2015', '2022'], default= *'2015'***
  The file format version.
  - format 2015: The zone/instant/variable 'list_keys' are stored in h5 attrs.
  - format 2022: The zone/instant/variable 'list_keys' are stored in h5 datasets. This format can handle very long lists of strings.
- **xmf: `bool`, default= *True***
  Write the associated .xmf file.

Preconditions

Postconditions

The name of the **filename** is suffixed with '.h5'. The name of the **filename** is suffixed with '.xmf' if **xmf** is True.

The .xmf file is not created for 1D data.

A base level attribute 'Antares version' is automatically added to the base. This attribute contains the version Antares used to write the files.

The base, zone and instant level attributes are written in the following groups:

- "Parameters" under the root group for base level attributes.
- "Zone parameters" under the zone group for zone level attributes.
- "Instant parameters" under the instant group for instant level attributes.

Example

The following example shows how to write one file.

```
import antares
myw = antares.Writer('hdf_antares')
myw['filename'] = 'file'
myw['base'] = base
myw.dump()
```

Available Formats:

3.3.4 Available Reader Formats

Check the specificity of each reader in the table below.

Table 2: File Formats (**S** = structured, **U** = unstructured, **MZ** = multi-zone, **MI** = multi-instant)

Format	S	U	MZ	MI	Extended description	library needed
bin_tp (page 30)	•	•	•	•	binary Tecplot[30] format version 112 and 75. The 75 version was developed for compatibility with elsA[31].	
fmt_tp (page 31)	•	•			ASCII Tecplot[32] format	
hdf_avbp (page 39)		•			HDF5[33] AVBP[34] solution or mesh (topological boundary information included)	h5py[35]
bin_avbp	•				binary file written by AVBP[36] (temporal)	
hdf_cgns (page 32)	•	•	•		HDF5[37] file with CGNS meshes (time solution are not taken into account for now). To convert an ADF file to HDF, use the adf2hdf utility program. Extra elements are added to take into account for some specificities introduced by Numeca[38] Autogrid/IGG[39] meshing software.	h5py[40]
netcdf	•	•			Network Common Data Form (NetCDF)[41]	
hdf_antares (page 34)	•	•	•	•	HDF5[42] format specific to Antares	h5py[43]
hdf_labs (page 35)		•	•	•	HDF5[44] format specific to (written by) LaBS. (triangles and hexahedra)	h5py[45]
column	•				column formatted file. If any, the lines of the header must start with a '#' sign. All columns must have the same size. Rows are composed of numbers separated by spaces.	numpy[46]
csv (page 36)	•				Comma-separated value. If any, the lines of the header must contain the name of variables separated by the separator string. Columns may have different sizes. Rows are composed of numbers separated by the separator string.	numpy[47]
igor	•				IGOR[48] formatted 1D signal	
matlab	•				Matlab[49]	scipy[50]
bin_v3d	•				binary voir3d elsA[51] file	
fmt_v3d	•				formatted voir3d elsA[52] file	
bin_plot3d	•				binary plot3D grid, Q or function files	
3.3. Antares I/O Classes						**51**
ens_case	•			•	Ensight[53]	vtk[54]

Table 3: Topology Formats

Format	Extended description
elsa	Topology from elsA[62] input file

Documentation of the file formats can be found on the redmine server[63].

[30] https://www.tecplot.com
[31] https://elsa.onera.fr
[32] https://www.tecplot.com
[33] https://www.hdfgroup.org/HDF5
[34] https://www.cerfacs.fr/avbp7x
[35] https://www.h5py.org
[36] https://www.cerfacs.fr/avbp7x
[37] https://www.hdfgroup.org/HDF5
[38] https://www.numeca.com
[39] https://www.numeca.com/index.php?id=411
[40] https://www.h5py.org
[41] https://www.unidata.ucar.edu/software/netcdf
[42] https://www.hdfgroup.org/HDF5
[43] https://www.h5py.org
[44] https://www.hdfgroup.org/HDF5
[45] https://www.h5py.org
[46] https://docs.scipy.org/doc/numpy
[47] https://docs.scipy.org/doc/numpy
[48] https://www.wavemetrics.com
[49] https://www.mathworks.com/products/matlab
[50] https://docs.scipy.org/doc/scipy/reference/index.html
[51] https://elsa.onera.fr
[52] https://elsa.onera.fr
[53] https://www.ceisoftware.com
[54] https://www.vtk.org
[55] https://www.ilight.com/en/products
[56] https://www.vtk.org
[57] https://gmsh.info
[58] http://www.prolb-cfd.com
[59] https://pycgns.github.io/
[60] https://github.com/pycgns
[61] https://www.ansys.com/fr-fr/products/fluids/ansys-fluent
[62] https://elsa.onera.fr
[63] https://ant.cerfacs.fr/projects/antares/wiki/DataFileFormats

3.3.5 Available Writer Formats

Table 4: File Formats (**S** = structured, **U** = unstructured, **MZ** = multi-zone, **MI** = multi-instant)

Format	S	U	MZ	MI	Extended description	library needed
bin_tp (page 44)	•	•	•	•	binary Tecplot[64] format version 112	
bin_tp_75	•				binary Tecplot[65] format version 75 developed for compatibility with elsA[66]	
fmt_tp	•	•			formatted Tecplot[67] format	
hdf_avbp (page 47)		•			HDF5[68] AVBP[69] solution or mesh (topological boundary information included).	h5py[70]
hdf_cgns (page 42)	•	•	•		HDF5[71] file with CGNS[72] meshes Time solution are not taken into account for now. To convert an ADF file to HDF, use the adf2hdf utility program. Extra elements are added to take into account for some specificities introduced by Autogrid/IGG[73] meshing software.	h5py[74]
hdf_antares (page 49)	•	•	•	•	HDF5[75] format specific to Antares with XDMF[76] associated file.	h5py[77]
bin_vtk (page 46)	•	•			vtk[78] binary format	vtk with numpy support
fmt_vtk	•	•			vtk[79] formatted format	vtk with numpy support
column	•				column formatted file. The header is made of two lines beginning with the '#' sign. The second line contains the name of variables separated by a space. All columns must have the same size. Rows are composed of numbers with the format '%.10e' separated by the tabulation pattern 't'.	numpy[80]
bin_column	•				binary column file	numpy[81]
csv (page 45)	•				Comma-separated value. The first line contain the name of variables separated by the separator string. Columns may have different sizes. Rows are composed of numbers separated by the separator string.	numpy[82]
gnuplot_2d	•				GNUPLOT[83] matrix formatted data file	
igor		•			IGOR[84] formatted 1D signal	
mat-lab		•			Matlab[85] file	scipy[86]

Table 5: Topology Formats

Format	Extended description
elsa	Topology output file for elsA[94]

Table 6: Script Formats

Format	Extended description
elsa	Script output files for elsA[95]

Documentation of the file formats can be found on the redmine server[96].

3.4 Antares Treatment Classes

3.4.1 Treatments

The class Treatment is a proxy class for real treatments. A Treatment implements a high-level processing procedure. Only one method **execute()** is to be used.

For instance, the following statements show how to duplicate a mesh.

```
>>> import antares
>>> dup = antares.Treatment('duplication')
>>> # in interactive mode, this prints the keys that can be set and
```

(continues on next page)

[64] https://www.tecplot.com
[65] https://www.tecplot.com
[66] https://elsa.onera.fr
[67] https://www.tecplot.com
[68] https://www.hdfgroup.org/HDF5
[69] https://www.cerfacs.fr/avbp7x
[70] https://www.h5py.org
[71] https://www.hdfgroup.org/HDF5
[72] https://cgns.github.io/CGNS_docs_current/index.html
[73] https://www.numeca.com/index.php?id=411
[74] https://www.h5py.org
[75] https://www.hdfgroup.org/HDF5
[76] https://www.xdmf.org/index.php/XDMF_Model_and_Format
[77] https://www.h5py.org
[78] https://www.vtk.org
[79] https://www.vtk.org
[80] https://docs.scipy.org/doc/numpy
[81] https://docs.scipy.org/doc/numpy
[82] https://docs.scipy.org/doc/numpy
[83] https://www.gnuplot.info
[84] https://www.wavemetrics.com
[85] https://www.mathworks.com/products/matlab
[86] https://docs.scipy.org/doc/scipy/reference/index.html
[87] https://elsa.onera.fr
[88] https://elsa.onera.fr
[89] https://www.ilight.com/en/products
[90] https://en.wikipedia.org/wiki/STL_(file_format)
[91] https://gmsh.info
[92] https://www.h5py.org
[93] https://en.wikipedia.org/wiki/PLY_(file_format)
[94] https://elsa.onera.fr
[95] https://elsa.onera.fr
[96] https://ant.cerfacs.fr/projects/antares/wiki/DataFileFormats

(continued from previous page)

```
>>> # their default values (if any).
>>> print(dup)
List of keys (default value)
    - base
    - vectors ([])
    - coordinates (['x', 'y', 'z'])
    - nb_duplication (in_attr)
    - pitch (in_attr)
>>> # set the different keys
>>> dup['base'] = base
>>> dup['nb_duplication'] = 12
>>> dup['pitch'] = 2 * 3.14157 / 10
>>> result = dup.execute()
```

Several treatments are available. They are the real subjects of the proxy class.

Some implementations include a minimum set of functionalities, such as a wrapping of VTK functions, to deliver results. Some other are more complex, and use other treatments for example. You may have a look at them if you need to buid your own treatment or an application script.

Discrete Fourier Transform

Description

This treatment performs a Discrete Fourier Transform (DFT) on all the given variables of a 3-D finite time-marching result.

The Fourier transform of a continuous-time signal $x(t)$ may be defined as:

$$X(f) = \int_{-\infty}^{+\infty} x(t) e^{-j2\pi f t} dt$$

The Discrete Fourier Transform implemented replaces the infinite integral with a finite sum:

$$X(k) = \frac{1}{N} \sum_{n=0}^{N-1} x(n) e^{-j2\pi \frac{nk}{N}} dt$$

where $x(n)$ is the N sampling terms of an analogic signal $x(t) = x(n\Delta t)$ and the N terms $X(k)$ are an approximation of the Fourier transform of this signal at the mode frequency defined as $f_k = k\Delta f/N = k/T$.

with:

- the sampling frequency: $\Delta f = \frac{1}{\Delta t}$
- the sampling interval: $T = N\Delta t$
- the mode $k = f_k \times T$

Construction

```
import antares
myt = antares.Treatment('dft')
```

Parameters

- **base: `Base`**
  The base on which the Fourier modes will be computed. It can contain several zones and several instants. DFT is performed on all variables except **coordinates**.
- **type: *str*, default= *'mod/phi'***
  The DFT type of the output data: *'mod/phi'* for modulus/phase decomposition or *'im/re'* for imaginery/real part decomposition. The phase is expressed in degrees.
- **coordinates: *list(str)*, default= `antares.Base.coordinate_names`**
  The variable names that define the set of coordinates. The coordinates will not be computed by the DFT treatment.
- **mode: *lists(int)*, default= *None***
  Give one mode or a list of mode ([1, 2, 4] for example). If empty, this returns all the mode including the mean part.

Preconditions

All the zones must have the same instant.

Postconditions

If dtype_in = 'mod/phi', the phase is expressed in degrees.

Example

```
import antares
myt = antares.Treatment('dft')
myt['base'] = base
myt['type'] = 'mod/phi'
myt['mode'] = [4, 18, 36]
dft_modes = myt.execute()
```

Warning: A mode is defined as $k = f_k \times T$, with T the sampling interval.

Main functions

class antares.treatment.TreatmentDft.**TreatmentDft**

execute()

Execute the treatment.

Returns

a base that contains many zones. Each zone contains one instant. Each instant contains two arrays (the FFT parts depending on the type of decomposition).

Return type

Base

Example

```
"""
This example illustrates the Discrete Fourier Transform
treatment of Antares.
"""
import os
if not os.path.isdir('OUTPUT'):
    os.makedirs('OUTPUT')

from antares import Reader, Treatment, Writer

# ------------------
# Reading the files
# ------------------
reader = Reader('bin_tp')
reader['filename'] = os.path.join('..', 'data', 'ROTOR37', 'GENERIC', 'flow_<zone>_
↪<instant>.dat')
base = reader.read()

# ----
# DFT
# ----
treatment = Treatment('dft')
treatment['base'] = base
treatment['type'] = 'mod/phi'
treatment['mode'] = list(range(0, 2))
result = treatment.execute()

# -------------------
# Writing the result
# -------------------
writer = Writer('bin_tp')
writer['filename'] = os.path.join('OUTPUT', 'ex_dft_<instant>.plt')
writer['base'] = result
writer.dump()
```

Dynamic Mode Decomposition

Description

This processing computes a Dynamic Mode Decomposition (DMD) on 2D/3D field.

The DMD allows to characterize the evolution of a complex non-linear system by the evolution of a linear system of reduced dimension.

For the vector quantity of interest $v(x_n, t) \in \mathbb{R}^n$ (for example the velocity field of a mesh), where t is the temporal variable and x_n the spatial variable. The DMD provides a decomposition of this quantity in modes, each having a complex angular frequency ω_i:

$v(x_n, t) \simeq \sum_{i=1}^{m} c_i \Phi_i(x_n) e^{j\omega_i t}$

or in discrete form:

$v_k \simeq \sum_{i=1}^{m} \lambda_i^k c_i \Phi_i(x_n),$ with $\lambda_i^k = e^{j\omega_i \Delta t}$

with Φ_i are the DMD modes.

Using the DMD, the frequencies ω_i are obtained with a least-square method.

Compared to a FFT, the DMD has thus the following advantages:

- Less spectral leaking
- Works even with very short signals (one or two periods of a signal can be enough)
- Takes advantage of the large amount of spatial information

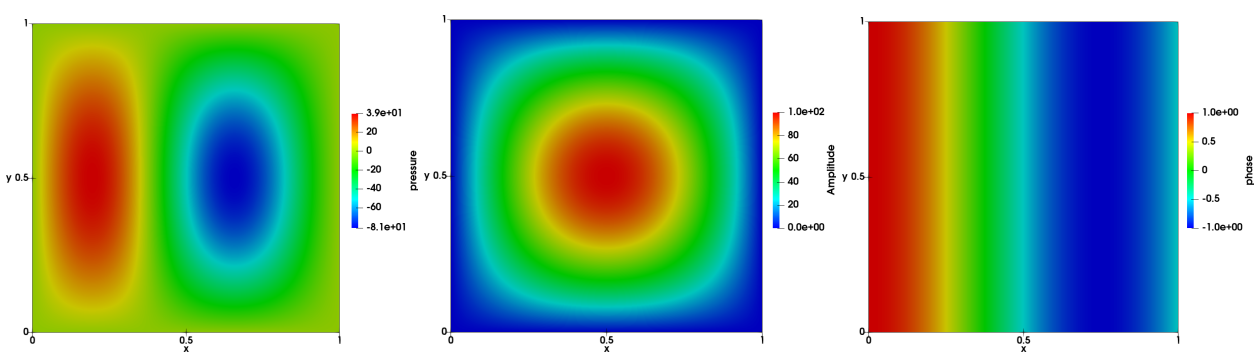
To illustrate the tool, the following reference case is chosen: a wave on a square 2D mesh of size 100×100 from $(x, y) = (0, 0)$ to $(x, y) = (1, 1)$. The unsteady field

$p(x, y, t) = Asin(n_x \pi x) sin(n_y \pi y) cos(\omega t + \Phi(x))$

is shown in the left figure below. The amplitude is shown in the middle figure and the phase on the right figure.

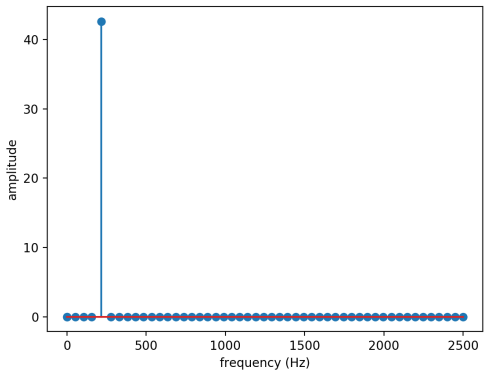
The following parameters are used:

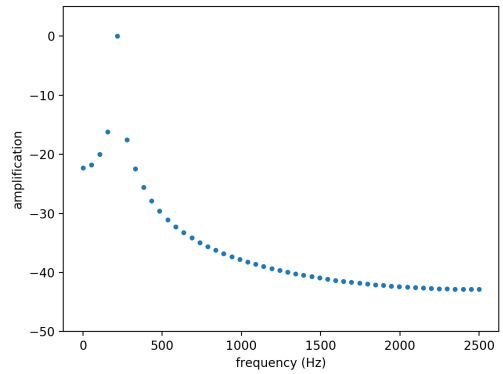
- $A = 100$
- $n_x = n_y = 1$
- $f = 215Hz$
- $\Phi(x) = 4/3\pi x$



A database of $N = 100$ snapshots is constructed using a time step $\Delta t = 2 \times 10^{-4} s$, leading to a Nyquist frequency $f_e = 1/2\Delta t = 2500Hz$ and a resolution $\Delta f = 1/N\Delta = 50Hz$.

Using the DMD algorithm, the spectrum and amplifications obtained are shown in the figures below:





The exact value of 215 Hz expected from the frequency peak is present on the amplitude spectrum. The amplification is very close to zero, which is expected since the wave is neither amplified nor attenuated.

Construction

```
import antares
myt = antares.Treatment('dmd')
```

Parameters

- **base: `Base`**
  The input base that contains one zone which contains one instant. The instant contains at least two 2-D arrays that represent the cell volumes and a scalar value.
- **time_step: *float***
  The time step between two instants Δt.
- **noiselevel: *float*, default= *1e-9***
  The noise level should be increased when ill-conditioned and non-physical modes appear.
- **type: *str*, default= *'mod/phi'***
  The decomposition type of the DMD:
  - *mod/phi* for modulus/phase decomposition or
  - *im/re* for imaginery/real part decomposition.
- **variables: *list(str)***
  The variable names. The variable representing the volume of each cell must be the first variable in the list of variables. The other variables are the DMD variables.
- **memory_mode: *bool*, default= *False***
  If True, the initial base is deleted on the fly to limit memory usage.
- **list_freq: *list(float)*, default= *[]***
  Frequency peaks to detect in Hertz. If no specific frequency is given, the DMD will be achieved on all modes (costly).
- **delta_freq: *float*, default= *0***
  The frequential resolution of the base, ie $\Delta f = 1/T$, with T the total time of the database ($T =$ number of instants $\times \Delta t$)

Preconditions

The input base must be an unstructured mesh.

The variable representing the volume of each cell must be the first variable in the list of variables. The other variables are the DMD variables.

Moreover, the frequential resolution of the base must be $\Delta f = 1/T$ in Hertz, with T the total time of the database ($T =$ number of instants $\times \Delta t$).

Postconditions

The treatment returns one output base containing:

- a zone named 'spectrum' with an instant named 'spectrum' with three variables 'frequency', 'amplification', and 'p_modulus' (magnitude)
- a zone named 'modes'. Each DMD variable modes will be stored in different instants. Each instant contains two variables depending on the **type** of DMD (e.g. 'p_mod' and 'p_phi')

The output zone 'modes' contains some keys 'peak_modes_<variable name>' in its `Zone.attrs`. These peak modes are the name of instants in the zone.

Each instant in the output zone 'modes' contains two keys 'frequency' and 'amplification' in its `Instant.attrs`.

References

P. J. Schmid: Dynamic mode decomposition of numerical and experimental data. J. Fluid Mech., vol. 656, no. July 2010, pp. 5–28, 2010.

Example

```
import antares
treatment = antares.Treatment('dmd')
treatment['time_step'] = 0.1
treatment['variables'] = ['volumes', 'pressure', 'vx']
treatment['noiselevel'] = 1e-4
treatment['base'] = base
treatment['list_freq'] = [300.0, 600.0]
treatment['delta_freq'] = 50.0
treatment['type'] = 'im/re'
result = treatment.execute()
```

Main functions

class antares.treatment.TreatmentDmd.**TreatmentDmd**

execute()

Execute the treatment.

Returns

a base that contains one zone with one instant. the instant has two variables.

Return type

Base

Example

```
"""
This example illustrates the Dynamic Mode Decomposition.
It is a stand alone example : the Base used for the DMD
is not read in a file, it is created in the example
"""
import os
import numpy as np
from antares import Base, Instant, Treatment, Writer, Zone


def myfunc(x, y, t, A, w, Phi):
    return A * np.sin(np.pi*x)*np.sin(np.pi*y)*np.cos(w*t+Phi*x)


if not os.path.isdir('OUTPUT'):
    os.makedirs('OUTPUT')

# ----------------------
# Create the input base
# ----------------------

# Initial parameters
time_step = 2e-4
nb_snapshot = 50
T = nb_snapshot*time_step
A1 = 100.0
A2 = 10.0
f1 = 215.0
f2 = 120.0
w1 = f1*2.0*np.pi
w2 = f2*2.0*np.pi
Phi = 4.0/3.0*np.pi

# Create a 2D mesh
n = 50
mgrid = np.mgrid[0:n, 0:n]
xt = (mgrid[0]/(n-1.))
yt = (mgrid[1]/(n-1.))
```

(continues on next page)

(continued from previous page)

```
# Initialise the base
base = Base()
base['zone_0'] = Zone()
base[0].shared['x'] = xt
base[0].shared['y'] = yt
base.unstructure()

# Define the cell volume
ncell = base[0].shared['x'].shape[0]
volume = np.ones((ncell))
base[0].shared['cell_volume'] = volume

# Extract coordinates and define time
x = base[0].shared['x']
y = base[0].shared['y']
time = np.linspace(0.0, T, nb_snapshot)

# Generate temporal database
for snapshot_idx in range(nb_snapshot):
    instant_name = 'snapshot_%s' % snapshot_idx
    base[0][instant_name] = Instant()
    base[0][instant_name]['pressure'] = myfunc(x, y, time[snapshot_idx], A1, w1, Phi)
    base[0][instant_name]['vx'] = myfunc(x, y, time[snapshot_idx], A2, w2, Phi)

# --------------
# DMD treatment
# --------------
treatment = Treatment('Dmd')
treatment['time_step'] = time_step
treatment['variables'] = ['cell_volume', 'pressure', 'vx']
treatment['noiselevel'] = 1e-9
treatment['base'] = base
treatment['list_freq'] = [f1, f2]
treatment['delta_freq'] = 50.0
treatment['type'] = 'im/re'
result = treatment.execute()

# -------------------
# Writing the result
# -------------------

# Write the spectrum (frequency, magnification and magnitude)
writer = Writer('column')
writer['filename'] = os.path.join('OUTPUT', 'ex_dmd_spectrum.dat')
writer['base'] = result[('spectrum', )]
writer.dump()

# Pick and write modes: here we retreive the peak modes of pressure for each variables
peak_modes = result['modes'].attrs['peak_modes_pressure']
newbase = result[('modes', ), peak_modes]
newbase[0].shared.connectivity = base[0][0].connectivity
```

(continues on next page)

(continued from previous page)

```
newbase[0].shared['x'] = base[0][0]['x']
newbase[0].shared['y'] = base[0][0]['y']
writer = Writer('bin_tp')
writer['filename'] = os.path.join('OUTPUT', 'ex_dmd_pressure_peak_mode_<instant>.plt')
writer['base'] = newbase
writer.dump()

# Pick and write modes: here we retreive the peak modes of velocity for each variables
peak_modes = result['modes'].attrs['peak_modes_vx']
newbase = result[('modes', ), peak_modes]
newbase[0].shared.connectivity = base[0][0].connectivity
newbase[0].shared['x'] = base[0][0]['x']
newbase[0].shared['y'] = base[0][0]['y']
writer = Writer('bin_tp')
writer['filename'] = os.path.join('OUTPUT', 'ex_dmd_vx_peak_mode_<instant>.plt')
writer['base'] = newbase
writer.dump()

# Write the effective frequencies of identified modes
fic = open(os.path.join('OUTPUT', 'ex_dmd_list_modes.dat'), 'w')
for key in result[('modes', ), peak_modes][0].keys():
    fic.write(key)
    fic.write('\t')
    fic.write(str(result[('modes', ), peak_modes][0][key].attrs['frequency']))
    fic.write('\n')
```

Signal Dynamic Mode Decomposition (1-D signals)

Description

This processing computes a Dynamic Mode Decomposition (DMD) of a 1D signal.

The DMD allows to characterize the evolution of a complex non-linear system by the evolution of a linear system of reduced dimension.

For the vector quantity of interest $v(x_n, t) \in \mathbb{R}^n$ (for example the velocity field at a node of a mesh), where t is the temporal variable and x_n the spatial variable. The DMD provides a decomposition of this quantity in modes, each having a complex angular frequency ω_i:

$v(x_n, t) \simeq \sum_{i=1}^{m} c_i \Phi_i(x_n) e^{j\omega_i t}$

or in discrete form:

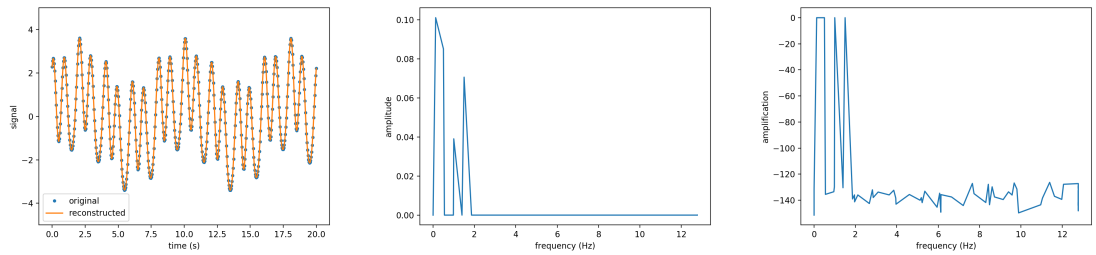
$v_k \simeq \sum_{i=1}^{m} \lambda_i^k c_i \Phi_i(x_n),$ with $\lambda_i^k = e^{j\omega_i \Delta t}$

with Φ_i the DMD modes.

Using the DMD, the frequencies ω_i are obtained with a least-square method.

Compared to a FFT, the DMD has thus the following advantages:

- Less spectral leaking
- Works even with very short signals (one or two periods of a signal can be enough)
- Takes advantage of the large amount of spatial information



Figures above show the results obtained from the signal $f(t)$ on the time interval $t \in [0, 20]$ discretized with $n = 512$ time steps:

$f(t) = 2cos(2\pi t) + 0.5sin(3\pi t) + cos(\pi t + 5) + sin(0.789t)$

From left to right, figures show the original and reconstructed signals (using 10 modes), the amplitude and the amplification. The amplitude exhibits four modes. Note that the amplification is close to zero as expected since the wave is neither amplified nor attenuated.

Construction

```
import antares
myt = antares.Treatment('dmd1d')
```

Parameters

- **base: `Base`**
  The input base that contains one zone which contains one instant. The instant contains at least two 1-D arrays that represent a time signal.
- **noiselevel: *float*, default= *1e-9***
  The noise level should be increased when ill-conditioned and non-physical modes appear.
- **reconstruction_modes_number: *int*, default= *-1***
  Number of modes used for the reconstruction (sorted by decreasing energy).
- **type: *str* in ['mod/phi', 'im/re'], default= *'mod/phi'***
  The decomposition type of the DMD:
  - *mod/phi* for modulus/phase decomposition or
  - *im/re* for imaginery/real part decomposition.
- **variables: *list(str)***
  The variable names. The variable representing time must be the first variable in the list of variables. The other variables are the DMD variables.
- **resize_time_factor: *float*, default= *1.0***
  Factor to re-dimensionalize the time variable (time used will be time_var * resize_time_factor). This is useful when you only have the iteration variable and when the time-step used is constant.
- **time_t0: *float*, default= *None***
  Time from which the analysis will begin.
- **time_tf: *float*, default= *None***
  Time to which the analysis will end.

- **complex:** ***bool*, default=** ***False***
  If True, the algorithm of the DMD uses complex arrays.

Preconditions

The variable representing time must be the first variable in the list of variables. The other variables are the DMD variables.

Postconditions

The treatment returns two output bases:

- the first base corresponds to the DMD (spectrum in the example below)
- the second base returns the input signal and the reconstructed signal for each variable.

Using the DMD algorithm, the amplitude $||\Phi||^2$ and the amplification $\mathbb{R}(\omega)$ of each mode are returned in the spectrum base, as well as the modulus and the phase (or the imaginary and real parts, depending on the **type** of DMD).

References

P. J. Schmid: Dynamic mode decomposition of numerical and experimental data. J. Fluid Mech., vol. 656, no. July 2010, pp. 5–28, 2010.

Example

```
import antares
myt = antares.Treatment('dmd1d')
myt['base'] = base
myt['variables'] = ['time', 'pressure', 'vx']
myt['noiselevel'] = 1e-4
myt['reconstruction_modes_number'] = 10
myt['resize_time_factor'] = 3.0E-7
myt['time_t0'] = 0.0294
myt['time_tf'] = 0.0318
spectrum, signal = myt.execute()
```

Main functions

class antares.treatment.TreatmentDmd1d.**TreatmentDmd1d**

execute()

Compute the Dynamic Mode Decomposition of the data.

Returns
The base with mode amplitude and amplification and the reconstructed signal.

Return type
Base

Example

```
"""
This example illustrates the Dynamic Mode Decomposition
of a 1D signal (massflow.dat).
"""
import os

import antares

if not os.path.isdir("OUTPUT"):
    os.makedirs("OUTPUT")

# ------------------
# Reading the files
# ------------------
reader = antares.Reader("column")
reader["filename"] = os.path.join("..", "data", "1D", "massflow.dat")
base = reader.read()

# -------
# DMD 1D
# -------
treatment = antares.Treatment("Dmd1d")
treatment["base"] = base
treatment["variables"] = ["iteration", "convflux_ro"]
treatment["noiselevel"] = 1e-4
treatment["reconstruction_modes_number"] = 10
treatment["resize_time_factor"] = 3.0e-7
treatment["time_t0"] = 0.0294
treatment["time_tf"] = 0.0318
spectrum, signal = treatment.execute()

# -------------------
# Writing the result
# -------------------
writer = antares.Writer("column")
writer["base"] = spectrum
writer["filename"] = os.path.join("OUTPUT", "ex_dmd1d_spectrum.dat")
writer.dump()

writer = antares.Writer("column")
writer["base"] = signal
writer["filename"] = os.path.join("OUTPUT", "ex_dmd_signal.dat")
writer.dump()
```

Proper Orthogonal Decomposition

Description

The POD treatment performs a *Proper Orthogonal Decomposition* (POD) for every zone within the base.

> **Warning:** A zone must contain all data to be decomposed.
>
> So here, a zone is not necessarily a part of a mesh (i.e.: from a structured multibloc mesh). If you want to apply the treatment on the whole mesh, then you have to merge all domains in one zone.

> **Warning:** The data contained in the zone will be concatenated to form a new vector. The treatment is performed on this vector. e.g.: if the instant contains 'p' and 't', the new vector will be ['p', 't']

Definition of the *POD*

The *Proper Orthogonal Decomposition* (POD) is a technique used to decompose a matrix and characterize it by its principal components which are called modes [*Chatterjee2000* (page 69)]. To approximate a function $z(x,t)$, only a finite sum is required:

$$z(x,t) \approx \sum_{k=1}^{m} a_k(t)\phi_k(x).$$

The basis function $\phi_k(x)$ can be chosen among Fourier series or Chebyshev polynomials, etc. For a chosen basis of functions, a set of unique time-functions $a_k(t)$ arises. In case of the POD, the basis function are chosen orthonormal, meaning that:

$$\int_x \phi_{k_1}\phi_{k_2}dx = \begin{cases} 1 & \text{if} \quad k_1 = k_2 \\ 0 & \text{if} \quad k_1 \neq k_2 \end{cases}, \ \alpha_k(t) = \int_x z(x,t)\phi_k(x)dx.$$

The principe of the POD is to chose $\phi_k(x)$ such that the approximation of $z(x,t)$ is the best in a least-squares sense. These orthonormal functions are called the *proper orthogonal modes* of the function.

When dealing with CFD simulations, the number of modes m is usually smaller than the number of measures (or snapshots) n. Hence, from the existing decomposition methods, the *Singular Value Decomposition* (SVD) is used. It is the snapshots methods [*Cordier2006* (page 69)].

The Singular Value Decomposition (SVD) is a factorization operation of a matrix expressed as:

$$A = U\Sigma V^T,$$

with V diagonalizes A^TA, U diagonalizes AA^T and Σ is the singular value matrix which diagonal is composed by the singular values of A. Knowing that a singular value is the square root of an eigenvector. u_i and v_i are eigenvectors of respectively U and V. form orthonormal basis. Thus, the initial matrix can be rewritten:

$$A = \sum_{i=1}^{r} \sigma_i u_i v_i^T,$$

r being the rank of the matrix. If taken $k < r$, an approximation of the initial matrix can be constructed. This allows to compress the data as only an extract of u and v need to be stored.

Parameters

- **base:** `Base`
  The input base that contains data.
- **tolerance:** *float*
  Tolerance for basis modes filtering [0->1].
- **dim_max:** *int*
  Maximum number of basis modes.
- **POD_vectors:** *bool*
  Output POD vectors as base attributes.
- **variables:** *list(str)*
  The variable names to take into account.

Preconditions

The input base must be an unstructured mesh.

Postconditions

A base with the same number and names of zones as the input base.

Each zone contains 2 or 4 instants depending on the value of the parameter **POD_vectors**.

The first instant contains the modes, the second the mean of the snapshots, the third the left-singular vectors, and the fourth the right-singular vectors.

References

A. Chatterjee: An introduction to the proper orthogonal decomposition. Current Science 78.7. 2000

L. Cordier: Reduction de dynamique par decomposition orthogonale aux valeurs propres (PDO). Ecole de printemps. 2006

Example

```
import antares
treatment = antares.Treatment('pod')
treatment['variables'] = ['pressure']
```

Main functions

class antares.treatment.TreatmentPOD.**TreatmentPOD**

execute()

Perform the POD treatment.

Returns

A base with the same number and the same names of zones as the input base.

Each zone contains 2 or 4 instants depending on the value of the parameter *POD_vectors*.

The first instant contains the modes, the second the mean of the snapshots, the third the left-singular vector, and the fourth the right-singular vectors.

Return type
Base

S

Singular values matrix, ndarray(nb of modes, nb of snapshots), only the diagonal is stored, of length (nb of modes).

U

Columns of matrix U are left-singular vectors of A, ndarray(nb of snapshots, nb of modes).

VT

Columns of matrix V are right-singular vectors of A, ndarray(nb of snapshots, nb of snapshots), after filtering (nb of snapshots, nb of modes).

Example

```
import os
if not os.path.isdir('OUTPUT'):
    os.makedirs('OUTPUT')


from antares import Reader, Treatment

# ------------------
# Read the files
# ------------------
r = Reader('fmt_tp')
r['filename'] = os.path.join('..', 'data', 'SNAPSHOTS_POD', '<instant>', 'pressure_<zone>
↪.dat')
base = r.read()

# ------------------
# View the data
# ------------------
print('Reading base: ', base)
print('Zone ex: ', base['ex'])
print('Instant ex -> 0: ', base['ex']['0'])
print('Value ex -> 0 -> p', base['ex']['0']['p'])
```

(continues on next page)

(continued from previous page)

```
# ------------------
# Setup
# ------------------
treatment = Treatment('POD')
treatment['base'] = base
treatment['tolerance'] = 0.99
treatment['dim_max'] = 100
treatment['POD_vectors'] = True
treatment['variables'] = ['p']

# ------------------
# Compute the POD
# ------------------
result = treatment.execute()

# ------------------
# Get some results
# ------------------
print("POD modes: ", result[0]['modes'][0])
print("POD modes: ", result[1]['modes'][0])
print("POD parameters: ")
for k, v in result.attrs.items():
    print(k, v)
# print("POD vectors: ", result.attrs['POD_vectors'])
```

Fast Fourier Transform (1-D signals)

Description

Computes a Fast Fourier Transform (FFT) of a signal.

Parameters

- **base: `Base`**
  The input base that contains many zones (independent to each others, typically many probes).
- **dtype_in: *str* in ['re', 'mod/phi', 'im/re'], default= *'re'***
  The decomposition type of the initial time signal: *re* if real signal, *mod/phi* or *im/re* if complex signal (modulus/phase or imaginery/real part decomposition respectively). If the signal is complex, a suffix must be added to the name of the variable depending on the decomposition (_im and _re for im/re, _mod and _phi for mod/phi). If given, the phase must be expressed in degrees.
- **dtype_out: *str*, default= *'mod/phi'***
  The decomposition type of the output signal: *mod/phi* or *im/re* (modulus/phase or imaginery/real part decomposition respectively). If given, the phase is expressed in degrees.
- **resize_time_factor: *float*, default= *1.0***
  Factor to re-dimensionalize the time variable (time used will be time_var * **resize_time_factor**). This is useful when you only have the iteration variables and when the time-step used is constant.
- **time_t0: *float*, default= *None***
  Time from which the analysis will start.

- **time_tf:** ***float*****, default=** ***None***
  Time to which the analysis will end.
- **zero_padding:** ***float*****, default=** ***1.0***
  Greater than 1.0 to use zero padding in the FFT computation.

Preconditions

Each zone contains one Instant object. Each instant contains at least:

- two 1-D arrays if the initial signal is real or
- three 1-D arrays (if it is complex).

The variable representing time must be the first variable in the instant. The second variable is the FFT variable if the signal is real. If the signal is complex, both the second and the third variables are used, and the **dtype_in** must be given to specify the decomposition (*mod/phi* or *im/re*). Other variables are ignored.

To change the ordering of variables, you may use base slicing[97].

Postconditions

The output base contains many zones. Each zone contains one instant. Each instant contains three 1-D arrays:

- The frequencies (variable named 'frequency')
- The FFT parts depending on the type of decomposition

Example

```
import antares
myt = antares.Treatment('fft')
myt['base'] = base
myt['dtype_out'] = 'mod/phi'
myt['resize_time_factor'] = 3.0E-7
myt['zero_padding'] = 4.0
myt['time_t0'] = 0.0294
myt['time_tf'] = 0.0318
fftbase = myt.execute()
```

Main functions

class antares.treatment.TreatmentFft.**TreatmentFft**

execute()

Execute the treatment.

Returns

a base that contains many zones. Each zone contains one instant. Each instant contains three 1-D arrays:

1. The frequencies (variable named 'frequency')

[97] https://cerfacs.fr/antares/src/tutorial/slicing.html

2. The FFT parts depending on the type of decomposition

Return type
`Base`

Example: real signal

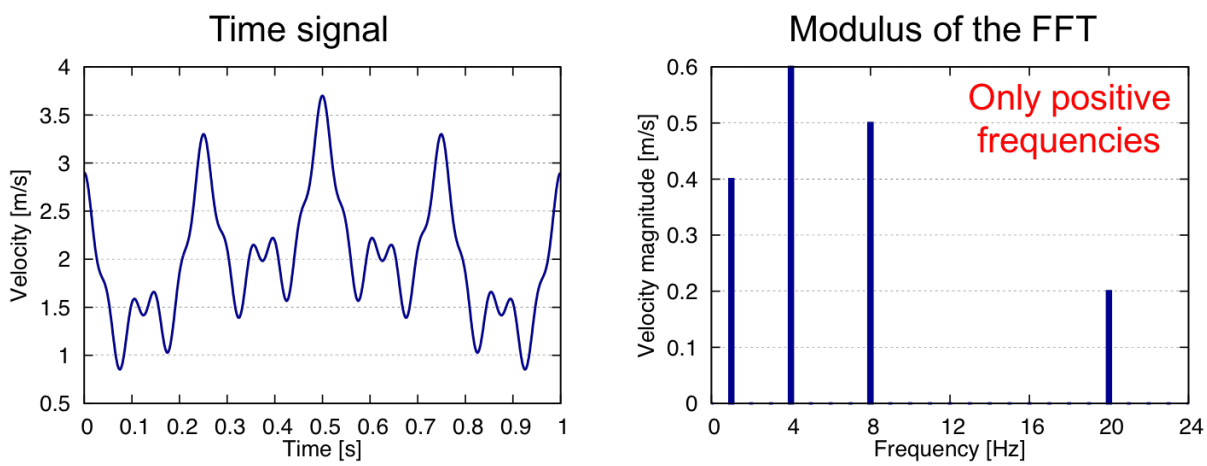
The signal is given by the following function:

$$u = a_0 + a_1 \cos(2\pi f_1 t) + a_2 \cos(2\pi f_2 t) + a_3 \cos(2\pi f_3 t) + a_4 \cos(2\pi f_4 t)$$

$$\left| \begin{array}{l} a_0 = 2 \\ a_1 = 0.6 \\ a_2 = 0.5 \\ a_3 = -0.4 \\ a_4 = 0.2 \end{array} \right. \quad \left| \begin{array}{l} f_1 = -4 \\ f_2 = 8 \\ f_3 = 1 \\ f_4 = -20 \end{array} \right.$$

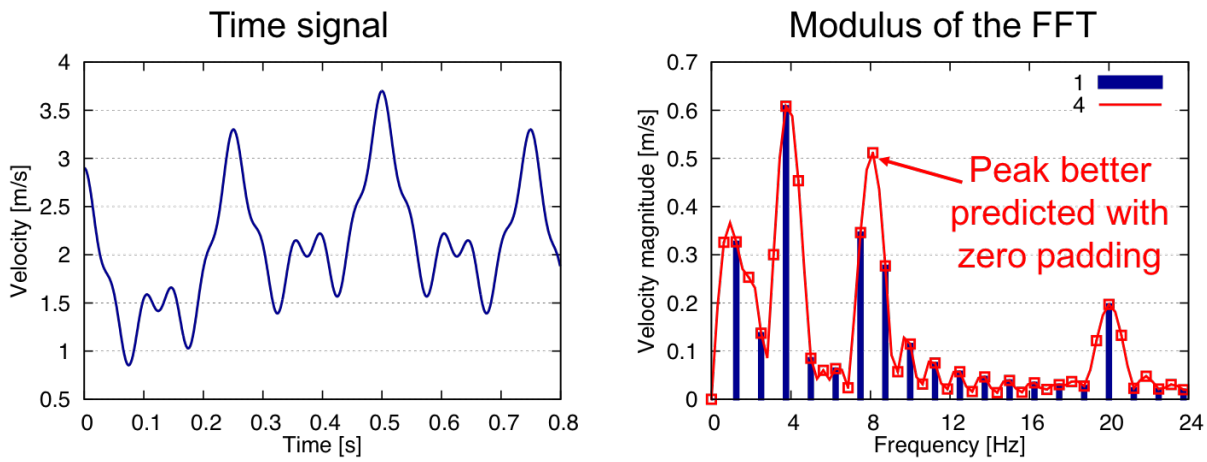
Periodic case

Since the signal is periodic with $t \in [0, 1]$, **zero padding** is not needed. The type of input data must be set to real (**'dtype_in' = 're'**). The base must contain two variables (t, u).



Non periodic case

Since the signal is not periodic with $t \in [0, 0.8]$, **zero padding** can be used to have a better prediction of the peaks. Example of results for **zero_padding = 1** (i.e. no zero padding) and **zero_padding = 4**.

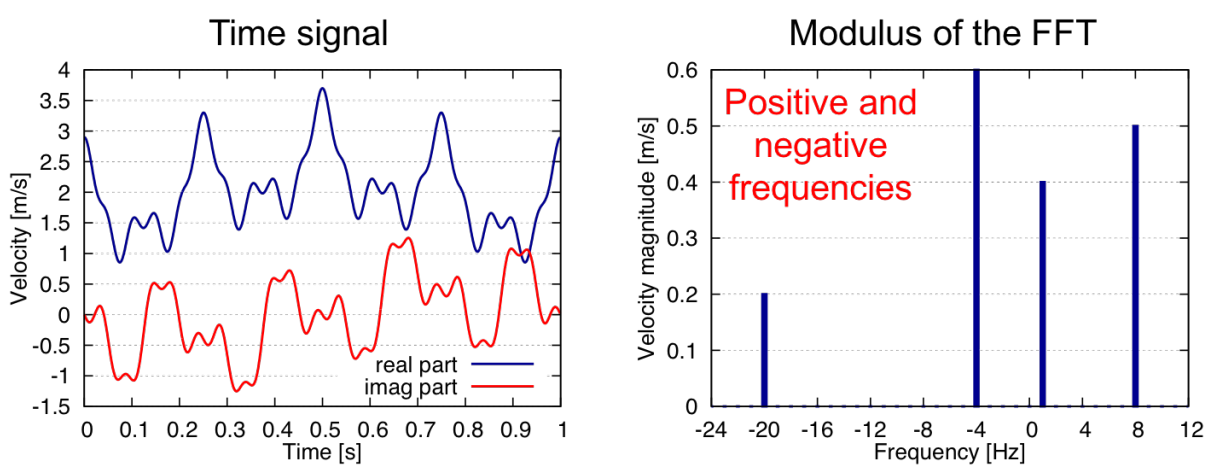


Example: complex signal

The signal is given by the following function:

$$u = a_0 + a_1 \exp(i2\pi f_1 t) + a_2 \exp(i2\pi f_2 t) + a_3 \exp(i2\pi f_3 t) + a_4 \exp(i2\pi f_4 t) \quad \left| \begin{array}{l} a_0 = 2 \\ a_1 = 0.6 \\ a_2 = 0.5 \\ a_3 = -0.4 \\ a_4 = 0.2 \end{array} \right. \quad \left| \begin{array}{l} f_1 = -4 \\ f_2 = 8 \\ f_3 = 1 \\ f_4 = -20 \end{array} \right.$$

Since the signal is periodic with $t \in [0, 1]$, **zero padding** is not needed. The type of input data must be set to complex (**'dtype_in' = 'im/re'** or **'mod/phi'**). The base must contain three variables (t, im(u), re(u)) or (t, mod(u), phi(u)) depending on the decomposition.



Example of Application Script

```
"""
This example illustrates the Fast Fourier Transform
treatment of Antares. The signal is analyzed on a time window
corresponding to [0.0294, 0.0318] and it is zero padded

WARNING, in this case, the file read contains only two
variables, iteration and convflux_ro, the treatment use
the first one as time variable and the second as fft variable.
In case of multiple variables, the first one must be the time variable
and the second one the fft variable. To do so, use base slicing.
"""
import os
if not os.path.isdir('OUTPUT'):
    os.makedirs('OUTPUT')

from antares import Reader, Treatment

# ------------------
# Reading the files
# ------------------
reader = Reader('column')
reader['filename'] = os.path.join('..', 'data', '1D', 'massflow.dat')
base = reader.read()

# ----
# FFT
# ----
treatment = Treatment('fft')
treatment['base'] = base
treatment['dtype_out'] = 'mod/phi'
treatment['resize_time_factor'] = 3.0E-7
treatment['zero_padding'] = 4.0
treatment['time_t0'] = 0.0294
treatment['time_tf'] = 0.0318
result = treatment.execute()

# the mean of the signal can be retrieved by calling
# 'mean' attr at instant level
print('signal temporal mean:', result[0][0].attrs['convflux_ro_mean'])

# each part of the decomposition is stored as a new variable
print('result variables:', list(result[0][0].keys()))
```

Signal Filtering (1-D signals)

Description

Applies a low-pass or high-pass filter to a time signal.

Parameters

- **base: `Base`**
  The input base that contains many zones (independent to each others, typically many probes).
- **type: *str* in ['low', 'high']**
  Type of filtering.
- **cutoff_frequency: *float***
  Cutoff frequency for the filter.

Preconditions

Each zone contains one instant object. Each instant contains at least two 1-D arrays that represent a time signal.

The variable representing time must be the first variable in the `Instant`. The second variable is the variable to filter. Other variables are ignored.

To change the ordering of variables, you may use base slicing[98].

Main functions

class `antares.treatment.TreatmentFilter.`**`TreatmentFilter`**

`execute``()`

Filter the time signal.

Returns

a base that contains many zones. Each zone contains one instant. Each instant contains two 1-D arrays:

1. The variable representing time
2. The filtered signal (real valued)

Return type
`Base`

[98] https://cerfacs.fr/antares/src/tutorial/slicing.html

Example

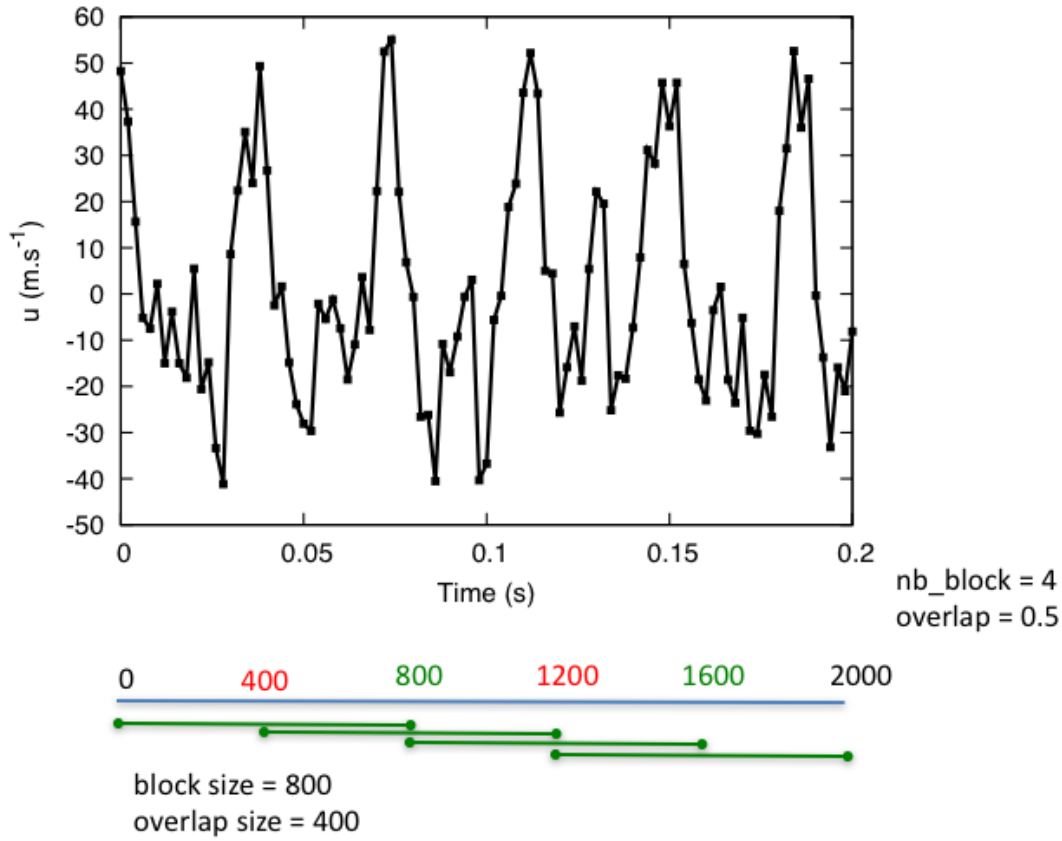
```
"""
This example how to generate a geometrical cut (of type plane).
To be able to use this functionnality you must have vtk installed.
"""
import os
if not os.path.isdir('OUTPUT'):
    os.makedirs('OUTPUT')

from antares import Reader, Treatment, Writer

# ------------------
# Reading the files
# ------------------
reader = Reader('column')
reader['filename'] = os.path.join('..', 'data', '1D', 'massflow.dat')
base = reader.read()

# -----------
# Filterring
# -----------
treatment = Treatment('filter')
treatment['base'] = base
treatment['cutoff_frequency'] = 2.0E-3
treatment['type'] = 'low'
result = treatment.execute()

# -------------------
# Writing the result
# -------------------
writer = Writer('column')
writer['filename'] = os.path.join('OUTPUT', 'ex_filter.dat')
writer['base'] = result
writer.dump()
```

Power Spectral Density (1-D signals)

Description

Computes the Power Spectral Density (PSD) of signals using Welch's average periodogram method[99]. The Welch's method splits the signal into different overlapping blocks. If needed, a window function might be applied to these blocks, in order to give more weight to the data at the center of the segments. We, then, compute the individual periodogram for each segment by calculating the squared magnitude of the discrete Fourier transform. Finally, the PSD is the average of the individual periodograms.

[99] https://en.wikipedia.org/wiki/Welch%27s_method



Note: Dependency on matplotlib.mlab[100]

Parameters

- **base: `Base`**
  The input base that contains many zones (independent to each others, typically many probes).
- **resize_time_factor: *float*, default= *1.0***
  Factor to re-dimensionalize the time variable (time used will be time_var * **resize_time_factor**). This is useful when you only have the iteration variables and when the time-step used is constant.
- **variables: *list(str)***
  The variables that serve as a basis for the PSD. The first variable must be the variable representing time.
- **time_t0: *float*, default= *None***
  Time from which the analysis will start.
- **time_tf: *float*, default= *None***
  Time at which the analysis will end.
- **nb_block: *int or float*, default= *1***
  Number of Welch's blocks.

[100] https://matplotlib.org/api/mlab_api.html#matplotlib.mlab.psd

- **pad:** ***int or float*, default=** ***1***
  Padding of the signal with zeros. **pad** =1 means that there is no padding. Otherwise, if the length of the signal is **Ls**, then the length of the padded signal will be **Ls*pad**.
- **overlap:** ***int or float*, default=** ***0.33***
  The percent of the length of the signal to be overlapped.
- **window:** ***str or function*, default=** ***'hanning'***
  The window function to be applied to each segment. The possible values are: 'hanning', 'none', 'blackman', 'hamming', 'bartlett'.

  If none of the above suits you, you can also set your own custom window function. The window function must be a function that accepts a numpy array representing the signal and it returns a single numpy array representing the signal multiplied by the window.

  - **Example** : The following code defines a triangular window using the function `triang` in the `scipy.signal` library

```
import scipy.signal

def my_triangular_window(signal):
    return signal * scipy.signal.triang(len(signal))

tt = antares.Treatment('psd')
tt['window'] = my_triangular_window
```

- **scale_by_freq:** ***bool or None*, default=** ***None***
  If the computed PSD should be scaled by the sampling frequency. If *True* the unit of the results will be $\propto \mathrm{Hz}^{-1}$.

Preconditions

Each zone contains one instant object. Each instant contains at least one 1-D array that represents a time signal.

To change the ordering of variables, you may use base slicing[101].

Postconditions

The output base contains as many zones as the input base. Each instant contains the following 1-D arrays:

- The frequencies (variable named 'frequency').
- The PSD values for each one of the variables.

[101] https://cerfacs.fr/antares/src/tutorial/slicing.html

Main functions

class antares.treatment.TreatmentPsd.**TreatmentPsd**

execute()

Compute the Power Spectral Density.

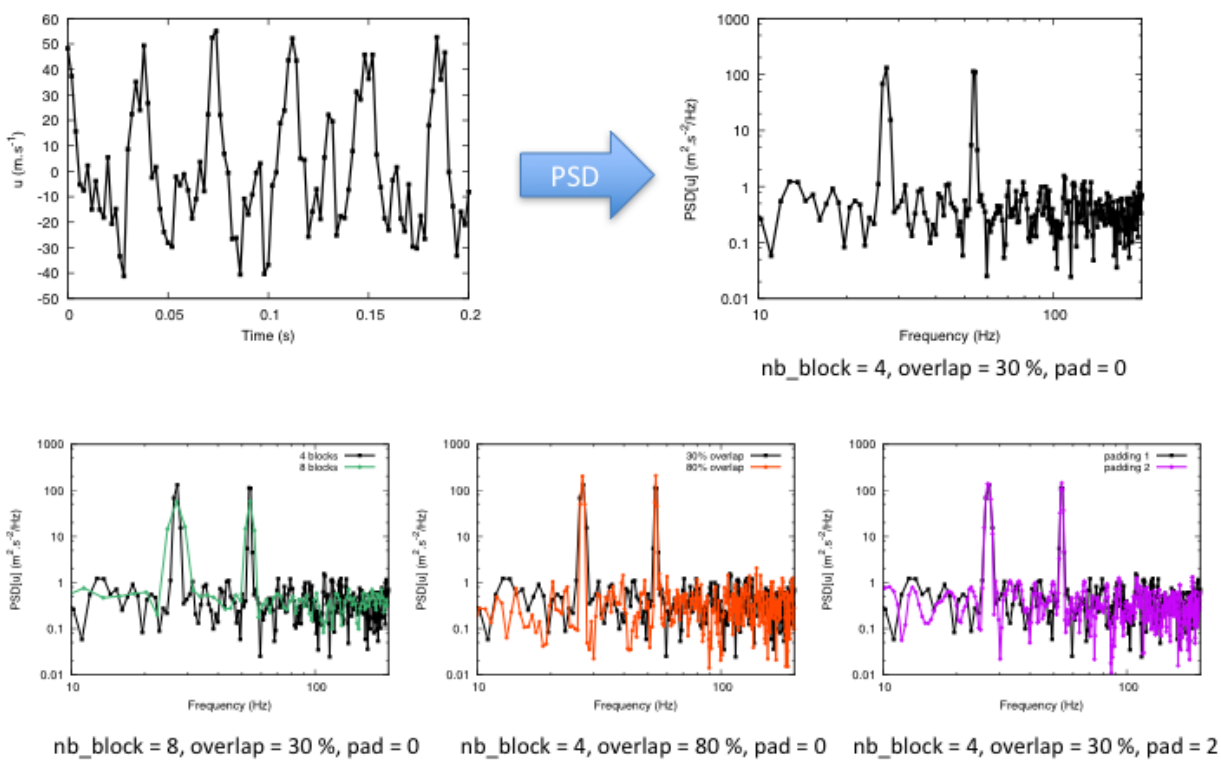
Returns

a base that contains many zones. Each zone contains one instant. Each instant contains two 1-D arrays:

1. The values for the power spectrum (real valued)
2. The frequencies corresponding to the elements in the power spectrum (real valued) (variable 'frequency')

Return type
Base

Example: impact of options on results



Example

```
"""
This example illustrates the Power Spectral Density
treatment of Antares. The signal is analyzed on a time window
corresponding to [0.0294, 0.0318].

WARNING, in this case, the file read contains only two
variables, iteration and convflux_ro, the treatment use
the first one as time variable and the second as psd variable.
In case of multiple variables, the first one must be the time variable
and the second one the psd variable. To do so, use base slicing.
"""
import os
import antares

# ------------------
# Reading the files
# ------------------
reader = antares.Reader('column')
reader['filename'] = os.path.join('..', 'data', '1D', 'massflow.dat')
base = reader.read()

# ----
# PSD
# ----
treatment = antares.Treatment('psd')
treatment['base'] = base
treatment['resize_time_factor'] = 3.0E-7
treatment['time_t0'] = 0.0294
treatment['time_tf'] = 0.0318
treatment['variables'] = ['iteration', 'convflux_ro']
result = treatment.execute()

# -------------------
# Plot the result
# -------------------
plot = antares.Treatment('plot')
plot['base'] = result
plot.execute()

# prompt the user
input('press enter to quit')
```

Spectral Proper Orthogonal Decomposition

Table of Contents

Description

This treatment computes the Spectral Proper Orthogonal Decomposition (SPOD) for a mono-zone base and for all or a subset of variables present in the base. The implementation is based on the description provided by *Schmidt et al. (2020)* (page 92).

The SPOD computes the spatio-temporal modes of the flow. These modes are calculated as the eigenvectors of the cross-spectral density (CSD) matrix. The SPOD relies heavily on Welch's method in which the temporal data is split into several overlapping blocks and then the DFT is computed on each block. Then, the CSD matrix for a given frequency is estimated from the frequency data of each of the DFT blocks. Finally, the modes $\hat{\Phi}$ at each frequency corresponds to the eigenvectors of the estimation of the CSD matrix $\hat{C}$.

$$\hat{C}W\hat{\Phi} = \hat{\Phi}\hat{\Lambda}$$

Where W is a weight matrix that can be used to take into consideration the mesh refinement or the difference in the fluctuation magnitude of the different variables, and $\hat{\Lambda}$ are the eigenvalues containing the energy of each mode.

Parameters

- **base: `Base`, default=** ***None***
  The input base with which the SPOD is computed. The base must contain only one zone.
- **dt:** ***float*** **, default=** ***None***
  Time step between two instants.
- **variables:** ***list(str)*** **, default=** ***None***
  The variables with which the SPOD is computed. If `None`, then the SPOD will be computed on all variables present in the base.

- **volume:** ***str*****, default=** ***None***
  The name of the variable containing the cell volume. If `None`, then the volume is computed from the base.
- **nb_modes:** ***int*** **, default=** ***None***
  In *basic mode* (page 85): the desired number of modes per frequency.
- **nb_frequencies:** ***int*** **, default=** ***None***
  In *basic mode* (page 85): the desired number of frequencies.
- **overlap_ratio:** ***float*** **, default=** ***0.5***
  In *basic mode* (page 85): the overlapping ratio between two DFT blocks.
- **nb_instants_per_block:** ***int*****, default=** ***None***
  In *advanced mode* (page 85): the number of instants per DFT block.
- **nb_overlap:** ***int*****, default=** ***None***
  In *advanced mode* (page 85): the number of overlapping instants in the DFT blocks.
- **nb_zero_padding:** ***int*****, default=** ***None***
  In *advanced mode* (page 85): the number of zeros added to the DFT blocks.
- **norm:** ***str*** **, default=** ***'identity'***
  The normalization matrix W:
  - `'identity'`: no normalization ($W = I$).
  - `'volume'`: normalize by the cell volume.
  - `'turbulent_kinetic_energy'`: normalize by the turbulent kinetic energy (variables must be the velocity vector components).
  - `'total_energy_ruvwt'`: normalize by the total energy when the variables are density, velocity and temperature.
  - `'total_energy_puvws'`: normalize by the total energy when the variables are pressure, velocity and entropy.
  - `'acoustic_energy'`: normalize by the acoustic energy when the variables are density and velocity.

  > **Warning:** Some normalization functions require a specific order in the **variables** keywords and/or the definition of **extra** parameters. See the *normalization* (page 86) section for further details on how to use the different options available.

- **window_type:** ***str*** **or** ***callable*** **, default=** ***'hamming'***
  The window used in the DFT. The user can pick between the following predefined windows: `'uniform'`, `'hanning'`, `'blackman'`, `'hamming'`, and `'bartlett'`.

  If none of the above suits you, you can also set your own custom window function. The window function must be a function that accepts an integer N representing the length of the unpadded block signal, and returns a single array of length N with the window data.

  Example : The following code defines a triangular window using the function `triang` in the `scipy.signal` library

```
import scipy.signal

def my_triangular_window(N):
    return scipy.signal.triang(N)
```

(continues on next page)

(continued from previous page)

```
tt = antares.Treatment('spod')
tt['window_type'] = my_triangular_window
```

- **confidence_lvl:** ***float*** **, default=** ***None***
  The confidence interval for the energy modes.
- **mean_type:** ***str*** **, default=** ***'long-term'***
  The type of mean subtracted to the DFT blocks:
  - `'long-term'`: subtract the mean of the entire base.
  - `'blockwise'`: subtract only the mean of the corresponding block.
- **rho:** ***str*** **or** ***float*** **, default=** ***None***
  The string containing the name of the density variable or a *float* value if it is assumed to be constant.
- **gamma:** ***str or float*** **, default=** ***None***
  The string containing the name of the specific heat ratio capacity variable or a *float* value if it is assumed to be constant.
- **R:** ***str*** **or** ***float*** **, default=** ***None***
  The string containing the name of the mass specific gas constant variable or a *float* value if it is assumed to be constant.
- **speed_of_sound:** ***str*** **or** ***float*****, default=** ***None***
  The string containing the name of the speed of sound variable or a *float* value if it is assumed to be constant.
- **output_modes:** ***bool*** **, default=** ***False***
  If `True`, the treatment will output 2 bases, the first will contain the energy of the modes and the second will contain the spatial modes. If `False`, only the modes energy is output.
- **large_data_directory:** ***str*** **, default=** ***None***
  If `not None`, intermediary results will be saved in this directory to decrease the memory footprint.
- **filter_output_frequencies:** ***list(list or float)*****, default=** ***None***

  If `None`, compute the SPOD modes for all frequencies.
  Otherwise, compute the SPOD modes only for specific frequencies. The syntax is a list whose elements can be:
  - A single number n: Compute the modes for the nearest frequency to n
  - A list of the form `[a, b]`: Compute the modes for all frequencies f in the range $a \leq f \leq b$

  Example 2 (page 89) shows the usage of this keyword.
- **frequencies_as_instants:** ***bool*****, default=** ***False***

  If `True`, the output modes base will be formatted as `base[variable][frequency][mode]`.
  If `False`, the output modes base will be formatted as `base[variable][mode][frequency]`.
- **compute:** ***str*** **, default=** ***'full'***
  - `'full'`: compute both the DFT and SPOD
  - `'dft'`: compute only the DFT part and save the result in **large_data_directory**
  - `'spod'`: read the DFT results in **large_data_directory** and compute the SPOD part

Preconditions

The following conditions must be met in order to ensure a correct execution of the treatment:

- The input base must be unstructured
- The input base must contain only one zone.
- The input base must be either 2D or 3D dimensional.
- All the instants must contain the variables specified in the **variables** keyword and any other extra variable related to the normalization

Postconditions

If **output_modes** = `False`, the treatment outputs one base containing the energy of each mode. This base contains one zone, and 1 or 3 instants called 'modes', 'confidence_lower', and 'confidence_upper'. The last two instants are present only if **confidence_lvl** is not `None`. Each of these instants contains a variable called 'frequency' with all the frequencies computed and several instants called 'mode_xxx' containing the energy per frequency of each mode, and the lower or upper end of the confidence intervals. The modes are sorted such as 'mode_000' is the most energetic mode, 'mode_001' is the second most energetic mode, and so on.

If **output_modes** = `True`, the treatment outputs a list of two bases. The first base is the energy base described above. The second base contains the SPOD modes. This base has as many zones as variables on which the SPOD was computed (similarly to the previous base, the modes are sorted by energy). The instants in each zone will be formatted according the **frequencies_as_instants** keyword.

Basic and advanced modes

This treatment can be used in "basic" or "advanced" mode. In the "basic" mode, the user specifies the number of desired modes and frequencies in the output, as well as the overlap ratio between the DFT blocks. The treatment will internally compute how the database should be divided and zero-padded to satisfy these conditions. In the "advanced" mode, the user selects the number of instant per DFT blocks, the number of overlapping instants between the block and the number of zeros padded to each DFT block. The number of modes and frequencies will, therefore, be determined by these choices. The keywords to specify for each mode are the following:

- Basic mode keywords:
  - **nb_modes**
  - **nb_frequencies**
  - **overlap_ratio** (optional)
- Advanced mode keywords:
  - **nb_instants_per_block**
  - **nb_overlap**
  - **nb_zero_padding**

> **Warning:** Only one **set** of keys should be specified for the treatment to work. Simultaneously setting "basic" and "advanced" keywords will lead to an error when executing the treatment.

Normalizations

- **'identity'**

$$W = I$$

- **'volume'**

$$W = \int_V dV$$

- **'turbulent_kinetic_energy'**

If **variables** is set to the velocity components u, v, w, then we can normalize by the turbulent kinetic energy (TKE). In this case, the weight matrix is given by:

$$W = \int_V \begin{bmatrix} 1 & & \\ & 1 & \\ & & 1 \end{bmatrix} dV \tag{3.1}$$

- **'total_energy_ruvwt'**

If **variables** is set to the density, velocity components and temperature (in that order), then we can normalize by the total energy. In this case, the weight matrix is given by:

$$W = \int_V \begin{bmatrix} \frac{\overline{a_0}^2}{\overline{\gamma}\,\overline{\rho}} & & & & \\ & \overline{\rho} & & & \\ & & \overline{\rho} & & \\ & & & \overline{\rho} & \\ & & & & \frac{\overline{\rho}\overline{R}}{(\overline{\gamma}-1)\overline{T}} \end{bmatrix} dV \tag{3.2}$$

In order to use this normalization the following keywords must be specified: **gamma**, **R**, and **speed_of_sound**

- **'total_energy_puvws'**

If **variables** is set to the pressure, velocity components and entropy (in that order), then we can normalize by the total energy. In this case, the weight matrix is given by:

$$W = \int_V \begin{bmatrix} \frac{\overline{\rho}\,\overline{a_0}^2}{\overline{\gamma}^2\,\overline{P}^2} & & & & \\ & \overline{\rho} & & & \\ & & \overline{\rho} & & \\ & & & \overline{\rho} & \\ & & & & \frac{(\overline{\gamma}-1)\overline{P}}{\overline{\gamma}\overline{R}^2} \end{bmatrix} dV \tag{3.3}$$

In order to use this normalization the following keywords must be specified: **gamma**, **R**, **speed_of_sound**, and **rho**

- **'acoustic_energy'**

If *variables* is set to the density, velocity components and temperature (in that order), then we can normalize by the total energy. In this case, the weight matrix is given by:

$$W = \int_V \begin{bmatrix} \frac{\overline{a_0}^2}{\overline{\gamma}\,\overline{\rho}} & & & \\ & \overline{\rho} & & \\ & & \overline{\rho} & \\ & & & \overline{\rho} \end{bmatrix} dV \tag{3.4}$$

In order to use this normalization the following keywords must be specified: **gamma**, and **speed_of_sound**

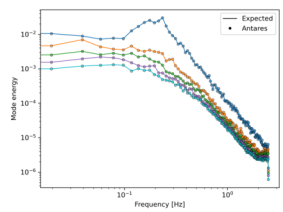
Validation

This treatment was validated comparing the results to 4 different publicly available SPOD implementations, one done in MATLAB[102] and another in Python[103].

These test cases correspond to:

- 2D subsonic jet flow (*Brès et al. 2018* (page 92))
- Flow over 2D Backward facing step (*He et al. 2022* (page 92))
- Pressure field over a compressor blade (*He et al. 2022* (page 92))
- 2D turbine cooling film (*Wang et al. 2021* (page 92))

As an example, we show below the comparison between the first 5 expected modes energy and the energy obtained using the treatment for the first test case:



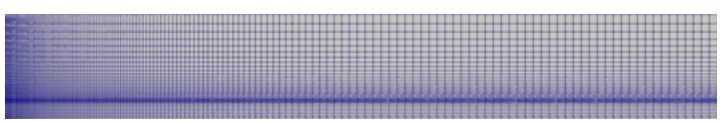
Main functions

class antares.treatment.TreatmentSPOD.**TreatmentSPOD**

execute()

Example 1: SPOD treatment in basic mode without normalization

The following example computes the SPOD on a 2D jet flow example. The data corresponds to LES simulation of a subsonic jet at $Ma = 0.9$ (*Brès et al. 2018* (page 92)). The original database can be found here[104]. The simulation domain is a 2D rectangle with a structured mesh with a refinement in the jet zone. The database is composed of the pressure field for 5000 timesteps ($\Delta t = 0.2$). The mesh and an animation of the pressure profile is shown below.



The script below computes 38 SPOD modes with 128 frequencies per mode for this configuration. The DFT blocks are split so there is a 50% overlap, as we use the hamming window function. For this case, no normalization is used the 95% confidence intervals are calculated for the each mode energy. We, additionally, show a post-processing example to plot the energy of the 3 most energetic modes along with their confidence intervals. Finally, we output the modes for future visualization.

[102] https://github.com/SpectralPOD/spod_matlab
[103] https://github.com/HexFluid/spod_python
[104] https://github.com/SpectralPOD/spod_matlab/tree/master/jet_data

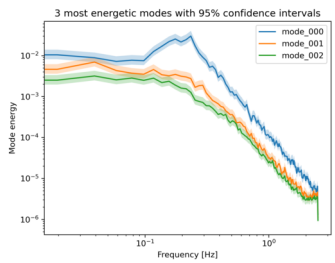
```
"""
Example of SPOD treatment in basic mode

to run:
python spod_basic.py
"""

import os
import antares
import matplotlib.pyplot as plt

data_folder = os.path.join('..', 'data', 'SPOD')
output_folder = os.path.join(data_folder, 'OUTPUT')
output_modes_folder = os.path.join(output_folder, 'modes')
os.makedirs(output_folder, exist_ok=True)
os.makedirs(output_modes_folder, exist_ok=True)

# Read 2D jet data
# ----------------
reader = antares.Reader('bin_tp')
reader['filename'] = os.path.join(data_folder, '2D_jet', 'jet<instant>.plt')
jet_data = reader.read()

# Check if the base is structured:
if jet_data.is_structured():
    jet_data.unstructure()

# Apply SPOD treatment
# --------------------
spod = antares.Treatment('spod')
spod['base'] = jet_data
spod['variables'] = ['pressure']
spod['nb_modes'] = 38
spod['nb_frequencies'] = 128
spod['overlap_ratio'] = 0.5
spod['dt'] = 0.2
spod['window_type'] = 'hamming'
spod['norm'] = 'identity'
spod['confidence_lvl'] = 0.95
spod['output_modes'] = True
spod['frequencies_as_instants'] = True
energy_base, spod_modes = spod.execute()

# Plot the energy for the 3 most energetic modes
# ----------------------------------------------

for idx in range(3):
    freq = energy_base[0]['modes']['frequency']
    energy = energy_base[0]['modes'][f'mode_{idx:03d}']
    conf_lower = energy_base[0]['confidence_lower'][f'mode_{idx:03d}']
    conf_upper = energy_base[0]['confidence_upper'][f'mode_{idx:03d}']

    plt.loglog(freq, energy, label = f'mode_{idx:03d}')
```

(continues on next page)

(continued from previous page)

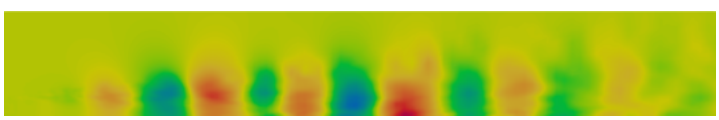
```
    plt.fill_between(freq, conf_lower, conf_upper, alpha=0.25)

plt.xlabel('Frequency [Hz]')
plt.ylabel('Mode energy')
plt.title('3 most energetic modes with 95% confidence intervals')
plt.legend()
plt.savefig(os.path.join(output_folder, 'spod_jet_mode_energies.png'), dpi=300)

# Save the spatial modes for visualization
# ----------------------------------------

writer = antares.Writer('hdf_antares')
writer['base'] = spod_modes
writer['filename'] = os.path.join(output_modes_folder, 'modes')
writer.dump()
```

The 3 most energetic modes with 95% confidence are shown below:



Example of visualization of two of the modes at two different frequencies:

- First mode at $f = 0.0585$ Hz



- Fifth mode at $f = 0.683$ Hz

Example 2: SPOD Treatment in advanced mode with total energy normalization

The following example computes the SPOD on a 2D film cooling flow DDES simulation (*Wang et al. 2021* (page 92)). The simulation domain is a 2D slice of the mixing between the coolant and the hot stream. The database contains the density, velocity, temperature and the speed of sound.

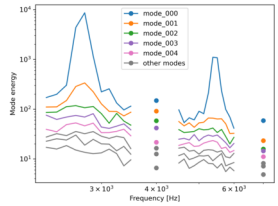
The script below shows how to use the advanced mode of the SPOD treatment. The database is divided into blocks of 128 points, with a 64 points overlap between blocks and no zero padding. The SPOD is normalized using the total energy ('total_energy_ruvwt'). Additionally, we only compute the SPOD modes for the frequencies between 2200Hz-3500Hz, 4500Hz-6000Hz, and for the frequency closest to 4010Hz and 7000Hz. Finally, the energy of all modes for the aforementioned frequency ranges is plotted in a single figure.

```
"""
Example of SPOD treatment in advanced mode

to run:
python spod_advanced.py
"""

import os
import antares
import matplotlib.pyplot as plt
import numpy as np

data_folder = os.path.join('..', 'data', 'SPOD')
output_folder = os.path.join(data_folder, 'OUTPUT')
output_modes_folder = os.path.join(output_folder, 'modes')
os.makedirs(output_folder, exist_ok=True)
os.makedirs(output_modes_folder, exist_ok=True)

# Read data
# ----------------
reader = antares.Reader('bin_tp')
reader['filename'] = os.path.join(data_folder, 'Turbine_film_cooling', 'cooling_<instant>
↪.plt')
data = reader.read()

# Check if the base is structured:
if data.is_structured():
    data.unstructure()

## Setup and run treatment
spod = antares.Treatment('spod')
spod['base'] = data
spod['variables'] = ['rho', 'u', 'v', 'T']
spod['nb_instants_per_block'] = 128
spod['nb_overlap'] = 64
spod['nb_zero_padding'] = 0
spod['dt'] = 6.25e-5
spod['window_type'] = 'hamming'
spod['norm'] = 'total_energy_ruvwt'
spod['gamma'] = 1.4
spod['R'] = 287
spod['speed_of_sound'] = 'speed_of_sound'
spod['output_modes'] = False
spod['filter_output_frequencies'] = [[2200, 3500], [4500, 6000], 4010, 7000]
spod['volume'] = 'volume'
energy_base = spod.execute()

# Plot the 5 most energetic modes with different colors
legend_lines = []
colored_modes = 5
mode_colors = ['tab:blue', 'tab:orange', 'tab:green', 'tab:purple', 'tab:pink']
for idx in range(colored_modes):
    for rdx, f in enumerate(spod['filter_output_frequencies']):
```

(continues on next page)

(continued from previous page)

```
        if type(f) is list:
            mask = (f[0] <= energy_base[0]['modes']['frequency']) & \
                   (energy_base[0]['modes']['frequency'] <= f[1])
        else:
            f_idx = (np.abs(f - energy_base[0]['modes']['frequency'])).argmin()
            mask = [False for _ in range(len(energy_base[0]['modes']['frequency']))]
            mask[f_idx] = True

        freq = energy_base[0]['modes']['frequency'][mask]
        energy = energy_base[0]['modes'][f'mode_{idx:03d}'][mask]
        if sum(mask) == 1:
            l = plt.loglog(freq, energy, label = f'mode_{idx:03d}', color=mode_
↪colors[idx], linestyle='-', marker='o')
        else:
            l = plt.loglog(freq, energy, label = f'mode_{idx:03d}', color=mode_
↪colors[idx], linestyle='-', marker='')
    legend_lines.append(l[0])


# Plot the remaining modes in grey
for idx in range(colored_modes, len(energy_base[0][0].keys())-1):
    for rdx, f in enumerate(spod['filter_output_frequencies']):
        if type(f) is list:
            mask = (f[0] <= energy_base[0]['modes']['frequency']) & \
                   (energy_base[0]['modes']['frequency'] <= f[1])
        else:
            f_idx = (np.abs(f - energy_base[0]['modes']['frequency'])).argmin()
            mask = [False for _ in range(len(energy_base[0]['modes']['frequency']))]
            mask[f_idx] = True

        freq = energy_base[0]['modes']['frequency'][mask]
        energy = energy_base[0]['modes'][f'mode_{idx:03d}'][mask]
        if sum(mask) == 1:
            l = plt.loglog(freq, energy, label = f'other modes', color='tab:grey',␣
↪linestyle='-', marker='o')
        else:
            l = plt.loglog(freq, energy, label = f'other modes', color='tab:grey',␣
↪linestyle='-', marker='')

legend_lines.append(l[0])

plt.xlabel('Frequency [Hz]')
plt.ylabel('Mode energy')
plt.legend(handles=legend_lines)
plt.savefig(os.path.join(output_folder, 'spod_cooling_mode_energies.png'), dpi=300)
```



References

G. A. Brès, P. Jordan, M. Le Rallic, V. Jaunet, A. V. G. Cavalieri, A. Towne, S. K. Lele, T. Colonius, O. T. Schmidt, (2018). Importance of the nozzle-exit boundary-layer state in subsonic turbulent jets, Journal of Fluid Mechanics 851, 83-124. (link[105]).

He, X., Zhao, F., and Vahdati, M. (2022). Detached Eddy Simulation: Recent Development and Application to Compressor Tip Leakage Flow. ASME. Journal of Turbomachinery 144(1): 011009. (link[106]).

Schmidt, O. T., Towne, A., Rigas, G., Colonius, T., & Bres, G. A. (2018). Spectral analysis of jet turbulence. Journal of Fluid Mechanics, 855, 953-982. (link[107]).

Schmidt, O. T., & Colonius, T. (2020). Guide to spectral proper orthogonal decomposition. AIAA journal, 58(3), 1023-1033. (link[108]).

Sieber, M., Paschereit, C. O., & Oberleithner, K. (2016). Spectral proper orthogonal decomposition. Journal of Fluid Mechanics, 792, 798-828. (link[109])

Wang, R., & Yan, X. (2021). Delayed-detached eddy simulations of film cooling effect on trailing edge cutback with land extensions. ASME Journal of Engineering for Gas Turbines and Power, 143(11), 111004. (link[110])

Link to MATLAB implementation repository[111]

Link to Python implementation repository[112]

Spectrogram

Table of Contents

[105] https://doi.org/10.1017/jfm.2018.476
[106] https://doi.org/10.1115/1.4052019
[107] https://doi.org/10.1017/jfm.2018.675
[108] https://doi.org/10.2514/1.J058809
[109] https://doi.org/10.1017/jfm.2016.103
[110] https://doi.org/10.1115/1.4051865
[111] https://github.com/SpectralPOD/spod_matlab
[112] https://github.com/HexFluid/spod_python

Description

Computes the Spectrogram of 1D signals.

The spectrogram of signal is computed by dividing the signal into N_b overlapping blocks of the same length and performing the Fourier transform on each block. This allows us to study the changes of the spectra as a function of time.

Note: Dependency on matplotlib.mlab[113]

Parameters

- **base:** `Base`
  The input base. The base can contain many zones, typically, many probes. Each zone must contain the variable(s) on which to compute the spectrogram and the time vector.
- **resize_time_factor:** ***float*, default= *1.0***
  Factor to re-dimensionalize the time variable (time used will be time_var * **resize_time_factor**). This is useful when you only have the iteration variables and when the time-step used is constant.
- **variables:** ***list(str)***
  The variables that serve as a basis for the spectrogram. The first variable must be the variable representing time.
- **time_t0:** ***float*, default= *None***
  Time from which the analysis will start. If `None`, the analysis will start from the beginning of the time array.
- **time_tf:** ***float*, default= *None***
  Time at which the analysis will end. If :python`None`, the analysis will go until the end of the time array.
- **nb_block:** ***int or float*, default= *1***
  Number of blocks
- **pad:** ***int or float*, default= *1***
  Padding of the signal with zeros. **pad** =1 means that there is no padding. Otherwise, if the length of the signal is **Ls**, then the length of the padded signal will be **Ls*pad**.
- **overlap:** ***int or float*, default= *0.33***
  The percent of the length of the signal to be overlapped.
- **window:** ***str or function*, default= *'hanning'***
  The window function to be applied to each segment. The possible values are: 'hanning', 'none', 'blackman', 'hamming', 'bartlett'.

  If none of the above suits you, you can also set your own custom window function. The window function must be a function that accepts a numpy array representing the signal and it returns a single numpy array representing the windowed signal.

  - **Example** : The following code defines a triangular window using the function `triang` in the `scipy.signal` library

```
import scipy.signal
```

(continues on next page)

[113] https://matplotlib.org/api/mlab_api.html#matplotlib.mlab.psd

(continued from previous page)

```
def my_triangular_window(signal):
    return signal * scipy.signal.triang(len(signal))

tt = antares.Treatment('specgram')
tt['window'] = my_triangular_window
```

- **scale_by_freq:** ***bool or None*****, default=** ***None***
  If the computed spectrogram should be scaled by the sampling frequency. If *True* the unit of the results will be $\propto \mathrm{Hz}^{-1}$.
- **t_and_f_in_attrs:** ***bool*****, defaut=** ***True***
  If `True`, the time and frequency vectors will be saved as 1D vectors in the attributes of the instant. If `False`, they will be saved as 2D matrices resulting from the `numpy.meshgrid(time, freq)`.
- **mode:** ***str*****, default=** ***'psd'***
  What sort of spectrum to use:
  - `'psd'` returns the power spectral density
  - `'complex'` returns the complex-valued frequency spectrum
  - `'magnitude'` returns the magnitude spectrum
  - `'angle'` returns the phase spectrum without unwrapping
  - `'phase'` returns the phase spectrum with unwrapping

Preconditions

Each instant must contain all the variables listed in `variables`.

Postconditions

The output base contains as many zones as the input base. Each zone contains the same number of instant found in the zone of the input base. Each instant contains the variables on which the spectrogram was calculated as a matrix of dimensions $(N_b \times N_f)$, where N_b is the number of blocks and N_f is the number of frequencies that can be captured by the Discrete Fourier Transform.

If `t_and_f_in_attrs = True` the attributes of the instant contain the `'time'` and `'frequency'` variables as 1D vectors of length N_b and N_f. If `t_and_f_in_attrs = False`, the `'time'` and `'frequency'` are stored as 2D matrices of dimensions $(N_b \times N_f)$.

Main functions

class `antares.treatment.TreatmentSpecgram.`**`TreatmentSpecgram`**

`execute()`

Compute the Power Spectral Density.

Returns

A base with the same number of zone and instants as the input base and the spectrogram of the `variables`. Time and frequency can be found either in the attributes of the instant or on the instant's variables depending on the value of `t_and_f_in_attrs`.

Return type
`Base`

Example

The following example computes the spectrogram of the following 1D signal between $0 \leq t \leq 5$ with a sampling rate of 10kHz :

$$f(t) = \sin(40\pi t) + \begin{cases} 1.5\sin(60\pi t) & \text{for } 0.6 \leq t \leq 0.8 \\ 1.5\sin(140\pi t) & \text{for } 2.0 \leq t \leq 2.8 \\ \sin(180\pi t) & \text{for } 3.0 \leq t \leq 4 \\ 0 & \text{otherwise} \end{cases}$$

A random uniform noise of amplitude 0.25 is also added to the signal.

The spectrogram is computed using 30 blocks with an overlap of 80% and no zero-padding. The Hanning window function is used.

```
import os
import matplotlib.pyplot as plt
import numpy as np

from antares.api import Base
from antares.treatment import Treatment

np.random.seed(19680801)
output_folder = os.path.join(os.getcwd(), 'OUTPUT', 'SPECGRAM')
os.makedirs(output_folder, exist_ok=True)


## Create signal
## =============

dt = 0.0001
time = np.arange(0.0, 5.0, dt)
s1 = 1 * np.sin(2 * np.pi * 20 * time)
s2 = 1.5 * np.sin(2 * np.pi * 30 * time)
s3 = 1.5 * np.sin(2 * np.pi * 70 * time)
s4 = 1.0 * np.sin(2 * np.pi * 90 * time)

# create a transient "chirp"
s2[time < .6] = s2[0.8 <= time] = 0
s3[time < 2.0] = s3[2.8 <= time] = 0
s4[time < 3.0] = s4[4.0 <= time] = 0

# add some noise into the mix
noise = 0.25 * np.random.uniform(-1, 1, size=len(time))

# The signal
signal = s1 + s2 + s3 + s4 + noise

base = Base()
base.init()
base[0][0]['time'] = time
```

(continues on next page)

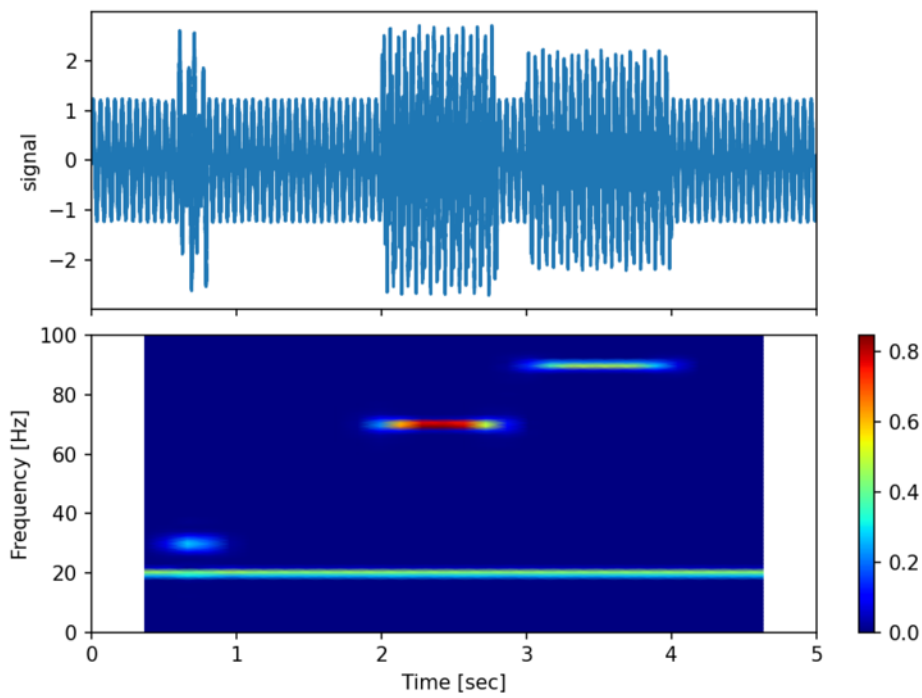
(continued from previous page)

```
base[0][0]['signal'] = signal

## Spectrogram treatment
## ====================

treatment = Treatment('specgram')
treatment['base'] = base
treatment['variables'] = ['time', 'signal']
treatment['nb_block'] = 30
treatment['pad'] = 1
treatment['overlap'] = 0.8
treatment['t_and_f_in_attrs'] = True
treatment['window'] = 'hanning'
result = treatment.execute()

t = result[0][0].attrs['time']
f = result[0][0].attrs['frequency']
Pxx = result[0][0]['signal']

## Plot spectrogram
## ===============
jet_colormap = plt.cm.get_cmap("jet")

fig, ax = plt.subplots(nrows=2, sharex=True, constrained_layout=True)
_ = ax[0].plot(time, signal)
p = ax[1].pcolormesh(t, f, Pxx, cmap=jet_colormap, shading='gouraud')

plt.xlabel('Time [sec]')
ax[0].set_ylabel('signal')
ax[1].set_ylabel('Frequency [Hz]')

ax[1].set_ylim([0, 100])
ax[1].set_xlim([0, 5])

fig.colorbar(p, ax=ax[1], orientation='vertical')

plt.savefig(os.path.join(output_folder, 'specgram_output.png'), dpi=150)
```

Temporal Reconstruction from Dynamic Mode Decomposition

Description

Computes the temporal evolution of modes from a DMD decomposition of a given input Base.



Construction

```
import antares
myt = antares.Treatment('DmdtoTemporal')
```

Parameters

- **base: `Base`**
  The input base resulting from a DMD treatment. (or has the same structure)
- **base_mesh: `Base`**
  The input base containing the mesh of the domain (used in the DMD treatment). This base must only contain one zone.
- **time_step:** ***float***
  Constant time step between snapshots (used in the DMD treatment).
- **nb_instant:** ***float***
  Number of snapshots in the base used for the dmd. the same number are reconstructed.
- **list_modes:** ***list(int)*****, default=** ***[]***
  The number of the modes that have to be reconstructed in time. Each mode is reconstructed one after the other. The output base will contain one zone for each given mode in **list_modes** with **nb_instant** instants in each zone.
- **sum_modes:** ***bool*****, default=** ***False***
  Sum the modes or not.

  If *False*, the output base will contain the instants coming from each mode of the **list_modes**. If *True*, it will only contain the instant for the sum of the modes.

- **dimension:** ***str*****, default=** ***2d_3d***
  Type of the DMD performed beforehand.
  - *1d*: Treatment('dmd1d')
  - *2d_3d*: Treatment('dmd')
- **variables:** ***list(str)*****, default=** ***[]***
  The variable names considered for the temporal reconstruction.
- **temporal_amplification:** ***bool*****, default=** ***True***
  Variables to perform the temporal reconstruction. Into DMD approach, a temporal amplification term can be accounted for conversely to classical FFT approaches. In numerical simulations for steady regimes, since the extraction time is generally sparse, the amplification term can be different than zero for periodic phenomena which is generally to be avoided and it is so advised to turn off this term. However, when looking at transient phenomena, this term can be activated.

Preconditions

The input base **base** must be issued from the DMD treatments (dmd1d, dmd).

The input base **base_mesh** must only contain one zone.

Postconditions

The output base contains the coordinates of **base_mesh** and the instants that have been reconstructed.

Example

```
import antares
myt = antares.Treatment('DmdtoTemporal')
myt['base'] = dmd_base
myt['base_mesh'] = mesh_base
myt['list_modes'] = [1, 2]
myt['sum_modes'] = True
myt['nb_instant'] = 4
myt ['time_step'] = 1e-4
myt ['variables'] = ['rhovx', 'rhovy', 'rhovz', 'rhoE']
output_base = myt.execute()
```

Main functions

class antares.treatment.TreatmentDmdtoTemporal.**TreatmentDmdtoTemporal**

execute()

Computes the temporal evolution of modes.

Returns
The base with reconstructed instants.

Return type
Base

Example

```
"""
This example illustrates the reconstruction of temporal
snapshots for modes (Dynamic Mode Decomposition)
of a 2D signal.
"""
import os
if not os.path.isdir('OUTPUT'):
    os.makedirs('OUTPUT')

import antares
import numpy as np

# ------------------
# Read the files
# ------------------
r = antares.Reader('bin_tp')
r['filename'] = os.path.join('..', 'data', 'CAVITE', 'DMD_CUTS', 'cut_y_ite<instant>')
r['instant_step'] = 2
base = r.read()
# 3D base with flowfield

base.compute_cell_volume()

# put values at node location
base.cell_to_node()

# only keep node values
base = base.get_location('node')

# keep only one zone
merge = antares.Treatment('merge')
merge['base'] = base
base = merge.execute()

# keep a base with 3D coordinates
base_mesh = base[:, :, (('x','node'),('y','node'),('z','node'))]

# remove coordinates for DMD treatment
del base[:, :, (('x', 'node'), ('y', 'node'), ('z', 'node'))]

# get modes
treatment = antares.Treatment('Dmd')
treatment['time_step'] = 4e-4
treatment['noiselevel'] = 1e-5
treatment['base'] = base
treatment['type'] = 'mod/phi'
treatment['variables'] = ['cell_volume', 'rovx', 'rovy', 'rovz', 'roE']
result = treatment.execute()

# reconstruct time evolution for two modes
```

(continues on next page)

(continued from previous page)

```
treatment = antares.Treatment('DmdtoTemporal')
treatment['base'] = result
treatment['base_mesh'] = base_mesh
treatment['list_modes'] = [1, 2]
treatment['sum_modes'] = True
treatment['nb_instant'] = len(base[0].keys())
treatment['time_step'] = 1e-4
treatment['variables'] = ['rovx', 'rovy', 'rovz', 'roE']
base_time = treatment.execute()

base_time.show()

w = antares.Writer('bin_tp')
w['filename'] = os.path.join('OUTPUT', 'test_dmd2time.plt')
w['base'] = base_time
w.dump()
```

Temporal Reconstruction from Proper Orthogonal Decomposition

Description

This treatment computes the temporal evolution of modes from a POD decomposition of a given input Base.

Parameters

- **base_pod: `Base`**
  The input base resulting from a treatment *`antares.treatment.TreatmentPOD`* (page 68), or having the same structure.
- **base_mesh: `Base`**
  The input base containing the mesh of the domain used in the POD treatment. This base must only contain one zone.
- **mode: *int*, default= *0***
  The POD mode number that has to be reconstructed in time. The first mode is the mode 0 and has the highest eigenvalue (largest energy content).
- **variables: *list(str)*, default= *[]***
  The variable names considered for the temporal reconstruction.
- **coordinates: *list(str)***
  The variable names that define the set of coordinates. If no value is given, the default coordinate system of the base **base_mesh** is used (see `Base.coordinate_names`).

Preconditions

The input base **base** must be issued from the *antares.treatment.TreatmentPOD* (page 68) treatment, or it must respect the same structure as the output base of this treatment.

The input base **base_mesh** must only contain one zone.

Postconditions

The output base will contain one zone for the given **mode**.

The output base contains the coordinates of **base_mesh** and the instants that have been reconstructed from the POD mode considered.

Example

```
import antares
myt = antares.Treatment('PODtoTemporal')
myt['base_pod'] = POD_base
myt['base_mesh'] = mesh_base
myt['mode'] = 0
myt['variables'] = ['psta']
output_base = myt.execute()
```

Main functions

class antares.treatment.TreatmentPODtoTemporal.**TreatmentPODtoTemporal**

execute()

Compute the temporal evolution of the considered POD mode.

Returns
The base with reconstructed instants.

Return type
Base

Example

```
"""
This example illustrates the Proper Orthogonal Decomposition.
"""
import os
import numpy as np
import antares

if not os.path.isdir('OUTPUT'):
    os.makedirs('OUTPUT')

# Create the input Base
```

(continues on next page)

(continued from previous page)

```
time_step = 0.0003
nb_snapshot = 50
nb_space_points = 100

base = antares.Base()
zone = base['zone_0'] = antares.Zone()
for snapshot_idx in range(nb_snapshot):
    instant_name = 'snapshot_%s' % snapshot_idx
    inst = zone[instant_name] = antares.Instant()
    inst['pressure'] = 25.4 * np.cos(2 * np.pi * 250 * snapshot_idx * time_step + np.
↪linspace(0, 2.5 * np.pi, nb_space_points))
    # constant pressure over the space domain evolving in time
    # base[0][instant_name]['pressure'] = np.cos(2 * np.pi * 0.01 * snapshot_idx * 2 + np.
↪zeros((nb_space_points,)))

# ------------------
# Compute the POD
# ------------------
treatment = antares.Treatment('POD')
treatment['base'] = base
treatment['tolerance'] = 0.99
treatment['dim_max'] = 100
treatment['POD_vectors'] = True
treatment['variables'] = ['pressure']
result = treatment.execute()
result.show()
# ------------------
# Get some results
# ------------------
print("POD modes: ", result[0]['modes'][0])
print("POD parameters: ")
for k, v in result.attrs.items():
    print(k, v)

base_mesh = antares.Base()
base_mesh.init()
base_mesh[0][0]['x'] = range(nb_space_points)

t = antares.Treatment('PODtoTemporal')
t['base_mesh'] = base_mesh
t['base_pod'] = result
t['variable'] = ['pressure']
t['mode'] = 0
t['coordinates'] = ['x']
base_pod_temporal = t.execute()

writer = antares.Writer('bin_tp')
writer['base'] = base_pod_temporal
writer['filename'] = os.path.join('OUTPUT', 'ex_mode_0_reconstructed_temporal.plt')
writer.dump()
```

Azimuthal decomposition

Description

This treatment performs an azimuthal mode decomposition on axial plane.

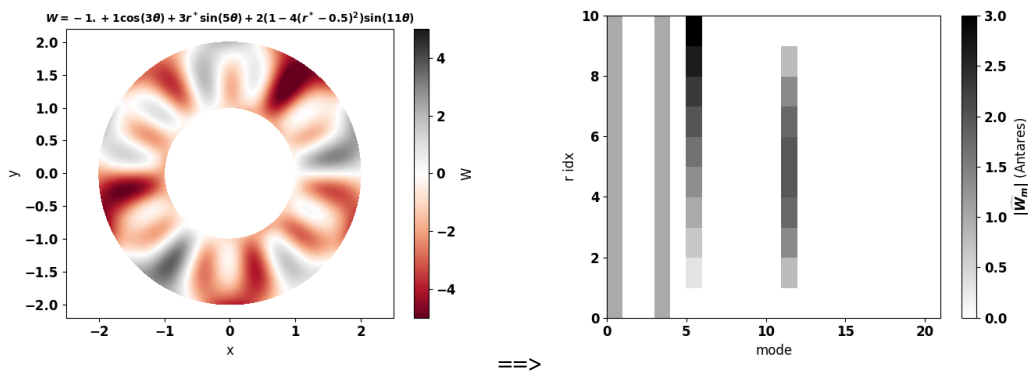
If $W(x, r, \theta, t)$ represents the azimuthal evolution at point (x, r) and instant t of the variable of interest, then the mode $\widehat{W}_m(x, r, t)$ can be computed by the formulas below. If Θ is the azimuthal extent (not necessarily 2π) of the data then the mode m corresponds to the frequency $\frac{m}{\Theta}$. N_θ (i.e. **nb_theta**) is the total number of points of the uniform azimuthal discretization and $\theta_k = \frac{k}{N_\theta}\Theta$. The convention that is used depends on the type of the input data (real or complex).

For real input data, $\widehat{W}_m(x, r, t)$ is defined for positive integers only by:

- $\widehat{W}_0(x, r, t) = \frac{1}{N_\theta}\sum_{k=0}^{N_\theta} W(x, r, \theta_k, t)$
- $\widehat{W}_m(x, r, t) = \frac{2}{N_\theta}\sum_{k=0}^{N_\theta} W(x, r, \theta_k, t)e^{(-im\theta_k)}$ for $m >= 1$

For complex input data, $W = \mathbb{R}\mathrm{e}\{W\} + i * \mathbb{I}\mathrm{m}\{W\} = \|W\|e^{i\arg(W)}$, $\widehat{W}_m(x, r, t)$ is defined for both positive and negative integers by:

- $\widehat{W}_m(x, r, t) = \frac{1}{N_\theta}\sum_{k=0}^{N_\theta} W(x, r, \theta_k, t)e^{(-im\theta_k)}$



On the left: initial field of W.

On the right: azimuthal modes $\|\widehat{W}_m\|$ as a function of the mode and the radial index (see example below).

Construction

```
import antares
myt = antares.Treatment('AzimModes')
```

Parameters

- **base: Base**
  The base on which the azimuthal modes will be computed. It can contain several zones and several instants. If **dtype_in** is *'re'*, then the different instants of the base should correspond to different time iterations. If **dtype_in** is in [*'mod/phi'*, *'im/re'*], then the instants of the base should contain the different time harmonics: 'Hxx_mod' and 'Hxx_phi' for *'mod/phi'* or 'Hxx_re' and 'Hxx_im' for 'im/re' (with xx the number of the harmonic), also an instant 'mean' containing real data can exist. Decomposition is performed on all variables except **coordinates**.

- **dtype_in:** ***str*****, default=** ***'im/re'***
  The decomposition type of the input data: *'re'* for real data, *'mod/phi'* for modulus/phase decomposition or *'im/re'* for imaginery/real part decomposition. If given, the phase must be expressed in degrees.
- **dtype_out:** ***str*****, default=** ***'im/re'***
  The decomposition type of the output data: *'mod/phi'* for modulus/phase decomposition or *'im/re'* for imaginery/real part decomposition. The phase is expressed in degrees.
- **ring:** ***boolean*****, default=** ***False***
  Perform the azimuthal decomposition on the specific case of a microphone azimuthal ring array. Warning: the distributed microphones is supposed to be regularly distributed in azimuthal and radial direction on the ring array.
- **coordinates:** ***list(str)*****, default=** `antares.Base.coordinate_names`
  The cylindrical coordinates. The angle coordinate must be expressed in radian.
- **x_values:** ***int*** **or** ***list(int)*** **or** ***None*****, default=** ***None***
  The axial positions of the planes on which the azimuthal modes are computed. If None, it is assumed that the base is already a plane.
- **r_threshold:** ***tuple*** **or** ***None*****, default=** ***None***
  Perform the azimuthal decomposition in the defined range of radial values. Useful for multiple coaxial cylinders.
- **nb_r:** ***int*****, default=** ***20***
  The number of points in the radial direction of the output base.
- **m_max:** ***int*****, default=** ***50***
  The maximum azimuthal mode order to compute. If **dtype_in** is *'re'*, then *'m_max'* + 1 modes are computed (from mode 0 to mode 'm_max'). If **dtype_in** in ['mod/phi', 'im/re'], then 2 * *'m_max'* + 1 modes are computed (from mode -*'m_max'* to mode *'m_max'*).
- **nb_theta:** ***int*****, default=** ***400***
  The uniform azimuthal line is reconstructed via an interpolation. **n_theta** is the number of points for this interpolation.

Preconditions

All the zones must have the same instant.

if **dtype_in** = *'im/re'* or *'mod/phi'*, instants must be named <harm>_im and <harm>_re or <harm>_mod and <harm>_phi where <harm> is the harmonic. If given, the phase must be expressed in degrees.

Variables must be at nodes, if not the method cell_to_node() is executed.

The input base must be continuous after axial cut but the azimuthal extent can be less than 2π.

Mesh is not moving.

Postconditions

If **dtype_in** = *'mod/phi'*, the phase is expressed in degrees.

The result base contain one zone and as much instant than in the input base with suffixes '_im' and '_re' or '_mod' and '_phi'. Each variable is a numpy array with dimensions (nb_x, **nb_r**, **m_max** + 1) if **dtype_in** = *'re'* and with dimensions (nb_x, **nb_r**, 2 * **m_max** + 1) else (nb_x is the number of x values).

The azimuthal coordinate is replaced by the azimuthal mode order 'm'.

Example

The following example shows azimuthal decomposition.

```
import antares
myt = antares.Treatment('AzimModes')
myt['base'] = base
myt['dtype_in'] = 're'
myt['dtype_out'] = 'mod/phi'
myt['coordinates'] = ['x', 'r', 'theta']
myt['x_values'] = 0.
myt['nb_r'] = 20
myt['m_max'] = 50
myt['nb_theta'] = 400
base_modes = myt.execute()
```

Warning: The frame of reference must be the cylindrical coordinate system. No checking is made on this point.

Main functions

class antares.treatment.TreatmentAzimModes.**TreatmentAzimModes**

execute()

Execute the treatment.

Returns

A base that contains one zone. This zone contains several instants. If 'dtype_in' is 're', then there are twice as much instants as in the input base (two instants per original instant in order to store seperately modulus and phase or imaginery and real parts). If 'dtype_in' is in ['mod/phi', 'im/re'], then there are as many instants in the output base as in the input base. The instant names are suffixed by '_mod' and '_phi' if 'dtype_out' is 'mod/phi' or by '_re' and '_im' if 'dtype_out' is 'im/re'. Each instant contains the azimuthal modes of the input base variables at different axial and radial positions. The azimuthal coordinate is replaced by the azimuthal mode order 'm'.

Return type

base

Example

```
"""
This example shows how to use the azimuthal decomposition treatment.
"""

import antares
import numpy as np
import matplotlib.pyplot as plt

# Define test case
################################################################################
# Geometry
x0 = 0.
rmin, rmax, Nr = 1., 2., 50
tmin, tmax, Nt = 0., 2.*np.pi, 1000
x, r, t = np.linspace(x0, x0, 1), np.linspace(rmin, rmax, Nr), np.linspace(tmin, tmax,␣
↪Nt)
X, R, T = np.meshgrid(x, r, t)

# Field
r_ = (R - rmin) / (rmax - rmin)
W = -1. +  1.*np.cos(3*T) + 3.*r_*np.sin(5*T) + 2*(1 - 4*(r_ - 0.5)**2)*np.sin(11*T)

# Create antares Base
base = antares.Base()
base['0'] = antares.Zone()
base['0']['0'] = antares.Instant()
base['0']['0']['X'] = X
base['0']['0']['R'] = R
base['0']['0']['T'] = T
base['0']['0']['Y'] = R*np.cos(T)
base['0']['0']['Z'] = R*np.sin(T)
base['0']['0']['W'] = W

# Azimutal decomposition
################################################################################
myt = antares.Treatment('AzimModes')
myt['base'] = base[:,:,('X', 'R', 'T', 'W')]
myt['dtype_in'] = 're'
myt['dtype_out'] = 'mod/phi'
myt['coordinates'] = ['X', 'R', 'T']
myt['x_values'] = None
myt['nb_r'] = 10
myt['m_max'] = 20
myt['nb_theta'] = 800
base_modes = myt.execute()

# Plot data
################################################################################
# font properties
font = {'family':'sans-serif', 'sans-serif':'Helvetica', 'weight':'semibold', 'size':12}
```

(continues on next page)

(continued from previous page)

```
# Plot initial field
t = antares.Treatment('tetrahedralize')
t['base'] = base
base = t.execute()

plt.figure()
plt.rc('font', **font)
plt.tripcolor(base[0][0]['Y'], \
              base[0][0]['Z'], \
              base[0][0].connectivity['tri'], \
              base[0][0]['W'], \
              vmin=-5, vmax=5)
plt.gca().axis('equal')
cbar = plt.colorbar()
cbar.set_label('W')
plt.set_cmap('RdGy')
plt.xlabel('y')
plt.ylabel('z')
plt.title('$W = -1. + 1\\cos(3\\theta) + 3r^*\\sin(5\\theta) + 2(1-4(r^*-0.5)^2)\\sin(11\
↪\theta)$', \
    fontsize=10)
plt.tight_layout()

# Modes computed by antares
mod, phi = base_modes[0]['0_mod'], base_modes[0]['0_phi']

def plot_modes(modes, title='', vmin=0, vmax=1):
    plt.figure()
    plt.rc('font', **font)
    plt.pcolor(modes, vmin=vmin, vmax=vmax)
    cbar = plt.colorbar()
    cbar.set_label(title)
    plt.set_cmap('binary')
    plt.xlabel('mode')
    plt.ylabel('r idx')
    plt.tight_layout()

# Plot modes computed by Antares
plot_modes(modes=mod['W'][0,:,:], title='$\|\\widehat{W}_m\|$ (Antares)', \
    vmin=0, vmax=3)

# Compute analytical solution
mod_th = np.zeros(shape=mod['W'][0,:,:].shape)
mod_th[:,0] = 1.
mod_th[:,3] = 1.
mod_th[:,5] = 3*(mod['R'][0,:,5] - rmin) / (rmax - rmin)
mod_th[:,11] = 2*(1 - 4*((mod['R'][0,:,11] - rmin) / (rmax - rmin) - 0.5)**2)

# Plot analytical solution
plot_modes(modes=mod_th, title='$\|\\widehat{W}_m\|$ (theory)', \
    vmin=0, vmax=3)
```

(continues on next page)

(continued from previous page)

```
# Compute relative error with analytical solution
rel_err = np.abs(mod_th - mod['W'][0,:,:]) / np.max(mod_th) * 100
print('Max relative error between theory and Antares is {:} %.'.format(np.amax(rel_err)))

# Plot relative error with analytical solution
plot_modes(modes=rel_err, title='Relative error with analytical solution [%]', \
    vmin=0, vmax=0.2)

plt.show()
```

Radial decomposition

Description

This treatment computes the radial modes (useful for duct acoustics) by a projection of each azimuthal mode over the radial eigenfunctions (combinaison of Bessel functions of first and second kinds).

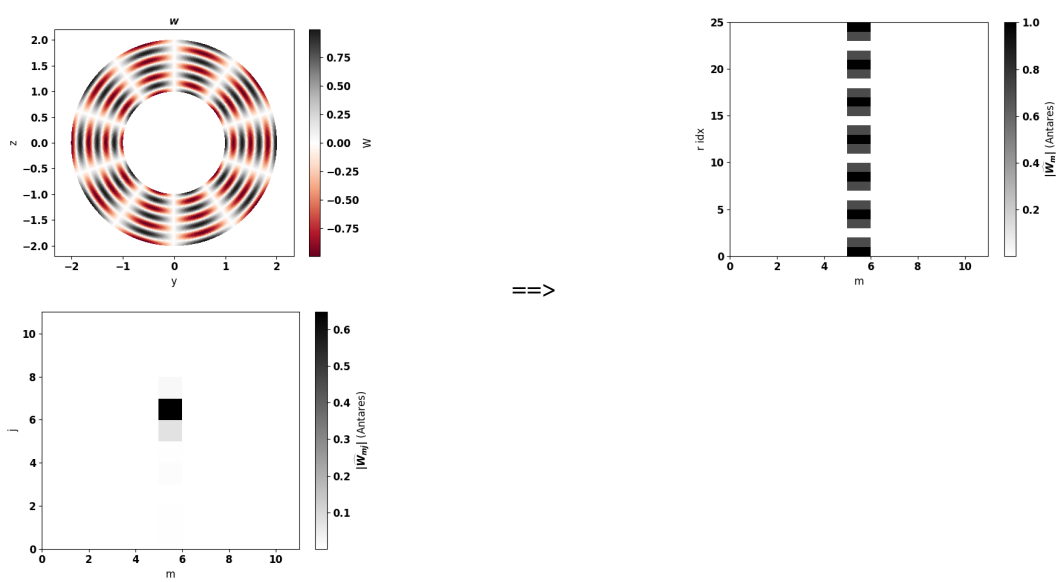
It must be used after computing the azimuthal modes (with the Azimuthal Mode Decomposition treatment `antares.treatment.TreatmentAzimModes` (page 103)). If $\hat{W}_m(x,r,t)$ denotes the radial evolution of the azimuthal mode of order m (with $R_{min} < r < R_{max}$), then the radial modes $\hat{W}_{mj}(x,t)$ (where j is the radial mode order) are given by

$$\hat{W}_{mj}(x,t) = \frac{2}{R_{max}^2 - R_{min}^2} \int_{R_{min}}^{R_{max}} \hat{W}_m(x,r,t) E_{mj}(x,r) r dr.$$

The radial eigenfunctions $E_{mj}(x,r)$ are expressed as

$$E_{mj}(x,r) = A_{mj}(x) J_m(K_{mj}(x) r) + B_{mj}(x) Y_m(K_{mj}(x) r),$$

where J_m and Y_m are the Bessel functions of order m of first and second kinds respectively and A_{mj}, B_{mj} and K_{mj} are duct coefficients (depending on the minimum and maximum duct radius R_{min} and R_{max}).



From left to right: initial field W, azimuthal modes $\|\widehat{W}_m\|$ computed with `antares.treatment.TreatmentAzimModes` (page 103) and radial modes $\|\widehat{W}_{mj}\|$ computed with the radmodes treatment.

Construction

```
import antares
myt = antares.Treatment('RadModes')
```

Parameters

- **base: `Base`**
  The input base on which the radial mode decomposition will be done. It must result from the Azimuthal Mode Decomposition treatment. It should therefore contain one zone with several instants. The instant names should be suffixed by '_mod' and '_phi' if **dtype_in** is *'mod/phi'* or by '_re' and '_im' if **dtype_in** is *'im/re'*. The first three variables of each instant should be the axial coordinate, the radial coordinate and the azimuthal mode order 'm' (in this order). The other variables are the ones on which the decomposition will be done.
- **dtype_in: *str*, default= *'im/re'***
  The decomposition type of the input data: *'mod/phi'* for modulus/phase decomposition or *'im/re'* for imaginery/real part decomposition. If given, the phase must be expressed in degrees.
- **dtype_out: *str*, default= *'im/re'***
  The decomposition type used for the output data: *'mod/phi'* for modulus/phase decomposition or *'im/re'* for imaginery/real part decomposition. If given, the phase is expressed in degrees.
- **j_max: *int*, default= *10***
  The maximum radial mode order to compute (**j_max** + 1 modes are computed).

Preconditions

The input base has to be constructed by the treatment azimmodes (`antares.treatment.TreatmentAzimModes` (page 103)).

Postconditions

The output base contains one zone and as much instant as in the input base with suffixes '_im' and '_re' or '_mod' and '_phi'. Each variable is a numpy array with dimensions (nb_x, **j_max** + 1, nb_m) where nb_x is the number of x values and nb_m is the number of azimuthal modes of the input base.

The radial coordinate is replaced by the radial mode order 'j'.

Example

The following example shows radial decomposition.

```
import antares
myt = antares.Treatment('RadModes')
myt['base'] = base_azimmodes
myt['dtype_in'] = 'mod/phi'
myt['dtype_out'] = 'mod/phi'
myt['j_max'] = 10
base_radmodes = myt.execute()
```

Warning: The frame of reference must be the cylindrical coordinate system. No checking is made on this point.

Main functions

class antares.treatment.TreatmentRadModes.**TreatmentRadModes**

execute()

Execute the treatment.

Returns

A base that contains one zone. This zone contains as many instants as in the input base. The instant names are suffixed by '_mod' and '_phi' if 'dtype_out' is 'mod/phi' or by '_re' and '_im' if 'dtype_out' is 'im/re'. Each instant contains the radial modes of the input base variables for all provided azimuthal mode orders and axial positions. The radial coordinate is replaced by the radial mode order 'j'.

Return type

base

Example

```
"""
This example shows how to use the radial decomposition treatment.
"""

import antares
import numpy as np
import matplotlib.pyplot as plt

# Define test case
################################################################################
# Geometry
x0 = 0.
rmin, rmax, Nr = 1., 2., 100
tmin, tmax, Nt = 0., 2.*np.pi, 1000
x, r, t = np.linspace(x0, x0, 1), np.linspace(rmin, rmax, Nr), np.linspace(tmin, tmax,␣
↪Nt)
X, R, T = np.meshgrid(x, r, t)

# Field
r_ = R / (rmax - rmin)
W = np.cos(2*np.pi*3*r_)*np.cos(5*T)

# Create antares Base
base = antares.Base()
base['0'] = antares.Zone()
base['0']['0'] = antares.Instant()
base['0']['0']['X'] = X
base['0']['0']['R'] = R
base['0']['0']['T'] = T
```

(continues on next page)

(continued from previous page)

```
base['0']['0']['Y'] = R*np.cos(T)
base['0']['0']['Z'] = R*np.sin(T)
base['0']['0']['W'] = W

# Azimutal decomposition
##############################################################################
myt = antares.Treatment('AzimModes')
myt['base'] = base[:,:,('X', 'R', 'T', 'W')]
myt['dtype_in'] = 're'
myt['dtype_out'] = 'mod/phi'
myt['coordinates'] = ['X', 'R', 'T']
myt['x_values'] = None
myt['nb_r'] = 25
myt['m_max'] = 10
myt['nb_theta'] = 800
base_modes = myt.execute()

# Plot data
##############################################################################
# font properties
font = {'family':'sans-serif', 'sans-serif':'Helvetica', 'weight':'semibold', 'size':12}

# Plot initial field
t = antares.Treatment('tetrahedralize')
t['base'] = base
base = t.execute()

plt.figure()
plt.rc('font', **font)
plt.tripcolor(base[0][0]['Y'], \
              base[0][0]['Z'], \
              base[0][0].connectivity['tri'], \
              base[0][0]['W'])
plt.gca().axis('equal')
cbar = plt.colorbar()
cbar.set_label('W')
plt.set_cmap('RdGy')
plt.xlabel('y')
plt.ylabel('z')
plt.title('$W$', \
    fontsize=10)
plt.tight_layout()

# Modes computed by antares
mod, phi = base_modes[0]['0_mod'], base_modes[0]['0_phi']

def plot_modes(modes, title='', xlabel='', ylabel=''):
    plt.figure()
    plt.rc('font', **font)
    plt.pcolor(modes)
    cbar = plt.colorbar()
    cbar.set_label(title)
```

(continues on next page)

(continued from previous page)

```
    plt.set_cmap('binary')
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.tight_layout()

# Plot modes computed by Antares
plot_modes(modes=mod['W'][0,:,:], title='$\|\\widehat{W}_m\|$ (Antares)', xlabel='m',␣
↪ylabel='r idx')

# Radial decomposition
##############################################################################
myt = antares.Treatment('RadModes')
myt['base'] = base_modes
myt['dtype_in'] = 'mod/phi'
myt['dtype_out'] = 'mod/phi'
myt['j_max'] = 10
base_radmodes = myt.execute()

# Modes computed by antares
mod, phi = base_radmodes[0]['0_mod'], base_radmodes[0]['0_phi']

# Plot modes computed by Antares
plot_modes(modes=mod['W'][0,:,:], title='$\|\\widehat{W}_{mj}\|$ (Antares)', xlabel='m',␣
↪ylabel='j')

plt.show()
```

Acoustic Power

Description

Compute the acoustic power in Watt across a surface S defined as

$$P = \iint_S \vec{I} \cdot \vec{n} \, dS$$

where $\vec{I} = p'\vec{v'}$ is the acoustic intensity, with p' the acoustic pressure (or pressure fluctuation), $\vec{v'}$ the velocity fluctuations, and $\vec{n}$ the unit vector normal to the surface.

Compute also the Sound Power Level in dB and the Sound Pressure Level in dB.

Two formulations are considered:

- For a plane acoustic wave as a duct mode, or a progressive acoustic wave at infinity in a stagnant uniform fluid, the component of the acoustic intensity in the direction of propagation is

  $$I = p'^2/(\rho_0 c_0)$$

  with p' the acoustic pressure, ρ_0 the density and c_0 the sound speed at infinity.

  It can be shown for a stagnant uniform fluid that I_x is given by

  $$I_x = p'u'$$

  with u' the axial component of the velocity disturbances.

- For homentropic non-uniform fluid, the acoustic intensity is expressed as

$$\vec{I} = \left(\frac{p'}{\rho_0} + \vec{v_0} \cdot \vec{v'}\right)\left(\rho_0 \vec{v'} + \rho' \vec{v_0}\right)$$

  with the mean velocity $\vec{v_0} = (U_0, V_0, W_0)$ and the velocity disturbances $\vec{v'} = (u', v', w')$.

Frequential formulations of previous definitions are also implemented to evaluate the contribution of each frequency to the total acoustic power.

For more details on aeroacoustics, you may refer to

Parameters

- **base: `Base`**
  The input base.
- **dtype_in: *str*, default= *'re'***
  If **dtype_in** is *'re'*, then the base is real. If **dtype_in** is in [*'mod/phi'*, *'im/re'*], the base is complex (modulus/phase or imaginary/real part decomposition respectively). If the signal is complex, a suffix must be added to the name of the variable depending on the decomposition (_im and _re for im/re, _mod and _phi for mod/phi). If given, the phase must be expressed in radians.
- **flow: *str*, default= *'stagnant'***
  The state of the medium: *'stagnant'* or *'non-uniform'*.
- **variables: *list(str)***
  The variable names.
- **rho_ref: *float*, default= *1.18***
  The value of the ambient density. The default value is $\rho = 1.18$ kg/m3, i.e. for a medium at an ambient temperature and pressure of T=298 K and P=101325 Pa, respectively.
- **c_ref: *float*, default= *346.0***
  The value of the ambient sound velocity, only for real data. The default value is c=346 m/s, i.e. for a medium at an ambient temperature of T=298 K.
- **mach_ref: *float*, default= *0.0***
  Mach number.

Preconditions

Zones may be either structured or unstructured.

Stagnant uniform fluid:

1. the required variables is the mean square of the acoustic pressure fluctuation p', i.e. $< p'p' >$
2. the reference uniform density ρ_0 and uniform sound velocity c_0

Homentropic non-uniform fluid:

1. the required variables are:

- the mean velocity vector $\vec{v_0} = (U_0, V_0, W_0)$
- the fluctuating velocity vector $\vec{v'} = (u', v', w')$
- the mean density field ρ_0
- the fluctuating pressure field p'

Postconditions

The output base contains a single zone with a single instant with 3 or 4 scalar variables depending on **dtype_in**:

- "Power (Watt)": the acoustic power (Watt)

or

- "Power_re(Watt)": the real part of the acoustic power (Watt)
- "Power_im(Watt)": the imaginary part of the acoustic power (Watt)

and

- "Sound Power Level (dB)": the Sound Power Level (dB)
- "Sound Pressure Level (dB)": the Sound Pressure Level (dB)

Example

```
import antares
myt = antares.Treatment('acousticpower')
myt['base'] = base
myt['dtype_in'] = 're'
myt['flow'] = 'stagnant'
myt['rho_ref'] = rho
myt['c_ref'] = c
power = myt.execute()
```

Main functions

class antares.treatment.TreatmentAcousticPower.**TreatmentAcousticPower**

execute()

Compute the acoustic power across a surface.

Translation

Description

Translates zones of a base with respect to translation vector.

Construction

```
import antares
myt = antares.Treatment('translation')
```

Parameters

- **base:** `Base`
  The input base to be translated.
- **coordinates:** *list(str)*, **default=** *None*
  The variable names that define the set of coordinates. If *None* the default coordinate system of the base is used.
- **vector:** `seq(float)`, **default=*[]***
  The displacement vector (sequence of floats) used to translate the base.
- **memory_mode:** *bool*, **default=** *False*
  If True, the modifications are done directly on the input base to limit memory usage. If False, a new base is created.

Preconditions

- The coordinate system must be the cartesian coordinate system.
- The dimension of *vector* must match the length of the coordinate system

Postconditions

- If **memory_mode** = False, the input base remains unchanged and the output base has the same structure as the input base. If **memory_mode** = True, the input base is modified in-place and the returned base is the same object as the input base.

Example

The following example shows a translation of a base by a vector [10., 10., 10.].

```
import antares
myt = antares.Treatment('translation')
myt['base'] = base
myt['coordinates'] = ['x', 'y', 'z']
myt['vector'] = [10., 10., 10.]
translated_base = myt.execute()
```

Main functions

class antares.treatment.TreatmentTranslation.**TreatmentTranslation**

execute()

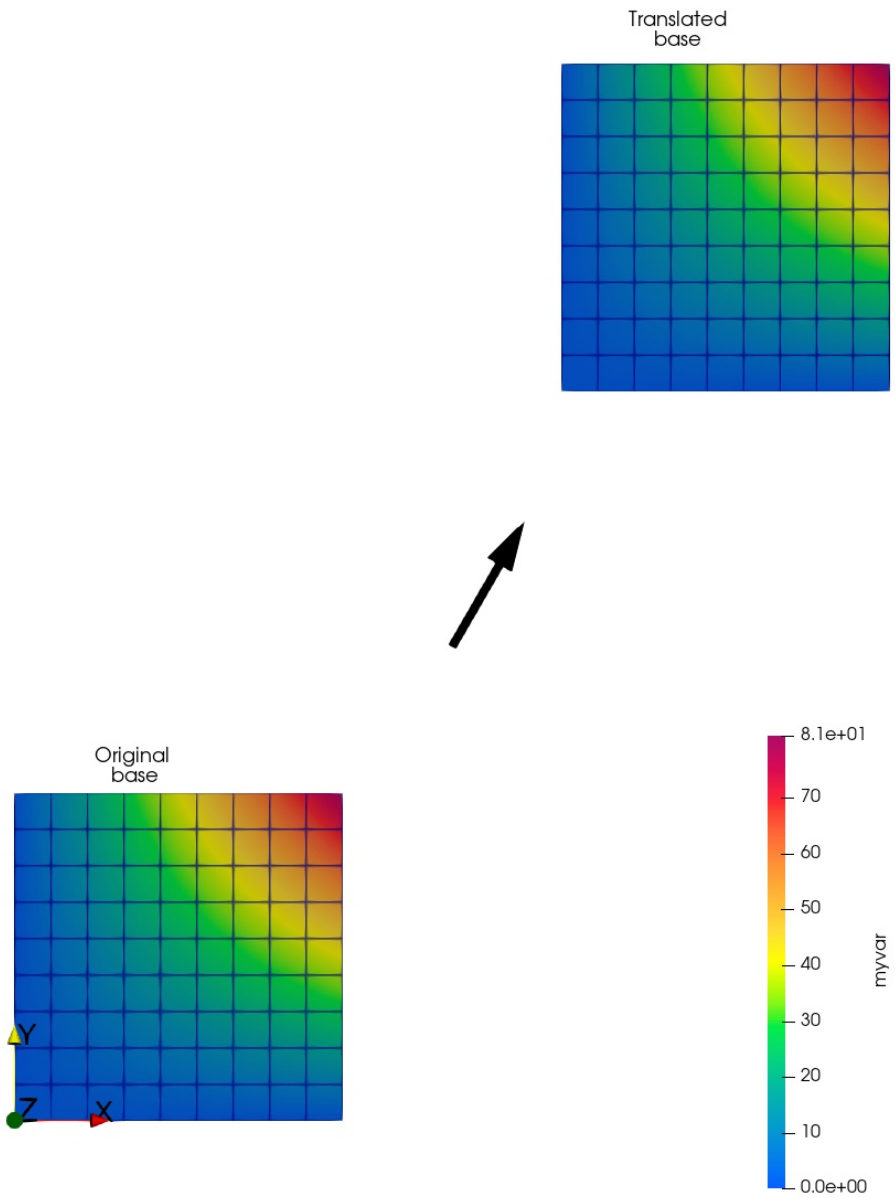
Translates the base.

Returns
the base containing the results.

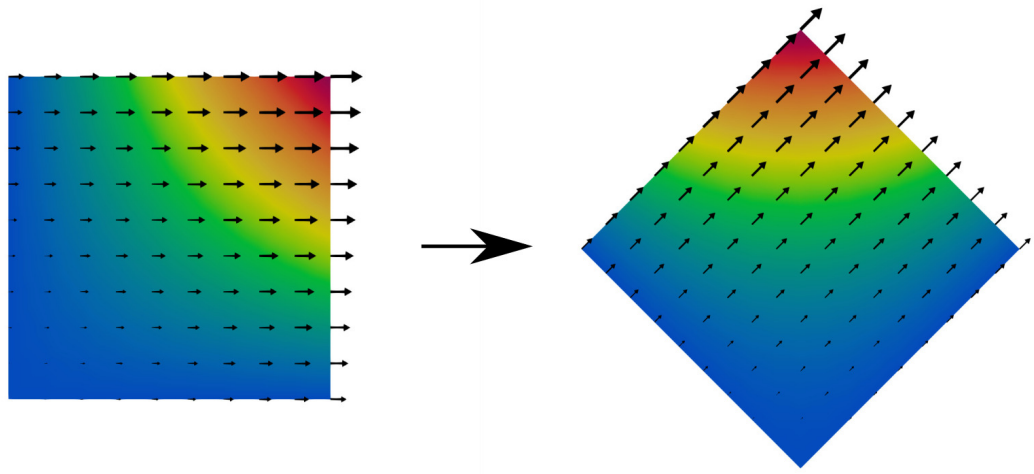
Return type
`Base`

Example

```
import antares
import numpy as np
import os

output_folder = os.path.join("OUTPUT", "TreatmentTranslation")
os.makedirs(output_folder, exist_ok=True)

X, Y = np.mgrid[0:10, 0:10]

# Create base to translate
base = antares.Base()
base.init()
base[0][0]['x'] = X
base[0][0]['y'] = Y
base[0][0]['myvar'] = X*Y # create a dummy variable
base.coordinate_names = ['x', 'y']

# Dump base to translate
w = antares.Writer('hdf_antares')
w['base'] = base
w['filename'] = os.path.join(output_folder, 'base_to_translate')
w.dump()

# Apply translation treatment
translation = antares.Treatment('translation')
translation['base'] = base
translation['vector'] = [15, 20]
translated_base = translation.execute()

# Dump result
w = antares.Writer('hdf_antares')
w['base'] = translated_base
w['filename'] = os.path.join(output_folder, 'translated_base')
w.dump()
```


Translated
base
Original
base
Y
Z
X
8.1e+01
70
60
50
40
30
20
10
0.0e+00
myvar

Rotation



Description

Rotates the geometry and the vector variables in it.

For two-dimensional bases, a rotation by a given angle is done from the first axis to the second axis in the coordinate system.

For three-dimensional bases, a rotation by a given angle around a given axis direction is performed using the right hand rule.

Parameters

- **base:** `Base`
  The input base to be rotated.
- **coordinates:** *list(str)*, **default=** *None*
  The variable names that define the set of coordinates. The coordinate system must be the cartesian coordinate system. If *None* the internal base coordinate system is used.
- **angle:** *int or float*
  Angle of rotation in radians.
- **axis:** *list(float)*, **default=** *None*
  The axis of rotation (only for three-dimensional bases).
- **vectors:** *list(list(str))*, **default=** *[]*
  A list of lists containing name the components of the vectors to be rotated. The components must be specified in the same order as the **coordinate** keyword.
- **memory_mode:** *bool*, **default=** *False*
  If True, the modifications are done directly on the input base to limit memory usage. If False, a new base is created.

Preconditions

- The base must be in the Cartesian system of coordinates.
- The base must be either two-dimensional or three-dimensional.
- If the base is two-dimensional, the axis keyword must not be specified.
- If the base is three-dimensional, both the angle and axis keywords are needed.

Postconditions

- If **memory_mode** = False, the input base remains unchanged and the output base has the same structure as the input base. If **memory_mode** = True, the input base is modified in-place and the returned base is the same object as the input base.

Example

The following example shows how to rotate a base by an angle of 45 degrees around the y axis, rotating also the velocity vector [u, v, w].

```
import antares
myt = antares.Treatment('rotation')
myt['base'] = base
myt['coordinates'] = ['x', 'y', 'z']
myt['axis'] = [0, 1, 0]
myt['angle'] = np.pi / 4
myt['vectors'] = [ ['u', 'v', 'w'] ]
rotated_base = myt.execute()
```

Main functions

class antares.treatment.TreatmentRotation.**TreatmentRotation**

execute()

Rotates the base.

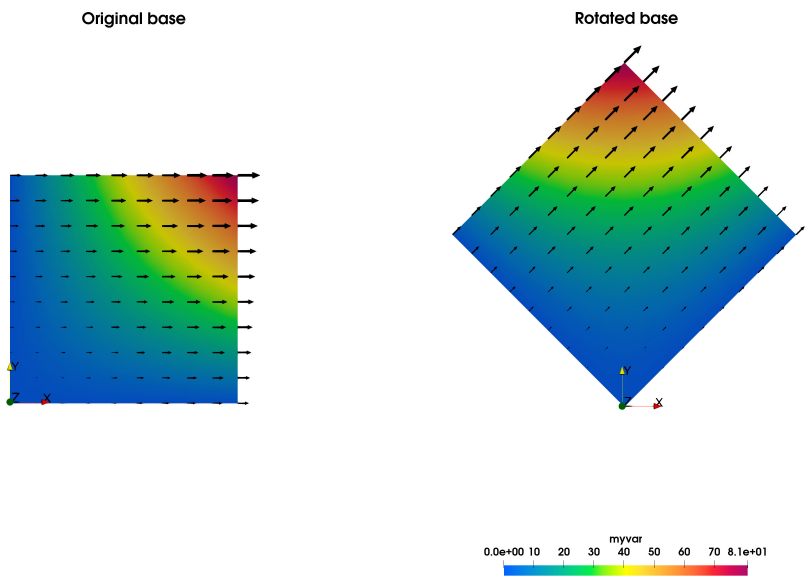
Returns
the base containing the results.

Return type
Base

Example

The following examples creates a 2D structured base with a scalar and a vector variable and it uses the rotation treatment to rotate the base by 45 degrees.

```
import antares
import numpy as np
import os

output_folder = os.path.join("OUTPUT", "TreatmentRotation")
os.makedirs(output_folder, exist_ok=True)

X, Y = np.mgrid[0:10, 0:10]

# Create base to translate
base = antares.Base()
base.init()
base[0][0]['x'] = X
base[0][0]['y'] = Y
base[0][0]['myvar'] = X*Y # create a dummy variable
base[0][0]['myvec_x'] = X+Y # create a vector variable (x component)
base[0][0]['myvec_y'] = np.zeros_like(X) # create a vector variable (y component)

base.coordinate_names = ['x', 'y']

# Dump base to rotate
w = antares.Writer('hdf_antares')
w['base'] = base
w['filename'] = os.path.join(output_folder, 'base_to_rotate')
w.dump()

# Apply rotation treatment
rotation = antares.Treatment('rotation')
rotation['base'] = base
rotation['angle'] = np.pi / 4
rotation['vectors'] = [ ['myvec_x', 'myvec_y'] ]
rotated_base = rotation.execute()

# Dump result
w = antares.Writer('hdf_antares')
w['base'] = rotated_base
w['filename'] = os.path.join(output_folder, 'rotated_base')
w.dump()
```



Scaling

Description

Scales base's geometry with respect to a factor and an origin location.

Parameters

- **base: `Base`**
  The input base to be scaled.
- **coordinates:** ***list(str)*****, default=** ***None***
  The variable names that define the set of coordinates. If None, the default coordinate system variables is used. The coordinate system must be the Cartesian coordinate system.
- **factor: class:*****seq(float)***
  The scaling factor for each of the axis of the base.
- **origin: class:*****seq(float)*****, default=** ***None***
  Point used as reference for the scaling. If None, the origin of the coordinate system is used.
- **memory_mode:** ***bool*****, default=** ***False***
  If True, the modifications are done directly on the input base to limit memory usage. If False, a new base is created.

Preconditions

- The base must be in the Cartesian system of coordinates

Postconditions

- If **memory_mode** = False, the input base remains unchanged and the output base has the same structure as the input base. If **memory_mode** = True, the input base is modified in-place and the returned base is the same object as the input base.

Example

The following example shows how to scale a base by a factor of 2 in the x direction, by a factor 0.6 in the y direction and leave it unchanged in the z direction.

```
import antares
myt = antares.Treatment('scaling')
myt['base'] = base
myt['coordinates'] = ['x', 'y', 'z']
myt['factor'] = [2, 0.6, 1]
scaled_base = myt.execute()
```

Main functions

class antares.treatment.TreatmentScaling.**TreatmentScaling**

execute()

Scales the base.

Returns
the base containing the results.

Return type
Base

Example

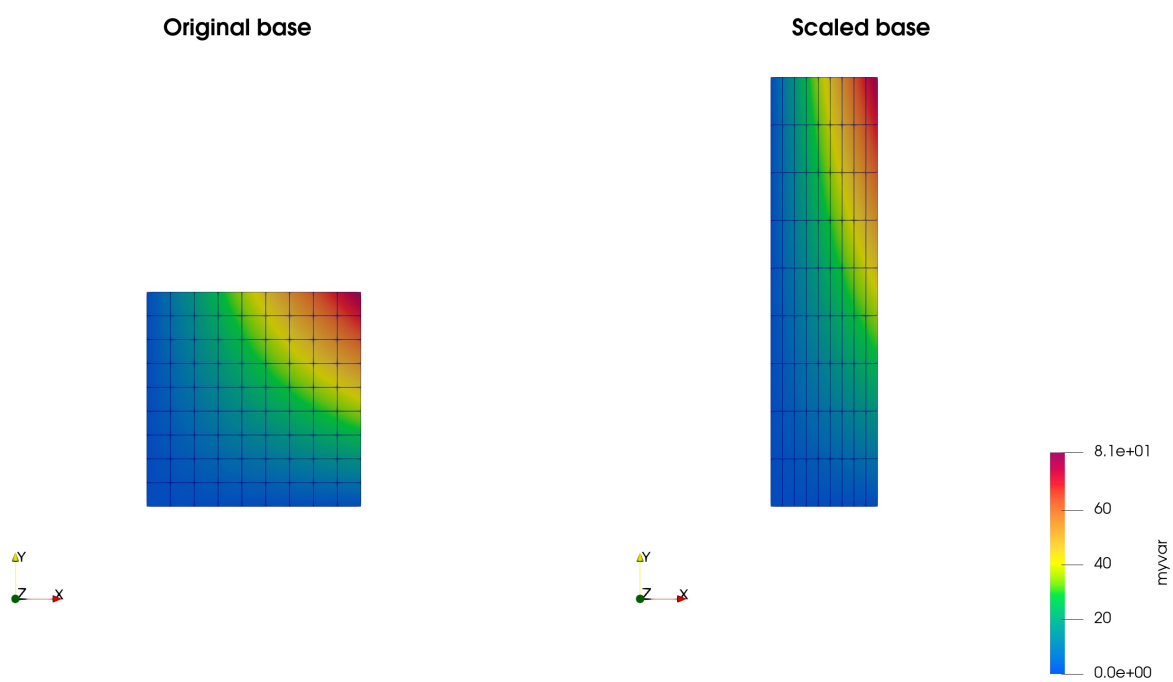
The following examples creates a 2D structured base with a scalar variable and it uses the scaling treatment to scale the base by a factor of 0.5 in the x direction (scaling down) and by a factor of 2 in the y direction (scaling up).

```
import antares
import numpy as np
import os

output_folder = os.path.join("OUTPUT", "TreatmentScaling")
os.makedirs(output_folder, exist_ok=True)

X, Y = np.mgrid[0:10, 0:10]

# Create base to translate
```

(continues on next page)

(continued from previous page)

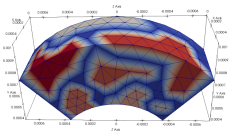
```
base = antares.Base()
base.init()
base[0][0]['x'] = X
base[0][0]['y'] = Y
base[0][0]['myvar'] = X*Y # create a dummy variable
base.coordinate_names = ['x', 'y']

# Dump base to translate
w = antares.Writer('hdf_antares')
w['base'] = base
w['filename'] = os.path.join(output_folder, 'base_to_scale')
w.dump()

# Apply rotation treatment
scaling = antares.Treatment('scaling')
scaling['base'] = base
scaling['factor'] = [0.5, 2]
scaled_base = scaling.execute()

# Dump result
w = antares.Writer('hdf_antares')
w['base'] = scaled_base
w['filename'] = os.path.join(output_folder, 'scaled_base')
w.dump()
```



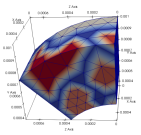
Clip

Description

Clip a grid (structured of unstructured) using a given clipping geometrical shape.



==>



Initial unstructured multi-element mesh on the left. Clipping with a plane on the right.

Parameters

- **base:** `Base`
  The input base to be clipped.
- **coordinates:** ***list(str)***
  The variable names that define the set of coordinates.
- **type:** ***str*****, default=** ***'plane'***
  The type of geometrical shape used to clip. The values are: - plane - cone - cylinder - sphere - revolution

  If none of the above, you can also set a VTK functions as a value.
- **origin:** ***tuple(float)*****,**
  The coordinates (tuple of 3 floats) of the origin used to define the clipping shape. Used for: - plane - cone - cylinder - sphere
- **normal:** ***tuple(float)*****,**
  The coordinates (tuple of 3 floats) of the normal vector used to define a clipping plane.
- **angle:** ***float*****,**
  Angle (in radian) of the clipping cone.
- **radius:** ***float*****,**
  Radius of a clipping cylinder or sphere.
- **line:** ***dict*****,**
  Definition of the line for the clipping of type 'revolution'.

  As an example, in a cylindrical coordinate system (x,r,t). Give the axial points as keys and radial points as values. The revolution will take place in the azimutal direction. The lowest and the greatest x-values will serve to set two x-constant clipping planes to only keep the domain in-between.
- **axis:** ***tuple(float)*****,**
  The coordinates (tuple of 3 floats) of the axis vector used to define a clipping cone or cylinder.
- **invert:** ***bool*****, default=** ***False***
  If True, invert the clipping domain.
- **memory_mode:** ***bool*****, default=** ***False***
  If True, the initial base is deleted on the fly to limit memory usage.
- **with_families:** ***bool*****, default=** ***False***
  If *True*, the output of the treatment contains rebuilt families based on the input **base**. All the families and

all the levels of sub-families are rebuilt, but only attributes and *Zone* are transfered (not yet implemented for *Boundary* and *Window*).

Preconditions

Zones may be either structured or unstructured.

Zones may contain multiple instants.

Postconditions

The output base is always unstructured. 3D bases only contain tetrahedral elements. 2D bases only contain triangle elements. If **with_families** is enabled, the output base contains the reconstructed families of **base**.

Example

The following example shows a clipping with a plane defined with the point (0., 0., 0.) and the normal vector (1., 0., 0.).

```
import antares
myt = antares.Treatment('clip')
myt['base'] = base
myt['type'] = 'plane'
myt['origin'] = [0., 0., 0.]
myt['normal'] = [1., 0., 0.]
clipped = myt.execute()
```

Note: dependency on VTK[114]

Main functions

class antares.treatment.TreatmentClip.**TreatmentClip**

execute()

Clip the geometry.

Returns
the unstructured Base obtained by clipping with a geometrical shape.

Return type
Base

[114] https://www.vtk.org/

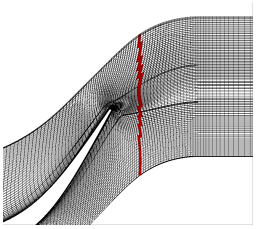
Example

```
"""
This example shows how to clip a base.

Note that even if the input base is structured, the output of the
clip treatment will be unstructured.
"""
import os

import antares

OUTPUT = 'OUTPUT'
if not os.path.isdir(OUTPUT):
    os.makedirs(OUTPUT)

# ------------------
# Reading the files
# ------------------
reader = antares.Reader('bin_tp')
reader['filename'] = os.path.join('..', 'data', 'ROTOR37', 'GENERIC', 'flow_<zone>_
↪<instant>.dat')
ini_base = reader.read()

# -----------------------
# Clipping of type plane
# -----------------------
# For this clipping the dataset is cut by a plane and only the
# cells under the plane are kept (under/over is defined by the
# normal orientation).
# Note that the output dataset present a smooth plane cut at the
# clipping position
treatment = antares.Treatment('clip')
treatment['base'] = ini_base
treatment['type'] = 'plane'
treatment['origin'] = [70., 0., 0.]
treatment['normal'] = [1., 0., 0.]
result = treatment.execute()

# -------------------
# Writing the result
# -------------------
writer = antares.Writer('bin_tp')
writer['filename'] = os.path.join(OUTPUT, 'ex_clip_plane.plt')
writer['base'] = result
writer.dump()
```

CrinkleSlice

Description

Keep cells with the given value.



Initial grid in black and crinkle slice for x=70 in red.

Construction

```
import antares
myt = antares.Treatment('crinkleslice')
```

Parameters

- **base: `Base`**
  The input base
- **variable: *str*, default= *None***
  The name of the variable used to extract cells.
- **value: *float***
  Keep cells with the given value.
- **memory_mode: *bool*, default= *False***
  If True, the initial base is deleted on the fly to limit memory usage
- **with_families: *bool*, default= *False***
  If *True*, the output of the treatment contains rebuilt families based on the input **base**. All the families and all the levels of sub-families are rebuilt, but only attributes and *Zone* are transfered (not yet implemented for *Boundary* and *Window*).

Preconditions

Postconditions

The output base is always unstructured. It does not contain any boundary condition.

The input base is made unstructured.

If **with_families** is enabled, the output base contains the reconstructed families of **base**.

Example

```
import antares
myt = antares.Treatment('crinkleslice')
myt['base'] = base
myt['variable'] = 'x'
myt['value'] = 70.
crinkle = myt.execute()
```

Main functions

class antares.treatment.TreatmentCrinkleSlice.**TreatmentCrinkleSlice**

execute()

Execute the treatment.

Returns
the unstructured Base obtained after applying the treatment

Return type
Base

Example

```
"""
This example shows how to extract a crinkle slice.
"""
import os
if not os.path.isdir('OUTPUT'):
    os.makedirs('OUTPUT')

from antares import Reader, Treatment, Writer

# ------------------
# Reading the files
# ------------------
reader = Reader('bin_tp')
reader['filename'] = os.path.join('..', 'data', 'ROTOR37', 'GENERIC', 'flow_<zone>_ite0.
↪dat')
ini_base = reader.read()

# -------------------------
# Crinkle Slice
# -------------------------
treatment = Treatment('crinkleslice')
treatment['base'] = ini_base
treatment['variable'] = 'x'
treatment['value'] = 70.
result = treatment.execute()

# --------------------
```

(continues on next page)

(continued from previous page)

```
# Writing the result
# ------------------
writer = Writer('hdf_antares')
writer['filename'] = os.path.join('OUTPUT', 'ex_crinkleslice')
writer['base'] = result
writer.dump()
```

Cut (with VTK)

Description

Cut a grid (structured or unstructured) using a given geometrical shape. Cut using the VTK[115] library.

Parameters

- **base: `Base`**
  The input base to be cut.
- **coordinates:** ***list(str)***
  The variable names that define the set of coordinates. If no value is given, the default coordinate system of the base is used (see `Base.coordinate_names`).
- **nb_cuts:** ***int*****, default=** ***1***
  The desired number of cuts of the same kind. It defines the length of the list of the varying arguments. Performing multiple cuts at once, 'origin', 'normal', 'axis', 'angle', 'radius' become lists of parameters.
- **type:** ***str*****, default=** ***'plane'***
  The type of geometrical shape used to cut. The values are: - plane - cone - cylinder - sphere - revolution - spline - polyline - vtk user-defined function The cone is a right circular cone.
- **origin:** ***tuple(float) or list(tuple(float))***
  The coordinates of the origin used to define the cutting shape. Used for:
  - plane
  - cone
  - cylinder
  - sphere

  They must be coherent with **coordinates**. Tuple of 3 floats for a single cut, or list of tuples of 3 floats for a multiple cut. The **origin** must be inside the domain.
- **normal:** ***tuple(float) or list(tuple(float))***
  The coordinates of the normal vector used to define a cutting plane (**type='plane'**). Tuple of 3 floats for a single cut, or list of tuples of 3 floats for a multiple cut.
- **angle:** ***float or list(float)***
  Angle (in radian) of the cone at the apex for a cutting cone (**type='cone'**). Float for a single cut, or list of floats for a multiple cut.
- **radius:** ***float or list(float)***
  Radius of the cutting cylinder or sphere (**type='cylinder'** or **type='sphere'**). Float for a single cut, or list of floats for a multiple cut.

[115] https://www.vtk.org/

- **axis:** ***tuple(3*floats) or list(tuple(3*floats))***
  The axis of rotation for **type='cylinder'** or **type='cone'**. Tuple of 3 floats for a single cut, or list of tuples of 3 floats for a multiple cut.
  - example: **axis** =[0., 0., 1.] to set 'z' as the axis in 3D with the cartesian **coordinates** ['x', 'y', 'z']
- **line_points:** ***list(tuple(float))***
  List of point coordinates for **type='polyline'** or **type='spline'**. Definition points must be given in the same frame of reference as the base. The points must defined a line in the space made by the **coordinates** except the extrusion axis. These points form a line that is extruded in the third dimension with respect to **coordinates**. List of tuples of two/three floats for a single cut, or list of lists of tuples of two/three floats for a multiple cut. If two floats are given, the third coordinate is set to zero.
  - example: **line_points** =[(-4., 2.), (1., 2.)]. List of (x,r) points with 'theta' the extrusion axis with **coordinates** ['x', 'r', 'theta'].
  - **example:**
    * [(x,y),...] for a single cut.
    * [[(x1,y1),...], [(x2,y2),...],...] for a multiple cut.

  See also `antares.utils.CutUtils.parse_json_xrcut()` (page 395)
- **resolution:** ***int*, default=** ***50***
  The desired number of points used to discretize the cutter line from the point list defined with **line_points**. Note that it is only used for **type='spline'**.
- **line:** ***list(tuple(float))***
  List of point coordinates for **type='revolution'**. The frame of reference must be the cylindrical coordinate system. No checking is made on this point. The points of the **'line'** must be given in order. List of tuples of two floats for a single cut, or list of lists of tuples of two floats for a multiple cut.
- **expert:** ***str***
  Options dedicated to expert users. Values are ['tri_and_qua']. 'tri_and_qua': do not triangulate the cut surface. Keep original elements (only segments, triangles, and quadrilaterals). Other elements are triangulated.
- **memory_mode:** ***bool*, default=** ***False***
  If True, the initial base is deleted on the fly to limit memory usage.
- **with_boundaries:** ***bool*, default=** ***False***
  Whether or not to use data from the boundaries.
- **with_families:** ***bool*, default=** ***False***
  If *True*, the output of the treatment contains rebuilt families based on the input **base**. All the families and all the levels of sub-families are rebuilt, but only attributes and *Zone* are transfered (not yet implemented for *Boundary* and *Window*).

Preconditions

If shared coordinates, ALL coordinates must be in the shared instant.

The Base must contain at least one non shared Instant.

Requirements for cut type:

- cone: 'angle', 'axis', 'origin'
- cylinder: 'axis', 'origin', 'radius'
- plane: 'origin', 'normal'

- sphere: 'origin', 'radius'
- polyline: 'line_points'
- revolution: 'line'
- spline: 'line_points'

For multiple cuts, the value of atomic keys can be replaced with a list of values. The length of this list must be equal to **'nb_cuts'**.

Postconditions

If **with_families** is enabled, the output base contains the reconstructed families of **base**.

Example

The following example shows a cutting with a plane defined with the point (0., 0., 0.) and the normal vector (1., 0., 0.).

```
import antares
myt = antares.Treatment('cut')
myt['base'] = base
myt['type'] = 'plane'
myt['origin'] = [0., 0., 0.]
myt['normal'] = [1., 0., 0.]
cutbase = myt.execute()
```

Main functions

class antares.treatment.TreatmentCut.**TreatmentCut**

Process to perform a Cut treatment with VTK.

Boundaries are supported, and are created in the generated zone.

execute()

Slice the input base with the given options.

Returns
an unstructured base

Return type
Base

Examples

Look at examples in antares/examples/treatment:

- cut.py
- cut_sector.py
- multicut_cylinder.py
- multicut_plane.py
- multicut_revolution.py

- multicut_spline.py
- spline.py
- decimate.py
- turboglobalperfo.py
- unwrapline.py

Cut (without VTK)

Description

Cut a grid (structured of unstructured) using a given geometrical shape.

Construction

```
import antares
myt = antares.Treatment('acut')
```

Parameters

- **base: `Base`**
  The input base to be cut.
- **coordinates: *list(str)***
  The variable names that define the set of coordinates. If no value is given, the default coordinate system of the base is used.
- **nb_cuts: *int*, default= *1***
  The desired number of cuts of the same kind. It defines the length of the list of the varying arguments. Performing multiple cuts at once, 'origin', 'normal', 'axis', 'angle', 'radius' become lists of parameters.
- **type: *str*, default= *'plane'***
  The type of geometrical shape used to cut. The values are: - plane - cone - cylinder - sphere - polyline The cone is a right circular cone.
- **origin: *tuple(float)*,**
  The coordinates (tuple of 3 floats) of the origin used to define the cutting shape. Used for: - plane - cone - cylinder - sphere
- **normal: *tuple(float)*,**
  The coordinates (tuple of 3 floats) of the normal vector used to define a cutting plane (**type='plane'**).
- **angle: *float*,**
  Angle (in radian) of the cone at the apex for a cutting cone (**type='cone'**).
- **radius: *float*,**
  Radius of the cutting cylinder or sphere (**type='cylinder'** or **type='sphere'**).
- **axis: *tuple(float)*,**
  The coordinates of the axis of rotation of the cutting cylinder or cone (**type='cylinder'** or **type='cone'**).
  - example: **axis**=[0., 0., 1.] to set 'z' as the axis in 3D with the cartesian **coordinates** ['x', 'y', 'z']

The coordinates of the extrusion direction of the cutting polyline (**type='polyline'**)

- example: **axis**=[0., 0., 1.] to set 'theta' as the extrusion axis in 3D with the cylindrical **coordinates** ['x', 'r', 'theta']

- **line_points:** ***list(tuple(float))*****,**
  List of point coordinates for **type='polyline'**. Definition points must be given in the same frame of reference as the base. The points must defined a line in the space made by the **coordinates** except the extrusion axis.
  - example: **line_points**=[(-4., 2., 0.), (1., 2., 0.)]. If 'theta' is the extrusion axis with **coordinates** ['x', 'r', 'theta'], then the last coordinates of points must be the same.
- **expert:** ***dict {int: string, ...}*****, default=** ***{2: 'bi', 3: 'tri', 4: 'qua'}***
  Options dedicated to expert users. The dict is the set of accepted polygons in resulting mesh. Keys are the number of vertices in polygons. Values are the element types.
- **memory_mode:** ***bool*****, default=** ***False***
  If True, the initial base is deleted on the fly to limit memory usage.
- **cutter:** ***dict*****, default=** ***None***
  If not None, re-use the cutter from a previous cut.
- **return_cutter:** ***bool*****, default=** ***False***
  If True, the treatment also returns the cutters. The returned variable is a dictionary whose keys are the zone names and the values are a list of cutters of type `antares.utils.geomcut.cutter.Cutter` (page 394).
- **with_families:** ***bool*****, default=** ***False***
  If *True*, the output of the treatment contains rebuilt families based on the input **base**. All the families and all the levels of sub-families are rebuilt, but only attributes and *Zone* are transfered (not yet implemented for *Boundary* and *Window*).

Preconditions

If shared coordinates, ALL coordinates must be in the shared instant.

Requirements for cut type: - Cone: 'angle', 'axis', 'origin' - Cylinder: 'axis', 'origin', 'radius' - Plane: 'origin', 'normal' - Sphere: 'origin', 'radius' - Polyline: 'axis', 'line_points'

Postconditions

If **with_families** is enabled, the output base contains the reconstructed families of **base**.

Example

The following example shows a cutting with a plane defined with the point (0., 0., 0.) and the normal vector (1., 0., 0.).

```
import antares
myt = antares.Treatment('acut')
myt['base'] = base
myt['type'] = 'plane'
myt['origin'] = [0., 0., 0.]
myt['normal'] = [1., 0., 0.]
cutbase = myt.execute()
```

Main functions

class antares.treatment.TreatmentAcut.**TreatmentAcut**

Define a cut treatment that does not use the VTK library.

Be aware that the interface for this treatment may not be the same as for the VTK-based cut treatment.

execute()

Execute the treatment.

Returns
an unstructured cut

Return type
Base

Examples

Look at examples in antares/examples/treatment:

- multicut_cylinder.py
- multicut_plane.py
- polyline.py

Cut (multi-threaded)

Description

Cut a grid (structured of unstructured) using a given geometrical shape. This treatment is multithreaded and can be used on computers with shared memory nodes. It may be used in co-processing with a workflow based on MPI/threads paradigm.

Parameters

- **base: Base**
  The input base to be cut.
- **coordinates: seq(str)**
  The variable names that define the set of coordinates. If no value is given, the default coordinate system of the base is used.
- **nb_cuts: int, default=** ***1***
  The desired number of cuts of the same kind. It defines the length of the list of the varying arguments. Performing multiple cuts at once, 'origin', 'normal', 'axis' become lists of parameters. Only works when **type="plane"**.
- **type: str, default=** ***"plane"***
  The type of geometrical shape used to cut. The values are:
  - plane
  - cylinder
  - sphere

  - isosurface

- **origin:** `seq(float) or list(seq(float))`
  The coordinates (sequence of 3 floats) of the origin used to define the cutting shape. Used for:
  - plane
  - cylinder
  - sphere

  They must be coherent with **coordinates**. Sequence of 3 floats for a single cut, or list of sequences of 3 floats for a multiple cut.

- **normal:** `seq(float) or list(seq(float))`
  The coordinates of the normal vector used to define a cutting plane (**type="plane"**). Sequence of 3 floats for a single cut, or list of sequences of 3 floats for a multiple cut.

- **axis:** `seq(3*floats) or list(seq(3*floats))`
  The axis of rotation for **type='cylinder'** or **type='cone'**. Sequence of 3 floats for a single cut, or list of sequences of 3 floats for a multiple cut.
  - example: **axis** =[0., 0., 1.] to set 'z' as the axis in 3D with the cartesian **coordinates** ['x', 'y', 'z']

- **radius:** `float`
  Radius of the cutting cylinder or sphere (**type='cylinder'** or **type='sphere'**).

- **variable:** `str`
  For **type="isosurface"**, the name of the variable to make the iso-surface.

- **value:** `float`
  For **type="isosurface"**, the value of the variable to make the iso-surface.

- **cutter:** `antares.utils.mshcppcutter._cut_mesh.CutMesh`**, default=** ***None***
  If set, will use the object to interpolate the variables onto the cut mesh without recutting the original Base. Does nothing with **type="isosurface"**.

- **return_cutter:** `bool`**, default=** ***False***
  If set to True, will return the `CutMesh` object instead of a new base. The CutMesh object can then be used with the 'cutter' parameter to interpolate variable. Useful when cutting multiple the same Base in succession with the same cut, but with different variables. Does nothing with **type="isosurface"**.

- **nb_threads:** `int`**, default=** ***1***
  The number of threads the treatment will try to use for each cut.

- **ignore_unsupported_cells:** `bool`**, default=** ***False***
  If set to True, this will ignore any cell that are not supported in this Treatment, instead of throwing an error.

- **ignore_openMP_checks:** `bool`**, default=** ***False***
  If set to True, this will prevent the Treatment to raise an Error when nb_threads is greater then 1, but the library was not compiled with openMP.

- **skip_checks:** `bool`**, default=** ***False***
  If set to True, the Treatment will skip most parameter checks. This will make it faster, at the cost of a potential error with no clear backtrace. Use this option when performance is critical.

Preconditions

If shared coordinates, ALL coordinates must be in the shared instant. This treatment handles 3d meshes only.

Cases with only shared variables are not handled.

Requirements for cut type:

- Plane: 'origin', 'normal'
- Cylinder: 'axis', 'origin', 'radius'
- Sphere: 'origin', 'radius'
- Iso-surface: 'variable', 'value'

Example

The following example shows a cutting with a plane defined with the point (0., 0., 0.) and the normal vector (1., 0., 0.).

```
import antares
myt = antares.Treatment('ccut')
myt['base'] = base
myt['type'] = 'plane'
myt['origin'] = [0., 0., 0.]
myt['normal'] = [1., 0., 0.]
myt['n_threads'] = 16
cutbase = myt.execute()
```

Main functions

class antares.treatment.TreatmentCcut.**TreatmentCcut**

Define a cut treatment using a custom C++ library.

The interface for this treatment is mostly compatible with ACut.

cut_batch()

cut_once()

execute()

Perform the cutting process with multiple threads.

Returns
an unstructured cut

Return type
Base

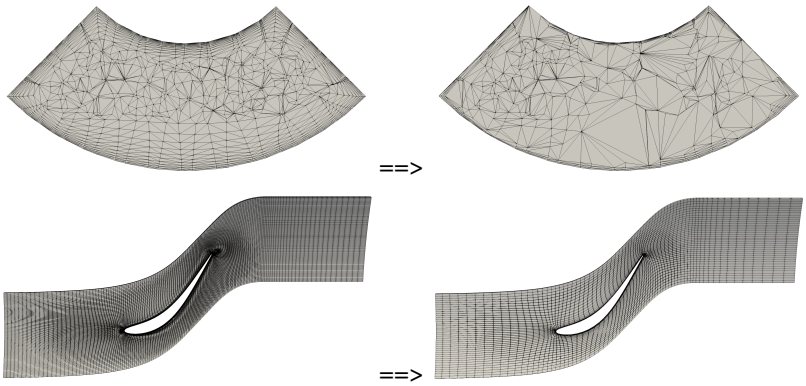
Decimate

Description

Decimation of a structured or 2D unstructured grid using a reduction target.

This treatment can be used to coarsen a mesh containing a large number of elements.

The decimation process of 2D unstructured grids uses the VTK[116] library to reduce the number of triangles.



On the left: initial 2D unstructured (top) and structured (bottom) meshes.

On the right: resulting meshes after decimation with **reduction** = 0.5 for the unstructured mesh and **reduction** = 2. for the structured mesh.

Parameters

- **base: `Base`**
  Base to decimate.
- **coordinates: *sequence(str)***
  The name of coordinates.
- **reduction: *float* in [0,1] or *int* > 1,**
  For unstructured grids (2D only), the factor of reduction (float) specifies the percentage of triangles to be removed. For structured grids, the factor of reduction (integer) is the number of points to remove in each topological direction. Example: If the mesh contains 100 elements, and the reduction factor is 0.75, then after the decimation there will be approximately 25 elements - a 75% reduction
- **memory_mode: *bool*, default= *False***
  If True, the initial base is deleted on the fly to limit memory usage.

[116] https://www.vtk.org/

Preconditions

Zones can be either structured or 2D unstructured.

Zones can contain multiple instants.

Postconditions

The boundaries and families are not preserved in the output base.

Example

This example shows how to decimate a base.

```
import antares
myt = Treatment('decimate')
myt['base'] = base
myt['coordinates'] = ['x', 'y', 'z']
myt['reduction'] = 0.5
myt['memory_mode'] = True
decimated_base = myt.execute()
```

Main functions

class antares.treatment.TreatmentDecimate.**TreatmentDecimate**

Decimation of meshes.

execute()

Decimate the input base.

Returns

Return type
Base

Example

```
"""
This example shows how to decimate a base.
This can be useful if you want to test some kinds of treatment
on large meshes.
"""
import os
if not os.path.isdir('OUTPUT'):
    os.makedirs('OUTPUT')

from antares import Reader, Treatment, Writer

# ------------------
# Reading the files
```

(continues on next page)

(continued from previous page)

```
# ------------------
r = Reader('hdf_avbp')
r['filename'] = os.path.join('..', 'data', 'SECTOR', 'hybrid_sector.mesh.h5')
r['shared'] = True
base = r.read()
r = Reader('hdf_avbp')
r['base'] = base
r['filename'] = os.path.join('..', 'data', 'SECTOR', 'hybrid_sector.sol.h5')
r.read()

treatment = Treatment('cut')
treatment['base'] = base
treatment['type'] = 'plane'
treatment['origin'] = [0.0002, 0.0006, 0.]
treatment['normal'] = [1., 0., 0.]
result = treatment.execute()

ini_size = result[0][0].connectivity['tri'].size
# -------------------------------------------------------
# Decimate the unstructured base with a reduction factor
# -------------------------------------------------------
treatment = Treatment('decimate')
treatment['base'] = result
treatment['reduction'] = 0.1
treatment['memory_mode'] = True
result = treatment.execute()

print("Effective reduction: ", (100.0*(ini_size-result[0][0].connectivity['tri'].size))/
↪ini_size, "%")
# -------------------
# Writing the result
# -------------------
w = Writer('bin_tp')
w['filename'] = os.path.join('OUTPUT', 'ex_uns_decimated.plt')
w['base'] = result
w.dump()

# ------------------
# Reading the files
# ------------------
reader = Reader('bin_tp')
reader['filename'] = os.path.join('..', 'data', 'ROTOR37', 'GENERIC', 'flow_<zone>_ite
↪<instant>.dat')
base = reader.read()

# -------------------------------------------------------
# Decimate the unstructured base with a reduction factor
# -------------------------------------------------------
treatment = Treatment('decimate')
treatment['base'] = base
treatment['reduction'] = 2
treatment['memory_mode'] = True
```

(continues on next page)

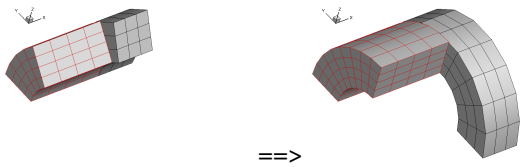
(continued from previous page)

```
result = treatment.execute()

# -------------------
# Writing the result
# -------------------
w = Writer('bin_tp')
w['filename'] = os.path.join('OUTPUT', 'ex_str_decimated.plt')
w['base'] = result
w.dump()
```

Duplication

Description

Duplicate zones of a mesh with rotation around an axis. Typically used in turbomachinery.



Initial mesh on the left. Duplicated mesh on the right with 1 rotation of $\pi/4$ for the front block and 2 rotations of $\pi/3$ for the back block.

Construction

```
import antares
myt = antares.Treatment('duplication')
```

Parameters

- **base: `Base`**
  The input base to be duplicated.
- **coordinates:** ***list(str)***
  The variable names that define the set of coordinates. The coordinate system must be the cartesian coordinate system. The first coordinate is taken as the rotational axis.
- **vectors:** ***tuple/list(tuple(str))*****, default=** ***[]***
  Coordinate names of vectors that need to be rotated during the duplication. It is assumed that these are given in the cartesian coordinate system.
- **nb_duplication:** ***int or tuple(int)*** **or in_attr, default=** ***'in_attr'***
  Number of duplications if an integer is given. Range of duplications if a tuple is given (from first element to second element included). If **in_attr**, then each zone of the base must have an attribute **nb_duplication**.
- **pitch:** ***int or float*** **or in_attr, default=** ***'in_attr'***
  Angular sector of the mesh if a scalar is given. If **in_attr**, then each zone of the base must have an attribute **pitch**.

- **axis:** ***str*, default= *'x'***
  Name of the rotation axis. Must be in **coordinates**.
- **omega:** ***int or float* or in_attr, default= *0.***
  Rotation speed expressed in radians per second. If **time** is given, then **omega** * **time** is added to the rotation angle. The input mesh is supposed to correspond to **time** =0. If **in_attr**, then each zone of the base must have an attribute **omega**.
- **time:** ***float*, default= *0.***
  Time that corresponds to the flow field. If the flow field comes from a computation that is performed in a relative reference frame, then the input mesh is supposed to correspond to **time** =0, and does not change during the computation. The **time** and rotation speed **omega** of the mesh are needed to retrieve the mesh in the absolute reference frame.

Preconditions

Zones may be either structured or unstructured.

Zones must only contain one instant. Multi-instant base is not supported.

Postconditions

The output base is the assembly of the input base with **nb_duplication**-1 more bases that have been duplicated from the input base. The name of the duplicated zones is composed of the input zone name with a suffix '_DUP' followed by the number of the duplication. This number has a leading zero for all numbers with less than 2 digits.

> **Warning:** In the duplicated (rotated) zones, the coordinate or vector variables are clearly not the same that the ones of the input zones. If other variables computed from the coordinate or vector variables are present in the input base, then they will **not** be set correctly in the duplicated base.
>
> e.g.: suppose that ['x', 'y', 'z'] is the coordinate system, and a=f(y), then 'a' will not be set correctly in the duplicated base.

The boundary conditions are also duplicated.

The families that contains only zones or only boundaries are duplicated.

Example

The following example shows a duplication of a base with an angular sector of $\pi/18$ around the rotation axis 'x' with the velocity vector to be rotated.

```
import antares
myt = antares.Treatment('duplication')
myt['base'] = base
myt['vectors'] = [('vx', 'vy', 'vz')]
myt['nb_duplication'] = 2
myt['pitch'] = np.pi / 18.
dupbase = myt.execute()
```

Main functions

class antares.treatment.TreatmentDuplication.**TreatmentDuplication**

execute()

Duplicate the base.

Returns
the base containing the results.

Return type
Base

Example

```
"""
This example illustrates the duplication of
a periodic field (for example turbomachinery
computations).
"""
import os
if not os.path.isdir('OUTPUT'):
    os.makedirs('OUTPUT')


import numpy as np

from antares import Reader, Treatment, Writer

# ------------------
# Reading the files
# ------------------
r = Reader('bin_tp')
r['filename'] = os.path.join('..', 'data', 'ROTOR37', 'ELSA_CASE', 'MESH', 'mesh_<zone>.
↪dat')
r['zone_prefix'] = 'Block'
r['topology_file'] = os.path.join('..', 'data', 'ROTOR37', 'ELSA_CASE', 'script_topo.py')
r['shared'] = True
r['location'] = 'node'
base = r.read()

r = Reader('bin_tp')
r['base'] = base
r['filename'] = os.path.join('..', 'data', 'ROTOR37', 'ELSA_CASE', 'FLOW', 'flow_<zone>.
↪dat')
r['zone_prefix'] = 'Block'
r['location'] = 'cell'
r.read()
# ------------------
# Duplication
# ------------------
treatment = Treatment('duplication')
treatment['base'] = base
```

(continues on next page)

(continued from previous page)

```
treatment['vectors'] = [('rovx', 'rovy', 'rovz')]
treatment['nb_duplication'] = 2
treatment['pitch'] = 2. * np.pi / 36.
result = treatment.execute()
# -------------------
# Writing the result
# -------------------
writer = Writer('bin_tp')
writer['filename'] = os.path.join('OUTPUT', 'ex_duplication.plt')
writer['base'] = result
writer['memory_mode'] = True
writer.dump()
```

ExtractBounds

Extract the bounds of each zone.

Parameters

- **base: `Base`**
  The Base on which the extraction will be performed.
- **faces: *bool*, default= *True***
  If True it extracts the cell faces otherwise it extracts the cells.
- **memory_mode: *bool*, default= *False***
  If memory mode is True, the base is deleted on 'the fly to limit memory usage.

Preconditions

The treatment is only coded for unstructured dataset so far. See function `Base.unstructure()` if needed to be used on a structured base.

Main functions

class `antares.treatment.TreatmentExtractBounds.`**`TreatmentExtractBounds`**

`execute()`

Execute the treatment.

Returns
the unstructured Base obtained

Return type
`Base`

Example

```
"""
This example shows how to extract the bounding cells of each zone
of a base.

Note that even if the input base is structured, the output of the
treatment will be unstructured.
"""
import os
if not os.path.isdir('OUTPUT'):
    os.makedirs('OUTPUT')

from antares import Reader, Treatment, Writer

# ------------------
# Reading the files
# ------------------
reader = Reader('bin_tp')
reader['filename'] = os.path.join('..', 'data', 'ROTOR37', 'GENERIC', 'flow_<zone>_
↪<instant>.dat')
base = reader.read()

# ------------------------
# Extract bounds
# ------------------------
# treatment is only coded for unstructured dataset so far
base.unstructure()

treatment = Treatment('extractbounds')
treatment['base'] = base
result = treatment.execute()

print(result[0][0])

# -------------------
# Writing the result
# -------------------
writer = Writer('hdf_antares')
writer['filename'] = os.path.join('OUTPUT', 'ex_bounds')
writer['base'] = result
writer.dump()
```

Isosurface

Description

Perform an isosurface using VTK.

On the left: initial unstructured multi-element mesh and ρu_x field.

On the right: resulting unstructured mesh after Isosurface Treatment with **value** = 0.

Parameters

- **base:** `Base`
  The base on which the isosurfacing will be applied.
- **coordinates:** ***list(str)*, default= <*base*>.*coordinate_names***
  The variable names that define the set of coordinates. The default coordinates comes from the input base attribute *coordinate_names*.
- **variable:** ***str*,**
  Name of the variable to make the isosurface.
- **value:** ***float* or *list(float)*,**
  Value of the isosurface or list of values for isosurfaces.
- **with_families:** ***bool*, default= *False***
  If *True*, the output of the treatment contains rebuilt families based on the input **base**. All the families and all the levels of sub-families are rebuilt, but only attributes and *Zone* are transfered (not yet implemented for *Boundary* and *Window*).
- **output_element:** ***str***
  Options dedicated to expert users. Values are ['triangle']. 'triangle': do triangulate the isosurface. All elements are triangulated. Only used when polyhedral input.

Preconditions

Coordinates and variables must be available at nodes.

Zones may be either structured or unstructured.

Zones may contain multiple instants.

Postconditions

The output base is always unstructured. If **with_families** is enabled, the output base contains the reconstructed families of **base**.

Example

This example shows how to extract an isosurface. To be able to use this functionnality you must have vtk installed.

```
import antares

myt = Treatment('isosurface')
myt['base'] = base
myt['variable'] = 'Ux'
myt['value'] = 0.
base_iso = myt.execute()
```

Main functions

class antares.treatment.TreatmentIsosurface.**TreatmentIsosurface**

Class for isosurfaces.

execute()

Create isosurfaces.

Returns
an unstructured base or a list of unstructured bases sorted as the list of provided iso values.

Return type
Base or list(Base)

Example

```
"""
This example shows how to extract an isosurface.
To be able to use this functionnality you must have vtk installed.
"""
import os
if not os.path.isdir('OUTPUT'):
    os.makedirs('OUTPUT')

from antares import Reader, Treatment, Writer

# ------------------
# Reading the files
# ------------------
reader = Reader('bin_tp')
reader['filename'] = os.path.join('..', 'data', 'ROTOR37', 'GENERIC', 'flow_<zone>_ite0.
↪dat')
ini_base = reader.read()
```

(continues on next page)

(continued from previous page)

```
# -----------
# Isosurface
# -----------
treatment = Treatment('isosurface')
treatment['base'] = ini_base
treatment['variable'] = 'rovx'
treatment['value'] = 0.
result = treatment.execute()

# -------------------
# Writing the result
# -------------------
writer = Writer('bin_tp')
writer['filename'] = os.path.join('OUTPUT', 'ex_isosurface.plt')
writer['base'] = result
writer.dump()

# -----------
# Isosurfaces
# -----------
treatment = Treatment('isosurface')
treatment['base'] = ini_base
treatment['variable'] = 'rovx'
treatment['value'] = [0., 0.01]
result = treatment.execute()

for num, base in enumerate(result):
    writer = Writer('bin_tp')
    writer['filename'] = os.path.join('OUTPUT', 'ex_isosurface{}.plt'.format(num))
    writer['base'] = base
    writer.dump()
```

Line

Description

Interpolate data over a line using the VTK[117] library.

[117] https://www.vtk.org/

Parameters

- **base:** `Base`
  The input base containing the data to extract over the line.
- **coordinates:** *list(str)*
  The variable names that define the set of coordinates. If no value is given, the default coordinate system of the base is used (see `Base.coordinate_names`).
- **point1:** *tuple(float)*
  The coordinates of the starting point of the line.
- **point2:** *tuple(float)*
  The coordinates of the end point of the line.
- **nbpoints:** *int*
  Number of points on the line.
- **memory_mode:** *bool*, **default=** *False*
  If True, the initial base is deleted on the fly to limit memory usage.
- **mask:** *bool*, **default=** *True*
  If False, the points that were not interpolated on the ‘ line are not deleted in the output base.
- **probe_tolerance:** *int* **in [0, . . . , 14], default=** *0*
  If different from 0, then set the VTK probe tolerance. Increasing probe tolerance could be really expensive.

Preconditions

Postconditions

The output base will only contain data located at nodes.

Example

```
import antares
myt = antares.Treatment('line')
myt['base'] = base
myt['coordinates'] = ['x', 'y', 'z']
myt['point1'] = [0.0, 0.0, 0.0]
myt['point2'] = [1.0, 1.0, 1.0]
myt['nbpoints'] = 100
line = myt.execute()
```

Main functions

class antares.treatment.TreatmentLine.**TreatmentLine**

execute()

Interpolate values on the line.

Returns

an unstructured base over a line

Return type

Base

Example

```
import os

if not os.path.isdir("OUTPUT"):
    os.makedirs("OUTPUT")

from antares import Reader, Treatment, Writer

# ------------------
# Reading the files
# ------------------
reader = Reader("bin_tp")
reader["filename"] = os.path.join(
    "..", "data", "ROTOR37", "GENERIC", "flow_<zone>_ite0.dat"
)
base = reader.read()
print(base)

base.compute_coordinate_system(current_coord_sys=["x", "y", "z"])
base.coordinate_names = ["x", "r", "theta"]

# ------------------
# Define line
# ------------------
nb = 20
p1 = [-10.0, 180.0, 0.0]
p2 = [95.0, 180.0, 0.34]

# ------------------------
# Interpolate on the line
# ------------------------
line = Treatment("line")
line["base"] = base
line["point1"] = p1
line["point2"] = p2
line["nbpoints"] = nb
line["memory_mode"] = True
line["mask"] = True
# line['probe_tolerance'] = 10
```

(continues on next page)

(continued from previous page)

```
new_base = line.execute()

print(new_base)
print(new_base[0])
print(new_base[0][0])

# ------------------------
# Write result
# ------------------------
w = Writer("column")
w["base"] = new_base
w["filename"] = os.path.join("OUTPUT", "ex_line.dat")
w.dump()
```

Merge

Description

Merge Zones of a given input Base.

==>

Multiple zones are shown on the left. A single zone is the result on the right.

Parameters

- **base: `Base`**
  The input base.
- **duplicates_detection:** ***bool*****, default=** ***False***
  Activate the detection of duplicated nodes. If True, the treatment will remove the points that are duplicated in the various zones. If some elements (or cells) are duplicated, then they will remain in the output connectivity. The duplicated elements are not removed. These may lead to get a larger number of cells than expected.

  If the coordinates are not shared, then the detection is performed only on the first Instant and applied to all the other Instants because the threshold could give different numbers of nodes per Instant. But as all the Instants of one Zone must have the same shape, it would lead to a shape AssertionError.

  Note that it is memory and time consuming.
- **coordinates:** ***list(str)***
  The variable names that define the set of coordinates used for duplicate detection **duplicates_detection** (not used otherwise).

- **tolerance_decimals:** ***int*****, default=** ***None***
  Number of decimal places to round the coordinates. Used with **duplicates_detection**. If negative, it specifies the number of positions to the left of the decimal point. If None, the coordinates will not be rounded.
- **memory_mode:** ***bool*****, default=** ***False***
  If True, the initial base is deleted on the fly to limit memory usage.
- **check_variables:** ***bool*****, default=** ***True***
  If True, only the variables common to all instants are merged. The common sets of variables is given by the instants of the first zone. If False, the same variables must be available in all instants, and if it is not respected, then this leads to an Error.

Preconditions

Zones may be either structured or unstructured.

Zones may contain multiple instants. It is supposed that all Zones contain the same number and name of Instants.

Zones must have Boundaries with different Names. If two zones have a boundary with the same name, then only one is kept. This may lead to a corrupted base.

Postconditions

The treatment returns a Base with only one Zone, and as many instants as the input base.

The output base is always unstructured.

The output base contains the same families as the input base.

The unique zone of the output base contains all the boundaries of the input base, except the grid connectivities of type 'abutting_1to1'.

Boundaries are not merged. (same topology)

The output base only contains the attributes from the input base. The zones and instants do not contain any attribute in the `AttrsManagement` object.

Example

```
import antares
myt = antares.Treatment('merge')
myt['base'] = base
myt['duplicates_detection'] = True
myt['tolerance_decimals'] = 13
merged = myt.execute()
```

Main functions

class antares.treatment.TreatmentMerge.**TreatmentMerge**

execute()

Merge zones.

Returns

The unstructured Base obtained after merging all Zones (Instant by Instant).

Return type

Base

Example

```
"""
This example illustrates how to use the merge treatment to join zones
of a multi-zone base.
"""
import os

from antares import Reader, Treatment

# Reading the files
reader = Reader('bin_tp')
reader['filename'] = os.path.join('..', 'data', 'ROTOR37', 'GENERIC', 'flow_<zone>_ite
↪<instant>.dat')
reader['zone_prefix'] = 'Block'
reader['topology_file'] = os.path.join('..', 'data', 'ROTOR37', 'GENERIC', 'script_
↪topology.py')
base = reader.read()

print(base)
print('Nb input base grid points: ', base.grid_points)
print('Families: ', sorted(base.families))
print('Boundaries of first zone: ', sorted(base[0].boundaries))

merge = Treatment('merge')
merge['base'] = base
merge['duplicates_detection'] = True
merged = merge.execute()

print(merged)
# the number of points is smaller  because of the duplicates removed
print('Nb merged base grid points: ', merged.grid_points)
print('Families: ', sorted(merged.families))
print('Boundaries of merged base: ',sorted(merged[0].boundaries))
```

Tetrahedralize

Description

Subdivide 3D mesh cells into tetrahedra, or 2D mesh cells into triangles.

When the base contains polygons or polyhedra, VTK must be installed.

Parameters

- **base:** `Base`
  The input base to be tetrahedralized.
- **coordinates:** *list(str)*
  The variable names that define the set of coordinates. If no value is given, the default coordinate system of the base is used. Only useful if the base contains polyhedra or polygons.

Preconditions

The input base may be structured or unstructured. It remains unchanged during treatment. If it is structured, then it is first deepcopied, and the copy is made unstructured. So, the memory footprint will be roughly three times the initial one.

When the base contains polyhedra, the cells must be convex and the faces planar. Generally, these conditions are not fulfilled because of numerical issues, but it is often not a problem.

Postconditions

A new unstructured base is returned.

> **Warning:** The extensive variables that are at cell centers in the input base are interpolated to order 0. It is likely that they are not coherent with the mesh of the output base. (e.g. volume, surface, and normal vector)

Example

```
import antares
myt = antares.Treatment('tetrahedralize')
myt['base'] = base
tetra_base = myt.execute()
```

Main functions

class antares.treatment.TreatmentTetrahedralize.**TreatmentTetrahedralize**

Treatment used to build a tetrahedralization of a mesh.

Principle: a quadrilateral face is subdivided into two triangular faces by the diagonal issued from the from the smallest vertices of the face.

execute()

Build the tetrahedralization of a mesh.

Returns
a base with tetrahedral elements.

Return type
Base

Example

```
"""
This example shows how to transform all elements of a base in tetrahedra.
"""
import os

from antares import Reader, Treatment, Writer

# ------------------
# Reading the files
# ------------------
r = Reader('hdf_avbp')
r['filename'] = os.path.join('..', 'data', 'SECTOR', 'hybrid_sector.mesh.h5')
r['shared'] = True
base = r.read()
r = Reader('hdf_avbp')
r['base'] = base
r['filename'] = os.path.join('..', 'data', 'SECTOR', 'hybrid_sector.sol.h5')
r.read()

treatment = Treatment('tetrahedralize')
treatment['base'] = base
result = treatment.execute()

result.show()
# ------------------
# Reading the files
# ------------------
reader = Reader('bin_tp')
reader['filename'] = os.path.join('..', 'data', 'ROTOR37', 'GENERIC', 'flow_<zone>_ite
↪<instant>.dat')
base = reader.read()

treatment = Treatment('tetrahedralize')
treatment['base'] = base
```

(continues on next page)

(continued from previous page)

```
result = treatment.execute()

result.show()

r = Reader('bin_tp')
r['filename'] = os.path.join('..', 'data', 'ROTOR37', 'ELSA_CASE', 'MESH', 'mesh_<zone>.
↪dat')
r['zone_prefix'] = 'Block'
r['topology_file'] = os.path.join('..', 'data', 'ROTOR37', 'ELSA_CASE', 'script_topo.py')
r['shared'] = True
base = r.read()

r = Reader('bin_tp')
r['base'] = base
r['filename'] = os.path.join('..', 'data', 'ROTOR37', 'ELSA_CASE', 'FLOW', 'flow_<zone>.
↪dat')
r['zone_prefix'] = 'Block'
r['location'] = 'cell'
base = r.read()

treatment = Treatment('tetrahedralize')
treatment['base'] = base
result = treatment.execute()

result.show()
```

—

Utilitary class and methods

class antares.treatment.TreatmentTetrahedralize.**Tetrahedralizer**(*dim*, *connectivity*)

Build the tetrahedralisation of a mesh.

interpolate(*value*, *location*)

Returns
interpolated value for the tetrahedral mesh.

interpolate_cell(*value*)

Returns
interpolated cells values (order 0)

interpolate_node(*value*)

Returns
node values

property connectivity

Returns
tetrahedral connectivity.

Return type
CustomDict

property src_cell

Returns
mapping between new cells and parent cells.

antares.utils.geomcut.tetrahedralizer.**tri2tri**(*tri_conn*)

Return a dummy subdivision for triangles.

antares.utils.geomcut.tetrahedralizer.**qua2tri**(*qua_conn*)

Subdivide quadrilateral cells into triangles.

Parameters
qua_conn – a quadrilateral cell-based connectivity

Returns

out_conn, src_cell

- out_conn: the resulting triangular connectivity.
- src_cell: the cell from which the new cell was generated.

antares.utils.geomcut.tetrahedralizer.**tet2tet**(*tet_conn*)

Return a dummy subdivision for tetrahedra.

antares.utils.geomcut.tetrahedralizer.**pyr2tet**(*pyr_conn*)

Subdivide pyramidal cells into tetrahedra.

Parameters
pyr_conn – a pyramidal cell-based connectivity.

Returns

out_conn, src_cell

- out_conn: the resulting tetrahedral connectivity.
- src_cell: the cell from which the new cell was generated.

antares.utils.geomcut.tetrahedralizer.**pri2tet**(*pri_conn*)

Subdivide prismatic cells into tetrahedra.

Parameters
pri_conn – a prismatic cell-based connectivity.

Returns

out_conn, src_cell

- out_conn: the resulting tetrahedral connectivity.
- src_cell: the cell from which the new cell was generated.

antares.utils.geomcut.tetrahedralizer.**hex2tet**(*hex_conn*)

Subdivide hexahedral cells into tetrahedra.

Parameters
hex_conn – a hexahedral cell-based connectivity.

Returns

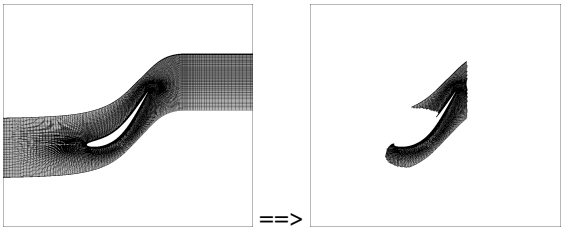
out_conn, src_cell

- out_conn: the resulting tetrahedral connectivity.
- src_cell: the cell from which the new cell was generated.

Threshold

Description

Keep grid points or cells that respect the given threshold values.



Initial grid shown on the left. Grid after threshold on the right.

Construction

```
import antares
myt = antares.Treatment('threshold')
```

Parameters

- **base: `Base`**
  The input base
- **variables:** ***list(str)***
  The name of variables on which the threshold is applied.
- **threshold:** ***list(tuple)***
  List of tuples of values (min, max) for each threshold variable. If a value is None, it will not be taken into account. So, **variables** must be greater or equal than min, **type** operator **variables** must be lesser or equal than max.

  example: if $4 < x < 5$, then **threshold** = [(3.0, 3.5)] will test

  - if $3.0 < x$
  - if $x < 3.5$
  - the **type** operator will be applied between the two previous conditions
- **type:** ***str*****, default=** ***and***
  Have all conditions simultaneously fulfilled (*and*) or at least one condition (*or*).
- **invert:** ***bool*****, default=** ***False***
  Keep points or cells that are located outside the given threshold range. In other words, the bitwise_not operator is applied after each **threshold** have been applied on all **variables**. The bitwise_not operator is not applied on each tuple.
- **memory_mode:** ***bool*****, default=** ***False***
  If True, the initial base is deleted on the fly to limit memory usage
- **with_families:** ***bool*****, default=** ***False***
  If *True*, the output of the treatment contains rebuilt families based on the input **base**. All the families and all the levels of sub-families are rebuilt, but only attributes and *Zone* are transfered (not yet implemented for *Boundary* and *Window*).

Preconditions

With data located at nodes, the threshold variable must be continuous in the given domain of definition (i.e. the base)

e.g.: You can not go from one point with a value π to its neighbour that has a value $-\pi$.

If the threshold variables are not shared, the threshold is based on the values of the threshold variables from the first Instant for each Zone. Indeed, the threshold could give different numbers of nodes per Instant, but as all the Instants of one Zone should have the same shape, it would lead to a shape AssertionError.

Postconditions

The output base is always unstructured. It does not contain any boundary condition.

The input base is made unstructured.

If **with_families** is enabled, the output base contains the reconstructed families of **base**.

Example

```
import antares
myt = antares.Treatment('threshold')
myt['base'] = base
myt['variables'] = ['x', 'ro']
myt['threshold'] = [(50., 70.), (None, 0.9)]
thres = myt.execute()
```

Main functions

class antares.treatment.TreatmentThreshold.**TreatmentThreshold**

execute()

Execute the threshold treatment.

Returns
The unstructured Base obtained after applying the threshold.

Return type
Base

Example

```
"""
This example shows how to apply a threshold on a base.

Note that even if the input base is structured, the output of the
threshold treatment will be unstructured.
"""
import os
if not os.path.isdir('OUTPUT'):
    os.makedirs('OUTPUT')
```

(continues on next page)

(continued from previous page)

```
from antares import Reader, Treatment, Writer

# ------------------
# Reading the files
# ------------------
reader = Reader('bin_tp')
reader['filename'] = os.path.join('..', 'data', 'ROTOR37', 'GENERIC', 'flow_<zone>_
↪<instant>.dat')
ini_base = reader.read()

# ------------------------
# Threshold
# ------------------------
# Only the cells which respect the given threshold condition are kept
# in the result base. Note that the output dataset present a crinkly
# interface at the threshold position
treatment = Treatment('threshold')
treatment['base'] = ini_base
treatment['variables'] = ['x', 'ro']
treatment['threshold'] = [(None, 70.), (None, 0.9)]
result = treatment.execute()

# -------------------
# Writing the result
# -------------------
writer = Writer('bin_tp')
writer['filename'] = os.path.join('OUTPUT', 'ex_threshold.plt')
writer['base'] = result
writer.dump()
```

Unstructure

Description

Transform a structured mesh into an unstructured mesh.

Parameters

- **base: Base**
  The input base to be unstructured.
- **boundary: *bool*, default= *True***
  Process boundary conditions of the input base.

Preconditions

None.

Postconditions

This treatment processes the input base in-place. The input base becomes an unstructured base. 3D bases will contain hexahedral elements and connectivities. 2D bases will contain quadrangle elements and connectivities.

Example

```
import antares
myt = antares.Treatment('unstructure')
myt['base'] = base
myt.execute()
```

Main functions

class antares.treatment.TreatmentUnstructure.**TreatmentUnstructure**

execute()

Transform a structured mesh into an unstructured mesh.

Process also the boundaries.

Returns
an unstructured Base. This is for backward compatibility (in-place treatment).

Return type
Base

Example

```
import os

import antares

if not os.path.isdir('OUTPUT'):
    os.makedirs('OUTPUT')

r = antares.Reader('bin_tp')
r['filename'] = os.path.join('..', 'data', 'ROTOR37', 'ELSA_CASE', 'MESH', 'mesh_<zone>.
↪dat')
r['zone_prefix'] = 'Block'
r['topology_file'] = os.path.join('..', 'data', 'ROTOR37', 'ELSA_CASE', 'script_topo.py')
r['shared'] = True
base = r.read()

r = antares.Reader('bin_tp')
```

(continues on next page)

(continued from previous page)

```
r['base'] = base
r['filename'] = os.path.join('..', 'data', 'ROTOR37', 'ELSA_CASE', 'FLOW', 'flow_<zone>.
↪dat')
r['zone_prefix'] = 'Block'
r['location'] = 'cell'
r.read()

base.unstructure()

w = antares.Writer('hdf_cgns')
w['base'] = base
w['filename'] = os.path.join('OUTPUT', 'unst_rotor37.cgns')
w.dump()
```

UnwrapLine

Description

Unwrap a one-dimensional set of points in the space to get a one-dimensional continuous set of points in space.

Parameters

- **base: `Base`**
  The input base.
- **sorting_variable: `str`, default=** ***None***
  The name of the variable used to sort the points.

  If specified, the sorting method will sort the data following the increasing order of the variable values.

  If not given, it will unwrap the input dataset using the connectivity and the criterion of proximity between the endpoints of the disconnected closed curves. In this case, the **coordinates** key has to be used.
- **coordinates: `list(str)`, default=** ***global_var.coordinates***
  The variable names that define the set of coordinates. It is used only when there are more than one endpoint in the connectivity (curves not connected) to find the closest segment to continue with to travel along the line.
- **memory_mode: `bool`, default=** ***False***
  If True, the input base is deleted on the fly to limit memory usage.

Preconditions

The input base must contain a one-dimensional set of points in the instants.

Postconditions

With the **sorting_variable** parameter:

- If the **sorting variable** is not in the shared, the shared variables will be ordered using the first instant ordering.
- Only variables that have the same location as the sorting variable are taken into account. The resulting base will only contain these variables.
- The element connectivity is lost.

Without the **sorting_variable** parameter:

- Unwrapping without **sorting_variable** only works for 'node' variables
- If the input dataset represents closed curves, one point will be duplicated for each closed curve. So, the output dataset may contain more points than the input dataset.
- The **coordinates** parameter is only needed if the curve contains more than one endpoint (it is composed of multiple separated curves) so the curve can be unwrapped using proximity criteria to find the next curve. Note that no node will be removed even if they are at a null distance of each other (if you want to do so look at the merge treatment which contains a duplicate removal functionality)

Example

```
import antares
myt = antares.Treatment('unwrapline')
myt['base'] = base
myt['sorting_variable'] = 'theta'
unwrapped = myt.execute()
```

Main functions

class antares.treatment.TreatmentUnwrapLine.**TreatmentUnwrapLine**

execute()

Unwrap a one-dimensional set of points.

Returns
the Base obtained after unwrapping.

Return type
Base

Example

```
"""
This example illustrates how to use the unwrap line treatment.
"""
import os

import antares
```

(continues on next page)

(continued from previous page)

```
if not os.path.isdir('OUTPUT'):
    os.makedirs('OUTPUT')

# ------------------
# Reading the files
# ------------------
reader = antares.Reader('bin_tp')
reader['filename'] = os.path.join('..', 'data', 'ROTOR37', 'GENERIC',
                                  'flow_<zone>_ite0.dat')
ini_base = reader.read()

ini_base.set_computer_model('internal')

# -------------------
# Cutting
# -------------------
# 2 cuts are done on a initial 3D dataset to create a 1D line.
cut1 = antares.Treatment('cut')
cut1['base'] = ini_base
cut1['type'] = 'cylinder'
cut1['origin'] = [0., 0., 0.]
cut1['radius'] = 180.
cut1['axis'] = 'x'
result = cut1.execute()

cut2 = antares.Treatment('cut')
cut2['base'] = result
cut2['type'] = 'plane'
cut2['origin'] = [62., 0., 0.]
cut2['normal'] = [1., 0., 0.]
result = cut2.execute()

# -------------------------------------
# Merge the multi-element line in one
# -------------------------------------
merge = antares.Treatment('merge')
merge['base'] = result
result = merge.execute()

# plot result to show that the multi-zone structure is not practical
result.compute('theta')

plot1 = antares.Treatment('plot')
plot1['base'] = result[:, :, ('theta', 'rovx')]
plot1.execute()

# prompt the user
# (allows to pause the script to visualize the plot)
input('press enter to see the result after unwrapping the line')

# ----------------------------
# Unwrap the line and Plot
```

(continues on next page)

(continued from previous page)

```
# ----------------------------
unwrap = antares.Treatment('unwrapline')
unwrap['base'] = result
unwrap['sorting_variable'] = 'theta'
result = unwrap.execute()

plot2 = antares.Treatment('plot')
plot2['base'] = result[:, :, ('theta', 'rovx')]
plot2.execute()

# prompt the user
input('press enter to quit')

w = antares.Writer('column')
w['filename'] = os.path.join('OUTPUT', 'ex_unwrap.dat')
w['base'] = result[:, :, ('theta', 'rovx')]
w.dump()
```

ComputeCylindrical

Description

Transform variables from the Cartesian coordinate system to the cylindrical coordinate system.

Parameters

- **base: `Base`**
  The input base to be transformed.
- **coordinates: *list(str)***
  The coordinate names of the Cartesian coordinate system. If no value is given, the default coordinate system of the base is used (see `antares.api.Base.Base.coordinate_names`) if any or ['x', 'y', 'z'] otherwise.
- **axis: *list(float)*,**
  The Cartesian coordinates of the revolution unit axis of the new cylindrical coordinate system. Valid values are [1., 0., 0.] or [0., 1., 0.] or [0., 0., 1.].
- **origin: *list(float)*,**
  The Cartesian coordinates (list of 3 floats) of the origin of the new cylindrical coordinate system.
- **vector: *list(str)*, default= *None***
  List of 3 strings denoting the name of the X, Y, Z components of the vector in the Cartesian coordinate system. The order is important. This vector will be projected into the cylindrical coordinate system.
- **matrix: *list(str)*, default= *None***
  List of 6 strings denoting the name of the XX, XY, XZ, YY, YZ, ZZ components of the matrix in the Cartesian coordinate system. The order is important. This matrix will be projected into the cylindrical coordinate system.

Preconditions

The coordinates must be (at least) in the first instant.

Postconditions

If **vector** and **matrix** are not given, the treatment will only return 'r' and 'theta' coordinates.

The output variables are:

- 'u_r' and 'u_theta' for the **vector**
- 'Mcyl_r_r', 'Mcyl_r_theta', 'Mcyl_r_axe', 'Mcyl_r_theta', 'Mcyl_theta_theta', 'Mcyl_theta_axe', 'Mcyl_r_axe', 'Mcyl_theta_axe', 'Mcyl_axe_axe' for the **matrix**

Example

The following example shows the full potential of this treatment. A transformation of the velocity vector and the Reynolds stress tensor is performed in the cylindrical coordinate system using the axis of rotation (1., 0., 0.) and (0., 0., 0.) as the origin.

```
import antares
myt = antares.Treatment('ComputeCylindrical')
myt['base'] = base
myt['origin'] = [0., 0., 0.]
myt['axis'] = [1., 0., 0.]
myt['vector'] = ['Vx', 'Vy', 'Vz']
myt['matrix'] = ['ReynoldsStress_XX', 'ReynoldsStress_XY', 'ReynoldsStress_XZ',
                 'ReynoldsStress_YY', 'ReynoldsStress_YZ',
                 'ReynoldsStress_ZZ']
base_cyl = myt.execute()
```

Main functions

class antares.treatment.TreatmentComputeCylindrical.**TreatmentComputeCylindrical**

calculate_matrix(*base*)

calculate_vector(*base*)

execute()

Execute the treatment.

Returns

Return type
Base

CellNormal

Description

Compute the normal vectors at centers of 1D or 2D mesh elements in the Cartesian/cylindrical coordinate system. The input data is a geometric surface or line.

The direction of the normal vector field is implicitly given by the connectivity of elements. This is slightly different from `antares.treatment.TreatmentBoundaryNormal` (page 167) with which the direction is explicitly chosen.

Parameters

- **base:** `Base`
  The input base representing a geometric surface or line. The base is modified in-place at the output.
- **coordinates:** `list(str)`
  The ordered coordinate names of the Cartesian coordinate system.
- **cylindrical_coordinates:** `list(str)`**, default=** ***None***
  The ordered coordinate names of the cylindrical coordinate system. If set, the components of the normal vector will also be given in the cylindrical coordinate system.
- **orientation:** `int` **in [-1, 1], default=*1***
  Normal orientation.
  - 1: the orientation defined by the connectivity of elements.
  - -1: the opposite orientation.
- **origin:** `list(float)`**, default=** ***None***
  Coordinates of the origin point used to set up the normal orientation. If it is not given, the orientation depends on the ordering of the mesh element nodes.
- **unit:** `bool`**, default=** ***False***
  If True then the output vectors are unit vectors.
- **auto_orient: class:*bool*, default=** ***False***
  If True then all normals are oriented in the same direction. Note that this may be memory and time consuming.

Preconditions

The treatment only works on a 1D or 2D dataset. Geometric 2D elements are in a 3D physical space (3 coordinates). Geometric 1D elements are in a 2D physical space (2 coordinates).

If **cylindrical_coordinates** is specified, the first cartesian coordinate is the revolution axis.

If **auto_orient** is True, the connectivity array must contain information for non-disjoint mesh parts. You may use `antares.treatment.TreatmentMerge` (page 150) with the options **duplicates_detection** and **tolerance_decimals** to merge all zones in a single one.

Postconditions

The input **base** is modified in-place, extended with the components of the normal vector for each mesh element. The three components of the normal vector in the Cartesian coordinate system are named with the following convention: ('normal_<cartesian_coordinate_name>', 'cell'). The cell surface is stored under ('surface', 'cell').

If **cylindrical_coordinates** is given, then the three components of the normal vector are also given in the cylindrical coordinate system: ('normal_<cylindrical_coordinate_name>', 'cell').

Example

The following example computes unit normal vectors at cell centers.

```
myt = ant.Treatment('CellNormal')
myt['base'] = base
myt['coordinates'] = ['x', 'y', 'z']
myt['unit'] = True
myt.execute()
```

Main functions

class antares.treatment.TreatmentCellNormal.**TreatmentCellNormal**

execute()

Compute the normal vector of mesh elements.

BoundaryNormal

Description

Compute the outward normal vectors at centers of 1D or 2D mesh elements in the Cartesian/cylindrical coordinate system. The input data is the boundaries of a geometric volume or surface.

The normal vector field points towards the outside of the domain by default. This is slightly different from *antares.treatment.TreatmentCellNormal* (page 166).

Parameters

- **base: `Base`**
  The input base representing a geometric volume or surface with boundary conditions. The boundary conditions of the base are modified in-place at the output.
- **coordinates: `list(str)`**
  The ordered coordinate names of the Cartesian coordinate system.
- **cylindrical_coordinates: `list(str)`, default= *None***
  The ordered coordinate names of the cylindrical coordinate system. If set, the components of the normal vector will also be given in the cylindrical coordinate system.
- **orientation: `int` in [-1, 1], default=*1***
  Normal orientation.

  - 1: the outward orientation or towards the outside of the bounded domain.
  - -1: the inward orientation or towards the inside of the bounded domain.

  For unstructured grids, it is assumed that the connectivity already respects the outward orientation. For structured grids, if the chosen orientation is the opposite of the one given by the connectivity, then note that the connectivity still remains unchanged.

- **unit: `bool`, default= *False***
  If True then the output vectors are unit vectors.

Preconditions

The treatment only works on a 2D or 3D dataset that owns boundary conditions.

If **cylindrical_coordinates** is specified, the first Cartesian coordinate is the revolution axis.

Postconditions

The boundary conditions of the input **base** are modified in-place at the output. They extended with the components of the normal vector for each mesh element. The three components of the normal vector in the Cartesian coordinate system are named with the following convention: ('normal_<Cartesian_coordinate_name>', 'cell'). The cell surface is stored under ('surface', 'cell').

If **cylindrical_coordinates** is given, then the three components of the normal vector are also given in the cylindrical coordinate system: ('normal_<cylindrical_coordinate_name>', 'cell').

Example

The following example computes unit normal vectors at cell centers.

```
myt = ant.Treatment('BoundaryNormal')
myt['base'] = base
myt['coordinates'] = ['x', 'y', 'z']
myt['unit'] = True
myt.execute()
```

Main functions

class `antares.treatment.TreatmentBoundaryNormal.`**`TreatmentBoundaryNormal`**

`execute``()`

Compute the normal vector of boundary mesh elements.

Flux

Description

Compute fluxes through a surface.

Parameters

- **base:** `Base`
  The input base on which the integration will be performed for each variable except coordinates.
- **coordinates:** *list(str)*
  The variable names that define the set of coordinates used for integration.
- **vectors:** *list()*, **default=** *[]*
  List of the vector names and their component variable names.

  Example: [[“massflow”, (“rhou”, “rhov”, “rhow”)],
  [“n”, (“nx”, “ny”, “nz”)]].

Preconditions

The coordinate variables and the normal unit vector must be available at node.

Postconditions

Example

```
import antares
myt = antares.Treatment('flux')
```

Main functions

class antares.treatment.TreatmentFlux.**TreatmentFlux**

execute()

Execute the integration treatment.

Returns
the base containing the results

Return type
`Base`

Example

```
"""
This example illustrates how to compute the flux over a surface
"""
import os
if not os.path.isdir('OUTPUT'):
    os.makedirs('OUTPUT')

from antares import Reader, Treatment

# ------------------
# Reading the files
# ------------------
reader = Reader('bin_tp')
reader['filename'] = os.path.join('..', 'data', 'ROTOR37', 'GENERIC', 'flow_<zone>_ite
↪<instant>.dat')
reader['zone_prefix'] = 'Block'
reader['topology_file'] = os.path.join('..', 'data', 'ROTOR37', 'GENERIC', 'script_
↪topology.py')
base = reader.read()

outlet_patch = base[base.families['F_row_1_OUTFLOW']]

print(outlet_patch[0][0])
outlet_patch.compute_cell_normal()
# -----------------
# Flux computation
# -----------------
treatment = Treatment('flux')
treatment['base'] = outlet_patch
treatment['vectors'] = [['massflow', ('rovx', 'rovy', 'rovz')]]
result = treatment.execute()

# the result base contain one zone with the integrated data
# the zone contains the values for the different instants
print(result[0])

# in each instant, is stored :
# - for each vector, the integrated flux (scalar product of vector with normal)
# - for each scalar variables (not vectors), the integration along the 3 directions␣
↪(variable * normal)
# - the surface of the dataset and the surface along each direction
print(result[0][0])

print(result[0][0]['surface'])
```

Gradient

Description

This treatment computes the gradient of variables located at nodes. It can also compute the curl, divergence, Qcriterion, Lambda2 and Lambdaci of vectors since they are closely related to the velocity gradient.

Two computing methods are available.

The first method (**vtk** = *False*) is based on the Green-Ostrogradski theorem to compute the gradient and the volume at the cell centers. i.e. $V = \frac{1}{3} \oint_S C.n\, dS$ with V the volume, C one point on the surface S, and n the outward normal. It handles both structured and unstructured grids.

The second method (**vtk** = *True*) is based on the VTK library to compute the gradients at nodes. It handles only unstructured grids. Note that the `antares.treatment.TreatmentUnstructure.TreatmentUnstructure` (page 160) may be used for structured grids.

Parameters

- **base:** `Base`
  The input base on which the gradient will be computed. This base is modified in-place.
- **coordinates:** *list(str)*
  The coordinates used for gradient.
- **variables:** *list(str)*, **default=** *[]*
  The variables for which gradient will be computed. If not given, gradient will be computed on all variables except the coordinates.
- **vtk:** *bool*, **default=** *False*
  If False, use the method based on Green-Ostrogradski theorem. if True, use the VTK library.
- **curl:** *list(list(str, list(str)))*, **default=** *[]*
  The list of list of the name of the curl to compute and the list of the vector variables from which the curl will be computed.
  - example: [[‘vorticity’, [‘U’, ‘V’, ‘W’]]]

  The variables vorticity_<coordinate_name> and vorticity_modulus will be computed. The vector variables must be ordered as the coordinates.
- **divergence:** *list(list(str, list(str)))*, **default=** *[]*
  The list of list of the name of the divergence to compute and the list of the vector variables from which the divergence will be computed.
  - example: [[‘dilatation’, [‘U’, ‘V’, ‘W’]]]

  The variable dilatation will be computed. The vector variables must be ordered as the coordinates.
- **Qcriterion:** *list(str)*, **default=** *[]*
  The list of velocity variables to compute the variable Qcrit.
  - example: [‘U’, ‘V’, ‘W’]

  The vector variables must be ordered as the coordinates.
- **Lambda2:** *list(str)*, **default=** *[]*
  The list of velocity variables to compute the variable Lambda2.
  - example: [‘U’, ‘V’, ‘W’]

The vector variables must be ordered as the coordinates.

- **L2_eig:** ***bool*****, default=** ***False***
  If True, the eigenvectors are saved in the variables: Lambda2_V1y, Lambda2_V1z, Lambda2_V2x, Lambda2_V2y, Lambda2_V2z
- **Lambdaci:** ***list(str)*****, default=** ***[]***
  The list of velocity variables to compute the variable Lambdaci.
  - example: ['U', 'V', 'W']

  The vector variables must be ordered as the coordinates.
- **Lci_eig:** ***bool*****, default=** ***False***
  If True, the eigenvectors are saved in the variables: Lambdaci_Vx, Lambdaci_Vy, Lambdaci_Vz

Preconditions

Zones can be either structured or unstructured without **vtk**. Zones must be unstructured with **vtk**.

Zones can contain multiple instants and a shared instant.

Gradient computation with the first method has only been implemented in 3D, and 2D with triangles.

Coordinates and variables must be available at nodes.

The computations of curl, divergence, Qcriterion, Lambda2 and Lambdaci assume 3D cartesian coordinates.

The computations of Lambda2 and Lambdaci may not work with **vtk** = *True*.

Postconditions

The input **base** is modified in-place.

If the input variables are shared, then the output gradients are shared.

The computed gradient is either at cells or at nodes depending on the method.

If curl, divergence, Qcriterion, Lambda2 and Lambdaci of vectors are computed, intermediate computed gradients are kept in the base.

The computed gradient are added to the input base with the names 'grad_<var_name>_<coordinate_name>'.

The volume variable is added to the input base with the name ('cell_volume', 'cell').

The curl variables are added to the input base with the names '<curl_name>_<coordinate_name>' and '<curl_name>_modulus'

The divergence $\nabla \cdot \vec{U} = \frac{\partial U_x}{\partial x} + \frac{\partial U_y}{\partial y} + \frac{\partial U_z}{\partial z}$ is added to the input base with the names '<div_name>'.

The Q-criterion $Q = 0.5(||\Omega||_F^2 - ||S||_F^2)$ is added to the input base with the name 'Qcrit'. $\nabla\vec{U} = S + \Omega$ is the velocity gradient tensor with its symmetric part, the strain rate tensor $S = 0.5(\nabla\vec{U} + (\nabla\vec{U})^T)$, and its antisymmetric part, the vorticity tensor $\Omega = 0.5(\nabla\vec{U} - (\nabla\vec{U})^T)$. The principal invariants of the rank-two tensor $\nabla\vec{U}$ of dimension three are the solutions of the cubic characteristic polynomial $det(\nabla\vec{U} - \lambda I) = 0$, or $\lambda^3 + P\lambda^2 + Q\lambda + R = 0$. The Q-criterion is the second invariant, $0.5((tr(\nabla\vec{U}))^2 - tr((\nabla\vec{U})^2))$ where tr is the trace. A vortex exists where $Q > 0$.

The λ_2-criterion is added to the input base with the name 'Lambda2'. The three eigenvalues of $\Omega^2 + S^2$ are computed. λ_2 is defined as the second eigenvalue. The two first eigenvectors 'Lambda2_V1x', 'Lambda2_V1y', 'Lambda2_V1z', 'Lambda2_V2x', 'Lambda2_V2y' and 'Lambda2_V2z' are added to the input base if **L2_eig** is *True*.

J. Jeong and F. Hussain. On the identification of a vortex. Journal of Fluid Mechanics, 285:69-94, (1995).

The Lambdaci criterion is added to the input base with the name 'Lambdaci'. Its eigenvectors 'Lambdaci_Vx', 'Lambdaci_Vy' and 'Lambdaci_Vz' are added to the input base if **Lci_eig** is *True*.

J. Zhou, R. J. Adrian, S. Balachandar, and T. M. Kendall. Mechanisms for generating coherent packets of hairpin vortices in channel flow. Journal of Fluid Mechanics, 387:353-396, (1999).

Example

The following example shows gradient computations.

```
import antares
myt = antares.Treatment('gradient')
myt['base'] = base
myt['coordinates'] = ['x', 'y', 'z']
myt['variables'] = ['Density', 'Pressure']
myt['vtk'] = False
myt['curl'] = [['vorticity', ['U', 'V', 'W']]]
myt['divergence'] = [['dilatation', ['U', 'V', 'W']]]
myt['Qcrit'] = ['U', 'V', 'W']
myt['Lambda2'] = ['U', 'V', 'W']
myt['L2_eig'] = True
myt['Lambdaci'] = ['U', 'V', 'W']
myt['Lci_eig'] = True
myt.execute()
```

Main functions

class antares.treatment.TreatmentGradient.**TreatmentGradient**

execute()

Execute the treatment.

Returns
the base containing the initial variables and the newly computed gradient variables

Return type
Base

Example

```
"""
This example shows how to compute the gradient of variables on a 3D base.
Note that the gradient vector computed for each variable is stored
at cell centers in the input base.
"""
import os
if not os.path.isdir('OUTPUT'):
    os.makedirs('OUTPUT')

from antares import Reader, Treatment, Writer
```

(continues on next page)

(continued from previous page)

```
# ------------------
# Reading the files
# ------------------
reader = Reader('bin_tp')
reader['filename'] = os.path.join('..', 'data', 'ROTOR37', 'GENERIC', 'flow_<zone>_ite0.
↪dat')
ini_base = reader.read()

# ---------------------
# Gradient computation
# ---------------------
treatment = Treatment('gradient')
treatment['base'] = ini_base
result = treatment.execute()

result.compute_cell_volume()
# -------------------
# Writing the result
# -------------------
writer = Writer('bin_tp')
writer['filename'] = os.path.join('OUTPUT', 'ex_gradients.plt')
writer['base'] = result
writer.dump()
```

```
"""
This example shows how to use the Treatment gradient on a poiseuille flow.
Gradients, vorticity, dilatation and stress tensors are computed.
"""
import os
import antares
import numpy as np

OUTPUT = 'OUTPUT'
if not os.path.isdir(OUTPUT):
    os.makedirs(OUTPUT)

# Define case
Re = 100000.                    # [-] Reynolds number
Mu = 8.9e-4                     # [Pa.s] Dynamic Viscosity
Rho = 997.                      # [Kg.m-3] Density
H = 0.1                         # [m] Height
L = 0.1                         # [m] Length
l = L*0.01                      # [m] thickness
Nx, Ny, Nz = 100, 100, 2        # [-] Number of nodes
P0 = 0.                         # [Pa] Ref Pressure

# Compute some values
U_mean = Re*Mu/H/Rho
U_max = 1.5*U_mean
Q = U_mean*H
dpdx = -Q/(H**3)*12.*Mu
Tau_w = -H/2.*dpdx
```

(continues on next page)

(continued from previous page)

```
# Define geometry
xmin, ymin, zmin = 0., 0., 0.
xmax, ymax, zmax = L, H, l

x = np.linspace(xmin, xmax, Nx)
y = np.linspace(ymin, ymax, Ny)
z = np.linspace(zmin, zmax, Nz)

X, Y, Z = np.meshgrid(x, y, z)

# Define velocity and pressure field
U = -H**2/2./Mu*dpdx*(Y/H*(1-Y/H))
V = np.zeros_like(U)
W = np.zeros_like(V)
Pressure = P0 + X*dpdx

# Initialisation
base = antares.Base()
zone = antares.Zone()
instant = antares.Instant()
base['zone_0'] = zone
base['zone_0'].shared[('x', 'node')] = X
base['zone_0'].shared[('y', 'node')] = Y
base['zone_0'].shared[('z', 'node')] = Z
base['zone_0']['instant_0'] = instant
base['zone_0']['instant_0'][('Pressure', 'node')] = Pressure
base['zone_0']['instant_0'][('U', 'node')] = U
base['zone_0']['instant_0'][('V', 'node')] = V
base['zone_0']['instant_0'][('W', 'node')] = W

# Use vtk or not
use_vtk = False

# Unstructure base if vtk
if use_vtk:
    t = antares.Treatment('unstructure')
    t['base'] = base
    base = t.execute()
    loc = 'node'
else:
    loc = 'cell'
    base.node_to_cell(variables=['Pressure'])

# Compute gradients
treatment = antares.Treatment('Gradient')
treatment['base'] = base
treatment['coordinates'] = ['x', 'y', 'z']
treatment['variables'] = ['Pressure']
treatment['vtk'] = use_vtk
treatment['curl'] = [['vorticity', ['U', 'V', 'W']]]
treatment['divergence'] = [['dilatation', ['U', 'V', 'W']]]
```

(continues on next page)

(continued from previous page)

```
treatment['Qcrit'] = ['U', 'V', 'W']
treatment['Lambda2'] = ['U', 'V', 'W']
treatment['L2_eig'] = True
treatment['Lambdaci'] = ['U', 'V', 'W']
treatment['Lci_eig'] = True
base = treatment.execute()

# Compute stress tensors
base.set_computer_model('internal')

base.set_formula('Mu={:f}'.format(Mu))

base.compute('tau_xx = -2. / 3. * Mu * dilatation + Mu * (2 * grad_U_x)', location=loc)
base.compute('tau_yy = -2. / 3. * Mu * dilatation + Mu * (2 * grad_V_y)', location=loc)
base.compute('tau_zz = -2. / 3. * Mu * dilatation + Mu * (2 * grad_W_z)', location=loc)
base.compute('tau_xy = Mu * (grad_V_x + grad_U_y)', location=loc)
base.compute('tau_xz = Mu * (grad_W_x + grad_U_z)', location=loc)
base.compute('tau_yz = Mu * (grad_W_y + grad_V_z)', location=loc)

base.compute('sigma_xx = Pressure + tau_xx', location=loc)
base.compute('sigma_yy = Pressure + tau_yy', location=loc)
base.compute('sigma_zz = Pressure + tau_zz', location=loc)
base.compute('sigma_xy = tau_xy', location=loc)
base.compute('sigma_xz = tau_xz', location=loc)
base.compute('sigma_yz = tau_yz', location=loc)

# Write output
writer = antares.Writer('bin_vtk')
writer['filename'] = os.path.join(OUTPUT, 'results')
writer['base'] = base
writer.dump()
```

Interpolation

Description

This treatment interpolates a multiblock source CFD domain on another target CFD domain.

The interpolation method is the Inverse Distance Weighting.

The complexity is N*logN thanks to the use of a fast spatial search structure (kd-tree).

The set of N known data points: $[\ldots,(x_i,y_i),\ldots]$

Interpolated value u at a given point x based on samples $u_i = u(x_i)$:

$$u(x) = \frac{\sum_1^N \frac{1}{d(x,x_i)^p} u_i}{\sum_1^N \frac{1}{d(x,x_i)^p}} \text{ if } d(x,x_i) \neq 0 \forall i$$

$$u(x) = u_i \text{ if } \exists i, d(x,x_i) = 0$$

x is the arbitrary interpolated point. x_i is the known interpolating point. $d(x,x_i)$ is the distance from the known point x_i to the unknown point x. N is the total number of known points used in interpolation. p is the power parameter.

Warning: dependency on scipy.spatial[118]

Parameters

- **source: `Base`**
  The source mesh with the variables you want to interpolate.
- **target: `Base`**
  The mesh to interpolate on. The base is modified in-place at the output.
- **coordinates: *list(str)*, default= *None***

  The variable names that define the set of coordinates used for interpolation.

  The KDTree extracts closest points based on these variables. If None, use the coordinates from the **source** or **target** base.
- **variables: *list(str)*, default= *None***
  List of variables to apply for the interpolation. If None, all variables will be interpolated, except the coordinates. Using this parameter allow to call the interpolation treatment on bases with different variable locations.
- **tolerance: *float*, default= *1e-10***
  The value that tells if the distance from the closest point is smaller than this tolerance, the interpolated value is equal to the value of that closest point.
- **duplicated_tolerance: *int*, default= *None***
  Number of decimal places to round to for duplicated points detection. If decimals is negative, it specifies the number of positions to the left of the decimal point. If None, the detection will be exact.
- **invdist_power: *int*, default= *1***
  The power of the inverse distance, this acts like a norm (1, 2, or infinity etc.) and can be used to smooth/refined the interpolation.
- **nb_points: *int*, default= *None***
  The number of closest points used for the interpolation. If not provided by the user, it will automatically be set depending on the type of cells in the mesh.
- **with_boundaries: *bool*, default= *False***
  Whether or not to use data from the boundaries.

Preconditions

Postconditions

The **target** base is modified in-place.

[118] https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.spatial.cKDTree.html#scipy.spatial.cKDTree

Example

```
import antares
myt = antares.Treatment('interpolation')
myt['source'] = source_base
myt['target'] = target_base
myt.execute()
```

Main functions

class antares.treatment.TreatmentInterpolation.**TreatmentInterpolation**

Class for Interpolation Treatment.

execute()

Interpolate data from a base onto data from another base.

Returns
the target base containing the results

Return type
Base

Example

```
"""
This example illustrates the interpolation of a
cloud of points against another cloud of points
"""

from copy import copy
import os
if not os.path.isdir('OUTPUT'):
    os.makedirs('OUTPUT')


from antares import Reader, Treatment, Writer

# ------------------
# Reading the files
# ------------------
reader = Reader('bin_tp')
reader['filename'] = os.path.join('..', 'data', 'ROTOR37', 'GENERIC', 'flow_<zone>_
↪<instant>.dat')
source_base = reader.read()

print(source_base)

# --------------------
# Build a target base
# --------------------
# (same mesh but at cell centers)
```

(continues on next page)

(continued from previous page)

```
tmp_base = copy(source_base[:, :, ('x', 'y', 'z')])
tmp_base.node_to_cell()
target_base = tmp_base.get_location('cell')

# --------------
# Interpolation
# --------------
treatment = Treatment('interpolation')
treatment['source'] = source_base
treatment['target'] = target_base
result = treatment.execute()

# Note that the result contains the same instants as the source and
# each instant contains the variable interpolated and the coordinates
# used for interpolation
print(result[0])
# >>>  Zone object
#      - instants : ['ite0', 'ite1', 'ite2']
print(result[0][0])
# >>>  Instant object
#      - variables : [('x', 'cell'), ('y', 'cell'), ('z', 'cell'), ('ro', 'cell'), ('rovx', 'cell'), (
↪'rovy', 'cell'), ('rovz', 'cell'), ('roE', 'cell')]
#      - shape : None

# -------------------
# Writing the result
# -------------------
writer = Writer('bin_tp')
writer['base'] = result
writer['filename'] = os.path.join('OUTPUT', 'ex_interpolation.plt')
writer.dump()
```

Integration

This method integrates variables on a base.

Parameters

- **base: Base**
  The base on which the integration will be performed for each variable except coordinates.
- **coordinates:** ***list(str)***
  The coordinates used for integration.
- **mean:** ***bool*****, default=** ***False***
  Compute mean.

Preconditions

The coordinate variables must be available at nodes. The integration will be performed on all the other variables. Variables at nodes are interpolated at cell centers.

Main functions

class antares.treatment.TreatmentIntegration.**TreatmentIntegration**

execute()

Execute the treatment.

Returns
a new base containing the results

Return type
Base

Example

```
"""
This example illustrates how to integrate data over a dataset
"""
import os
if not os.path.isdir('OUTPUT'):
    os.makedirs('OUTPUT')

from antares import Reader, Treatment

# ------------------
# Reading the files
# ------------------
reader = Reader('bin_tp')
reader['filename'] = os.path.join('..', 'data', 'ROTOR37', 'GENERIC', 'flow_<zone>_
↪<instant>.dat')
ini_base = reader.read()

# -----------
# Integration
# -----------
treatment = Treatment('integration')
treatment['base'] = ini_base
result = treatment.execute()

# the result base contain one zone with the integrated data
# the zone contains the values for the different instants
print(result[0])

# in each instant, the result of the integration for each
# variable is stored as well as the volume of the dataset
print(result[0][0])
```

(continues on next page)

(continued from previous page)

```
print(result[0][0]['volume'])
```

Time Average

Description

Compute a basic time average of the base on a characteristic integration time using an online algorithm.

The treatment stores the number of accumulated iterations and the averaged base so it can be called many times to perform the averaging.

Parameters

- **base: `Base`**
  The input base to be averaged.
- **nb_ite_averaging: *int* or in_attr, default= *'in_attr'***
  Number of iterations over which averaging is done. If **in_attr**, then each zone of the base must have an attribute **nb_ite_averaging**. This is the number of iterations performed by the solver, which ends up with the **base**. In particular, it may be different from the number of instants.
- **extracts_step: *int*, default= 1**
  The number of time iterations between two instants of the base. For example, if the number of physical iterations is 10, maybe you could have sampled the signal every 2 iterations so you get 5 instants in the **base**.
- **memory_mode: *bool*, default= *False***
  If True, the initial base is deleted on the fly to limit memory usage.

Preconditions

Zones may contain multiple instants and shared variables.

The extract timestep is assumed constant.

Postconditions

The output base contains one instant named 'average' and the shared instant from the input base.

Example

The following example shows a time averaging on the first 1000 iterations with an extraction step of 10.

```
import antares
myt = antares.Treatment('onlinetimeaveraging')
myt['base'] = base
myt['nb_ite_averaging'] = 1000
myt['extracts_step'] = 10
```

(continues on next page)

(continued from previous page)

```
myt['memory_mode'] = False
base_avg = myt.execute()
```

Main functions

class antares.treatment.TreatmentOnlineTimeAveraging.**TreatmentOnlineTimeAveraging**

execute()

Average variables.

Example

```
import numpy as np

import antares

b = antares.Base()
b['0000'] = antares.Zone()
x, y, z = np.meshgrid(np.linspace(0.0, 1.0, 3),
                      np.linspace(0.0, 1.0, 3),
                      np.linspace(0.0, 1.0, 3))
b[0].shared['x'] = x
b[0].shared['y'] = y
b[0].shared['z'] = z

for i in range(10):
    b[0]['%d'%i] = antares.Instant()
    b[0][i]['v'] = np.ones(x.shape) * (i+1) * 2

myt = antares.Treatment('onlinetimeaveraging')
myt['base'] = b
myt['nb_ite_averaging'] = 20
myt['extracts_step'] = 2
base_avg = myt.execute()

print(base_avg[0]['average']['v'])
```

Chorochronic Reconstruction (Synchronous)

Description

Compute the chorochronic (or multi-chorochronic) reconstruction of a CFD computation for synchronous phenomenon.

In turbomachinery, a chorochronic simulation takes into account two types of periodicity of the flow:

(1) a space-time periodicity, when rows return back to the same relative position of rows shifted of a number of blades.

(2) a time periodicity, when rows return back to the same relative position with no shift.

Formally, for scalar fields, those periodicies are expressed as follows:

(1) $w(\theta + \Delta\theta, t) \ = \ w(\theta, t + \Delta t)$,

(2) $w(\theta, t + T) \ = \ w(\theta, t)$.

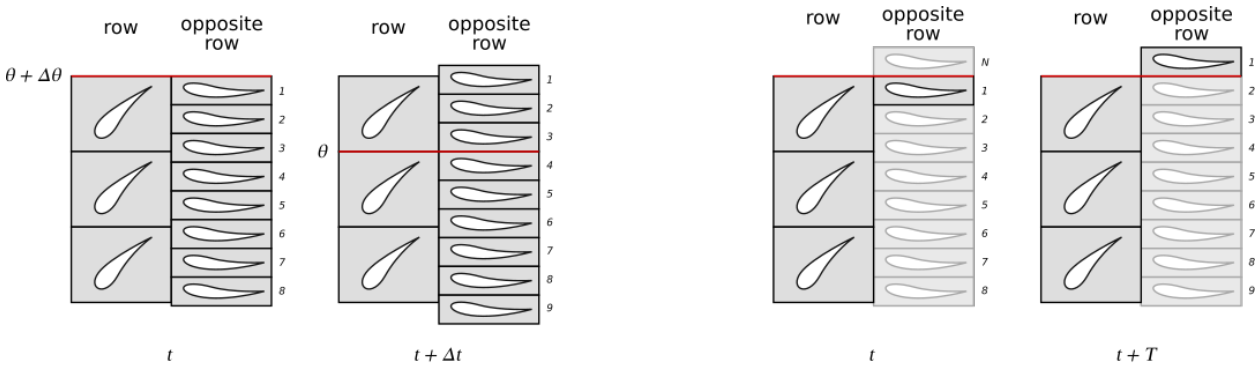
For vector fields in cartesian coordinates, those expressions are:

(1) $\vec{w}(\theta + \Delta\theta, t) \ = \ \mathrm{rot}_{\Delta\theta} \cdot \vec{w}(\theta, t + \Delta t)$,

(2) $\vec{w}(\theta, t + T) \ = \ \vec{w}(\theta, t)$.

The these relations, θ is the azimuth in the rotating frame of the current row.

The azimuthal continuity of the flow (2π periodicity) can be retrieved from those relations (see *Formulae*).

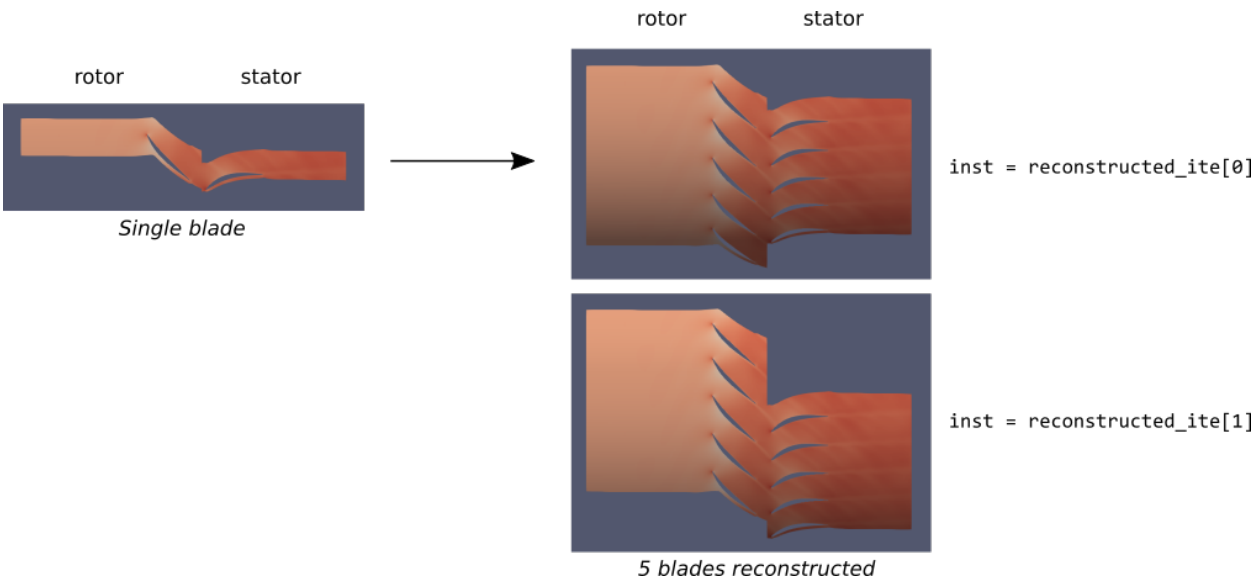


(1) Space-time periodicity

(2) Time periodicity

The reconstruction is based on Discrete Fourier Transform that simplifies chorochronic condition 1. The treatment is based on instant numbers and time iterations instead of real time. For theoretical details, see *Formulae*.

The data is given on a grid in relative cartesian coordinates (unmoving mesh). The reconstructed data is put a a grid in absolute cartesian coordinates, that can move for rows that have a nonzero rotation speed. In particular, after reconstruction, vectors (assumed expressed in cartesian coordinates) are additionally rotated.



Construction

```
import antares
myt = antares.Treatment('chororeconstruct')
```

Parameters

- **base: `Base`**
  The input base that will be chorochronically duplicated.
- **type: *str*, default= *'fourier'***
  The type of approach ('fourier' or 'least_squares') used to determine the amplitudes and phases of each interaction mode. If 'fourier', a DFT is performed: in the case of a single opposite row, the base should contain enough instants to cover the associated pattern of blades passage (**'nb_ite_rot' / 'extracts_step'** * **'nb_blade_opp_sim'/ 'nb_blade_opp'**); in the case of multiple opposite rows, the base should contain enough instants to cover a complete rotation (**'nb_ite_rot' / 'extracts_step'**). If 'least_squares', a least-squares approach is used. For robustness and accuracy reasons, it is advised to have enough instants in the base to cover at least two blade passages (2 * **'nb_ite_rot' / 'extracts_step' / 'nb_blade_opp'**).
- **coordinates: *list(str)***
  The variable names that define the set of coordinates used for duplication. It is assumed that these are in cartesian coordinates.
- **vectors: *list(tuple(str)*, default= *[]***
  If the base contains **vectors**, these must be rotated. It is assumed that they are expressed in cartesian coordinates.
- **reconstructed_ite: *list(int)*, default= *[0]***
  List of iterations to be reconstructed.
- **nb_blade: *int or 'in_attr'*, default= *'in_attr'***
  Number of blades of the current row. If 'in_attr', then each zone of the base must have an attribute 'nb_blade'.

- **nb_blade_sim:** ***int or 'in_attr'*****, default=** ***1***
  Number of blades simulated of the current row. If 'in_attr', then each zone of the base must have an attribute 'nb_blade_sim'.
- **nb_blade_opp:** ***int or list(int) or 'in_attr'*****, default=** ***'in_attr'***
  Number of blades of the opposite row (or list of numbers of blades of the opposite rows in the case of multiple opposite rows). If 'in_attr', then each zone of the base must have an attribute 'nb_blade_opp'.
- **nb_blade_opp_sim:** ***int or list(int) or 'in_attr'*****, default=** ***'in_attr'***
  Number of blades simulated of the opposite row (or list of numbers of blades simulated of the opposite rows in the case of multiple opposite rows). If 'in_attr', then each zone of the base must have an attribute 'nb_blade_opp_sim'.
- **omega:** ***float or 'in_attr'*****, default=** ***'in_attr'***
  Rotation speed of the current row expressed in radians per second. If 'in_attr', then each zone of the base must have an attribute 'omega'.
- **omega_opp:** ***float or list(float) or 'in_attr'*****, default=** ***'in_attr'***
  Rotation speed of the opposite row (or list of rotation speeds of the opposite rows in the case of multiple opposite rows) expressed in radians per second. If 'in_attr', then each zone of the base must have an attribute 'omega_opp'.
- **nb_harm:** ***int or 'in_attr'*****, default=** ***None***
  Number of harmonics used for the reconstruction. If it is not given, all the harmonics are computed for type 'fourier' and the first three harmonics are computed for type 'least_squares'. If 'in_attr', then each zone of the base must have an attribute 'nb_harm.
- **nb_ite_rot:** ***int***
  The number of time iterations to describe a complete rotation, *i.e.* the period in number of iterations for which rows get back to the same relative position (*e.g.* if blade 1 of row 1 is in front of blade 3 of row 2, then it is the period in number of iterations to get back to the same configuration).
- **extracts_step:** ***int*****, default=** ***1***
  The number of time iterations between two instants of the base. This number is assumed constant over pairs of successive instants.
- **nb_duplication:** ***int or tuple(int) or 'in_attr'*****, default=** ***'in_attr'***
  Number of duplications if an integer is given (given base included). Range of duplications if a tuple is given (from first element to second element included). If 'in_attr', then each zone of the base must have an attribute 'nb_duplication'.
- **ite_init:** ***int*****, default=** ***0***
  Iteration of the first instant of the base.

Preconditions

The coordinates of each row must be given in the relative (rotating) frame, *i.e.* constant over instants.

In case of a single opposite row, there must be enough instants to cover a time period T.

In case of multiple opposite rows, there must be enough instants to cover a complete rotation, *i.e.* the period for which rows get back to the same relative position.

As the treatment is based of Discrete Fourier Transform on the time variable, there must be enough instants to fulfill hypotheses of the Nyquist-Shannon sampling theorem.

Vectors and coordinates must be given in cartesian coordinates.

Postconditions

The output base contains for each row *nb_duplication* times more zones than the input base.

The output base is given in the absolute (fixed) frame, *i.e.* moving mesh over instants.

Warnings

If you use an equationmanager which assumes that some quantities are constant (for instance the adiabatic index *gamma* and the ideal gas constant *R* in equationmanager *"internal"*) and if you compute new variables (for instance entropy *s*) BEFORE the reconstruction: After the reconstruction, you should delete those constants quantities and compute them again. If not, the value of those such constant quantities are reconstructed and thus are approximated.

Academic example

```
"""
Example of two rows chorochronic reconstruction of an academic case.

Let us consider the following two rows machine:
- Rotor: 25 blades, at 22000 rad/s.
- Stator: 13 blades, at 0 rad/s.

The flow is described by the following field v, fulfilling both periodicities:
(1) v(theta + DTHETA, t) = v(theta, t + DT)
(2) v(theta, t + T) = v(theta, t)
where DTHETA is the pitch of the rotor, DT is the chorochronic period, and T
is the opposite row blade passage period,
    v(theta, t) = cos(5.0*r)* sin((NB_BLADE_OPP - NB_BLADE)
                                  *(theta + t*DTHETA/DT))

Let us consider an axial cut of one sector (one blade) of the rotor of this
machine.
This code shows how to reconstruct 9 blades over 30 iterations of
one stator blade passage.
"""
import os

import numpy as np
import antares

OUTPUT = 'OUTPUT'
if not os.path.isdir(OUTPUT):
    os.makedirs(OUTPUT)

# properties of the machine
NB_BLADE = 25
NB_BLADE_OPP = 13
OMEGA = 22000.0
OMEGA_OPP = 0.0

# chorochronic properties of the machine
```

(continues on next page)

(continued from previous page)

```
DTHETA = 2.0*np.pi/NB_BLADE
DT = 2.0*np.pi*(1.0/NB_BLADE_OPP - 1.0/NB_BLADE)/(OMEGA_OPP - OMEGA)
T = 2.0*np.pi/NB_BLADE_OPP/abs(OMEGA_OPP - OMEGA)

# properties of discretization
N_THETA = 50
N_R = 50
EXTRACTS_STEP = 1
TIMESTEP = 1.0e-7

# number of instants to cover the time period T
NB_INST_ALL_POSITIONS = int(np.ceil(T/TIMESTEP/EXTRACTS_STEP))

# build the sector
b = antares.Base()
b['rotor_xcut'] = antares.Zone()
r = np.linspace(1.0, 2.0, N_R)
theta = np.linspace(0.0, DTHETA, N_THETA)
r, theta = np.meshgrid(r, theta)
b[0].shared['x'] = np.zeros_like(r)
b[0].shared['r'] = r
b[0].shared['theta'] = theta
b.compute('y = r*cos(theta)')
b.compute('z = r*sin(theta)')
del b[0].shared['r']
del b[0].shared['theta']
for idx_inst in range(NB_INST_ALL_POSITIONS):
    inst_key = str(idx_inst)
    b[0][inst_key] = antares.Instant()
    t = TIMESTEP*EXTRACTS_STEP*idx_inst
    b[0][inst_key]['v'] = np.cos(5.0*r) * np.sin((NB_BLADE_OPP - NB_BLADE)
                                                 *(theta + t*DTHETA/DT))

# number of iterations to perform a complete rotation of the opposite
# row relatively to the current row
NB_ITE_ROT = 2.0*np.pi/abs(OMEGA_OPP - OMEGA)/TIMESTEP

# perform chorochronic reconstruction on one blade passage
ITE_BEG = 0
ITE_END = T/TIMESTEP
ITE_TO_RECONSTRUCT = np.linspace(ITE_BEG, ITE_END, 30)
t = antares.Treatment('chororeconstruct')
t['base'] = b
t['type'] = 'fourier'
t['coordinates'] = ['x', 'y', 'z']
t['vectors'] = []
t['reconstructed_ite'] = ITE_TO_RECONSTRUCT # covers one blade passage
t['nb_blade'] = NB_BLADE
t['nb_blade_sim'] = 1
t['nb_blade_opp'] = NB_BLADE_OPP
t['nb_blade_opp_sim'] = 1
t['omega'] = OMEGA
```

(continues on next page)

(continued from previous page)

```
t['omega_opp'] = OMEGA_OPP
t['nb_harm'] = None
t['nb_ite_rot'] = NB_ITE_ROT
t['extracts_step'] = EXTRACTS_STEP
t['nb_duplication'] = 9 # to get 9 blades
t['ite_init'] = 0
b_choro = t.execute()

# write the result
w = antares.Writer('hdf_antares')
w['base'] = b_choro
w['filename'] = os.path.join(OUTPUT, 'ex_choro_academic')
w.dump()
```

Real-case example

```
"""
Example for the treatment chororeconstruct

This test case is CPU intensive
"""
import antares
from math import pi
import os
if not os.path.isdir('OUTPUT'):
    os.makedirs('OUTPUT')

results_directory = os.path.join("OUTPUT", 'ex_CHORO_RECONSTRUCTION')
if not os.path.exists(results_directory):
    os.makedirs(results_directory)

# Chorochronic properties
N_front = 30  # number of blades of front row
N_rear  = 40  # number of blades of rear row
nb_duplication_front = N_front  # number of duplications for front row (can be different␣
↪from the number of blades)
nb_duplication_rear = N_rear  # number of duplications for rear row (can be different␣
↪from the number of blades)

nb_ite_per_rot = 9000  # number of iterations per rotation
extracts_step = 5  # extraction step

omega_front = -1.947573300000e-03  # speed of rotation of front row (rad/s)
omega_rear  = 0.  # speed of rotation of rear row (rad/s)

nb_snapshots = 60  # number of snapshots to reconstruct (the snapshots are reconstructed␣
↪at each extracts_step)

# Read the unsteady data
r = antares.Reader('bin_tp')
```

(continues on next page)

(continued from previous page)

```
r['filename'] = os.path.join('..', 'data', 'ROTOR_STATOR', 'CHORO', 'CUTR', 'cutr_
↪<instant>.plt')
b = r.read()

# Create front row family and rear row family:
# here, the first five zones of the slice belongs to front row and the last five to rear␣
↪one
front_family = antares.Family()
for zone in range(0, 5):
    front_family['%04d' % zone] = b[zone]
b.families['front'] = front_family

rear_family = antares.Family()
for zone in range(5, 10):
    rear_family['%04d' % zone] = b[zone]
b.families['rear'] = rear_family

# Define the properties of the chorochronic treatment for each row
b.families['front'].add_attr('nb_duplication', nb_duplication_front)
b.families['front'].add_attr('nb_blade', N_front)
b.families['front'].add_attr('nb_blade_opp', N_rear)
b.families['front'].add_attr('nb_blade_opp_sim', 1)
b.families['front'].add_attr('omega', omega_front)
b.families['front'].add_attr('omega_opp', omega_rear)

b.families['rear'].add_attr('nb_duplication', nb_duplication_rear)
b.families['rear'].add_attr('nb_blade', N_rear)
b.families['rear'].add_attr('nb_blade_opp', N_front)
b.families['rear'].add_attr('nb_blade_opp_sim', 1)
b.families['rear'].add_attr('omega', omega_rear)
b.families['rear'].add_attr('omega_opp', omega_front)

# Execute the chorochronic resconstruction treatment
t = antares.Treatment('chororeconstruct')
t['base'] = b
t['type'] = 'fourier'
t['vectors'] = [('rovx', 'rovy', 'rovz')]
t['reconstructed_ite'] = range(0, nb_snapshots * extracts_step, extracts_step)
t['nb_blade'] = 'in_attr'
t['nb_blade_opp'] = 'in_attr'
t['nb_blade_opp_sim'] = 'in_attr'
t['nb_duplication'] = 'in_attr'
t['omega'] = 'in_attr'
t['omega_opp'] = 'in_attr'
t['nb_ite_rot'] = nb_ite_per_rot
t['extracts_step'] = extracts_step
t['ite_init'] = 36001
result = t.execute()

# Write the results
w = antares.Writer('bin_tp')
w['base'] = result
```

(continues on next page)

(continued from previous page)

```
w['filename'] = os.path.join(results_directory, 'choro_reconstruction_<instant>.plt')
w.dump()
```

Formulae

Those formulae are applicable in the case of one row and one opposite row.

Space and time periods

Using the following notations:

- Number of blades of the current row: N
- Number of blades simulated of the current row: N_{sim}
- Rotation speed of the current row: ω
- Number of blades of the opposite row: N_{opp}
- Number of blades simulated of the opposite row: $N_{opp_{sim}}$
- Rotation speed of the opposite row: ω_{opp}

The space and time periods are defined as follows:

- chorochronic space period: $\Delta\theta \ = \ 2\pi\dfrac{N_{sim}}{N}$
- chorochronic time period: $\Delta t \ = \ \dfrac{2\pi}{\omega_{opp}-\omega}(\dfrac{N_{opp_{sim}}}{N_{opp}} - \dfrac{N_{sim}}{N})$
- time period (opposite row blade passage period): $T \ = \ \dfrac{2\pi}{(\omega_{opp}-\omega)}\dfrac{N_{opp_{sim}}}{N_{opp}}$

Since $\dfrac{N}{N_{sim}}$ and $\dfrac{N_{opp}}{N_{opp_{sim}}}$ are integers, chorochronic and time periodicities imply azimuthal periodicity:

- $w(\theta+2\pi,t) \ = \ w(\theta+\dfrac{N}{N_{sim}}\Delta\theta,t) \ = \ w(\theta,t+\dfrac{N}{N_{sim}}\Delta t) \ = \ w(\theta,t+T\cdot[\dfrac{N}{N_{sim}} - \dfrac{N_{opp}}{N_{opp_{sim}}}]) \ = \ w(\theta,t)$

Chorochrony and Fourier transform

When solving for only one blade passage of the real geometry, the flow is time-lagged from one blade passage to another. The phase-lagged periodic condition is used on the azimuthal boundaries of a single blade passage. It states that the flow in a blade passage at time t is the flow at the next passage, but at another time $t+\Delta t$: $w(\theta+\Delta\theta,t) = w(\theta,t+\Delta t)$.

The Fourier transform of the left hand side of this relation is:

$\hat{w}(\theta+\Delta\theta,\xi) \ = \ \int_{\mathbb{R}} w(\theta+\Delta\theta,t)\cdot e^{-2i\pi\xi t}dt \ = \ \int_{\mathbb{R}} w(\theta,t+\Delta t)\cdot e^{-2i\pi\xi t}dt$

Which simplifies to the following relation after a change of variable:

$\hat{w}(\theta+\Delta\theta,\xi) \ = \ e^{2i\pi\xi\Delta t}\hat{w}(\theta,\xi)$.

This last relation is used to reconstruct the flow over time on several blade passages using a Discrete Fourier Transform.

Relations between instants, time iterations and real time

Using the following notations:

- Real time: t
- Iteration: ite
- Instant: $inst$
- Timestep: $timestep$
- Extract step (number of iterations between two instants): $extract\_step$

We have the following basic relations between instants, time iterations and real time:

- $ite = \dfrac{t}{timestep}$
- $inst = \dfrac{ite}{extract\_step}$

From these relations, we can get:

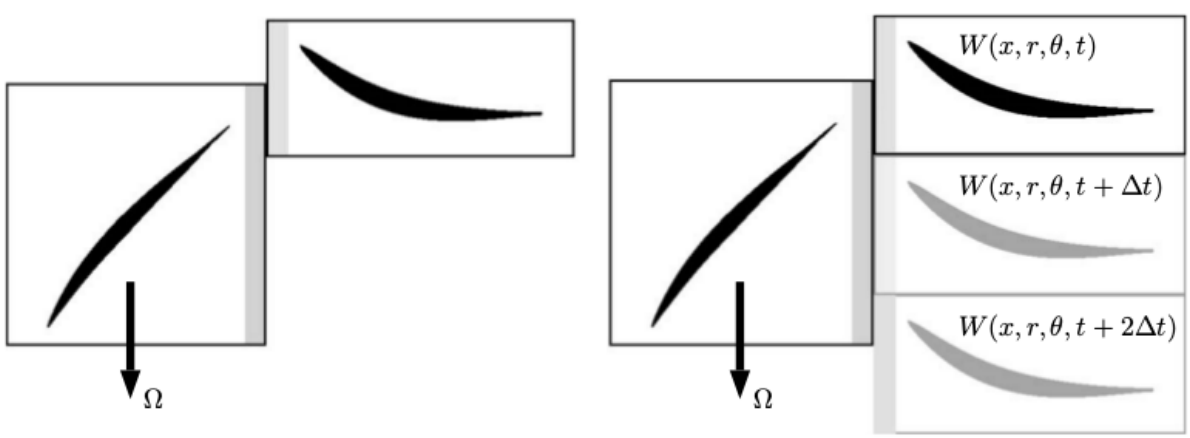
- Number of instants for one opposite row blade passage: $nb\_inst\_T \ = \left\lceil \dfrac{T}{timestep \cdot extract\_step} \right\rceil$
- Number of iterations of one rotation: $nb\_ite\_rot \ = \dfrac{2\pi}{|\omega_{opp} - \omega| \cdot timestep}$

Chorochronic Reconstruction (Asynchronous)

Description

Compute the chorochronic (or multi-chorochronic) reconstruction of a CFD computation for asynchronous phenomenon.

Here is given an example of the classic chorochronic reconstruction (only two rows and one resolved frequency per row). The same principles apply for the multi-chorochronic reconstruction (more than two rows and multiple frequencies resolved per row).



On the left, relative mesh position of two blade rows. On the right, duplication with phase-lagged periodic conditions.

When solving for only one blade passage of the real geometry, the flow is time-lagged from one blade passage to another. The phase-lagged periodic condition is used on the azimuthal boundaries of a single blade passage. It states that the flow in a blade passage at time t is the flow at the next passage, but at another time $t+\Delta t$: $W(x,r,\theta+\theta_p,t) = W(x,r,\theta,t+\Delta t)$.

This time lag can be expressed as the phase of a rotating wave traveling at the same speed as the relative rotation speed of the opposite row: $\Delta t = \beta/\omega_\beta$. The interblade phase angle β depends on each row blade number and relative rotation velocity: $\beta = 2\pi sgn(\omega_r - \omega_s)(1 - N_s/N_r)$.

The Fourier series (of the above flow equation) are: $\sum_{k=-\infty}^{k=\infty} \hat{W}_k(x,r,\theta + \theta_p)\exp(ik\omega_\beta t) = \sum_{k=-\infty}^{k=\infty} \hat{W}_k(x,r,\theta)\exp(ik\omega_\beta t)\exp(ik\omega_\beta \Delta t)$. The spectrum of the flow is then equal to the spectrum of the neighbor blade passage modulated by a complex exponential depending on the interblade phase angle: $\hat{W}_k(x,r,\theta+\theta_p) = \hat{W}_k(x,r,\theta)\exp(ik\omega_\beta\Delta t)$.

This last relation is used to reconstruct the flow over time on several blade passages.

This treatment has two possibilities for flow reconstruction with partial information. One very efficient is the 'least_square' and is doing a least square optimization to compute fourier coefficient. The other one is the 'shape_correction' and is usually used during cfd computation to update coefficient

Parameters

- **base: `Base`**
  The input base that will be chorochronically duplicated.
- **type: *str*, default= *'fourier'***
  The type of approach ('fourier' or 'least_squares') used to determine the amplitudes and phases of each interaction mode. If 'least_squares', a least-squares approach is used. For robustness and accuracy reasons, it is advised to have enough instants in the base to cover at least three oscillations 'shape_correction' is an online algorithm and converge slowly.
- **coordinates: *list(str)***
  The variable names that define the set of coordinates used for duplication. It is assumed that these are in cartesian coordinates.
- **vectors: *list(tuple(str)*, default= *[]***
  If the base contains **vectors**, these must be rotated. It is assumed that they are expressed in cartesian coordinates.
- **reconstructed_ite: *list(int)*, default= *[0]***
  List of iterations to be reconstructed.
- **nb_blade: *int or 'in_attr'*, default= *'in_attr'***
  Number of blades of the current row. If 'in_attr', then each zone of the base must have an attribute 'nb_blade'.
- **omega: *float or 'in_attr'*, default= *'in_attr'***
  Rotation speed of the current row expressed in radians per second. If 'in_attr', then each zone of the base must have an attribute 'omega'.
- **waves_freq: *float or list(float) or 'in_attr'*, default= *'in_attr'***
  Frequency simulated in the current row (or list of frequencies in the current row in the case of multiple rotating wave phenomenon) expressed in Hz. If 'in_attr', then each zone of the base must have an attribute 'freq_of_interest'.
- **waves_omega: *float or list(float) or 'in_attr'*, default= *'in_attr'***
  Rotation speed of the wave (or list of rotation speeds of the waves in the case of multiple rotating wave phenomenon) expressed in radians per second. If 'in_attr', then each zone of the base must have an attribute 'wave_omega'.

- **waves_nb_harm:** ***int or list(int) or 'in_attr'*****, default=** ***None***
  Number of harmonics used for the reconstruction of waves. If not given, all the harmonics are computed for type 'fourier' and the first three harmonics are computed for type 'least_squares'. If 'in_attr', then each zone of the base must have an attribute 'wave_nb_harm.
- **extracts_step:** ***int*****, default=** ***1***
  The number of time iterations between two instants of the base. This number is assumed constant over pairs of successive instants.
- **nb_duplication:** ***int or tuple(int) or 'in_attr'*****, default=** ***'in_attr'***
  Number of duplications if an integer is given (given base included). Range of duplications if a tuple is given (from first element to second element included). If 'in_attr', then each zone of the base must have an attribute 'nb_duplication'.
- **ite_init:** ***int*****, default=** ***0***
  Iteration of the first instant of the base.
- **theta_init:** ***float*****, default=** ***None***
  Initial rotation of the first instant of the base. If not prescribed theta_init will be infered from **ite_init** while assuming timestep has been kept constant during the simulation.
- **timestep:** ***float*****, default=** ***1.0e-8***
  Constant time interval used during the simulation.

Main functions

class `antares.treatment.TreatmentChoroReconstructAsync.`**`TreatmentChoroReconstructAsync`**

Asynchronous multi-chorochronic reconstruction of a CFD computation.

`execute``()`

Perform the chorochronic reconstruction.

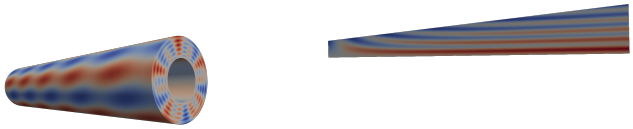
Returns
the base containing the results

Return type
`Base`

Azimuthal Average

Description

Azimuthal averaging around a given axis of a single-zone base.



==>

3D axisymmetric mesh on the left. 2D meridional mesh on the right resulting from the azimuthal average.

This treatment spawns multiple processes thanks to the multiprocessing package.

A number of azimuthal slices are interpolated on a reference slice.

Parameters

- **base:** `Base`
  The single-zone input base.
- **axis:** *list(float)*
  The Cartesian coordinates of the revolution unit axis of the averaging. Valid values are [1., 0., 0.] or [0., 1., 0.] or [0., 0., 1.].
- **origin:** *list(float)*
  The Cartesian coordinates (list of 3 floats) of the origin of the cylindrical coordinate system.
- **nb_cuts:** *int*
  The desired number of azimuthal slices for the averaging. This gives a uniform discretization of the azimuthal direction.
- **nb_procs:** *int*
  The desired number of processes used to perform the averaging.
- **type:** *str*, **default=** *'Full'*
  The type of averaging needed. The values are:
  - 'Full' for a 360 degree average
  - 'Sector'

  If 'Sector' is selected, the user must enter the **angles** values.
- **angles:** *list(float)*, **default=** *[0, pi]*
  Angles (in radian) defining the averaging sector.

  example: treatment['angles'] = [0, pi/2] will perform the averaging between 0 and pi/2.

  The angles are defined differently depending on the axis:
  - if axis = [1,0,0] ; angle=0 lies on y-axis while angle=pi/2 lies on z-axis
  - if axis = [0,1,0] ; angle=0 lies on z-axis while angle=pi/2 lies on x-axis
  - if axis = [0,0,1] ; angle=0 lies on x-axis while angle=pi/2 lies on y-axis
- **vector:** *list(str)*, **default=** *None*
  List of 3 strings denoting the name of the X, Y, Z components of the vector in the Cartesian coordinate system. The order is important. This vector will be transformed into cylindrical coordinates during the azimuthal averaging procedure.
- **matrix:** *list(str)*, **default=** *None*
  List of 6 strings denoting the name of the XX, XY, XZ, YY, YZ, ZZ components of the matrix in the Cartesian coordinate system. The order is important. This matrix will be transformed into cylindrical coordinates during the azimuthal averaging procedure.
- **duplicate:** *bool*, **default=** *False*
  If True, and if **type** = 'Full', it duplicates the fields and paste it on the other side of the axis of rotation. Can be useful (esthetically) for symetric plots or colormaps.
- **multithreading:** *bool*, **default=** *False*
  If True, the treatment will use a more memory efficient method to compute the azimuthal average. It will take longer in average, but can be useful when the script is restricted in memory. You need to install Antares with –mshcppcutter and the right flags for this option to work. **nb_procs** will become the number of threads used for each cut.

- **cut_type: *str*, default= *"cut"***
  The cut treatment to use. By default, uses VTK's cut. Valid options are "cut" for *antares.treatment.TreatmentCut.TreatmentCut* (page 131), "acut" for *antares.treatment.TreatmentAcut.TreatmentAcut* (page 134), and "ccut" for *antares.treatment.TreatmentCcut.TreatmentCcut* (page 136).

Preconditions

The input **base** must contain only one zone. The coordinates must be (at least) in the first instant.

Requirements for averaging type:

- Full: 'axis', 'origin', 'nb_cuts', 'nb_procs'
- Sector: 'axis', 'origin', 'nb_cuts', 'nb_procs', 'angles'

Postconditions

Example

The following example shows an azimuthal average around the axis (1., 0., 0.) using (0., 0., 0.) as origin. The average is performed over 360 degrees ('Full' selected)

```
import antares
myt = antares.Treatment('AzimuthalAverage')
myt['base'] = base
myt['type'] = 'Full'
myt['origin'] = [0., 0., 0.]
myt['axis'] = [1., 0., 0.]
myt['nb_procs'] = 10
myt['nb_cuts'] = 180
cutbase = myt.execute()
```

Main functions

class antares.treatment.TreatmentAzimuthalAverage.**TreatmentAzimuthalAverage**

execute()

Execute the treatment.

Returns

Return type
Base

execute_cpp()

final_average(*cut_bases*)

Complete the average from the partial average coming from processes.

Parameters
cut_bases (list(Base)) – bases of all cuts given by processes

Returns
the final cut with the average.

Return type
`Base`

`get_mesh_ori`(*base*)

`reduction_to_one_side`(*cut_final*, *full_render*)

Spanwise Average

Description

Spanwise averaging along a given axis of a base.

This treatment spawns multiple processes thanks to the multiprocessing package.

Parameters

- **base:** `Base`
  The input base.
- **axis:** ***list(float)*,**
  The Cartesian coordinates of the axis along with the averaging will be performed.
- **nb_cuts:** ***int*,**
  The desired number of cuts for the averaging.
- **nb_procs:** ***int*,**
  The desired number of proc used to perform the averaging.
- **minmax:** ***list(float)*, default=** ***[np.amin(base[0][0]['x']), np.amax(base[0][0]['x'])]***
  Minimum and maximum values defining the spanwise averaging window.

Preconditions

The coordinates must be (at least) in the first instant.

Postconditions

Example

The following example shows an azimuthal average along the x-axis (1., 0., 0.) from x=0.1 to x=0.2.

```
import antares
myt = antares.Treatment('SpanwiseAverage')
myt['base'] = base
myt['axis'] = [1., 0., 0.]
myt['minmax'] = [0.1, 0.2]
myt['nb_procs'] = 10
myt['nb_cuts'] = 180
cutbase = myt.execute()
```

Main functions

class antares.treatment.TreatmentSpanwiseAverage.**TreatmentSpanwiseAverage**

execute()

Execute the treatment.

Returns

Return type
Base

final_average(*cut_bases*)

Complete the average from the partial average coming from processes.

Parameters
cut_bases (list(Base)) – bases of all cuts given by processes

Returns
the final cut with the average.

Return type
Base

get_mesh_ori(*base*)

Grid line

Grid line extraction using topology information

Parameters

- **base: Base**
  The base in which search the grid line.
- **indices:** ***list(int)***
  List of indices of the grid line starting point.
- **indices_location:** ***str*****, default=** ***'node'***
  Location of the given indices.
- **zone_name:** ***str***
  Name of the zone of the grid line starting point.
- **line_type:** ***str***
  Type of grid line, should be in ['I', 'J', 'K', '+I', '+J', '+K', '-I', '-J', '-K'].

Preconditions

This only works structured grids. The base must only contain variables located at the given location.

Main functions

class antares.treatment.TreatmentGridline.**TreatmentGridline**

execute()

Execute the treatment.

Returns
the family containing the *Window* (page 22) corresponding to the grid line

Return type
Family (page 20)

line_types = ['I', 'J', 'K', '+I', '+J', '+K', '-I', '-J', '-K']

Example

```
"""
This example illustrates how to extract a topological grid line
from a multi-block multi-instant dataset using the gridline treatment.

Note that the treatment is returning a family which
can then be applied to the base to get the data or used
to get the elsA extractor corresponding
"""
import os
if not os.path.isdir('OUTPUT'):
    os.makedirs('OUTPUT')

from antares import Reader, Treatment, Writer

# ----------------------------------------------
# Reading the files and topological information
# ----------------------------------------------
reader = Reader('bin_tp')
reader['filename'] = os.path.join('..', 'data', 'ROTOR37', 'GENERIC', 'flow_<zone>_ite
↪<instant>.dat')
reader['zone_prefix'] = 'Block'
reader['topology_file'] = os.path.join('..', 'data', 'ROTOR37', 'GENERIC', 'script_
↪topology.py')
base = reader.read()

# -----------------------------
# Retreive the gridline family
# -----------------------------
treatment = Treatment('gridline')
treatment['base'] = base
treatment['zone_name'] = 'Block0000'
```

(continues on next page)

(continued from previous page)

```
treatment['line_type'] = 'K'
treatment['indices'] = (10, 2, 1)
gridline_family = treatment.execute()

print(gridline_family)
# >>>  Family object
#      - objects : ['Block0000_11_11_3_3_1_29', 'Block0001_19_19_3_3_1_117']


# add the family to the base
base.families['my_grid_line'] = gridline_family


# extract the data on the gridline
gridline = base[gridline_family]

print(gridline)
# >>>  Base object
#      - zones : ['Block0000_11_11_3_3_1_29', 'Block0001_19_19_3_3_1_117']


# -------------------
# Writing the result
# -------------------
writer = Writer('bin_tp')
writer['filename'] = os.path.join('OUTPUT', 'ex_gridline.plt')
writer['base'] = gridline
writer.dump()
```

Grid Plane

Grid plane extraction using topology information (for structured grid only)

Parameters

- **base: `Base`**
  The base in which search the grid plane.
- **index: *int***
  Grid plane starting index.
- **zone_name: *str***
  Grid plane starting zone name.
- **plane_type: *str***
  Type of grid plane, should be in ['I', 'J', 'K'].

Main functions

class antares.treatment.TreatmentGridplane.**TreatmentGridplane**

execute()

Execute the treatment.

Returns
the family containing the *Window* (page 22) corresponding to the grid plane

Return type
Family (page 20)

plane_types = ['I', 'J', 'K']

Example

```
"""
This example illustrates how to extract a topological grid plane
from a multi-block multi-instant dataset using the gridplane treatment.

Note that the treatment is returning a family which
can then be applied to the base to get the data or used
to get the elsA extractor corresponding
"""
import os
if not os.path.isdir('OUTPUT'):
    os.makedirs('OUTPUT')

from antares import Reader, Treatment, Writer

# ----------------------------------------------
# Reading the files and topological information
# ----------------------------------------------
reader = Reader('bin_tp')
reader['filename'] = os.path.join('..', 'data', 'ROTOR37', 'GENERIC', 'flow_<zone>_ite
↪<instant>.dat')
reader['zone_prefix'] = 'Block'
reader['topology_file'] = os.path.join('..', 'data', 'ROTOR37', 'GENERIC', 'script_
↪topology.py')
base = reader.read()

# -----------------------------
# Retreive the gridplane family
# -----------------------------
treatment = Treatment('gridplane')
treatment['base'] = base
treatment['zone_name'] = 'Block0000'
treatment['plane_type'] = 'J'
treatment['index'] = 2
gridplane_family = treatment.execute()

print(gridplane_family)
```

(continues on next page)

(continued from previous page)

```
# >>>  Family object
#      - objects : ['Block0000_1_81_3_3_1_29', 'Block0001_1_29_3_3_1_117',
#                   'Block0002_1_25_3_3_1_33', 'Block0003_1_25_3_3_1_29',
#                   'Block0004_1_21_3_3_1_161', 'Block0005_1_29_3_3_1_117']


# add the family to the base
base.families['my_grid_plane'] = gridplane_family


# extract the data on the gridplane
gridplane = base[gridplane_family]

print(gridplane)
# >>>  Base object
#      - zones : ['Block0000_1_81_3_3_1_29', 'Block0001_1_29_3_3_1_117',
#                 'Block0002_1_25_3_3_1_33', 'Block0003_1_25_3_3_1_29',
#                 'Block0004_1_21_3_3_1_161', 'Block0005_1_29_3_3_1_117']


# -------------------
# Writing the result
# -------------------
writer = Writer('bin_tp')
writer['filename'] = os.path.join('OUTPUT', 'ex_gridplane.plt')
writer['base'] = gridplane
writer.dump()
```

Family Probe

Probe extraction

see also *antares.treatment.TreatmentPointProbe* (page 204)

Description

Extract a point probe from a given input Base.

Parameters

- **base: Base**
  Probes will be searched in this input base.
- **coordinates:** ***list(str)***
  The variable names that define the set of coordinates. The coordinates must always be located at nodes whatever the value of **location**.
- **location:** ***str*** **in** ***LOCATIONS*** **(page 369), default=** ***'node'***
  Location of values that are concerned by the search. If **location** is *'node'*, then the probe value will be computed with variables located at 'node'.

- **points:** ***list(tuple)***
  List of point coordinates.
- **tolerance:** ***float*****, default=** ***None***
  The threshold in the closest-point search.

Preconditions

The base must only contain variables located at the given location.

Main functions

class antares.treatment.TreatmentProbe.**TreatmentProbe**

execute()

Execute the treatment. (Uses function `antares.Base.closest()`)

Returns
the family containing the *Window* (page 22) corresponding to the given points

Return type
Family (page 20)

Example

```
"""
This example illustrates how use the probe treatment.

Note that the treatment is returning a family which
can then be applied to the base to get the data or used
to get the elsA extractor corresponding
"""
import os
if not os.path.isdir('OUTPUT'):
    os.makedirs('OUTPUT')

from antares import Reader, Treatment

# ---------------------------------------------
# Reading the files and topological information
# ---------------------------------------------
reader = Reader('bin_tp')
reader['filename'] = os.path.join('..', 'data', 'ROTOR37', 'GENERIC', 'flow_<zone>_ite0.
↪dat')
reader['zone_prefix'] = 'Block'
base = reader.read()

# ----------------------------
# Retrieve the probes family
# ----------------------------
treatment = Treatment('probe')
```

(continues on next page)

(continued from previous page)

```
treatment['base'] = base
treatment['points'] = [(-20., 160., 0.), (110, 160., 0.)]
probes_family = treatment.execute()

print(probes_family)
# >>>  Family object
#      - objects : ['probe_0000', 'probe_0001']


# add the family to the base
base.families['my_probes'] = probes_family


# extract the data at the probes locations
probes = base[probes_family]

print(probes)
# >>>  Base object
#      - zones : ['probe_0000', 'probe_0001']


# ----------------------------------
# Get the elsA extractor associated
# ----------------------------------
print(probes_family.get_extractor())

# extractor_antares_win = DesGlobWindow('extractor_antares_win')
#
# extractor_antares_probe_0000 = window('Block0002', name='extractor_antares_probe_0000')
# extractor_antares_probe_0000.set('wnd', [ 15, 15, 1, 1, 11, 11])
# extractor_antares_win.attach(extractor_antares_probe_0000)
#
# extractor_antares_probe_0001 = window('Block0000', name='extractor_antares_probe_0001')
# extractor_antares_probe_0001.set('wnd', [ 1, 1, 1, 1, 17, 17])
# extractor_antares_win.attach(extractor_antares_probe_0001)
#
# extractor_antares = extractor('extractor_antares_win', name='extractor_antares')
# extractor_antares.set('var', 'xyz conservative')
# extractor_antares.set('loc', 'node')
# extractor_antares.set('writingmode', 1)
# extractor_antares.set('file', 'extractor_antares')
# extractor_antares.set('format', 'bin_tp')
```

Point Probe

Description

Extract data values from a given input Base at specified point locations.

The probe values are computed at the user-defined point positions by interpolating into the input data. The interpolation can be linear or 0-order based on the closest points.

Parameters

- **base:** `Base`
  Sample probes in this input base.
- **coordinates:** ***list(str)***
  The variable names that define the set of coordinates. The coordinates must always be located at nodes whatever the value of **location**.
- **location:** ***str*** **in** ***LOCATIONS*** **(page 369), default=** ***'node'***
  Location of values that are concerned by the search. If **location** is *'node'*, then the probe value will be computed with variables located at 'node'.
- **points:** ***list(tuple)***
  List of point coordinates.
- **interpolation:** ***str in ['closest', 'linear']*****, default=** ***closest***
  Type of interpolation to apply to the input values to get the probe value.
- **axis:** ***str*** **or** ***None*****, default=** ***None***
  Name of the rotation axis. Must be in **coordinates**. If None, then no rotation will be performed on the zones to find the probe location.
- **angle_name:** ***str*****, default=** ***None***
  Name of the rotation angle that is a key in the dictionary `Zone.attrs`.
- **vectors:** ***tuple/list(tuple(str))*****, default=** ***[]***
  Coordinate names of vectors that need to be rotated.
- **one_zone_per_probe:** ***bool*****, default=** ***True***
  If True, each valid point probe gives one Zone, else all probes are set in a single Zone. This option is not applied with **interpolation** = "closest" and **location** = "node". The value *False* should be used for single-zone input **base**.

Preconditions

The underlying mesh must be time-independent (non-deformable and non-moving mesh).

If the zones have to be rotated, then the input zones must contain a key named as the value given by the parameter **angle_name** in `Zone.attrs`.

Postconditions

The output `Base` object contains as many zones as probe points if they are all detected. Undetected probe points are ignored. See also **one_zone_per_probe**.

The name of an output zone is <name of the input zone>_x_y_z with (x,y,z) the coordinates of the probe. If the zone has been rotated N times, then the name of an output zone is <name of the input zone>_<'DUP%02d'%N>_x_y_z.

If **location** is 'node' and **interpolation** is 'closest', then the probe value will be the node value of the nearest mesh point. The process is implemented in python and the probe coordinates will be found from the input mesh and available in the output base.

If **location** is 'cell' and **interpolation** is 'closest', then the probe value will be the cell value of the mesh element in which the probe is located. The process uses the VTK library and the probe coordinates will not be available in the output base. To get the coordinates in the output, you may use the following instruction before calling the treatment:

```
base.node_to_cell(coordinates)
```

If **location** is 'node' and **interpolation** is 'linear', then the probe value will be the value interpolated from the vertices of the mesh element in which the probe is located. The process uses the VTK library and the probe coordinates will come from the user-defined probes and will be available in the output base.

If **one_zone_per_probe** is False, then the array 'valid_interpolation' is set in the instant of the zone. This boolean array tells if a given point could have been interpolated. This should be used for single-zone input **base**.

Example

```
import antares
myt = antares.Treatment('pointprobe')
myt['base'] = base
myt['points'] = [(-20., 160., 0.), (110, 160., 0.)]
myt['location'] = 'node'
myt['interpolation'] = 'linear'
probes_base = myt.execute()
```

Main functions

class antares.treatment.TreatmentPointProbe.**TreatmentPointProbe**

execute()

Probe the data from the base.

Returns
The base containing the values of the given points.

Return type
`Base`

Example

```
"""
This example illustrates how use the pointprobe treatment.

Compare with probe.py
"""
import os
if not os.path.isdir('OUTPUT'):
    os.makedirs('OUTPUT')

import numpy as np

from antares import Reader, Treatment

# ----------------------------------------------
# Reading the files and topological information
# ----------------------------------------------
reader = Reader('bin_tp')
reader['filename'] = os.path.join('..', 'data', 'ROTOR37', 'GENERIC', 'flow_<zone>_ite0.
↪dat')
reader['zone_prefix'] = 'Block'
base = reader.read()

# -----------------------------
# Retrieve the probes family
# -----------------------------
treatment = Treatment('pointprobe')
treatment['base'] = base
treatment['points'] = [(-20., 160., 0.), (110, 160., 0.)]
treatment['location'] = 'node'
treatment['interpolation'] = 'closest'
probes = treatment.execute()

print(probes)
# >>>  Base object
#      - zones : ['Block0002_-20.0_160.0_0.0', 'Block0000_110_160.0_0.0']

for zonen, zone in probes.items():
    print('Probe {}'.format(zonen))
    for inst in zone.values():
        for varn, var in inst.items():
            print('Variable {}: {}'.format(varn[0], np.squeeze(var)))
```

Swap Axes

Swap the axis of a base. Set i and j as the axis system for two-dimensional structured base.

Parameters

- **base: Base**
  The Base on which the swapping will be performed.

Preconditions

Swapping is based on the axes of the FIRST zone, and is applied to all zones. The orientation remains unchanged: either right-handed orientation or left-handed orientation.

Main functions

class antares.treatment.TreatmentSwapAxes.**TreatmentSwapAxes**

execute()

Execute the treatment.

Returns
a new base with swapped axis

Return type
Base

Example

```
import os
if not os.path.isdir('OUTPUT'):
    os.makedirs('OUTPUT')

from antares import Reader, Treatment, Writer

r = Reader('hdf_cgns')
r['filename'] = os.path.join('..', 'data', 'SHEARLAYER', 'Delville_mmfinal4_047.cgns')
base = r.read()

print(base)
print(base[0])
print(base[0].boundaries)
print(base[0].boundaries[0])

t = Treatment('swapaxes')
t['base'] = base
base = t.execute()

print(base)
print(base[0])
```

(continues on next page)

(continued from previous page)

```
print(base[0].boundaries)
print(base[0].boundaries[0])

w = Writer('hdf_cgns')
w['base'] = base
w['filename'] = os.path.join('OUTPUT', 'ex_swap_Delville_mmfinal4_047.hdf')
w.dump()
```

Cell to Node

Description

Compute node values from cell-centered values.

This treatment contains several features:

- the detection of all connected zones to any node (even for edges and corners).
- connected boundaries inside a same zone result in an appropriate average value.

Processing of the extensive variable 'surface'

The surface area assigned to a point is the average of areas of the elements that contain this vertex. It is defined as: $s_i = \sum_\Omega s_\Omega / N_\Omega$ with Ω the elements that contain the vertex i, s_Ω the surface area of the element Ω, and N_Ω the number of vertices defining the element Ω.

Processing of the normal vector

The normal vector assigned to a point is the sum of the normal vectors of the elements that contain this vertex. If the normal vectors at the cell centers of these elements are unit vectors, then the normal vector at the point is normalized with a norm equal to one. Otherwise, it is normalized with a norm equal to the surface area computed at this point.

Processing of the other variables

The value assigned to a point is the arithmetic average of values of the elements that contain this vertex.

Warning: Other extensive variables may not be computed correctly (e.g. volume).

Parameters

- **base: Base**
  The input base on which the cell2node will be performed.
- **variables: *list(str)*, default= *None***
  List of names of variables to process. If *None*, all variables are processed.
- **report_boundaries: *bool*, default= *False***
  If *True*, the node data in the zone is overwritten with the boundary node data.
- **vectors: *list(str) or list(list(str))*, default= *None***
  List of vector variables which must follow a periodic condition (rotation) if defined.
- **bnd_keep_zeros: *bool*, default= *True***
  If *True*, when zeroes are encountered as bnd node value, they will be kept when merging several boundary node values.

Preconditions

The input base may be structured or unstructured.

Postconditions

This treatment processes the input base in-place. Variables located at cell centers are processed to give new variables located at nodes.

Support boundary data - in this case, boundary values will be reported into the main part.

Example

```
import antares
myt = antares.Treatment('cell2node')
myt['base'] = base
myt['variables'] = ['dens', 'pres']
myt.execute()
```

Main functions

class antares.treatment.TreatmentCell2Node.**TreatmentCell2Node**

Main class for TreatmentCell2Node.

execute()

Compute nodal values from cell-centered values.

To perform the cell2node, they are 2 steps: First, perform a loop on each zone, and perform the cell2node in this zone only. Every nodal point will receive a value coming from the zone. Second, the interfaces (between zones) are processed, again in two steps:

-> mean value on the interface internal part -> edges and corners list data appending (slice).

The edges and corners are processed at the end, to know all contributions (avoid all issues with overwrite, diagonal missing contribution. Other advantage, each interface will be processed only once

TreatmentHOSol2Output

Description

Interpolate a JAGUAR high-order solution onto a user chosen number of output points.

Construction

```
import antares
myt = antares.Treatment('')
```

Parameters

- **base: Base**
  The input base.
- **mesh: Base**
  The input mesh.
- **output_npts_dir: *int*,**
  output_npts_dir.

Preconditions

.

Postconditions

.

Example

.

```
import antares
myt = antares.Treatment('')
 myt.execute()
```

Main functions

class antares.treatment.codespecific.jaguar.TreatmentHOSol2Output.**TreatmentHOSol2Output**

execute()

.

Returns

.

Return type
Base

Example

```
"""
This example shows how to process a jaguar solution file.

Read the jaguar file.
Read the Gmsh file.
Create a uniform mesh.
"""
import os
import antares

if not os.path.isdir('OUTPUT'):
    os.makedirs('OUTPUT')

r = antares.Reader('jag')
r['filename'] = os.path.join('..', 'data', 'HIGHORDER', 'covo2D.jag')
base = r.read()

print(base)
print(base[0])
print(base[0][0])

reader = antares.Reader('fmt_gmsh')
reader['filename'] = os.path.join('..', 'data', 'HIGHORDER', 'covo2D.msh')
mesh_real = reader.read()
# mesh_real.attrs['max_vpc'] = 8

t = antares.Treatment('HOSol2Output')
t['base'] = base
t['mesh'] = mesh_real
t['output_npts_dir'] = 4
result = t.execute()

writer = antares.Writer('bin_tp')
writer['base'] = result
writer['filename'] = os.path.join('OUTPUT', 'ex_jaguar_covo2d.plt')
writer.dump()
```

TreatmentJaguarInit

Description

Interpolate coordinates of a GMSH grid to a basis based on the jaguar input file p parameter. Then compute a solution with the new coordinates and save it as h5py solution file to be read with the coprocessing version of jaguar.

Construction

```
import antares
myt = antares.Treatment('jaguarinit')
```

Parameters

- **base: Base**
  The input base coming from the GMSH reader.
- **jag_config:** *antares.utils.high_order.Jaguar* **(page 221)**
  The Jaguar object built from the jaguar input file.
- **init_func:** *Callable[[antares.utils.high_order.HighOrderSol* **(page 220)** *], list(str)]*
  A function computing initial variables from coordinates and input parameter. Return a list of variable names. Check the example below.

Example

The following example shows how to create an initial solution for the covo/3D example.

```
import numpy as np
import antares
from antares.utils.high_order import Jaguar

def densityVortexInitialization2D(init_sol):

    with open("2DVortex.dat",'r') as fin:
        lines = [float(i.split('!')[0]) for i in fin.readlines()]

    G   = lines[0]
    Rc  = lines[1]
    xC  = lines[2]
    yC  = lines[3]
    cut = lines[4]

    radiusCut = cut*Rc

    gamma = init_sol.gas['gamma']
    Rgas  = init_sol.gas['Rgas']
    Cp    = init_sol.gas['Cp']

    roInf = init_sol.infState['rho']
```

(continues on next page)

(continued from previous page)

```
    uInf = init_sol.infState['u']
    vInf = init_sol.infState['v']

    roEInf   = init_sol.infState['rhoe']
    presInf  = (roEInf - 0.5 * roInf * (uInf*uInf + vInf*vInf)) * (gamma - 1.0)
    TInf     = presInf / (roInf * Rgas)

    X = init_sol.base['z0'].shared['x']
    Y = init_sol.base['z0'].shared['y']

    R2 = (X-xC)**2 + (Y-yC)**2


    # limit the vortex inside a cut*Rc box.
    expon = np.exp(-0.5*R2/Rc**2)

    u = uInf - (Y-yC) / Rc * G * expon
    v = vInf + (X-xC) / Rc * G * expon

    temp = TInf - 0.5 * (G * G) * np.exp(-R2 / Rc**2) / Cp
    ro   = roInf * (temp / TInf)**( 1. / (gamma - 1.) )
    pres = ro * Rgas * temp
    roE = pres / (gamma - 1.) + 0.5 * ( u*u + v*v ) * ro

    # Build solution array as [ncells, nvars, nSolPts] array.
    vlist = ['rho','rhou','rhov','rhow','rhoe']
    init_sol.base['z0']['0'][vlist[0]]  = np.where(np.sqrt(R2)<=radiusCut, ro, roInf)
    init_sol.base['z0']['0'][vlist[1]] = np.where(np.sqrt(R2)<=radiusCut, ro*u,␣
↪roInf*uInf)
    init_sol.base['z0']['0'][vlist[2]] = np.where(np.sqrt(R2)<=radiusCut, ro*v,␣
↪roInf*vInf)
    init_sol.base['z0']['0'][vlist[3]] = 0.
    init_sol.base['z0']['0'][vlist[4]] = np.where(np.sqrt(R2)<=radiusCut, roE, roEInf)

    return vlist


def TGVinit(init_sol):
    gamma = init_sol.gas['gamma']
    Rgas  = init_sol.gas['Rgas']
    Cp    = init_sol.gas['Cp']

    roInf = 1.
    TInf = 351.6
    presInf = 101325.
    u0 = 1
    L = 2*np.pi

    X = init_sol.base['z0'].shared['x']
    Y = init_sol.base['z0'].shared['y']
    Z = init_sol.base['z0'].shared['z']
```

(continues on next page)

(continued from previous page)

```
    u =  u0 * np.sin(X)*np.cos(Y)*np.cos(Z)
    v = -u0 * np.cos(X)*np.sin(Y)*np.cos(Z)
    w = 0.0

    roE = presInf / (gamma - 1.) + 0.5 * ( u*u + v*v + w*w) * roInf

    # Build solution array as [ncells, nvars, nSolPts] array.
    vlist = ['rho','rhou','rhov','rhow','rhoe']
    init_sol.base['z0']['0'][vlist[0]] = roInf
    init_sol.base['z0']['0'][vlist[1]] = roInf*u
    init_sol.base['z0']['0'][vlist[2]] = roInf*v
    init_sol.base['z0']['0'][vlist[3]] = 0.0
    init_sol.base['z0']['0'][vlist[4]] = roE

    return vlist


##################################################"

def main():

    jaguar = Jaguar("input.txt")

    # READER
    # ------

    # Read the GMSH mesh
    r = antares.Reader('fmt_gmsh')
    r['filename'] = jaguar.config['GENERALITIES']['Grid file']
    base_in = r.read()
    base_in.attrs['filename'] = r['filename']


    # TREATMENT
    # ---------

    treatment = antares.Treatment('jaguarinit')

    treatment['base'] = base_in
    treatment['jag_config'] = jaguar
    treatment['init_func'] = TGVinit

    base_out = treatment.execute()


    # WRITER
    # ------

    w = antares.Writer('hdf_jaguar_restart')
    w['base'] = base_out
    w['strategy'] = 'monoproc'
    w.dump()
```

(continues on next page)

(continued from previous page)

```
if __name__ == "__main__":
    main()
```

Main functions

class antares.treatment.codespecific.jaguar.TreatmentJaguarInit.**TreatmentJaguarInit**

execute()

Create an initial solution for Jaguar.

Returns
the unstructured Base obtained from the initial function applied on the input Base coordinates.

Return type
Base

Example

```
"""
This example creates a (restart) initial solution for a jaguar computation with␣
↪coprocessing.
It is based on the covo3D jaguar test case.

Execute it directly in the PATH_TO_JAGUAR/tests/covo/3D directory.
Tested on kraken with:
1/ source /softs/venv_python3/vant_ompi401/bin/activate
2/ source <path_to_antares>/antares.env
3/ python create_jaguar_init_sol.py

Output should be:
```
===
                          ANTARES 1.14.0
                     https://cerfacs.fr/antares
Copyright 2012-2019, CERFACS. This software is licensed.
===

CREATE JAGUAR INSTANCE...
>>> Read input file input.txt...
>>> Mesh file cube_8_p4.h5 Found.
>>> Light load mesh file cube_8_p4.h5 ...
>>> Mesh file cube_8_p4_vtklag.h5 Found.
>>> Light load mesh file cube_8_p4_vtklag.h5 ...
input x_mesh.shape:: (512, 8)
output x_new.shape:: (512, 125)
> 5 var in sol: ['rho', 'rhou', 'rhov', 'rhow', 'rhoe']
> 0 var in additionals: []
>>> XMF file restart_00000000.xmf written.
```

(continues on next page)
```

(continued from previous page)

```
>>> Solution file restart_0000000.jag written.
> 5 var in sol: ['rho', 'rhou', 'rhov', 'rhow', 'rhoe']
> 0 var in additionals: []
>>> XMF file restart_0000000_vtklag.xmf written.
>>> Solution file restart_0000000_vtklag.jag written.
```

4/ Read the file restart_0000000.xmf (solution pts) or restart_0000000_vtklag.xmf␣
↪(output pts) in ParaView.
"""
import os

import numpy as np

import antares
from antares.utils.high_order import Jaguar

def pulse2D(mesh, init_sol):
    ro = init_sol.infState['rho']
    uInf = init_sol.infState['u']
    vInf = init_sol.infState['v']

    p0=1e5
    p0prime=1e3
    x0 = 0.05
    y0 = 0.05

    pres=p0+p0prime*np.exp((-np.log10(2.0)/.0001)*((X-x0)*(X-x0)+(Y-y0)*(Y-y0)))
    roE=pres/(init_sol.infState['gamma']-1)+0.5*ro*(uInf*uInf+vInf*vInf)

    # Build solution array as [ncells, nvars, nSolPts] array.
    vlist = ['rho','rhou','rhov','rhow','rhoe']
    init_sol.base['z0']['0'][vlist[0]] = ro
    init_sol.base['z0']['0'][vlist[1]] = init_sol.infState['rhou']
    init_sol.base['z0']['0'][vlist[2]] = init_sol.infState['rhov']
    init_sol.base['z0']['0'][vlist[3]] = 0.
    init_sol.base['z0']['0'][vlist[4]] = roE

    return vlist

def densityVortexInitialization2D(init_sol):

    with open(os.path.join('..', 'data', 'GMSH', '2DVortex.dat'),'r') as fin:
        lines = [float(i.split('!')[0]) for i in fin.readlines()]

    G   = lines[0]
    Rc = lines[1]
    xC = lines[2]
    yC = lines[3]
    cut = lines[4]

    radiusCut = cut*Rc

(continues on next page)

(continued from previous page)

```
    gamma = init_sol.gas['gamma']
    Rgas  = init_sol.gas['Rgas']
    Cp    = init_sol.gas['Cp']

    roInf = init_sol.infState['rho']
    uInf = init_sol.infState['u']
    vInf = init_sol.infState['v']

    roEInf   = init_sol.infState['rhoe']
    presInf  = (roEInf - 0.5 * roInf * (uInf*uInf + vInf*vInf)) * (gamma - 1.0)
    TInf     = presInf / (roInf * Rgas)

    X = init_sol.base['z0'].shared['x']
    Y = init_sol.base['z0'].shared['y']

    R2 = (X-xC)**2 + (Y-yC)**2


    # limit the vortex inside a cut*Rc box.
    expon = np.exp(-0.5*R2/Rc**2)

    u = uInf - (Y-yC) / Rc * G * expon
    v = vInf + (X-xC) / Rc * G * expon

    temp = TInf - 0.5 * (G * G) * np.exp(-R2 / Rc**2) / Cp
    ro   = roInf * (temp / TInf)**( 1. / (gamma - 1.) )
    pres = ro * Rgas * temp
    roE = pres / (gamma - 1.) + 0.5 * ( u*u + v*v ) * ro

    # Build solution array as [ncells, nvars, nSolPts] array.
    vlist = ['rho','rhou','rhov','rhow','rhoe']
    init_sol.base['z0']['0'][vlist[0]]  = np.where(np.sqrt(R2)<=radiusCut, ro, roInf)
    init_sol.base['z0']['0'][vlist[1]] = np.where(np.sqrt(R2)<=radiusCut, ro*u,␣
↪roInf*uInf)
    init_sol.base['z0']['0'][vlist[2]] = np.where(np.sqrt(R2)<=radiusCut, ro*v,␣
↪roInf*vInf)
    init_sol.base['z0']['0'][vlist[3]] = 0.
    init_sol.base['z0']['0'][vlist[4]] = np.where(np.sqrt(R2)<=radiusCut, roE, roEInf)

    return vlist


def TGVinit(init_sol):
    gamma = init_sol.gas['gamma']
    Rgas  = init_sol.gas['Rgas']
    Cp    = init_sol.gas['Cp']

    roInf = 1.
    TInf = 351.6
    presInf = 101325.
    u0 = 1
```

(continues on next page)

(continued from previous page)

```
    L = 2*np.pi

    X = init_sol.base['z0'].shared['x']
    Y = init_sol.base['z0'].shared['y']
    Z = init_sol.base['z0'].shared['z']

    u =  u0 * np.sin(X)*np.cos(Y)*np.cos(Z)
    v = -u0 * np.cos(X)*np.sin(Y)*np.cos(Z)
    w = 0.0

    roE = presInf / (gamma - 1.) + 0.5 * ( u*u + v*v + w*w) * roInf

    # Build solution array as [ncells, nvars, nSolPts] array.
    vlist = ['rho','rhou','rhov','rhow','rhoe']
    init_sol.base['z0']['0'][vlist[0]] = roInf
    init_sol.base['z0']['0'][vlist[1]] = roInf*u
    init_sol.base['z0']['0'][vlist[2]] = roInf*v
    init_sol.base['z0']['0'][vlist[3]] = 0.0
    init_sol.base['z0']['0'][vlist[4]] = roE

    return vlist


##################################################"
def main():

    jaguar_input_file = os.path.join('..', 'data', 'GMSH', 'input.txt')
    jaguar = Jaguar(jaguar_input_file)

    # READER
    # ------
    # Read the GMSH mesh
    r = antares.Reader('fmt_gmsh')
    r['filename'] = os.path.join('..', 'data', 'GMSH', 'cube_8.msh')
    base_in = r.read()
    base_in.attrs['filename'] = r['filename']

    # TREATMENT
    # ---------
    treatment = antares.Treatment('jaguarinit')

    treatment['base'] = base_in
    treatment['jag_config'] = jaguar
    treatment['init_func'] = densityVortexInitialization2D
    # treatment['init_func'] = TGVinit # TGV function may also be tested

    base_out = treatment.execute()

    # WRITER
    # ------
    w = antares.Writer('hdf_jaguar_restart')
    w['base'] = base_out
```

(continues on next page)

(continued from previous page)

```
    w['strategy'] = 'monoproc'
    w.dump()


if __name__ == "__main__":
    main()
```

High Order Tools

class antares.utils.high_order.**HighOrderTools**

LagrangeGaussLegendre(*p*)

cell_sol_interpolation(*sol_in*, *p_in*, *x_out*)

Interpolate sol_in, p_in on a mesh based on 1D isoparam location x_out

Arguments:

sol_in {array} – Input solution p_in {array} – array of order of each cell of sol_in x_out {array} – 1D location of new interpolation points in [0,1]

getDerLi1D(*x*)

Compute derivative of Lagrange basis. :x: input 1D points where data is known return: matrix : dL1 .. dLn

x1 [dL1(x1) dLn(x1)] .. [....] xn [dL1(xn) .. dLn(xn)]

getDerLi2D(*x*)

Compute the partial derivative values of the 2D Lagrange basis at its construction points x.

Detail: To match the jaguar decreasing x order we have to flip the Li(x) before the tensor product, ie: dLidx = flip(dLi(x)) * Li(y) dLidy = flip(Li(x)) * dLi(y) = Li(x) * dLi(y) because Li = np.eye = identity (np.eye is used instead of getLi1D because this is the result of the Lagrange basis on its contruction points.)

getDerLi3D(*x*)

Compute the partial derivative values of the 3D Lagrange basis at its construction points x.

Detail: To match the jaguar decreasing x order we have to flip the Li(x) before the tensor product, ie: dLidx = flip(dLi(x)) * Li(y) * Li(z) dLidy = flip(Li(x)) * dLi(y) * Li(z) = Li(x) * dLi(y) * Li(z) because Li = np.eye = identity dLidz = flip(Li(x)) * Li(y) * dLi(z) (np.eye is used instead of getLi1D because this is the result of the Lagrange basis on its contruction points.)

getLi1D(*x_in*, *x_out*)

getLi2D(*x_in*, *x_out*, *flatten='no'*)

Return a matrix to interpolate from 2D based on 1D locations x_in to 2D based on 1D locations x_out

getLi3D(*x_in*, *x_out*, *flatten='no'*)

Return a matrix to interpolate from 3D based on 1D locations x_in to 3D based on 1D locations x_out

ndim = 0

class antares.utils.high_order.**HighOrderMesh**(*base=None*)

Class to handle high order mesh with Jaguar.

What you can do: - read a gmsh hexa mesh (order 1 or 2) and write a new linear mesh with a given basis to xdmf format.

add_output_mesh(*basis*, *p_out*)

create_from_GMSH()

Read the GMSH mesh with antares and generate x,y(,z)_mesh arrays of input mesh vertices. Warning: this is still a monoproc function.

Set self.x/y/z_mesh arrays

get_names()

class antares.utils.high_order.**HighOrderSol**(*parent*)

Solution of a high order computation with several variables.

create_from_coprocessing(*mesh*)

Link a list of mesh to the solution and set the jagSP as default

Args:
mesh (HighOrderMesh): HighOrder Mesh instance.

create_from_init_function(*mesh*, *init_func*)

Apply init_func on mesh coordinates to generate a initial solution field.

Args:
mesh (OutputMesh): Instance of a OutputMesh init_func (function): Function taking in argument a HighOrderSol and returning an ordered list of variables.

set_active_mesh(*name*)

class antares.utils.high_order.**OutputMesh**(*parent*)

Class to handle high order mesh with Jaguar What you can do: - read a gmsh hexa mesh (order 1 or 2) and write a new linear mesh with a given basis to xdmf format. -

invert(*list1*, *list2*)

Given two maps (lists) from [0..N] to nodes, finds a permutations between them. :arg list1: a list of nodes. :arg list2: a second list of nodes. :returns: a list of integers, l, such that list1[x] = list2[l[x]] source:https://github.com/firedrakeproject/firedrake/blob/661fc4597000ccb5658deab0fa99c3f7cab65ac3/firedrake/paraview

light_load()

Load only some scalar parameter from existing mesh file

set_output_basis(*basis*, *p=None*, *custom_1D_basis=None*)

Set the output basis. :basis: - 'jagSP' : Use SP as OP - 'jagSPextra' : use SP as OP and add extrapolated points at 0 and 1 (in iso coor) to have a continuous render - 'vtklag' : same as above but boundaries are extended to [0,1] - 'custom' : use custom_1D_basis as interpolation basis. :p: basis order, if None use the attached solution order

vtk_hex_local_to_cart(*orders*)

Produces a list of nodes for VTK's lagrange hex basis. :arg order: the three orders of the hex basis. :return a list of arrays of floats. source:https://github.com/firedrakeproject/firedrake/blob/661fc4597000ccb5658deab0fa99c3f7cab65ac3/firedrake/paraview

write()

write_h5()

Write high order mesh to different available format

write_xmf()

Write xmf file to read the mesh without solution

class antares.utils.high_order.**SolWriter**(*base=None*)

write_monoproc()

Write a single-file solution in single-proc mode.

Mainly used up to now to write init solution.

write_parallel(*restart_mode='h5py_ll'*)

Parallel HDF5 write: multi proc to single file

Args:

restart_mode (str, optional): See below. Defaults to 'h5py_ll'. basis (str, optional): Output basis. Coprocessing returns solution at solution points, but for visualization purpose 'vtklag' may be use to generate uniform basis and so be able to use vtk Lagrange cells. Defaults to 'jagSP'.

Several restart modes have been tested: - h5py_NOTsorted_highAPI: 'h5py_hl_ns' - h5py_sorted_highAPI: 'h5py_hl' - h5py_sorted_lowAPI: 'h5py_ll' - fortran_sorted_opt2: 'fortran' In sorted results, the efficiency is from best to worst: fortran > h5py_ll > h5py hl

Here, only h5py low level API (restart_mode=h5py_ll) is used because it has an efficiency near of the pure fortran, but it is much easier to modify and debug. For final best performance, a pure fortran code can be derived quite easily based of existing one from the python h5py_ll code.

class antares.utils.high_order.**Jaguar**(*input_fname*)

class CoprocOutput

Use the formalism <name> and <name>_size when you add array to be able to use jaguar_to_numpy function

conn

Structure/Union member

conn_size

Structure/Union member

coor

Structure/Union member

coor_size

Structure/Union member

data

Structure/Union member

data_size

Structure/Union member

extra_vars_size

Structure/Union member

iter

Structure/Union member

rms_vars_size

Structure/Union member

time

Structure/Union member

vars_names
Structure/Union member

vars_size
Structure/Union member

class CoprocParam
Parameters are relative to each partition

antares_io
Structure/Union member

cp_ite
Structure/Union member

cp_type
Structure/Union member

input_file
Structure/Union member

input_file_size
Structure/Union member

nTot_OP
Structure/Union member

n_OP
Structure/Union member

n_OP_dir
Structure/Union member

n_extra_vars
Structure/Union member

n_rms_vars
Structure/Union member

ndim
Structure/Union member

need_mesh
Structure/Union member

nloc_SP
Structure/Union member

nloc_cell
Structure/Union member

ntot_SP
Structure/Union member

ntot_cell
Structure/Union member

nvars
Structure/Union member

class CoprocRestart

Use the formalism <name> and <name>_size when you add array to be able to use jaguar_to_numpy function

c_l2g

Structure/Union member

c_l2g_size

Structure/Union member

iter

Structure/Union member

maxPolynomialOrder

Structure/Union member

nloc_cell

Structure/Union member

nprocs

Structure/Union member

p

Structure/Union member

p_size

Structure/Union member

sol

Structure/Union member

sol_size

Structure/Union member

time

Structure/Union member

PostProcessing()

PreProcessing()

Run()

coprocessing(*cp_type*)

debug(*double_arr*, *int_arr*, *int_scal*)

Callback function to be used for quick debuging

Args:

arr (ct.POINTER): ctypes pointer passing from fortran, may be in or float size (ct.POINTER(ct.c_int)): ctypes pointer to c_int passing from fortran giving the size of arr

Usage:

1/ Add 'use coprocessing' in the fortran file you want to debug 2/ Add a 'call callback_debug(double_array, int_array, int_scal)' where you want.

Example of call on fortran side in subroutine computeAverageVars:

integer, dimension(2) :: tmp tmp(1) = Mesh%nTotSolutionPoints tmp(2) = userData%n_RMS_vars call callback_pydebug(RMS%averageVariable, tmp, 2)

Giving the following output for the COVO3D:
Iter Time deltaT 1 0.23119E-05 0.23119E-05 pydebug [64000 2] (64000, 2)

Then do whatever you want on python side (Antares treatment and output, etc.)

Warning: there is no optional argument here, you must pass 3 arguments on the fortran side. So you can:
A/ either on fortran side create a temporary variable to fit the debug function call B/ or on python side modify:

1/ the call of the debug function above 2/ the definition of ptrf2 = ct.CFUNCTYPE(..) in the self.Run() method.

`getLi1D`(*x_in*, *x_out*)

`getLi2D`(*x_in*, *x_out*)

`getLi3D`(*x_in*, *x_out*, *flatten='no'*)

Return a matrix to interpolate from 3D based on 1D locations x_in to 3D based on 1D locations x_out

`init_mesh_and_sol`(*additional_mesh={}*)

Create and initialize HighOrderMesh and HighOrderSol instance for coprocessing.

Args:

additional_mesh (list, optional): List of dictionnary to add optional output mesh basis.
Keys are 'type' and 'p', if 'p' is not created string input file p is used. Defaults to None.

`python_str_to_fortran`(*s*)

Source: http://degenerateconic.com/fortran-json-python/

`txt2ini`(*fname*)

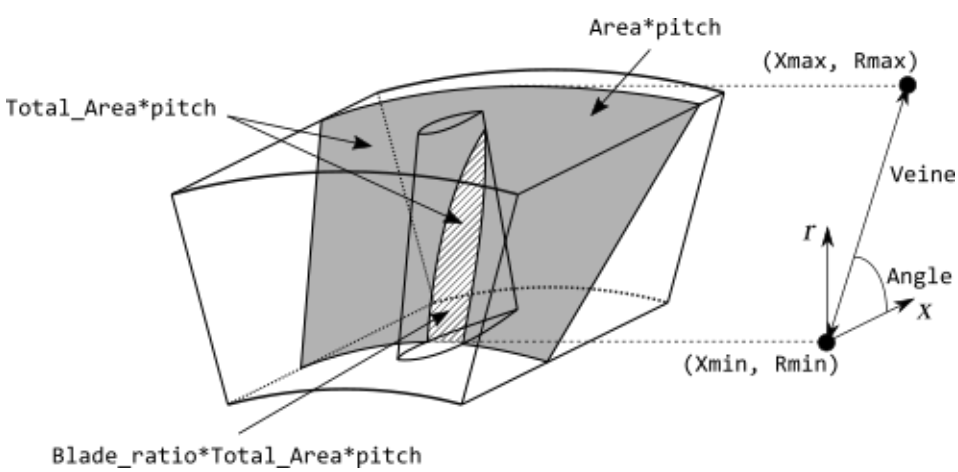
Convert txt input file in ini file format to be read by the parser later.

Thermodynamic Averages

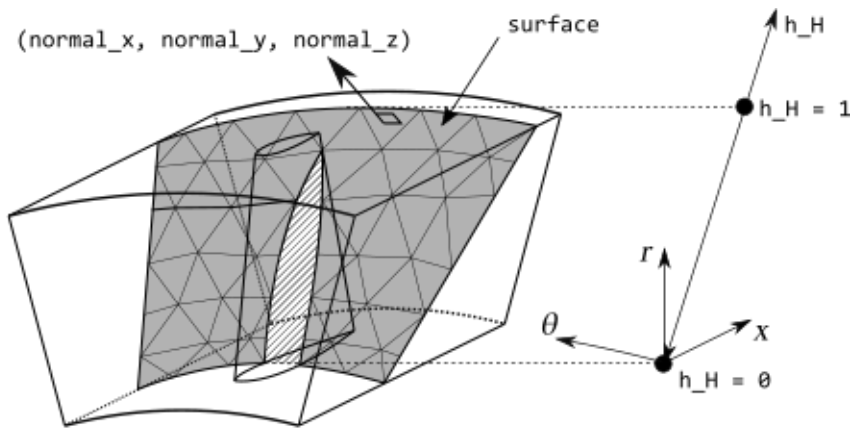
Geometrical properties of thermodynamic averages

Description

This treatment computes 0D and 2D geometrical properties of a 2D section resulting from a cut with a revolution surface around the x-axis of an axisymmetric configuration. The following picture illustrates the computed 0D properties:



The following picture illustrates the computed 2D properties:



Construction

```
import antares
myt = antares.Treatment('thermogeom')
```

Parameters

- **base: `Base`**
  The input **base** on which geometrical properties will be computed. Read section *Preconditions* to know the assumptions on this **base**. This input **base** is modified in-place. Read section *Postconditions* for more details.
- **cartesian_coordinates: `list(str)`, default=** *['x', 'y', 'z']*
  Ordened names of the cartesian coordinates.
- **cylindrical_coordinates: `list(str)`, default=** *['x', 'r', 'theta']*
  Names of the cylindrical coordinates: the axis of rotation (axial coordinate), the distance to this axis (radius), and the azimuth.
- **def_points: `list(tuple(float))`, default=** *None*
  List of coordinates of 2 points in the cylindrical coordinates system (only axis and radius). The first point gives the location of the hub. The second point gives the location of the shroud. These points do not necessarily belong to the hub nor to the shroud.

Preconditions

The treatment must be applied on a mono-zone **base** containing a 2D section resulting from a cut with a revolution surface around the 'x'-axis of an axisymmetric configuration. This *Zone* must contain only one *Instant* (steady-state).

The specified coordinates must be available at nodes. The rotation axis is the Cartesian first component of **cartesian_coordinates**.

The pitch of the row(s) is necessary : it can be available as an attribute called 'pitch' in the zone in mono-row case, but must be available at cells in multi-row case.

Postconditions

The input **base** is modified in-place.

The **base** contains cartesian coordinates of normals, surfaces (see 'base.compute_cell_normal' for naming conventions) and cylindrical coordinates at cells (same name than **cylindrical_coordinates**) and h/H local to the surface (called 'h_H') at nodes. The **base** is extended with one attribute named '0D/Geometry', which is a dictionary with variables:

- **Xmin, Rmin**
  Coordinates (in the unit of the input data) of the hub point in the (x, r) plane.
- **Xmax, Rmax**
  Coordinates (in the unit of the input data) of the shroud point in the (x, r) plane.
- **Veine**
  Length (in the unit of the input data) between the hub and the shroud in the surface.
- **Angle**
  Angle (in degrees) between the x-axis and the projection of the surface in the (x, r) plane.
- **Area**
  The area (in the unit of the input data) of the surface on 360 degrees.
- **Blade_ratio**
  Ratio of (in %) of the blade area on the surface with blades.
- **Normal_x, Normal_r**
  Coordinates of a unit normal of the surface in cylindrical coordinates.

Main functions

`class antares.treatment.turbomachine.TreatmentThermoGeom.TreatmentThermoGeom`

`execute()`

Compute the geometrical values of thermodynamic averages.

Thermodynamic Spatial Average of type 0

Description

This treatment computes the spatial mean of thermodynamic quantities with a type-0 formula.

It must be applied on 2D surfaces resulting from a revolution cut of an axisymmetric mono-row or multi-row configuration.

This average is recommended for unsteady flows. Then, the input quantities should come from an instantaneous flow solution.

Construction

```
import antares
myt = antares.Treatment('thermo0')
```

Parameters

- **base: `Base`**
  The base on which the treatment will be applied.
- **cylindrical_coordinates: `list(str)`, default=** ***['x', 'r', 'theta']***
  The ordered coordinate names of the cylindrical system.
- **conservative: `list(str)`, default=** ***['rho', 'rhou', 'rhov', 'rhow', 'rhoE']***
  Names of conservative variables: density, momentum components in the cartesian coordinate system, energy stagnation density. These quantities are expressed in the relative (rotating) frame.
- **ref_values: `(float, float)`, default=** ***None***
  Reference values (total pressure, total temperature) for averaging. These values may be obtained in a previous section.

Preconditions

The treatment must be applied on a mono-zone **base** containing a 2D section resulting from a cut with a revolution surface around the 'x'-axis of an axisymmetric configuration. This `Zone` must contain only one `Instant` (steady-state).

The specified cylindrical coordinates must be available at nodes. The rotation axis is the given by the first component of **cylindrical_coordinates**.

Four constants are necessary for the computation: two gas properties (ideal gas constant and specific heat ratio) and two row properties (rigid rotation of the rows in rad/s and pitch in rad). The gas properties must be available either as attributes or at cells, named respectively 'Rgas' or 'Rgaz' or 'R_gas' and 'gamma'. These quantities are assumed constant: if there are taken at cells, only one value is kept within the computations. The row properties can be available as attributes in mono-row case, but must be available at cells in multi-row case.

The **conservative** variables must be available at nodes or cells and must be expressed with the relative velocity formulation in the cartesian coordinate system. Psta, Pta, Tsta, Tta, Ttr, alpha, beta, phi, Ma, Mr must be available at cells.

The *`antares.treatment.turbomachine.TreatmentThermoGeom.TreatmentThermoGeom`* (page 226) must have been called beforehand. Then, the input base must contain the attribute '0D/Geometry'.

Postconditions

The input **base** is returned, extended with two attributes named '0D/Moyenne#Steady' and '0D/Moyenne0#Steady'.

The attribute '0D/Moyenne0#Steady' is a dictionary with variables:

- **Xmin, Rmin**
  Coordinates (in the unit of the input data) of the hub point in the (x, r) plane.
- **Xmax, Rmax**
  Coordinates (in the unit of the input data) of the shroud point in the (x, r) plane.

- **Veine**
  Length (in the unit of the input data) between the hub and the shroud in the surface.
- **Angle**
  Angle (in degrees) between the x-axis and the projection of the surface in the (x, r) plane.
- **Area**
  The area (in the unit of the input data) of the surface on 360 degrees.
- **Ep_aube**
  Ratio of (in %) of the blade area on the surface with blades.
- **SDebit**
  Signed instantaneous massflow rate in the section (kg/s) (integral of density*normal velocity to the surface).
- **Debit**
  Absolute instantaneous massflow rate in the section (kg/s) (|integral of density*normal velocity to the surface|).
- **retour**
  Reverse instantaneous massflow rate (between 0 and 100) defined as 100*((surface integral of absolute massflow rate) - (massflow rate through the oriented surface)) / (surface integral of absolute massflow rate).
- **Gcorrige**
- **Greduit**
- **alpha**
  arctan2(tangential velocity, meridional velocity norm) (in degree).
- **beta**
  arctan2(rotating tangential velocity, meridional velocity norm) (in degree).
- **phi**
  arctan2(radial velocity, axial velocity) (in degree).
- **Mv**
  Instantaneous absolute Mach number built from spatial mean values.
- **Mw**
  Instantaneous relative Mach number built from spatial mean values.
- **TIR_Tt1**
- **PI_Pt1**
- **TI_Tt1**
- **Pt1 (Pa)**
- **Tt1 (K)**
- **Ps_Pt1**
- **Ps(sect)**
  Static pressure (spatial integral of Ps weighted by the surface).
- **ETAis**
- **ETApoly**

The Instant contained in the input **base** is extended with variables at cells:

- **Vn**
  Normal velocity in the absolute frame.

- **massflow**
  Massflow rate.
- **Tsta**
  Static temperature.
- **Tta**
  Total temperature in the absolute frame.
- **Ttr**
  Total temperature in the relative frame.
- **Psta**
  Static pressure.
- **Pta**
  Total pressure in the absolute frame.
- **alpha**
  arctan2(tangential velocity, meridional velocity norm) (in degree).
- **beta**
  arctan2(rotating tangential velocity, meridional velocity norm) (in degree).
- **phi**
  arctan2(radial velocity, axial velocity) (in degree)
- **Ma**
  Absolute Mach number.
- **Mr**
  Relative Mach number.

Main functions

class `antares.treatment.turbomachine.TreatmentThermo0.`**`TreatmentThermo0`**

`execute``()`

Compute the thermodynamic average of type 0 on a surface.

Simple Thermodynamic Average

Standard Thermodynamic Average.

This treatment computes the spatial average of thermodynamic quantities over a 2D surface. The weighting can be made with surface or massflowrate.

The specific gas constant is hard-coded: Rgaz = 287.053 [J/kg/K].

The Heat capacity ratio is hard-coded: gamma = 1.4 [-].

Parameters

- **base: `Base`**
  The base on which the treatment will be applied. It is modified in-place.
- **coordinates:** ***list(str)***
  The ordered names of the mesh coordinates.
- **conservative:** ***list(str)*****, default=** ***['rho', 'rhou', 'rhov', 'rhow', 'rhoE']***
  Names of conservative variables. example: ['Density', 'MomentumX', 'MomentumY', 'MomentumZ', 'EnergyStagnationDensity']
- **avg_type:** ***str in ['surface', 'massflowrate']*****, default=** ***massflowrate***
  Type of space averaging.

Preconditions

The **base** must represent a revolution surface around the 'x'-axis of an axisymmetric configuration. The conservative variables must be computed at the cell location.

Postconditions

The cell normal and cell centroids will be computed during the process and returned in the input base. The treatment only considers the first instant of zones from the input **base** to compute the averages of variables. Other instants are ignored. The treatment returns the input base extended with an attribute named `average`. The attribute is a dictionary with variables *Q*, *rho*, *Vn*, *Vt1*, *Vt2*, *Ps*, *Ts*, and *Tt*.

- **Q:** $\int \rho V_n dS$
  mass flow rate through the surface

Names of means of thermodynamic values:

- **Tt**
  Total temperature.
- **Ps**
  Static pressure.
- **Ts**
  Static temperature.
- **rho**
  Density.
- **Vn**
  Normal velocity component.
- **Vt1**
  Tangential velocity component 1.
- **Vt2**
  Tangential velocity component 2.

Main functions

class antares.treatment.turbomachine.TreatmentThermo1.**TreatmentThermo1**

execute()

Compute the standard thermodynamic average.

Returns
the input base with an attribute named `average`. The attribute is a dictionary with variables *Q*, *rho*, *Vn*, *Vt1*, *Vt2*, *Ps*, *Ts*, and *Tt*

Return type
`Base`

Thermodynamic Spatial Average of type 7

Description

This treatment computes the spatial mean of thermodynamic quantities with a type-7 formula. This formula is a weighted mean of 5 quantities. The weight may be the mass flow or the area.

It must be applied on 2D surfaces resulting from a revolution cut of an axisymmetric mono-row or multi-row configuration.

This average is recommended for unsteady flows. Then, the input quantities should come from an instantaneous flow solution.

Construction

```
import antares
myt = antares.Treatment('thermo7')
```

Parameters

- **base: `Base`**
  The base on which the treatment will be applied.
- **cylindrical_coordinates: `list(str)`, default= *['x', 'r', 'theta']***
  The ordered coordinate names of the cylindrical system.
- **conservative: `list(str)`, default= *['rho', 'rhou', 'rhov', 'rhow', 'rhoE']***
  Names of conservative variables: density, momentum components in cartesian coordinates, energy stagnation density.
- **velocity_formulation: `str`, default= *'relative'***
  If **velocity_formulation** is 'relative', then the **conservative** quantities are supposed to be relative quantities in the relative (rotating) frame. If **velocity_formulation** is 'absolute', then the **conservative** quantities are supposed to be absolute quantities in the rotating frame.
- **ref_values: `(float, float)`, default= *None***
  Reference values (total pressure, total temperature) for averaging. These values may be obtained in a previous section.

- **weights: `str` in *['massflow', 'area']*, default= *'massflow'***
  Type of weighting to use for averaging.
- **label: `str`, default= *'0'***
  Label to append to '0D/Moyenne7#' for the resulting attribute name.
- **dmin: `float`, default= *400.e-4***
  This parameter is only used for logging messages if Q_abs/int_area < dmin. Its unit is kg/s. See *Logging Messages* (page 380) to activate logging messages.

Preconditions

The treatment must be applied on a mono-zone **base** containing a 2D section resulting from a cut with a revolution surface around the 'x'-axis of an axisymmetric configuration. This `Zone` must contain only one `Instant` (steady-state).

The specified cylindrical coordinates must be available at nodes. The rotation axis is the given by the first component of **cylindrical_coordinates**.

Four constants are necessary for the computation: two gas properties (ideal gas constant and specific heat ratio) and two row properties (rigid rotation of the rows in rad/s and pitch in rad). The gas properties must be available either as attributes or at cells, named respectively 'Rgas' or 'Rgaz' or 'R_gas' and 'gamma'. These quantities are assumed constant: if there are taken at cells, only one value is kept within the computations. The row properties can be available as attributes in mono-row case, but must be available at cells in multi-row case.

The **conservative** variables must be available at nodes or cells and must be expressed with the relative velocity formulation in the cartesian coordinate system.

The `antares.treatment.turbomachine.TreatmentThermoGeom.TreatmentThermoGeom` (page 226) must have been called beforehand. Then, the input base must contain the attribute '0D/Geometry'.

Postconditions

The input **base** is returned, extended with two attributes named '0D/Moyenne#Steady' and '0D/Moyenne7#<label>'.

The attribute '0D/Moyenne#Steady' is a dictionary with variables:

- **Xmin, Rmin**
  Coordinates (in the unit of the input data) of the hub point in the (x, r) plane.
- **Xmax, Rmax**
  Coordinates (in the unit of the input data) of the shroud point in the (x, r) plane.
- **Veine**
  Length (in the unit of the input data) between the hub and the shroud in the surface.
- **Angle**
  Angle (in degrees) between the x-axis and the projection of the surface in the (x, r) plane.

The attribute '0D/Moyenne7#<label>' is a dictionary with variables:

- **Debit**
  Absolute instantaneous massflow rate in the section (kg/s) (|integral of density*normal velocity to the surface|).
- **retour**
  Reverse instantaneous massflow rate (between 0 and 100) defined as 100*((surface integral of absolute massflow rate) - (massflow rate through the oriented surface)) / (surface integral of absolute massflow rate).

- **Gcorrige**
- **Greduit**
- **alpha**
  arctan2(tangential velocity, meridional velocity norm) (in degree).
- **VelocityCylindricalX**
  Axial absolute velocity (spatial integral of Vx weighted by the instantaneous massflow rate).
- **VelocityCylindricalR**
  Radial absolute velocity (spatial integral of Vt weighted by the instantaneous massflow rate).
- **VelocityCylindricalTheta**
  Tangential absolute velocity (spatial integral of Vr weighted by the instantaneous massflow rate).
- **Mv**
  Instantaneous absolute Mach number built from spatial mean values.
- **Ts**
  Static temperature built from integrals.
- **TI**
  Absolute total temperature (spatial integral of Tta weighted by the instantaneous massflow rate).
- **Ps**
  Static pressure built from integrals.
- **Ps(sect)**
  Static pressure (spatial integral of Ps weighted by the surface).
- **PI**
  Absolute total pressure built from integrals.
- **PI(massflow)**
  Absolute total pressure (spatial integral of Pta weighted by the instantaneous massflow rate).
- **S_std**
  Entropy (spatial integral of entropy weighted by the instantaneous massflow rate).

The Instant contained in the input **base** is extended with variables at cells:

- **VelocityX**
  Velocity in the first **coordinate** direction in the absolute frame.
- **VelocityY**
  Velocity in the second **coordinate** direction in the absolute frame.
- **VelocityZ**
  Velocity in the third **coordinate** direction in the absolute frame.
- **Vr**
  Radial velocity in the absolute frame.
- **Vt**
  Tangential velocity in the absolute frame.
- **Vn**
  Normal velocity in the absolute frame.
- **Temperature**
  Static temperature.
- **Tta**
  Total temperature in the absolute frame.

- **Ttr**
  Total temperature in the relative frame.
- **Pressure**
  Static pressure.
- **Pta**
  Total pressure in the absolute frame.
- **Ptr**
  Total pressure in the relative frame.
- **Entropy**
  Entropy production.

Main functions

`class antares.treatment.turbomachine.TreatmentThermo7.TreatmentThermo7`

`execute()`
Compute the thermodynamic average.

Time-averaged Thermodynamic Spatial Average of Type 7

Description

This treatment computes the time-averaged spatial mean of thermodynamic quantities with a type-7 formula. This formula is a weighted mean of 5 quantities. The weight may be the mass flow or the area.

This average is recommended for unsteady flows. Then, the input quantities should include many snapshots of an unsteady flow.

Construction

```
import antares
myt = antares.Treatment('thermo7timeaverage')
```

Parameters

- **base: `Base`**
  The Base that contains the time snapshots of the unsteady flow.
- **coordinates: `list(str)`, default=** ***['x', 'y', 'z']***
  The ordered coordinate names of the Cartesian system.
- **conservative: `list(str)`, default=** ***['rho', 'rhou', 'rhov', 'rhow', 'rhoE']***
  Names of conservative variables: density, momentum components in cartesian coordinates, energy stagnation density.
- **velocity_formulation: `str`, default=** ***'relative'***
  If **velocity_formulation** is 'relative', then the **conservative** quantities are supposed to be relative quantities in the relative (rotating) frame. If **velocity_formulation** is 'absolute', then the **conservative** quantities are supposed to be absolute quantities in the rotating frame.

- **ref_values: `(float, float)`, default=** ***None***
  Reference values (total pressure, total temperature) for averaging. These values may be obtained in a previous section.
- **weights: `str` in** ***['massflow', 'area']*****, default=** ***'massflow'***
  Type of weighting to use for averaging.
- **def_points: `list(tuple(float))`, default=** ***None***
  list of coordinates of 2 points in the cylindrical coordinate system. The first point gives the location of the hub. The second point gives the location of the shroud.

Preconditions

It must be applied on 2D **plane** sections resulting from a slice of an axisymmetric configuration. The cylindrical coordinate system must already be computed. The coordinates are expressed in the cylindrical coordinate system (x, r, t). The rotation axis is the Cartesian 'x'-coordinate.

Postconditions

The input **base** is returned, extended with two attributes named '0D/Moyenne#Steady' and '0D/Moyenne7#TimeAverage'.

The attribute '0D/Moyenne#Steady' is a dictionary with variables:

- **Xmin, Rmin**
  Coordinates of the hub point in the meridional plane. (in m/mm, given by the input data)
- **Xmax, Rmax**
  Coordinates of the shroud point in the meridional plane. (in m/mm, given by the input data)
- **Veine**
  Length between the hub and the shroud.
- **Angle**
  Angle between the x-axis and the (x,r) line (in degree).

The attribute '0D/Moyenne7#TimeAverage' is a dictionary with variables:

- **Debit**
  time-averaged massflow rate in the section (kg/s) (integral of density * normal velocity to the surface) * number of sector over 360 degrees (kg/s)
- **Mv**
  time-averaged absolute Mach number built from spatial mean values
- **VelocityCylindricalX**
  axial absolute velocity (spatial integral of Vx weighted by the instantaneous massflow rate)
- **VelocityCylindricalR**
  radial absolute velocity (spatial integral of Vt weighted by the instantaneous massflow rate)
- **VelocityCylindricalTheta**
  tangential absolute velocity (spatial integral of Vr weighted by the instantaneous massflow rate)
- **TI**
  absolute total temperature (spatial integral of Tta weighted by the instantaneous massflow rate)
- **PI**
  absolute total pressure built from integrals

- **PI(massflow)**
  absolute total pressure (spatial integral of Pta weighted by the instantaneous massflow rate)
- **S_std**
  entropy (spatial integral of entropy weighted by the instantaneous massflow rate)
- **Ps**
  static pressure built from integrals
- **Ts**
  static temperature built from integrals
- **Ps(sect)**
  static pressure (spatial integral of Ps weighted by the surface)

Main functions

class
`antares.treatment.turbomachine.TreatmentThermo7TimeAverage.`**`TreatmentThermo7TimeAverage`**

`execute``()`

Compute the thermodynamic mean average of type 7 (time-averaged).

Choro-averaged Thermodynamic Spatial Average of Type 7

Thermodynamic Spatial Average of type 7 with Chorochronic Averaging.

see `antares.treatment.TreatmentChoroReconstruct.TreatmentChoroReconstruct` for the chorochronic reconstruction.

This treatment computes the time-averaged spatial mean of thermodynamic quantities with a type-7 formula. This formula is a weighted mean of 5 quantities. The weight may be the mass flow or the area.

This average is recommended for unsteady flows. Then, the input quantities should include many snapshots of an unsteady flow.

It must be applied on 2D **plane** sections resulting from a slice of an axisymmetric configuration. The coordinates are expressed in the cylindrical coordinate system (x, r, t). The rotation axis is the Cartesian 'x'-coordinate.

Parameters

- **base: `Base`**
  The base on which the treatment will be applied.
- **coordinates:** ***list(str)***
  The ordered coordinate names of the Cartesian system. example: ['x', 'y', 'z']
- **conservative:** ***list(str)*****, default=** ***['rho', 'rhou', 'rhov', 'rhow', 'rhoE']***
  Names of conservative variables. example: ['ro', 'rovx', 'rovy', 'rovz', 'roE']
- **ref_values:** ***(float, float)*****, default=** ***None***
  Reference values (total pressure, total temperature) for averaging. This may be the values obtained in a previous section.
- **weights:** ***str*****, default=** ***'massflow'***
  Weights for the averaging. Available massflow (default) or area.

- **def_points:** ***list(tuple(float))*****, default=** ***None***
  List of coordinates of 2 points in the cylindrical coordinate system. The first point gives the location of the hub. The second point gives the location of the shroud.
- **extracts_step:** ***int*****, default=** ***1***
  The number of time iterations between two instants of the base.
- **nb_ite_rot:** ***int***
  The number of time iterations to describe a complete rotation.
- **nb_blade:** ***int or 'in_attr'*****, default=** ***'in_attr'***
  Number of blades of the current row. If 'in_attr', then each zone of the base must have an attribute 'nb_blade'.
- **nb_blade_opp:** ***int or list(int) or 'in_attr'*****, default=** ***'in_attr'***
  Number of blades of the opposite row (or list of numbers of blades of the opposite rows in the case of multiple opposite rows). If 'in_attr', then each zone of the base must have an attribute 'nb_blade_opp'.
- **omega_opp:** ***float or list(float) or 'in_attr'*****, default=** ***'in_attr'***
  Rotation speed of the opposite row (or list of rotation speeds of the opposite rows in the case of multiple opposite rows) expressed in radians per second. If 'in_attr', then each zone of the base must have an attribute 'omega_opp'.
- **type:** ***str*****, default=** ***'fourier'***
  The type of approach ('fourier' or 'least_squares') used to determine the amplitudes and phases of each interaction mode. If 'fourier', a DFT is performed: in the case of a single opposite row, the base should contain enough instants to cover the associated blade passage (**'nb_ite_rot' / 'extracts_step' / 'nb_blade_opp'**); in the case of multiple opposite rows, the base should contain enough instants to cover a complete rotation (**'nb_ite_rot' / 'extracts_step'**). If 'least_squares', a least-squares approach is used. For robustness and accuracy reasons, it is advised to have enough instants in the base to cover at least two blade passages (2 * **'nb_ite_rot' / 'extracts_step' / 'nb_blade_opp'**).

Preconditions

The conservative variables must be expressed with the relative velocity formulation in the Cartesian coordinate system. The cylindrical coordinate system must already be computed.

Main functions

class antares.treatment.turbomachine.TreatmentThermo7ChoroAverage.**TreatmentThermo7ChoroAverage**

execute()

Compute the thermodynamic mean average of type 7 (choro-averaged).

Average of type 7 with chorochronic DFT averaging.

Returns
the input base extended with 2 attributes named '0D/Moyenne#Steady', '0D/Moyenne7#TimeAverage'

Return type
Base

The attribute '0D/Moyenne#Steady' is a dictionary with variables:

Xmin, Rmin: coordinates of the hub point in the meridional plane. (in m/mm, given by the input data)

Xmax, Rmax: coordinates of the shroud point in the meridional plane. (in m/mm, given by the input data)

Variables

- **`Veine`** – Length between the hub and the shroud.
- **`Angle`** – Angle between the x-axis and the (x,r) line (in degree).

The attribute '0D/Moyenne7#TimeAverage' is a dictionary with variables:

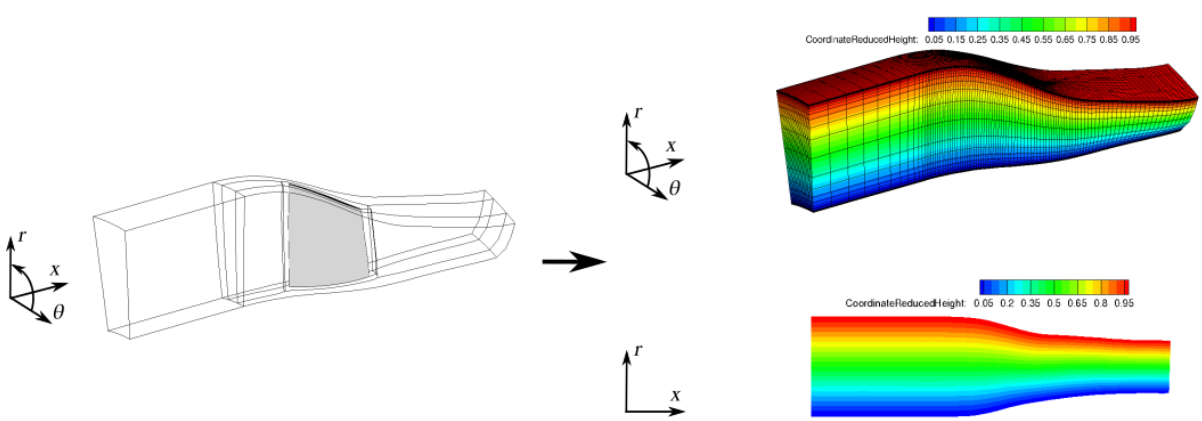
Variables

- **`Debit`** – time-averaged massflow rate in the section (kg/s) (integral of density* normal velocity to the surface) * number of sector over 360 degrees (kg/s)
- **`Mv`** – time-averaged absolute Mach number built from spatial mean values
- **`VelocityCylindricalX`** – axial absolute velocity (spatial integral of Vx weighted by the instantaneous massflow rate)
- **`VelocityCylindricalR`** – radial absolute velocity (spatial integral of Vt weighted by the instantaneous massflow rate)
- **`VelocityCylindricalTheta`** – tangential absolute velocity (spatial integral of Vr weighted by the instantaneous massflow rate)
- **`TI`** – absolute total temperature (spatial integral of Tta weighted by the instantaneous massflow rate)
- **`PI`** – absolute total pressure built from integrals
- **`PI(massflow)`** – absolute total pressure (spatial integral of Pta weighted by the instantaneous massflow rate)
- **`S_std`** – entropy (spatial integral of entropy weighted by the instantaneous massflow rate)
- **`Ps`** – static pressure built from integrals
- **`Ts`** – static temperature built from integrals
- **`Ps(sect)`** – static pressure (spatial integral of Ps weighted by the surface)

h/H parametrization

Description

Parameterize the grid.



Construction

```
import antares
myt = antares.Treatment('hh')
```

Parameters

- **base: `Base`**
  The input base.
- **families: `list`**
  List of family names associated to the turbomachine rows.

  Example: ['ROW1', 'ROW2', 'ROW3', 'ROW4']
- **hub_pts: `ndarray`**
  Points of the meridional hub line. May be computed with *antares.treatment.turbomachine.TreatmentMeridionalLine* (page 243).
- **shroud_pts: `ndarray`**
  Points of the meridional hub line. May be computed with *antares.treatment.turbomachine.TreatmentMeridionalLine* (page 243).
- **number_of_heights: `int`, default= *5***
  Number of points for the CoordinateReducedHeight direction. **number_of_heights** + 2 is one dimension of the 2D parameterization grid.
- **dmax: `float`, default= *1.0e-05***
  Maximum distance in metre between two points in a spline discretization.
- **precision: `float`, default= *1.0e-05***
  Maximum distance in metre between two points in a spline discretization.
- **extension: `float`, default= *10.0e-03***
  Length extension of meridional splines to compute parametrization grid.
- **coordinates: `list(str)`, default= *antares.core.GlobalVar.coordinates***
  The ordered names of the mesh cartesian coordinates.
- **flow_type: `str` in ['axial', 'other'], default= *'axial'***
  Characterize the flow in the turbomachine. Used to set extension points of hub and shroud meridional lines (see key **extension**).
- **output_dir: `str`, default= *None***
  Directory name for output files. If None, then no output files are written.
- **coprocessing: `bool`, default= *False***
  Deactivate code lines if coprocessing with a CFD solver. This avoids a conflict with the symbol *splint* that appears in both the CFD code and scipy.

Preconditions

Postconditions

The treatment returns

- the input **base** completed with the h/H parametrization variables at node location
  - CoordinateReducedMeridional
  - CoordinateSigma
  - CoordinateHeightFromHub
  - CoordinateHeightToShroud
  - CoordinateReducedHeight
- the `Base` of the 2D structured parametrization grid

Example

```
import antares
myt = antares.Treatment('hh')
myt['base'] = base
myt['families'] = ['ROW1']
myt['hub_pts'] = np.array()
myt['shroud_pts'] = np.array()
hhbase, parambase = myt.execute()
```

Main functions

class antares.treatment.turbomachine.TreatmenthH.**TreatmenthH**

execute()

Parameterize the grid, and add h/H variables.

Returns
the base containing h/H parametrization variables

Return type
`Base`

Returns
the 2D structured parametrization grid

Return type
`Base`

Example

```
import os

import antares

import numpy as np

import matplotlib as mpl
import matplotlib.pyplot as plt
mpl.use('Agg')
font = {'family':'serif','weight':'medium','size':40}
mpl.rc('font', **font)
mpl.rcParams['axes.linewidth'] = 2.0

output = 'OUTPUT'
if not os.path.isdir(output):
    os.makedirs(output)

r = antares.Reader('bin_tp')
r['filename'] = os.path.join('..', 'data', 'ROTOR37', 'ELSA_CASE', 'MESH', 'mesh_<zone>.
↪dat')
r['zone_prefix'] = 'Block'
r['topology_file'] = os.path.join('..', 'data', 'ROTOR37', 'ELSA_CASE', 'script_topo.py')
r['shared'] = True
base = r.read()

r = antares.Reader('bin_tp')
r['base'] = base
r['filename'] = os.path.join('..', 'data', 'ROTOR37', 'ELSA_CASE', 'FLOW', 'flow_<zone>.
↪dat')
r['zone_prefix'] = 'Block'
r['location'] = 'cell'
r.read()

print(base.families)

archi_fams = {
    'ROWS':   [['superblock_0000']],
    'HUB':    [['HUB']],
    'SHROUD': [['CASING']]
}

tre = antares.Treatment('MeridionalLine')
tre['base'] = base
tre['families'] = archi_fams
hub_points, shroud_points = tre.execute()

# Definition of the treatment
tr = antares.Treatment('hH')
tr['base'] = base
tr['families'] = ['superblock_0000']
```

(continues on next page)

(continued from previous page)

```
tr['hub_pts'] = hub_points
tr['shroud_pts'] = shroud_points
tr['extension'] = 0.1
tr['output_dir'] = output
hhbase, paramgrid = tr.execute()

print(hhbase[0][0])

writer = antares.Writer('bin_tp')
writer['filename'] = os.path.join(output, 'ex_hh.plt')
writer['base'] = hhbase
writer.dump()
writer = antares.Writer('bin_tp')
writer['filename'] = os.path.join(output, 'ex_paramgrid.plt')
writer['base'] = paramgrid
writer.dump()

archi_fams = {
    'ROWS':    [['superblock_0000']],
    'HUB':     [['HUB']],
    'SHROUD':  [['CASING']],
    'INLETS':  [['INLET']],
    'OUTLETS': [['OUTLET']],
    'BLADES':  [ {'SKIN': [ ['BLADE'] ],
                  'TIP': [ [] ]}
               ]
}

tre = antares.Treatment('meridionalview')
tre['base'] = base
tre['param_grid'] = paramgrid
tre['families'] = archi_fams
tre['hub_pts'] = hub_points
tre['shroud_pts'] = shroud_points
tre['extension'] = 0.1
tre['height_value'] = 0.1
component = tre.execute()

# ---------------------------------------
# Display geometry
# ---------------------------------------
fig = plt.figure(figsize=(30,20), dpi=300, facecolor='w', edgecolor='k')
ax = fig.add_subplot(111)
ax.set_aspect('equal', adjustable='datalim')
meridional_lines = {}
meridional_lines['Hub'] = hub_points
meridional_lines['Shroud'] = shroud_points
for idx, row in enumerate(component['Row']):
    for jdx, blade in enumerate(row['Blade']):
        list_of_points = blade['profiles']
        for i, profile_3D in enumerate(list_of_points):
            name = 'row_%d_blade_%d_%d' % (idx, jdx, i)
```

(continues on next page)

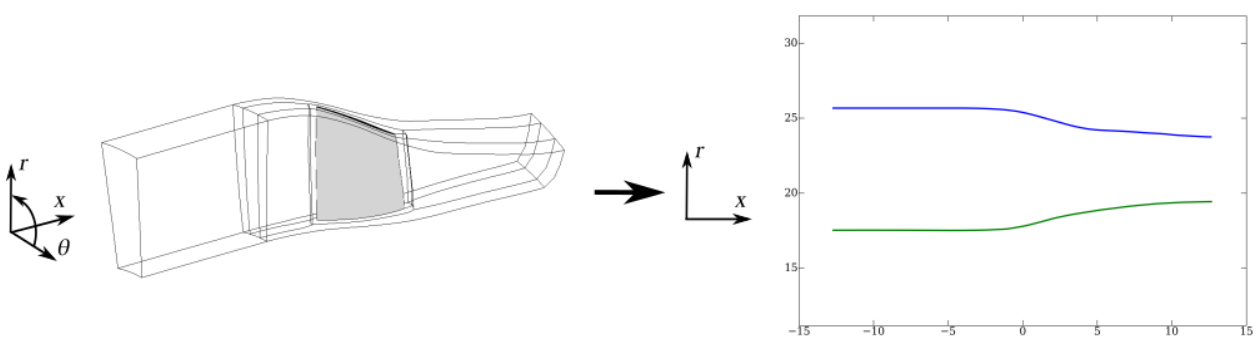
(continued from previous page)

```
            line = np.zeros((np.shape(profile_3D)[0], 2))
            line[:, 0] = profile_3D[:, 0]
            y = profile_3D[:, 1]
            z = profile_3D[:, 2]
            line[:, 1] = np.sqrt(y**2 + z**2)
            meridional_lines[name] = line
            i += 1

for part in meridional_lines.keys():
    ax.plot(meridional_lines[part][:, 0], meridional_lines[part][:, 1], linewidth=6)

for row in component['Row']:
    ax.plot(row['inletMeridionalPoints'][:, 0],  row['inletMeridionalPoints'][:, 1],␣
↪linewidth=6)
    ax.plot(row['outletMeridionalPoints'][:, 0], row['outletMeridionalPoints'][:, 1],␣
↪linewidth=6)
    for blade in row['Blade']:
        ax.plot(blade['LE'][:, 0], blade['LE'][:, 1], linewidth=6)
        ax.plot(blade['TE'][:, 0], blade['TE'][:, 1], linewidth=6)
        if 'rootMeridionalPoints' in blade:
            ax.plot(blade['rootMeridionalPoints'][:, 0],
                    blade['rootMeridionalPoints'][:, 1], linewidth=6)
        if 'tipMeridionalPoints' in blade:
            ax.plot(blade['tipMeridionalPoints'][:, 0],
                    blade['tipMeridionalPoints'][:, 1], linewidth=6)

ax.tick_params(which='major', width=2, length=20)
plt.savefig(os.path.join(output, 'meridional_view.png'), bbox_inches='tight')
plt.close()
```

Meridional Line

Description

Create hub and shroud meridional lines from hub and shroud mesh surfaces described by families in a tri-dimensional mesh.



Construction

```
import antares
myt = antares.Treatment('MeridionalLine')
```

Parameters

- **base: `Base`**
  The input base.
- **families: `dict`**
  Complex data structure that contains the family names associated to some turbomachine entities. The turbomachine entities described are the rows ('ROWS'), the hub part concerned by the rows ('HUB'), the shroud part concerned by the rows ('SHROUD'). Each entity (key of the dictionary) contains a list which size is the number of turbomachine rows. The rows must be ordered from the inlet to the outlet of the machine. This is the same order for the list elements. Each element of this list is another list containing the family names that characterize this entity in the specific row.

  Example: Description of a machine with 4 rows ordered from the inlet to the outlet of the machine.

  ```
  archi_fams = {
    'ROWS':    [['ROW1'],    ['ROW2'],    ['ROW3'],    ['ROW4']],
    'HUB':     [['HUB1'],    ['HUB2'],    ['HUB3'],    ['HUB4', 'STATIC_H4']],
    'SHROUD':  [['SHROUD1'], ['SHROUD2'], ['SHROUD3'], ['SHROUD4']]
  }
  ```
- **coordinates: `list(str)`, default=** ***antares.core.GlobalVar.coordinates***
  The ordered names of the mesh cartesian coordinates.

Preconditions

The input **base** must contain the families detailed in the parameter **families**.

Postconditions

The treatment returns:

- `ndarray` with points that define the hub meridional line.
- `ndarray` with points that define the shroud meridional line.

Example

```
import antares
myt = antares.Treatment('MeridionalLine')
myt['base'] = base
myt['families'] = archi_fams
hub_points, shroud_points = myt.execute()
```

Main functions

class antares.treatment.turbomachine.TreatmentMeridionalLine.**TreatmentMeridionalLine**

execute()

Compute hub and shroud meridional lines from mesh surfaces.

Returns
Points that define the hub meridional line

Return type
ndarray

Returns
Points that define the shroud meridional line

Return type
ndarray

Example

```
import os

import antares

import numpy as np

import matplotlib as mpl
import matplotlib.pyplot as plt
mpl.use('Agg')
font = {'family':'serif','weight':'medium','size':40}
mpl.rc('font', **font)
mpl.rcParams['axes.linewidth'] = 2.0

output = 'OUTPUT'
if not os.path.isdir(output):
    os.makedirs(output)

r = antares.Reader('bin_tp')
r['filename'] = os.path.join('..', 'data', 'ROTOR37', 'ELSA_CASE', 'MESH', 'mesh_<zone>.
↪dat')
r['zone_prefix'] = 'Block'
r['topology_file'] = os.path.join('..', 'data', 'ROTOR37', 'ELSA_CASE', 'script_topo.py')
r['shared'] = True
base = r.read()

r = antares.Reader('bin_tp')
r['base'] = base
r['filename'] = os.path.join('..', 'data', 'ROTOR37', 'ELSA_CASE', 'FLOW', 'flow_<zone>.
↪dat')
r['zone_prefix'] = 'Block'
r['location'] = 'cell'
r.read()
```

(continues on next page)

(continued from previous page)

```
print(base.families)

archi_fams = {
    'ROWS':   [['superblock_0000']],
    'HUB':    [['HUB']],
    'SHROUD': [['CASING']]
}

tre = antares.Treatment('MeridionalLine')
tre['base'] = base
tre['families'] = archi_fams
hub_points, shroud_points = tre.execute()

# Definition of the treatment
tr = antares.Treatment('hH')
tr['base'] = base
tr['families'] = ['superblock_0000']
tr['hub_pts'] = hub_points
tr['shroud_pts'] = shroud_points
tr['extension'] = 0.1
tr['output_dir'] = output
hhbase, paramgrid = tr.execute()

print(hhbase[0][0])

writer = antares.Writer('bin_tp')
writer['filename'] = os.path.join(output, 'ex_hh.plt')
writer['base'] = hhbase
writer.dump()
writer = antares.Writer('bin_tp')
writer['filename'] = os.path.join(output, 'ex_paramgrid.plt')
writer['base'] = paramgrid
writer.dump()

archi_fams = {
    'ROWS':    [['superblock_0000']],
    'HUB':     [['HUB']],
    'SHROUD':  [['CASING']],
    'INLETS':  [['INLET']],
    'OUTLETS': [['OUTLET']],
    'BLADES':  [ {'SKIN': [ ['BLADE'] ],
                  'TIP': [ [] ]}
               ]
}

tre = antares.Treatment('meridionalview')
tre['base'] = base
tre['param_grid'] = paramgrid
tre['families'] = archi_fams
tre['hub_pts'] = hub_points
tre['shroud_pts'] = shroud_points
tre['extension'] = 0.1
```

(continues on next page)

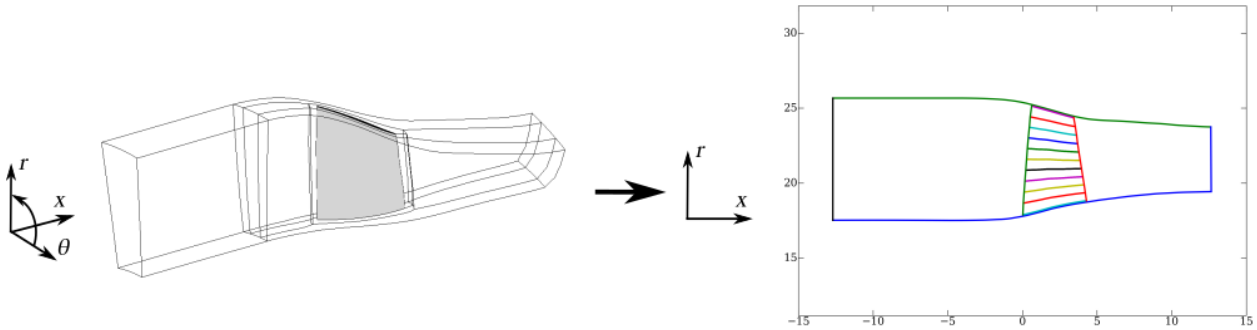
(continued from previous page)

```
tre['height_value'] = 0.1
component = tre.execute()

# ---------------------------------------
# Display geometry
# ---------------------------------------
fig = plt.figure(figsize=(30,20), dpi=300, facecolor='w', edgecolor='k')
ax = fig.add_subplot(111)
ax.set_aspect('equal', adjustable='datalim')
meridional_lines = {}
meridional_lines['Hub'] = hub_points
meridional_lines['Shroud'] = shroud_points
for idx, row in enumerate(component['Row']):
    for jdx, blade in enumerate(row['Blade']):
        list_of_points = blade['profiles']
        for i, profile_3D in enumerate(list_of_points):
            name = 'row_%d_blade_%d_%d' % (idx, jdx, i)
            line = np.zeros((np.shape(profile_3D)[0], 2))
            line[:, 0] = profile_3D[:, 0]
            y = profile_3D[:, 1]
            z = profile_3D[:, 2]
            line[:, 1] = np.sqrt(y**2 + z**2)
            meridional_lines[name] = line
            i += 1

for part in meridional_lines.keys():
    ax.plot(meridional_lines[part][:, 0], meridional_lines[part][:, 1], linewidth=6)

for row in component['Row']:
    ax.plot(row['inletMeridionalPoints'][:, 0],  row['inletMeridionalPoints'][:, 1],␣
↪linewidth=6)
    ax.plot(row['outletMeridionalPoints'][:, 0], row['outletMeridionalPoints'][:, 1],␣
↪linewidth=6)
    for blade in row['Blade']:
        ax.plot(blade['LE'][:, 0], blade['LE'][:, 1], linewidth=6)
        ax.plot(blade['TE'][:, 0], blade['TE'][:, 1], linewidth=6)
        if 'rootMeridionalPoints' in blade:
            ax.plot(blade['rootMeridionalPoints'][:, 0],
                    blade['rootMeridionalPoints'][:, 1], linewidth=6)
        if 'tipMeridionalPoints' in blade:
            ax.plot(blade['tipMeridionalPoints'][:, 0],
                    blade['tipMeridionalPoints'][:, 1], linewidth=6)

ax.tick_params(which='major', width=2, length=20)
plt.savefig(os.path.join(output, 'meridional_view.png'), bbox_inches='tight')
plt.close()
```

Meridional View

Description

Give a schematic view of a turbomachine in a meridional plane.



Parameters

- **base: `Base`**
  The input base.
- **families: `dict`**
  Complex data structure that contains the family names associated to some turbomachine entities. The turbomachine entities described are the rows ('ROWS'), the inlets of rows ('INLETS'), the outlets of rows ('OUTLETS'), the hub part concerned by the rows ('HUB'), the shroud part concerned by the rows ('SHROUD'), and the blades of the rows ('BLADES'). Each entity (key of the dictionary) except 'BLADES' contains a list which size is the number of turbomachine rows. The rows must be ordered from the inlet to the outlet of the machine. This is the same order for the list elements. Each element of this list is another list containing the family names that characterize this entity in the specific row. The 'BLADES' entity is a list of dictionary. Each dictionary contains two keys to describe the skin of the blades ('SKIN') and the tip part of the blades ('TIP').

  If 'HUB' and 'SHROUD' are not given, then **hub_pts** and **shroud_pts** are used.

  Example: Description of a machine with 4 rows ordered from the inlet to the outlet of the machine.

```
archi_fams = {
  'ROWS':    [['ROW1'],    ['ROW2'],    ['ROW3'],    ['ROW4']],
  'INLETS':  [['INLET'],   ['R1S1'],    ['S2R1'],    ['R2S2']],
  'OUTLETS': [['S1R1'],    ['R1S2'],    ['S2R2'],    ['OUTLET']],
  'HUB':     [['HUB1'],    ['HUB2'],    ['HUB3'],    ['HUB4', 'STATIC_H4']],
  'SHROUD':  [['SHROUD1'], ['SHROUD2'], ['SHROUD3'], ['SHROUD4']]
  'BLADES':  [ {'SKIN': [ ['BLADE1'] ],
                'TIP': [ [] ]},
               {'SKIN': [ ['BLADE2'] ],
                'TIP': [ [] ]},
               {'SKIN': [ ['BLADE3'] ],
                'TIP': [ [] ]},
               {'SKIN': [ ['BLADE4'] ],
                'TIP': [ [] ]}
             ]
}
```

- **hub_pts: `ndarray`**
  Points of the meridional hub line. May be computed with *antares.treatment.turbomachine.TreatmentMeridionalLine* (page 243).
- **shroud_pts: `ndarray`**
  Points of the meridional hub line. May be computed with *antares.treatment.turbomachine.TreatmentMeridionalLine* (page 243).
- **param_grid: `Base`**
  Base of the 2D structured grid with the parametrization. This base must be computed with *antares.treatment.turbomachine.TreatmenthH* (page 238).
- **number_of_heights: `int`, default= *5***
  Number of points for the CoordinateReducedHeight direction.
- **dmax: `float`, default= *1.0e-05***
  Maximum distance in metre between two points in a spline discretization.
- **precision: `float`, default= *1.0e-05***
  Maximum distance in metre between two points in a spline discretization.
- **extension: `float`, default= *10.0e-03***
  Length extension of meridional splines to compute parametrization grid.
- **coordinates: `list(str)`, default= *antares.core.GlobalVar.coordinates***
  The ordered names of the mesh cartesian coordinates.
- **height_value: `float`, default= *5.0e-04***
  Distance in metre from tip and root at which to start extracting the points for LE and TE (in reduced height coordinate).
- **le_te: `bool`, default= *True***
  Activation of leading and trailing edges detection
- **output_dir: `str`, default= *None***
  Directory name for output files. If None, then no output files are written.

Preconditions

The parametrization grid **param_grid** must have been computed with *antares.treatment.turbomachine.TreatmenthH.TreatmenthH* (page 240). To be consistent, common keys with the latter treatment must have the same values.

Postconditions

The treatment returns a data structure named `component`. This data structure contains some characteristics of the turbomachine.

Key values of the dictionary `component` are: **'Row'**

- **'Row': `list(dict)`**
  List of dictionaries containing information associated to rows. The list order is given by the input parameter **families**. Each element of the **'Row'** list is a dictionary with the following keys:
  - **'Blade': `dict`**
    The dictionary has the following keys:
    * **'rootMeridionalPoints': `ndarray`**
      Points of the meridional line of the root

    * **'tipMeridionalPoints': ndarray**
      Points of the meridional line of the tip
    * **'LE': ndarray**
      Points of the leading edge, output of *antares.treatment.turbomachine.TreatmentLETE* (page 251)
    * **'TE': ndarray**
      Points of the trailing edge, output of *antares.treatment.turbomachine.TreatmentLETE* (page 251)
    * **'profiles': list(11* ndarray)**
      Blade profiles, output of *antares.treatment.turbomachine.TreatmentLETE* (page 251)
  - **'hubMeridionalPoints': ndarray**
    Points that define the hub meridional line of the row.
  - **'shroudMeridionalPoints': ndarray**
    Points that define the shroud meridional line of the row.
  - **'inletMeridionalPoints': ndarray**
    Points that define the inlet meridional line of the row.
  - **'outletMeridionalPoints': ndarray**
    Points that define the outlet meridional line of the row.

Example

```
import antares
myt = antares.Treatment('meridionalview')
myt['base'] = base
myt['families'] = archi_fams
myt['hub_pts'] = np.array()
myt['shroud_pts'] = np.array()
component = myt.execute()
```

Main functions

class antares.treatment.turbomachine.TreatmentMeridionalView.**TreatmentMeridionalView**

execute()

Compute the meridional view.

Returns
the component data structure

Return type
dict

Leading Edge / Trailing Edge computation

Treatment Leading Edge / Trailing Edge computation

Parameters

- **base: `Base`**
  The base on which the treatment will be applied.
- **coordinates: *list(str)***
  The ordered names of the mesh coordinates.
- **deviation: *float*, default= *None***
  Deviation of the row in which the treated blade is located.
- **height_value: *float*, default= *5.0e-04***
  Distance in metre from tip and root at which to start extracting the points for LE and TE (in reduced height coordinate).

Preconditions

Surface base expected. A complete reduced coordinate parametrization must have been applied on the blade beforehand.

Main functions

class `antares.treatment.turbomachine.TreatmentLETE.`**`TreatmentLETE`**

`execute`()

Radial distribution

Compute the radial distribution by performing an azimuthal average over an 'x'-plane. The mass-flow variable is used as the averaging variable ($\rho v_x \partial\theta$)

Parameters

- **base: `Base`**
  The base must contain:
  - the mesh coordinates x, y, and z
  - the solution
  - 'hb_computation' as an `Base.attrs` (if HB/TSM type).
- **family_name: *str*, default= *None***
  The name of the family from which the percent will be computed.
- **r_percent: *tuple(float)*, default= *None***
  The radius value given as a percentage of the radius. The argument should be a tuple of min and max values. These limits the lower/upper bounds of the radial distribution. If not given, the lower/upper bounds of the radial distribution are computed.

- **x_value:** ***float*****, default=** ***None***
  The absolute position value of the plane.
- **r_value:** ***tuple(float)*****, default=** ***None***
  The radius value. The argument should be a tuple of min and max values.
- **vectors:** ***list(tuple(str)*****, default=** ***[]***
  If the base contains vectors, they must be rotated. It is assumed that they are expressed in cartesian coordinates.
- **var-equ:** ***list(str)*****, default=** ***[]***
  Compute these values/equations on the 'x'-plane. Values and equations must be ordered so as to avoid dependency issues.
- **num:** ***int*****, default=** ***100***
  The number of points in the radial distribution.

Preconditions

The coordinate variables must be available at node and the integration will be performed on all the other variables.

Main functions

class antares.treatment.turbomachine.TreatmentAzimuthalMean.**TreatmentAzimuthalMean**

execute()

Execute the treatment.

Returns

Return type
Base

Example

```
import os

if not os.path.isdir('OUTPUT'):
    os.makedirs('OUTPUT')

import numpy as np

from antares import Reader, Treatment, Writer

#

# Data can be downloaded from
# https://cerfacs.fr/antares/downloads/application1_tutorial_data.tgz

r = Reader('bin_tp')
r['filename'] = os.path.join('..', 'data', 'ROTOR37', 'ELSA_CASE', 'MESH',
                             'mesh_<zone>.dat')
r['zone_prefix'] = 'Block'
```

(continues on next page)

(continued from previous page)

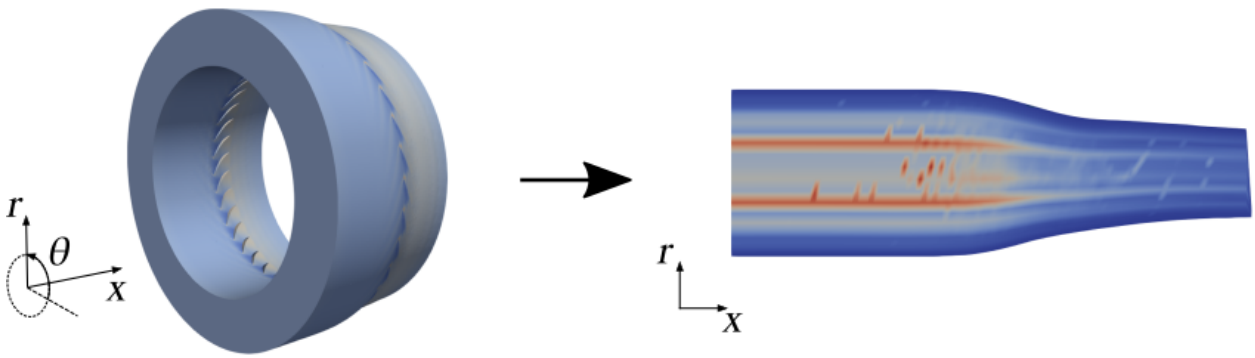
```
r['topology_file'] = os.path.join('..', 'data', 'ROTOR37', 'ELSA_CASE',
                                  'script_topo.py')
r['shared'] = True
base = r.read()
print(base.families)

r = Reader('bin_tp')
r['base'] = base
r['filename'] = os.path.join('..', 'data', 'ROTOR37', 'ELSA_CASE', 'FLOW',
                             'flow_<zone>.dat')
r['zone_prefix'] = 'Block'
r['location'] = 'cell'
r.read()

base.set_computer_model('internal')

# Needed for turbomachinery dedicated treatments
base.cell_to_node()
base = base.get_location('node')
print(base.families)

base.compute('psta')
base.compute('Pi')
base.compute('theta')
P0_INF = 1.9
base.compute('MachIs = (((%f/psta)**((gamma-1)/gamma)-1.) * (2./(gamma-1.))  )**0.5' %␣
↪P0_INF)

# Definition of the treatment
t = Treatment('azimuthalmean')
t['base'] = base
t['family_name'] = 'BLADE'
t['num'] = 60

writer = Writer('column')  # for 2D plot

# Azimuthal mean
res_dir = os.path.join('OUTPUT', 'AZ_MEAN')
if not os.path.isdir(res_dir):
    os.makedirs(res_dir)

NUM = 9
x = np.linspace(-12.5, 12.5, NUM)
for i in range(0, NUM):
    print('radial distribution at x = {}'.format(x[i]))

    t['x_value'] = x[i]
    azim_base = t.execute()

    writer['filename'] = os.path.join(res_dir, 'flow_azim_%i.plt' % x[i])
    writer['base'] = azim_base[:, :, ('R', 'rovx', 'vx', 'ro')]
    writer.dump()
```

Averaged Meridional Plane

Description

Compute an averaged meridional plane of a turbomachine at steady-state.

The azimuthal averages are performed using a surface thermodynamic average on small surfaces that are normal to the curvilinear abscissa of the machine.



Construction

```
import antares
myt = antares.Treatment('averagedmeridionalplane')
```

Parameters

- **base: `Base`**
  The input base on which the meridional plane is built.
- **cartesian_coordinates: `list(str)`, default=** *['x', 'y', 'z']*
  Ordened names of the cartesian coordinates.
- **merid_coordinates: `list(str)`, default=** *['x', 'r']*
  Names of the meridional plane coordinates: the axis of rotation (axial coordinate) and the distance to this axis (radius).
- **machine_coordinates: `list(str)`, default=** *['x', 'r', 'theta']*
  Names of three coordinates related to the machine: a curvilinear abscissa, a height coordinate and an azimuthal coordinate. It usually depends on the type of machine processed. The curvilinear abscissa is used as the normal to the small averaging surfaces.
- **out_coordinates: `list(str)`, default=** *['x', 'y', 'z']*
  Names of the coordinates in the output *Base*. The two first are the meridional plane coordinates.
- **out_variables: `list(str)`, default=** *[]*
  Names of the output averaged variables contained in the output *Base*.
- **cut_treatment: `Treatment`**
  A preconfigured cut treatment, used to perform cuts of the 3D volume within the algorithm.
- **thermo_treatment: `Treatment`**
  A preconfigured thermodynamic average treatment, used to average the output variables **merid_coordinates** and **out_variables** on 2D surfaces.

- **abscissa_nb: `int`, default= *50***
  Number of curvilinear abscissas (first component of **machine_coordinates**) taken to discretize the meridional plane.
- **abscissa_distrib: `str` in *['uniform', 'normal']*, default= *'uniform'***
  Distribution of the discretization of the curvilinear abscissa.
- **height_nb: `int`, default= *20***
  Number of heights (second component of **machine_coordinates**) taken to discretize the meridional plane.
- **height_distrib: `str` in *['uniform', 'normal']*, default= *'normal'***
  Distribution of the discretization of the height coordinate.

Preconditions

The input **base** must fulfill the preconditions of the input **thermo_treatment**, except the *mono-zone* requirement. The **base** is mono-instant (steady-state).

The **merid_coordinates** and the **machine_coordinates** must be available in the input **base** at nodes.

The given cut treatment must have an input called **base**, an input **type** which is the geometrical type of cut that can take the *'plane'* value, an input **coordinates**, the system of coordinates used, an input **normal**, the normal vector of the plane cut and an input **origin**, a point such that the cut plane passes through.

The given thermodynamic average must have an input called **base**, a 2D surface, an input **def_points** allowing to define the normal of the surface, and an output dictionnary called '0D/AveragedMeridionalPlane' containing the variables given in **merid_coordinates** and **out_variables**.

The **abscissa_nb** and the **height_nb** must be greater or equal to 1.

Postconditions

The output is a new mono-zone and mono-instant *Base* containing the computed meridional plane with coordinates called **out_coordinates** and variables called **out_variables** located at nodes.

The input **base** is unchanged.

Example

```
import antares
import numpy as np

# code that creates a Base fulfilling the preconditions
# it is a structured approximation of an annulus (r_min = 1.0, r_max = 2.0,
# pitch = pi/2) extruded from x = 0.0 to x = 1.0
# having Cartesian coordinates called 'x', 'y', 'z'
# having machine coordinates called 'x', 'r', 'theta'
base = antares.Base()
base.init()
base[0][0]['x'] = [[[0.0, 0.0], [0.0, 0.0]], [[1.0, 1.0], [1.0, 1.0]]]
base[0][0]['r'] = [[[1.0, 2.0], [1.0, 2.0]], [[1.0, 2.0], [1.0, 2.0]]]
base[0][0]['theta'] = [[[0.0, 0.0], [np.pi/2.0, np.pi/2.0]], [[0.0, 0.0],
                        [np.pi/2.0, np.pi/2.0]]]
base.compute('y = r*cos(theta)')
```

(continues on next page)

(continued from previous page)

```
base.compute('z = r*sin(theta)')
# having conservative variables called rho, rhou, rhov, rhow, rhoE at nodes
base[0][0]['rho'] = [[[1.0, 1.0], [1.0, 1.0]], [[1.2, 1.2], [1.2, 1.2]]]
base[0][0]['rhou'] = [[[1.0, 1.0], [1.0, 1.0]], [[1.0, 1.0], [1.0, 1.0]]]
base[0][0]['rhov'] = [[[0.0, 0.0], [0.0, 0.0]], [[0.0, 0.0], [0.0, 0.0]]]
base[0][0]['rhow'] = [[[0.0, 0.0], [0.0, 0.0]], [[0.0, 0.0], [0.0, 0.0]]]
base[0][0]['rhoE'] = [[[1.0e5, 1.0e5], [1.0e5, 1.0e5]], [[1.0e5, 1.0e5],
                       [1.0e5, 1.0e5]]]
# having gas properties and row properties as attributes (mono-row case)
# the row rotates at 60 rpm
base.attrs['Rgas'] = 250.0
base.attrs['gamma'] = 1.4
base.attrs['omega'] = 2.0*np.pi
base.attrs['pitch'] = np.pi/2.0

# construction and configuration of the cut treatment used by the algo
t_cut = antares.Treatment('cut') # VTK-triangles

# construction and configuration of the thermo average treatment
t_thermo = antares.Treatment('thermo7')
t_thermo['cylindrical_coordinates'] = ['x', 'r', 'theta']
t_thermo['conservative'] = ['rho', 'rhou', 'rhov', 'rhow', 'rhoE']

# computation of the averaged meridional plane
t = antares.Treatment('averagedmeridionalplane')
t['base'] = base
t['cartesian_coordinates'] = ['x', 'y', 'z']
t['merid_coordinates'] = ['x', 'r']
t['machine_coordinates'] = ['x', 'r', 'theta']
t['out_variables'] = ['alpha', 'beta', 'phi', 'Ma', 'Mr', 'Psta', 'Pta',
                      'Ptr', 'Tsta', 'Tta', 'Ttr']
t['cut_treatment'] = t_cut
t['thermo_treatment'] = t_thermo
t['abscissa_nb'] = 2
t['height_nb'] = 3
base_merid = t.execute()

# export the averaged meridional plane
w = antares.Writer('hdf_antares')
w['filename'] = 'my_averaged_meridional_plane'
w['base'] = base_merid
w.dump()
```

Main functions

class antares.treatment.turbomachine.TreatmentAveragedMeridionalPlane.
TreatmentAveragedMeridionalPlane

execute()

Meridional Plane

Description

Compute the meridional plane by performing several azimuthal means over 'x'-planes. The mass-flow is used as averaging variable ($\rho v_x \partial\theta$).

Construction

```
import antares
myt = antares.Treatment('meridionalplane')
```

Parameters

- **base: Base**
  The input base on which the meridional plane is built.
- **x_min: float, default= *None***
  The "x"-value of the axial start position of the plane.
- **x_max: float, default= *None***
  The "x"-value of the axial end position of the plane.
- **num_x: int, default= *20***
  The number of points in the axial distribution.
- **num_r: int, default= *30***
  The number of points in the radial distribution.
- **m_var: str, default= *'rovx'***
  The variable to be averaged on the plane.

Preconditions

The input **base** must contain the mesh coordinates x, y, and z, the solution and 'hb_computation' as an `Base.attrs` (if HB/TSM type).

Postconditions

The meridional output plane contains x, r coordinates and values.

Main functions

class antares.treatment.turbomachine.TreatmentMeridionalPlane.**TreatmentMeridionalPlane**

execute()

Compute the meridional plane.

Returns

Return type
Base

Example

```
import os

from antares import Reader, Treatment, Writer

if not os.path.isdir('OUTPUT'):
    os.makedirs('OUTPUT')

# Data can be downloaded from
# https://cerfacs.fr/antares/downloads/application1_tutorial_data.tgz

r = Reader('bin_tp')
r['filename'] = os.path.join('..', 'data', 'ROTOR37', 'ELSA_CASE', 'MESH',
                             'mesh_<zone>.dat')
r['zone_prefix'] = 'Block'
r['topology_file'] = os.path.join('..', 'data', 'ROTOR37', 'ELSA_CASE',
                                  'script_topo.py')
r['shared'] = True
base = r.read()
print(base.families)

r = Reader('bin_tp')
r['base'] = base
r['filename'] = os.path.join('..', 'data', 'ROTOR37', 'ELSA_CASE', 'FLOW',
                             'flow_<zone>.dat')
r['zone_prefix'] = 'Block'
r['location'] = 'cell'
r.read()

base.set_computer_model('internal')

# Needed for turbomachinery dedicated treatments
base.cell_to_node()
base = base.get_location('node')
print(base.families)
```

(continues on next page)

(continued from previous page)

```
base.compute('psta')
base.compute('Pi')
base.compute('theta')
P0_INF = 1.9
base.compute('MachIs = (((%f/psta)**((gamma-1)/gamma)-1.) * (2./(gamma-1.))  )**0.5' %␣
↪P0_INF)

# Meridional plane
res_dir = os.path.join('OUTPUT', 'MERID')
if not os.path.isdir(res_dir):
    os.makedirs(res_dir)

NUM_R = 50
NUM_X = 25
t = Treatment('meridionalplane')
t['base'] = base
t['num_r'] = NUM_R
t['num_x'] = NUM_X
t['x_min'] = -12.5
t['x_max'] = 12.5
t['m_var'] = 'psta'
merid_base = t.execute()

writer = Writer('bin_tp')
writer['filename'] = os.path.join(res_dir, 'flow_merid_%i_%i.plt' % (NUM_X, NUM_R))
writer['base'] = merid_base
writer.dump()
```

Compute the isentropic Mach number on a blade

Description

Compute the 3D isentropic Mach number field on a blade's skin.

Parameters

- **base: `Base`**
  The input base corresponds to the blade's skin.
- **TreatmentPtris: `TreatmentPtris`**
  An object defined upstream with at least **in_vars** and **out_vars** options that compute the meridional coordinate and the isentropic pressure variable.

  For more details, please refer to the corresponding TreatmentPtris<convention>.
- **h_H_name: `str`, default = *'CoordinateReducedHeight'***
  The coordinate reduced height's name. Basically, 'CoordinateReducedHeight' from the *`antares.treatment.turbomachine.TreatmenthH.TreatmenthH`* (page 240).
- **h_H_range: `list(float, float)`, default = *None***
  Allows to clip the **h_H_name**'s range of values. Might be useful to avoid technological effects. Ei-

ther let it unfilled then the geometry bounds the variable **h_H_name**. Or a **list of float** clip it : **[minimum_h_H_name, maximum _h_H_name]**.

- **number_of_heights: `int`, default = *101***
  Spatial discretisation of the height, based on the **h_H_name**. In other words, number of profiles the blade will be composed of.
- **d_D_range: `str`, default = *'0_1'***
  Defines the curvilinear reduced coordinates' range of value. Either ranged from -1 to 1 or from 0 to 1.
- **begin_unwrap: `str`, default = *'LE'***
  Convention : *'LE'* (leading edge) or *'TE'* (trailing edge). In other words, either sorted the blade from the *LE* to the *TE* or vice versa. Therefore, if **begin_unwrap** is set to *'LE'*, the curvilinear coordinate's zero is at the leading edge.
- **communicator: `antares.parallel.controller.ParallelController`, default = *PARA***
  Communicator to communicate with a subset of the original group of processes at once. The default communicator is the singleton PARA created at the import of antares.

Preconditions

The treatment must be applied on a 3D `Zone` corresponding to a **blade skin and its fillet**. The `Base` must have only one `Zone` and one `Instant`. Moreover, the treatment requires a reduced height coordinate (**h_H_name**) in its `Instant`, for instance, computed with *`antares.treatment.turbomachine.TreatmenthH.TreatmenthH`* (page 240). Last but not least, a TreatmentPtris<convention> has to be defined upstream with options 'in_vars' and 'out_vars'.

Postconditions

The input **base** is returned, extended with the variables:

- **Mis**
  The isentropic Mach number with a reference pressure based on the given **TreatmentPtris**.
- **d**
  The curvilinear coordinate around each profil discretizing the 3D blade.
- **d_D**
  More precisely, out_vars[0] variable's name defined in the TreatmentPtris<convention>: **d_D** by default but may be changed by the user. This variable corresponds to the reduced form of the previous one.

Example

```
import antares

# Fill Ptris information. For more details,
# refer to the corresponding treatment.
myt_Ptris = antares.Treatment('Ptris'+'convention_name')
myt_Ptris['in_vars'] = ['x', 'y', 'z', 'r', 'CoordinateReducedMeridional',
                        'gamma', 'Cp', 'omega', 'Psta', 'Tsta']
myt_Ptris['out_vars'] = ['d_D','Ptris']

# Parameters the Mis calculation on the blade
myt = antares.Treatment('MisOnBlade')
```

(continues on next page)

(continued from previous page)

```
myt['base'] = blade_skin
myt['number_of_heights'] = 101
myt['d_D_range'] = '0_1'
myt['TreatmentPtris'] = myt_Ptris
blade_skin_result = myt.execute()
```

Main functions

class antares.treatment.turbomachine.TreatmentMisOnBlade.**TreatmentMisOnBlade**

execute()

Compute the isentropic Mach number on a 3D blade.

Returns
The 3D blade with the isentropic Mach number computed

Return type
Base

Example

```
"""
This script may ne run in parallel with
mpirun -np 16 python misonblade.py

There is no data available for this example.
"""
import os
import numpy as np
import antares
from antares.parallel.controller import PARA

output = 'OUTPUT'
if not os.path.isdir(output):
    os.makedirs(output)

reader = antares.Reader('hdf_cgns')
reader['filename'] = 'data_with_many_zones.cgns'
reader['rename_vars'] = False
base = reader.read()

hub_pts = np.genfromtxt('hub_meridional_line.dat', unpack=False)
shroud_pts = np.genfromtxt('shroud_meridional_line.dat', unpack=False)

t = antares.Treatment('hH')
t['base'] = base
t['families'] = ['ROW1']  # family (volume zone) where to compute hH
t['hub_pts'] = hub_pts
t['shroud_pts'] = shroud_pts
t['extension'] = 1.5
```

(continues on next page)

(continued from previous page)

```
t['number_of_heights'] = 21
t['dmax'] = 1e-05
t['precision'] = 1e-05
t['coordinates'] = ['CoordinateX', 'CoordinateY', 'CoordinateZ']
base, paramgrid = t.execute()

# get the base corresponding to a blade given by a family name
# in parallel, some processors may not hold a part of the blade
try:
    skin_blade = base[base.families['BLADE1']]
except KeyError:
    skin_blade = antares.Base()

myt = antares.Treatment('merge')
myt['base'] = skin_blade
myt['duplicates_detection'] = True
myt['tolerance_decimals'] = 13
skin_blade = myt.execute()

# rename the zone
if skin_blade:
    skin_blade['zone%d'%PARA.rank] = skin_blade[0]
    del skin_blade[0]

skin_blade.set_computer_model('constant_gamma')
skin_blade.compute('gamma')
skin_blade.compute('CP')
skin_blade.compute('R')
skin_blade.compute('Pressure')
skin_blade.compute('Temperature')
wrot = 0#17188.7 * 2*np.pi/60.
skin_blade.compute('omega = {}'.format(wrot))

# compute the reference isentropic Ptr
myt_Ptris = antares.Treatment('ptris<convention>')
myt_Ptris['in_vars'] = ['CoordinateX', 'CoordinateY', 'CoordinateZ', 'R',
↪'CoordinateReducedMeridional',
                        'gamma', 'CP', 'omega', 'Pressure', 'Temperature']
myt_Ptris['out_vars'] = ['d_D','Ptris']

# Parameters the Mis calculation on the blade
myt = antares.Treatment('MisOnBlade')
myt['base'] = skin_blade
myt['number_of_heights'] = 101
myt['d_D_range'] = '0_1'
myt['TreatmentPtris'] = myt_Ptris
blade_skin_result = myt.execute()

writer = antares.Writer('hdf_cgns')
writer['filename'] = os.path.join(output, 'blade_skin_para_{}p.cgns'.format(PARA.size))
writer['coordinates'] = ['CoordinateX', 'CoordinateY', 'CoordinateZ']
writer['base'] = blade_skin_result
```

(continues on next page)

(continued from previous page)

```
writer.dump()
```

Radial profile of thermodynamics treatments

Description

This treatment computes the radial distribution of thermodynamics treatments. For each radial section, quantities are averaged in space. The input base is not modified but dictionnaries are added to its attributes.

It must be applied on 2D surfaces resulting from a revolution cut of an axisymmetric mono-row or multi-row configuration.

Construction

```
import antares
myt = antares.Treatment('thermo1D')
```

Parameters

- **base:** `Base`
  The base on which the treatment will be applied.
- **nb_radius:** `int`**, default =** ***100***
  Number of radius necessary to discretize radially the geometry.
- **radial_clip:** `list(float, float)`**, default =** ***None***
  List of radius to clip the 2D geometry. First element is the minimum and the second one is the maximum radius. Per default no clip is performed.
- **cylindrical_coordinates:** `list(str, str, str)`**, default =** ***['x', 'r', 'theta']***
  Names of the cylindrical coordinates: the axis of rotation (axial coordinate), the distance to this axis (radius), and the azimuth.
- **gamma_hypothesis:** `str`**, default =** ***'cst'***
  Either *'cst'* so a value of **gamma** is expected in the input **base** attributes, or *'0D'* then a 0D value of **gamma_var** is computed for the whole input **base**.
- **gamma_var:** `str`**, default =** ***'gamma_bar'***
  Name of specific heat ratio. Only useful if **gamma_hypothesis** is equal to *'0D'*.
- **distribution_type:** `str`**, default =** ***'cst'***
  Distribution type of the radial cuts (*'cst'* or *'erf'*). Either the distribution is constant, which means one goes through a radial cut to the next one with a $\Delta r = \frac{r_{max}-r_{min}}{nb\_radius}$ where r_{max} and r_{min} are the maximun and minimum radius of the **base**. Warning, that is the case if **radial_clip** is *None* otherwise, r_{max} and r_{min} are defined by **radial_clip**'s values.
- **dp0:** `float`**, default =** ***1.0e-02***
  If **distribution_type** is *'erf'*, specify the **dp0**'s option of the function `antares.utils.math_utils.norm_cdf_distribution()`.

- **ThermoGeom:** *antares.treatment.turbomachine.TreatmentThermoGeom* **(page 224), default =** ***None***
  Call a TreatmentThermoGeom defined upstream. Only options **cartesian_coordinates**, **cylindrical_coordinates** and **def_points** might be filled. For more details, please refer to the corresponding treatment.
- **ThermoLES:** *antares.treatment.turbomachine.TreatmentThermoLES* **(page 265), default =** ***None***
  Call a TreatmentThermoLES defined upstream. Only options **conservative** and **cell_surface_var** might be filled. For more details, please refer to the corresponding treatment.

Preconditions

The treatment must be applied on a mono-zone **base** containing a 2D section resulting from a cut with a revolution surface around the 'x'-axis of an axisymmetric configuration. This `Zone` must contain only one `Instant` (steady-state).

Preconditions are case dependant whether *antares.treatment.turbomachine.TreatmentThermoGeom* (page 224), or *antares.treatment.turbomachine.TreatmentThermoLES* (page 265) is called. Please refer directly to the corresponding treatment. Most of them need the gas properties either as attributes or at cells, named respectively 'Rgas' or 'Rgaz' or 'R_gas' and 'gamma'.

Postconditions

The input **base** is returned, extended with the attribute named '1D/Moyenne1D'.

This attribute is a dictionary with variables:

- **radius**
  The array corresponding to the radial discretization of the input **base**.
- **hH**
  The array corresponding to the coordinate reduced height discretization based on the radial range of value.

In addition, the input **base** might be returned extended with attributes:

- '1D/Geometry' if *antares.treatment.turbomachine.TreatmentThermoGeom* (page 224)
- '1D/MoyenneLES#Steady' if *antares.treatment.turbomachine.TreatmentThermoLES* (page 265)

has been set up upstream and called with TreatmentThermo1D.

Those attributes are dictionaries with variables, more details can be found in their relative documentation. They are no more single values but arrays of values for each radial section.

Example

```
import antares

# Fill TreatmentThermoGeom options
# Do not fill ['base'] option nor perform .execute()
ThermoGeom = Treatment('ThermoGeom')
ThermoGeom['def_points'] = cut_points

# Fill TreatmentThermoLES options
# Do not fill ['base'] option nor perform .execute()
ThermoLES = Treatment('ThermoLES')
ThermoLES['conservative'] = ['rho', 'rhou', 'rhov', 'rhow', 'rhoEtotal']
```

(continues on next page)

(continued from previous page)

```
# Fill TreatmentThermo1D options then call the previous ones
t = Treatment('Thermo1D')
t['base'] = base
t['nb_radius'] = 100
t['gamma_hypothesis'] = '0D'
t['ThermoGeom'] = ThermoGeom      # call the TreatmentThermoGeom set up upstream
t['ThermoLES'] = ThermoLES            # call the TreatmentThermo1 set up upstream
result_base = t.execute()
```

Main functions

class antares.treatment.turbomachine.TreatmentThermo1D.**TreatmentThermo1D**

execute()

Compute the 1D thermodynamic spatial average.

Thermodynamic LES average weighted by mass flux

Description

This treatment computes the spatial and temporal mean of three LES quantities. The formula of the average is weighted by mass flux.

This thermodynamic average is available for the mean flow and instantaneous flow fields of unsteady flows.

Parameters

- **base: `Base`**
  The base on which the treatment will be applied.
- **conservative: `list(str)`, default=** ***['rhou', 'rhouPtotal', 'rhouTtotal', 'alpha']***
  Names of turbomachinery conservative variables. The order has to be respected:
  - first, the momentum in the x direction,
  - then the total pressure multiplied by the x-momentum,
  - then the total temperature multiplied by the x-momentum,
  - and finally the gyration angle (optional).
- **cell_surface_var: `str`, default=** ***'cell_volume'***
  Name of the surface variable of the mesh elements in the `Instant`. For instance, if the *antares.treatment.turbomachine.TreatmentThermoGeom* (page 224) has been called beforehand, then the variable is named 'surface'.

Preconditions

The treatment must be applied on a mono-zone **base** containing a 2D section resulting from a cut with a revolution surface around the 'x'-axis of an axisymmetric configuration. This `Zone` may contain one `Instant` (steady-state) or several of them (instantaneous solutions at several instants).

For **steady flows**, the input quantities should directly come from the solver's output. In fact, due to ergodicity, make sure that quantities involved are well defined in the flow solution. For instance, for the AVBP solver, the **run.params** file must contain at least the following options in the **OUTPUT-CONTROL** section:

::

save_average = yes save_average.mode = packages save_average.packages = conservative save_average.packages = turbomachinery

For **unsteady flows**, the input **base** must contain several `Instant`. Every instant will be taken into account to perform the time averaging.

The **conservative** variables must be available at nodes or cells.

Postconditions

The input **base** is returned, extended with the attribute named '0D/MoyenneLES#Steady'.

This attribute is a dictionary with the following key:

- **Ptotal**
  associated to the quantity $\frac{<\overline{\rho u P_{total}}>}{<\overline{\rho u}>}$
- **Ttotal**
  associated to the quantity $\frac{<\overline{\rho u T_{total}}>}{<\overline{\rho u}>}$
- **alpha**
  associated to the quantity $<\overline{\alpha}>$

where

$<\phi(\vec{x},t)>=\frac{1}{S}\iint_S \phi(\vec{x},t).\vec{n}\,dS$ with $\vec{n}$ the normal vector to the surface **S** and,

$\overline{\phi(\vec{x},t)}=\frac{1}{t_f-t_0}\int_{t_0}^{t_f}\phi(\vec{x},t)\,dt$ with t_0 and t_f respectively the starting and ending times of the temporal averaging.

Examples

We consider a LES solution averaged in time from the AVBP solver. Therefore, the **conservative** variables are already in the **base** as solver's outputs with the names rhou, rhouPt and rhouTt. If the turbomachinery package is used, the **alpha** variable is also already computed under the name *angle_alpha*.

```
import antares

# if the surface variable is missing at cell centers
base.compute_cell_volume(coordinates=['x', 'y', 'z'])

myt = antares.Treatment('ThermoLES')
myt['base'] = base
myt['conservative'] = ['rhou', 'rhouPt', 'rhouTt', 'angle_alpha']
myt['cell_surface_var'] = 'cell_volume'
base = myt.execute()
```

We now consider several instantaneous LES solutions. Therefore, **conservative** variables may not be in the solver's outputs. However, they must be in the **base**, hence here they are computed through `antares.api.Base.Base.compute()`. However, **alpha** must be in the base if turbomachinery package is on.

```
import antares

# compute the missing variable
base.compute('rhou = rho*u')
base.compute('rhouPt = rhou*Ptotal')
base.compute('rhouTt = rhou*Ttotal')

myt = antares.Treatment('ThermoLES')
myt['base'] = base
myt['conservative'] = ['rhou', 'rhouPt', 'rhouTt', 'angle_alpha']
base = myt.execute()
```

Main functions

class antares.treatment.turbomachine.TreatmentThermoLES.**TreatmentThermoLES**

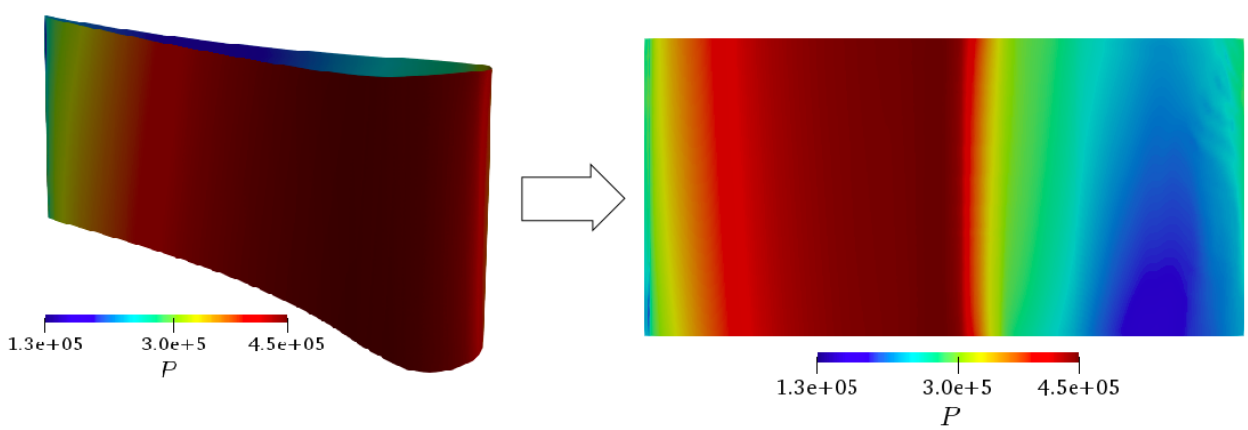
execute()

Compute the thermodynamic average of LES type on a surface.

Unwrap a blade's skin

Description

Unwrap a 3D blade into a 2D plan.



Warning:

dependency on:

- scipy.spatial.Delaunay[119]

[119] https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.spatial.Delaunay.html

Parameters

- **base: `Base`**
  The input base must correspond to the blade's skin. In addition, the variable *CoordinateReducedHeight* computed from the `antares.treatment.turbomachine.TreatmenthH` (page 238) must be included in the input **base**.
- **variables: `list(str)`, default = *None***
  Names of variables present in the **input base** that will also be in the **output base**. If *None*, all input variables will be conserved. Variables like *'d'*, *'d_D'* and *'R'* respectively associated to curvilinear coordinate and its reduced form, and the radius.
- **number_of_heights: `int`, default = *101***
  The number of iso-h/H needed to discretize the blade through the variable *CoordinateReducedHeight*.
- **number_of_d_D: `int`, default = *1001***
  The number of discretization along the blade's curvilinear coordinate. **This number has to be strictly greater than one.**
- **begin_unwrap: `str`, default = *'LE'***
  Convention : *'LE'* (leading edge) or *'TE'* (trailing edge). In other words, either sort the blade from the LE to the TE or vice versa. Therefore, if *begin_unwrap* is set to *'LE'*, the curvilinear coordinate's zero is at the leading edge.
- **p_var: `str`, default = *'Psta'***
  Name of the pressure variable. Necessary to discriminate pressure from suction side on every profile.
- **h_H_name: `str`, default = *'CoordinateReducedHeight'***
  Name of the coordinate reduced height variable.
- **h_H_range: `str`, default = *None***
  Either *None* and h/H borders are limited by the CAD, or borders are manually fixed: *hH_range = [min_hH, max_hH]*.
- **cartesian_coordinates: `list(str, str, str)`, default = *['x', 'y', 'z']***
  Names of the 3 cartesian coordinates.
- **TreatmentPtris: TreatmentPtris<convention>, default = *None***
  Call an object defined upstream with at least **in_vars** options.

  For more details, please refer to the corresponding treatment.

Preconditions

The treatment must be applied on a 3D `Zone` corresponding to a **blade skin and its fillet**. The `Base` must have only one `Zone` and one `Instant` with reduced height and meridional coordinates (computed with *antares.treatment.turbomachine.TreatmenthH* (page 238)).

Postconditions

The output is a mono-zone base containing the **variables** from the input **base**, plus the curvilinear curvature coordinate and its reduced form, and the radius. If **TreatmentPtris** is not *None*, the input **base** is also extended with the isentropic Mach number *Mis* and its reference pressure *Ptris*.

The input **base** is a mono-zone base containing the **variables** defined by the user. It is also extended with the variables:

- **d**
  The curvilinear curvature coordinate.
- **d_D**
  The reduced form of the previous one.
- **Ptris**
  The total isentropic pressure of reference, necessary to compute **Mis**. Only if a TreatmentPtris<convention> has been called.
- **Mis**
  The isentropic Mach number. Only if a TreatmentPtris<convention> has been called.

Examples

```
import antares

myt = antares.Treatment('UnwrapBlade')
myt['base'] = blade_skin_base
myt['number_of_heights'] = 101
myt['number_of_d_D'] = 1001
base = myt.execute()
```

If you like to plot a 2D contouring of the unwraped blade on a (d/D, R) plan, the following lines should be added to the previous ones.

```
from matplotlib.tri import Triangulation

tri = Triangulation(base[0][0]['x'],
                    base[0][0]['y'])
tri.x = base[0][0]['d_D']
tri.y = base[0][0]['R']
```

If you like the get Mis on the 2D unwraped blade, the following exemple shows an efficient way to do so.

```
import antares

# Fill Ptris information. For more details,
# refer to the corresponding treatment.
```

(continues on next page)

(continued from previous page)

```
myt_Ptris = antares.Treatment('Ptris'+'convention_name')
myt_Ptris['in_vars'] = ['x', 'y', 'z', 'r', 'CoordinateReducedMeridional',
                        'gamma', 'Cp', 'omega', 'P', 'T']

myt = antares.Treatment('UnwrapBlade')
myt['base'] = blade_skin_base
myt['TreatmentPtris'] = myt_Ptris
base = myt.execute()
```

Main functions

class antares.treatment.turbomachine.TreatmentUnwrapBlade.**TreatmentUnwrapBlade**

execute()

Unwrap a 3D blade into a 2D plan.

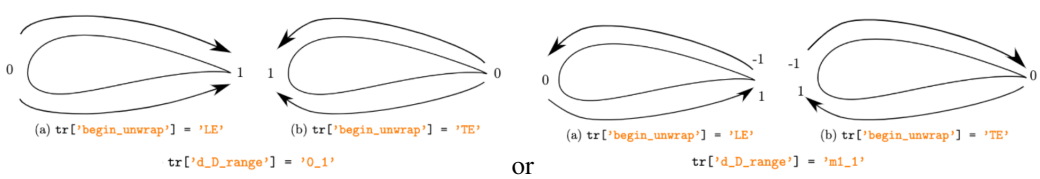
Unwrap a blade's profile

Description

This treatment unwraps a line's profile.

It must be applied on a 1D line resulting from a cut of a 3D blade.

As a result, the indices of nodes from pressure and suction sides will be available as attributes of the base.



Examples of **begin_unwrap** and **d_D_range** options.

Parameters

- **base: Base**
  The base on which the treatment will be applied.
- **p_var: str, default = *'Psta'***
  Name of pressure variable.
- **d_D_range: str in ['m1_1', '0_1'], default = *'0_1'***
  Range of the curvilinear reduced coordinates. Either 'm1_1' for a range from -1 to 1 or *'0_1'* for a range from 0 to 1.
- **begin_unwrap: str in ['LE', 'TE'], default = *'LE'***
  Convention: *'LE'* (leading edge) or *'TE'* (trailing edge). Either sort the blade from the LE to the TE or vice versa. Therefore, if **begin_unwrap** is set to *'LE'*, the origin of the curvilinear coordinate is at the leading edge.

- **tolerance_decimals: `int`, default = *None***
  Number of decimal places to round the coordinates. When the base is multi-zone, it is used to merge the intersection points of the zones. If *None*, the coordinates will not be rounded.
- **memory_mode: `bool`, default = *False***
  If *True*, the input **base** is deleted on the fly to limit memory usage.
- **cartesian_coordinates: `list(str, str, str)`, default = *['x', 'y', 'z']***
  Names of the 3 Cartesian coordinates.
- **crm_var: `str`, default = *'CoordinateReducedMeridional'***
  Name of the reduced meridional coordinate variable.
- **out_vars: `sequence(str, str)`, default = *['d_D', 'd']***
  Name of the output reduced curvilinear curvature coordinate and its non reduced form.

Preconditions

The treatment must preferably be applied on a 1D `Zone` resulting from a blade cut. However, a merge treatment can handle a base with multiple zones. The `Zone` must contain only one `Instant`.

Postconditions

The input base may be modified in-place depending on the result of the inner merge treatment.

The output base is a merge of the input **base** extended with the attribute named *'Profil'* in `Base.attrs`.

This attribute is a dictionary with variables:

- **IndicesPressureSide**
  Instant's nodes indices corresponding to the profile's pressure side.
- **IndicesSuctionSide**
  Instant's nodes indices corresponding to the profile's suction side.

The Instant contained in the input **base** is extended with the following variables:

- **out_vars[1]**
  The curvilinear curvature coordinate of the profile.
- **out_vars[0]**
  Reduced form of the curvilinear curvature coordinate. In other words, the curvilinear curvature coordinate of the profile divided by the total curvilinear curvature length.

Example

```
import antares

myt = antares.Treatment('UnwrapProfil')
myt['base'] = base
base = myt.execute()
```

The following statements plot the temperature as a function of the reduced curvilinear abscissa coordinate.

```
import matplotlib.pyplot as plt

# First, retrieve pressure and suction sides indices:
ind_PS = base.attrs['Profil']['IndicesPressureSide']
ind_SS = base.attrs['Profil']['IndicesSuctionSide']

# Get variables you want to plot
T = base[0][0]['T']
d_D = base[0][0]['d_D']

# Create a basic figure
plt.figure()
plt.plot(d_D[ind_PS], T[ind_PS], label='Pressure side')
plt.plot(d_D[ind_SS], T[ind_SS], label='Suction side')
plt.legend()
plt.show()
```

Main functions

class antares.treatment.turbomachine.TreatmentUnwrapProfil.**TreatmentUnwrapProfil**

execute()

Extract variables along a profile of a blade.

Returns

Return type
`Base`

Cp (Pressure Coefficient)

Computation of pressure coefficient

Parameters

- **base: `Base`**
  The base must contain:
  - the mesh coordinates x, y, and z
  - the solution
  - 'hb_computation' as an `Base.attrs` (if HB/TSM type).
- **coordinates:** ***list(str)***
  The variable names that define the set of coordinates.
- **vectors:** ***tuple/list(tuple(str))*****, default=** ***[]***
  Component names of vectors that need to be rotated. It is assumed that these are given in the cartesian coordinate system.
- **rho_inf:** ***float*****, default=** ***'in_attr'***
  The infinite density corresponding to the reference state.

- **v_inf:** ***float*, default= *'in_attr'***
  The infinite axial velocity corresponding to the reference state.
- **p_inf:** ***float*, default= *'in_attr'***
  The infinite static pressure corresponding to the reference state.
- **family_name:** ***str***
  The name of the family from which the percent will be computed and on which Cp is computed.
- **percent:** ***float*, default= *None***
  The percentage relative to the family to determine the absolute position value.
- **position:** ***float*, default= *None***
  The absolute position value relative to the family where the cut must be made.
- **form:** ***int in [1,2,3]*, default= *1***
  The definition of Cp (see below).

Main functions

class antares.treatment.turbomachine.TreatmentCp.**TreatmentCp**

execute()

Execute the treatment.

Compute the pressure coefficient at a given radius percent of the given blade. Thee formulae for Cp are proposed:

form 1: $Cp_1 = -\frac{p - p_{inf}}{\rho_{inf} n^2 D^2}$

form 2: $Cp_2 = 2\frac{p - p_{inf}}{\rho_{inf}(v_{inf} n^2 r^2)}$

form 3: $Cp_3 = 2.0\frac{p - p_{inf}}{\rho_{inf} v_{inf}^2}$.

The mean and the harmonics of Cp can also be computed if duplication is enabled. Note that the amplitude of the harmonics are divided by the mean value.

Returns

Return type
Base

Example

```
import os

if not os.path.isdir('OUTPUT'):
    os.makedirs('OUTPUT')

from antares import Reader, Treatment, Writer

#

# Data can be downloaded from
```

(continues on next page)

(continued from previous page)

```
# https://cerfacs.fr/antares/downloads/application1_tutorial_data.tgz

r = Reader('bin_tp')
r['filename'] = os.path.join('..', 'data', 'ROTOR37', 'ELSA_CASE', 'MESH',
                             'mesh_<zone>.dat')
r['zone_prefix'] = 'Block'
r['topology_file'] = os.path.join('..', 'data', 'ROTOR37', 'ELSA_CASE',
                                  'script_topo.py')
r['shared'] = True
base = r.read()
print(base.families)

r = Reader('bin_tp')
r['base'] = base
r['filename'] = os.path.join('..', 'data', 'ROTOR37', 'ELSA_CASE', 'FLOW',
                             'flow_<zone>.dat')
r['zone_prefix'] = 'Block'
r['location'] = 'cell'
r.read()

base.set_computer_model('internal')

# Needed for turbomachinery dedicated treatments
base.cell_to_node()
base = base.get_location('node')
print(base.families)

base.compute('psta')
base.compute('Pi')
base.compute('theta')
P0_INF = 1.9
base.compute('MachIs = (((%f/psta)**((gamma-1)/gamma)-1.) * (2./(gamma-1.))  )**0.5' %␣
↪P0_INF)

# Definition of the treatment
t = Treatment('Cp')
t['base'] = base
t['family_name'] = 'BLADE'
t['coordinates'] = ['x', 'y', 'z']
t['rho_inf'] = 0.873
t['p_inf'] = 0.59
t['v_inf'] = 1.5
t['form'] = 3

# Cp
res_dir = os.path.join('OUTPUT', 'CP')
if not os.path.isdir(res_dir):
    os.makedirs(res_dir)

writer = Writer('column')

for loc in [0.25, 0.5, 0.75, 0.9]:  # radius in percent
```

(continues on next page)

(continued from previous page)

```
    t['percent'] = loc
    Cp_blade = t.execute()

    writer['filename'] = os.path.join(res_dir, 'Cp_%s.dat' % (loc))
    writer['base'] = Cp_blade
    writer.dump()
```

Slice at an axial position (x-coordinate)

Cut at an axial position

Parameters

- **base: `Base`**
  The base must contain:
  - the mesh coordinates x, y, and z
  - the solution
  - 'hb_computation' as an `Base.attrs` (if HB/TSM type).
- **vectors: *tuple/list(tuple(str))*, default= *[]***
  Component names of vectors that need to be rotated. It is assumed that these are given in the cartesian coordinate system.
- **nb_duplication: *int*, default= *'in_attr'***
  number of duplications to apply after doing the axial cut if duplicate is True. If set to 'in_attr', then it is computed from 'nb_blade' in `Instant.attrs`.
- **duplicate: *bool*, default= *False***
  Duplication of the axial cut. Chorochronic if HB/TSM type.
- **family_name: *str***
  The name of the family from which the percent will be computed and on which the cut is computed.
- **percent: *float*, default= *None***
  The percentage relative to the family to determine the absolute position value of the cut.
- **position: *float*, default= *None***
  The absolute position value relative to the family where the cut must be made.

Main functions

class antares.treatment.turbomachine.TreatmentSliceX.**TreatmentSliceX**

execute()

Execute the treatment.

This method performs a cut at an axial position. Either the value of the axial position is given, or it is computed knowing the family name and the percentage. The latter are used to determine the absolute position of the cut.

Then, you can make a simple duplication or a chorochronic duplication of this cut.

Warning: The axial coordinate should be named 'x'.

Returns

Return type
None or Base

Example

```
import os

if not os.path.isdir('OUTPUT'):
    os.makedirs('OUTPUT')

import numpy as np

from antares import Reader, Treatment, Writer

#

# Data can be downloaded from
# https://cerfacs.fr/antares/downloads/application1_tutorial_data.tgz

r = Reader('bin_tp')
r['filename'] = os.path.join('..', 'data', 'ROTOR37', 'ELSA_CASE', 'MESH',
                             'mesh_<zone>.dat')
r['zone_prefix'] = 'Block'
r['topology_file'] = os.path.join('..', 'data', 'ROTOR37', 'ELSA_CASE',
                                  'script_topo.py')
r['shared'] = True
base = r.read()
print(base.families)

r = Reader('bin_tp')
r['base'] = base
r['filename'] = os.path.join('..', 'data', 'ROTOR37', 'ELSA_CASE', 'FLOW',
                             'flow_<zone>.dat')
r['zone_prefix'] = 'Block'
r['location'] = 'cell'
r.read()

base.set_computer_model('internal')

# Needed for turbomachinery dedicated treatments
base.cell_to_node()
base = base.get_location('node')
print(base.families)

base.compute('psta')
base.compute('Pi')
base.compute('theta')
```

(continues on next page)

(continued from previous page)

```
P0_INF = 1.9
base.compute('MachIs = (((%f/psta)**((gamma-1)/gamma)-1.) * (2./(gamma-1.))  )**0.5' %␣
↪P0_INF)

res_dir = os.path.join('OUTPUT', 'SLICEX')
if not os.path.isdir(res_dir):
    os.makedirs(res_dir)

t = Treatment('slicex')
t['base'] = base
t['family_name'] = 'BLADE'

writer = Writer('bin_tp')

NUM = 9
x = np.linspace(-12.5, 12.5, NUM)
for i in range(0, NUM):
    print('cut at x = {}'.format(x[i]))

    t['position'] = x[i]
    base = t.execute()

    writer['filename'] = os.path.join(res_dir, 'slicex_%i.plt' % x[i])
    writer['base'] = base
    writer.dump()
```

Slice at a radial position

Cut at a radial position

Parameters

- **base: `Base`**
  The base must contain:
  - the mesh coordinates x, y, and z
  - the solution
  - 'hb_computation' as an `Base.attrs` (if HB/TSM type).
- **vectors: *tuple/list(tuple(str))*, default= *[]***
  Component names of vectors that need to be rotated. It is assumed that these are given in the cartesian coordinate system.
- **nb_duplication: *int*, default= *'in_attr'***
  number of duplications to apply after doing the axial cut if duplicate is True. If set to 'in_attr', then it is computed from 'nb_blade' in `Instant.attrs`.
- **duplicate: *bool*, default= *False***
  Duplication of the axial cut. Chorochronic if HB/TSM type.
- **family_name: *str***
  The name of the family from which the percent will be computed and on which the cut is computed.

- **percent:** ***float*****, default=** ***None***
  The percentage relative to the family to determine the absolute position value of the cut.
- **position:** ***float*****, default=** ***None***
  The absolute position value relative to the family where the cut must be made.

Main functions

class antares.treatment.turbomachine.TreatmentSliceR.**TreatmentSliceR**

execute()

Execute the treatment.

This method performs a cut at a radial position. Either the radius value is given, or it is computed knowing the family name and the percentage. The latter are used to determine the absolute position of the cut.

Returns

Return type
None or Base

Example

```
import os

if not os.path.isdir('OUTPUT'):
    os.makedirs('OUTPUT')

import numpy as np

from antares import Reader, Treatment, Writer

#

# Data can be downloaded from
# https://cerfacs.fr/antares/downloads/application1_tutorial_data.tgz

r = Reader('bin_tp')
r['filename'] = os.path.join('..', 'data', 'ROTOR37', 'ELSA_CASE', 'MESH',
                             'mesh_<zone>.dat')
r['zone_prefix'] = 'Block'
r['topology_file'] = os.path.join('..', 'data', 'ROTOR37', 'ELSA_CASE',
                                  'script_topo.py')
r['shared'] = True
base = r.read()
print(base.families)

r = Reader('bin_tp')
r['base'] = base
r['filename'] = os.path.join('..', 'data', 'ROTOR37', 'ELSA_CASE', 'FLOW',
                             'flow_<zone>.dat')
r['zone_prefix'] = 'Block'
r['location'] = 'cell'
```

(continues on next page)

(continued from previous page)

```
r.read()

base.set_computer_model('internal')

# Needed for turbomachinery dedicated treatments
base.cell_to_node()
base = base.get_location('node')
print(base.families)

base.compute('psta')
base.compute('Pi')
base.compute('theta')
base.compute('R')
P0_INF = 1.9
base.compute('MachIs = (((%f/psta)**((gamma-1)/gamma)-1.) * (2./(gamma-1.))  )**0.5' %␣
↪P0_INF)

res_dir = os.path.join('OUTPUT', 'SLICER')
if not os.path.isdir(res_dir):
    os.makedirs(res_dir)

t = Treatment('slicer')
t['base'] = base
t['family_name'] = 'BLADE'

writer = Writer('bin_tp')

NUM = 9
x = np.linspace(18., 25.5, NUM)
for i in range(0, NUM):
    print('cut at r = {}'.format(x[i]))

    t['position'] = x[i]
    base = t.execute()

    writer['filename'] = os.path.join(res_dir, 'slicer_%i.plt' % x[i])
    writer['base'] = base
    writer.dump()
```

Slice at an azimuthal position

Cut at an azimuthal position

Parameters

- **base: `Base`**
  The base must contain:
  - the mesh coordinates x, y, and z
  - the solution
  - 'hb_computation' as an `Base.attrs` (if HB/TSM type).
- **vectors: *tuple/list(tuple(str))*, default= *[]***
  Component names of vectors that need to be rotated. It is assumed that these are given in the cartesian coordinate system.
- **nb_duplication: *int*, default= *'in_attr'***
  number of duplications to apply after doing the axial cut if duplicate is True. If set to 'in_attr', then it is computed from 'nb_blade' in `Instant.attrs`.
- **duplicate: *bool*, default= *False***
  Duplication of the axial cut. Chorochronic if HB/TSM type.
- **family_name: *str***
  The name of the family from which the percent will be computed and on which the cut is computed.
- **percent: *float*, default= *None***
  The percentage relative to the family to determine the absolute position value of the cut.
- **position: *float*, default= *None***
  The absolute position value relative to the family where the cut must be made.

Main functions

class `antares.treatment.turbomachine.TreatmentSliceTheta.`**`TreatmentSliceTheta`**

`execute``()`

Execute the treatment.

This method performs a cut at an azimuthal position. Either the value of the azimuthal position is given, or it is computed knowing the family name and the percentage. The latter are used to determine the absolute position of the cut.

Returns

Return type
`Base`

Example

```
import os

if not os.path.isdir('OUTPUT'):
    os.makedirs('OUTPUT')

import numpy as np

from antares import Reader, Treatment, Writer
```

(continues on next page)

(continued from previous page)

```
#

# Data can be downloaded from
# https://cerfacs.fr/antares/downloads/application1_tutorial_data.tgz

r = Reader('bin_tp')
r['filename'] = os.path.join('..', 'data', 'ROTOR37', 'ELSA_CASE', 'MESH',
                             'mesh_<zone>.dat')
r['zone_prefix'] = 'Block'
r['topology_file'] = os.path.join('..', 'data', 'ROTOR37', 'ELSA_CASE',
                                  'script_topo.py')
r['shared'] = True
base = r.read()
print(base.families)

r = Reader('bin_tp')
r['base'] = base
r['filename'] = os.path.join('..', 'data', 'ROTOR37', 'ELSA_CASE', 'FLOW',
                             'flow_<zone>.dat')
r['zone_prefix'] = 'Block'
r['location'] = 'cell'
r.read()

base.set_computer_model('internal')

# Needed for turbomachinery dedicated treatments
base.cell_to_node()
base = base.get_location('node')
print(base.families)

base.compute('psta')
base.compute('Pi')
base.compute('theta')
P0_INF = 1.9
base.compute('MachIs = (((%f/psta)**((gamma-1)/gamma)-1.) * (2./(gamma-1.))  )**0.5' %␣
↪P0_INF)

base.attrs['nb_blade'] = 1

res_dir = os.path.join('OUTPUT', 'SLICET')
if not os.path.isdir(res_dir):
    os.makedirs(res_dir)

t = Treatment('slicetheta')
t['base'] = base
t['family_name'] = 'BLADE'

writer = Writer('bin_tp')

NUM = 9
```

(continues on next page)

(continued from previous page)

```
x = np.linspace(-0.162, -0.27, NUM)
for i in range(0, NUM):
    print('cut at theta = {}'.format(x[i]))

    # t['position'] = x[i]
    t['percent'] = 0.2
    result = t.execute()
    print(result)

    if result:
        writer['filename'] = os.path.join(res_dir, 'slicet_%i.plt' % x[i])
        writer['base'] = result
        writer.dump()
```

Extraction of variables along a profile of a blade at a given radius

Extraction of variables along a profile of a blade at a given radius

Parameters

- **base: `Base`**
  A base containing:
  - the mesh coordinates x, y, and z
  - the solution
  - 'hb_computation' as an `Base.attrs` (if HB/TSM type).
- **coordinates:** ***list(str)***
  The name of the variables that defines the mesh.
- **family_name:** ***str***
  The name of the family from which the percent will be computed and on which the cut is computed.
- **percent:** ***float*****, default=** ***None***
  The percentage relative to the family to determine the absolute position value.
- **position:** ***float*****, default=** ***None***
  The absolute position value relative to the family.

Main functions

class `antares.treatment.turbomachine.TreatmentExtractBladeLine.`**`TreatmentExtractBladeLine`**

`execute()`

Extraction of variables along a profile of a blade at a given radius.

Returns

Return type
`Base`

Example

```
import os

if not os.path.isdir('OUTPUT'):
    os.makedirs('OUTPUT')

import numpy as np

from antares import Reader, Treatment, Writer

#

# Data can be downloaded from
# https://cerfacs.fr/antares/downloads/application1_tutorial_data.tgz

r = Reader('bin_tp')
r['filename'] = os.path.join('..', 'data', 'ROTOR37', 'ELSA_CASE', 'MESH',
                             'mesh_<zone>.dat')
r['zone_prefix'] = 'Block'
r['topology_file'] = os.path.join('..', 'data', 'ROTOR37', 'ELSA_CASE',
                                  'script_topo.py')
r['shared'] = True
base = r.read()
print(base.families)

r = Reader('bin_tp')
r['base'] = base
r['filename'] = os.path.join('..', 'data', 'ROTOR37', 'ELSA_CASE', 'FLOW',
                             'flow_<zone>.dat')
r['zone_prefix'] = 'Block'
r['location'] = 'cell'
r.read()

base.set_computer_model('internal')

# Needed for turbomachinery dedicated treatments
base.cell_to_node()
base = base.get_location('node')
print(base.families)

base.compute('psta')
base.compute('Pi')
base.compute('theta')
P0_INF = 1.9
base.compute('MachIs = (((%f/psta)**((gamma-1)/gamma)-1.) * (2./(gamma-1.))  )**0.5' %␣
↪P0_INF)


# Extraction on blade
res_dir = os.path.join('OUTPUT', 'ON_BLADE')
if not os.path.isdir(res_dir):
```

(continues on next page)

(continued from previous page)

```
    os.makedirs(res_dir)

t = Treatment('extractbladeline')
t['base'] = base
t['family_name'] = 'BLADE'
t['percent'] = 0.75
t['coordinates'] = ['x', 'y', 'z']
ex_on_blade = t.execute()

writer = Writer('column')
writer['filename'] = os.path.join(res_dir, 'extract_on_blade.dat')
writer['base'] = ex_on_blade
writer.dump()
```

Extract Wake

Compute the wake in a specified 'x'-plane.

Parameters

- **base: `Base`**
  The base must contain:
  - the mesh coordinates x, y, and z
  - the solution
- **family_name:** ***str***
  The name of the family from which the percent will be computed.
- **x_value:** ***float*****, default=** ***None***
  The axial position of the plane.
- **r_percent:** ***float*****, default=** ***None***
  The 'r'-value given as a percentage of the radius.
- **r_value:** ***tuple(float)*****, default=** ***None***
  The radius value. The argument should be a tuple of min and max values.
- **cut_plane:** ***bool*****, default=** ***True***
  Flow contains an axial cut and the wake is extracted from this plane.
- **duplicate:** ***bool*****, default=** ***False***
  Duplication of the axial cut. Not needed for mixing-plane simulation as there is a periodicity condition.

Main functions

class antares.treatment.turbomachine.TreatmentExtractWake.**TreatmentExtractWake**

execute()

Execute the treatment.

Returns

Return type
Base

Example

```
import os

if not os.path.isdir('OUTPUT'):
    os.makedirs('OUTPUT')

import numpy as np

from antares import Reader, Treatment, Writer

#

# Data can be downloaded from
# https://cerfacs.fr/antares/downloads/application1_tutorial_data.tgz

r = Reader('bin_tp')
r['filename'] = os.path.join('..', 'data', 'ROTOR37', 'ELSA_CASE', 'MESH',
                             'mesh_<zone>.dat')
r['zone_prefix'] = 'Block'
r['topology_file'] = os.path.join('..', 'data', 'ROTOR37', 'ELSA_CASE',
                                  'script_topo.py')
r['shared'] = True
base = r.read()
print(base.families)

r = Reader('bin_tp')
r['base'] = base
r['filename'] = os.path.join('..', 'data', 'ROTOR37', 'ELSA_CASE', 'FLOW',
                             'flow_<zone>.dat')
r['zone_prefix'] = 'Block'
r['location'] = 'cell'
r.read()

base.set_computer_model('internal')

# Needed for turbomachinery dedicated treatments
base.cell_to_node()
base = base.get_location('node')
print(base.families)
```

(continues on next page)

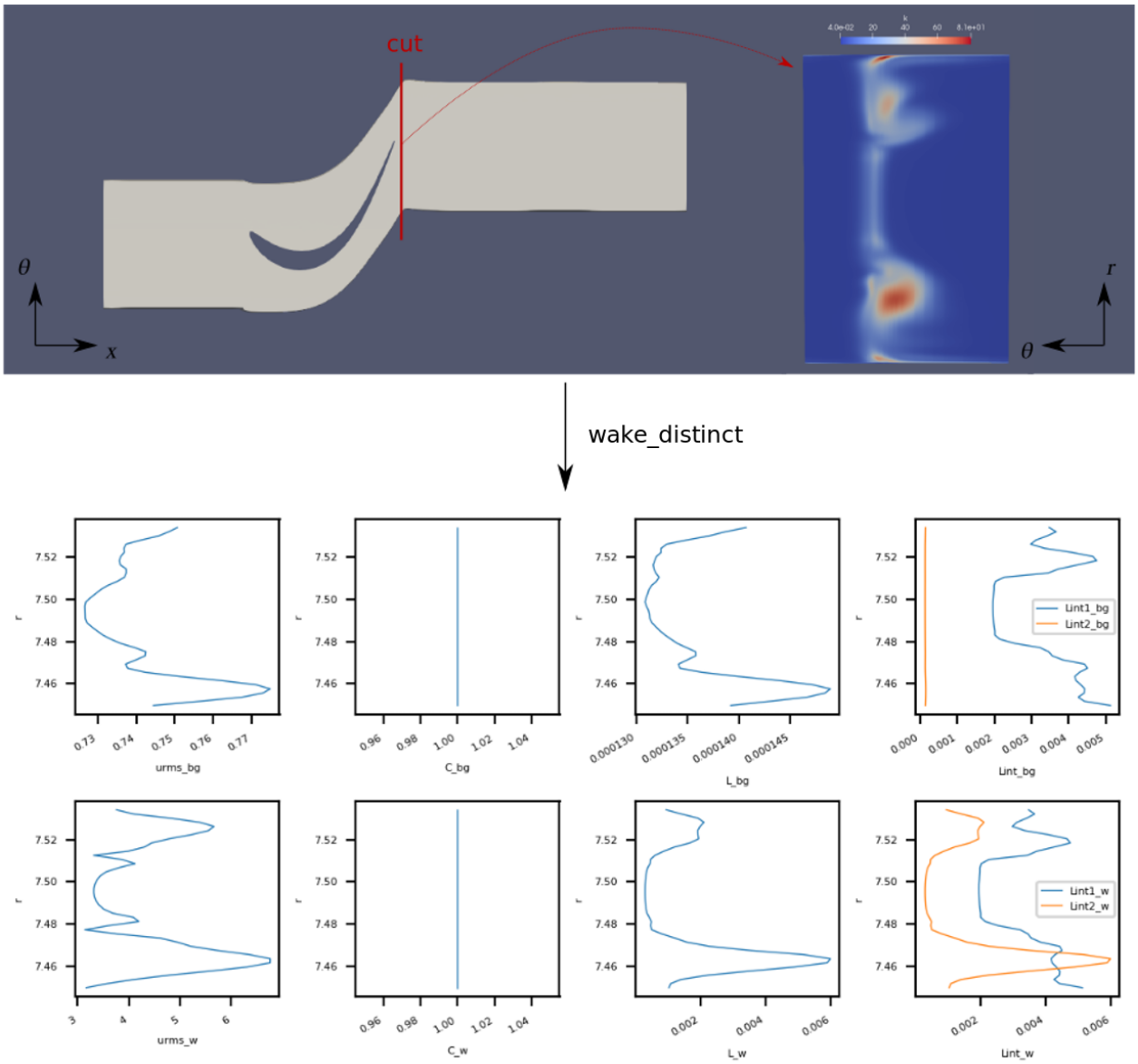
(continued from previous page)

```
base.compute('psta')
base.compute('Pi')
base.compute('theta')
P0_INF = 1.9
base.compute('MachIs = (((%f/psta)**((gamma-1)/gamma)-1.) * (2./(gamma-1.))  )**0.5' %␣
↪P0_INF)

# Definition of the treatment
t = Treatment('extractwake')
t['base'] = base
t['family_name'] = 'BLADE'
t['r_percent'] = 0.5
t['x_value'] = 5.0
wake = t.execute()

# Wake
res_dir = os.path.join('OUTPUT', 'WAKE')
if not os.path.isdir(res_dir):
    os.makedirs(res_dir)

writer = Writer('bin_tp')
writer['filename'] = os.path.join(res_dir, 'wake.plt')
writer['base'] = wake
writer.dump()
```

Wake Acoustic

Description

Analyse the wake turbulence on a structured revolution surface around the axis of a turbomachine.

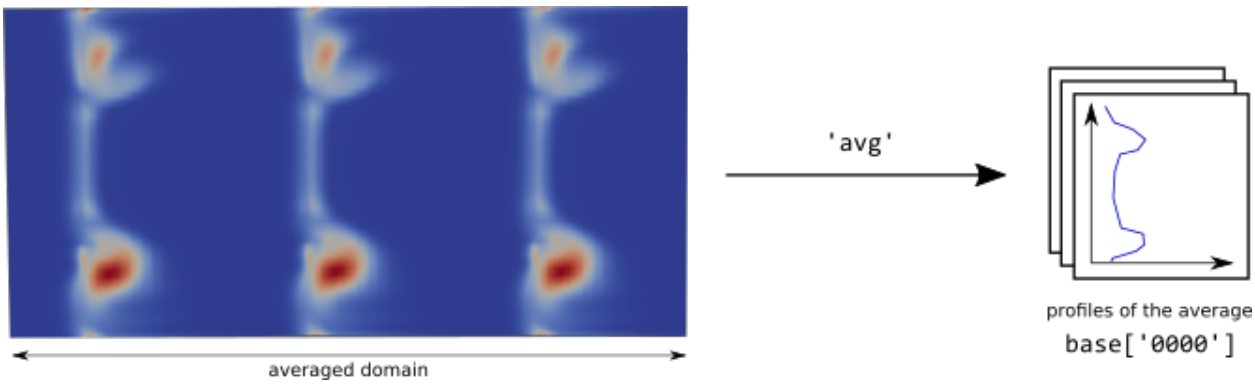


The turbulence of the flow is assumed to be modelled by the $k-\varepsilon$, $k-\omega$ or $k-l$ (Smith) models.

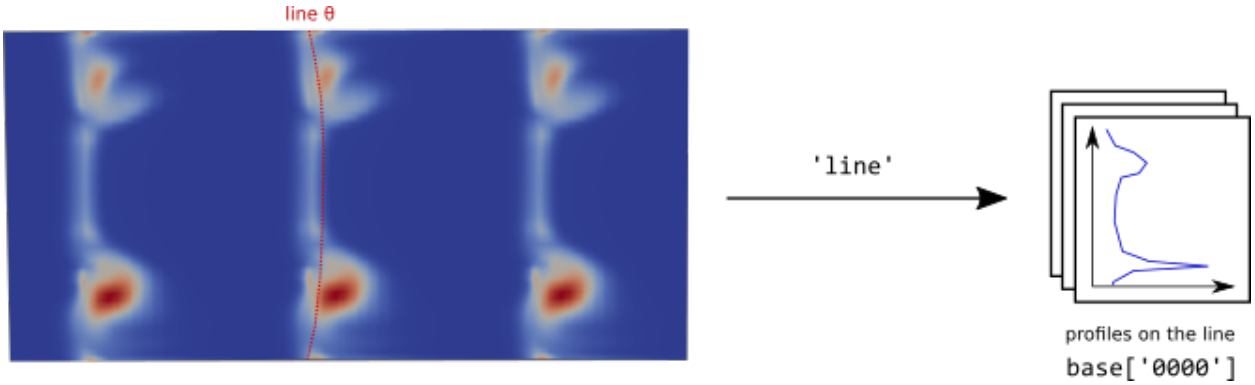
The analysis provided by the treatment is a set of radial profiles of values of interest: a set of variables defined by the user, plus $urms$ (turbulent velocity), $Lint_1$ (integral length scale estimated through a gaussian fit of the wake), $Lint_2$ (integral length scale estimated through turbulent variables), C and L (coefficients used to define $Lint_2$).

Different analyses are possible depending on the needs of the user:

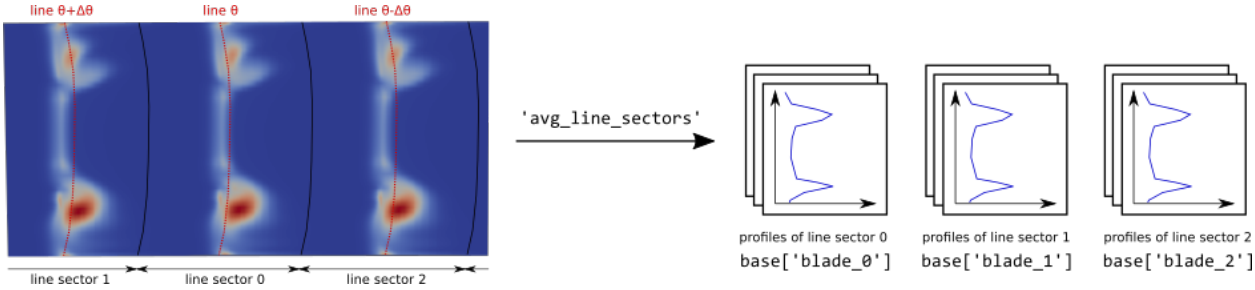
- **avg**: azimuthal average of values of interest at each radius.



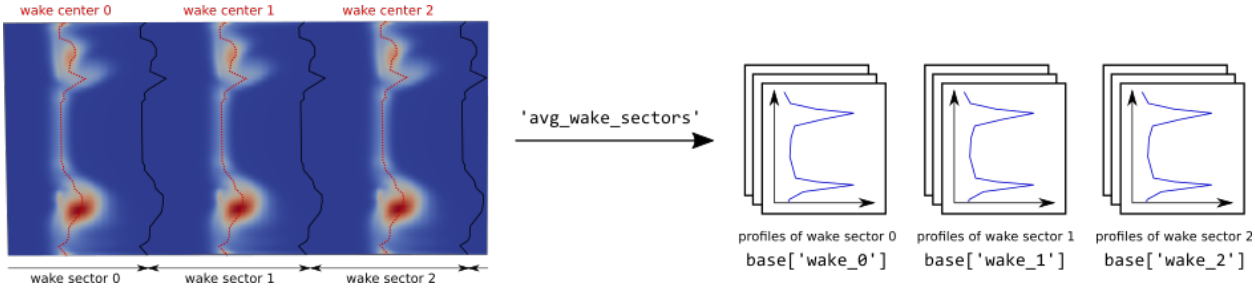
- **line**: extraction of values of interest on a line $\theta(r)$ defined by the user.



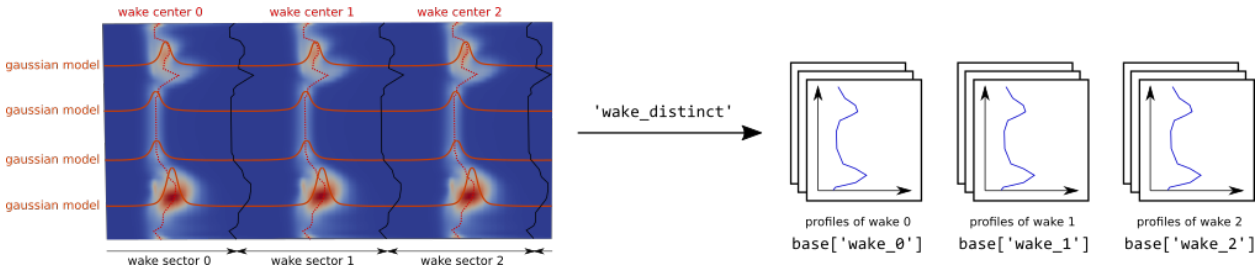
- **avg_line_sectors**: azimuthal average of values of interest over sectors centered on a $\theta(r)$ line.



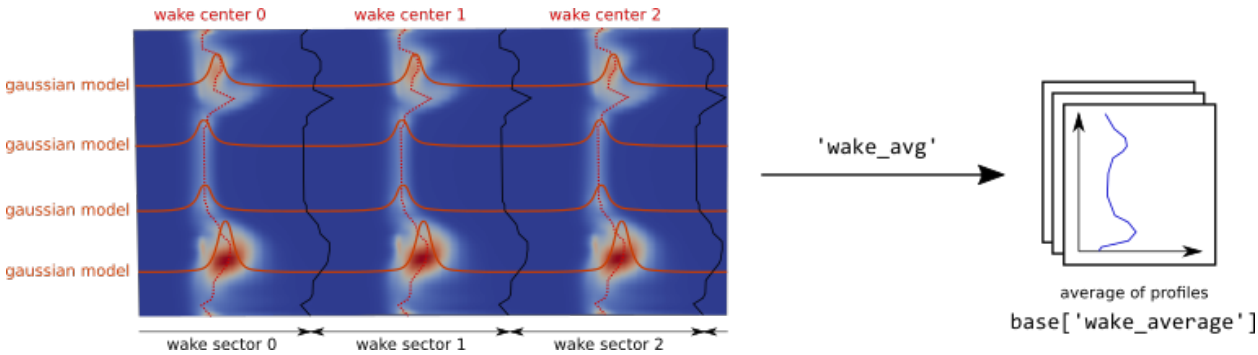
- **avg_wake_sectors**: azimuthal average of values of interest over sectors centered on the center of wakes.



- **wake_distinct**: extraction of values over sectors centered on the center of wakes with wake gaussian modeling.



- **wake_avg**: average over sectors of profiles of **wake_distinct**.



These analyses and the formulae used are documented in the *Formulae* section.

Parameters

- **base: `Base`**
  The input **base** on which the wake is analysed. Read section *Preconditions* to know the assumptions on this **base**.
- **cylindrical_coordinates: `list(str)`, default= *['x', 'r', 'theta']***
  Names of the cylindrical coordinates: the axis of rotation (axial coordinate), the distance to this axis (radius), and the azimuth.
- **type: `str`**
  Type of analysis to perform (possibilities are *avg*, *line*, *avg_line_sectors*, *avg_wake_sectors*, *wake_distinct* and *wake_avg*).
- **turbulence_model: `list(str)`**
  Name of the turbulence model used (possibilities are *k-epsilon*, *k-omega* and *k-l*).
- **turbulence_variables: `list(str)`**
  Name of the turbulence variables, *e.g. ['k', 'l']*.
- **coef_variable: `str`, default= *None***
  Name of the coefficient C used to compute $Lint2$. When *None*, the default formulation given in subsection *Extra Formulae* is used, otherwise the formulation given in the computer model of the **base** is used (optional).
- **static_temp_variable: `str`, default= *None***
  Name of the static temperature variable, used to compute the default formulation of C (optional when **coef_variable** is not *None*).
- **avg_variables: `list(str)`, default= *[]***
  Name of the variables to average (called $user\_variable$ in section *Formulae*). (optional).

- **nb_azimuthal_sector: `int`**
  The input **base** is considered as one azimuthal sector. This parameter is the number of azimuthal sectors needed to fill a 360 deg configuration (*e.g.* when the **base** is 360 deg, **nb_azimuthal_sector** has value 1, when **base** is < 360 deg, **nb_azimuthal_sector** has value more than 1). (optional when using *avg* analysis).
- **nb_blade: `int`**
  Number of blades per azimuthal sector. (optional when using *avg* and *line* analyses).
- **line_base: `Base`**
  The base representing the line $\theta(r)$. (optional: only for *line* and *avg_line_sectors* analyses).
- **wake_factor: `float`**
  A percentage (in [0, 1]) used for wake analysis. (optional: only for *wake_distinct* and *wake_avg* analyses).
- **wake_detection: `str` or `tuple(str,str)`, default= 'urms'**
  The physical quantity that is used to detect the wake. The gaussian fit is applied on this quantity. The quantity must be among 'V', 'Vx', 'tke', or 'urms'.

  If **wake_detection** is a tuple, the first string of the tuple must be among 'V', 'Vx', or 'tke'. The second string is the variable name found in the input base. 'V' is the magnitude of the velocity. 'Vx' is the axial component of the velocity vector. 'tke' is the turbulent kinetic energy.

  If the tuple notation is not used, then the variable in the input base must be among 'V', 'Vx', or 'tke'. The variables urms is always computed in the treatment.

  (optional: only for *wake_distinct* and *wake_avg* analyses).
- **option_X_bg: `str`, default= *avg***
  Option for background variables formulation, see *Extra Formulae*. (optional: only for *wake_distinct* and *wake_avg* analyses).
- **option_X_w: `str`, default= *avg***
  Option for wake variables formulation, see *Extra Formulae*. (optional: only for *wake_distinct* and *wake_avg* analyses).

Preconditions

The **base** must be 2D, and periodic in azimuth. The **base** must fulfill the following hypotheses: structured, monozone, shared radius and azimuth, regularity in r and theta (*i.e.* it is a set of rectangular cells in cylindrical coordinates), r is the first dimension and theta is the second dimension of the surface.

The **base** can be multi-instant.

The **line_base** must be 1D. If the **line_base** does not fulfill the following assumptions, the **line_base** is "normalized": structured, monozone, shared radius, radius values sorted strictly ascending.

The **line_base** must be mono-instant.

Postconditions

New variables are added to the input **base**.

The output base contains one zone per set of profiles.

Example

```
# construction of a 2D base fulfilling the assumptions (see Preconditions).
# with cylindrical coordinates: 'x', 'r', 'theta'
# using k-epsilon turbulence model, with turbulent variables: 'k', 'eps'
# representing 1 over 22 azimuthal sectors, with 2 blades per azimuthal sector

b.set_formula('C = 1.0') # user-defined C coefficient (chosen arbitrarily in this␣
↪example)

t = Treatment('wakeacoustic')
t['base'] = b
t['cylindrical_coordinates'] = ['x', 'r', 'theta']
t['type'] = 'avg_wake_sectors'
t['turbulence_model'] = 'k-epsilon'
t['turbulence_variables'] = ['k', 'eps']
t['coef_variable'] = 'C'
t['nb_azimuthal_sector'] = 22
t['nb_blade'] = 2
base_out = t.execute()

# plot profiles
# e.g. urms_bg profile with base_out['wake_0'].shared['r'], base_out['wake_0'][0]['urms_bg']
```

Known limitations

For the moment, the wake centers are only computed on the first *Instant*, and then, multi-instant bases are not correctly handled for **avg_wake_sectors**, **wake_distinct**, **wake_avg**. analyses.

Formulae

Definition of the Profiles

- **avg**
  - $user\_variable(r) = \text{mean}_\theta(user\_variable(r,\theta))$
  - $urms_{bg}(r) = \sqrt{\text{mean}_\theta(urms^2(r,\theta))}$
  - $urms_w(r) = 0$
  - $Lint1_{bg}(r) = \texttt{NaN}$
  - $Lint1_w(r) = \texttt{NaN}$
  - $C_{bg}(r) = \text{mean}_\theta(C(r,\theta))$
  - $C_w(r) = \text{mean}_\theta(C(r,\theta))$
  - $L_{bg}(r) = \text{mean}_\theta(L(r,\theta))$
  - $L_w(r) = \text{mean}_\theta(L(r,\theta))$
  - $Lint2_{bg}(r) = \text{mean}_\theta(Lint2(r,\theta))$
  - $Lint2_w(r) = \text{mean}_\theta(Lint2(r,\theta))$

- **line**

  In these formulae, $\theta(r)$ is a user-defined line as $line\_base$.

  - $user\_variable(r) = \text{mean}_\theta(user\_variable(r,\theta))$
  - $urms_{bg}(r) = \sqrt{\text{mean}_\theta(urms^2(r,\theta))}$
  - $urms_w(r) = 0$
  - $Lint1_{bg}(r) = \texttt{NaN}$
  - $Lint1_w(r) = \texttt{NaN}$
  - $C_{bg}(r) = C(r,\theta(r))$
  - $C_w(r) = C(r,\theta(r))$
  - $L_{bg}(r) = L(r,\theta(r))$
  - $L_w(r) = L(r,\theta(r))$
  - $Lint2_{bg}(r) = Lint2(r,\theta(r))$
  - $Lint2_w(r) = Lint2(r,\theta(r))$

- **avg_line_sectors**

Same formulae than **avg**, but the treatment generates one set of profiles per blade, on sectors centered on $\theta(r) + k \cdot \Delta\theta$ of azimuthal width $\Delta\theta = \frac{2\pi}{nb\_azimuthal\_sector \cdot nb\_blade}$, where $\theta(r)$ is given by the user as $line\_base$, $k \in \{0, \cdots, nb\_blade\}$.

- **avg_wake_sectors**

Same formulae than **avg**, but the treatment generates one set of profiles per blade, on sectors centered on $\theta(r,k)$ of azimuthal width $\Delta\theta = \frac{2\pi}{nb\_azimuthal\_sector \cdot nb\_blade}$, where $\theta(r,k)$ is the center of the wake k at radius r, $k \in \{0, \cdots, nb\_blade\}$.

- **wake_distinct**

  In these formulae, the set of θ is the subset of all azimuthal values of the current wake k.

  - $user\_variable(k,r) = \text{mean}_\theta(user\_variable(r,\theta))$
  - $urms_{bg}(k,r) = \sqrt{\text{min}_\theta(urms^2(r,\theta))}$ when existing wake, otherwise $\sqrt{\text{mean}_\theta(urms^2(r,\theta))}$
  - $urms_w(k,r) = a1$, given by the gaussian model when existing wake, otherwise 0
  - $Lint1_{bg}(k,r) = 0.21 \cdot \sqrt{\log 2} \cdot a3 \cdot r$, where $a3$ is given by the gaussian model when existing wake, otherwise `NaN`
  - $Lint1_w(k,r) = 0.21 \cdot \sqrt{\log 2} \cdot a3 \cdot r$, where $a3$ is given by the gaussian model when existing wake, otherwise `NaN`
  - $X_{bg}(k,r)$ **with** $X \in \{C, L, Lint2\}$**:**
    - if option_X_bg = 'avg' and existing wake, then $X_{bg}(k,r) = \text{mean}_{\theta bg}(X(r,\theta))$
    - if option_X_bg = 'min'` and existing wake, then $X_{bg}(k,r) = \min_{\theta bg}(X(r,\theta))$
    - if wake does not exist, $X_{bg}(k,r) = \text{mean}_\theta(X(r,\theta))$
  - $X_w(k,r)$ **with** $X \in \{C, L, Lint2\}$**:**
    - if option_X_w = 'avg' and existing wake, then $X_w(k,r) = \text{mean}_{\theta w}(X(r,\theta))$
    - if option_X_w = 'k_max' and existing wake, then $X_w(k,r) = X(r, \text{argmax}_\theta(urms^2))$
    - if option_X_w = 'gaussian_center' and existing wake, then $X_w(k,r) = X(r, a2)$
    - if wake does not exist, $X_w(k,r) = \text{mean}_\theta(X(r,\theta))$

where θw is the set of θ such that $\sqrt{urms^2 - \min_\theta(urms^2)} > wake\_factor \cdot \sqrt{\min_\theta(urms^2)}$, and θbg is its complement. Note that extracting values at $\text{argmax}_\theta(urms^2)$, *i.e.* using option_X_w = 'gaussian_center', can lead to discontinuities in the profiles, *e.g.* when $urms^2(\theta)$ has a "M"-shape near its maximum.

- **wake_avg**

  Average of profiles of **wake_avg**: $X(r) = \frac{1}{N}\sum_{k=0}^{N-1} X(k,r)$.

Gaussian Modelling

When using analyses **wake_distinct** and **wake_avg**, the wake is modelled at each radius by a gaussian function, enabling the computation of new values of interest (in particular $Lint1$).

This modelling is conditionned by the wake existence, which is fulfilled when there exists θ such that $urms\_env(\theta) > wake\_factor \cdot \sqrt{\min_\theta(urms^2)}$, where $urms\_env = \sqrt{urms^2 - \min_\theta(urms^2)}$.

When the wake exists, $urms\_env$ is fitted to the gaussian function $g(\theta) = a_1 \cdot e^{-\left(\frac{\theta - a_2}{a_3}\right)^2}$, parameterized by a_1, a_2 and a_3, using a nonlinear least squares method.

Extra Formulae

The variables $urms$, C and L are computed using the following relations, where k, ε, ω and l are the turbulent variables:

- $urms^2 = \frac{2}{3}k$.
- $Lint2 = C \cdot L$
- $L = \frac{k^{3/2}}{\varepsilon} = \frac{\sqrt{k}}{0.09 \cdot \omega} = \frac{l}{0.0848^{3/4}}$.
- $C = 0.43 + \frac{14}{Re_\lambda^1.05}$ or C is user-defined using *base.set_formula*.
- $Re_\lambda = \sqrt{\frac{20}{3} Re_L}$.
- $Re_L = \frac{k^2}{\varepsilon \cdot \nu} = \frac{k}{0.09 \cdot \omega \cdot \nu} = \frac{l \cdot \sqrt{k}}{0.0848^{0.75} \cdot \nu}$.
- $\nu = c_0 + c_1 \cdot T + c_2 \cdot T^2 + c_3 \cdot T^3$ (air kinematic viscosity), $c_0 = -3.400747 \cdot 10^{-6}$, $c_1 = 3.452139 \cdot 10^{-8}$, $c_2 = 1.00881778 \cdot 10^{-10}$ and $c_3 = -1.363528 \cdot 10^{-14}$.

Evaluate Spectrum

Evaluate Spectrum

Parameters

- **base: `Base`**
  The base must contain:
  - the mesh coordinates x, y, and z
  - the solution
  - 'hb_computation' as an `Base.attrs` (if HB/TSM type).
- **family_name:** ***str***
  The name of the family from which the percent will be computed.
- **x_percent:** ***tuple(float)*****, default=** ***None***
  The argument should be a tuple of min and max values. These limits the lower/upper bounds of the axial distribution. If not given, the lower/upper bounds of the axial distribution are computed.
- **r_percent:** ***tuple(float)*****, default=** ***None***
  The radius value given as a percentage of the radius. The argument should be a tuple of min and max values. These limits the lower/upper bounds of the radial distribution. If not given, the lower/upper bounds of the radial distribution are computed.
- **x_value:** ***float*****, default=** ***None***
  The absolute position value of the plane.
- **r_value:** ***tuple(float)*****, default=** ***None***
  The radius value. The argument should be a tuple of min and max values.
- **rho_inf:** ***float*****, default=** ***'in_attr'***
  The infinite density corresponding to the reference state.
- **v_inf:** ***float*****, default=** ***'in_attr'***
  The infinite axial velocity corresponding to the reference state.
- **num:** ***int*****, default=** ***100***
  The number of points in the radial distribution.

Main functions

class `antares.treatment.turbomachine.TreatmentEvalSpectrum.`**`TreatmentEvalSpectrum`**

`execute()`

Execute the treatment.

Returns

Return type
`Base`

Example

Global performance of turbomachinery

Global performance of turbomachinery (expressed in thermodynamic mean values, X_o standing for values of the outlet plane, and X_i standing for values of the inlet plane.

Mass flow rate: $Q = \int \rho V dS$

Total-to-total pressure ratio: $\Pi = \frac{Pt_o}{Pt_i}$

Isentropic efficiency:

compressor - $\eta_{is} = \frac{\Pi^{\frac{\gamma-1}{\gamma}} - 1}{\frac{Tt_o}{Tt_i} - 1}$ turbine - $\eta_{is} = \frac{\frac{Tt_o}{Tt_i} - 1}{\Pi^{\frac{\gamma-1}{\gamma}} - 1}$

Polytropic efficiency:

compressor - $\eta_{pol} = \frac{\frac{\log(\frac{Ps_o}{Ps_i})}{\log(\frac{\rho_o}{\rho_i})} * (\gamma - 1)}{\gamma * (\frac{\log(\frac{Ps_o}{Ps_i})}{\log(\frac{\rho_o}{\rho_i})} - 1)}$ turbine - $\eta_{pol} = \frac{\gamma * (\frac{\log(\frac{\rho_o}{\rho_i})}{\log(\frac{Ps_o}{Ps_i})} - 1)}{\frac{\log(\frac{\rho_o}{\rho_i})}{\log(\frac{Ps_o}{Ps_i})} * (\gamma - 1)}$

The specific gas constant is hard-coded: Rgaz = 287.053 [J/kg/K].

The Heat capacity ratio is hard-coded: gamma = 1.4 [-].

Parameters

- **inlet_base: `Base`**
  The inlet base used for global performance.
- **outlet_base: `Base`**
  The outlet base used for global performance.
- **coordinates: *list(str)***
  The variable names that define the set of coordinates. If no value is given, the default coordinate system of the base is used (see `Base.coordinate_names`).
- **type: *str***
  Type of turbomachine (turbine, compressor, CROR).
- **avg_type: *str in ['surface', 'massflowrate']*, default= *massflowrate***
  Type of space averaging.
- **conservative: *list(str)*, default= *['rho', 'rhou', 'rhov', 'rhow', 'rhoE']***
  Names of conservative variables. example: ['ro', 'rovx', 'rovy', 'rovz', 'roE']

Postconditions

The treatment returns a dictionary with variables `Q`, `eta_is`, `eta_pol`, and `Pi`.

- **Q:**
  mass-flow rate at the outlet plane,
- **eta_is**
  Isentropic efficiency.
- **eta_pol**
  Polytropic efficiency.
- **Pi**
  Total-to-total pressure ratio.

Main functions

class antares.treatment.turbomachine.TreatmentTurboGlobalPerfo.**TreatmentTurboGlobalPerfo**

execute()

Compute the global performance of a turbomachine.

Returns
a dictionary with variables `Q`, `eta_is`, `eta_pol`, and `Pi`

Return type
dict(str, float)

Variables

- **Q** – mass-flow rate at the outlet plane,
- **eta_is** – isentropic efficiency,
- **eta_pol** – polytropic efficiency,
- **Pi** – total-to-total pressure ratio

Example

```
import os

if not os.path.isdir('OUTPUT'):
    os.makedirs('OUTPUT')

import matplotlib.pyplot as plt
import numpy as np

import antares

x_inlet, x_outlet = -0.07, 0.15

perfos = np.empty((1, 4))

R = antares.Reader('bin_tp')
```

(continues on next page)

(continued from previous page)

```
R['filename'] = os.path.join('..', 'data', 'ROTOR37', 'rotor37.plt')
data = R.read()

data.set_computer_model('internal')

data.coordinate_names = ['x', 'y', 'z']

T = antares.Treatment('cut')
T['origin'] = (float(x_inlet), 0., 5.0)
T['type'] = 'plane'
T['normal'] = (1.0, 0.0, 0.0)
T['base'] = data
inlet_plane = T.execute()
inlet_plane.rel_to_abs(omega=-1.7999965e+03, angle=0.0)

T = antares.Treatment('cut')
T['origin'] = (float(x_outlet), 0., 5.0)
T['type'] = 'plane'
T['normal'] = (1.0, 0.0, 0.0)
T['base'] = data
outlet_plane = T.execute()
outlet_plane.rel_to_abs(omega=-1.7999965e+03, angle=0.0)

print(inlet_plane[0][0])
print(outlet_plane[0][0])
T = antares.Treatment('turboglobalperfo')
T['inlet_base'] = inlet_plane
T['outlet_base'] = outlet_plane
T['avg_type'] = 'massflowrate'
T['type'] = 'compressor'
T['coordinates'] = ['x', 'y', 'z']
T['conservative'] = ['ro', 'rovx', 'rovy', 'rovz', 'roE']
test = T.execute()
print(test)

perfos[0, 0] = -test['Q']*36
perfos[0, 1] = test['Pi']
perfos[0, 2] = test['eta_is']
perfos[0, 3] = test['eta_pol']

plt.figure()
plt.plot(perfos[:, 0], perfos[:, 1], 'D', label='MBS')
plt.xlabel(r'$\dot{m}\; [kg.s^{-1}]$', fontsize=22)
plt.ylabel(r'$\Pi$', fontsize=22)
plt.grid(True)
plt.legend(loc='best')
plt.tight_layout()
plt.savefig(os.path.join('OUTPUT', 'ex_Pi.png'), format='png')
# plt.show()

plt.figure()
plt.plot(perfos[:, 0], perfos[:, 2], 'D', label='MBS')
```

(continues on next page)

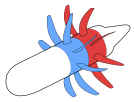
(continued from previous page)

```
plt.xlabel(r'$\dot{m}\; [kg.s^{-1}]$', fontsize=22)
plt.ylabel(r'$\eta_{is}$', fontsize=22)
plt.grid(True)
plt.legend(loc='best')
plt.tight_layout()
plt.savefig(os.path.join('OUTPUT', 'ex_eta_is.png'), format='png')
# plt.show()

plt.figure()
plt.plot(perfos[:, 0], perfos[:, 2], 'D', label='MBS')
plt.xlabel(r'$\dot{m}\; [kg.s^{-1}]$', fontsize=22)
plt.ylabel(r'$\eta_{pol}$', fontsize=22)
plt.grid(True)
plt.legend(loc='best')
plt.tight_layout()
plt.savefig(os.path.join('OUTPUT', 'ex_eta_pol.png'), format='png')
# plt.show()
```

Isentropic Pressure

No implementation yet available

Counter-Rotating Open Rotors performance evaluation



Performance for Contra-Rotating Open Rotors

Parameters

- **base: `Base`**
  The base must contain:
  - the axial forces ('flux_rou' and 'torque_rou')
  - two zones, 'front' and 'rear', to separate the contribution from each rotor
  - 'nb_blade' as an `Base.attrs`
  - 'n' the rotation frequency as an `Base.attrs`
  - 'D' the rotor diameter as an `Base.attrs`.
- **rho_inf:** ***float***
  Infinite density.
- **v_inf:** ***float***
  Infinite velocity.
- **duplication:** ***bool*****, default=** ***True***
  Duplication (rotation) of the forces. If not, for each row, the coefficients represent the forces, acting on a single blade, multiplied by the number of blades.

Main functions

class antares.treatment.turbomachine.TreatmentCRORPerfo.**TreatmentCRORPerfo**

execute()

Compute the similarity coefficients (forces) for a CROR. It is assumed that the forces come from a single canal computation (periodic or chorochronic when using HB/TSM approach). The forces are then duplicated by the number of blades.

Four coefficients are computed: the traction defined as

$C_t = |\frac{flux\_rou}{rho\_inf \cdot n^2 \cdot D^4}|$,

the power coefficient defined as

$C_p = |\frac{2\pi \cdot torque\_rou}{rho\_inf \cdot n^2 \cdot D^5}|$ (note the simplification of the rotation frequency on the expression of the power coefficient),

the propulsive efficiency computed from the traction and the power coefficients

$\eta = J\frac{C_t}{C_p}$, where J is the advance ratio defined as $J = |\frac{v\_inf}{n \cdot D}|$ (note that this widely used formulation for propeller might be reconsidered in presence of a second propeller. Indeed, the second rotor "doesn't see" the speed v_inf),

and the figure of merit

$FM = \sqrt{\frac{2}{\pi}}\frac{C_t^{3/2}}{C_p}$

These formulae are computed rotor per rotor. The global performance is evaluated as follow:

$C_t^{global} = C_t^{front} + C_t^{rear}$,

$C_p^{global} = C_p^{front} + C_p^{rear}$,

$\eta^{global} = \frac{J^{front} \cdot C_t^{front} + J^{rear} \cdot C_t^{rear}}{C_p^{global}}$

$FM^{global} = \sqrt{\frac{2}{\pi}}\frac{(C_t^{global})^{3/2}}{C_p^{global}}$

Returns

the input base with the forces, a new zone 'global', and as many instants as the input base has. If the input base comes from a HB/TSM computation, the mean, and the harmonics are also computed. Note that the amplitude of the harmonics are given divided by the mean value.

Boundary Layer

Description

Analyze wall-resolved boundary layers.

Parameters

- **base: `Base`**
  The input base.
- **coordinates: *list(str)***
  The physical Cartesian coordinate names.
- **bl_type: *str*, default= *aerodynamic***
  Type of boundary layer:
  - 'aerodynamic': aerodynamic boundary layer
  - 'thermal': thermal boundary layer
- **edge_crit: *int*, default= *7***
  Criteria used for the boundary layer edge detection. Add the integer value associated to each criterion to get the chosen combination.
  - 1. $dV/ds < 1\%$
  - 2. Stock & Haase
  - 4. isentropic velocity
  - 8. isentropic Mach number

  if **edge_crit** includes 8, then the edge results from the closest point from the wall above all criteria used. Otherwise, the edge comes from the average of all edges computed independent from each criteria.

  e.g.: **edge_crit** = 5 means a mix between the criteria 1 and 4.
- **wall_type: *str*, default= *resolved***
  Type of wall resolution ('resolved' or 'modeled').
- **mesh_type: *str* in ['2D', '2.5D', '3D'], default= *'3D'***
  Dimensionality of the mesh.
- **max_bl_pts: *int*, default= *100***
  Maximum number of points in the boundary layer.
- **added_bl_pts: *int*, default= *2***
  The number of points inside the boundary layer = number of calculated boundary layer points + **added_bl_pts**
- **smoothing_cycles: *int*, default= *0***
  Number of smoothing cycles for boundary layer results.
- **families: *dict(str: int) or list(str)***
  Surface parts on which the boundary layer is to be computed.

  If the value is a dictionary: The key value is a family name. This family name must be refered by the boundary condition objects. The boundary condition associated to this family name will be processed. The value associated to a key is a marker of the corresponding family. Two families may have the same marker.

  As an example,

  t['families'] = {'WingExt': 3, 'Flap': 4, 'WingInt': 3}

  Both 'WingExt' and 'WingInt' families have the same marker. Then, all corresponding mesh points will be processed alltogether. They will be concatenated in the output base.

  If the value is a list, then a different marker will be assigned to each item of the list.

  As an example,

t['families'] = ['WingExt', 'Flap', 'WingInt']

Value 0 is special, and applied to all points that do not belong to an input family. So if you give a marker 0 to a family, then you will get all boundary conditions in the ouput base.

- **only_profiles:** ***bool*****, default=** ***False***
  Only the profiles selected by **profile_points** will be calculated.
- **profile_points:** ***list(list(float))*****, default=** ***[]***
  Profiles will be plotted according to the surface points. If you give the coordinates of a point in the mesh, the boundary layer profile issued from the nearest point to the surface will be computed.
- **bl_parameters:** ***dict*****, default=** ***{}***
  Dictionary with keys in ["2D_offset_vector", "reference_Mach_number", "reference_velocity", "reference_pressure", "reference_density", "reference_temperature", "Prandtl_number", "gas_constant_gamma", "gas_constant_R"]
- **offsurf:** ***int*****, default=** ***-1***
  Distance from the wall. To get values on a surface lying at this distance from the wall.
- **eps_Mis:** ***float*****, default=** ***0.01***
  Threshold on the isentropic Mach number for **edge_crit** = 8.
- **variables:** ***list(str)*****, default=** ***['density', 'x_velocity', 'y_velocity', 'z_velocity', 'pressure']***
  Names of the input primitive variables (density, velocity components, static pressure).
- **nb_zone_passingthrough:** ***int*****, default=** ***50***
  Maximum number of zone a vector could pass through. When the maximum is reached, the process will stop to propagate the vector (will be cut before the end).
- **keep_interpolation_data:** ***bool*****, default=** ***False***
  Allow intermediary results (interpolation coefficients, C objects to remain instantiated in the treatment, in order not to be recomputed in a further call of the same treatement. Warning, the mesh must remain the same.
- **instant_selection:** ***int or str*****, default=** ***0***
  Instant name (or index) to execute in the base.
- **change_input_data_type:** ***bool*****, default=** ***False***
  C routines need to have a specific type for the provided variables. Mostly float64 for float, and int32 for int. And C ordered tables. If the provided base does not have the correct type and ordering, a retyped/ordered copy is performed. To save memory, this copy could replace the input variable. This will avoid to keep both tables instantiated at the same time, but may increase the memory used in the initial base (as new copies will mostly use a bigger type size). This copy will only apply to coordinates and retrieved variables.

Preconditions

Zones must be unstructured.

Input flow field values must be dimensionalized. The standard variables named **density**, **x_velocity**, **y_velocity**, **z_velocity** and **pressure** must be given in instants.

If **edge_crit** >= 8, then the variables **total_temperature** and **Mach_number** must also be given.

Postconditions

Three output bases.

If you give the coordinates of a point in the mesh, it will compute the boundary layer profile issued from the nearest point to the surface.

A profile is a line issued from a point (mesh point) on a surface that is perpendicular to this surface. The points along the boundary layer profile are computed from the volume grid points. A point along the boundary layer profile is the result of the intersection between the normal and a face of a volume element.

Example

```
import antares
myt = antares.Treatment('Bl')
myt['base'] = base
myt['coordinates'] = ['points_xc', 'points_yc', 'points_zc']
myt['families'] = {'Slat': 2, 'Flap': 4, 'Main': 3}
myt['mesh_type'] = '2.5D'
myt['max_bl_pts'] = 50
myt['added_bl_pts'] = 2
myt['smoothing_cycles'] = 0
myt['only_profiles'] = False
myt['bl_type'] = 'aerodynamic' # 'aerodynamic' 'thermal'
myt['offsurf'] = 0.5
myt['profile_points'] = [[1.0 , 1.0 , 0.0],]
t['bl_parameters'] = {'2D_offset_vector': 2, 'gas_constant_R': 287.0,
                      'reference_Mach_number': 0.1715, 'reference_velocity': 36.
↪4249853252,
                      'reference_pressure': 101325.0, 'reference_density': 3.14466310931,
                      'reference_temperature': 112.269190122, 'gas_constant_gamma': 1.4,
                      'Prandtl_number': 0.72, }
res_base, profbase, offsurfbase = t.execute()
```

Main functions

class antares.treatment.codespecific.boundarylayer.TreatmentBl.**TreatmentBl**

This class is used to analyze boundary layers.

The treatment is divided in many parts. Some of them are developed in C language to get CPU performance. C routines are made available in python with the module 'ctypes'.

1. read results and user parameters
2. build a volume element connectivity structure
3. compute the surface normals
4. interpolate the flow field values to the surface normals
5. compute the boundary layer edge see file calculate_boundary_layer_edge.c
6. define the local (s, n) coordinate system
7. compute the boundary layer values in the (s, n) coordinate system see file calculate_boundary_layer.c

8. modify the boundary layer edge
9. define the local (s, n) coordinate system
10. compute the boundary layer values in the (s, n) coordinate system

static asvoid(*arr*)

View the array as dtype np.void (bytes).

Based on https://stackoverflow.com/a/16973510/190597(Jaime, 2013-06) The items along the last axis are viewed as one value. This allows comparisons to be performed which treat entire rows as one value.

compute_constant_data(*base*)

Compute normal vectors and interpolation coefficients.

compute_interzonetopo(*base*, *cstzone*, *zone_loc_bnds*)

Create additional boundary to consider elements coming from other zone when doing normal vector calculation.

execute()

Analyze the boundary layer.

Returns
the base containing the results

Return type
`Base`

initialization()

Set the constant part.

Only needed once (both global then cstzone)

merge_prism_layer(*nb_zone_passingthrough*)

Merge prism layer layer between additional vector and main one.

Notations:

Indices δ and ∞ are used respectively for the boundary layer edge and for the free stream conditions.

's' denotes the coordinate of the streamwise direction, and 'n' the coordinate of the normal to the streamwise direction. 's' and 'n' form the (s, n) streamline-oriented coordinate system. This coordinate system is curvilinear, orthogonal and attached to the surface.

Boundary layer profile (normal to the wall):

If asked, the profiles are output in a formated ascii file containing:

- a first section (header) with free stream conditions (given as parameters):

Ma	Mach number	M_∞
P	Pressure	p_∞
Rho	Density	ρ_∞
T	Temperature	
V	Velocity modulus	V_∞
Cp	Pressure coefficient	
Ptot	Total pressure	
Pdyn	Dynamic pressure	

- a section that is repeated as long as there are profiles:

Profile number	
requested	Requested coordinates of the profile origin
real	True coordinates of the profile origin. Profiles are computed at surface grid points
Normal vector	Normal unit vector at the profile origin

then come the integral values:

Del	Boundary layer thickness
Dels	Displacement thickness [s-dir]
Deln	Displacement thickness [n-dir]
Delt	Thermal boundary layer thickness approximation
Thtss	Momentum thickness [s-dir]
Thtnn	Momentum thickness [n-dir]
H12	Incompressible shape factor

then the wall values:

FirstLayer height	a
P	Pressure
cp	Pressure coefficient
cfsI, cfnI	Skin friction coefficients [s, n dirs], V_∞
cfrI	Resulting skin friction coefficient, V_∞
cfsD, cfnD	Skin friction coefficients [s, n dirs], V_δ
cfrD	Resulting skin friction coefficient, V_δ
Tw	Temperature
Rhow	Density
muew	Dynamic viscosity
nuew	Kinematic viscosity
Taus	Shear stress component [s-dir]
Taun	Shear stress component [n-dir]
Taur	Resulting shear stress

then the edge values:

Prism height	Total height of the prism layer
Vdel	Velocity modulus
Ma	Mach number
ptot	Total pressure
cp	Pressure coefficient

finally, come different values along the profile:

S	Distance to the wall
S/S_edge	adimensionalized distance to the wall
Vx, Vy, Vz	Cartesian velocity components
Vres/V_inf	Velocity modulus normalized by the free stream velocity modulus
Vs/Ve	s normalized velocity component
Vn/Ve	n normalized velocity component
beta	Skew angle
log10(Y+)	logarithm of the nondimensional wall distance
Vr/V	Velocity modulus normalized by the shear stress velocity modulus
Vr/V*_theo	Velocity modulus normalized by the shear stress velocity modulus based on a flow over a flat plate (theory)
Rho	Density
cp_static	Static pressure coefficient
cp_dyn	Dynamic pressure coefficient
Ptot	Total pressure
PtotLoss	Total pressure loss
T	Temperature
Ttot	Total temperature
Trec	to be defined
Vi	Isentropic velocity modulus
Vi-Vr	Isentropic velocity modulus - velocity modulus
MassFlow	Mass flow
tke	Turbulent Kinetic Energy
mu_t/mu_l	ratio of viscosities
mu_t	Eddy viscosity
Mach	Mach number
Mach_is	Isentropic Mach Number

Boundary layer surface:

A surface contains in the mesh can be divided in many parts. A **marker** is a set of triangular and quadrilateral faces.

All the surfaces of a computation may be defined as families (or markers).

It can return a 2D field on the surface (or selected markers) with the following boundary layer quantities:

X, Y, Z	Cartesian coordinates of the surface point
VecNx, VecNy, VecNz	Cartesian coordinates of the surface normal vector
VxSurf, VySurf, VzSurf	Cartesian velocity components close to the wall [first grid layer] for wall streamlines
VxEdge, VyEdge, VzEdge	Cartesian velocity components at the boundary layer edge for inviscid streamlines
Vs, Vn	s, n velocity components close to the wall
beta_w	Profile skew angle between wall and boundary layer edge in the s, n coordinate system
Y+	Nondimensional wall distance
delta	Boundary layer thickness, δ
deltas	Displacement thickness [s-dir] based on the velocity at the boundary layer edge, δ_s^*
deltan	Displacement thickness [n-dir] based on the velocity at the boundary layer edge, δ_n^*

continues on next page

Table 7 – continued from previous page

deltat	Thermal boundary layer thickness, δ_T (approximation)
deltaes	Kinetic energy thickness [s-dir]
deltaen	Kinetic energy thickness [n-dir];
thetass	Momentum thickness [s-dir], θ_{ss}
thetann	Momentum thickness [n-dir], θ_{nn}
H12	Incompressible shape factor. A shape factor is used in boudary layer flow to determine the nature of the flow. The higher the value of H, the stronger the adverse pressure gradient. A large shape factor is an indicator of a boundary layer near separation. A high adverse pressure gradient can greatly reduce the Reynolds number at which transition into turbulence may occur. Conventionally, H = 2.59 (Blasius boundary layer) is typical of laminar flows, while H = 1.3 - 1.4 is typical of turbulent flows.
Ptot	Total pressure
cp	Pressure coefficient
Ma_edge	Mach number at the boundary layer edge
CfsInf, CfnInf	Skin friction coefficients [s, n dirs], V_∞
CfrInf	Resulting skin friction coefficient, V_∞
CfsDel, CfnDel	Skin friction coefficient [s, n dirs], V_δ
CfrDel	Resulting skin friction coefficient, V_δ
maxEddyVisc	Maximum eddy viscosity along the profile
T_wall	Wall temperature
T_edge	Temperature at the boundary layer edge
T_rec	Temperature at the boundary layer edge, TO BE DETAILED
profTag	Profile tag
pointMeshIndex	Index of the point in the array of grid points
heightPrismLayers	Total height of the prism layer
nbPrismLayers	Number of prism layers
BlThickOut-PrismThick,	(delta - heightPrismLayers)/delta*100
firstLayerHeight	Height of the first prism layer
ptsExceeded	1 if the number of profile points is close to the maximum number of boundary layer points given as input, NOT YET AVAILABLE
profFound	100 tells that the boundary layer edge is found
Lam/Turb	Transition Laminar/Turbulent
nbBlPts	Number of points in the boundary layer

The mathematical definitions are given below:

Displacement thicknesses (based on the velocity at the boundary layer edge):	$\delta_s^* = \int_0^\delta (1 - \frac{\rho}{\rho_\delta}\frac{V_s}{V_\delta})ds$ (deltas), $\delta_n^* = -\int_0^\delta (1 - \frac{\rho}{\rho_\delta}\frac{V_n}{V_\delta})ds$ (deltan)
Momentum thicknesses:	$\theta_{ss} = \int_0^\delta \frac{\rho}{\rho_\delta}\frac{V_s}{V_\delta}(1 - \frac{V_s}{V_\delta})ds$ (thetass), $\theta_{nn} = \int_0^\delta \frac{\rho}{\rho_\delta}\frac{V_n}{V_\delta}(1 - \frac{V_n}{V_\delta})ds$ (thetann)
Kinetic energy thicknesses:	$\theta_{es} = \int_0^\delta \frac{\rho}{\rho_\delta}\frac{V_s}{V_\delta}(1 - \frac{V_s^2}{V_\delta^2})ds$ (deltaes), $\theta_{en} = \int_0^\delta \frac{\rho}{\rho_\delta}\frac{V_n}{V_\delta}(1 - \frac{V_n^2}{V_\delta^2})ds$ (deltaen)
Shape factors:	$H12 = \frac{\delta_s^*}{\theta_{ss}}$, $H22 = \frac{\delta_n^*}{\theta_{nn}}$
Pressure coefficients:	$Cp = \frac{p - p_\infty}{\frac{1}{2}\rho_\infty V_\infty^2}$ (cp_static), $Kp = \frac{p_{tot} - p_\infty}{\frac{1}{2}\rho_\infty V_\infty^2}$ (cp_dyn)
Skin friction coefficients:	$Cf_\delta = (\mu_w + \mu_t)\frac{\frac{\partial V}{\partial w}\vert_w}{\frac{1}{2}\rho_w V_\delta^2}$, $Cf_\infty = (\mu_w + \mu_t)\frac{\frac{\partial V}{\partial w}\vert_w}{\frac{1}{2}\rho_\infty V_\infty^2}$
	$Y^+ = Y\frac{V^\star}{\nu}$
Velocity normalized by the shear stress velocity:	$V^+ = \frac{V}{V^\star}$
Shear stress at the wall:	$\tau_w = \mu_w \frac{\partial u}{\partial y}$
Shear stress velocity (friction velocity) at the wall:	$V^\star = \sqrt{\frac{\tau_w}{\rho_w}}$
Total pressure loss:	$\frac{\Delta P_t}{P_t} = \frac{P_t - P_{t\infty}}{P_{t\infty}}$ (PtotLoss)
Skew angles:	$\beta = \gamma - \gamma_\delta$, $\beta_w = \gamma_w - \gamma_\delta$

Users must keep in mind the following restrictions when analyzing their results.

Restrictions:

Regions:

There are two types of regions where an exact boundary layer edge can not be found:

1. **The velocity component Vs becomes very small or zero:**
    1. **Stagnation point regions**
       At a stagnation point, Vs=0. Then, there is no boundary layer thickness. Near stagnation points, it is also difficult to find a boundary layer thickness accurately.
    2. **Trailing edge regions**
       In thick trailing edge regions, Vs becomes very small.
2. **The surface normals do not leave the boundary layer:**
    1. **Intersection regions (wing-body, body-tail, etc)**

1. The surface normal does not leak from the boundary layer.
2. The surface normal passes the boundary layer from another component first.
3. The surface normal hits another surface.

No accurate boundary layer edge can be found in these cases.

Thickness:

The best situation is when the boundary layer edge lies in the prism layer. In this case, the difference between the true boundary layer edge and the grid point laying on the profile that is upper (or lower) than this true edge is small.

If the boundary layer edge lies outside the prism layer, then the diffence may be more important. Of course, that depends on the mesh quality, but, in general, mesh cell ratio increase quite fast outside the prism layer. In this case, the computed boundary layer thickness can be overestimated, and the step between two successive points on the profile can be large. Then, it is necessary to check the displacement (or momentum) thickness to check the lower pointsand the upper point (respective to the boundary layer edge).

Boundary layer edge:

The edge of the boundary layer is detected with the following multistage algorithm.

A first guess of the boundary layer edge is made with four different methods (see input parameter **edge_crit**) (function `calculate_boundary_layer_edge()`):

1. the first derivative of the boundary layer profile velocity is smaller than the 1% and the step between one derivative and the previous one is smaller than the 0.01%. (function `edge_velocity_slope()`)
2. $\delta = \epsilon y_{max}$ with y_{max} is the wall distance for which the diagnostic function $y^a|\frac{\partial u}{\partial y}|^b$ is maximum. The first 10 points are omitted from this maximum search. The constants for a turbulent boundary layer are evaluated from Coles velocity profiles: $a = 1, b = 1, \epsilon = 1.936$. The constants for a laminar boundary layer come from quasi-similar compressible solutions including heat transfer effects: $a = 3.9, b = 1, \epsilon = 1.294$. The reference is [STOCKHAASE].
4. the relative difference between the isentropic velocity and the velocity of the profile is smaller than a threshold.

   $V_r = \sqrt{V_x^2 + V_y^2 + V_z^2}$ and $V_i = V_\infty \sqrt{\frac{2}{(\gamma-1)*M_\infty^2} * (1 - (0.5 * \frac{2*(P-P_\infty)}{\rho_\infty * V_\infty^2} * \gamma * M_\infty^2 + 1)^{(\gamma-1)/\gamma}) + 1}$

   and

   The edge is the first point giving $\frac{|V_r - V_i|}{Vr} < \epsilon_{V_i}$ with $\epsilon_{V_i} = 2.5\,10^{-3}$

   If none, then the edge is given by $\min \frac{|V_r - V_i|}{Vr}$
8. hypothesis: the isentropic Mach number is constant in the boundary layer. Comparing with the Mach number gives the edge. $P_{t_{is}} = P_{ref}(T_t/T_{ref})^{\gamma/(\gamma-1)}$, $T_{t_{is}} = (P_{t_{is}}/P)^{(\gamma-1)/\gamma}$, $M_{is} = \sqrt{2|T_{t_{is}} - 1|/(\gamma-1)}$

   $\delta M = M_{is} - M$, $\delta M_{min} = \min(\delta M)$, $M_{is_{cor}} = M_{is} - \delta M_{min}$.

   The edge is the first point giving $|M - M_{is_{cor}}|/M_{is_{cor}} < \epsilon_{M_{is}}$ with $\epsilon_{M_{is}} = 0.01$ by default.

The boundary layer edge is at the position where the first derivative of "V+" becomes approximately zero.

Start defining a range for the edge search. This range, stored in "points", will be the 160% of the first guess of the boundary layer edge.

The first step is to search the first maximum of the first derivative of "V+" and then, starting from this maximum, search where the first derivative is smaller than two. Once this first edge is found, the next maximum is compared to the first one. If it is bigger, search again the boundary layer edge starting from this new maximum.

When the final edge is found, perform two checks. If the thickness of the boundary layer is greater than two, move the edge back to the last maximun or minimum of the first derivate of "V+" before the distance to the wall is two.

The second check is done using the isentropic velocity. Search for the point where the isentropic velocity meets the profile velocity but only until three points below the current boundary layer edge. If the point is found, average it with the current edge to get the final and corrected boundary layer edge.

Interpolation:

The interpolation of the CFD values to the normal (profile) is done with a polynomial expansion of the form $G^i = a + bx + cy$, which represents a linear variation of G in both x and y directions within a triangular element. The three constants a, b, and c are computed with the three triangle points. more details can be found in [CHUNG].

Note: Sutherland law for molecular viscosity: $\mu = \frac{C_1\sqrt{T_w}}{1+\frac{S}{T_w}}$ with $C_1 = 1.4610^{-6} kg.m^{-1}.s^{-1}.K^{-1/2}$ and $S = 112.K$ (functions `Skinfric()` of calculate_boundary_layer.c and `bl_profile()`, `clear_values()` of calculate_boundary_layer_edge.c)

Note: Function U^+ versus y^+:

linear part: $U^+ = y^+$ for $y^+ < 11.63$

log part: $U^+ = 5.75 * \log(y^+) + 5.5$ for $y^+ > 11.63$

Note: if not given, T_wall is computed with the equation of state of perfect gas: $T_w = \frac{\gamma}{(\gamma-1)C_p}\frac{P}{\rho}$ where $C_p = 1,003.41 J.kg^{-1}.K$ is here the specific heat at constant pressure (at T = 250 K), and $\gamma = \frac{C_p}{C_v}$ given as input, with C_v the specific heat at constant volume

Note: The boundary layer thickness (δ) is the average of a numerical value (distance of the boundary layer edge from the wall) and a correlation value coming from the shape factor (function `Thickness()` of calculate_boundary_layer.c).

$\delta = \frac{\delta_{sf} + \delta_{num}}{2}$

$\delta_{sf} = \delta^{*I} * (\frac{\theta_s^I}{\delta_s^{*I}} H_1 + 1)$ where the superscript I means that the incompressible formula is used and with H_1 the mass flow shape factor computed with a correlation of Green [GREEN].

Note: δ_T (deltat) is the E. Polhausen approximation of the thermal boundary layer thickness for Prandtl number greater than 0.6

$\delta_T = \frac{\delta}{Pr^{1/3}}$

Note: Numerical values that are not available as inputs are computed with:

$$M_a = \frac{V}{\sqrt{(\gamma * R * T)}}$$

$P_{tot} = (1 + .2 * M_a * M_a)^{3.5} * P$ (so $\gamma = 1.4$)

Example

```
import os
if not os.path.isdir('OUTPUT'):
    os.makedirs('OUTPUT')

import antares

r = antares.Reader('netcdf')
r['filename'] = os.path.join('..', 'data', 'BL', 'EUROLIFT_2D', 'grid1_igs_Local_Sweep.nc
↪')
base = r.read()

r = antares.Reader('netcdf')
r['base'] = base
r['filename'] = os.path.join('..', 'data', 'BL', 'EUROLIFT_2D', 'sol_Local_Sweep_ETW1.
↪pval.30000')
r.read()

t = antares.Treatment('Bl')
t['base'] = base
t['coordinates'] = ['points_xc', 'points_yc', 'points_zc']
t['families'] = {'Slat_2': 2, 'Flap_4': 4, 'Main_3': 3}
t['mesh_type'] = '2.5D'
t['max_bl_pts'] = 50
t['added_bl_pts'] = 2
t['smoothing_cycles'] = 0
t['only_profiles'] = False
t['bl_type'] = 'thermal' # 'aerodynamic' 'thermal'
t['bl_type'] = 'aerodynamic' # 'aerodynamic' 'thermal'
t['offsurf'] = 0.5
t['profile_points'] = [[1.0 , 1.0 , 0.0],
                       [2.0 , 1.0 , 0.0],
                       [3.0 , 1.0 , 0.0],
                       [5.0 , 1.0 , 0.0],
                       [7.0 , 1.0 , 0.0],
                       [9.0 , 1.0 , 0.0],
                       [10.0, 1.0,  0.0],
                       [15.0, 1.0,  0.0]
                      ]
t['bl_parameters'] = {'2D_offset_vector': 2, 'gas_constant_R': 287.0,
                      'reference_Mach_number': 0.1715, 'reference_velocity': 36.
↪4249853252,
                      'reference_pressure': 101325.0, 'reference_density': 3.14466310931,
```

(continues on next page)

(continued from previous page)

```
                        'reference_temperature': 112.269190122, 'gas_constant_gamma': 1.4,
                        'Prandtl_number': 0.72, }
res_base, profbase, offsurfbase = t.execute()

w = antares.Writer('bin_tp')
w['base'] = res_base
w['filename'] = os.path.join('OUTPUT', 'ex_blsurf.plt')
w.dump()

if offsurfbase:
    w = antares.Writer('bin_tp')
    w['base'] = offsurfbase
    w['filename'] = os.path.join('OUTPUT', 'ex_offblsurf.plt')
    w.dump()
```

Ffowcs Williams & Hawkings Analogy

Table of Contents

Description

This treatment predicts the fluctuating pressure in far-field using the Ffowcs Williams & Hawkings Analogy. The formulation 1A (Farassat 2007) and the formulation 1C (Najafi-Yazdi 2011) are available for both static surfaces and surfaces in a rotating frame of reference. The advanced time approach (Casalino, 2003) is used for the prediction of acoustic fields. Variants of the Ffowcs Williams - Hawkings equation introduced by Morfey in 2007 and Shur in 2005 are also available (Spalart 2009). The mean flow is assumed to be along the $\vec{x}$ direction, if not the reference frame must be rotated to satisfy this condition.

The interested reader can make reference to the papers of Casalino (2003) and Najafi-Yazdi et al. (2011).

The FW-H acoustic analogy involves enclosing the sound sources with a control surface that is mathematically represented by a function, $f(\mathbf{x}, t) = 0$. The acoustic signature at any observer position can be obtained from the FW-H equation:

$$p'(\mathbf{x}, t) = \frac{\partial}{\partial t} \int_{f=0} \left[\frac{Q_i n_i}{4\pi |\mathbf{x} - \mathbf{y}|} \right]_{\tau_e} \mathrm{d}S - \frac{\partial}{\partial x_i} \int_{f=0} \left[\frac{L_{ij} n_j}{4\pi |\mathbf{x} - \mathbf{y}|} \right]_{\tau_e} \mathrm{d}S + \frac{\partial^2}{\partial x_i \partial x_j} \int_{f>0} \left[\frac{T_{ij}}{4\pi |\mathbf{x} - \mathbf{y}|} \right]_{\tau_e} \mathrm{d}V$$

where $[]_{\tau_e}$ denotes evaluation at the emission time τ_e. and V represents the volume outside the control surface. The source terms under the integral sign are:

$$Q_i = \rho \left(u_i - v_i \right) + \rho_0 v_i$$
$$L_{ij} = \rho u_i \left(u_j - v_j \right) + P_{ij}$$

and T_{ij} is referred to as Lighthill's stress tensor:

$$T_{ij} = \rho u_i u_j + \left[(p - p_0) - c_0^2 \left(\rho - \rho_0 \right) \right] \delta_{ij} - \sigma_{ij}$$

The vectors $\mathbf{u}$ and $\mathbf{v}$ are the flow and the surface velocities, respectively. The compression tensor P_{ij} is defined as:

$$P_{ij} = \left(p - p_0 \right) \delta_{ij} - \sigma_{ij}$$

The three source terms in the formal definition of $p'(\mathbf{x}, t)$ are known as the thickness (Q_i, monopole), loading (L_{ij}, dipole) and quadrupole (T_{ij}) source terms, respectively.

Features

- Farassat 1A in a medium at rest.
- Convective Nayafi-Yasdi 1C (with mean flow).
- Both unstructured & structured surfaces.
- Static surfaces and surfaces with a rotating frame of reference.
- Porous and solid formulations.
- Multi-code tool, tested and validated on:
  - AVBP
  - elsA

- ProLB
- Cosmic (University of Leicester)
- FLUSEPA

Parameters

- **base: `Base`, default= *None***
  The surface base on which the FWH surface integral will be computed. It can contain several zones and several instants. The object will be modified at the output.
- **meshfile: *list(str)*, default= *None***
  A two element list or tuple containing, first the name of the meshfile, followed by the reader format used to read the mesh.
- **datafile: *list(str)*, default= *None***
  A two element list or tuple containing, first the name of the datafile, followed by the reader format used to read the data.

  > **Warning:** This keyword only works if the base is written such as every instant is stored in a different file.

- **modify_surface: *function*, default= *None***
  A function used to modify the FWH surface at runtime. See here for more details.
- **coordinate_names: *list(str)*, default= *None***
  A three element list or tuple containing the name variable of spatial coordinates. If `None`, the names will be inferred from the database using the `antares.Constants.KNOWN_COORDINATES` list.
- **surface_normal_components: *list(str)*, default= *None***
  A three element list or tuple containing the variable name of the three surface normal components.
- **compute_normal_vectors: *function*, default= *None***
  The function used to compute the surface normal vectors and align them in the right direction. See *Compute normal vectors* (page 318) for more details. Additionally, *FWH Utils* (page 319) offers a few functions that can be used to compute the normal vectors in most scenarios.
- **redim: *list(float)*, default= *None***
  A list of 6 floats used to convert the following values: length, density, velocity, pressure, temperature, and time.
- **volume_base: `Base`, default= *None***
  The volume base on which the FWH volume integral will be computed. It can contain several zones and several instants. The object will be modified at the output.
- **volume_meshfile: *list(str)*, default= *None***
  A two element list or tuple containing, first the name of the volume meshfile, followed by the reader format used to read the mesh.
- **volume_datafile: *list(str)*, default= *None***
  A two element list or tuple containing, first the name of the datafile containing the volume base, followed by the reader format used to read the data.

  > **Warning:** This keywords only works if the base is written such as every instant is stored in a different file

- **modify_volume:** ***function*****, default=** ***None***
  A function used to modify the FWH volume at runtime. See here for more details.
- **pressure_variable:** ***str*****, default= 'pressure'**
  A string containing the name of the pressure variable.
- **pressure_equation:** ***str*****, default=** ***None***
  A string containing an equation to compute the pressure from the variables present in the base.
- **velocity_variables:** ***list(str)*****, default=** ***None***
  A list of strings containing the name of the velocity components.
- **velocity_equations:** ***list(str)*****, default=** ***None***
  A list of strings containing the equations to compute the velocity variables from the variables present in the base.
- **density_variable:** ***str*****, default=** ***None***
  A string containing the name of density variable.
- **density_equation:** ***list(str)*****, default=** ***None***
  A string containing an equation to compute the density variable from the variables present in the base.
- **obs_file:** ***str***
  A column-type file defining the position for each observer in the far-field. The base must contain the variables `'x'`, `'y'`, and `'z'`.
- **type:** ***str*****, default=** ***porous***
  The type of acoustic analogy formulation: `'porous'` or `'solid'`.
- **analogy:** ***str*****, default=** ***1A***
  Acoustic analogy formulation: `'1A'` or `'1C'`.
- **form:** ***str*****, default=** ***density***
  Form of the acoustic analogy (pressure or density based):
  - `'density'` stand for the original FW-H formulation: $\rho' = \rho - \rho_0$
  - `'pressure1'` stand for the Morfey modification: $\rho^\star = \rho_0 + p/c_0^2$
  - `'pressure2'` stand for Shur modification: $\rho^\diamond = \rho_0(1 + p/p_0)^{1/\gamma}$
- **quadrupole_term:** ***str*****, default=** ***None***
  - `None`: do not compute the quadrupole term.
  - `'full'`: compute the volume + surface integrals (a volume base must be specified).
  - `'only'`: compute **only** the volume integral (a volume base must be specified).
  - `'cancel_spurious_noise'`: compute the quadrupole surface correction following the model of Rahier et al. (2015)
  - `'frozen_turbulence'`: compute the quadrupole surface correction following the model of Ikeda et al. (2017)
- **eddy_convective_velocity:** ***str*****, default=** ***None***
  The type of eddy convective velocity needed to compute the `'frozen_turbulence'` quadrupole correction term
  - `'temporal_mean'`: velocity time average.
  - `'mean_flow'`: mean flow in the x direction.
- **pref:** ***float*****, default=** ***101325.0***
  The value of the reference pressure in Pa.

- **tref:** ***float*, default=** ***298.0***
  The value of the reference temperature in K.
- **mach:** ***float*, default=** ***0.0***
  Ambient medium Mach number for analogy 1C.
- **R_gas:** ***float*, default=** ***287.058***
  Mass-specific gas constant in J/(Kg K) = m2/(s2 K).
- **gamma:** ***float*, default =** ***1.4***
  Heat capacity ratio (c_p/c_v).
- **mesh_kinetics:** ***str*, default =** ***static***
  Describes the type of movement of the mesh. Two possible values:
  - `'static'`: a static mesh.
  - `'rotating_reference_frame'`: a mesh with a rotating reference frame. The angular velocity and the rotation axis are assumed to be constant.
- **rotation_velocity:** ***float*, default =** ***0***
  Angular velocity in rad/s for a mesh in a rotating reference frame.
- **rotation_axis:** ***list(float)*, default =** ***[0, 0, 0]***
  The direction of the rotation axis of a mesh in a rotating reference frame.
- **dt:** ***float*, default=** ***None***
  Simulation time step.
- **sample:** ***int*, default=** ***1***
  The number of time steps between two instants. The Δt between two instants should be equal to $dt \times$ sample.
- **start_propagation_at:** ***float*, default =** ***None***
  The physical time with respect to the source at which the FWH propagation should start. If `None` the propagation starts from the first instant in the database.
- **end_propagation_at:** ***float*, default =** ***None***
  The physical time with respect to the source at which the FWH propagation should end. If `None` the propagation will be performed until the last instant in the database.
- **initial_time:** ***float*, default =** ***0.0***
  The physical time of the first instant in the database.
- **nb_revolutions:** ***int*, default=** ***1***
  Number of extra revolutions to be propagated. This assumes that the database contains only one revolution or a partial revolution and **partial_periodicity** is set to `True`.
- **partial_periodicity:** ***bool*, default=** ***False***
  If `False`, the input base is assumed to contain a full 360 ^{o} revolution. If `True`, the input base is assume to contain a partial part of a full rotation and the missing parts are assumed to be periodic with respect to the input base.
- **derivative_order: int, default= 1,**
  Order of the finite difference scheme used to compute the time derivatives.
- **output:** ***str*, default=** ***fwh_result***
  Output file name (hdf_antares format).
- **output_contributions:** ***bool*, default=** ***False***
  It `True`, the output base will contain the individual thickness loading and quadrupole noise contributions.

- **verbose:** ***bool*****, default=** ***True***
  If `True`, the output will print the information of the treatment progress and performance. If `False`, nothing will be printed.

Preconditions

The following conditions must be met in order to ensure a correct execution of the treatment:

- The treatment depends on mpi4py (even if run without MPI).
- All the zones must have the same number of instants.
- In case of a single zone base, all the cell elements must be of the same type.
- The observer file must be in a `'column'` format.
- The mean flow is assumed to be along the $\vec{x}$ direction. If not, the reference frame must be rotated to satisfy this condition.
- The rotation axis is assumed to pass by the origin of the reference frame. If not, the reference frame must be translated to satisfy this condition.
- The rotation axis direction must not be the zero vector when **mesh_kinetics** = `'rotating_reference_frame'`.
- The mesh topology is assumed to be constant when **mesh_kinetics** = `'rotating_reference_frame'`.

Postconditions

The output base contains $N+1$ zones where N is the number of observers. By convention, the first zone contains one instant with only one variable, the time vector common to all observers. The rest of the zones contain the information for each observer. They all contain also one instant with the pressure related variables and the convergence vector.

The base also stores useful information in its attributes. The base level attributes contain the information of the value of all keywords used in the treatment (including the default ones). If the value of a keyword is an *antares.Base*, the stored attributes will be the string "User defined". If the value of a keyword is a function, only the name of the function will be saved in the attributes.

If the library GitPython[120] is installed in the system, the following attributes will be saved:

- The path to the antares repository.
- The name of the current branch.
- The SHA of the latest commit.
- If the repository is in a dirty state (A git repo is dirty if there are modified tracked files and/or staged changes, but untracked content doesn't count)

Additionally, several system related information is also saved in the attributes such as:

- The host system information as provided by the platform.uname()[121] function.
- The path and version of the current python binary.
- The current working directory.
- The name and arguments of the main script.
- The date at which the treatment ended.

[120] https://gitpython.readthedocs.io/en/stable/
[121] https://docs.python.org/3/library/platform.html

- The CPU time.
- The number of MPI procs used.
- The path of the script containing the FWH treatment.

Furthermore, the zone level attributes store the spatial coordinates for each observer in the variables "x", "y", and "z".

Write a file in hdf_antares format (default: fwh_result.h5).

Modify surface and volume databases

This treatment allows the user to modify the database at run-time. This could allow the user to modify the geometry on which perform the FWH treatment (by removing nodes, zones, etc). This can also be used to modify the physical variables in a more complex way than what it is allowed with the **redim** or **equations** keywords

For performance reasons, this modification is not done over the whole database at once, but one instant at a time during the FWH main execution loop. This reduces the memory footprint of the treatment as the whole database will not be uploaded into memory. This also allows to implement some additional performance optimizations.

The modification is done by using the functions provided by the user using the **modify_surface** and **modify_volume** keywords. These keywords require as value a function that accepts 3 arguments:

1. **base**: An `antares.Base` with only one instant (the instant currently being processed).
2. **it**: an integer containing the current iteration index starting from 0.
3. **user_defined_vars**: a dictionary in which the user can store any variable to be re-used for future iterations.

A skeleton and usage of this function is shown below:

```
import antares

def modify_surface(base: antares.Base,
                   it: int,
                   user_defined_vars):

    # Modify base
    new_base = # ...

    # return modified base
    return new_base

treatment = antares.Treatment('fwh')
treatment['modify_surface'] = modify_surface
# .
# .
# .
treatment.execute()
```

Internally, the FWH treatment stores the geometric and the physical variables in different bases. If **mesh_kinetics** is either `'static'` or `'rotating_reference_frame'`, the variable `geometry` is taken from the output of the `modify_surface` function at `it=0`. Trying to modify the geometric variables for `it>0` will have no effect.

> **Warning:** The treatment does not check that the modification applied to the base are consistent with what the treatment needs.

> The user must ensure consistency between all the instants and the geometry. This is especially true when trying to remove nodes, cells or zones. They must be removed from each instant.

Compute normal vectors

In order to correctly propagate the noise using the FWH analogy, the normal vectors of the FWH surface must be computed. These vectors **must** point outwards. If the vector components are already present in the original database, then the user can use **surface_normal_components** keywords to indicate the variable names. If they are not present in the database, the user should use the **compute_normal_vectors** to compute the normal vectors, already pointing outwards.

This keywords accepts a function of 3 arguments:

1. **geometry**: an `antares.Base` with as many zones as the original input database, and only one instant containing the spatial coordinates as variables.
2. **it**: an integer representing the current iteration in the FWH loop
3. **coord_names**: a list of 3 strings with the names of the spatial coordinates

And the function must return a list with the names of the 3 variables representing the normal vectors components.

A skeleton of implementation and usage of this function is shown below:

```
import antares
from typing import List

def compute_normals(geometry: antares.Base,
                    it: int,
                    coord_names: List[str]):

    # Compute normal vectors for all zones
    for zone in geometry.keys():
        geometry[zone][0]['normal_x'] = # ...
        geometry[zone][0]['normal_y'] = # ...
        geometry[zone][0]['normal_z'] = # ...

    # return the name of the newly created variables storing the normal vector components
    return ('normal_x', 'normal_y', 'normal_z')

treatment = antares.Treatment('fwh')
treatment['compute_normal_vectors'] = compute_normals
# .
# .
# .
treatment.execute()
```

In the case that **mesh_kinetics** is either `'static'` or `'rotating_reference_frame'`, the function `compute_normals` is only called once at the beginning of the treatment for `it=0`.

> **Warning:** The treatment does not check that the normal vectors are in fact pointing outwards. It is up to the user to ensure this condition.

See *here* (page **??**) for predefined helper functions to calculate the normal vectors.

FWH Utils

Table of Contents

Extract converged signal

`antares.utils.FWH.`**`extract_converged_signal`**(*base*, *subtract_mean=False*, *start_at_zero=False*)

Extract the converged part of a FWH result base.

The converged signal corresponds to the part of the signal where the convergence variable is at its maximum. An additional check if performed to ensure that this converged region corresponds to a continuous temporal signal. A warning is printed otherwise.

The output base contains as many zones as observers. Each zone contains one instant. The instant contains the converged time vector and the converged pressure variables.

Additionally, the pressure signals can be shifted by subtracting its mean. And the vector time can be shifted so it always starts at zero for each observer.

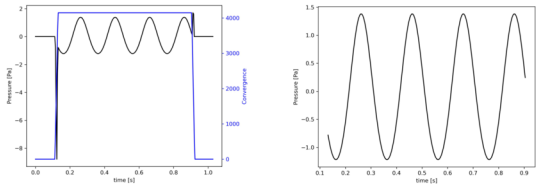
Parameters

- **base** (Base) – The base containing the FWH results
- **subtract_mean** (*bool*) – True if should subtract the mean value for each contribution.
- **start_at_zero** (*bool*) – True if should shift the observers time vectors so they all start at zero.

Returns

An antares Base with the converged signals

```
import antares
from antares.utils.FWH import extract_converged_signal

reader = antares.Reader('hdf_antares')
reader['filename'] = 'my_fwh_results.h5'
base = reader.read()

converged_results = extract_converged_signal(base, subtract_mean=False, start_at_
↪zero=False)
```



Raw convergence and pressure results

Only converged pressure

Find signal periodicity

`antares.utils.FWH.`**`find_periodicity`**(*base*, *equal_for_all_obs=False*)

Cut the observer signal to make it as periodic as possible.

This helps to reduce the spectral leakage when doing the FFT Treatment.

The cutting point is defined as the point where the cross-correlation between the first and second halves of the cut signal is maximum.

Parameters

- **base** (Base) – The base containing converged FWH results
- **equal_for_all_obs** (*bool*) – True if the cutting point is the same for all observers, this will reduce computational time

Returns

An antares Base with the same format as the input base, but with the signal cut for all observers.

Usage:

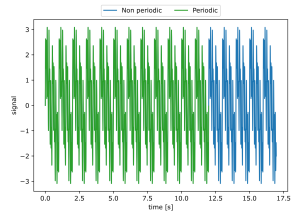
```
import antares
from antares.utils.FWH import find_periodicity

reader = antares.Reader('hdf_antares')
reader['filename'] = 'fwh_results.h5'
base = reader.read()

periodic_base = find_periodicity(base, equal_for_all_obs=False)
```

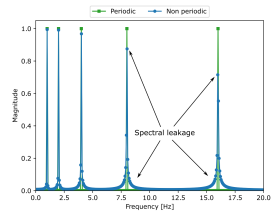
The following example illustrates the effects of the spectral leakage[122] when a FFT is performed on a non perfectly periodic signal. The signal corresponds to the function:

$$y(t) = sin(2\pi t) + sin(4\pi t) + sin(8\pi t) + sin(16\pi t) + sin(32\pi t)$$

This signal has a period $T = 2\pi$ but it was sampled from $[0, 5.5\pi]$ which does not correspond to an integer multiple of T. We remark that doing a Fourier transform over such signal leads to wrong amplitude values as well as parasitic noise. The function `antares.utils.FWH.find_signal_periodicty` cuts the signal at the right place to have a perfectly periodic signal. The FFT of such signal is exactly 5 peaks at the right frequencies and with the correct magnitude.

[122] https://en.wikipedia.org/wiki/Spectral_leakage





Add FWH results

`antares.utils.FWH.`**`add_fwh_results`**(*bases*)

Add the pressure signal (and contributions) of different FWH result bases into one single base.

Parameters

bases (list(`Base`)) – list of bases containing different fwh results.

Returns

A base with the same format as the FWH result base but with the pressure signal added for each observer.

> **Warning:** The time discretizations for all input bases are assumed to be identical.

Usage:

```
import antares
from antares.utils.FWH import add_fwh_results

reader1 = antares.Reader('hdf_antares')
reader1['filename'] = 'fwh_surface_1.h5'
surface1 = reader1.read()

reader2 = antares.Reader('hdf_antares')
reader2['filename'] = 'fwh_surface_2.h5'
surface2 = reader2.read()

total_results = add_fwh_results([surface1, surface2])
```

Print Attributes

`antares.utils.FWH.print_fwh_attributes(base)`

Print base level attributes of FWH result bases in a pretty format.

The attributes are split into 4 different categories:

- Treatment keywords.
- Git information.
- System information.
- Miscellaneous information.

The attributes of each category are printed in alphabetical order (case insensitive).

Parameters
base (Either `Base` or a str with the path for the base.) – The base containing the FWH results.

Normal vectors helper functions

The module **antares.utils.FWH** contains several helper functions that implement the most common scenarios to compute the normal vectors:

Compute normal vectors using mesh topology

`antares.utils.FWH.compute_normals_from_topology(invert=False)`

Compute the normal vectors using the connectivity list of the mesh.

If the connectivity list is correctly ordered, the normal vectors computed from the connectivity will all by oriented in the same way (either inwards or outwards).

Parameters
invert (*bool*) – False if the normal should be computed using the order in the connectivity list. True if they should be inverted.

Usage:

```
import antares
from antares.utils.FWH import compute_normals_from_topology

treatment = antares.Treatment('fwh')
treatment['compute_normal_vectors'] = compute_normals_from_topology(invert=False)
# .
# .
# .
treatment.execute()
```

Compute normal vectors and auto-orient them

antares.utils.FWH.**compute_normals_auto_orient**(*orientation=1*, *extra_params={}*)

Compute the normal vectors using the treatment 'cellnormal' and tries to auto-orient them in the same direction using the 'auto_orient' parameter.

The orientation can be changed using the orientation parameter and extra parameters can be passed to the treatment using the extra_params argument.

The limitations of this function are the same limitations as the 'auto_orient' feature in the 'cellnormal' treatment.

Parameters

- **orientation** (*int*) – The orientation parameter for the 'cellnormal' treatment.
- **extra_params** (*dict*) – Any other parameter that should be passed to the 'cellnormal' treatment.

Usage:

```
import antares
from antares.utils.FWH import compute_normals_auto_orient

treatment = antares.Treatment('fwh')
treatment['compute_normal_vectors'] = compute_normals_auto_orient(orientation=1,␣
↪extra_params={})
# .
# .
# .
treatment.execute()
```

Compute normal vectors and orient them using a reference point

antares.utils.FWH.**compute_normals_orient_using_point**(*ref_point*)

Compute the normal vectors and orient them so all the vectors are pointing away from a reference point.

Parameters

ref_point (*list(float)*) – The reference point

Usage:

```
import antares
from antares.utils.FWH import compute_normals_orient_using_point

treatment = antares.Treatment('fwh')

# all normal vectors will point away from the origin ([0, 0, 0])
treatment['compute_normal_vectors'] = compute_normals_orient_using_point([0, 0, 0])
# .
# .
# .
treatment.execute()
```

Modify surface and volume databases helper functions

`antares.utils.FWH.`**`keep_nodes`**(*variables*, *thresholds*)

Modify the FWH database keeping only the nodes that respect a given threshold for a given list of variables.

Parameters

- **variables** (*list(str)*) – List of variables to which apply the threshold
- **thresholds** – List of tuples containing the lower and upper limit for the threshold. If either the lower or upper limit is None, the threshold will not be applied to that limit.

Usage:

```
import antares
from antares.utils.FWH import keep_nodes

treatment = antares.Treatment('fwh')

# Keep all nodes with a pressure less than 1e15 and with a density between 0 and␣
↪1e15
treatment['modify_surface'] = keep_nodes(['pressure', 'density'], [(None, 1e15), (0,
↪ 1e15)])
# .
# .
# .
treatment.execute()
```

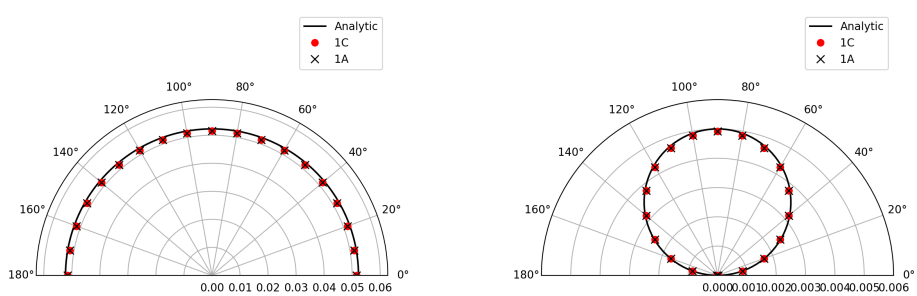
Validation

This treatment was validated using a serie of academic test cases for which the theoretical solution can be derived. These test cases can be found in the papers of Casalino (2003) and Najafi-Yazdi et al. (2011):

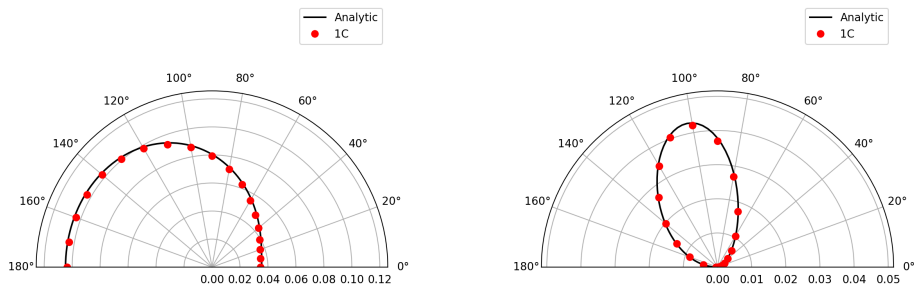
- Static monopole, dipole or quadrupole in a static or moving medium.
- Rotating monopole in a static or moving medium.
- Rotating dipole in a static medium.

Below, we can find a few results of these test cases:

- Farfield directivity pattern (rms pressure) of a static point monopole (left) and a point dipole (right) measured at $r = 40l$, radiating in a medium at rest (M=0):



- Farfield directivity pattern (rms pressure) of a static point monopole (left) in a flow at M=0.5 and a point dipole (right) in a flow at M=0.8, measured at $r = 40l$

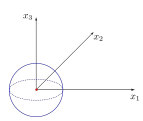


Examples

Example 1: Acoustic directivity of a static monopole

The following example shows the setup to compute the acoustic field using a static mesh. The test case corresponds to a single monopole located at the origin. The chosen FWH surface is a sphere of 50cm diameter centered at the monopole position. A scheme and an example of the generated database is presented below:

The treatment script can be found in **antares/examples/treatment/fwh_mono.py** and the post-treatment script: **antares/examples/treatment/post_fwh_mono.py**




```
"""
Ffowcs Williams & Hawkings Analogy

In parallel: mpirun -np NPROCS python fwh_mono.py
In serial: python fwh_mono.py
"""
import os
if not os.path.isdir('OUTPUT'):
    os.makedirs('OUTPUT')

import antares
from antares.utils.FWH import compute_normals_from_topology



# Read base
r = antares.Reader('bin_tp')
r['filename'] = os.path.join('..', 'data', 'FWH', 'Monopole', 'monopole_<instant>.plt')
base = r.read()
```

(continues on next page)

(continued from previous page)

```
# Apply FWH treatment
treatment = antares.Treatment('fwh')
treatment['base'] = base
treatment['analogy'] = '1A'
treatment['type'] = 'porous'
treatment['compute_normal_vectors'] = compute_normals_from_topology()
treatment['obs_file'] = os.path.join('..', 'data', 'FWH', 'fwhobservers_monopole.dat')
treatment['pref'] = 97640.7142857
treatment['tref'] = 287.6471952408
treatment['dt'] = 0.00666666666667
treatment['output'] = os.path.join('OUTPUT', "fwh_result")
fwhbase = treatment.execute()
```

Post-treatment example: The directivity pattern

The following script shows an example of how to compute the directivity pattern in the $x_1 - x_2$ plane

```
"""

Post-Treatment script for a FW-H solver output database

input: FW-H database
output: raw signal, RMS, SPL, OASPL

"""

# ------------------------------------------------------------ #
# Import modules
# ------------------------------------------------------------ #

import os
if not os.path.isdir('OUTPUT'):
    os.makedirs('OUTPUT')

import numpy as np
import matplotlib.pyplot as plt
from antares import *
from antares.utils.Signal import *


# ------------------------------------------------------------ #
# Input parameters
# ------------------------------------------------------------ #

# Input path
IN_PATH = 'OUTPUT' + os.sep
INPUT_FILE = ['fwh_result.h5']
# Can sum different surface if needed, then give a list
# INPUT_FILE = ['surface1.h5', 'surface2.h5', 'surface3.h5']
```

(continues on next page)

(continued from previous page)

```
# Output path
OUT_PATH = 'OUTPUT' + os.sep
# output name extension
NAME = 'FWH_'
# output name extension
EXT = 'CLOSED'

# Azimuthal Average
AZIM = False

# Convert in dB
LOG_OUT = False

# Keep the largest convective time (azimuthal mean should not work)
MAX_CONV = False

# Extract converged pressure signal in separate files
RAW = False

# Extract R.M.S from raw pressure fluctuation (only if AZIM=False)
RMS = False

# Compute Third-Octave
ThirdOctave = False

# PSD parameter
PSD_Block = 1
PSD_Overlap = 0.6
PSD_Pad = 1

# smoothing filter for psd (Only for a broadband noise)
Smooth = False
# 1: mean around a frequency, 2: convolution
type_smooth = 1
# percent of frequency to filter
alpha = 0.08

# filter initial signal (if frequency = -1.0 no filtering)
Filter = False
Lowfreq = -1.0
Highfreq = -1.0

# downsampling signal if necessary
downsampling = False
# 1: order 8 Chebyshev type I filter, 2: fourier method filter, 3: Gaussian 3 points␣
↪filter
type_down = 3
# downsampling factor
step = 3

# frequency to keep for OASPL (if not all, give a range [100, 1000] Hz)
FREQ_RANGE = 'all'
```

(continues on next page)

(continued from previous page)

```
# Observers informations
# number of obs. in azimuthal direction
NB_AZIM = 1
# Angle between two microphone in  the streamwise direction
dTH = 45.
# Minimum and maximum angle of observers in the streamwise direction
a_min = 0.0
a_max = 315.0
# observers
observers = np.arange(a_min, a_max+dTH, dTH)
nb_observer = len(observers)*NB_AZIM

# ------------------------------------------------------------- #
# Script template
# ------------------------------------------------------------- #

print(' \n>>> ------------------------------------------------')
print(' >>>        Post-processing of FW-H results')
print(' >>> ------------------------------------------------')

nbfile = len(INPUT_FILE)
PSD = [PSD_Block, PSD_Overlap, PSD_Pad]
FILTER = [Filter, Lowfreq, Highfreq, Smooth, type_smooth, alpha, downsampling, type_down,
↪ step]
obs_info = [nb_observer, NB_AZIM, dTH, observers]
outfile = [OUT_PATH, NAME, EXT]

spl_datafile, oaspl_datafile, oaspl_raw_datafile, raw_datafile = build_outputfile_
↪name(outfile, FILTER, AZIM, LOG_OUT)

if nbfile == 1:
    r = Reader('hdf_antares')
    r['filename'] = IN_PATH+INPUT_FILE[0]
    base = r.read()
else:
    r = Reader('hdf_antares')
    r['filename'] = IN_PATH+INPUT_FILE[0]
    base = r.read()
    for file in INPUT_FILE[1:]:

        r = Reader('hdf_antares')
        r['filename'] = IN_PATH+file
        base2add = r.read()

        # Sum signal
        #  + CONV=True: only the common converged part of each surfaces is keep
        #  + CONV=False: Keep all the converged part
        base = AddBase2(base, base2add, CONV=True)

# Keep only converged signal
#   + if MAX_CONV = False the signal will be truncated to the common converged part␣
↪between each observers
```

(continues on next page)

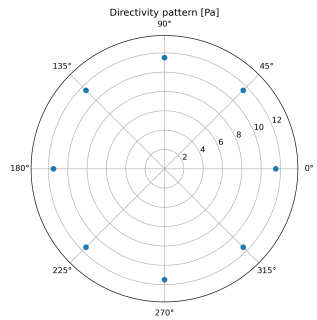
(continued from previous page)

```
base, observers_name = prepare_data(base, obs_info, MAX_CONV=True)

# Compute SPL and OASPL
# OUT_BASE
SPL, OASPL, OASPL_RAW, RAW, SPL_BAND = compute_FWH(base, obs_info, PSD, FILTER,
                                                   MAX_CONV, LOG_OUT,
                                                   FREQ_RANGE,
                                                   RAW_DATA=RAW,
                                                   AZIMUTHAL_MEAN=AZIM,
                                                   CHECK_RMS=RMS,
                                                   BAND=ThirdOctave)

# Checking if the output_path exists
OUT_PATH = '.' + os.sep
if not os.path.exists(OUT_PATH):
    os.makedirs(OUT_PATH)

# write result
w = Writer('column')
w['base'] = SPL
w['filename'] = spl_datafile
w.dump()

w = Writer('column')
w['base'] = OASPL
w['filename'] = oaspl_datafile
w.dump()

if OASPL_RAW is not None:
    w = Writer('column')
    w['base'] = OASPL_RAW
    w['filename'] = oaspl_raw_datafile
    w.dump()

if RAW is not None:
    w = Writer('column')
    w['base'] = RAW
    w['filename'] = raw_datafile
    w.dump()

if SPL_BAND is not None:
    index = spl_datafile.find('.dat')
    spl_third_octave = spl_datafile[:index] + '_third_octave' + spl_datafile[index:]
    w = Writer('column')
    w['base'] = SPL_BAND
    w['filename'] = spl_third_octave
    w.dump()



## Plot directivity pattern
fig = plt.figure(figsize=(6.2,6))
```

(continues on next page)

(continued from previous page)

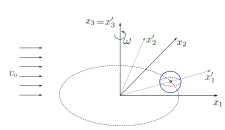
```
ax = fig.add_subplot(111, polar=True)
theta_plot = OASPL[0][0][0]*np.pi/180
p_rms = (OASPL[0][0][1])**0.5

plt.title('Directivity pattern [Pa]')
ax.plot(theta_plot, p_rms, marker='o', linestyle='')
ax.set_rmax(p_rms.max()*1.2)
plt.savefig(os.path.join('.', 'OUTPUT', 'directivity.png'), dpi=300)
```



Example 2: Pressure signal of a rotating monopole

The treatment script can be found in **antares/examples/treatment/fwh_rotating_reference_frame.py** and the post-treatment script: **antares/examples/treatment/post_fwh_rotating_reference_frame.py**

The following example shows how to setup a case with a mesh in a rotating reference frame. The test case is a single acoustic monopole rotating around the axis x_3 with an angular velocity equals to $2\pi\frac{rad}{s}$. There is a absolute mean flow ($M = 0.5$) in the x_1 direction. There are two different frame of references: $x_1 - x_2 - x_3$ is the absolute static frame of reference and $x_1' - x_2' - x_3'$ is frame of reference rotating with the monopole. All the quantities of the FWH surface are computed with respect to this rotating frame of reference. The figures below show a schematic representation of the test case and an example of the database



The script to study this case is as follow:

```
import antares
import os
import numpy as np
from antares.utils.FWH import compute_normals_orient_using_point

if not os.path.isdir('OUTPUT'):
    os.makedirs('OUTPUT')

## Read base
reader = antares.Reader('bin_tp')
reader['filename'] = os.path.join('..', 'data', 'FWH', 'Rotating_monopole', 'rotating-
↪monopole-x1x2_<instant>.plt')
```

(continues on next page)

(continued from previous page)

```
base = reader.read()

# Apply FWH treatment
treatment = antares.Treatment('fwh')
treatment['base'] = base
treatment['analogy'] = '1C'
treatment['type'] = 'porous'
treatment['obs_file'] = os.path.join('..', 'data', 'FWH', 'Rotating_monopole',
↪'fwhobservers.dat')
treatment['mach'] = 0.5
treatment['pref'] = 97640.71428571429
treatment['tref'] = 287.6471952407826
treatment['dt'] = 0.005025125628140704
treatment['compute_normal_vectors'] = compute_normals_orient_using_point([1, 0, 0])
treatment['rotation_velocity'] = 2*np.pi*1.0
treatment['rotation_axis'] = [0, 0, 1]
treatment['mesh_kinetics'] = 'rotating_reference_frame'
treatment['pressure_variable'] = 'p'
treatment['density_variable'] = 'rho'
treatment['velocity_variables'] = ['u', 'v', 'w']
treatment['output'] = os.path.join('OUTPUT', 'rotating_monopole')
treatment.execute()
```

Post-Treatment example: The signal pressure for a single observer

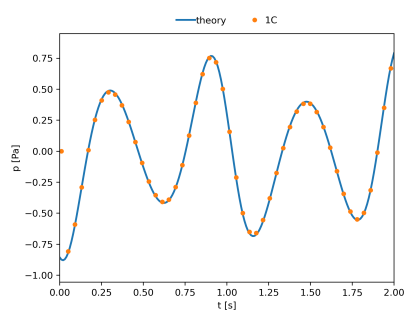
We can compare the analytical solution with the solution obtained with the treatment using the following script:

```
import antares
import os
import matplotlib.pyplot as plt
import numpy as np


## Read theoretical pressure and time vectors
theoretical_filename = os.path.join('..', 'data', 'FWH', 'Rotating_monopole',
↪'analytical_pressure.dat')
theoretical_reader = antares.Reader('column')
theoretical_reader['filename'] = theoretical_filename
theoretical_base = theoretical_reader.read()
theoretical_time = theoretical_base[0][0]['time']
theoretical_p = theoretical_base[0][0]['pressure1']

## Read antares pressure and time vectors
ant_filename = os.path.join('OUTPUT', 'rotating_monopole.h5')
ant_reader = antares.Reader("hdf_antares")
ant_reader['filename'] = ant_filename
ant_base = ant_reader.read()
ant_time = ant_base[0][0]['time']
ant_pressure = ant_base['obs_1'][0]['pressure']

# plt.figure(figsize=(6.2,4.65))
```

(continues on next page)

(continued from previous page)

```
## Plot 1C results
plt.plot(theoretical_time, theoretical_p, linewidth=2, label='theory')
plt.plot(ant_time, ant_pressure, marker='o', linestyle='',
         markevery=4, label='1C', markersize=4)

## Add labels, legends and other configurations
plt.xlim([0, np.max(theoretical_time)])
plt.ylim([1.2*np.min(theoretical_p), 1.2*np.max(theoretical_p)])
plt.xlabel('t [s]')
plt.ylabel('p [Pa]')
plt.legend(ncol=3,  loc='lower center', bbox_to_anchor=(0.5, 1.0),
           handletextpad=0.1, labelspacing=0, frameon=False)

## Save plot
output = os.path.join('OUTPUT', 'plot.png')
plt.savefig(output, bbox_inches='tight', dpi=300)
plt.close()
```



FAQ

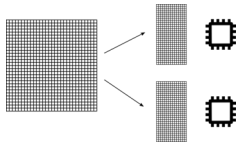
Datafile vs base keywords

The input database can be specified by using either the **base** keywords or **datafile** (with the optional **meshfile**) keyword. The latter only works when the base is written such as every instant is stored in a different file (i.e. using the <instant> tag). In this case, each instant file is read independently and not as part of a whole database. This is a more efficient reading strategy as only 2-3 instants are only ever fully loaded into memory. This could be more efficient than feeding the whole database through the **base** keywords as, depending on each reader, the lazy loading of variables or the shared mesh feature might not be available.

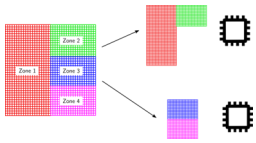
How does the treatment split the database when using MPI?

The splitting of the input database in the different processors of an MPI run depends on the number of zones present in the database:

1. If the input database contains only one zone, the database is split by distributing as equally as possible the **cells** into the total number of processors. This ensures that workload is well balanced between all processors. It also ensure that we can use as many processors as cell there are in the database.



2. If the input database contains 2 or more zones, the database is split by distributing as equally as possible the **zones** into the total number of processors. This does not ensure that the workload is well balanced between all processors; as different zones could contain different numbers of cells. This also limits the number of active processors the treatment will use, because it will never be greater than the number of zones. If you have a small number of zones and you want to use a large number of processors, a merge treatment must be performed before using the FWH treatment.



In which order are the equation variables computed?

The order in which the equations for the physical variables are computed is the following:

1. Pressure.
2. Density.
3. Velocity in x direction.
4. Velocity in y direction.
5. Velocity in z direction.

References

Casalino, D. (2003). An advanced time approach for acoustic analogy predictions. Journal of Sound and Vibration, 261(4), 583-612. (link[123]).

Shur, M. L., Spalart, P. R., & Strelets, M. K. (2005). Noise prediction for increasingly complex jets. International journal of aeroacoustics, 4(3), 213-245. (part 1[124], part 2[125])

Farassat, F. (2007). Derivation of Formulations 1 and 1A of Farassat. Nasa/TM-2007-214853, p. 1–25.(link[126]).

[123] https://doi.org/10.1016/S0022-460X(02)00986-0
[124] https://doi.org/10.1260/1475472054771376
[125] https://doi.org/10.1260/1475472054771385
[126] https://ntrs.nasa.gov/citations/20070010579

Morfey, C. L., & Wright, M. C. M. (2007). Extensions of Lighthill's acoustic analogy with application to computational aeroacoustics. Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences, 463(2085), 2101-2127. (link[127])

Spalart, P. R., & Shur, M. L. (2009). Variants of the Ffowcs Williams-Hawkings equation and their coupling with simulations of hot jets. International journal of aeroacoustics, 8(5), 477-491. (link[128]).

Najafi-Yazdi, A., Brès, G. A., & Mongeau, L. (2011). An acoustic analogy formulation for moving sources in uniformly moving media. Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences, 467(2125), 144-165. (link[129]).

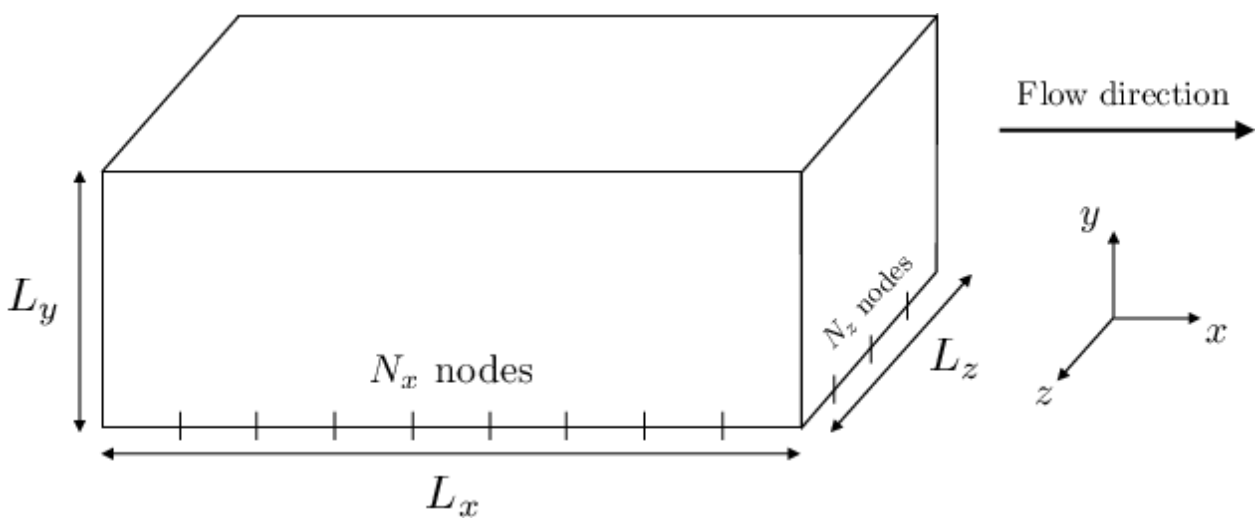
Rahier, G., Huet, M., & Prieur, J. (2015). Additional terms for the use of Ffowcs Williams and Hawkings surface integrals in turbulent flows. Computers & Fluids, 120, 158-172. (link[130])

Ikeda, T., Enomoto, S., Yamamoto, K., & Amemiya, K. (2017). Quadrupole corrections for the permeable-surface Ffowcs Williams–Hawkings equation. AIAA Journal, 55(7), 2307-2320. (link[131])

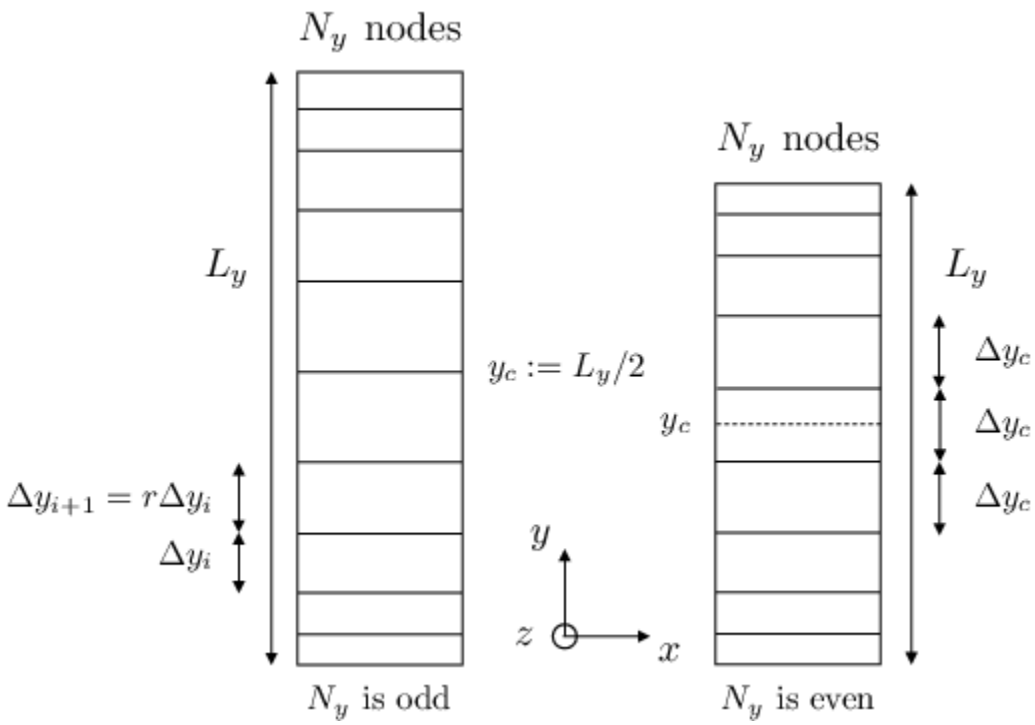
Bi-Periodic Plane Channel Initialization

Description

Create the mesh and the initial condition (white noise or spanwise vortices) for a plane channel flow.



[127] https://doi.org/10.1098/rspa.2007.1864
[128] https://doi.org/10.1260/147547209788549280
[129] https://doi.org/10.1098/rspa.2010.0172
[130] https://dx.doi.org/10.1016/j.compfluid.2015.07.014
[131] https://dx.doi.org/10.2514/1.J055328



Construction

```
import antares
myt = antares.Treatment('initchannel')
```

Parameters

- **domain_size:** ***list(int)***
  Size of the channel in longitudinal (**Lx**), normal (**Ly**), and transverse (**Lz**) directions.
- **mesh_node_nb:** ***list(int)***
  Number of nodes used to create the mesh in the longitudinal (**Nx**), normal (**Ny**), and transverse (**Nz**) directions
- **stretching_law:** ***str*****, default=** ***'uniform'***
  The stretching law used in the wall-normal direction. 4 laws are available: 'uniform' (default), 'geometric', 'tanh1', and 'tanh2'.
- **ratio:** ***float*****, default=** ***1.***
  The expansion ratio (**r**) of the cells size in the wall-normal direction in case of non-uniform stretching.
- **perturbation_amplitude:** ***list(float)*****, default=** ***[0.05, 0.]***
  White noise amplitude in the longitudinal direction and vortex amplitude. White noise amplitude in the other directions is equal to 2 times this value.
- **bulk_values:** ***list(float)*****, default=** ***[0.05, 0.]***
  Bulk velocity in the longitudinal direction and bulk density.
- **t_ref:** ***float*****,**
  Temperature of reference.

- **cv:** ***float*****,**
  Specific heat coefficient at constant volume.

Preconditions

Postconditions

The returned base contains one structured zone with one instant with default names. It contains the mesh at nodes and conservative variables (ro, rou, rov, row, roE) at cell centers. The following attributes are stored in the base: Distance from the wall of the first row nodes **y_1** = Δy_1, uniform grid spacings Δx and Δz, and grid spacing at the channel center Δy_c.

The mesh can be uniform or stretched in the wall-normal direction thanks to the ratio parameter **r** (1<r<1.05). According to the parity of nodes number **Ny** in the wall-normal direction, the center of the cell **y_c** is either on a cell (even case) or on a node (odd case). In the even case, Δy_1 is computed such that Δy_c is equal to its neighboring values. So given **N_y** and **r**:

- the first hyperbolic tangent law is defined by: $\forall i \leq N_y - 1, y_i = \dfrac{1}{r}\tanh(\xi_i atanh(r)) + 1$ with $\xi_i = -1 + 2\frac{i}{N_y - 1}$ and $0 < r < 1$.

This law has been used by *Moin* (page **??**) and *Abe* (page **??**).

- the second hyperbolic tangent law is defined by: $\forall i \leq N_y - 1, y_i = 1 - \dfrac{\tanh(r\xi_i)}{\tanh(r)}$ with $\xi_i = 1 - 2\frac{i}{N_y - 1}$.

It has been used by *Gullbrand* (page **??**).

- the geometric stretching is defined by: $\dfrac{\Delta y_{i+1}}{\Delta y_i} = r$

It is advised to choose $r \in [1; 1.05]$. According to the parity of nodes number **Ny** in the wall-normal direction, the center of the cell **y_c** is either on a cell (even case) or on a node (odd case). In the even case, Δy_1 is computed such that Δy_c is equal to its neighboring values. So given **N_y** and **r**:

$\Delta y_1 = \dfrac{L_y}{2}\dfrac{1-r}{1-r^{\frac{N_y-1}{2}}}$ if **Ny** is odd

$\Delta y_1 = \dfrac{L_y}{2}\dfrac{1-r}{1-r^{N_y/2-1}}\dfrac{1}{1+\frac{(1-r)r^{N_y/2-2}}{2(1-r^{N_y/2-1})}}$ if **Ny** is even

Of course, $\Delta y = L_y/(N_y - 1)$ if **r** = 1.

The initial condition is based on a power law for the longitudinal velocity $u(y) = \frac{8}{7}u_b\left(1 - |1 - \frac{y}{h}|\right)^{1/7}$ with u_b the bulk velocity.

The density ρ is chosen uniform, and equals to the bulk density ρ_b.

To ease the transition to turbulence, one can add:

- a white noise which maximum level in the streamwise direction equal to twice those in the wall-normal and spanwise directions
- or/and vortex rings as used by *LeBras* (page **??**).

It is not recommended to mix them. A white noise is enough on coarse mesh while vortex rings are well-suited to fine grid. The initial temperature should be chosen to the wall temperature to avoid entropy waves.

Perturbations are added to the power law velocity:

$u_{\text{pert}} = \alpha u_b \frac{y-y_0}{b}\exp\left(-\frac{a^2\ln 2}{b^2}\right)\left(1 + 0.5|\sin(\frac{4\pi z}{L_z})|\right)$

$v_{\text{pert}} = -\alpha u_b \frac{x-x_0}{b} \exp\left(-\frac{a^2 \ln 2}{b^2}\right)\left(1 + 0.5|\sin(\frac{4\pi z}{L_z})|\right)$

with:

- x_0 and $y_0 = 0.3L_y/2$ the coordinates of the center vortex
- α a constant which represents the amplitude and can be set with the *perturbation_amplitude* parameter (0.6 advised, and 0. by default)
- $a = \sqrt{(x-x_0)^2 + (y-y_0)^2}$
- $b = 4\Delta_x$

Vortex rings are spaced $20\Delta_x$ in the longitudinal direction.

Main functions

class antares.treatment.init.TreatmentInitChannel.**TreatmentInitChannel**

execute()

Create the mesh and the initial condition for a plane channel flow.

Returns

base of the mesh (at nodes and cells) + flow initialization (at cells)

Return type

Base

Example

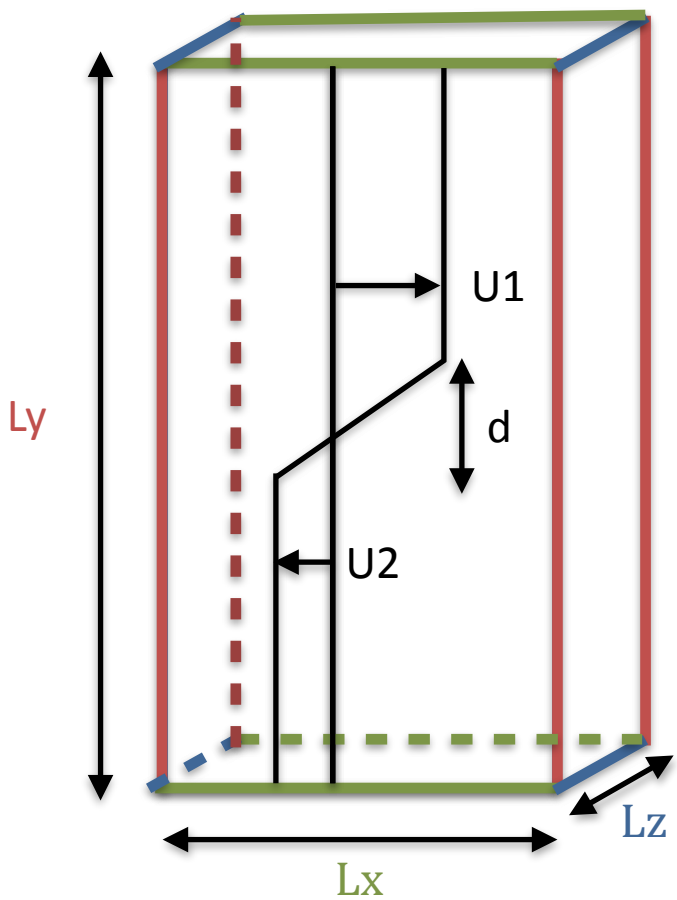
```
import math
import os

import antares

if not os.path.isdir('OUTPUT'):
    os.makedirs('OUTPUT')

gam = 1.4
mach = 0.2

t = antares.Treatment('initchannel')
t['domain_size'] = [2*math.pi, 2., math.pi]  # [Lx,Ly,Lz]
t['mesh_node_nb'] = [49, 41, 41]  # [Nx,Ny,Nz]
t['ratio'] = 1.0  # stretch ratio in wall-normal direction
t['stretching_law'] = 'uniform'
# Flow init parameter
t['bulk_values'] = [1., 1.]  # bulk velocity and bulk density
t['perturbation_amplitude'] = [0., 0.6]  # [white noise=0.05,vortex rings=0.] by default
t['t_ref'] = 1.0
t['cv'] = 1./(gam*(gam-1)*mach**2)

b = t.execute()

# w = ant.Writer()
```

(continues on next page)

(continued from previous page)

```
# w['filename'] = 'mesh'
# w['file_format'] = 'fmt_tp'
# w['base'] = b[:,:,('x','y','z')]
# w.dump()

w = antares.Writer('fmt_tp')
w['filename'] = os.path.join('OUTPUT', 'ex_flow_init_channel.dat')
w['base'] = b[:, :, (('x', 'cell'), ('y', 'cell'), ('z', 'cell'),
                     'ro', 'rovx', 'rovy', 'rovz', 'roE')]
w.dump()
```

Plane Shear Layer Initialization



The following treatment creates the mesh and the initial condition for a plane shear layer.

Parameters

- **domain_size:** *list(float)*
  Size of the box in longitudinal (**Lx**), normal (**Ly**), and transverse (**Lz**) directions.
- **mesh_node_nb:** *list(int)*
  Number of nodes used to create the mesh in the longitudinal (**Nx**), normal (**Ny**), and transverse (**Nz**) directions.
- **base_flow_velocities:** *list(float)*
  Base-flow velocities up and down the shear-layer.
- **shear_layer_thickness:** *float*
  Thickness of the shear layer to meet top to down base-flow velocities.
- **perturbation_amplitude:** *float*
  Amplitude of the perturbation related to the exponential term.
- **inf_state_ref:** *list(float)*
  Conservative variables at infinity.

Preconditions

Perfect Gas Assumption: $\gamma = 1.4$.

Main functions

class antares.treatment.init.TreatmentInitShearLayer.**TreatmentInitShearLayer**

execute()

Compute an initial field with a shear.

The returned base contains one structured zone with one instant with default names. It contains a structured mesh and conservative variables (ro, rou, rov, row, roE) at nodes.

The flow field is composed of a base flow based on a uniform velocity U_1 for $y > \frac{d}{2}$ and U_2 for $y < -\frac{d}{2}$.

In the shear layer domain ($-\frac{d}{2} < y < \frac{d}{2}$), the base flow profile is a linear profile that ensures velocity continuity between the two uniform domains.

$U(y) = U_1$ for $y \geq \frac{d}{2}$

$U(y) = \frac{U_1 - U_2}{d} y + \frac{U_1 - U_2}{2} y$ for $-\frac{d}{2} < y < \frac{d}{2}$

$U(y) = U_2$ for $y \leq \frac{d}{2}$

A perturbation is added to the base flow:

$u_1 = \alpha B \exp(-\alpha y) \sin(\alpha x), v_1 = \alpha B \exp(-\alpha y) \cos(\alpha x)$ if $y \geq 0$

$u_2 = \alpha B \exp(-\alpha y) \cos(\alpha x), v_2 = \alpha B \exp(-\alpha y) \sin(\alpha x)$ if $y < 0$

with $\alpha = \frac{2\pi}{\lambda}$, λ being the perturbation wavelength.

Example

```
import os
if not os.path.isdir('OUTPUT'):
    os.makedirs('OUTPUT')

import antares

t = antares.Treatment('InitShearLayer')
t['domain_size'] = [1.0, 1.0, 1.0]       # [Lx,Ly,Lz]
t['mesh_node_nb'] = [20, 20, 20]         # [Nx,Ny,Nz]
t['base_flow_velocities'] = [10.0, 6.0]  # U1 et U2 velocities
t['shear_layer_thickness'] = 0.1         # shear layer thickness (d)
t['perturbation_amplitude'] = 0.0005     # Coefficient B
t['inf_state_ref'] = [1.16, 0.0, 0.0, 0.0, 249864.58]
b = t.execute()

print(b[0][0])
w = antares.Writer('hdf_antares')
w['base'] = b
w['filename'] = os.path.join('OUTPUT', 'ex_test')
w.dump()
```

Harmonic Balance computations

Treatments

Several specific treatments for Harmonic Balance computations are available such as:

HB Discrete Fourier Transform

Computes the Discrete Fourier Transform of a HB/TSM computation

Parameters

- **base: `Base`**
  The Base that will be DFT computed.
- **coordinates: *list(str)*, default= `antares.Base.coordinate_names`**
  The variable names that define the set of coordinates. The coordinates will not be computed by the DFT treatment.
- **variables_to_ignore: *list(str)*, default= *['time', 'iteration']***
  Variables that won't be DFT computed, these are not the coordinates but can be for instance the iteration vector.
- **hb_computation: *:class:`HbComputation* or in_attr, default= *'in_attr'***
  The object that defines the attributes of the current HB/TSM computation.
- **type: *str*, default= *'mod/phi'***
  The DFT type of the output data: *'mod/phi'* for modulus/phase decomposition or *'im/re'* for imaginery/real part decomposition. The phase is expressed in degrees.

- **mode:** ***list(int)*** **or in_attr, default=** ***None***
  If you want to extract only some harmonics, you can put here the harmonic (i.e. 1 for example) or a list of harmonics ([1, 2, 4] for example). If empty, this return all the harmonics including the mean part.

Initialization

To initialize a DFT object:

```
>>> treatment = Treatment('hbdft')
```

Main functions

class antares.hb.TreatmentHbdft.**TreatmentHbdft**

execute()

Execute the treatment.

Returns
the base containing the results

Return type
Base

Example

```
"""
This example illustrates the Discrete Fourier Transform
treatment on a single frequency (TSM) computation.
As the HB/TSM methods are spectral ones, this DFT is exact
"""
import os
if not os.path.isdir('OUTPUT'):
    os.makedirs('OUTPUT')

from antares import HbComputation, Reader, Treatment, Writer

# ------------------
# Reading the files
# ------------------
reader = Reader('bin_tp')
reader['filename'] = os.path.join('..', 'data', 'HARMONIC_BALANCE', 'flow_<zone>.dat')
reader['n_hbt'] = 1
ini_base = reader.read()

# -------------------------------
# Create an HbComputation object
# -------------------------------
hb_comp = HbComputation()
hb_comp['frequencies'] = [6.2344674e-04]
ini_base.attrs['hb_computation'] = hb_comp
```

(continues on next page)

(continued from previous page)

```
# ----
# DFT
# ----
treatment = Treatment('hbdft')
treatment['base'] = ini_base
treatment['type'] = 'mod/phi'
treatment['mode'] = (0, 1)
result = treatment.execute()

# -------------------
# Writing the result
# -------------------
writer = Writer('bin_tp')
writer['filename'] = os.path.join('OUTPUT', 'ex_hbdft.plt')
writer['base'] = result
writer.dump()
```

HB Temporal interpolation

Temporal interpolation for a HB/TSM computation.

Parameters

- **base: `Base`**
  The input base that will be temporally interpolated.
- **hb_computation: *:class:`HbComputation* or in_attr, default= *'in_attr'***
  The object that defines the attributes of the current HB/TSM computation.
- **time: *float or list(numpy.ndarray)* or in_attr, default= *0.***
  Time instant at which solution is sought.
- **coordinates: *list(str)***
  The variable names that define the set of coordinates used for the interpolation. It is assumed that they are cartesian coordinates.

Initialization

To initialize a temporal interpolation object:

```
>>> treatment = Treatment('hbinterp')
```

Main functions

class antares.hb.TreatmentHbinterp.**TreatmentHbinterp**

execute()

Execute the treatment.

Returns
the base containing the results

Return type
Base

Example

```
"""
This example illustrates the temporal interpolation
treatment on a single frequency (TSM) computation.
"""
import os
if not os.path.isdir('OUTPUT'):
    os.makedirs('OUTPUT')

from antares import HbComputation, Reader, Treatment, Writer

# ------------------
# Reading the files
# ------------------
reader = Reader('bin_tp')
reader['filename'] = os.path.join('..', 'data', 'HARMONIC_BALANCE', 'flow_<zone>.dat')
reader['n_hbt'] = 1
ini_base = reader.read()

# -------------------------------
# Create an HbComputation object
# -------------------------------
hb_comp = HbComputation()
hb_comp['frequencies'] = [6.2344674e-04]
ini_base.attrs['hb_computation'] = hb_comp

# -------------------
# HbInterpolation
# -------------------
treatment = Treatment('hbinterp')
treatment['base'] = ini_base
treatment['time'] = 17.14957
result = treatment.execute()

# -------------------
# Writing the result
# -------------------
writer = Writer('bin_tp')
writer['filename'] = os.path.join('OUTPUT', 'ex_hbinterp.plt')
```

(continues on next page)

(continued from previous page)

```
writer['base'] = result
writer.dump()
```

HB Chorochronic duplication

Computes the chorochronic duplication of a HB/TSM computation.

Parameters

- **base: `Base`**
  The input base that will be chorochronically duplicated.
- **coordinates: *list(str)***
  The variable names that define the set of coordinates used for the duplication. It is assumed that they are cartesian coordinates.
- **vectors: *list(tuple(str))*, default= *[]***
  If the base contains vectors, they must be rotated, so put them here. It is assumed that they are expressed in the cartesian coordinate system.
- **hb_computation: *:class:`HbComputation* or in_attr, default= *'in_attr'***
  The object that defines the attributes of the current HB/TSM computation.
- **nb_duplication: *int* or in_attr, default= *'in_attr'***
  The number of duplications.
- **pitch: *int or float* or in_attr, default= *'in_attr'***
  The pitch of the current row.
- **omega: *int or float* or in_attr, default= *0.***
  Rotation speed expressed in radians per second so that the mesh can be rotated accordingly.
- **time: *float*, default= *0.***
  Time instant at which solution is sought.

Initialization

To initialize a Chorochronic duplication object:

```
>>> treatment = Treatment('hbchoro')
```

Main functions

class antares.hb.TreatmentHbchoro.**TreatmentHbchoro**

execute()

Execute the treatment.

Returns
the base containing the results

Return type
`Base`

Example

```
"""
This example illustrates the chorochronic duplication
treatment on a single frequency (TSM) computation
"""
import os
if not os.path.isdir('OUTPUT'):
    os.makedirs('OUTPUT')

import numpy as np

from antares import HbComputation, Reader, Treatment, Writer

# ------------------
# Reading the files
# ------------------
reader = Reader('bin_tp')
reader['filename'] = os.path.join('..', 'data', 'HARMONIC_BALANCE', 'flow_<zone>.dat')
reader['n_hbt'] = 1
ini_base = reader.read()

# -------------------------------
# Create an HbComputation object
# -------------------------------
# AEL configuration with IBPA = 8 and nb_blade = 20
hb_comp = HbComputation()
hb_comp['frequencies'] = [6.2344674e-04]
hb_comp['phaselag'] = [2 * np.pi * 8. / 20.]
ini_base.attrs['hb_computation'] = hb_comp

# ------------
# Duplication
# ------------
treatment = Treatment('hbchoro')
treatment['base'] = ini_base
treatment['vectors'] = [('rovx', 'rovy', 'rovz')]
treatment['nb_duplication'] = 20
treatment['pitch'] = 2. * np.pi / 20.
result = treatment.execute()

# -------------------
# Writing the result
# -------------------
writer = Writer('bin_tp')
writer['filename'] = os.path.join('OUTPUT', 'ex_hbchoro.plt')
writer['base'] = result
writer.dump()
```

Specific function

antares.**prepare4tsm**(*nharm*, *list_omega*, *list_nbblade*)

Initialize the HbComputations for a TSM computation with two rows.

Parameters

- **nharm** – the number of harmonics of the computation
- **list_omega** – rotation speed of both rows expressed in radians per second
- **list_nbblade** – the number of blades of both rows

Returns
the two HbComputations

HbComputation object

Defines an Almost-Periodic Computation.

The HbComputation object can ease setting up the Harmonic Balance computations.

Parameters

- **frequencies: numpy.ndarray**
  List of frequencies considered. For TSM, put also the harmonics, not only the fundamental frequency.
- **timelevels: numpy.ndarray**
  List of timelevels. Default is evenly spaced timelevels on the smallest frequency.
- **phaselag: numpy.ndarray**
  List of phaselags associated to each frequency.

Main functions

class antares.**HbComputation**

Define an Almost-Periodic Computation.

The IDFT and DFT Almost-Periodic Matrix can be computed. All the definitions are based on the following article[132]

ap_dft_matrix(*frequencies=None*, *timelevels=None*)

Compute the Almost-Periodic DFT matrix

ap_idft_matrix(*frequencies=None*, *timelevels=None*)

Compute the Almost-Periodic IDFT matrix

ap_source_term(*frequencies=None*, *timelevels=None*)

Compute the Almost-Periodic source term which is to $D_t[\cdot] = iA^{-1}PA$, where A denotes the DFT matrix, A^{-1} the IDFT matrix and $P = diag(-\omega_N, \cdots, \omega_0, \cdots, \omega_N)$

conditionning()

Returns the condition number of the almost periodic IDFT matrix

get_evenly_spaced(*base_frequency=None*)

Set the timelevels vector as evenly spaced over the base frequency period.

optimize_timelevels(*target=0.0*)

Optimization of the timelevels using gradient-based algorithm. See HbAlgoOPT for more infos.

p_source_term(*frequencies=None*, *timelevels=None*)

Compute the analytical mono-frequential source term

Note: Some treatments can use `attrs` from antares API instead of input keys. If so, the argument *can use* `in_attr` is set to 'yes' in the key documentation.

3.5 Antares Helper Functions and Utilities

3.5.1 Progress Bar

A progress bar is available in some functions, methods, and treatments. It relies on the package tqdm[133]. If this package is not available, then you recover the old Antares 1.8.2 progress bar.

antares.**set_progress_bar**(*value=False*)

Set the progress bar status (on/off).

Parameters

value (*bool*) – status of the progress bar

antares.**disable_progress_bar**()

Disable the progress bar status.

[132] https://dx.doi.org/10.1016/j.jcp.2012.11.010
[133] https://github.com/tqdm/tqdm

CHAPTER

FOUR

TUTORIALS

- Overview of Antares basic classes: Base, Zone Instant
- Slicing: how to extract sub-parts of a Base data structure
- Readers: how to read data from files
- Writers: how to write data in files
- Family: gather objects in a set
- Equation: how to compute variables with equations

4.1 Equation Management

By default, each instant owns a **computer**. A computer is used to compute variables thanks to python user-defined functions. See *Equation Manager* (page 374) for technical details.

If you want to use the computing system in an Antares python script, you may choose the **modeling** of the computer.

```
base.set_computer_model('internal')
```

internal is a very basic modeling that is provided by Antares.

If you do not want to use any available modeling, then you can set your own temporary formula.

```
base.set_formula('A = B * C')
```

Then, you can compute the variable you just have defined. By default, the variable is stored in the computer. Then, if you asked to compute the variable again, it will give you back the stored value.

```
base.compute('A')
```

You may want to reset the value of the variable.

```
base.set_formula('A = B * D')
base.compute('A', reset=True)
```

You could also not have stored the value of the variable at the first computation.

```
base.set_formula('A = B * C')
base.compute('A', store=False)
```

All the previous methods are also available for zone and instant objects.

4.1.1 Modeling

A modeling is basically a set of equations. The equations may be given a priori in python files as with the modeling **internal** or given in a script with the function `antares.Base.set_formula()`.

The equations may use the following operators:

```
'**', '/', '*', '+', '-', ',', '>', '<',
'abs(', 'angle(', 'arccos(', 'arcsin(', 'unwrap(',
'arctan2(', 'arctan(', 'conj(', 'cosh(',
'cos(', 'exp(', 'imag(', 'log(', 'log10(', 'max(', 'min(', 'mean(',
'maximum(', 'minimum(', 'ones(', 'real(', 'sign(',
'sinh(', 'sin(', 'tanh(', 'tan(', 'zeros(',
'(', ')']
```

Antares internal modeling

antares.eqmanager.formula.internal.equations.**Cp**(*gamma*, *Rgaz*)

antares.eqmanager.formula.internal.equations.**Cv**(*gamma*, *Rgaz*)

antares.eqmanager.formula.internal.equations.**E**(*rhoE*, *rho*)

antares.eqmanager.formula.internal.equations.**Ec**(*u*, *v*, *w*)

antares.eqmanager.formula.internal.equations.**Pi**(*psta*, *mach*, *gamma*)

antares.eqmanager.formula.internal.equations.**R**(*y*, *z*)

antares.eqmanager.formula.internal.equations.**Rgaz**()

antares.eqmanager.formula.internal.equations.**Ti**(*tsta*, *mach*, *gamma*)

antares.eqmanager.formula.internal.equations.**bunch**(*rho*, *E*, *Ec*, *gamma*, *Rgaz*)

antares.eqmanager.formula.internal.equations.**c**(*rho*, *psta*, *gamma*)

antares.eqmanager.formula.internal.equations.**cons_abs**(*x*, *y*, *z*, *rho*, *rhou*, *rhov*, *rhow*, *rhoE*, *omega*, *gamma*)

antares.eqmanager.formula.internal.equations.**entropy**(*psta*, *tsta*, *Rgaz*, *Cp*, *Pi0*, *Ti0*)

antares.eqmanager.formula.internal.equations.**gamma**()

antares.eqmanager.formula.internal.equations.**hi**(*Cp*, *Ti*)

antares.eqmanager.formula.internal.equations.**mach**(*Ec*, *c*)

antares.eqmanager.formula.internal.equations.**pi**()

antares.eqmanager.formula.internal.equations.**psta**(*rho*, *E*, *Ec*, *gamma*)

antares.eqmanager.formula.internal.equations.**theta**(*y*, *z*)

antares.eqmanager.formula.internal.equations.**tsta**(*E*, *Ec*, *gamma*, *Rgaz*)

antares.eqmanager.formula.internal.equations.**u**(*rhou*, *rho*)

antares.eqmanager.formula.internal.equations.**v**(*rhov*, *rho*)

antares.eqmanager.formula.internal.equations.**w**(*rhow*, *rho*)

Antares constant_gamma modeling

Ce calculateur gamma constant suppose que les données d'entrée sont exprimées dans le repère absolu. Les données d'entrée sont les variables conservatives (Density, MomentumX, MomentumY, MomentumZ, EnergyStagnationDensity), les coordonnées dans le repère absolu (CoordinateX,CoordinateY,CoordinateZ), la vitesse de rotation (omega), le rapport des chaleurs spécifiques (gamma), la constante spécifique du gaz (Rgas), la pression totale relative isentropique (Ptris).

Le repère direct (CoordinateX,CoordinateY,CoordinateZ) suit une convention spéciale. L'axe CoordinateX est l'axe de rotation.

antares.eqmanager.formula.constant_gamma.equations.**CP**(*gamma*, *R_melange*)

Specific heat at constant pressure.

antares.eqmanager.formula.constant_gamma.equations.**Cv**(*gamma*, *R_melange*)

Specific heat at constant volume.

antares.eqmanager.formula.constant_gamma.equations.**Dynalpy_n**(*Density*, *Vn*, *Pressure*)

return Dynalpy.

antares.eqmanager.formula.constant_gamma.equations.**E_r**(*E_t*, *V*, *W*)

Relative stagnation (total) energy per unit mass.

antares.eqmanager.formula.constant_gamma.equations.**E_t**(*EnergyStagnationDensity*, *Density*)

antares.eqmanager.formula.constant_gamma.equations.**Hta**(*hs*, *VelocityX*, *VelocityY*, *VelocityZ*)

antares.eqmanager.formula.constant_gamma.equations.**Htr**(*hs*, *W*)

antares.eqmanager.formula.constant_gamma.equations.**KinematicViscosity**(*Viscosity*, *Density*)

antares.eqmanager.formula.constant_gamma.equations.**Mis**(*Ptris*, *Pressure*, *gamma*)

Isentropic Mach number.

antares.eqmanager.formula.constant_gamma.equations.**Pressure**(*Density*, *R_melange*, *Temperature*)

antares.eqmanager.formula.constant_gamma.equations.**Pta**(*Pressure*, *Tta*, *Temperature*, *CP*, *R_melange*)

Compute absolute total pressure.

antares.eqmanager.formula.constant_gamma.equations.**Ptr**(*Pressure*, *Ttr*, *Temperature*, *CP*, *R_melange*)

Compute relative total pressure.

antares.eqmanager.formula.constant_gamma.equations.**R**(*CoordinateZ*, *CoordinateY*)

Cylindrical coordinate r from (x, r, theta).

Radial direction.

antares.eqmanager.formula.constant_gamma.equations.**R_melange**()

antares.eqmanager.formula.constant_gamma.equations.**Temperature**(*e_int*, *R_melange*, *CP*)

`antares.eqmanager.formula.constant_gamma.equations.`**`Theta`**(*CoordinateY*, *CoordinateZ*)

Cylindrical coordinate theta from (x, r, theta).

Azimuthal direction.

`antares.eqmanager.formula.constant_gamma.equations.`**`Tta`**(*Hta*, *CP*)

`antares.eqmanager.formula.constant_gamma.equations.`**`Ttr`**(*Htr*, *CP*)

`antares.eqmanager.formula.constant_gamma.equations.`**`V`**(*VelocityX*, *VelocityY*, *VelocityZ*)

Velocity magnitude.

vector V = Vx e_x + Vy e_y + Vz e_z magnitude V = sqrt(V.V)

`antares.eqmanager.formula.constant_gamma.equations.`**`VelocityX`**(*MomentumX*, *Density*)

Velocity component in the x-direction.

Vx = V.e_x

`antares.eqmanager.formula.constant_gamma.equations.`**`VelocityY`**(*MomentumY*, *Density*)

Velocity component in the y-direction.

Vy = V.e_y

`antares.eqmanager.formula.constant_gamma.equations.`**`VelocityZ`**(*MomentumZ*, *Density*)

Velocity component in the z-direction.

Vz = V.e_z

`antares.eqmanager.formula.constant_gamma.equations.`**`Vn`**(*VelocityX*, *Vr*, *incl*)

Velocity component in the normal direction.

Vn = V.n

`antares.eqmanager.formula.constant_gamma.equations.`**`Vr`**(*Theta*, *VelocityY*, *VelocityZ*)

Velocity component in the radial r-direction.

Vr = V.e_r

`antares.eqmanager.formula.constant_gamma.equations.`**`Vt`**(*Theta*, *VelocityY*, *VelocityZ*)

Velocity component in the azimuthal theta-direction.

Vt = V.e_theta

`antares.eqmanager.formula.constant_gamma.equations.`**`Vt2`**(*VelocityX*, *Vr*, *incl*)

`antares.eqmanager.formula.constant_gamma.equations.`**`W`**(*Wx*, *Wr*, *Wt*)

`antares.eqmanager.formula.constant_gamma.equations.`**`Wr`**(*Vr*)

`antares.eqmanager.formula.constant_gamma.equations.`**`Wt`**(*omega*, *R*, *Vt*)

`antares.eqmanager.formula.constant_gamma.equations.`**`Wx`**(*VelocityX*)

`antares.eqmanager.formula.constant_gamma.equations.`**`Wy`**(*Wt*, *Vr*, *Theta*)

`antares.eqmanager.formula.constant_gamma.equations.`**`Wz`**(*Wt*, *Vr*, *Theta*)

`antares.eqmanager.formula.constant_gamma.equations.`**`e_int`**(*VelocityX*, *VelocityY*, *VelocityZ*, *E_t*)

Static internal energy per unit mass.

`antares.eqmanager.formula.constant_gamma.equations.`**`gamma`**()

antares.eqmanager.formula.constant_gamma.equations.**hs**(*Temperature*, *CP*)

antares.eqmanager.formula.constant_gamma.equations.**incl**(*nr*, *nx*)

antares.eqmanager.formula.constant_gamma.equations.**nr**(*Theta*, *ny*, *nz*)

antares.eqmanager.formula.constant_gamma.equations.**nt**(*Theta*, *ny*, *nz*)

antares.eqmanager.formula.constant_gamma.equations.**pi**()

Antares variable_gamma modeling

Ce calculateur gamma variable suppose que les données d'entrées sont exprimées dans le repère absolu.

Les données d'entrée sont les variables conservatives (Density, MomentumX, MomentumY, MomentumZ, EnergyStagnationDensity), les coordonnées dans le repère absolu (CoordinateX,CoordinateY,CoordinateZ), la vitesse de rotation (omega), le rapport des chaleurs spécifiques (gamma), la constante spécifique du gaz (Rgas), la pression totale relative isentropique (Ptris).

Le repère direct (CoordinateX,CoordinateY,CoordinateZ) suit une convention spéciale. L'axe CoordinateX est l'axe de rotation.

antares.eqmanager.formula.variable_gamma.equations.**CP**(*poly_coeff*, *Temperature*)

antares.eqmanager.formula.variable_gamma.equations.**Cv**(*CP*, *R_melange*)

antares.eqmanager.formula.variable_gamma.equations.**Dynalpy_n**(*Density*, *Vn*, *Pressure*)

return Dynalpy

antares.eqmanager.formula.variable_gamma.equations.**E_r**(*E_t*, *V*, *W*)

Relative stagnation (total) energy per unit mass.

antares.eqmanager.formula.variable_gamma.equations.**E_t**(*EnergyStagnationDensity*, *Density*)

antares.eqmanager.formula.variable_gamma.equations.**H0**(*Far*, *War*)

antares.eqmanager.formula.variable_gamma.equations.**Hta**(*hs*, *VelocityX*, *VelocityY*, *VelocityZ*)

antares.eqmanager.formula.variable_gamma.equations.**Htr**(*hs*, *W*)

antares.eqmanager.formula.variable_gamma.equations.**KinematicViscosity**(*Viscosity*, *Density*)

antares.eqmanager.formula.variable_gamma.equations.**Mis**(*Ptris*, *Pressure*, *gamma*)

antares.eqmanager.formula.variable_gamma.equations.**Pressure**(*Density*, *R_melange*, *Temperature*)

antares.eqmanager.formula.variable_gamma.equations.**Pta**(*Pressure*, *Tta*, *Temperature*, *poly_coeff*, *R_melange*)

Compute absolute total pressure.

antares.eqmanager.formula.variable_gamma.equations.**Ptr**(*Pressure*, *Ttr*, *Temperature*, *poly_coeff*, *R_melange*)

Compute relative total pressure.

antares.eqmanager.formula.variable_gamma.equations.**R**(*CoordinateZ*, *CoordinateY*)

Cylindrical coordinate r from (x, r, theta).

Radial direction.

antares.eqmanager.formula.variable_gamma.equations.**R_melange**(*Far*, *War*)

antares.eqmanager.formula.variable_gamma.equations.**Temperature**(*e_int*, *poly_coeff*, *R_melange*, *H0*)

antares.eqmanager.formula.variable_gamma.equations.**Theta**(*CoordinateY*, *CoordinateZ*)

Cylindrical coordinate theta from (x, r, theta).

Azimuthal direction.

antares.eqmanager.formula.variable_gamma.equations.**Tta**(*Hta*, *poly_coeff*, *H0*)

antares.eqmanager.formula.variable_gamma.equations.**Ttr**(*Htr*, *poly_coeff*, *H0*)

antares.eqmanager.formula.variable_gamma.equations.**V**(*VelocityX*, *VelocityY*, *VelocityZ*)

Velocity magnitude.

vector V = Vx e_x + Vy e_y + Vz e_z magnitude V = sqrt(V.V)

antares.eqmanager.formula.variable_gamma.equations.**VelocityX**(*MomentumX*, *Density*)

Velocity component in the x-direction.

Vx = V.e_x

antares.eqmanager.formula.variable_gamma.equations.**VelocityY**(*MomentumY*, *Density*)

Velocity component in the y-direction.

Vy = V.e_y

antares.eqmanager.formula.variable_gamma.equations.**VelocityZ**(*MomentumZ*, *Density*)

Velocity component in the z-direction.

Vz = V.e_z

antares.eqmanager.formula.variable_gamma.equations.**Vn**(*VelocityX*, *Vr*, *incl*)

Velocity component in the normal direction.

Vn = V.n

antares.eqmanager.formula.variable_gamma.equations.**Vr**(*Theta*, *VelocityY*, *VelocityZ*)

Velocity component in the radial r-direction.

Vr = V.e_r

antares.eqmanager.formula.variable_gamma.equations.**Vt**(*Theta*, *VelocityY*, *VelocityZ*)

Velocity component in the azimuthal theta-direction.

Vt = V.e_theta

antares.eqmanager.formula.variable_gamma.equations.**Vt2**(*VelocityX*, *Vr*, *incl*)

antares.eqmanager.formula.variable_gamma.equations.**W**(*Wx*, *Wr*, *Wt*)

antares.eqmanager.formula.variable_gamma.equations.**Wr**(*Vr*)

antares.eqmanager.formula.variable_gamma.equations.**Wt**(*omega*, *R*, *Vt*)

antares.eqmanager.formula.variable_gamma.equations.**Wx**(*VelocityX*)

antares.eqmanager.formula.variable_gamma.equations.**Wy**(*Wt*, *Vr*, *Theta*)

antares.eqmanager.formula.variable_gamma.equations.**Wz**(*Wt*, *Vr*, *Theta*)

antares.eqmanager.formula.variable_gamma.equations.**e_int**(*VelocityX*, *VelocityY*, *VelocityZ*, *E_t*)

Static internal energy per unit mass.

antares.eqmanager.formula.variable_gamma.equations.**gamma**(*CP*, *Cv*)

antares.eqmanager.formula.variable_gamma.equations.**hs**(*Temperature*, *poly_coeff*, *H0*)

antares.eqmanager.formula.variable_gamma.equations.**incl**(*nr*, *nx*)

antares.eqmanager.formula.variable_gamma.equations.**nr**(*Theta*, *ny*, *nz*)

antares.eqmanager.formula.variable_gamma.equations.**nt**(*Theta*, *ny*, *nz*)

antares.eqmanager.formula.variable_gamma.equations.**phi_T**(*poly_coeff*, *Temperature*)

Returns $\phi(T) = \int c_p racdTT$.

antares.eqmanager.formula.variable_gamma.equations.**pi**()

antares.eqmanager.formula.variable_gamma.equations.**poly_coeff**(*Far*, *War*)

Compute polynomial coefficients for heat capacities.

AVBP variable_gamma modeling

Get additional variables from AVBP conservative variables. The species_database.dat file is needed. The formulas are set for 3D solutions.

antares.eqmanager.formula.avbp.equations.**Cp**(*DictMassFractions*, *T*)

Mass Heat capacity of the mixture at constant pressure **[J/K/kg]**.

$C_p(T) = \sum_{k=1}^{N} Y_k C_{p,k}(T)$

Computed with static temperature and AVBP's species_database.dat.

antares.eqmanager.formula.avbp.equations.**Cv**(*DictMassFractions*, *T*)

Mass Heat capacity of the mixture at constant volume **[J/K/kg]**.

$C_v(T) = \sum_{k=1}^{N} Y_k C_{v,k}(T)$

Computed with static temperature and AVBP's species_database.dat.

antares.eqmanager.formula.avbp.equations.**DictMassFractions**(*rho*)

Get the dictionary of the species.

This dictionary is involved in the mixture and their mass fractions.

DictMassFractions is a specific keyword that might be called in functions' arguments. It retrieves the following dictionary: $DictMassFractions = \{'k' : [Y_k], ...\}_{k \in \{Specie1,...\}}$ where SpecieX are the species involved in the mixture.

antares.eqmanager.formula.avbp.equations.**Ec**(*u*, *v*, *w*)

Mass mixture kinetic energy **[J/kg]**.

$E_c = \frac{1}{2}(u^2 + v^2 + w^2)$

antares.eqmanager.formula.avbp.equations.**Eint**(*Etotal*, *Ec*)

Mass internal energy **[J/kg]**.

$E_{int} = E_{total} - E_c$

`antares.eqmanager.formula.avbp.equations.`**`Etotal`**(*rhoE*, *rho*)

Mass mixture total energy **[J/kg]**.

$E_{total} = \rho E_{total}/\rho$

`antares.eqmanager.formula.avbp.equations.`**`Htotal`**(*hs*, *Ec*)

Mixture total mass enthalpy **[J/kg]**.

$H_{total} = \sum_{k=1}^{N} Y_k h_{s,k} + e_c$

`antares.eqmanager.formula.avbp.equations.`**`Htr`**(*hs*, *W*)

Relative total enthalpy (non-chemical).

`antares.eqmanager.formula.avbp.equations.`**`Mis`**(*Ptris*, *P*, *gamma*)

Isentropic mach number **[-]**.

Need a reference total isentropic pressure P_{tris}.

$M_{is} = \sqrt{2\frac{(P_{tris}/P)^{\frac{\gamma-1}{\gamma}}-1}{\gamma-1}}$

`antares.eqmanager.formula.avbp.equations.`**`P`**(*rho*, *T*, *rgas*)

Static pressure **[Pa]**.

$P = \rho r_{gas} T$

`antares.eqmanager.formula.avbp.equations.`**`P_KURT`**(*P*, *P2*, *P3*, *P4*)

Pressure kurtosis **[Pa^4]**.

Requires an averaged AVBP solution with high stat.

`antares.eqmanager.formula.avbp.equations.`**`P_RMS`**(*P*, *P2*)

Pressure root mean square **[Pa]**.

Requires an averaged AVBP solution.

`antares.eqmanager.formula.avbp.equations.`**`P_SKEW`**(*P*, *P2*, *P3*)

Pressure skewness **[Pa^3]**.

Requires an averaged AVBP solution with high stat.

`antares.eqmanager.formula.avbp.equations.`**`Ptotal`**(*DictMassFractions*, *rgas*, *P*, *T*, *Ttotal*)

Absolute total pressure **[Pa]**.

$P_{total} = P_s \exp(\int_{T_s}^{T_{total}} \frac{C_p}{rT} dT)$

`antares.eqmanager.formula.avbp.equations.`**`Ptotal_RMS`**(*Ptotal*, *Ptotal2*)

Total pressure root mean square **[Pa]**.

Requires an averaged AVBP solution.

`antares.eqmanager.formula.avbp.equations.`**`Ptotal_gamma0D`**(*P*, *coeff_gamma0D*, *exp_gamma0D*)

Useful when computing under γ constant assumption.

$Ptotal_{\gamma_{0D}} = P(1 + \frac{\gamma_{0D}-1}{2} M_{\gamma_{0D}}^2)^{\frac{\gamma_{0D}}{\gamma_{0D}-1}}$

`antares.eqmanager.formula.avbp.equations.`**`Ptr`**(*DictMassFractions*, *rgas*, *P*, *T*, *Ttr*)

Relative total pressure.

`antares.eqmanager.formula.avbp.equations.`**`R`**(*z*, *y*)

Compute the radius under the assumption x is the rotation-axis.

`antares.eqmanager.formula.avbp.equations.T`(*DictMassFractions*, *Eint*)

Static temperature **[K]**.

Computed from the internal energy and the AVBP's species_database.dat.

`antares.eqmanager.formula.avbp.equations.T_KURT`(*T*, *T2*, *T3*, *T4*)

Static temperature kurtosis **[K^4]**.

Requires an averaged AVBP solution with high stats.

`antares.eqmanager.formula.avbp.equations.T_RMS`(*T*, *T2*)

Static temperature root mean square **[K]**.

Requires an averaged AVBP solution.

`antares.eqmanager.formula.avbp.equations.T_SKEW`(*T*, *T2*, *T3*)

Static temperature skewness **[K^3]**.

Requires an averaged AVBP solution with high stats.

`antares.eqmanager.formula.avbp.equations.Theta`(*y*, *z*)

Compute the cylindrical angle under the assumption x is the rotation-axis.

`antares.eqmanager.formula.avbp.equations.Ttotal`(*DictMassFractions*, *Htotal*)

Absolute total temperature **[K]**.

Computed with the absolut total enthalpy and the AVBP's species_database.dat.

`antares.eqmanager.formula.avbp.equations.Ttotal_RMS`(*Ttotal*, *Ttotal2*)

Total temperature root mean square **[K]**.

Requires an averaged AVBP solution.

`antares.eqmanager.formula.avbp.equations.Ttotal_gamma0D`(*T*, *coeff_gamma0D*)

Useful when computing under γ constant assumption.

$Ttotal_{\gamma_{0D}} = T(1 + \frac{\gamma_{0D}-1}{2} M^2_{\gamma_{0D}})$

`antares.eqmanager.formula.avbp.equations.Ttr`(*DictMassFractions*, *Htr*)

Relative total temperature.

`antares.eqmanager.formula.avbp.equations.V`(*u*, *v*, *w*)

`antares.eqmanager.formula.avbp.equations.Vm`(*Wm*)

`antares.eqmanager.formula.avbp.equations.Vr`(*R*, *y*, *z*, *v*, *w*)

`antares.eqmanager.formula.avbp.equations.Vt`(*R*, *y*, *z*, *v*, *w*)

`antares.eqmanager.formula.avbp.equations.W`(*Wx*, *Wr*, *Wt*)

`antares.eqmanager.formula.avbp.equations.Wm`(*Wx*, *Wr*)

`antares.eqmanager.formula.avbp.equations.Wr`(*Vr*)

`antares.eqmanager.formula.avbp.equations.Wt`(*omega*, *R*, *Vt*)

`antares.eqmanager.formula.avbp.equations.Wx`(*u*)

`antares.eqmanager.formula.avbp.equations.Wy`(*Wt*, *Vr*, *Theta*)

`antares.eqmanager.formula.avbp.equations.`**`Wz`**(*Wt*, *Vr*, *Theta*)

`antares.eqmanager.formula.avbp.equations.`**`alpha`**(*Vt*, *Vm*)

Sign is consistent with AVBP's angle_alpha.

`antares.eqmanager.formula.avbp.equations.`**`beta`**(*Wt*, *Vm*)

Relative flow angle in degrees.

The sign is consistent with alpha, so it is the opposite of the convention used in elsA computations.

`antares.eqmanager.formula.avbp.equations.`**`c`**(*P*, *rho*, *gamma*)

Sound speed **[m/s]**.

$c = \sqrt{\gamma P/\rho}$

`antares.eqmanager.formula.avbp.equations.`**`c_gamma0D`**(*P*, *rho*, *gamma0D*)

Useful when computing under γ constant assumption.

$c = \sqrt{\gamma_{0D} P/\rho}$

`antares.eqmanager.formula.avbp.equations.`**`coeff_gamma0D`**(*gamma0D*, *mach_gamma0D*)

Useful when computing under γ constant assumption.

$coeff\_gamma0D = 1 + \frac{\gamma_{0D}-1}{2} M^2$

`antares.eqmanager.formula.avbp.equations.`**`exp_gamma0D`**(*gamma0D*)

Useful when computing under γ constant assumption.

$exp\_gamma0D = \frac{\gamma_{0D}}{\gamma_{0D}-1}$

`antares.eqmanager.formula.avbp.equations.`**`gamma`**(*Cp*, *Cv*)

Heat capacity ratio **[-]**.

$\gamma(T) = \frac{C_p(T)}{C_v(T)}$

`antares.eqmanager.formula.avbp.equations.`**`h`**(*DictMassFractions*, *T*)

Mixture enthalpy **[J/kg]**.

Sum of formation and sensible enthalpies $h = \sum_{k=1}^{N} Y_k (h_{s,k} + \Delta h_{f,k}^0)$

`antares.eqmanager.formula.avbp.equations.`**`hs`**(*Eint*, *P*, *rho*)

Mass sensible enthalpy **[J/kg]**.

$h_s = E_{int} + P/\rho$

`antares.eqmanager.formula.avbp.equations.`**`mach`**(*V*, *c*)

Absolute mach number.

`antares.eqmanager.formula.avbp.equations.`**`mach_gamma0D`**(*V*, *c_gamma0D*)

Useful when computing under γ constant assumption.

`antares.eqmanager.formula.avbp.equations.`**`mach_rel`**(*W*, *c*)

Relative mach number.

`antares.eqmanager.formula.avbp.equations.`**`mixture_W`**(*DictMassFractions*)

Mean molecular weight of the mixture W **[kg]**.

$W = \left(\sum_{k=1}^{N} \frac{Y_k}{W_k}\right)^{-1}$ with Y_k and W_k respectively mass fraction and molecular weight of specie k. Computed with AVBP's species_database.dat.

`antares.eqmanager.formula.avbp.equations.`**`mixture_sensible_enthalpy`**(*DictMassFractions*, *T*)

Mixture mass sensible enthalpy **[J/kg]**.

$h_s = \sum_{k=1}^{N} Y_k h_{s,k}$

NOTE: Equal to h_s but computed with species_database.dat

`antares.eqmanager.formula.avbp.equations.`**`phi`**(*u*, *Vr*)

Meridional flow angle in degrees.

`antares.eqmanager.formula.avbp.equations.`**`rgas`**(*mixture_W*)

Mixture specific gas constant **[J/kg/K]**.

$r_{gas} = R/W$

with, respectively R and W associated to the perfect gas constant and the mean mixture molecular weight.

`antares.eqmanager.formula.avbp.equations.`**`s`**(*DictMassFractions*, *T*)

Mass mixture entropy **[J/kg]**.

$s = \sum_{k=1}^{N} Y_k s_k$

`antares.eqmanager.formula.avbp.equations.`**`u`**(*rhou*, *rho*)

`antares.eqmanager.formula.avbp.equations.`**`v`**(*rhov*, *rho*)

`antares.eqmanager.formula.avbp.equations.`**`w`**(*rhow*, *rho*)

4.2 # Reading boundary data in a HDF-CGNS file

This tutorial shows how to:

- read a turbomachinery configuration stored in HDF-CGNS format
- compute the h/H variable
- get the base corresponding to the blade given as a family
- perform node-to-cell and cell-to-node on this latter base
- make profile at some heights of the blade
- output all curves in a single HDF-CGNS file

Reading data

If we have a mesh file *mesh.cgns* and a solution file *elsAoutput.cgns*, then we can use the Antares reader *hdf_cgns*.

```
reader = antares.Reader('hdf_cgns')
reader['filename'] = 'mesh.cgns'
reader['shared'] = True
base = reader.read()
```

We put the mesh as shared variables.

```
reader = antares.Reader('hdf_cgns')
reader['base'] = base
reader['filename'] = 'elsAoutput.cgns'
base = reader.read()
```

We append the solution to the previous base.

Computing h/H variable

The letter *h* means the hub, and the letter *H* the shroud. The h/H variable is the distance of a point from the hub on a line going from the hub to the shroud.

```
tr = antares.Treatment('hH')
tr['base'] = base
tr['row_name'] = 'ROW'
tr['extension'] = 0.1
tr['coordinates'] = ['CoordinateX', 'CoordinateY', 'CoordinateZ']
base = tr.execute()
```

The option 'row_name' tells the convention used to prefix the names of rows in the configuration. The option 'extension' is 10% of the radius of the configuration.

Get the family base

Next, we extract the blade given by the family ROW(1)_BLADE(1)

```
row_1_blade_1 = base[base.families['ROW(1)_BLADE(1)']]
```

Perform node-to-cell and cell-to-node on this latter base

The base row_1_blade_1 is a 2D base on which we can perform node2cell and cell2node operations.

```
row_1_blade_1.node_to_cell(variables=['ro'])

row_1_blade_1.cell_to_node(variables=['Pressure', 'Temperature'])
```

Then, we save the base in a HDF-CGNS file.

```
writer = antares.Writer('hdf_cgns')
writer['base'] = row_1_blade_1
writer['filename'] = 'all_wall_row_1_blade_1.cgns'
writer['base_name'] = 'row_1_blade_1'
writer['dtype'] = 'float32'
writer.dump()
```

Do not forget to remove the following file due to the append writing

```
try:
    os.remove('all_iso.cgns')
except OSError:
    pass
```

Make profile at some heights of the blade ## Output all curves in a single HDF-CGNS file

Then, we loop on 5 heights.

```
for hoH_val in [0.05, 0.25, 0.50, 0.75, 0.95]:
    t = antares.Treatment('isosurface')
    t['base'] = row_1_blade_1
    t['variable'] = 'CoordinateReducedHeight'
    t['value'] = hoH_val
    result = t.execute()
```

(continues on next page)

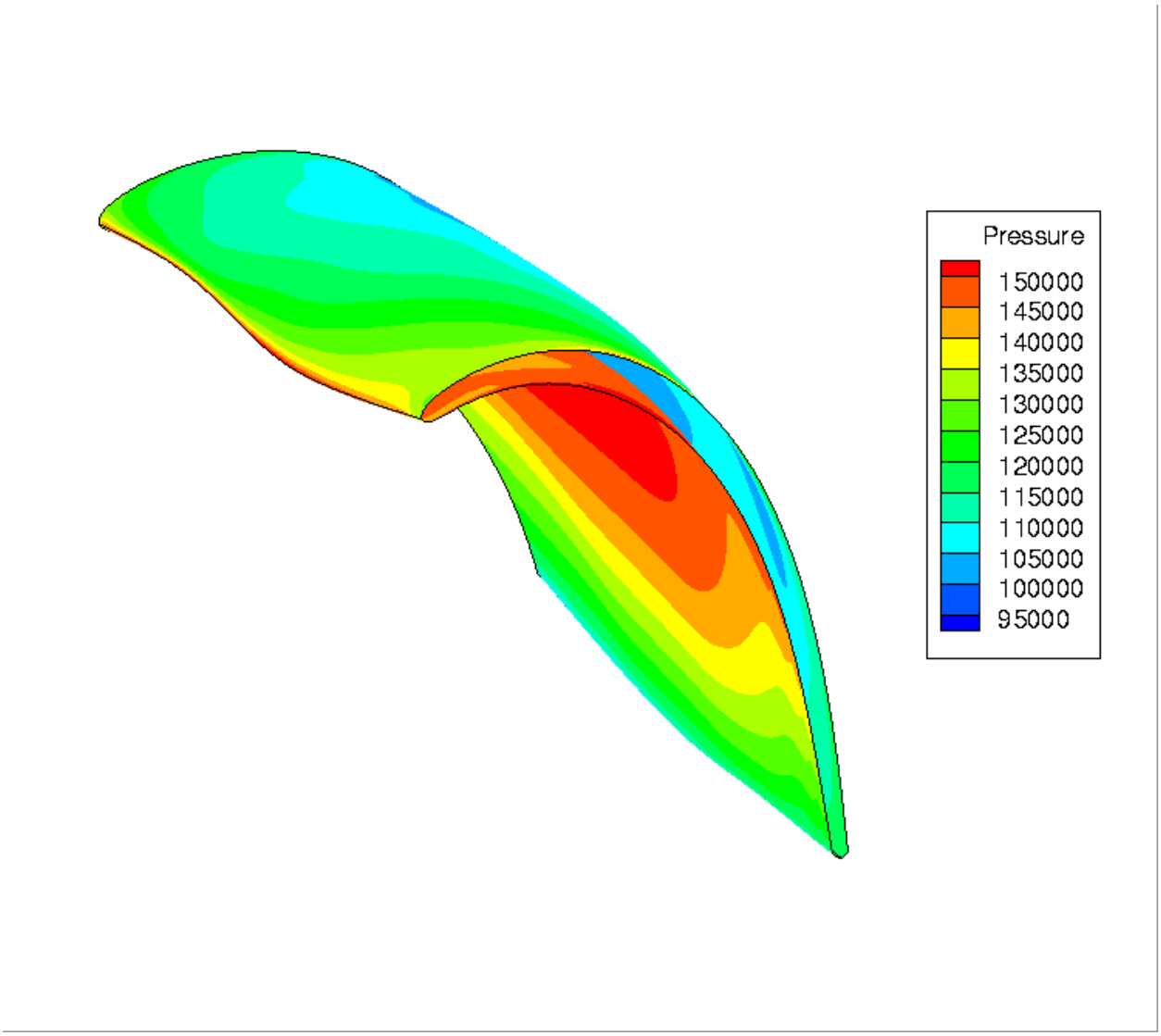


Fig. 1: The pressure is stored in BC_t node of the HDF-CGNS file.

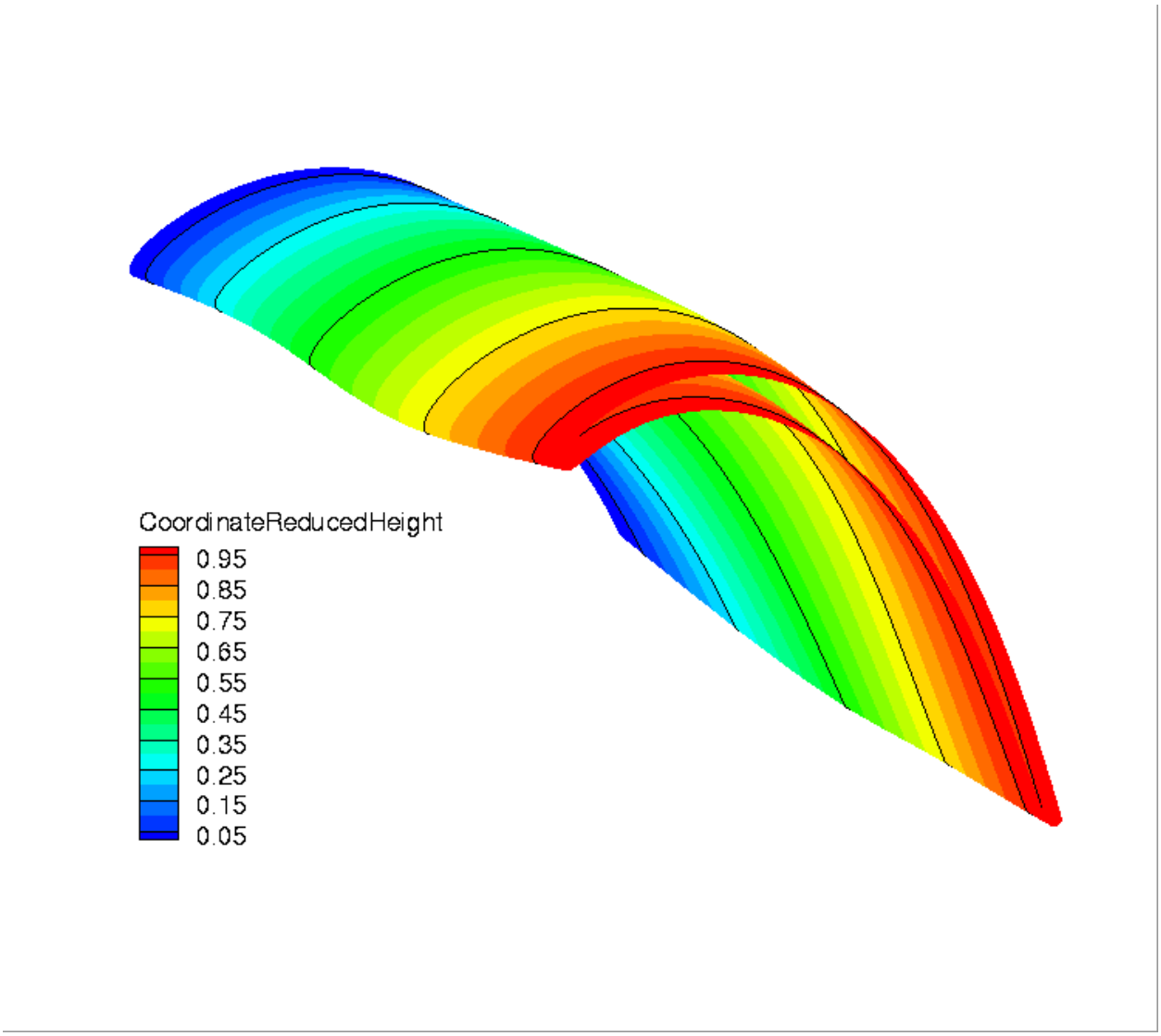


Fig. 2: Location of heights on the blade.

(continued from previous page)

```
writer = antares.Writer('hdf_cgns')
writer['base'] = result
writer['filename'] = os.path.join('all_iso.cgns')
writer['append'] = True
writer['base_name'] = '%s' % hoH_val
writer['dtype'] = 'float32'
writer.dump()
```

'CoordinateReducedHeight' corresponds to the h/H variable. Note the option 'append' of the writer to concatenate results in a single file.

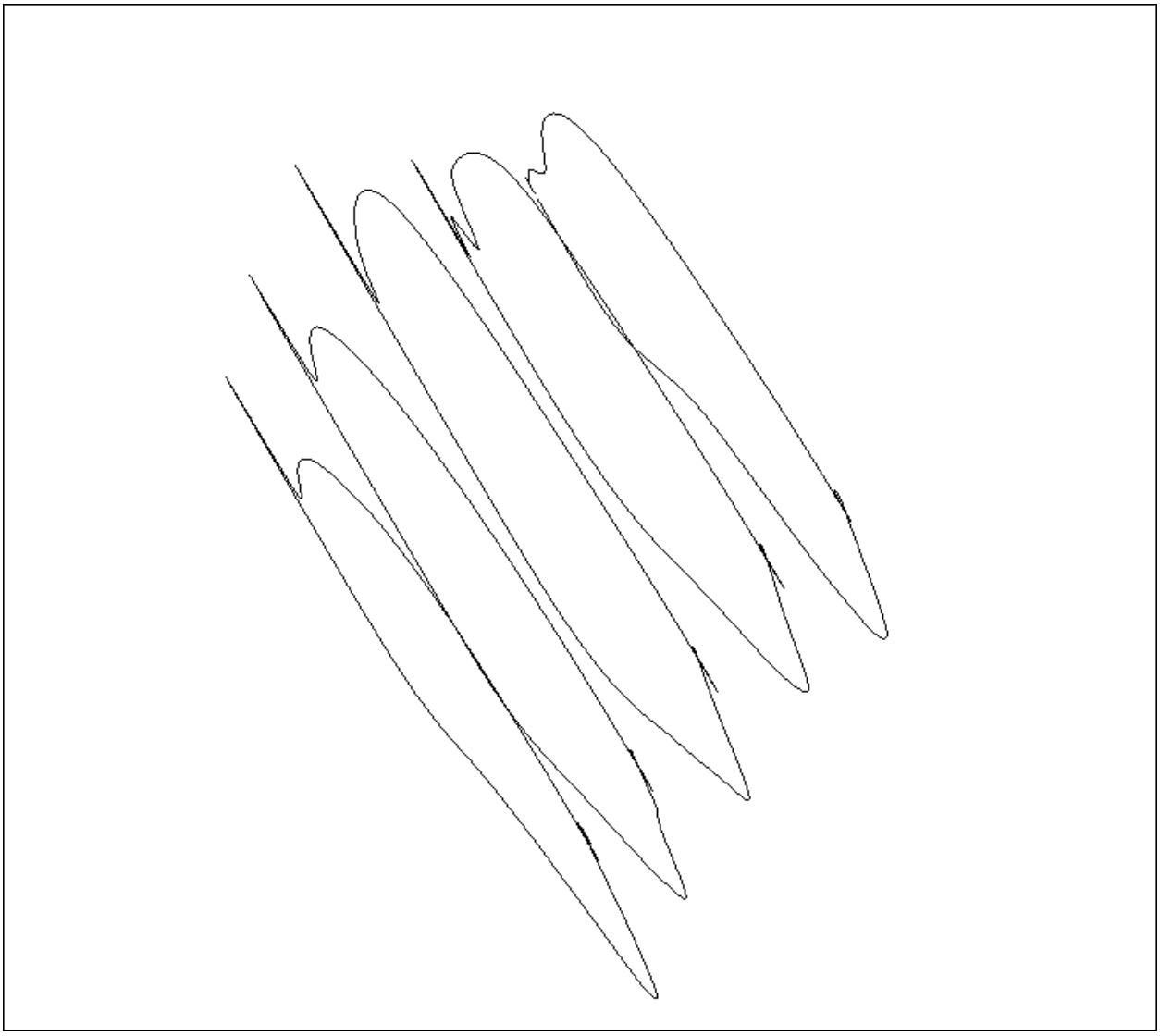


Fig. 3: Pressure plots on the five blade profiles.

4.3 Getting started with parallelism with MPI

4.3.1 Installing dependencies

Some dependencies are required for parallel support in Antares.

- mpi4py >= 2.0.0
- METIS >= 5.1.0
- HDF5 >= 1.8.15 parallel
- h5py >= 2.7.1 parallel

METIS is required when reading a base with unstructured zones. Parallel support in HDF5 and h5py is required for writers.

Installing h5py parallel through *pip* or *conda*:

```
$ conda install mpi4py
$ export HDF5_DIR=/path/to/hdf5/parallel
$ CC=mpicc HDF5_MPI=ON pip install --no-binary=h5py h5py
```

check with:

```
h5cc -showconfig
```

4.3.2 Writing an Antares script

Going parallel is straightforward

Any regular antares script can be executed both in parallel and sequential.

Example:

```
# file script.py
from antares import Reader, Writer, Treatment

# Read a base:
# - structured zones are distributed between cores
# - unstructured zones are splitted, each proc gets 1 partition
r = Reader('hdf_cgns')
r['filename'] = 'input_base.cgns'
base = r.read()

# Apply a treatment:
# - cores internally synchronise with each other if needed
cut = Treatment('cut')
cut['type'] = 'plane'
cut['origin'] = [0.001, -0.01, 0.2]
cut['normal'] = [0.145, -0.97, -0.2]
new_base = cut.execute()

# Write the result:
# - cores collectively build a single output file
```

(continues on next page)

(continued from previous page)

```
w = Writer('hdf_cgns')
w['base'] = new_base
w['filename'] = 'output_base.cgns'
w.dump()
```

Sequential execution:

```
$ python script.py
```

Parallel execution on *NPROC* cores:

```
$ mpirun -n NPROC python script.py
```

Work In Progress: support of parallelism in Antares is still very sparse - supported I/O format: `hdf_cgns` - supported treatments: `cut`, `integration`, `thermo1`

Parallelism is based on MPI

[MPI](https://en.wikipedia.org/wiki/Message_Passing_Interface) (Message Passing Interface) is a programming standard that allows different processes to communicate with each other by sending data.

One does not need to known the MPI programming model to use antares in a parallel script. Just write your code normally, then execute it as stated above.

Antares uses the python library *mpi4py* that implements the MPI standard. mpi4py's function always return a valid result, even in sequential. Antares wraps a small subset of the MPI interface into the *antares.parallel.Controller.PARA* object. *PARA*'s wrappers act as a stub when *mpi4py* is not installed (for sequential execution only !)

```
# accessing MPI interface
# -----------------------

import antares

para = antares.controller.PARA

# get antares' MPI communicator:
# this is the main entry point for handling communications between processes
comm = para.comm

# get processor rank
# this the identifier of the process
# ranks take their values in the internal [1, NPROC]
rank = para.rank
# or
rank = comm.rank

# playing with MPI
# ----------------

# add all ranks and broadcast results to all cores
result = para.allreduce(rank, para.SUM)
# or
from mpi4py import MPI
```

(continues on next page)

(continued from previous page)

```
result = comm.allreduce(rank, MPI.SUM)

# get number of cores
nproc = para.size
# or
nproc = comm.size

# display result, one core after the other
for p in range(nproc):
    if p == rank:
        print(rank, result)
    comm.Barrier()

# display something by process rank 0 only
if rank == 0:
    print 'done'
```

An internal function is provided to print a message in a ring communication.

```
from antares.parallel.controller import ring_print
ring_print("my message")
```

4.3.3 Troubleshooting

Dumping a base to HDF-CGNS format is too slow !

Collectively creating an HDF file can be very slow if the file is structured with a large number of groups and datasets. HDF-based writers take an optional argument 'link' for selecting a different mode for writing the file.

Available modes are:

- 'single': Default mode. All cores collectively build a single file.
- 'merged': cores build their own files, then merge them into a single one.
- 'proc': cores build their own files, then link them into a main index file.
- 'zone': cores build 1 file per zone, then link them into a main index file.

Every auxiliary files built by modes 'proc' and 'zone' store valid antares bases, but only the main file give access to every zones and store shared information.

'single' is optimal in sequential. 'proc' is optimal in parallel.

Output file contains many zones with odd names instead of a single one !

By default, readers split every unstructured zone so that every processor get a partition. Every processor have to handle zones with different names, so the original name is suffixed by the processor rank.

Merging back zones is not yet implemented in writers.

I already have multiple unstructured zones and don't want to have them split.

Readers take an optional 'split' argument you can set to 'never'. Both structured and unstructured zones will be distributed to cores.

4.4 Attributes Management

Each API object base, zone, instant, *Boundary* (page 11), and *Family* (page 20) contains an **attrs** attribute that behaves like a Python dictionary.

An attribute can be set using the classical *(key, value)* pair.

```
import antares
base = antares.Base()
base.attrs['omega'] = 10
```

In the above example, an attribute is set at the base level. This attribute will be visible for all zones and instants belonging to the base. As an example, if you are dealing with a single rotor configuration, then you can set the rotation velocity to the base.

We can add an attribute to a zone or an instant too.

```
import antares
base = antares.Base()
base['rotor'] = antares.Zone()
base['rotor'].attrs['omega'] = 10
base['stator'] = antares.Zone()
base['stator'].attrs['omega'] = 0
```

In the above example, the attribute *omega* is set according to the type of the zone.

The value you can add to the **attrs** attribute can be of any type. The user (the applicative script writer) is solely responsible for the proper management of these new entities *(key, value)*.

Except if documented, the *key* can not be known from the Antares library. As such, the *value* can not be used in other functions or methods from the library.

The following tutorials explain how to use some generic treatments through three applications. If you want to exercise, then you have to consider getting the notebooks. Otherwise, the results are given if you follow the links.

- NASA Rotor 37, `notebook for NASA Rotor 37`
- PRECCINSTA combustor, `notebook for PRECCINSTA combustor`
- DLR F12 aircraft, `notebook for DLR F12 aircraft`

CHAPTER

FIVE

ADVANCED INFORMATION

5.1 Environment Variables

You can modify the behaviour of antares before launching the python interpreter by setting some environment variables.

5.1.1 Header

If you want to print the antares header when importing the module, then set the environment variable `ANTARES_NOHEADER` to false.

```
export ANTARES_NOHEADER=false  # in bash
setenv ANTARES_NOHEADER false  # in csh
```

5.1.2 Early Logger

If you want to print debugging information raised during the import of antares, then set the environment variable `ANTARES_EARLY_LOGGER` to true.

```
export ANTARES_EARLY_LOGGER=true  # in bash
setenv ANTARES_EARLY_LOGGER true  # in csh
```

5.2 Constant Variables

The variables defined here are used globally in the code. If these variables are modified, the code behaves differently.

The following variables are the default coordinate names in Antares. If a variable name is included in `KNOWN_COORDINATES`, then the variable will be considered as a mesh coordinate, except otherwise stated in a specific file reader.

`Constants.`**`KNOWN_COORDINATES = [['x', 'y', 'z'], ['x', 'y', 'Z'], ['x', 'Y', 'z'], ['x', 'Y', 'Z'], ['X', 'y', 'z'], ['X', 'y', 'Z'], ['X', 'Y', 'z'], ['X', 'Y', 'Z'], ['CoordinateX', 'CoordinateY', 'CoordinateZ']]`**

`Constants.`**`LOCATIONS = ['node', 'cell']`**

5.3 Global Variables

5.3.1 Coordinates

`GlobalVar.coordinates = ['x', 'y', 'z']`

This variable defines the default coordinate names. The first coordinate is, by default, the rotation axis (when needed).

This variable can be modified explicitely with the following function.

`antares.core.GlobalVar.set_default_name(var, value)`

Set default variable name to value.

Parameters

- **var** (`str` in ['coordinates', 'base', 'zone', 'instant']) – variable name.
- **value** (`list(str)`, or `str`) – new default name.

`GlobalVar.cartesian_coordinates = ['x', 'y', 'z']`

This variable defines the default cartesian coordinate names (same as GlobalVar.coordinates).

`GlobalVar.cylindrical_coordinates = ['x', 'r', 'theta']`

This variable defines the default cylindrical coordinate names.

This variable can be modified explicitely with the following function.

`antares.core.GlobalVar.set_default_cylindrical_coordinates(coord_names)`

Set default GlobalVar.cylindrical_coordinates to coord_names.

Parameters

coord_names (`list(str)`) – new default names of coordinates.

5.3.2 Default names

`GlobalVar.base_name = '0000'`

`GlobalVar.zone_name = '0000'`

`GlobalVar.instant_name = '0000'`

These variables can be modified explicitely with the following function.

`antares.core.GlobalVar.set_default_name(var, value)`

Set default variable name to value.

Parameters

- **var** (`str` in ['coordinates', 'base', 'zone', 'instant']) – variable name.
- **value** (`list(str)`, or `str`) – new default name.

5.3.3 FILES_CACHED

This singleton manages the file cache system to reduce read operations in some situations. It is not exposed to the antares user interface yet.

`OpenFilesCache.FILES_CACHED = None`

This variable can be modified explicitely with the following function.

`antares.io.OpenFilesCache.set_open_files_cache(enable, nb_max_open_files=10)`

Set the status of the global module variable FILES_CACHED.

Parameters

- **enable** (*bool*) – True if the cache should be enabled, False if the cache should be disabled.
- **nb_max_open_files** (*int*) – The maximum number of files that can be stored in the cache (when the cache is enabled).

File Cache System for Lazy Loading Pattern

Implementation of the caching system for open files in Antares.

The lazy loading feature of many readers works by opening the target file, reading one variable and then closing the file. This is a poor strategy performance-wise since it can potentially mean opening the same file several times (if all the zones and instants are in the same file for example). This is especially true when using the h5py library, since opening the file also means reading the file metadata. Therefore, the solution implemented in this module aims at keeping files open as long as possible for their reuse in the code. This can significantly improve the I/O performance of the code.

The drawbacks of this strategy is that the file descriptors are kept open until the end of the script. This can be potentially dangerous if the user tries to modify an open file externally. For example, the h5py library will not allow to overwrite an already open file, or the OS might create additional hidden files when deleting an open file over an NFS file system. Some of these issues are addressed internally in Antares. Writers using the h5py library will check if the target file is open and they will close the descriptor before writing the new file to the disk. Additionally, the cache keeps track of the modification time to ensure that any open file descriptor has not been modified since the time it was added to the cache. However, not all edge cases can be treated, and the user can close individual files or flush the entire cache if it is necessary.

5.3.4 Main functions

class `antares.io.OpenFilesCache.OpenFilesCache(nb_max_open_files=10)`

Class to manage cached files.

The OpenFilesCache is a class aiming at keeping open files descriptors for their reuse. This class is especially used in conjunction with the lazy loading feature of many readers. In many readers, the lazy loading is done by opening the file, reading a single variable and closing the file. This is costly if many variables need to be read from a single file (It is especially costly with .h5 files open with h5py, because the file metadata is read every time the file is open). The OpenFilesCache keeps the files descriptor open, avoiding the extra cost of opening the same file many times.

Files can be added or retrieved from this cache via the *add_file()* (page 372) and *get_file()* (page 372) method. We can also check if a file exists in the cache using the method *is_file_open()* (page 372). Additionally, all the files can be removed from the cache using the method *flush()* (page 372).

The cache works by keeping a dictionary whose key is the absolute path of the open files and the values is another dictionary that stores the actual file descriptor and the last time the descriptor was required (via the method *get_file()* (page 372)).

This cache limits the number of stored descriptors using the attribute `nb_max_open_files` (default 10). This is set at the declaration time and it cannot be changed later. When a new file is added, the object checks if adding the new file will not exceed the limit of open files. If it does not, then the file is added. If it does exceed the limit, then the oldest accessed file is removed from the cache and the new file is added. Removing a file from the cache means invoking the close method on the descriptor (WARNING: if the descriptor does not have a close method then an error will be thrown).

add_file(*filename*, *fid*)

Add a file descriptor to the list of cached files.

Parameters

- **filename** (*str*) – path to the file (absolute or relative to the current working directory)
- **fid** – Open descriptor of the file

close_file(*filename*)

Close an specific file in the cache.

Parameters

filename (*str*) – path to the file to close (absolute or relative to the current working directory)

enable(*nb_max_open_files*)

Enable the cache for use.

If this method is called in an already enabled cache, then it will reset the attribute `nb_max_open_files`. If the new value is greater than the old one, then nothing is done and `nb_max_open_files` is set to the new value. However, if the new value is lower than the previous value, then the oldest files are closed until the length of the caches is equal to the new **nb_max_open_files** value.

Parameters

nb_max_open_files (*int*) – Number maximum of files that can be stored in the cache

flush()

Close all files in the cache.

get_file(*filename*)

Get the file descriptor associated to a filename.

An error is thrown if the file is not present in the cache.

Parameters

filename (*str*) – path to the file.

Returns

The file descriptor associated to the filename.

is_file_open(*filename*)

Check if a file is present in the cache.

Parameters

filename (*str*) – path to the file to be checked (absolute or relative to the current working directory)

Returns

True if the file is present in the cache, False otherwise

5.4 Memory Layout of Arrays for Variables

The term Variable means a physical quantity (pressure, density, etc.) or a geometric quantity (distance, coordinate, etc.) or another quantity.

A variable can have many values which can be gathered in a container which is equivalent to a mathematical vector. Many variables are not gathered in a single container. As examples, mesh coordinates are stored separately in three vectors, and not in a matrix. Also, primitive variables are stored separately in vectors, and not in a matrix.

So a variable may be stored in the container `Instant` as a numpy[134] array.

Even if the variable vector has only one mathematical dimension, the values of the variable may be related to a specific topology.

For structured grids, an array may have more than one topological dimension, say N. Then, the shape of the array is the tuple of array dimensions. Indeed, a structured grid can be represented as a topologically regular array of points. Each point or element in this N-cube topology can be addressed with N independent indices. As an example, for a two dimensional structured grid, the number of topological dimensions is two, the topology is a square, and there are two indices i and j.

To browse the values of a one-dimensional topological array, there is no ambiguity since there is only one index. But, for a multi-dimensional topological array, the dimensions can be accessed in different ways.

With antares, the point order increases the fastest as one moves from the last to the first dimension. Applied to the above example, the point order increases in j fastest, then i.

5.4.1 Interaction with the VTK[Page 373, 135] library

Some treatments requires the VTK[136] library. The VTK[137] library has its own underlying data structures. Then, antares data arrays must be converted into VTK[138] data structures. This is particularly important for multidimensional arrays where the memory layout may be different, and then copies of arrays may be required. Indeed, with VTK[139], the point order increases in i fastest, then j.

From VTK < 8.1, it was not possible to avoid copies of arrays for coordinates since they were required as a matrix (Array of Structures) by the library. For VTK > 8.1, it is now possible to avoid copies of arrays for coordinates since they can be passed as vectors (Structure of Arrays) by the library.

The other important point to avoid array copies is the memory layout of the numpy[140] arrays. If the fortran-contiguous memory layout may not induced any copy in the best case, whereas the C-contiguous memory layout will surely imply copies.

Another important point is the datatype of the array values. If the datatype is the single precision floating-point type (float32, 4 memory bytes), then a copy is made to get a double precision floating-point type (float64, 8 memory bytes).

For technical details, you can refer to the following functions:

`antares.utils.VtkUtilities.`**`instant_to_vtk`**(*instant*, *coordinates*, *dtype='float64'*, *memory_mode=False*)

Convert an antares instant to a vtk object.

Parameters

- **`coordinates`** (*`list(str)`*) – the coordinates of the current instant

[134] https://numpy.org
[135] https://www.vtk.org/
[136] https://www.vtk.org/
[137] https://www.vtk.org/
[138] https://www.vtk.org/
[139] https://www.vtk.org/
[140] https://numpy.org

- **dtype** (*str*) – the data type used to write the file

`antares.utils.VtkUtilities.instant_to_vtk_sgrid(instant, coordinates, dtype='float64', memory_mode=False, missing_coordinates=None)`

Convert an structured instant to a vtkStructuredGrid object.

An instant has a shape with many dimensions, e.g (nx, ny) in a 2D configuration. It can be viewed as a matrix with nx lines and ny columns. The elements of this matrix are then ordered in a C-order in which the index on the last dimension (ny) changes first. The VTK library requires the opposite: the index on the first dimension (nx) must change first, like a fortran order. That is the reason of the transpose operation performed on variables. Note that the above has nothing to do with the memory layout of the data.

This matrix must be stored in memory. If the memory layout is a fortran contiguous layout, then data can be passed to the VTK library without copy. With nympy, there are at least 3 equivalent ways to do it:

- arr.T.reshape(-1) # order='C' is implicit
- arr.reshape(-1, order='F')
- arr.ravel(order='F')

If the memory layout is a C contiguous layout, then data can not be passed to the VTK library without copy.

If the memory layout is not contiguous at all, then data can not be passed to the VTK library without copy.

Parameters

- **coordinates** (*list(str)*) – The coordinate names of the current instant.
- **dtype** (*str*) – The data type used to write the file.
- **missing_coordinates** (*list(str)*) – A list with the coordinate name that is missing to create a 3D coordinate system. List of one element.

`antares.utils.VtkUtilities.instant_to_vtk_ugrid(instant, coordinates, dtype='float64', memory_mode=False, missing_coordinates=None)`

Convert an unstructured instant to a vtkUnstructuredGrid object.

Parameters

- **coordinates** (*list(str)*) – The coordinate names of the current instant.
- **dtype** (*str*) – The data type used to write the file.
- **missing_coordinates** (*list(str)*) – A list with the coordinate name that is missing to create a 3D coordinate system. List of one element.

5.5 Equation Manager

Python module for an equation manager.

5.5.1 Detailed Class Documentation

class antares.eqmanager.kernel.eqmanager.**EqManager**(*modeling*)

Class of Equation Manager.

Kernel of formula management.

__init__(*modeling*)

Constructor.

Parameters

modeling (*str*) – The name of a set of equations. This modeling contains equations that can be found in the directory that has the same name under the directory named *formula*. eg: the modeling *internal* must be in the directory *formula*

Variables

- **_models** (*dict*) – The dictionary that contains all modeling. A modeling is a dictionary with the key *files*. The value of the key *files* is a list of filenames (without extension) in which the functions are defined.
- **_results** (*dict*) – Store the result of each computed formula to avoid recomputation.
- **_functions** (*dict*) – The function dictionary has keys *functions* and *args*. The value of the key *functions* is a python function object. The value of the key *args* is a list of input names that are the arguments of the function.

```
ex: {'c':  {'function': <function ...>,
            'args': ['ro', 'psta', 'gamma']}}.
```

model

Store the current modeling of the computer.

Type

str or None

results(*name*)

Return the result of the formula.

Parameters

name (*str*) – The name of a variable.

remove_result(*elt*)

Remove an element from the stored results.

Parameters

elt (*tuple(str, str)*) – Formated variable name.

remove_results(*safe=None*)

Remove all the stored results except those given as args.

Parameters

safe (*list(tuple(str, str))*) – Protected variable names.

add_located_input(*inputs*)

Add inputs that will be used in a known formula.

Parameters

inputs (*dict*) – Set of items to consider as inputs.

ex: inputs = {('ro', 'node'): array([1.3, 1.3]),
('roE', 'node'): array([253312.5, 259812.5])}

`add_input`(***kwargs*)

Add an input that will be used in a known formula.

Parameters
inputs (*depend on each value.*) – Set of items to consider as inputs.

`get_str_computed_formulae`()

Return a string with known formulae.

`get_list_computed_formulae`()

Return the list of computed formulae.

`get_str_formula_names`()

Return a string with known formula names.

`get_list_formula_names`()

Return the list of known formula names.

`get_formula_name_synonyms`()

Return the synonyms of known formula names.

`get_computable_formulae`()

Return a string with computable formulae.

`change_model`(*modeling*, *species_database=None*, *addons=None*)

Change the modeling of the equation manager.

Parameters

- **modeling** (*str*) – The name of a set of equations.
- **addons** (*list(str)*) – The names of additional files for equations.

`set_formula`(*formula*)

Set a new formula in the modeling.

Parameters
formula (*str*) – The name of the new formula.

Operations available are in _numpy_operators.

`add_function`(*new_func*)

Set a new function in the modeling.

Parameters
new_func (*func*) – a python function.

`compute`(*variable*)

Get the result of the given variable.

Launch computation if needed.

Parameters
variable (*tuple(str, str)*) – The name and the location of the variable to compute.

Returns
The value of the formula.

Return type
Depending on the formula.

_handle_syn(*missing_var*, *loc*)

Perform operations to use synonyms of input variables.

Parameters

- **missing_var** (*list(var)*) – The name of variables that have not been found so far.
- **loc** (*str in ['node', 'cell']*) – The location of the variables.

Returns
True if no variables are missing (false otherwise).

Return type
bool

_check_formula(*formula*, *checked_vars=None*)

Check if all dependencies of the given formula are available.

Parameters
formula (function or str) – The formula function or the formula name to be checked.

Returns
True if the formula can be computed (false otherwise), and the missing formula/var values.

Return type
bool, list(str)

_compute(*formula*, *loc*)

Return the value of the given formula computation.

Launch computation of dependencies if needed.

Parameters

- **formula** (*:obj: function*) – The formula to compute.
- **loc** (*str in ['node', 'cell']*) – The location of the variable.

Returns
The value of the computation.

Return type
Depending on the formula.

_store_function(*name*, *obj*)

Store the function object in the dictionary.

Parameters

- **name** (*str*) – Name of the function.
- **obj** (*:obj:function*) – Function object.
- **container** (*dict*) – Dictionary of functions.

_create_function_from_string(*equation*)

Create a function object from a string.

Parameters
equation (*str*) – The literal expression of the equation.

_find_var_and_symbols(*equation*)

Find variables and symbols needed by the equation.

Parameters
equation (*str*) – The literal expression of the equation.

_load_species_data(*needed_vars*)

Load species database.

Parameters
needed_vars (*list(str)*) – Names of variables.

_get_all_formula_dependencies(*formula*)

Get all dependencies of the formula.

Parameters
formula (*str*) – The formula name.

Returns
The list of variables used to compute the formula.

Return type
list(str)

5.6 Equation Modeling

Python module for the modeling of equations.

A modeling is defined as a set of equations governing a flow or any other physical phenomenon.

As example, you can defined a modeling *incompressible* that will contain all the equations you choose to define this physics.

5.6.1 Equation files

Functions of a particular modeling are defined in files in a path associated this modeling. e.g.: *equations.py* in *formula/internal* contains the functions associated to the modeling *internal*.

5.6.2 Functions

Each function defines a variable or a set of variables.

- If it is a single variable, then the name of the function is the name of the variable.

e.g.:

```
def u(rhou, rho):
    return rhou/rho
```

- If it is a set of variables, then the name of the function is simply a name.

e.g.:

```
def cons_abs(rhov, rhow):
    return {'rhov': rhov,
            'rhow': rhow}
```

The body of the function can be as complex as you want. You can query databases, read files, perform Newton algorithms, etc.

The arguments of the function are either variable names or function names.

5.6.3 Detailed Class Documentation

class antares.eqmanager.kernel.eqmodeling.**EqModeling**

Class of EqModeling.

This class reads and stores all modeling available.

__init__()

Constructor.

Variables

_models (*dict*) – Store all modeling information. The dictionary that contains all equation modeling. A particular modeling is a dictionary with the keys *files*. The value of the key *files* is a list of filenames (without extension) in which the functions are defined.

_load_formula()

Read all formula files (python files).

Read all files that include a function definition.

Return models

The dictionary that contains all equation modeling.

- A modeling is a dictionary with the key *files*. The value of the key *files* is a list of filenames (without extension) in which the functions are defined.
- A modeling has also key *functions* and *synonyms*. The value of the key *functions* is a dictionary with the keys corresponding to the name of each function of the modeling. Each function name corresponds to a dictionary with a key *function* and a value of a function object. The value of the key *synonyms* is a dictionary with the keys corresponding to the name of each variable that have been chosen to write the equations of the modeling. Each variable name corresponds to a dictionary of synonyms.
- A modeling may also have a key *addons*. The value of the key *addons* is a dictionary with the keys corresponding to the name of each file in the addons repository of the modeling. Each file name corresponds to a dictionary with names of functions as keys and function objects as values.

Rtype models

dict

```
{'compressible':
    {'files': ['equations', 'complement_1'],
     'functions': {'c': {'function': <function c at 0x>},
                   'Cp': {'function': <function Cp at 0x>},
                   'E': {'function': <function E at 0x>}},
```

(continues on next page)

(continued from previous page)

```
    'addons': {'kl': {'k': <function k at 0x>},
                      'l': <function l at 0x>},
               'spalart': {'nut': <function nut at 0x>}}}},
'incompressible':
   {'files': ['equations'],
    'functions': {'c': {'function': <function c at 0x>},
                  'u': {'function': <function u at 0x>}}}}
```

5.7 List of Available Equation Modelings

5.8 Logging Messages

Antares uses the logging[141] module of python to print messages. For more information on this module, you may consider the following two urls:

- https://python-guide-pt-br.readthedocs.io/fr/latest/writing/logging.html
- https://realpython.com/python-logging/#formatting-the-output

As a reminder, the rank of logging messages is: NOTSET, DEBUG, INFO, WARNING, ERROR, CRITICAL.

Starting from version 1.14.0, antares does not produce any output messages by default, except the header when importing the module. The *progress bar* (page 347) is also disabled. If you want to get the old behaviour back, then you can use the following instructions:

```
import logging
import antares
antares.add_stderr_logger(logging.INFO)
antares.set_progress_bar(value=True)
```

The standard output is set to INFO, so the console logger will show warnings, errors, and criticals as well. If you also want available debugging information, then use:

```
antares.add_stderr_logger(logging.DEBUG)
```

The function *add_stderr_logger()* (page 380) is really a helper function. You may consider to customize your application with what follows in the next section.

antares.**add_stderr_logger**(*level=10*)

Add a StreamHandler to the antares logger.

Useful for debugging.

Returns the handler after adding it.

[141] https://docs.python.org/2/library/logging.html

5.8.1 How to customize your outputs

When creating your application, you can ask the module antares to output messages available in its component. First, create your own logger with a specific handler (or more) with custom formats such as:

```
import logging
logger = logging.getLogger("antares") # get the logger of antares
handler = logging.StreamHandler()     # messages will go to the console
formatter = logging.Formatter('%(asctime)s %(name)-12s %(levelname)-8s %(message)s')
handler.setFormatter(formatter)       # format of the messages
logger.addHandler(handler)            # put the handler to the logger
logger.setLevel(logging.INFO)         # set the logging level of verbosity
```

All informative elements will be output to the console as

```
2019-07-02 14:06:41,672 antares.treatment.TreatmentCut INFO     Cut
2019-07-02 14:06:41,693 antares.treatment.TreatmentMerge INFO     Merge
2019-07-02 14:06:41,715 antares.treatment.TreatmentUnwrapLine INFO     Unwrap
```

If you want to output messages in a file, then use a FileHandler as:

```
import logging
logger = logging.getLogger("antares")     # get the logger of antares
handler = logging.FileHandler('logg.txt') # messages will go to the file 'logg.txt'
formatter = logging.Formatter('%(asctime)s %(name)-12s %(levelname)-8s %(message)s')
handler.setFormatter(formatter)           # format of the messages
logger.addHandler(handler)                # put the handler to the logger
logger.setLevel(logging.DEBUG)            # set the logging level of verbosity
```

The file 'logg.txt' will contain all messages above DEBUG.

When creating your application, you can ask for only a submodule of antares to output messages available in its component. For example, if you only want messages from the merge treatment, then get the logger of this module as

```
logger = logging.getLogger("antares.treatment.TreatmentMerge")
```

and use your favorite handler.

—

5.9 Package structure

the Antares root directory contains the following elements:

```
bin/                (binaries)
antares/,           (package)
doc/,               (documentation)
examples/           (examples)
setup.py            (setup file)
antares.env         (sh source file)
antares_csh.env     (csh source file)
```

5.10 Special functions

antares.**store_vtk**(*value=None*)

Store vtk data structure.

Function to make vtk efficient for multiple cuts.

When using treatment that uses VTK, Antares has to convert its API structure to the corresponding VTK structure. This copy the data and is hence costly. To save the converted data for reuse, consider using this function.

Parameters
value – set to True to activate this functionnality

5.11 User-specific Modules

When the library antares is installed, some default treatments are made available to users. However, you can tell the library to look for your own treatments.

5.11.1 Using Third-party Treatments

You can provide Antares with treatments that are not in the default path. Just setup the environment variable *EXT_ANT_TREATMENT_PATHS* (page 382).

EXT_ANT_TREATMENT_PATHS

path1:path2:path3

The files that implement treatments are prefixed by *Treatment* and suffixed by *.py*. All such files found in these previous paths will be loaded and made available in Antares.

5.12 Miscellaneous

5.12.1 API Utility Functions

antares.utils.ApiUtils.**get_coordinates**(*base*, *user_keys*)

Return the coordinate names.

Look for the coordinate names in the base, or in the user_keys.

Parameters

- **base** (Base) – base
- **user_keys** (*dict*) – user dictionary

Returns
the name of coordinates

Return type
list(str)

5.12.2 Gradient Utility Functions

The following functions are only used internally in Antares. They give some insights on the methods used to compute gradients.

`antares.utils.GradUtils.tri_face(pt0, pt1, pt2, instant, coord_names, vol, grad_variables=None, grad=None)`

The face is a triangle with points p_0, p_1, and p_2. The normal to this face is given by $\vec{n} = \frac{1}{2}(p_1 \vec{-} p_0)(p_2 \vec{-} p_0)$. The contribution of this face to the volume is given by $\frac{1}{3}(\frac{1}{3}(p_0 + p_1 + p_2)) \cdot \vec{n}$.

Parameters

- **pt2** (*int pt0, pt1,*) – points of the triangle
- **instant** (Instant) – instant containing the variables for which the gradient is to be computed
- **coord_names** (*list(str)*) – coordinate names
- **vol** (*ndarray*) – volume
- **grad_variables** (*None or list(str)*) – list of variables for which the gradient is to be computed
- **grad** (*ndarray*) – gradients

Return vol
volume is incremented with the face contribution

Return grad
gradients are incremented with the face contribution

`antares.utils.GradUtils.qua_face(pt0, pt1, pt2, pt3, instant, coord_names, vol, grad_variables=None, grad=None)`

The face is a quadrilateral with points p_0, p_1, p_2, and p_3. The normal to this face is given by $\vec{n} = \frac{1}{2}(p_2 \vec{-} p_0)(p_3 \vec{-} p_1)$. The contribution of this face to the volume is given by $\frac{1}{3}(\frac{1}{4}(p_0 + p_1 + p_2 + p_3)) \cdot \vec{n}$

Parameters

- **pt3** (*int pt0, pt1, pt2,*) – points of the quadrilateral
- **instant** (Instant) – instant containing the variables for which the gradient is to be computed
- **coord_names** (*list(str)*) – coordinate names
- **vol** (ndarray) – volume
- **grad_variables** (*None or list(str)*) – list of variables for which the gradient is to be computed
- **grad** (ndarray) – gradients

Return vol
volume is incremented with the face contribution

Return grad
gradients are incremented with the face contribution

`antares.utils.GradUtils.compute_grad(instant, coordinates, grad_variables=None)`

Compute gradients at cell centers.

It handles both structured and unstructured grids.

if grad_variables is None, then only the cell volumes will be computed.

The method to compute the volume and the gradients of/in a cell is based on the Green-Ostrogradski theorem.

i.e. $V = \frac{1}{3} \oint_S C.n \, dS$ with V the volume, C one point on the surface S, and n the normal.

Parameters

- **instant** (Instant) – container of variables
- **coordinates** (*list(str)*) – coordinate names
- **grad_variables** (*list(str)*) – variable names for which gradient will be computed.

5.12.3 Utils for cutting treatment

Class for canonical geometric cells

The class for canonical geometric cells.

Given a cell described by its vertices, provides:

- recipes for building faces and edges from cell vertices.
- recipes for building intersection polygons from cell components intersecting the surface.

The recipes for building faces and edges are arrays of local vertices. The recipes for building intersection polygons requires the list of intersecting vertices (ON-SURFACE vertices) and the list of intersecting edges (cross-edges). A binary hash is computed the 2 lists and is used as a key for finding the corresponding recipe in a hashtable. The recipe is the ordered list of cell components that would form the intersecting polygon.

Cell components are the union of N vertices and M edges:

- Vertices are listed from 0 to N-1
- Edges are listed from N to N+M-1

`class antares.utils.geomcut.geokernel.AbstractKernel`

Abstract base classe for geometric kernels.

It provides a kind of hashtable. Depending on which components of a cell intersect whith a cut-surface, returns a recipe for building the intersection polygon. The set of components that are part of the intersection is used as a key.

Attributes:

- edgemap: list of edges for 1 cell, expressed as couples of local vertices.
- **facemap: list of faces for 1 cell. A face has 4 points at most.**
  facemap is expressed as lists of four vertices.
- ctypeset: list of possible polygons resulting from a cut, expressed as couples (nb. of vertices, name)
- nvtxface: number of vertices per face

static build_component(*vrtx_array*, *edge_array*)

Vertices and edges are both considered as cell 'components'. Properly merge 2 arrays of vertices and edges to output an array of components.

Parameters

- **vrtx_array** – array of cell vertices. element i holds a value for vertex i.
- **edge_array** – array of cell edges. element j holds a value for edge j.

Returns
array of cell components. element k holds a value for component k, which can be either a vertice or a edge.

static build_hash(*component_bool*, *edge_bool=None*)

Every component may be cut or not by the cut-surface. A vertex is cut when it is part of the surface. An edge is cut when its 2 ends are on opposite sides of the surface.

Parameters

- **component_bool** – a boolean mask saying wether each component is cut or not.
- **edge_bool** – a boolean mask saying wether each each is cut or not. Provided if and only if component_bool contains vertex information only.

Returns
a cell-hash (binary key) that carry the same information as the boolean mask.

classmethod recipe(*hashlist*)

A Geokernel object aims at providing an easy way of building polygons defined by the intersection of a plane of cut and a 3D cell. Any component cut by the surface issues a cut-point on the surface. Depending on wich cell component are cut by the surface, this function returns the ordered list of components whose corresponding cut-points draw a cut-polygon.

Parameters
hashlist – list of cell-hash

Returns
ctypes, recipes - ctypes: the number of cut-points per polygons. - recipes: the actual list of cut-points for each polygon.

classmethod validate_hash(*hashlist*)

Test if a hash represents a cut pattern that can be handled by the Geokernel object.

Parameters
hashlist – list of cell-hash

Returns
array of boolean

ctypeset

edgemap

facemap

nvtxface

class antares.utils.geomcut.geokernel.**TriKernel**

Kernel for triangle.

ctypeset = [(2, 'bi'), (3, 'tri')]

edgemap = array([[0, 1], [0, 2], [1, 2]])

facemap = array([[0, 1, 1, 1], [0, 2, 2, 2], [1, 2, 2, 2]])

nvtxface = array([2, 2, 2])

class antares.utils.geomcut.geokernel.**QuaKernel**

Kernel for quadrilateral.

ctypeset = [(2, 'bi'), (4, 'qua')]

edgemap = array([[0, 1], [0, 3], [1, 2], [2, 3]])

facemap = array([[0, 1, 1, 1], [0, 3, 3, 3], [1, 2, 2, 2], [2, 3, 3, 3]])

nvtxface = array([2, 2, 2, 2])

class antares.utils.geomcut.geokernel.**TetKernel**

Kernel for tetrahedra.

ctypeset = [(3, 'tri'), (4, 'qua')]

edgemap = array([[0, 1], [0, 2], [0, 3], [1, 2], [1, 3], [2, 3]])

facemap = array([[0, 1, 2, 2], [0, 1, 3, 3], [0, 2, 3, 3], [1, 2, 3, 3]])

nvtxface = array([3, 3, 3, 3])

class antares.utils.geomcut.geokernel.**PyrKernel**

Kernel for pyramids.

ctypeset = [(3, 'tri'), (4, 'qua'), (5, 'pen')]

edgemap = array([[0, 1], [0, 3], [0, 4], [1, 2], [1, 4], [2, 3], [2, 4], [3, 4]])

facemap = array([[0, 1, 2, 3], [0, 1, 4, 4], [0, 3, 4, 4], [1, 2, 4, 4], [2, 3, 4, 4]])

nvtxface = array([4, 3, 3, 3, 3])

class antares.utils.geomcut.geokernel.**PriKernel**

Kernel for prisms.

ctypeset = [(3, 'tri'), (4, 'qua'), (5, 'pen')]

edgemap = array([[0, 1], [0, 2], [0, 3], [1, 2], [1, 4], [2, 5], [3, 4], [3, 5], [4, 5]])

facemap = array([[0, 1, 2, 2], [0, 1, 4, 3], [0, 2, 5, 3], [1, 2, 5, 4], [3, 4, 5, 5]])

nvtxface = array([3, 4, 4, 4, 3])

class antares.utils.geomcut.geokernel.**HexKernel**

Kernel for hexahedra.

ctypeset = [(3, 'tri'), (4, 'qua'), (5, 'pen'), (6, 'hxg')]

```
edgemap = array([[0, 1], [0, 3], [0, 4], [1, 2], [1, 5], [2, 3], [2, 6], [3, 7], [4,
5], [4, 7], [5, 6], [6, 7]])

facemap = array([[0, 1, 2, 3], [0, 1, 5, 4], [0, 3, 7, 4], [1, 2, 6, 5], [2, 3, 7,
6], [4, 5, 6, 7]])

nvtxface = array([4, 4, 4, 4, 4, 4])
```

—

Class for geometric surfaces

Module defining classes for different types of geometric surfaces.

The class for geometric surface objects. The abstract class provides a common interface for any specific surface.

A surface is composed of 1 or more internal component.

Provided mono-component surfaces are:

- plane: defined by an origin point and a normal vector
- sphere: center point and radius
- cylinder: axis (origin point and direction vector) and radius
- cone: axis and apex angle

The only component in these surfaces encloses the whole surface.

Provided poly-component surface is:

- extruded polyline: sequence of coplanar points and extrusion vector

Each component in a polyline is an extruded segment, defined by 2 successive points.

The API exposes the following methods:

- partition_nodes:
  - Assign each point with a label that describe the relative position of the point to the surface.
  - The 3 partitions are FRONT, REAR, ON-SURFACE.
  - FRONT and REAR contains points on opposite sides that do not belong to the surface.
  - ON-SURFACE contains points that belong to the surface. These points are also assigned the corresponding internal component identifier.
- bind_crossedges:
  - For mesh edges whose 2 vertices belong to opposite partitions (FRONT and REAR).
  - Computes interpolation weights that defines the intersection point from the edges vertices.
  - Assign each edges with the identifier of the internal component the intersection point belongs to.
- normal:
  - Uses point coordinates and corresponding component id.
  - Returns vectors normal to the surface at given points, all of them oriented toward the same side of the surface. (the FRONT side)

class antares.utils.geomcut.geosurface.**AbstractSurface**

Abstract base class for geometric surfaces.

Declare two methods that derived classes must implement.

abstract bind_crossedges(*dict_cell_conn*, *node_label*, *xyz_coords*, *kernels*)

Abstract method.

Bind a set of edges to the surface. It consists in:

- assigning every edges with an internal surface id.
- computing interpolation weights for edge vertices.

abstract normal(*node_surf*, *node_coords*)

Abstract method.

Return a vector normal to the surface for each point in a set.

Parameters
node_coords – (x,y,z) coordinates of a set of points.

Returns
unit vectors normal to the surface at given points.

abstract partition_nodes(*xyz_coords*, *epsilon=1e-08*)

Abstract method.

Partition a set of points into 3 subsets:

- FRONT: one side of the surface
- REAR: the other side
- ON-SURFACE: points that are exactly on the surface (modulo an epsilon)

Parameters
xyz_coords – (x,y,z) coordinates of a set of points.

Returns
label of the partition a node belongs to

class antares.utils.geomcut.geosurface.**Plane**(*origin*, *normal*)

A parametric plane.

bind_crossedges(*dict_cell_conn*, *node_label*, *xyz_coords*, *kernels*)

Bind a set of edges to the surface.

It consists in:

- assigning every edges with an internal surface id.
- computing interpolation weights for edge vertices.

normal(*node_surf*, *node_coords*)

Return a vector normal to the plane for each point in a set.

Parameters
xyz_coords – (x,y,z) coordinates of a set of points.

Returns
unit vectors normal to the surface at given points.

partition_nodes(*xyz_coords*, *epsilon=1e-08*)

Partition a set of points into 3 subsets.

- FRONT: one side of the surface
- REAR: the other side
- **ON-SURFACE: points that are exactly on the surface**
  (modulo an epsilon)

Parameters
xyz_coords – (x,y,z) coordinates of a set of points.

Returns
label of the partition a node belongs to

class antares.utils.geomcut.geosurface.**Cylinder**(*origin*, *direction*, *radius*)

A parametric cylinder.

bind_crossedges(*dict_cell_conn*, *node_label*, *xyz_coords*, *kernels*)

Bind a set of edges to the surface.

It consists in:

- assigning every edges with an internal surface id.
- computing interpolation weights for edge vertices.

normal(*node_surf*, *node_coords*)

Return a vector normal to the cylinder for each point in a set.

Normals are directed to the outside of the cylinder.

Parameters
xyz_coords – (x,y,z) coordinates of a set of points.

Returns
unit vectors normal to the surface at given points.

partition_nodes(*xyz_coords*, *epsilon=1e-08*)

Partition a set of points into 3 subsets.

- FRONT: one side of the surface
- REAR: the other side
- **ON-SURFACE: points that are exactly on the surface**
  (modulo an epsilon)

Parameters
xyz_coords – (x,y,z) coordinates of a set of points.

Returns
label of the partition a node belongs to

class antares.utils.geomcut.geosurface.**Sphere**(*origin*, *radius*)

A parametric sphere.

bind_crossedges(*dict_cell_conn*, *node_label*, *xyz_coords*, *kernels*)

Bind a set of edges to the surface.

It consists in:

- assigning every edges with an internal surface id.
- computing interpolation weights for edge vertices.

normal(*node_surf*, *node_coords*)

Return a vector normal to the sphere for each point in a set.

Normals are directed to the outside of the sphere.

Parameters
xyz_coords – (x,y,z) coordinates of a set of points.

Returns
unit vectors normal to the surface at given points.

partition_nodes(*xyz_coords*, *epsilon=1e-08*)

Partition a set of points into 3 subsets.

- FRONT: one side of the surface
- REAR: the other side
- **ON-SURFACE: points that are exactly on the surface**
  (modulo an epsilon)

Parameters
xyz_coords – (x,y,z) coordinates of a set of points.

Returns
label of the partition a node belongs to

radius

class antares.utils.geomcut.geosurface.**Cone**(*origin*, *direction*, *angle*)

A parametric cone.

bind_crossedges(*dict_cell_conn*, *node_label*, *xyz_coords*, *kernels*)

Bind a set of edges to the surface.

It consists in:

- assigning every edges with an internal surface id.
- computing interpolation weights for edge vertices.

normal(*node_surf*, *node_coords*)

Return a vector normal to the cone for each point in a set.

Normals are directed to the outside of the sphere.

Parameters
xyz_coords – (x,y,z) coordinates of a set of points.

Returns
unit vectors normal to the surface at given points.

partition_nodes(*xyz_coords*, *epsilon=1e-08*)

Partition a set of points into 3 subsets.

- FRONT: one side of the surface
- REAR: the other side
- **ON-SURFACE: points that are exactly on the surface**
  (modulo an epsilon)

Parameters
xyz_coords – (x,y,z) coordinates of a set of points.

Returns
label of the partition a node belongs to

class antares.utils.geomcut.geosurface.**PolyLine**(*point_coord*, *axis*)

An extruded polyline.

bind_crossedges(*dict_cell_conn*, *node_label*, *xyz_coords*, *kernels*)

Bind a set of edges to the surface.

It consists in:

- assigning every edges with an internal surface id.
- computing interpolation weights for edge vertices.

Parameters

- **node_label** (*ndarray of size 'total number of mesh points'*) – label of nodes (which side of the surface)
- **dict_cell_conn** (*dict*) – subset of mesh cell connectivity. It includes cells that cross the surface.

normal(*node_surf*, *node_coords*)

Return a vector normal to the cone for each point in a set.

Normals are directed to the outside of the sphere.

Parameters
xyz_coords – (x,y,z) coordinates of a set of points.

Returns
unit vectors normal to the surface at given points.

partition_nodes(*node_coords*, *epsilon=1e-08*)

Partition a set of points into 3 subsets.

- FRONT: one side of the surface
- REAR: the other side
- **ON-SURFACE: points that are exactly on the surface**
  (modulo an epsilon)

Parameters
xyz_coords – (x,y,z) coordinates of a set of points.

Returns
label of the partition a node belongs to

axis

coords

—

Class for Tetrahedralizer

Subdivide 3D mesh cells into tetrahedra, or 2D mesh cells into triangles.

class antares.utils.geomcut.tetrahedralizer.**Tetrahedralizer**(*dim*, *connectivity*)

Build the tetrahedralisation of a mesh.

interpolate(*value*, *location*)

Returns
interpolated value for the tetrahedral mesh.

interpolate_cell(*value*)

Returns
interpolated cells values (order 0)

interpolate_node(*value*)

Returns
node values

property connectivity

Returns
tetrahedral connectivity.

Return type
CustomDict

property src_cell

Returns
mapping between new cells and parent cells.

—

Class for Cutter

This module provides a class for building the geometric intersection between a cell-based unstructured mesh and a parametric surface.

The resulting connectivity is built during object initialization. This object can next be used to interpolate mesh data at the surface.

An object that internally computes the intersection between a surface and a 3D unstructured mesh. It records information about intersection connectivity and interpolation operands / weights.

— Cut-Connectivity —

The intersection between mesh elements and the surface.

- Cut-nodes: The intersection points between the edge-based connectivity and the surface
- Cut-edges: The intersection segments between the face-based connectivity and the surface.

- Cut-cells: The intersection polygons between the cell-based connectivity and the surface.

The cut-connectivity is computed from, and is expressed as, a cell-based connectivity. This connectivity is exposed as an object read-only attribute, as long as cut-node coordinates.

— Interpolation —

The Cutter class exposes methods for computing values on the intersection from mesh values.

- Cut-nodes: values are interpolated from pairs of mesh-node values (weighted average).
- Cut-cells: values are inherited from parent mesh-cell values (copy).

— Algorithm —

The Cutter works as follow: - Find the list of mesh components that intersect the surface - Build intersection connectivity and record interpolation data

1/ Mesh Component intersecting the surface

- Partition nodes depending on their position to the surface:
  - ON-SURFACE partition: nodes that belong to the surface
  - FRONT partition: nodes that are "in front of" the surface.
  - REAR partition: nodes that are "behind" the surface.
  - FRONT and REAR partition denote opposite sides and depend on the surface parameters only.
- Find cells intersecting the surface using node partitioning:
  - Cells with at least 2 vertices on opposite sides (FRONT and REAR)
  - Cells with a face belonging to the surface (ON-SURFACE face)
    - Such cells may be paired together, sharing a common face.
    - If so, only 1 cell per pair is selected.
- In selected cells, using node partitioning, find:
  - ON-SURFACE vertices
  - Cross-edges: edges with 2 vertices and opposite sides (FRONT and REAR)

2/ Interpolating cut-nodes (intersection points)

- ON-SURFACE nodes are also cut-nodes.
- 1 cross-edge gives 1 cut-node. Interpolation is made from its 2 vertices.
- Records pairs of input mesh nodes, and the corresponding weights.
- Also build a mapping between cell components and cut-nodes, used in next step. - cell components are the union of cell vertices and edges

3/ Building cut-connectivity

- 1 selected mesh cell <=> 1 cut-cell (intersection polygon).
- The union of ON-SURFACE vertices and cross-edges forms a unique cut-pattern.
- Use the cut-pattern to build intersection polygons.
- Records a cell-based connectivity composed of polygons whose vertices are cut-nodes' indices.

4/ Finalisation:

- Polygon normals are oriented randomly. - Reorganise vertices so that normals are all oriented in the same consistent direction.
- Some cells may lead to polygons with up to 6 vertices, not handled by Antares. - Such polygons are recursively split, until the output contains only triangles and quads.

— Architecture —

The algorithm is executed during the cutter initialisation. It takes as arguments:

- mesh description: cell-based connectivity organized by cell types (dictionnary type) and node coordinates (single array).
- surface descrption: a single object that provides methods for partitioning mesh elements and computing interpolation weights.

It also internally relies on a set of GeoKernel objects that provides:

- mapping for building faces and edges from canonical cell vertices.
- pre-computed recipes for building intersection polygons from canonical cell components.

class `antares.utils.geomcut.cutter.`**`Cutter`**(*dict_cell_conn*, *node_coord*, *surface*, *ctypedict={2: 'bi', 3: 'tri', 4: 'qua'}*)

Compute the geometric intersection between a surface and a mesh.

The intersection connectivity is built during initialisation. The instanciated object can then be used as a simple wrapper for interpolating data on the surface.

This class accepts point coordinates expressed in any 3D system.

`interpolate`(*value*, *location*)

Get the interpolated value.

Returns
interpolated value on the cut.

`interpolate_cell`(*value*)

Compute Interpolated value at the node of the cut.

Parameters
`value` – 1D array of cell-values to interpolate.

Returns
dict with cell-values interpolated at the cut plane.

`interpolate_node`(*value*)

Compute Interpolated value at the node of the cut.

Parameters
`value` – 1D array of node-values to interpolate.

Returns
node-values interpolated at the cut plane.

`property cell_conn`

Returns
a dictionary with keys for cell types, containing the cell-base connectivity of the cut.

—

When using the treatment *cut* with the option *line_points*, then the following function can be used to get the values *points* of many slices (or sections) named *section_names* from a file *sections.json*.

```
from antares.utils.CutUtils import parse_json_xrcut
section_names, points = parse_json_xrcut('sections.json')
```

antares.utils.CutUtils.**parse_json_xrcut**(*fname*)

Read points of a (x, r) line.

Json file format: { "name of section 1": { "type":"x1r1x2r2", "x1r1x2r2": [1., 1., -4., 6.] }, "name of section 2": { "type":"x1r1x2r2", "x1r1x2r2": [7., 1., 6., 1.] } }

Parameters
fname (*str*) – json filename that contains the definition of (x, r) lines. Their units must be the millimeter.

Returns
Names of sections.

Return type
list(str)

Returns
Coordinates (x, r) of points of sections in (x, r, theta) system. Their units is the meter.

Return type
list(tuple(float,float))

5.12.4 Duct modes

antares.utils.DuctUtils.**compute_duct_modes**(*r_duct*, *nb_mode_max*)

Compute eigenvalues of a duct.

Solve the eigensystem of the convected wave equation. The location of the zeros for the function associated with the eigensystem correspond to the eigenvalues for the convected wave equation. The solution to the Bessel equation with boundary conditions applied yields a system of two linear equations.

$$\frac{d^2\psi_m}{dr^2} + \frac{1}{r}\frac{d\psi_m}{dr} + \left(\alpha^2 - \frac{m^2}{r^2}\right)\psi_m = 0$$

with $\alpha^2 = (\omega + Mk)^2 - k^2$

with the boundary condition at hub and tip radii: $\left.\frac{d\psi}{dr}\right|_{R_h,R_t} = 0$, for $R_h \leq r \leq R_t$

The general solution of this o.d.e. is:

$$\psi(r) = AJ_m(\alpha r) + BY_m(\alpha r)$$

with J_m and Y_m first and second species Bessel function.

Applying BC leads to:

$$Gx = \begin{bmatrix} J'_m(\alpha R_h) & Y'_m(\alpha R_h) \\ J'_m(\alpha R_t) & Y'_m(\alpha R_t) \end{bmatrix} \begin{bmatrix} A \\ B \end{bmatrix} = 0$$

which has non-trivial solutions as long as the determinant is zero.

$$det(G) = f(x) = J_m(\alpha R_h)Y_m(\alpha R_t) - J_m(\alpha R_h)Y_m(\alpha R_h) = 0$$

This function is solved using chord method.

Once the eigenvalues :math:`alpha has been computed, contants A and B are assigned one of the following two sets of values:

$$\begin{cases} A = 1 \\ B = -\frac{J'_m(\kappa_{mn} r_D)}{Y'_m(\kappa_{mn} r_D)} \end{cases} \quad \text{or} \quad \begin{cases} A = -\frac{Y'_m(\kappa_{mn} r_D)}{J'_m(\kappa_{mn} r_D)} \\ B = 1 \end{cases}$$

Of these two sets of values, the one for which $\left(A^2 + B^2\right)$ is the smaller value is chosen. If $\left(A^2 + B^2\right)$ is the same for both, then the second set is picked.

with:

x (float): coordinate

m (int): azimuthal wave number

R_h (float): inner (hub) radius of a circular annulus

R_t (float): outer (tip) radius of a circular annulus

Parameters

- **r_duct** (`[float, float]`) – the hub radius and the tip radius
- **nb_mode_max** (`[int, int]`) – the maximum azimuthal mode order and the maximum radial mode order to compute

Daroukh, M. "Effect of distortion on modern turbofan tonal noise", PhD thesis, Toulouse University, 2017

Daviller, G. "AITEC2: Theoretical & Technical Report on Dedicated Acoustic Treatment in Antares", 2018

Moinier and Giles, "Eigenmode Analysis for Turbomachinery Applications", Journal of Propulsion and Power, Vol. 21, No. 6, 2005.

CHAPTER

SIX

CITATIONS & ACKNOWLEDGEMENTS

Here's an example of a BibTeX entry. Please consider citing the library antares in a paper or a presentation with:

```
@MISC{antares,
author = {{antares Development Team}},
title = {{Antares Documentation Release 2.2.0}},
month = {April},
year = {2024},
url = {https://cerfacs.fr/antares/}
}
```

You may also refer to antares in the acknowledgments with:

```
... was performed using the python library antares (release 2.2.0, https://www.cerfacs.
↪fr/antares)
```

If expertise was provided, the following acknowledgement may be added:

```
We thank ... in Cerfacs for assistance with the python library antares (release 2.2.0,␣
↪https://www.cerfacs.fr/antares)
```

CHAPTER

SEVEN

RELEASE NOTES

Antares changelog, based on keepachangelog.com.

[2.2.0] - 2024-04-15

Added - Reader Ensight: read zone names. - Reader bin_vtk: read tensor variables. - Reader hdf_cgns: add base_subregion keyword. - Reader hdf_avbp: read AVBP isosurfaces. - Reader hdf_labs: read variables at cell. - Reader hdf_cngs: Add support for CGNS4 files - Reader pycgns: Add support for CGNS4 files - Treatment ExtractBounds: add support for polyhedral and polygonal bases. - Treatment FWH: show an estimation of the remaining time. - Treatment translation: New treatment added to translate bases. - Treatment scaling: New treatment added to scale bases. - Treatment rotation: New treatment added to rotate bases. - Treatment spectgram: New treatment to compute a spectrogram. - Treatment DFT: accept 're/im' and 'phi/mod' as type.

Changed - Treatment UnwrapProfil: Improve polyline algorithm performance.

Fixed - Reader hdf_labs: correctly identify zones when they contain underscore in their names. - Reader hdf_antares: store zone shared variables. - Reader hdf_cgns: do not read variables that don't have data. - Reader prf: Strip trailing newline character from instant names. - Treatment gradient: Add container variables when their name is longer than 1 char.

[2.1.0] - 2023-10-15

- TreatmentFWH: Added 'modify_surface' keyword.
- TreatmentFWH: Added 'volume_base', 'volume_meshfile', and 'volume_datafile' keywords.
- TreatmentFWH: Added 'modify_volume' keyword.
- TreatmentFWH: Added 'pressure_variable', 'velocity_variables', 'density_variable' keywords.
- TreatmentFWH: Added 'pressure_equation', 'density_equation', 'velocity_equations' keywords.
- TreatmentFWH: Added 'quadrupole_term' keyword.
- TreatmentFWH: Added 'eddy_convective_velocity' keyword.
- TreatmentFWH: Added 'derivative_order' keyword.
- TreatmentFWH: Added 'verbose' keyword.
- UtilsFWH: Added extract_converged_signal method.
- UtilsFWH: Added find_periodicity method.
- UtilsFWH: Added add_fwh_results method.
- HDF_Antares: Write and read base attributes.
- TreatmentFWH: Treatment information written in output.
- TreatmentFWH: Added 'start_propagation_at' keyword.

- TreatmentFWH: Added 'end_propagation_at' keyword.
- TreatmentFWH: Added 'initial_time' keyword.
- ProLB: Reader compatible with version 3.
- EquationManager: Add support for scientific notation.
- ParallelController: The environment variable ANTARES_NO_MPI controls if mpi4py is loaded.

Changed

- TreatmentFWH: Removed 'LABSCleaning' keyword.
- Refactor the unit test of TreatmentTurboGlobalPerfo.

Fixed

- Fix Isentropic Mach number in avbp equations.
- Fix unpack function to get writeable arrays.
- Fix time attribute in FWH treatment.
- Fix duplicated variables from shared instant in node_to_cell process.
- Improve FWH documentation.
- Fix the surface weighted average in the treatment thermo1.
- Fix SPL and PWL equations in AcousticPower treatment.
- Fix compatibility with new cgnslib.
- Fix type cast problem in utils/geomcut/cutter.py.
- Fix do not read zones with no name in ReaderPyCGNS.

[2.0.0] - 2023-04-15

Added

- MultiThreaded Treatment Ccut
- Spectral Proper Orthogonal Decomposition
- Impose double precision data type for VTK points in clip and cut treatments
- Enable tuple of 3 floats for line_points in treatment cut
- Add an early logger to check module load errors
- New option to read HDF CGNS file in parallel as if it was serial
- Treatment Isosurface: output polygons when using polyhedral meshes. Option for triangles.
- Reader HDF5-CGNS: accept multiple NGON Elements_t nodes.
- Reader HDF5-LABS: read files with surface forces
- Read node DonorPatch/PointRangeDonor for GridConnectivity_t in reader HDF5-CGNS

Changed

- Remove module initializers for python 2
- Remove try/except for python2/3 compatibility
- Remove import occurrences of module future
- Remove occurrences of the module __future__

- Stop printing the antares header by default
- Output probe positions in the treatment PointProbe
- Writer ASCII tecplot: output all element types of one zone in a single file
- UserInterface class derives from UserDict.
- Base Class derives from IndexedUserDict class.
- Datasets Class derives from IndexedUserDict class.
- Instant Class derives from UserDict class.
- CustomDict Class derives from IndexedUserDict class.
- Family Class derives from IndexedUserDict class.

Fixed

- Fix VTK operations on 2D unstructured grids with VTK>=9
- Fix azimuthal average for a case with a y-rotation axis
- Impose 64 bits integers in face2elt_connectivity
- Impose 64 bits integers in reader pycgns
- TreatmentCut: recover polyline behavior as spline behavior
- TreatmentCell2Node: delete temporary variable for hybrid grids
- Fix sorted list of families in get_family
- Fix get_location when only cell variables in instants
- Fix slicing of shared variables when only shared instants
- Fix writer HDF5-CGNS for unstructured mesh with only variables located at cell centers
- Fix fortran vs C based indexing in readers pyCGNS and HDF5-CGNS
- Fix names of gradient variables when using vtk.
- Deactivate slicing check in reader pyCGNS.

[1.20.0] - 2022-10-15

Added

- TreatmentFWH: rotating surfaces
- TreatmentBoundaryNormal: compute the outward normal vector on the boundary conditions of 3D configurations
- WriterHdfCgns: write face-based connectivity
- ReaderBinaryFluent: read single-precision field data
- Treatment CellNormal compatible with face-based connectivity and polygons
- Treatment Cell2Node: compatible with face-based connectivity and polygons and polyhedra
- Treatment Merge: compatible with face-based connectivity and polygons and polyhedra
- Method node_to_cell compatible with face-based connectivity, and polygons and polyhedra
- ReaderPyCgns/ReaderHdfCgns: read BC with face-based connectivity and store face-based connectivity
- ReaderPyCgns: read BC with face-based connectivity and store vertex-based connectivity

- TreatmentMisOnBlade: parallel version
- TreatmentUnwrapline: use in parallel environment
- Reader CSV: compliant with cantera format
- TreatmentSpanWiseAverage
- ReaderHdfAntares: add new 'format' key to select the antares HDF5 format '2015' or '2022'
- WriterHdfAntares: add new 'format' key to select the antares HDF5 format '2015' or '2022'
- Improve equations for AVBP computations
- File Cache System

Changed

- ReaderHdfCgns: read boundary conditions with face-based connectivity
- Treatment Merge: the input base is now unchanged if nothing has to be done
- Refactoring of the PRF reader

Fixed

- Set omega and pitch as constants in all Thermo Treatments
- Fix the Q-criterion formula in the treatment gradient

[1.19.0] - 2022-04-15

Added

- TreatmentMeridionalView: option to activate LE/TE detection
- TreatmentAzimuthalAverage: azimuthal average with multiple processes
- TreatmentWakeAcoustics: new option to choose a variable to detect the wake
- TreatmentCellNormal: compute the normal vectors of 1D or 2D mesh elements
- TreatmentBl enabled on Windows platforms
- TreatmentGradient: Lambda_2 and Lambda_ci criteria computation
- Writer HDF-CGNS: write surface elements for unstructured grids. Fix for family
- Reader HDF-CGNS: read abutting 1-to-1 connectivity for unstructured grids
- Reader HDF-CGNS: read element-based connectivity of segments for unstructured grids
- TreatmentMerge: remove degenerated segments if duplicated points are removed
- Python 3 support for extension library ngon
- Reader HDF AVBP: option to read given groups or variables
- Writer HDF AVBP: option to write variables in given groups
- Support VTK>=9 for unstructured grids
- Reader PyCGNS: read unstructured grids with face-based connectivities
- TreatmentThermo7/TreatmentThermo7TimeAverage: new key to introduce absolute velocity formulation
- TreatmentPSD: add new 'window' key to set the window function
- TreatmentPSD: add new 'scale_by_freq' key to set if PSD should be scaled by the sampling frequency
- TreatmentPSD: add unit test

- TreatmentPSD: Improve documentation

Changed

- Remove KNOWN_CONSTANTS
- Reader Binary Fluent: improve CPU performance
- TreatmentCut: directly return the cutter output for the type polyline
- Reader Tecplot Binary: remove option use_ztitles. Always set the zone names from the tecplot zone titles
- Reader HDF Labs: allow moving surface
- do not use the treatment merge inside HDF AVBP writer anymore
- Axis parameter format from str to list of vector components for cylinder/cone cut/clip

Fixed

- Reader HDF-CGNS: fix array shape for unstructured grid
- Fix Base rename_zones method to handle families of type Zone
- TreatmentDuplication: fix donor boundary name for 1-to-1 abutting connectivity
- Writer HDF-CGNS: check the validity of family name
- remove useless boundary condition made of interior faces in ReaderBinaryFluent
- Fix the construction of element-based hexahedra from the face-based connectivity
- Set the data type in numpy arange in function face2elt_connectivity (python 3 on windows platform)
- TreatmentCut: Fix empty structure in get_vtk_surface for polyline/spline due to vtkProbeFilter
- Fix integer division for python 3 in treatment unwrapline

[1.18.0] - 2021-10-15

Added

- Reader HDF Labs: lazy loading
- TreatmentPODtoTemporal: reconstruct time signals from POD modes
- TreatmentCut: add resolution key for type=spline
- Pass options with kwargs in reader, writer, and treatment constructors
- ReaderHdfCgns: read ZoneIterativeData_t, modify reading of BaseIterativeData_t
- Refactoring of Treatments Creation: create factory instead of proxy

Changed

- TreatmentMerge: enable when instants contains different numbers of variables
- Replace the implementation of base.cell_to_node with the treatment TreatmentCell2Node
- TreatmentPOD: add the parameter 'variables'
- TreatmentThermoLES: add the gyration angle

Fixed

- TreatmentClip: fix for multiple instants
- TreatmentCut: fix for revolution type
- TreatmentMerge: fix duplicate detection when only shared variables

- TreatmentMerge: fix location of variables
- TreatmentGradient: fix indentations for the divergence operator
- WriterHdfCgns: fix attributes with float, integer types
- computer: fix when formula include variables from standard instant and shared instant

[1.17.0] - 2021-04-15

Added

Changed

- TreatmentClip: type=revolution does not compute the cylindrical coordinates anymore
- TreatmentPointProbe: enable unstructured grids and many points in one zone
- Unify documentation and web site content

Fixed

- Writer HDF CGNS: write base attributes with the correct type
- Reader HDF CGNS: Read UserDefinedData_t nodes under the CGNSBase_t node
- Flip boundary data during family slicing
- Method stats of Base to handle correctly variables located at cells
- Mesh orientation in case of a degenerated axis
- Mesh orientation in case of a degenerated axis
- TreatmentCell2Node: fix for unstructured grids
- Skip Dataset attributes that cannot be deepcopied
- Reader HDF antares: h5py attribute types change in version > 3

[1.16.0] - 2020-10-15

Added

- Reader HDF CGNS: read Zone SubRegions
- Treatment PointProbe
- Base: method to rename zones
- Reader VTK: structured grid with '.vts' extension
- Equations for AVBP simulations (thermodynamic tables)
- TreatmentMisOnBlade: compute the isentropic Mach number on a blade surface
- TreatmentLES: thermodynamic average for LES computations
- TreatmentThermo1D: radial profile of thermodynamic averages
- TreatmentUnwrapBlade: unwrapping of blade surface on a plane
- TreatmentUnwrapProfil: unwrapping of blade profile on a line
- TreatmentPlotContouring: plot contour graph

Changed

- Reader HDF Antares: improve CPU performance for large data
- Reader HDF CGNS: remove key change_topo_name

- Treatment Cell2node: specific processing of surface and normal vectors
- Writer Binary Tecplot: dump base with zones that do not contain the same variables
- TreatmentUnwrapline: enable many closed curves.
- TreatmentAcut: simplify the User Interface when reusing previous cutters
- TreatmentUnstructure: set the connectivity to respect the orientation of normals given by the structured mesh
- Reader Tecplot Binary: option use_ztitles to set the zone names from the tecplot zone titles

Fixed

- Flip data of boundaries when slicing a structured base with a family.
- Ensure that the equation computing system complies with the lazy loading.
- Deepcopy in method set_coordinate_names of Base class.
- Deepcopy of Base object.
- Deserialization of Boundary object.
- Reader bin_fvuns: fix settings of boundary conditions in the fieldview reader to comply with the current architecture
- Treatment Acut: fix for shared coordinates
- Reader hdf_antares: fix python 2/3 bytes/str for element names and location names

[1.15.0] - 2020-04-15

Added

- Reader VTK (vtu and tensors)
- Reader Fluent (case and dat files)
- Treatment Meridional Line
- Treatment Meridional View
- Treatment Thermo Geom (geometrical computation of treatment Thermo7)
- Reader VTK binary: accept multi-element for unstructured grids
- ReaderHdfCgns: add an option to follow links in a HDF5 CGNS file
- Writer PLY

Changed

- Refactoring of treatment hH (extract meridional line and view)
- Refactoring of treatment Thermo7 (extract geometrical computation)

Fixed

- Treatment Cut: create a shared instant in the output base if the input base contains a shared instant
- Treatment Merge: attributes of the input zones/instants are cleared.
- Reader Tecplot Binary: option 'shared_mesh' does not work with <instant> tag and 2D
- Treatment Duplication and Families when Zone attributes 'nb_duplication' are different
- Reader PyCGNS: fix slicing of Boundary object

[1.14.0] - 2019-10-15

Added

- Treatment Radial Modes
- Treatment Azimuthal Modes
- Treatment Acoustic Power
- Coprocessing option in treatment h/H
- Read velocity RMS components in format hdf labs
- Coordinate accepted as isovariable in Treatment Isosurface
- Polyhedral elements
- Polyline cut with VTK Treatment Cut
- Option to read specific FlowSolution_t data structures in ReaderHdfCgns
- Examples for TreatmentAcut
- Utility method to reorient boundary data coming from files to base extracted from boundaries
- Reorient faces when extracting boundaries on structured meshes

Changed

- Treatment DMD 1D with complex outputs
- Refactoring of Treatment.py
- Refactoring of CustomDict and AttrsManagement
- Remove non standard help mecanism
- Refactoring to use the logging module consistently
- ReaderPrf: accept headers with ## (third-party code version > 2.5)
- Accept many instances of readers and writers
- Messages handled by the logging module

Fixed

- Fix attribute reading in reader hdf cgns
- Fix writer hdf cgns so as not to modify the input base
- Fix parameters section in writer hdf avbp
- Rotate vectors directly in TreatmentChoroReconstruct
- Compute_coordinate_system: check that coordinate names are in the shared instant
- Write_zone unstructured in Writer HDF CGNS
- Reader HDF LaBS for python 3 and using tag <zone>
- Add writers avbp, gmsh, and ansys in documentation

[1.13.0] - 2019-04-15

Added

- Add unittest for BL with result checks
- Parallelism with MPI

- Option to not process boundary condition in Treatment Unstructure
- Enable to give primitive variable names in treatment BL
- Number of points in the boundary layer in surface output file in treatment BL
- Check valid profiles before writing files in treatment BL
- Add family name in BC when reading AVBP files
- Replace index by family names in zone names of output bases from treatment BL
- Accept list for families option in treatment BL (marker automatically assigned)
- Criteria on isentropic Mach number in Treatment Boundary Layer
- Treatment for reading Jaguar solution
- Treatment Unstructure (used for base.unstructure())
- Treatment Dmdtotemporal (Temporal Reconstruction from Dynamic Mode Decomposition)
- '*.vtp' (XML PolyData) VTK reader
- Function delete_variables in Base

Changed

- Multizone Treatment BL
- Add special vtk2instant for tetrahedral elements only (better CPU performance)
- Use another clipper in the Clip Treatment (better CPU performance)
- Refactoring of Treatment Thermo*
- Stop embedding documentation in the package
- Treatment merge now handles boundary conditions
- Move thermodynamic average treatments in turbomachine directory
- Allow multiple blades in meridional view in hH treatment

Fixed

- Fix cone and cylinder issues in Treatment Clip with refactoring
- Fix origin issues in Treatment Cut for cone
- Fix axis and origin issues in Treatment Cut for cylinder
- Fix CrinkleSlice with the new Treatment Unstructure
- Fix Threshold with the new Treatment Unstructure
- Avoid extra works when same marker for different families in treatment BL
- List profile_points can be void in treatment BL
- Reader HdfCgns and family links
- Clip Treatment with unstructured mesh and shared instant
- Misusage of connectivity for unstructured meshes fixed in interpolation treatment
- Issue with shared connectivity when slicing (Bug #1281)
- Stable computation of local hh in thermo averages
- Treatment hH due to change in zone __copy__ and Base __get_family__

[1.12.0] - 2018-10-15

Added

- New Cell2Node Treatment, able to fully compute all contribution through multi zone edges, and apply periodicity
- method to build rotation matrix from angles

Changed

- add periodicity storage member in Boundary object
- add periodicity extraction for PyCGNS and HdfCGNS readers

Fixed

- PyCGNS reader: fix bnd slicing + set correct shape in instant

[1.11.0] - 2018-07-10

Added

- Check duplicate names in blockName UserDefineData CGNS node in ReaderHdfCgns
- Read face-based connectivity if also cell-based one in ReaderHdfCgns
- Read BCData_t nodes in ReaderHdfCgns
- Add multiple isosurfaces at the same time in Treatment Isosurface
- Add Treatment MultiChoroChronic Asynchronous
- Add Treatment MultiChoroChronic Synchronous
- Add boundary condition connectivity when reading face-based connectivity in Reader HDF CGNS
- Add datasets to boundary conditions
- Reader for in memory pyCGNS/pyTree objects
- Add Treatment Tetrahedralize
- Add Treatment Cut without VTK
- Add families in results of the boundary layer treatment
- Changelog
- Accept exponents D and d in Tecplot ascii reader
- Add unit test for reader/writer gmsh
- GitLab Continuous Integration
- Add instantaneous entropy averaging
- Add Treatment for basic mono-flux parametrization between hub and shroud
- Add treatment to get LE/TE in turbomachinery configuration
- Add GMSH Writer
- Add __copy__ method to Base class
- Shallow copy of connectivity in Instant initialization

Changed

- Remove unused lazy loading from writer column
- Stop supporting vtk < 6 in treatment isosurface

- New conception of Zone that inherits from Datasets
- Interpolation now modify the target base
- Improve threshold doc.
- Improve pod doc.
- Create a default family with the BC name if family not given in Reader HDF-CGNS
- Modify the logger behavior
- Update ReaderHdfCgns.py
- Modify method unstructure in Zone

Deprecated

Removed

Fixed

- Fix write 2D HDF5 AVBP mesh
- Fix reader binary tecplot when using option 'base' (addition of data to an existing base)
- Fix glitches in surfaces from treatment Acut (cut without vtk)
- Fix boundary layer treatment (check tri and qua)
- Fix compute bounding box with shared only
- Fix crash when writing empty base in cgns.
- Fix the reader tecplot on zone name when using the tag <instant>
- Fix open mode to 'rb' only due to windows os in the reader formatted tecplot
- Fix unit tests using NamedTemporaryFile from tempfile module
- Fix bug on windows os (transpose was omitted)
- Fix zone _parent when using family slicing

1.10.0 (October 24, 2017)

Added

Contents of release 1.10.0 / changes wrt 1.9.0

The progress bar is new. It relies on tqdm[142]. If you do not want to get this package, then you recover the old Antares 1.8.2 progress bar.

More details can be found at 1.10.0[143].

- Treatment Boundary Layer Extraction (Feature #1526)
- Treatment Ffowcs Williams & Hawkings Analogy
- Reader for Lattice-Boltzmann solver LaBs or ProLB: PRF format and XDMF output files (Feature #1869)
- Treatment for initialization of test case: shear layer (Treatment InitShearLayer)
- Writer STL (Feature #1724)
- Reader HDF CGNS reads and converts NGON_n elements into canonical elements

[142] https://github.com/tqdm/tqdm
[143] https://ant.cerfacs.fr/versions/66

- Treatment FFT for complex signals
- TreatmentGradient on 2D surfaces made of triangles (Feature #1786)
- Rotation in both negative and positive ways in Treatment Duplication (Feature #1741)
- Boundary class exposed at high level (Feature #1666)
- New Loggers (using logging module) (Feature #1743)
- Add many option to Writer HDF CGNS (Feature #1739, Bug #1950)
- Introduction of attrs in the class Family (Bug #1950)

Changed - New progress bar based tqdm if available. Otherwise, a simple progress bar is provided. - Reduce memory consumption of ReaderHdfAvbp - Reader 'hdf_cgns' can now keep the original names - Simplify function read_variable in ReaderV3DFormated - Many modifications in ReaderHdfavbp/WriterHdfavbp - Add method clear() for the custom dictionary (CustomDict) - Refactoring of Reader HDF CGNS (Bug #1687, Feature #1540) - Import zone names in Tecplot readers (Feature #1938) - Write parameters from Family in writer HDF CGNS - Write base with only shared instant in writer HDF CGNS - Write family names of zones in writer HDF CGNS - Refactor Boundary class - Merge branch 'feature/reader_xdmf_labs' into develop - Merge branch 'feature/grad2D' into develop - Merge branch 'feature/fft_complex' into develop

Fixed

- Fix file.seek on Windows platform
- Read every n files (Feature #1632)
- Enable more than one tag instant '<instant>' in the filename
- Fix use of tag <instant> with the hdf_antares reader
- Pass correctly families on new base issued from Base.get_location (Bug #1755)
- Reading an unstructured CGNS file (Bug #1660)
- Improve base slicing (families are correctly handled)
- Fix variables in writer due to Igor reserved keywords
- antares/core/DefaultGlobalVar has been removed; Check antares/core/GlobalVar and antares/core/Constant

1.9.0 (February 10, 2017)

Added

- POD Treatment (Proper Orthogonal Decomposition)
- Writer HDF5 AVBP (Feature #1173)
- Reader (Feature #1486) and Writer CSV (Feature #1487)
- Read every n file (Feature #1632)
- Shortcut to print a Base in a compact and fancy manner (Feature #1624)
- Enable shared mesh and connectivity in Ensight Reader with a user's key

- Use matplotlib for TreatmentPlot
- New progress bar
- Improve overall documentation, and online help for Readers, Writers, and Treatments
- Add memory estimate at each internal Antares print statement
- TreatmentChoroReconstruct: looking for an example and documentation (Document #1339)
- **Remove method delete_variables from Base </br>**
  New default dtype=float32 for Writers
- **Modify base.rename_variables (change coordinate_names accordingly) </br>**
  Modify writerHdfCgns (with different types of location for solution nodes) </br> Add equations from a user file </br> Warning if no variables to treat in the gradient treatment
- Modify stats.discrete_xxx to allow performing xxx operations even if NaN are in the arrays
- **Add rhou, rhov, rhow, rhoE in VARIABLE_NAMES </br>**
  Remove roe (confusing with internal energy)
- Fix bug in ReaderInstant and WriterInstant (add call to __init__())
- Add thermodynamic averages based on surface or massflow rate
- Remove warning message in the formated tecplot reader
- Minor change in bin/fast… tools to plot help message
- **New model for LIST_KEYS formating (longer lines possible) </br>**
  Introduce 'example' key in LIST_KEYS dedicated to give some explanation for users </br> Introduce optional dependencies at the top of module (use of decorators)
- **Refactor gradient computation (Instant, TreatmentGradient, GeomUtils) </br>**
  Comments for documentation
- **Remove set_coordinate_system function (now do base.coordinate_names =) </br>**
  In base.compute_coordinate_system, replace substring 'actual' by 'current' for arguments
- **mlab.psd not correctly set in TreatmentPsd.py (Bug #1622) </br>**
  Pad default value changed to 1 </br> Documentation updated
- Interpolation treatment: enable different instant names between the source base and the target base (Bug #1338)
- TreatmentCut: memory mode broken (Bug #1563)
- Change import module mlab in TreatmentPsd.py (Bug #1583)
- Base unstructure with 2D mesh and with shape of length 3 (Bug #1523)
- ReaderVtk with <zone> tag (Bug #1522)
- Reader Vtk: shape mismatch when using tag <zone> (Bug #1553)
- TreatmentCut with only shared variables (Bug #1552)
- Base slicing with variables, and shared shape (Bug #1519)
- Python / ImportError: Import by filename is not supported (Bug #1575)

1.8.2 (July 19, 2016) - Treatment Cut: type 'revolution' (Bug #1537)

1.8.1 (July 05, 2016) - fix bug in ReaderInstant and WriterInstant (add call to __init__()) (Bug #1524)

1.8.0 (June 29, 2016)

- **Store coordinate names as attribute in class Base (Feature #1433) </br>**
  Default coordinate names are gone. If you get "TypeError: object of type 'NoneType'" related to 'coordinates' in some treatment, then consider using either the key 'coordinates', or 'base.set_coordinate_names()', or "base.coordinate_names=['x','y','z']"
- Reader Fluent (Feature #1420)
- Reader NetCDF (Feature #1357)
- Reader Ensight (Feature #1325)
- **new instant_regex (interval of integers + leading zeroes) </br>**
  can now use: </br> reader['instant_regex'] = (1, 3, 4) or </br> reader['instant_regex'] = (10, 33) </br>
- **writer hdf_cgns: write solutions at nodes and cells </br>**
  reader bin_tp: read face_based connectivity (but do not use it)
- add a method (base.rename_variables) to rename variables (Feature #1501)
- Treatment Cut: multiple cuts in a single execute() statement (Feature #1500)
- Treatment Cut: avoid useless zones as cheaply as possible (Feature #1499)
- Treatment Cut: option not to triangulate the surface systematically (Feature #1494)
- Treatment Cut with Splines (Feature #1492)
- add a method (compute_coordinate_system) to compute the cylindrical coordinate system from the cartesian coordinate system
- add a method (delete_variables) to remove variables from base, zone, or instant
- Treatment for initialization of test case: channel flow (Feature #1483)
- **TreatmentMerge unstructured bases made of triangle elements in a single-zone base (Feature #1235) </br>**
  Treatment CrinkleSlice </br> cell_to_node for unstructured grids </br> Merge with sorting variables renamed into UnwrapLine </br>
- **Thermodynamic Average Treatment (Feature #1308) </br>**
  Turbomachine Performance Treatment (Feature #1311)
- **Change API of Readers and writers (Feature #1505) </br>**
  change Reader/Writer constructor. </br> Now give the value of the old key 'file_format' as argument. </br> E.g. Reader('bin_tp') instead of reader['file_format'] = 'bin_tp'
- refactoring of TreatmentLine

- **treatment cut, 2D meshes with zone detection </br>**
  topology readers, initialize attributes
- introduce path finding process for external treatments
- Shorcuts/compound scripts/commands for users in a hurry (External #1497)
- modify treatmentCut for revolution type
- refactor TreatmentCut (only one cutter.Update() with shared coordinates)
- source and target bases can now have different coordinate names in the interpolation treatment
- change key 'value' into key 'position' especially in turbomachine treatments
- **introduce routines to change default names (base, zone, instant).**
  modify HDF CGNS writer accordingly.
- accept files with 2-node lines in Gmsh Reader
- add documentation about origin in the cut treatment
- change clip into threshold in examples
- Family are now ordered properly instead of alphabetical order (Bug #1353)
- Writer Binary Tecplot and memory mode (Bug #1496)
- Create Gridline without shared instant (Bug #1435)
- Zone.is_structured() based on shared_instant is confusing (Bug #1424)

1.7.0 (January 19, 2016)

- Reader and Writer for 'hdf_antares' format (and writer of xdmf file associated) (Feature #1289)
- Gradient, Integration, Flux treatments
- Base functions to compute cell volumes and cell normals
- allow to use sophisticated delete on Base/Zone/Instant (using slicings, lists and indices like in the get_item)
- Reader and Writer for matlab format (Feature #1332)
- ReaderTecplotBinary.py: handle files containing AuxData (Feature #1351)
- Add-ons for Feature #1196 and Feature #1194
- WriterHdfCgns: add an option to avoid the use of links (Feature #1320)
- ReaderHdfCgns: read links in all circumstances (Feature #1323)
- Add-ons (concatenation of connectivities, and periodicity) for hdf_cgns reader and writer (Feature #1274)
- still Take into account rind cells in the hdf_cgns writer (Feature #1276)
- Decimation of a mesh (Feature #1314)
- Return a Null Base for Base.get_location (Feature #1291)

- Insert capabilities coming from the tool pyCROR (Feature #1266)
- Gridline treatment for cell-centered data (Feature #1196)
- Merge treatment for cell-centered data (Feature #1195)
- Probe treatment for cell-centered data (Feature #1194)
- Enable multi-space around equality operator in fmt_tp reader (Feature #1273)
- Reader V3D Formatted (Feature #1277)
- Add-ons for topology writer (Feature #1275)
- Package system management (Feature #1279)
- Face-based connectivity computation for elsA solver (Feature #1283)
- Writer HDF5 CGNS (Feature #1176)
- mask option for non-interpolated points in TreatmentLine
- enable the plus sign (+) in filenames
- **add comments to treatments in relation with**
  Spectral analysis treatment based on the Instant objects (Feature #1313)
- move function that swap axes from Reader to a new treatment
- for rel_to_abs method, adding keys coordinates and conservative_vars in order to be more general
- Replace the clip treatment of type 'value' with a threshold treatment (Feature #1341)
- **improve documentation of Reader and Writer**
  change the default coordinate names in the hdf_cgns reader and writer
- better handling of shared data in the duplication treatment (Bug #1331)
- ReaderHdfCgns: some boundaries not stored in the base (Bug #1302)
- correct the duplication of topology in duplication treatment (Bug #1340)
- Read files, regex on windows system (Bug #1304)

1.6.2 (September 02, 2015) - fix read hdf_cgns files with block connectivities

1.6.1 (August 24, 2015) - fix Swap axis for 2D structured base with shared instant (Bug #1261) - fix Reader HdfCgns try to import something that does not exist (Bug #1259)

1.6.0 (July 31, 2015)

- a timer function to know time between print statements
- readers for : - binary files in plot3D format (grid, Q, solution) (Feature #1172) - binary files in Fieldview format (Feature #1171) - binary files from AVBP (temporal)
- Automatic test procedure (Feature #1181)

- writer for : - binary files in Fieldview format (Feature #1171) - binary files in column format
- a 'variables' attribute to Window in order to store window data that will be set in the instant resulting from slicing
- vtk_unstructuredgrid_to_instant do not handle 'tri' and 'qua' (Feature #1220)
- Include prisms and pyramids in Tecplot Binary and Formatted Writers (Feature #1225)
- TreatmentChoroReconstruct should copy shared connectivities (Feature #1237)
- Treatment: extract data over a line (Feature #1236)
- Pre-rotate geometry at initial time in treatmentChoroReconstruct (Feature #1238)
- Allow duplication along other axis than x (Feature #1255)
- Writer elsA CFD input file (Feature #1257)
- Reader bin_vtk (Feature #1256) - in the hdf_avbp reader : - patch names are now retrieved from the file - make it compatible with v7 solutions (vectors and multi-variables arrays) (Feature #1229)
  - memory and speed improvement of :
  - clip treatment (with coordinates)
  - signal windowing
  - interpolation treatment
- Post-processing cannelle/elsa files - bad slice definition (Support #1230)
- change CGNS documentation reference
- some bugs on variable slicing of shared variables and removed potential force read (in case of lazy loading)
- TreatmentChoroReconstruct is not behaving as expected with cell data (Bug #1231)
- Small problem with HdfCGNSReader (Bug #1219)
- Tecplot binary writer does not write anything when only shared variables are present (Bug #1234)

1.5.3 (May 20, 2015)

- fix Reader tecplot v75 binary format with version ## 1.5.2 (Lazy loading not working with F=POINT) (Bug #1208)

1.5.2 (May 18, 2015)

- fix location='node' in v75 tecplot binary reader (Bug #1207)

1.5.1 (April 28, 2015)

- Using vtk methods with 2D meshes (Feature #1185)
- add more default values for coordinates (Feature #1179)
- add antares_csh.env csh script to set the environment for Antares
- improve bin_tp V75 reader to enable variable loading on demand

- merge read_variable_112 and read_variable_75 functions
- extension of bin_tp V75 format to FEBLOCK format
- pass attrs to sliced zone
- **slicing of shared variables generates now variables in the shared space and not in the instant space anymore.**
  reduce useless reading of variables
- Get the right attrs in the zone when reader has 'shared'='True' (Bug #1183)
- Extend reader for formatted tecplot format (Bug #1182)

1.5.0 (January 20, 2015)

- chorochronic reconstruction
- cut treatment of type 'revolution'
- geometric clipping based on vtk
- topology writer for elsA computation
- computation of conservative variables from the relative frame to the absolute frame
- tools for IO management in utils
- extension of HDF CGNS reader for boundary filename
- improvement of DFT with matrix formulation
- add pangle in the elsA topology reader
- add global borders for 'nomatch' joins in the elsA topology reader
- compatibility with vtk6 for vtk writer

1.4.1 (September 10, 2014)

- for AVSP users, add i_freq, r_freq, and modeindex to attrs
- compatibility with vtk6 for clip and cut treatment

1.4.0 (April 11, 2014)

- read and write unstructured cell centered data in Tecplot binary files
- spatial mean operator in compute function
- **antares.env script to automatically set the**
  PYTHONPATH and PATH environment variables
- target condition number value to optimize_timelevels function
- del_deep function on attrs object to remove an attribute at any level underneath
- squeeze function to a base object
- **APFT algorithm to optimize the time instance of a HbComputation**
  object
- **TreatmentDft to perform a discrete Fourier transform on time-marching results.**
  This has been developed and implemented by M. Daroukh.
- new examples available in examples/ folder
- **time argument to duplication treatment to ease the automatic rotation**
  with respect to time

- removing color in print
- **extras keyword to attrs and add_extra method to add_attr. This has been**
  done to be more compliant with the HDF data structure.
- 10 * log(psd) normalization has been removed in TreatmentPsd
- hpc_mode/hpc keywords to memory_mode
- **installation steps are now clearer and the libraries needed to use Antares**
  are detailed
- when dumping only shared variables in Tecplot binary files
- in treatment merge when using shared variables
- when reading a single point base in hdf_avbp (issue #33)
- fixed multiple location reading in Tecplot binary files

1.3.3 (July 5, 2013)

- gnuplot_2d writer, look at the file_format specification for more infos
- improved compute equation to fully take into account shared variables
- **bug correction in Tecplot fmt_tp format, data were not written using scientific notation**
  which could lead to precision issues
- shape management with shared (issue #32)
- copy and deepcopy of API elements

1.3.2 (June 7, 2013)

- hpc_mode in writer class and cut treatment. The given base is deleted on the fly which results in better memory performances.
- removing explicit gc.collect() calls using weakref module.
- hbdft, interp and choro 30% faster.
- Cell centered data shape management
- robustness of hdf_cgns reader

1.3.1 (May 30, 2013)

- Full python bin_v3d writer
- Formula of speed of sound changed to only take conservative variables as input, no gas constant is needed anymore
- Improve hbdft treatment to take into account cell values
- Add file lazy loading if variable lazy loading is not available
- For file format hdf_avbp: bug fix on reading the connectivity of flagged zones
- Reader bin_v3d bug fixes
- Bug fix in reading overlap boundaries in python topology card
- Bug fix when using the psd treatment with several instants

1.3.0 (April 19, 2013)

- Variable lazy-loading for all readers

- Reader for format hdf_cgns
- **Attribute dtype is now available for writer bin_tp and bin_vtk**
  to write in simple precision
- Examples for each treatment in the documentation
- Modified Antares header
- Multiple bugs fixed in deepcopy
- Writer can now write a base with only data in the shared instant
- Bug fixed for variable slicing with shared variables
- **Reader of multi-instant files (bin_tp format) was adding the various**
  instants as new zones

1.2.1 (April 10, 2013)

- Rotation of field using omega for HbChoro treatment
- Style option in plot Treatment
- prepare4tsm function for HB/TSM computations
- **Update doc: compute equation, global variables and**
  binary Tecplot format description.
- **Performance improvement of Instant object, cut treatment when**
  using shared variables and equation computation
- Robustness of Tecplot formatted file reader
- Bug in Base initialization when a Base object is given as input
- In merge treatment, the extras were lost

1.2.0 (March 20, 2013)

- Support for shared variables in cut/slice treatment
- python bin_v3d and fmt_v3d reader
- **Doc on the file format available for the Reader and the Writer objects. Add also**
  the library they need. (issue #14)
- **Igor reader and writer (formatted file), the file_format**
  key has to be set to igor (issue #13)
- **Ability to give a vtk implicit function to the cut treatment**
  (issue #19)
- **Multi-element support for unstructured grids**
  (issue #17)
- Pyramid element support
- ipython support
- init function on the Base object
- Instant can now have different shapes
- Removed pickle file format
- **Reading fmt_tp data only located at cells gave the wrong shape**
  (issue #21)

- **Ability to read fmt_tp files that have no title**
  (issue #20)
- **Bug in Filter treatment when a non-uniform time vector is given**
  (issue #12)
- **Formatted structured Tecplot files that have no K dimension**
  For HB/TSM users:
- p_source_term function which is the analytical TSM source term
- Bug correction in bounds of OPT algorithm

1.3 (January 29, 2013)

- Environment variable ANTARES_VERBOSE that can be used to set the default verbose within Antares (issue #6)
- Treatment Merge to merge 1D zones into a single one (issue #9)
- **title, xrange and yrange options**
  to the plot treatment
- Search box into the documentation
- Vtk files extension

1.2 (January 17, 2013)

- Families of Families
- **deep argument to the add_extra function. It allows to force setting**
  an extra at each level of a Family object
- handling prism elements
- variable_to_ignore attribute to hb_dft treatment
- reading/writing mixed cell/nodes binary Tecplot files
- **multiple file hability to writer vtk. It now creates an additional .pvd file that can be opened**
  within paraview and works as a multi-block file. Please note that the extension of the written files are replaced by .vtu for unstructured files and .vts for structured files.
- formatted Tecplot file writer. It is now easier to read it
- the use of connectivity attribute now forces the file to be read
- bug correction when using vtk with cell-centered data
- improve filename handling in Reader when the user gives a non-unix like filename
- Window slicing can not be applied on cell data

1.1 (December 12, 2012)

- min and max functions to Base.compute
- bug in vtk treatments when a base with cell values is given
- bug in version name, the 'dirty' word was displayed because of a bad git setting
- bug correction in vtk import, was done even if not needed
- bug correction in antares_plot binary, -v option had no effects

1.0 (December 5, 2012)

- Dmd and Dmd1d treatments
- Airfoil treatment
- Plot treatment
- io with shared variables
- io bin_tp (TDV112 and TDV75) in pure python. Please note that the binary tecplot format compatible with elsA is bin_tp_75
- io fmt_tp in pure python
- io vtk (fmt_vtk and bin_vtk)
- Base.get_location
- Harmonic Balance algorithm to optimize the timelevels
- Harmonic Balance source term computation
- The computation of Base.stats is memory efficient (done block per block)
- Base.find_superblocks function returns the superblocks found
- vtk object creation from Antares API is more efficient
- duplication and chorochronic duplication are now new zones not new instants
- location attribute on writer is removed, the user is pleased to use the function Base.get_location at base level
- reader HDF reads now the AVBP family and stores them into Base.families
- iPlot is renamed antares_plot for consistency, take a look at bin for information
- bug when reading a column file that has only one value per variable
- bug when deleting element of the API. These were not properly deleted, resulting in a memory leak
- bug when using Base.copy with a base that has shared variables
- bug when using Base.node_to_cell on a base that has shared variables
- bug when deepcopying a Family object
- memory leak when using slicing on an API object
- bug in isosurface treatment

1.0.9 (October 8, 2012)

- Filter treatment for signal processing to apply a low or high pass filter on a 1D signal
- function to compute node values from cells (for structured grid only, taking into account join condition if given)
- color function to remove color print (for log file for example)
- equations for variables Cp, Cv, Ti, Pi, hi
- Reader can take a base in input to fill with the data read
- shared Instant management (shape, variables positions, force read…)
- Clip treatment remove the zone if all the nodes are removed during the clipping
- Rgaz value has been removed from global variables to avoid any misuse

1.0.8 (September 7, 2012)

- in iPlot, one can now can remove legend using the -L option.

- unstructure function which allows to convert a structured base into an unstructured one
- 'clip' treatment which allow to remove a part of the base
- getitem functions behaviour
- added hexaedron cell type for unstructured writers
- 'slice' treatment name is now 'cut' and allows to make cut not only planar, but also cylindrical, spherical and conic
- refresh in iPlot now work correctly

1.0.72 (September 3, 2012)

- Hbinterp treatment time key can use in_extra feature,
- FFT treatment only returns the positive frequency part of the FFT.

1.0.71 (August 28, 2012)

- in iPlot, one can now export the picture as a png using the -s option.
- closest has been re-written in numpy instead of fortran.

1.0.7 (August 23, 2012)

- stats function on Base object that, for each variable, gives the mean, min, max and the variance,
- full python Tecplot binary file reader and writer,
- more treatments examples.
- Antares does not use tvtk anymore for slice and iso-surface treatments, it uses vtk instead
- bug when using the lazing loading and the shape attribute of instant

1.0.61 (July 31, 2012)

- bugs in TreatmentIsosurface.

1.0.6 (July 27, 2012)

- node_to_cell function (for both structured and unstructured grid),
- Window attributes, now use a zone_name attribute instead of the _parent attribute to know in which zone to work
- Zone/Instant shape attribute (the attribute can now be set with just a = instead of using set_shape function)
- Zone/Instant connectivity is now an attribute instead of a function,
- bugs in shared Instant management,
- bug in Base.grid_points,
- bug in Gridline treatment (for O grid),
- bug in FFT/PSD when using keys time_t0/tf to restrict the signal,
- bug in Base getitem.
- bug in Family getitem.

1.0.5 (July 16, 2012)

- topological extraction of planes, lines and probes,
- antares can now read Tecplot formatted files in point format,
- bug in writer column, for false 1D array (3D but with two dimensions that have shape 1),

- bug in FFT/PSD, were not working anymore.

1.0.4 (July 12, 2012)

- bug in setup.py when the fortran compilers were not understood by python
- bug in compute function when variables are shared between instants,
- bug in slice treatment (was time-consuming),

1.0.3 (July 9, 2012)

- iPlot, plotting tool based on Gnuplot.py,
- deepcopy function in API classes,
- some examples,
- clean option to the setup.py script,
- doc enhancement (in particular treatment part),
- bug in print that was not using the custom clint library,
- bug in tree slicing when a variable was shared in antares.Base,

1.0.2 (June 29, 2012)

- location can be set on reader/writer,
- example for restart tools for HB computations,
- phaselag parameter is now settable on a antares.HbComputation.
- doc enhancement,
- several bug fixes.

1.0.1 (June 5, 2012)

First version

This documentation contains the details of all classes and functions that can be used. To quickly find the documentation of a specific item, consider using the search box on the top right-hand side. If you prefer a whole document, consider the `pdf` documentation.

BIBLIOGRAPHY

[HIRSCHBERG] Hirschberg, A. and Rienstra, S.W. "An Introduction to Aeroacoustics", Instituut Wiskundige Dienstverlening (Eindhoven) (2004).

[TETRA] How to subdivide pyramids, prisms and hexaedra into tetrahedra, J. Dompierre, P. Labbe, M-G. Vallet, R. Camarero, Rapport CERCA R99-78, 24 august 1999 Conference paper from the 8th International Meshing Roundtable, Lake Tahoe, Cal., 10-13/10/1999

[NEUBAUER] Aerodynamique 3D instationnaire des turbomachines axiales multi-etage, Julien Neubauer, PhD thesis, 2004.

[NEUBAUER] Aerodynamique 3D instationnaire des turbomachines axiales multi-etage, Julien Neubauer, PhD thesis, 2004

[Giovannini] Evaluation of unsteady CFD models applied to the analysis of a transonic HP turbine stage, M. Giovannini & al., ETC10, 2013

[HE] Method of Simulating Unsteady Turbomachinery Flows with Multiple Perturbations, He, L., AIAA J., Vol. 30, 1992, pp. 2730{2735)"

[GREEN] PROCUREMENT EXECUTIVE MINISTRY OF DEFENCE AERONAUTICAL RESEARCH COUNCIL REPORTS AND MEMORANDA no 3791 Prediction of Turbulent Boundary Layers and Wakes in Compressible Flow by a Lag-Entrainment Method By J. E. GREEN, D. J. WEEKS AND J. W. F. BRODMAN, Aerodynamics Dept., R.A.E, Farnborough

[STOCKHAASE] Hans W. Stock and Werner Haase. "Feasibility Study of e^N Transition Prediction in Navier-Stokes Methods for Airfoils", AIAA Journal, Vol. 37, No. 10 (1999), pp. 1187-1196. https://doi.org/10.2514/2.612

[CHUNG] Computational Fluid Dynamics. T. J. Chung. Cambridge University Press, 2002. pp 273–275

[Moin] P. Moin and J. Kim (1982). "Numerical investigation of turbulent channel flow". In: Journal of Fluid Mechanics, 118, pp 341-377

[Abe] H. Abe et al. (2001). "Direct Numerical Simulation of a Fully Developed Turbulent Channel Flow With Respect to the Reynolds Number Dependence". In: Journal of Fluids Engineering, 123, pp 382-393.

[Gullbrand] J. Gullbrand (2003). "Grid-independent large-eddy simulation in turbulent channel flow using three-dimensional explicit filtering". In: Center for Turbulence Research Annual Research Briefs.

[LeBras] S. Le Bras et al. (2015). "Development of compressible large-eddy simulations combining high-order schemes and wall modeling". In: AIAA Aviation, 21st AIAA/CEAS Aeroacoustics Conference. Dallas, TX.

[TETRA] How to subdivide pyramids, prisms and hexaedra into tetrahedra, J. Dompierre, P. Labbe, M-G. Vallet, R. Camarero, Rapport CERCA R99-78, 24 august 1999 Conference paper from the 8th International Meshing Roundtable, Lake Tahoe, Cal., 10-13/10/1999

PYTHON MODULE INDEX

a

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z