AirfRANS

Florent Bonnet

Jul 03, 2023

NOTES

AirfRANS is a Python library for handling simulations coming from the AirfRANS dataset which makes available numerical resolutions of the incompressible Reynolds-Averaged Navier–Stokes (RANS) equations over the NACA 4 and 5 digits series of airfoils and in a subsonic flight regime setup.

It consists of a utility for downloading and loading the dataset under a NumPy format and a class to manipulate and execute some basic operations on the simulations. In particular, it allows to compute the force coefficent straightforwardly and it includes a method to visualize boundary layers and trails. Moreover, it allows to sample uniformly or with respect to the mesh density and to get rid of the mesh points constraint for training.

The original paper accepted at the 36th Conference on Neural Information Processing Systems (NeurIPS 2022) Track on Datasets and Benchmarks can be found here and the preprint here. **Disclaimer: An important update correcting an inconsistency in the Machine Learning experiments proposed in the main part of the NeurIPS version of the paper has been done. Please refer to the** ArXiv version **for the up to date version.**

CHAPTER

ONE

INTRODUCTION

The AirfRANS dataset is a collection of numerical simulations solving the incompressible Reynolds-Averaged Navier-Stokes equations over two dimensional airfoils in a subsonic flight regime. The associated paper has been accepted at the 36th Conference on Neural Information Processing Systems (NeurIPS 2022) Track on Datasets and Benchmarks. In addition to this library two GitHub repositories have been proposed to reconduct the paper experiments and to generate new compressible or incompressible simulations over NACA airfoils. The setup to generate those simulations have been confronted to the Langley Research Center experiments available in the Turbulence Modeling Resource for the NACA 0012 and 4412.

This dataset has been built to lower the potential barrier between Machine Learning and Physics research communities. It proposes data on a simple but realistic case which already includes some of the major challenges of Machine Learning for solving Fluid Dynamics, namely:

- working with unstructured data coming from raw numerical simulations,
- being able to deal with the number of nodes required in simulation meshes (from hundreds of thousands to hundreds of million in 3D cases),
- treating cases with a realistic Reynolds number,
- regressing the entire velocity, pressure and turbulent viscosity fields from a geometry and the boundary conditions,
- being accurate on global forces or coefficient such as drag and lift,
- being consistent between the predicted fields and the predicted forces,
- regressing accurately boundary layers and area of simulations where sharp signals appear,
- producing solutions that respect the conservation equation and the momentum equations.

We hope that this library will ease the manipulation of such simulations and the usage of the AirfRANS dataset.

1.1 Raw OpenFOAM data

The dataset comes under different form. A pre-processed version of cropped simulations and including only the minimum number of fields is proposed as a work basis. A full raw OpenFOAM data version is also available but its manipulation necessitates some basic knowledge of how OpenFOAM works. However, raw data include more information than the processed data and especially if you are interested in the gradient of the fields. Each raw data contains each term of the momentum and conservation equations as fields that could be used, for example, to compare the gradients of the approximation with the gradients of the simulation.

1.2 Tutorials

We used OpenFOAM v2112 to generate our simulation. The manipulation and visualization of the results can be done with ParaView and/or with a pythonic interface such as PyVista. Finally, the treatment of those data in a deep learning point of view can be done with Geometric Deep Learning library such as PyTorch Geometric or Deep Graph Library. As those tools and domains are not necessarily well known in the Machine Learning community, we would like to share some tutorials and books that helped us to be more comfortable with the subject:

- One of the OpenFOAM wiki is a must for learning this powerful tool. For just a taste, you can follow the First Glimpse Series and for a more in-depth introduction, the Three Weeks Series.
- Concerning ParaView, a part of the Three Weeks Series is dedicated to it but can be followed independently. You can find it here.
- This book proposes an overview of the mathematics in OpenFOAM.
- The Turbulence Modeling for CFD book for understanding how to model turbulence in CFD.
- The Fundamentals of Aerodynamics book for an aerodynamics centered presentation of fluid dynamics.
- More fundamentaly, the Fluid Mechanics book for a general introduction to fluid mechanics.

1.3 License

This work is proposed by Extrality and the MLIA team of Sorbonne Université, Paris.

CHAPTER TWO

INSTALLATION

Create a virtual environment with Python 3.7 or greater and simply run:

```
pip install airfrans
```

It will, in addition, install the following libraries: NumPy, PyVista and, tqdm.

CHAPTER

THREE

DATASET

The AirfRANS dataset makes available numerical resolutions of the incompressible Reynolds-Averaged Navier-Stokes (RANS) equations over the NACA 4 and 5 digits series of airfoils and in a subsonic flight regime setup.

Its features are:

- 1000 simulations
- Reynolds number between 2 and 6 million
- Angle of attacks between -5° and 15°
- Airfoil drawn in the NACA 4 and 5 digit series
- Four machine learning tasks representing different challenges.

The four tasks are defined as followed:

- **Full data regime**: 800 simulations are used for the training and 200 are kept for testing. Both the trainset and the testset are drawn from the same distribution. This defines an interpolation task.
- **Scarce data regime**: Same testset as the *Full data regime* task but with only 200 simulations in the trainset. This also defines an interpolation task but in a low data regime scenario.
- **Reynolds extrapolation regime**: Simulations with Reynolds number between 3 and 5 million are kept for the trainset, the others are kept for the testset. This defines an extrapolation task for the reynolds number parameter.
- **Angle of attack extrapolation regime**: Simulations with angle of attack between -2.5° and 12.5° are kept for the trainset, the others are kept for the testset. This defines an extrapolation task for the angle of attack parameter.

3.1 Downloading the dataset

You can download the dataset by using the function `airfrans.dataset.download`

```
import airfrans as af

af.dataset.download(root = PATH_TO_SAVING_DIRECTORY, file_name = 'Dataset', unzip = True,
↪ OpenFOAM = False)
```

for the pre-processed dataset where simulations have been cropped, minimum features have been kept and `.vtu` / `.vtp` files have been generated. For the raw OpenFOAM dataset, simply set `True` to the `OpenFOAM` argument. You can also directly download it here for the pre-processed version and there for the raw OpenFOAM.

Finally, you can access to a ready-to-use version via PyTorch Geometric datasets.

3.2 Loading the dataset

After dowloading the pre-processed dataset, you can load the list of simulations in NumPy arrays with the function *airfrans.dataset.load*

```
dataset_list, dataset_name = af.dataset.load(root = PATH_TO_DATASET, task = 'full',␣
↪train = True)
```

We recommend to handle this data with a Geometric Deep Learning library such as PyTorch Geometric or Deep Graph Library.

Note: The dataset loaded when using *airfrans.dataset.load* contains point clouds defined as the nodes of the simulation mesh. Do not hesitate to build a custom dataset if you want to use volume/surface sampling of simulations instead of the nodes of the mesh via the *airfrans.Simulation*. The only constraint is that the test scores of models for the four target fields have to be computed at the position of the nodes of the simulation meshes.

3.3 Raw versus pre-processed dataset

The raw dataset contains the simulations of the AirfRANS dataset as they were outputed by OpenFOAM. The pre-processed version differs from the raw version as it only includes a processed version of the `.vtu` / `.vtp` files of the simulations.

For the internal field given in the file ending by `internal.vtu`, we followed:

- **Clipping**: Clip to a box of size `[(-2, 4), (-1.5, 1.5), (0, 1)]` where each tuple represents an axis in the order x, y, and z. We did it with the PyVista clip filter and we set `crinkle = True` in the parameters to leave the cells untouched.
- **Slicing**: Take a slice at $z = 0.5$ to go from a 3D mesh to a 2D mesh. We did it with the PyVista slice filter and we set `generate_triangles = False`.
- **Distance function**: Compute the distance function with the PyVista compute_implicit_distance filter.
- **Keeping only the required fields**: Only keep the pressure, the velocity, the turbulent viscosity, and the distance function in the data.

For the aerofoil patch (internal boundary) given in the file ending by `aerofoil.vtp`, we followed:

- **Normals**: Compute the normals with the PyVista compute_normals filter. We set `flip_normals = False` leading to inward-pointing normals.
- **Slicing**: Take a slice at $z = 0.5$ to go from a 2D mesh to a 1D mesh. We did it with the PyVista slice filter and we set `generate_triangles = False`.
- **Keeping only the required fields**: Only keep the pressure, the velocity, the turbulent velocity, and the normals in the data.

For the freestream patch (external boundary) given in the file ending by `freestream.vtp`, we followed:

- **Slicing**: Take a slice at $z = 0.5$ to go from a 2D mesh to a 1D mesh. We did it with the PyVista slice filter and we set `generate_triangles = False`.
- **Keeping only the required fields**: Only keep the pressure, the velocity, and the turbulent velocity in the data.

Note: Since version `0.1.4`, the normals in the `airfrans.Simulation` class are inward-pointing instead of outward-pointing as previously. This for consistency with the normals given in the AirfRANS dataset.

CHAPTER

FOUR

SIMULATION

Each simulation is defined via boundary conditions (inlet velocity and angle of attack) and a geometry (NACA 4 or 5 digits). We recall that in an incompressible setup and when neglecting gravity, the only parameter of Navier-Stokes equations (and also of RANS equations) is the Reynolds number. If we fix the kinematic viscosity (*i.e.* if we fix the temperature of air one for all), the Reynolds number is completely defined by the inlet velocity (as the characteristic length of airfoil is chosen to be 1 meter). In this dataset, we chose to arbitrarily fix the temperature to 298.15K and to work with properties of air at that temperature, at a pressure of 1013.25 hPa.

You can run new simulation with the help of OpenFOAM v2112 and the NACA simulation GitHub.

Note: During the generation of the dataset, we fixed the kinematic viscosity at a numerical value of 1.56e-5. After the addition of the temperature parameters in the simulation, the computed kinematic viscosity at 298.15K is of roughly 1.55e-5 leading to slight discrepancies at a fix inlet velocity between newly generated simulations and simulations from the dataset. Those discrepencies do not exist if we only take as parameter the Reynolds number.

In the following, we present attributes and methods of the `airfrans.Simulation` class.

To load a simulation (for example simulation `'airFoil2D_SST_43.597_5.932_3.551_3.1_1.0_18.252'`), simply run:

```
import airfrans as af

simulation = af.Simulation(root = PATH_TO_DATASET, name = 'airFoil2D_SST_43.597_5.932_3.
↪551_3.1_1.0_18.252', T = 298.15)
```

Note: The name of a simulation gives all the information about the boundary conditions used to generate it. For example, in `'airFoil2D_SST_43.597_5.932_3.551_3.1_1.0_18.252'`, we read that the inlet velocity magnitude is $43.597 m \cdot s^{-1}$, and the angle of attack $5.932°$. The last numbers are for the parameters of the NACA arifoil, if we have 3 parameters it means that we are dealing with an airfoil from the 4-digits series, and if we have 4 parameters then this airfoil comes from the 5-digits series (this is due to the fact that the last two digits gives the thickness of the airfoil which only defines a single parameter). To keep with our example, we here deal with an airfoil of the 5-digits series with parameters given by `(3.551, 3.1, 1.0, 18.252)` (which are real numbers and not integers). Another example, the NACA 0012 will lead to parameters `(0, 0, 12)` and a name ending by `0_0_12`.

4.1 Attributes

One part of the attributes of the class is the properties of air at the given temperature:

- `airfrans.Simulation.MOL` is the molar weigth of air in $kg \cdot mol^{-1}$
- `airfrans.Simulation.P_ref` is the reference pressure in Pa
- `airfrans.Simulation.RHO` is the specific mass in $kg \cdot m^{-3}$
- `airfrans.Simulation.NU` is the knimeatic viscosity in $m^2 \cdot s^{-2}$
- `airfrans.Simulation.C` is the sound velocity in $m \cdot s^{-1}$.

A second part is the boundary conditions:

- `airfrans.Simulation.inlet_velocity` is the inlet velocity in $m \cdot s^{-1}$
- `airfrans.Simulation.angle_of_attack` is the angle of attack in rad.

A third part is the PyVista object of the reference simulation:

- `airfrans.Simulation.internal` is the PyVista object of the internal patch
- `airfrans.Simulation.airfoil` is the PyVista object of the aerofoil patch

Finally, the last part is the fields associated with the simulation under the form of NumPy ndarray. Those fields are either defined on the mesh nodes, are the airfoil patch nodes directly:

- `airfrans.Simulation.input_velocity` is the inlet velocity copied on each nodes of the internal mesh in $m \cdot s^{-1}$
- `airfrans.Simulation.sdf` is the distance function on the internal mesh in m
- `airfrans.Simulation.surface` is a boolean on the internal mesh, it is `True` if the node lie on the airfoil
- `airfrans.Simulation.position` is the position of the nodes of the internal mesh in m
- `airfrans.Simulation.airfoil_position` is the position of the nodes of the airfoil mesh in m
- `airfrans.Simulation.normals` is the inward-pointing normals of the surface on the internal mesh, it is set to 0 for points not lying on the airfoil
- `airfrans.Simulation.airfoil_normals` is the inward-pointing normais of the surface on the airfoil mesh

and for the targets:

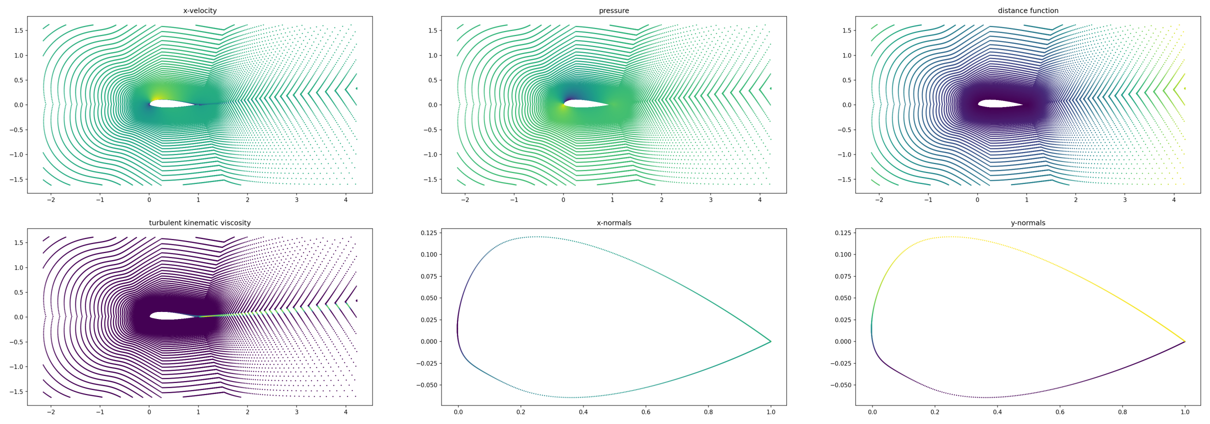
- `airfrans.Simulation.velocity` is the air velocity on the internal mesh in $m \cdot s^{-1}$
- `airfrans.Simulation.pressure` is the air pressure on the internal mesh (divided by the specific mass in the incompressible case)
- `airfrans.Simulation.nu_t` is the kinematic turbulent viscosity on the internal mesh in $m^2 \cdot s^{-2}$

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(3, 2, figsize = (36, 12))
ax[0, 0].scatter(simulation.position[:, 0], simulation.position[:, 1], c = simulation.
↪velocity[:, 0], s = 0.75)
ax[0, 1].scatter(simulation.position[:, 0], simulation.position[:, 1], c = simulation.
↪pressure[:, 0], s = 0.75)
ax[0, 2].scatter(simulation.position[:, 0], simulation.position[:, 1], c = simulation.
↪sdf[:, 0], s = 0.75)
ax[1, 0].scatter(simulation.position[:, 0], simulation.position[:, 1], c = simulation.nu_
```

(continues on next page)

(continued from previous page)

```
↪t[:, 0], s = 0.75)
ax[1, 1].scatter(simulation.airfoil_position[:, 0], simulation.airfoil_position[:, 1], c␣
↪= simulation.airfoil_normals[:, 0], s = 0.75)
ax[1, 2].scatter(simulation.airfoil_position[:, 0], simulation.airfoil_position[:, 1], c␣
↪= simulation.airfoil_normals[:, 1], s = 0.75)
...
```



Note: Be careful that the ordering of points over the airfoil in the internal mesh or in the airfoil mesh is not the same. The function *airfrans.reorganize* is built to reordered the points as we want.

```
internal_normals = simulation.normals[simulation.surface]
print((internal_normals == simulation.airfoil_normals).all())
>> False

reordered_normals = af.reorganize(simulation.position[simulation.surface], simulation.
↪airfoil_position, internal_normals)
print((reordered_normals == simulation.airfoil_normals).all())
>> True
```

4.2 Methods

Sampling methods are available allowing to potentially free the constrainte of the mesh structure:

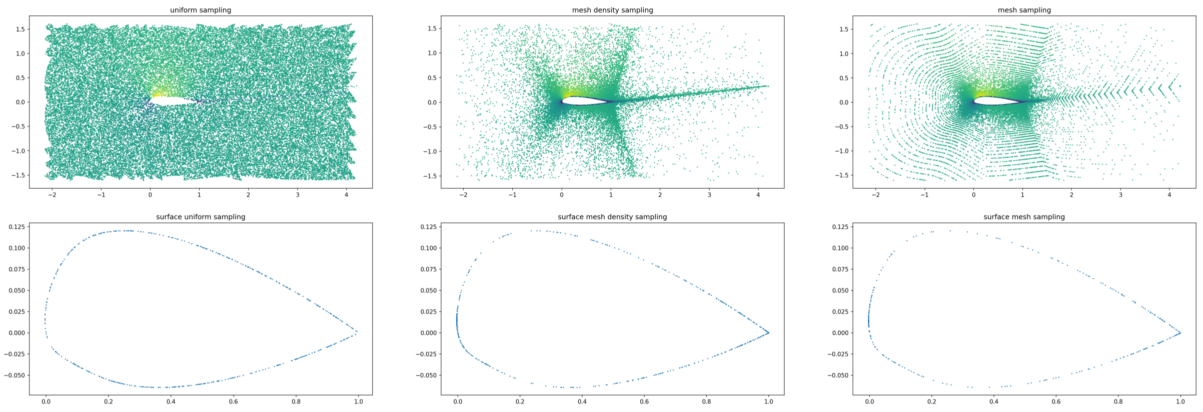
- *airfrans.Simulation.sampling_volume* allows sampling from two different densities on the internal mesh domain
- *airfrans.Simulation.sampling_surface* allows sampling from two different densities on the airfoil mesh domain
- *airfrans.Simulation.sampling_mesh* allows the sampling of nodes in the internal mesh

```
seed = 0

sampling_volume_uniform = simulation.sampling_volume(seed, 50000, density = 'uniform')
sampling_volume_mesh = simulation.sampling_volume(seed, 50000, density = 'mesh_density')

```

(continues on next page)

(continued from previous page)

```
sampling_surface_uniform = simulation.sampling_surface(seed, 500, density = 'uniform')
sampling_surface_mesh = simulation.sampling_surface(seed, 500, density = 'mesh_density')

sampling_mesh = simulation.sampling_mesh(seed, 50000)
sampling_mesh_surface = sampling_mesh[sampling_mesh[:, 2].astype('bool')]

fig, ax = plt.subplots(2, 3, figsize = (36, 12))
ax[0, 0].scatter(sampling_volume_uniform[:, 0], sampling_volume_uniform[:, 1], c =␣
↪sampling_volume_uniform[:, 3], s = 0.75)
ax[0, 1].scatter(sampling_volume_mesh[:, 0], sampling_volume_mesh[:, 1], c = sampling_
↪volume_mesh[:, 3], s = 0.75)
ax[0, 2].scatter(sampling_mesh[:, 0], sampling_mesh[:, 1], c = sampling_mesh[:, 8], s =␣
↪0.75)
ax[1, 0].scatter(sampling_surface_uniform[:, 0], sampling_surface_uniform[:, 1], s = 0.
↪75)
ax[1, 1].scatter(sampling_surface_mesh[:, 0], sampling_surface_mesh[:, 1], s = 0.75)
ax[1, 2].scatter(sampling_mesh_surface[:, 0], sampling_mesh_surface[:, 1], s = 0.75)
...
```



You can also directly compute the wall shear stress and the force coefficient with the class attributes or the reference simulation:

```
simulation.velocity = np.zeros_like(simulation.velocity)
simulation.pressure = np.zeros_like(simulation.pressure)

print(simulation.force())
>> (array([0., 0.]), array([-0., -0.]), array([0., 0.]))

print(simulation.force(reference = True))
>> (array([-79.15, 907.93]), array([-87.92, 906.80]), array([8.78, 1.14]))

print(simulation.force_coefficient())
>> ((0.0, 0.0, 0.0), (0.0, 0.0, 0.0))

print(simulation.force_coefficient(reference = True))
>> ((0.0134, 0.0056, 0.0079), (0.8099, 0.8097, 0.0002))
```

Some classical metrics between the attributes fields/forces and the reference fields/forces, for example the mean squared error:

```
print(simulation.mean_squared_error())
>> array([1100.53, 228.03, 227577.73, 0.])

simulation.reset()
print(simulation.mean_squared_error())
>> array([0., 0., 0., 0.])
```

Finally, you can save new `.vtu` and `.vtp` files with the fields given in attributes of the class:

```
simulation.save(root = SAVING_PATH)
```

CHAPTER FIVE

VISUALIZATION

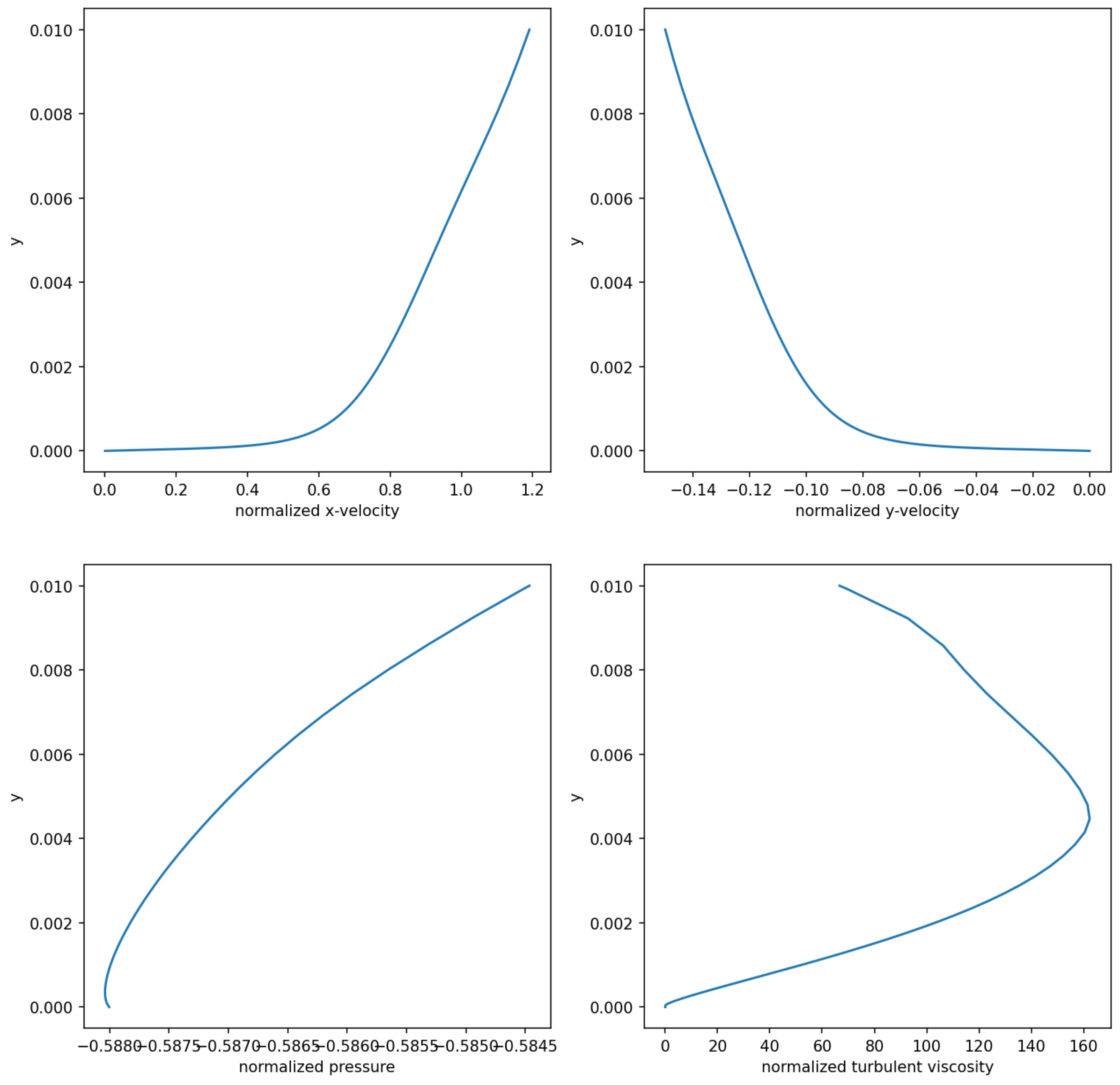
Visualizations entire simulations is often not sufficient to qualitatively assess the performance of a model. Boundary layers or trails are important parts of dynamics and they are often very localized leading to difficulties to visualize performances through a global representation of the simulation.

We encourage the usage of ParaView and one of its pythonic interface PyVista to manipulate and visualize the saved `.vtu` or `.vtp` generated with the `save` method of the *airfrans.Simulation* class.

5.1 Boundary layers plots

However, we propose a ready-to-use method *airfrans.Simulation.boundary_layer* for visualizing boundary layers and trails to facilitate workflows. This method is a wrapper of the `Plot Over Line` function of ParaView using PyVista. Note that this method uses the fields given in the attribute of the Simulation class to generate the line sampling.

```
import airfrans as af
import matplotlib.pyplot as plt

# Sample the boundary layer
simulation = af.Simulation(root = PATH_TO_DATASET, name = 'airFoil2D_SST_43.597_5.932_3.
↪551_3.1_1.0_18.252')
boundary_layer = simulation.boundary_layer(x = 0.5, y = 0.01)

# Plot the sampling with matplotlib
fig, ax = plt.subplots(2, 2, figsize = (12, 12))
ax[0, 0].plot(boundary_layer[1], boundary_layer[0])
...
```



5.2 Surface plots

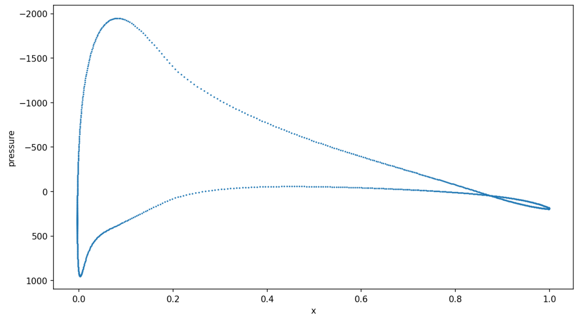
You could also need to check the predictions of your model over the airfoil. The important quantities are the pressure and the intensity of the wall shear stress (also called the skin friction), computed via the jacobian of the velocity via *airfrans.Simulation.wallshearstress*.

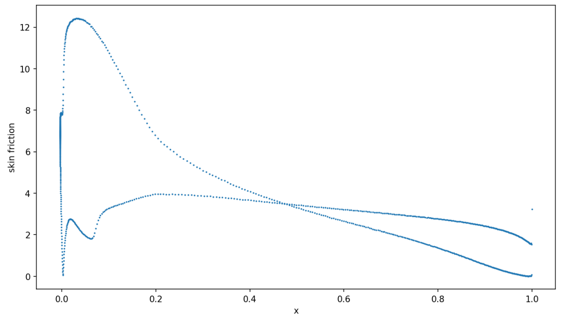
```
import numpy as np
import airfrans as af
import matplotlib.pyplot as plt

# Compute the wall shear stress
simulation = af.Simulation(root = PATH_TO_DATASET, name = 'airFoil2D_SST_43.597_5.932_3.
↪551_3.1_1.0_18.252')
norm_wss = np.linalg.norm(simulation.wallshearstress()[simulation.surface], axis = 1)
```

(continues on next page)

(continued from previous page)

```
# Plot the sampling with matplotlib
fig, ax = plt.subplots(1, 2, figsize = (24, 6))
ax[0].plot(simulation.position[simulation.surface, 0], simulation.pressure[simulation.
↪surface, 0], s = 0.75)
ax[1].plot(simulation.position[simulation.surface, 0], norm_wss, s = 0.75)
...
```





CHAPTER

SIX

AIRFRANS

reorganize(*in_order_points*, *out_order_points*, *quantity_to_reorder*)

Reorder points coming from different source of data with the same positions.

Parameters

- **in_order_points** (*np.ndarray*) – Positions of the features we want to reorder (shape *(N, 2)*).
- **out_order_points** (*np.ndarray*) – Positions for the new ordering (shape *(N, 2)*).
- **quantity_to_reorder** (*np.ndarray*) – Features attached to the points we want to reorder (shape *(N, F)*).

class Simulation(*root*, *name*, *T=298.15*)

Wrapper to make the study of AirfRANS simulations easier. It only takes in input the location of the `.vtu` and `.vtp` files from the internal and airfoil patches of the simulation.

Define the air properties at temperature `T`, the parameters, input features and targets for the simulation.

Parameters

- **root** (*str*) – Root directory where the dataset is.
- **name** (*str*) – Name of the simulation.
- **T** (*float, optional*) – Temperature set for the simulation in Kelvin (in incompressible settings, this only defines the air properties. Simulations do not depend on temperature, only on the Reynolds number). Default: 298.15

sampling_volume(*seed*, *n*, *density='uniform'*, *targets=True*)

Sample points in the internal mesh following the given density. The outputs is a ndarray of shape (n, 2 + 1) or (n, 2 + 1 + 4) if targets is `True`. The two first columns are the position of the sampled points, followed by the distance to the airfoil and the targets (in the order x-velocity, y-velocity, pressure, and turbulent viscosity) if called.

Parameters

- **seed** (*int*) – Seed for the random number generator.
- **n** (*int*) – Number of sampled points.
- **density** (*str, optional*) – Density from which the sampling is done. Choose between `'uniform'` and `'mesh_density'`. Default: `'uniform'`
- **targets** (*bool, optional*) – If `True`, velocity, pressure and kinematic turbulent viscosity will be returned in the output ndarray. Default: `True`

`sampling_surface`(*seed*, *n*, *density='uniform'*, *targets=True*)

Sample points in the airfoil mesh following the given density. The outputs is a ndarray of shape (n, 2 + 2) or (n, 2 + 2 + 4) if targets is `True`. The two first columns are the position of the sampled points, followed by the x and y components of the inward-pointing normals and the targets (in the order x-velocity, y-velocity, pressure, and turbulent viscosity) if called.

Parameters

- **seed** (*int*) – Seed for the random number generator.
- **n** (*int*) – Number of sampled points.
- **density** (*str, optional*) – Density from which the sampling is done. Choose between `'uniform'` and `'mesh_density'`. Default: `'uniform'`
- **targets** (*bool, optional*) – If `True`, velocity, pressure and kinematic turbulent viscosity will be returned in the output ndarray. Default: `True`

`sampling_mesh`(*seed*, *n*, *targets=True*)

Sample points over the simulation mesh without replacement. The outputs is a ndarray of shape (n, 2 + 6) or (n, 2 + 6 + 4) if targets is `True`. The two first columns are the position of the sampled points, followed by a boolean set to `True` if the sample points belongs to the surface, the distance to the airfoil, the x and y components of the normals (set to 0 if the sample points does not belong to the surface), the x and y components of the input velocity and the targets (in the order x-velocity, y-velocity, pressure, and turbulent viscosity) if called.

Parameters

- **seed** (*int*) – Seed for the random number generator.
- **n** (*int*) – Number of sampled points, this number has to be lower than the total number of points in the mesh.
- **targets** (*bool, optional*) – If `True`, velocity, pressure and kinematic turbulent viscosity will be returned in the output ndarray. Default: `True`

`wallshearstress`(*over_airfoil=False*, *reference=False*)

Compute the wall shear stress.

Parameters

- **over_airfoil** (*bool, optional*) – If `True`, return the wall shear stress over the airfoil mesh. If `False`, return the wall shear stress over the internal mesh. Default: `False`
- **reference** (*bool, optional*) – If `True`, return the wall shear stress computed with the reference velocity field. If `False`, compute the wall shear stress with the velocity attribute of the class. Default: `False`

`force`(*compressible=False*, *reference=False*)

Compute the force acting on the airfoil. The output is a tuple of the form *(f, fp, fv)*, where *f* is the force, *fp* the pressure contribution of the force and *fv* the viscous contribution of the force.

Parameters

- **compressible** (*bool, optional*) – If `False`, multiply the force computed with the simulation field by the specific mass. Default: `False`
- **reference** (*bool, optional*) – If `True`, return the force computed with the reference fields. If `False`, compute the force with the fields attribute of the class. Default: `False`

force_coefficient(*compressible=False*, *reference=False*)

Compute the force coefficients for the simulation. The output is a tuple of the form *((cd, cdp, cdv), (cl, clp, clv))*, where *cd* is the drag coefficient, *cdp* the pressure contribution of the drag coefficient and *cdv* the viscous contribution of the drag coefficient. Same for the lift coefficient *cl*.

Parameters

- **compressible** (*bool, optional*) – If `False`, multiply the force computed with the simulation field by the specific mass. Default: `False`
- **reference** (*bool, optional*) – If `True`, return the force coefficients computed with the reference fields. If `False`, compute the force coefficients with the fields attribute of the class. Default: `False`

mean_absolute_error()

Compute the mean absolute error between the reference target fields and the attribute target fields of the class. The target fields are given in this order: velocity_x, velocity_y, pressure, kinematic turbulent viscosity.

mean_squared_error()

Compute the mean squared error between the reference target fields and the attribute target fields of the class. The target fields are given in this order: velocity_x, velocity_y, pressure, kinematic turbulent viscosity.

r_squared()

Compute the r_squared between the reference target fields and the attribute target fields of the class. The target fields are given in this order: velocity_x, velocity_y, pressure, kinematic turbulent viscosity.

coefficient_relative_error()

Compute the mean relative error between the reference force coefficient and the force coefficient computed with the attribute target fields of the class. The force coefficients are given in this order: drag coefficient, lift coefficient.

boundary_layer(*x*, *y=0.1*, *extrado=True*, *direction='vertical'*, *local_frame=False*, *resolution=1000*, *compressible=False*, *reference=False*)

Return the boundary layer profile or the trail profile at abscissas x over a line of length y.

The fields are returned in the following order: the position on the line (in chord length), the first component of the velocity in the chosen frame normalized by the inlet velocity, the second component of the velocity in the chosen frame normalized by the inlet velocity, the pressure normalized by the inlet dynamic pressure, and the turbulent kinematic viscosity normalized by the knimatic viscosity.

Parameters

- **x** (*float*) – Abscissa in chord length. It must be strictly positive. If x < 1, return the boundary layer. If x >= 1, return the trail profile.
- **y** (*float, optional*) – Length of the sampling line. If x < 1, this length is taken from the airfoil surface. If x >= 1, this length is taken from one side to the other of the trail. Default: 0.1
- **extrado** (*bool, optional*) – If `True`, the boundary layer of the extrado is returned, If `False`, the boundary layer of the intrado is returned. If x>= 1, this parameter is not taken into account. Default: `True`
- **direction** (*str, optional*) – Choose between `'vertical'` and `'normals'`. If x < 1, the sampling line is defined as the line starting at the surface of the airfoil at abscissa x, of length y, in the direction of the normals if `'normals'` and in the vertical direction if `'vertical'`. If x>= 1, this parameter is not taken into account and the vertical direction is adopted. Default: `'vertical'`

- **local_frame** (*bool, optional*) – If `True`, the sampled velocity components along the lines are given in the local frame, i.e. the frame defined by the normals. Else, the sampled velocity components are given in the cartesian frame. If x>= 1, this parameter is not taken into account and the cartesian frame is adopted. Default: False resolution (int, optional): Resolution of the sampling. Default: 1000
- **compressible** (*bool, optional*) – If `True`, add the specific mass to the normalization constant for the pressure. Default: `False`
- **reference** (*bool, optional*) – If `True`, return the sampled fields of reference. If `False`, return the sampled fields from the class attribute. Default: `False`

`save`(*root*)

Save the internal and the airfoil patches with the attribute targets fields of the class in the root directory.

Parameters
root (*str*) – Root directory where the files will be saved.

CHAPTER SEVEN

AIRFRANS.DATASET

download(*root*, *file_name='Dataset'*, *unzip=True*, *OpenFOAM=False*)

Download AirfRANS dataset.

Parameters

- **root** (*str*) – Root directory where the dataset will be downloaded and unzipped.
- **file_name** (*str, optional*) – Name of the dataset file. Default: `'Dataset'`
- **unzip** (*bool, optional*) – If `True`, unzip the dataset file. Default: `True`
- **OpenFOAM** (*bool, optional*) – If `True`, it will download the raw OpenFOAM simulation with no post-processing to manipulate it through PyVista. If `False`, it will download the `.vtu` and `.vtp` of cropped simulations with a reduced quantity of features. Those cropped simulations have been used to train models proposed in the associated paper. Default: `False`

load(*root*, *task*, *train=True*)

The different tasks (`'full'`, `'scarce'`, `'reynolds'`, `'aoa'`) define the utilized training and test splits. Please note that the test set for the `'full'` and `'scarce'` tasks are the same. Each simulation is given as a point cloud defined via the nodes of the simulation mesh. Each point of a point cloud is described via 7 features: its position (in meters), the inlet velocity (two components in meter per second), the distance to the airfoil (one component in meter), and the normals (two components in meter, set to 0 if the point is not on the airfoil).

Each point is given a target of 4 components for the underlying regression task: the velocity (two components in meter per second), the pressure divided by the specific mass (one component in meter squared per second squared), the turbulent kinematic viscosity (one component in meter squared per second).

Finally, a boolean is attached to each point to inform if this point lies on the airfoil or not.

The output is a tuple of a list of np.ndarray of shape *(N, 7 + 4 + 1)*, where N is the number of points in each simulation and where the features are ordered as presented in this documentation, and a list of name for the each corresponding simulation.

We highly recommend to handle those data with the help of a Geometric Deep Learning library such as PyTorch Geometric or Deep Graph Library.

Parameters

- **root** (*string*) – Root directory where the simulation directories have been saved.
- **task** (*string*) – The task to study (`'full'`, `'scarce'`, `'reynolds'`, `'aoa'`) that defines the utilized training and test splits.
- **train** (*bool, optional*) – If `True`, loads the training dataset, otherwise the test dataset. Default: `True`

CHAPTER EIGHT

AIRFRANS.SAMPLING

`cell_sampling_2d`(*seed*, *cell_points*, *cell_attr=None*)

Sample points in a two dimensional cell via parallelogram sampling and triangle interpolation via barycentric coordinates. The vertices have to be ordered in a certain way.

Parameters

- `seed` (*`int`*) – Seed for the random number generator.
- `cell_points` (*`np.ndarray`*) – Vertices of the 2 dimensional cells. Shape *(N, 4)* for N cells with 4 vertices.
- `cell_attr` (*`np.ndarray, optional`*) – Features of the vertices of the 2 dimensional cells. Shape *(N, 4, k)* for N cells with 4 edges and k features. If given shape *(N, 4)* it will resize it automatically in a *(N, 4, 1)* tensor. Default: `None`

`cell_sampling_1d`(*seed*, *line_points*, *line_attr=None*)

Sample points in a one dimensional cell via linear sampling and interpolation.

Parameters

- `seed` (*`int`*) – Seed for the random number generator.
- `line_points` (*`np.ndarray`*) – Edges of the 1 dimensional cells. Shape *(N, 2)* for N cells with 2 edges.
- `line_attr` (*`np.ndarray, optional`*) – Features of the edges of the 1 dimensional cells. Shape *(N, 2, k)* for N cells with 2 edges and k features. If given shape *(N, 2)* it will resize it automatically in a *(N, 2, 1)* tensor. Default: `None`

CHAPTER NINE

AIRFRANS.NACA_GENERATOR

thickness_dist(*t*, *x*, *CTE=True*)

Standard NACA profile to warp with the help of a camber line to define all the 4 and 5 digits profiles.

Parameters

- **t** (*float*) – Thickness of the airfoil in percentage of the chord length.
- **x** (*np.ndarray*) – Abscissas in chord unit.
- **CTE** (*bool, optional*) – If `True` the profile will be closed at the trailing edge. Default: `True`

camber_line(*params*, *x*)

Camber line definition for the NACA 4 and 5 digits series.

Parameters

- **params** (*np.ndarray*) – Parameters of the NACA 4 or 5 digits profile (ndarray of shape *(3)* or *(4)*).
- **x** (*np.ndarray*) – Abscissas in chord unit.

naca_generator(*params*, *nb_samples=400*, *scale=1*, *origin=(0, 0)*, *cosine_spacing=True*, *verbose=True*, *CTE=True*)

Definition of a complete profile from the NACA 4 and 5 digits series.

Parameters

- **params** (*np.ndarray*) – Parameters of the NACA 4 or 5 digits profile (ndarray of shape *(3)* or *(4)*).
- **nb_samples** (*int, optional*) – Number of points to define the profile. Default: 400
- **scale** (*float, optional*) – Chord length in meters. Default: 1
- **origine** (*tuple, optional*) – Absolute position of the leading edge. Default: *(0, 0)*
- **cosine_spacing** (*bool, optional*) – If `True`, points are sampled via a cosine distance instead of uniformly. Default: `True`
- **verbose** (*bool, optional*) – Comments on the generation process. Default: `True`
- **CTE** (*bool, optional*) – If `True` the profile will be closed at the trailing edge. Default: `True`

PYTHON MODULE INDEX

INDEX