eddylicious Documentation

Release 0.0.5

Timofey Mukha

Oct 05, 2017

Table of Contents

Eddylicious is a python library for generating inflow boundary fields for scale-resolving simulations of turbulent flow.

The goal of eddylicious is to serve as a central place-holder for existing inflow generation methods and make them easily accessible to the CFD community.

Here, all the documentation for the library is collected. This includes a user guide and tutorials, a guide for developers, and source code documentation.

CHAPTER 1

User guide

Legal information

Eddylicious is free software and is provided under the GNU GPL Version 3.

Please take notice of the following disclaimers.

Disclaimer of warranty

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

Limitation of liability

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Introduction

Generation of inflow boundary fields for scale-resolving simulations of fluid flow has received significant attention from the CFD community. A review of existing methods can be found in *[TBA10]* and *[Jar08]*.

As correctly pointed out in *[TBA10]*, in order to benchmark existing methods one would require to implement all of them in a single code. Typically, however, a small number of methods is implemented within the framework of a concrete solver. A solution to this problem is to create a solver-agnostic library which would implement various generation methods found in the literature. Eddylicious strives to become such a library.

In order to support a large variety of CFD solvers, all the inflow fields generated by eddylicious are saved to disk. The only requirement on the solver's side is then to be able to read in boundary data from the hard-drive. Since different solvers support different formats for boundary data storage, eddylicious supports output in several different formats and can easily be extended to support more.

The fact that eddylicious is written in python implies that one is able to harness the power of all the well-developed libraries written for this language, such as numpy, scipy, h5py etc. This makes eddylicious a convenient tool for prototyping new inflow generation methods.

Interfacing with solvers via files also removes the necessity to consider solver-related issues, for instance geometry decomposition for parallel processing. This ensures that the code in eddylicious is free of clutter and makes it an ideal place for showcasing newly-developed methods and making them instantly available to the CFD community.

The package consists of the following components.

- The source code of the library, which implements the available functionality. If you are interested in contributing to the library, take a look at the *Developer guide*. The *Source code reference* can be used as help when using the already existing modules and functions in your own code.
- Executable python scripts that are used to drive the inflow field generation, see *Inflow generation methods*.
- Utilities, which are also executable python scripts, that provide extra functionality, like computing statistical data or conversion between different file formats. See *Utilities* for more information.
- Documentation, which includes this *User guide*, the *Developer guide*, and the *Source code reference*.
- *Tutorials* which provide examples of how the package can be used.

Installing the package

Dependencies

Relatively new versions of both python 2 and 3 should work, but the package has only been extensively tested with python 2.7.

Some MPI compatable with `mpi4py` should be installed on your system.

The package depends on the following python packages:

- `numpy`
- `scipy`
- `matplotlib`
- `mock`
- `Sphinx` and `sphinxcontrib-bibtex`
- `h5py` with support for MPI I/O.

- `mpi4py`

Make sure these are available before you install eddylicious.

Installing from git

Clone the following repository to a location of your choice using `git` https://github.com/timofeymukha/eddylicious

The catalog `eddylicious` is created. Go inside the catalog. Run `python setup.py install` to install the package. As usual, run with `sudo` if that is necessary. The `--prefix` flag can be used to install to a custom directory.

If you intend to try latest updates immediately run `python setup.py develop` instead. This way, if you update the files by running `git pull` you don't have to reinstall the package.

Common notation

Here we define the notation for several important physical quantities.

- U_0 — the freestream streamwise velocity.
- u_τ — the friction velocity.
- ν — kinematic viscosity.
- δ_ν — the viscous lengthscale, computed as ν/u_τ.
- c_f — the skin friction coefficient.
- **Measures of the thickness of a boundary layer.**
  - δ_{99} — the location where the mean velocity is equal to $0.99U_0$.
  - δ_* — the displacement thickness.
  - θ — the momentum thickness.
- **The velocity field.**

  $u,\ v,\ w$ — the streamwise, wall-normal, and spanwise components of the velocity field.
  - $U,\ V,\ W$ — the mean streamwise, wall-normal, and spanwise components of the velocity field.
  - $u',\ v',\ w'$ — the streamwise, wall-normal, and spanwise components of the fluctuations of the velocity field.
- Spatial coordinates.
  - $x,\ y,\ ,z$ — axes aligned with the streamwise, wall-normal, and spanwise directions.
  - η — the wall normal coordinate scaled with the 'outer scale', commonly δ_{99}.
  - y^+ — the wall normal coordinate scaled with the viscous lengthscale δ_ν.

Inflow generation methods

Using the generators

Each generation method provided by the library has an executable Python script associated with it. Running the appropriate script executes the generation procedure. All inflow generation methods depend on a certain amount of

parameters. These parameters are communicated to the script via a configuration file, which is passed as a command-line argument.

```
nameOfTheScript --config=configurationFileName
```

The configuration file is a simple text file, with the following layout

```
# This is a comment, it can describe the parameter below
parameterOne    valueOfParameter1

# Another comment
parameterTwo    valueOfParameter2
```

Which parameters should be present depends on the method that is being used. They are therefore described for each method individually below.

Lund's rescaling

Theory

This method was presented in a paper by Lund, Wu and Squires *[LWS98]*. The method is suitable for generating an accurate inflow field for a simulation which involves a turbulent boundary layer approaching from upstream. For example, it can be used in a simulation of flow around a smooth ramp or a backward-facing step.

In order to introduce the method, let us consider a zero-pressure flat plate turbulent boundary layer (TBL). At the inlet of the domain the desired characteristics of the boundary layer, such as Re_τ and $\mathrm{Re}_{\delta_{99}}$, are known.

The central idea of the method is to obtain the values at the inlet by rescaling the velocity field from a plane located some place downstream, see Fig. 1.1. The rescaling procedure involved insures that the target characteristics of the boundary layer at the inlet are met. The plane located downstream is referred to as the recycling plane.

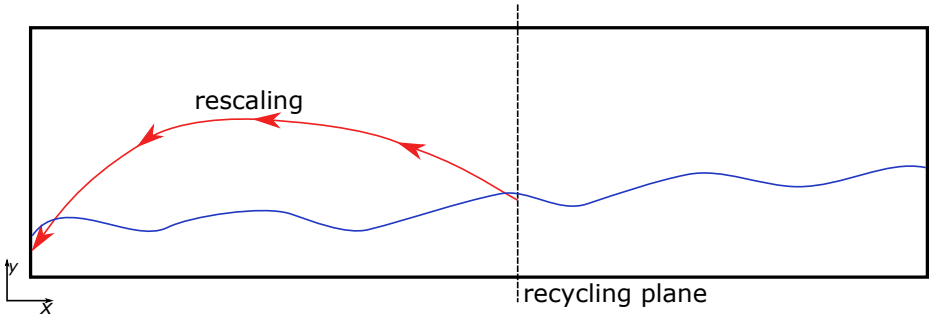

Fig. 1.1: Schematic of the rescaling from a plane located downstream proposed by Lund et al *[LWS98]*.

Since eddylicious does not interact directly with an ongoing simulation, the method has to be reformulated. Instead of using a recycling plane located downstream of the inlet in the main simulation, the proposed approach is to have a separate, precursor simulation dedicated to generating a database of two-dimensional velocity distributions. This database then serves as input for eddylicious, which applies the rescaling procedure as defined in *[LWS98]*. The rescaled velocity fields are then saved to the hard-drive and serve as inflow fields in the main simulation. The whole processes is schematicaly illustrated in Fig. 1.2

From now on the subscript prec will be used to refer to the values obtained in the precursor simulation. The subscript infl will be used to refer to values at the inlet of the main simulation

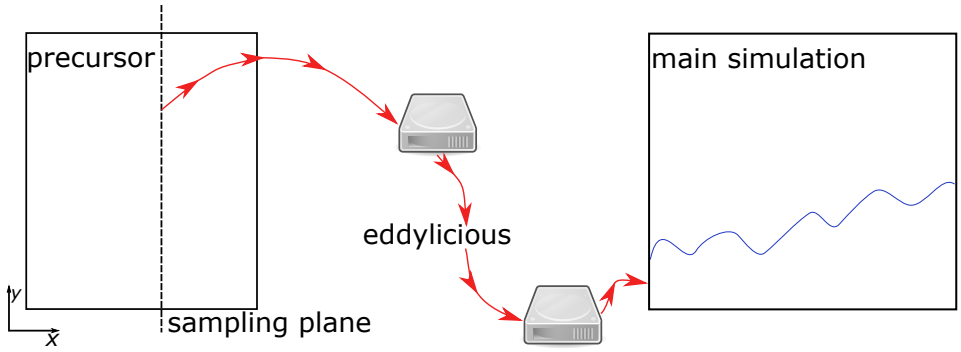

Fig. 1.2: Schematic showing how the rescaling proposed in *[LWS98]* is implemented in eddylicious.

The rescaling procedure is based on the assumption of existence of similarity solutions for the mean stream-wise velocity profile in the inner and outer layers of a TBL. The following relationships stem from this assumption.

$$U^{\text{inner}}(y^+) = u_\tau f_1(y^+),$$
$$U_0 - U^{\text{outer}}(\eta) = u_\tau f_2(\eta).$$

Another assumption, that is fulfilled automatically in the setting proposed by Lund et al *[LWS98]*, but not within the framework of eddylicious, is that the relationships above are valid for both the precursor simulation and the main simulation. Strictly speaking, this requires the precursor simulation to be a TBL itself. However, a flow sufficiently similar to a TBL, like channel flow, can also be used with success.

Let $\gamma = u_{\tau,\text{infl}}/u_{\tau,\text{prec}}$. Then, if the assumption above is fulfilled, the rescaling procedure for the mean streamwise velocity is

$$U^{\text{inner}}_{\text{infl}}(y^+_{\text{infl}}) = \gamma U^{\text{inner}}_{\text{prec}}(y^+_{\text{infl}}),$$
$$U^{\text{outer}}_{\text{infl}}(\eta_{\text{infl}}) = \gamma U^{\text{outer}}_{\text{prec}}(\eta_{\text{infl}}) + U_{0,\text{infl}} - \gamma U_{0,\text{prec}}.$$

The rescaling for the mean wall-normal velocity is defined simpler, and is not as rigorously based on any physical assumption.

$$V^{\text{inner}}_{\text{infl}}(y^+_{\text{infl}}) = V^{\text{inner}}_{\text{prec}}(y^+_{\text{infl}}),$$
$$V^{\text{outer}}_{\text{infl}}(\eta_{\text{infl}}) = V^{\text{outer}}_{\text{prec}}(\eta_{\text{infl}}).$$

The rescaling for the fluctuations is defined as

$$(u'_i)^{\text{inner}}_{\text{infl}}(y^+_{\text{infl}}) = \gamma (u'_i)^{\text{inner}}(y^+_{\text{infl}}),$$
$$(u'_i)^{\text{outer}}_{\text{infl}}(\eta_{\text{infl}}) = \gamma (u'_i)^{\text{outer}}(\eta_{\text{infl}}).$$

The inner and outer components are blended together using a weighted average:

$$u_{i,\text{infl}} = u^{\text{inner}}_{i,\text{infl}}[1 - W(\eta_{\text{infl}})] + u^{\text{outer}}_{i,\text{infl}} W(\eta_{\text{infl}}).$$

The weight function $W(\eta)$ is defined as

$$W(\eta) = \frac{1}{2}\left\{1 + \frac{\tanh\left(\frac{\alpha(\eta-b)}{(1-2b)\eta+b}\right)}{\tan(\alpha)}\right\},$$

where $\alpha = 4$ and $b = 0.2$.

Usage and practical information

The *runLundRescaling* script should be used to generate the fields. The script is parallelized using MPI, so it is possible to take advantage of all the available cores present on the machine.

Depending on what data is available for the TBL desired at the inlet it may be convinient to either use δ_{99} or θ as the outer scale (that is the length used to normalize y to obtain η). Eddylicous can work with both and will use the scale which is provided in the config file, i.e. one of the two should be present:

- `delta99` — desired δ_{99} at the inlet of the main simulation.
- `theta` — desired momentum thickness at the inlet of the main simulation.

Note that using θ requires to scale η before it can be plugged into function $W(\eta)$. The value of 8 is used, based on the fact that θ is around 8 times less than δ_{99} for a wide range of Reynolds numbers.

As evident from the equations, defining the rescaling procedure, the value of the friction velocity at the inlet, $u_{\tau,\text{infl}}$, is needed for the procedure. To this end, two options are available to the user. One is to simply provide the value of the friction velocity directly. The other is to let eddylicious compute it using the skin friction coefficient, c_f, and an empirical estimate connecting it to either $\text{Re}_{\delta_{99}}$ or Re_θ.

$$c_f = 0.02\text{Re}_{\delta_{99}}^{-1/6},$$
$$c_f = 0.013435(\text{Re}_\theta - 373.83)^{-2/11}.$$

The friction velocity is then obtained as $U_0\sqrt{c_f/2}$. The related parameter in the configuration file is

- `uTauInflow` — the friction velocity at the inlet of the main simulation. Either the value of the velocity or `compute`, which tells eddylicious to use one of the correlations above.

Another important feature is that eddylicious will always use only half of the datapoints in the wall-normal direction available from the precursor simulation. This is natural if the precursor is channel flow, but is in fact unnecessary when it comes to rescaling from another TBL simulation. Basically, this demands that the boundary layer used as a precusor does not occupy more than half of the computational domain in the wall-normal direction.

It is possible to choose which half of the precursor plane to consider, the bottom or the top. The following parameter in the configuration file controls this choice.

- `half` — which half of the precursor plane to grab the data from. Either `bottom` or `top`.

Note, that this means that a single channel flow precursor actually contains two independent precursor datasets.

The rescaling formulas involve the velocity from the precursor simulation evaluated for the values of y^+ and η defined by the TBL at the inflow of the main simulation. These values are obtained using linear interpolation. This means that the values of Re_τ and Re_θ for the precursor simulation must be higher than that at the inflow of the main simulation. Applied to rescaling from a precursor TBL this means that one can only rescale from "downstream".

In the current implementation, eddylicious will compute the highest value of η available for the precursor simulation. Then it will pick the points in the main simulation for which η is lower than this computed value. This ensures that interpolation is possible for the outer part of the profile. These chosen points will be considered as containing the inflow TBL. In all points above, the freestream velocity will be prescribed. If the range of η in the precusor is not sufficient to cover the whole inflow TBL, a jump in the mean streamwise velocity will be observed.

Note, that no similar procedure is performed for y^+. Therefore, if the range of y^+ in the precursor does not cover that in the inflow TBL, eddylicious will simply crash.

Besides for the parameters mentioned above, the configuration file should also define the following parameters.

- All parameters associated with the chosen input and output formats. Refer to the associated parts of the User guide for information.

- `yOrigin` — the wall-normal coordinate of wall which the boundary layer is attached to in the main simulation. This is used when evaluating non-dimensional coordinates like y^+. Also this is used to determine the "orientation" of the TBL with respect to the wall-normal coordinate.
- `nuInflow` — the kinematic viscosity value in the main simulation.
- `nuPrecursor` — the kinematic viscosity value in the precursor simulation.
- `U0` — desired freestream velocity at the inlet of the main simulation.
- `dt`— the time-step in the main simulation.
- `t0` — the start-time of the main simulation.
- `tEnd` — the end-time of the simulation.
- `tPrecision` — write precision for time values. Should be chosen according to `dt`.

Example configuration files can be found in the tutorial *Channel flow in OpenFOAM using Lund's rescaling*.

Interpolation

Theory

This is a simple generator that just interpolates data from one two-dimensional point set to another. This may be useful when some inflow data already generated and it should be applied for different inflow patches, discretized by a different mesh. In particular example can be applying the method proposed in *[ML17]*, where a channel flow precursor is used to generate inflow for a turbulent boundary layer simulation. Given precursor data, it only remains to interpolate it onto the mesh of the inflow boundary patch.

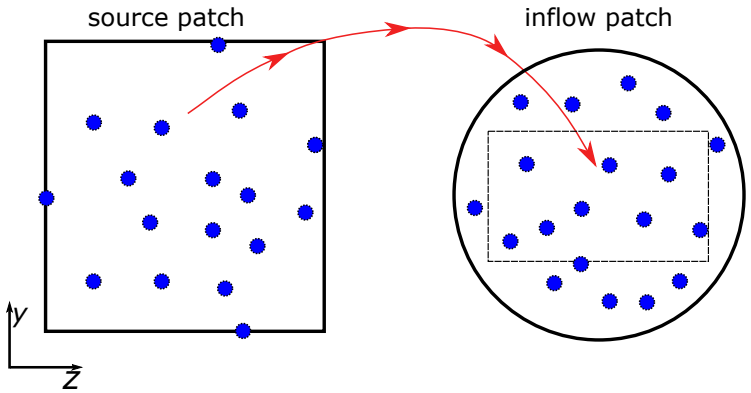

Fig. 1.3: Schematic showing how the interpolation is performed. Thick solid lines represent the geometry of the patches, Blue circles represent the grid points. For the inflow patch, a user-defined bounding box is used, this filtering out a part of the points.

The interpolation type used is linear. A bounding box is found for each set of points. Alternatively, the bounding boxes can be prescribed explicitly by the user, allowing to filter out a part of the points, see Fig. 1.3. The points are then scaled to lie in unit square prior to interpolation.

Usage and practical information

The *runInterpolation* script should be used to generate the fields. The script is parallelized using MPI, so it is possible to take advantage of all the available cores present on the machine.

As usual, all parameters associated with the chosen input and output formats should be included in the config file. Refer to the associated parts of the User guide for information. The following additional, optional, parameters can be included as well.

- `xOrigin` — the streamwise location of the inflow patch.
- `minYPrec`, `maxYPrec` — wall-normal bounds for the source points.
- `minZPrec`, `maxZPrec` — spanwise bounds for the source points.
- `minYInfl`, `maxYInfl` — wall-normal bounds for the target points.
- `minZInfl`, `maxZInfl` — spanwise bounds for the target points.

Supported input formats

Some of the methods for inflow generation require input. Typically these are precursor-based methods, such as the one based on the rescaling procedure presented in *[LWS98]*.

Also, the geometry of the inlet is defined via reading the appropriate data from a file.

Warning: At this point eddylicious only supports rectangular inlets meshed with a rectilinear grid. The inlet plane is assumed to be perpendicular to the flow direction, which, in turn, is assumed to be aligned with the x axis.

The foamFile format

This file format is associated with the CFD solver OpenFOAM. Data from a user-defined sampling surface can be saved in this format. OpenFOAM creates a catalog each time the data is output, named with the value of the simulation time. Inside this catalog yet another folder is created, named identically to the name of the sample-surface as defined by the user. At the root level of the catalog the data related to the mesh is stored. That includes the coordinates of the points, a list of faces each defined as list of points, and the coordinates of the face centres. Since the data resides in the face centres, they are used in eddylicious to represent the geometry of the surface. A folder `vectorField` is created to store field with vector-valued data. In particular, the field `U` which represents the velocity field, will be located there.

Let N_p be the total number of face centres. The file containing the face centres, called `faceCentres`, has the following simple format.

The format of the face centres file

$$
\begin{aligned}
&N_p \\
&(\\
&(x_0,\ y_0,\ z_0) \\
&(x_1,\ y_1,\ z_1) \\
&(x_2,\ y_2,\ z_2) \\
&\vdots \\
&(x_{p-1},\ y_{p-1},\ z_{p-1}) \\
&)
\end{aligned}
$$

The format of the file containing a sampled vector field is identical, however instead of the coordinates each row contains the three components of the vector.

The order in which the data is written corresponds to the order in which the face centres are written to `faceCentres`.

In order for eddylicious to read in the geometry of the inlet stored as a list of face centres in the foamFile format the following should be added to the configuration file.

```
inflowGeometryReader    foamFile
inflowGeometryPath      /path/to/the/faceCentres/file
xOrigin                 the coordinate of the inlet on the axis parallel to
                        the main streamwise direction
```

In order for eddylicious to read in previously sampled velocity fields stored in the foamFile format the following should be added to the configuration file.

```
reader                  foamFile
readPath                /path/to/OpenFOAM/case
sampleSurfaceName       name of sample surface as defined in controlDict
```

Eddylicious will search for the catalogs containing the data for different time steps in `readPath/postProcessing/sampledSurface/*time_value*/sampleSurfaceName`.

Important: This offering is not approved or endorsed by OpenCFD Limited, producer and distributor of the OpenFOAM software and owner of the OPENFOAM® and OpenCFD® trade marks.

The HDF5 format

HDF5 is a file format specifically developed for storing large scientific datasets. More details regarding HDF5 can be found here.

In HDF5, data can be sorted into groups. The data itself is stored in the form of datasets. One can think of datasets as of multidimensional arrays.

In eddylicious the file is expected to contain two groups: `points` and `velocity`.

Let N_y and N_z be the number of points in inlet plane in the wall-normal and spanwise direction respectively.

The `points` group contains the following two-dimensional datasets:

- `pointsY`, $N_y \times N_z$ — dataset with the wall-normal coordinates of the points.
- `pointsZ`, $N_y \times N_z$ — dataset with the spanwise coordinates of the points.

This structure implies, that all the columns of `pointsY` are identical, as well as all the rows of `pointsZ`.

Let N_t be the amount of time steps for which velocity data is available.

The `velocity` group contains the following three-dimensional datasets:

- `uX`, $N_t \times N_y \times N_z$ — dataset with the values of the streamwise component of the velocity field.
- `uY`, $N_t \times N_y \times N_z$ — dataset with the values of the wall-normal component of the velocity field.
- `uZ`, $N_t \times N_y \times N_z$ — dataset with the values of the spanwise component of the velocity field.

The values located at position `[k, i, j]` in these arrays correspond to the point with coordinates (`pointsY[i, j]`, `pointsZ[i,j]`).

Also, the following one-dimensional arrays are stored in the `velocity` group:

- `uMeanX`, N_y — the values of the mean streamwise velocity.
- `uMeanY`, N_y — the values of the mean wall-normal velocity.
- `times`, N_t — the time values associated with the velocity fields.

The way the data is stored in the HDF5 file coincides with how it is represented internally. This implies that the overhead from reading the data is minimal. HDF5 supports parallel processing of the data via MPI. Different processes can therefore read in the required data simultaneously.

Therefore, this file format can be considered optimal. Since solvers will not typically support output in this particular format, utilities for converting a precursor database saved in a different format into the HDF5 format are part of eddylicious.

In order for eddylicious to read in previously generated velocity fields stored as an HDF5 file, the following should be added to the configuration file.

```
reader                  hdf5
readPath                /path/to/hdf5/file
```

Supported output formats

Eddylicious supports several file formats for outputting the generated inflow fields. The choice of the file format is usually dictated by the CFD solver.

The HDF5 file format

HDF5 is a file format specifically developed for storing large scientific datasets. More details regarding HDF5 can be found here.

In HDF5 data can be sorted into groups. The data itself is stored in the form of datasets. One can think of datasets as of multidimensional arrays.

Let N_p be the number of points at which the inflow fields are generated and N_t the amount of time-values for which the inflow fields are generated. Eddylicious creates the following datasets inside the HDF5 file.

- `points`, $N_p \times 3$ — the points associated with the values of the inflow fields. The three columns represent the x, y, and z coordinates respectively.
- `times`, $N_t \times 1$ — the time values associated with the inflow fields.
- `velocity`, $N_t \times N_p \times 3$ — the values of the velocity field. The first index is associated with time, the second with the available points and the third one with the components of the velocity field. Same order as in `points` applies.

The following parameters need to be provided in the configuration file in order to output the velocity fields in the HDF5 file format.

```
writer          hdf5
writePath       path to the where the database will be stored
hdf5FileName    name of the hdf5 file
```

OpenFOAM native format

This is natively supported by OpenFOAM. In order to read in boundary data from the hard-drive OpenFOAM has a special boundary conidtion called `timeVaryingMappedFixedValue`. This boundary condition expects a folder

called `boundaryData/\<patchname\>` to be located in the `constant` directory of the case. Inside the folder a file named `points` should reside. This file provides a list of the points where the boundary data is available. The boundary data itself resides in folders named as the time-value associated with the data. The data for each available field is stored in its own file named identically to the internal name of the field in OpenFOAM (for instance `U` for the velocity field). The format of each such file is quite similar to the *The foamFile format* but has some additional headers.

The following parameters need to be provided in the configuration file in order to output the velocity fields in the OpenFOAM native format.

```
writer          ofnative
writePath       /path/to/OpenFOAM/case
inletPatchName  name of the inlet patch
```

Utilities

inflowStats

This utility allows to compute mean velocity and the diagonal components of the Reynolds stress tensor of a database of inflow fields. The database should be stored as an HDF5 file, see *The HDF5 file format* in *Supported output formats*.

The utility accepts two command line arguments.

- `--database, -d` — the HDF5 file containing the inflow fields.
- `--writepath, -w` — the location where to write the files containing the computed results.

It is possible to run it in parallel using MPI.

The utility will create the following files in the location specified by `writePath`

- `uMeanX`, `uMeanY`, `uMeanZ` — contain the corresponding component of the mean velocity field.
- `uPrime2MeanXX`, `uPrime2MeanYY`, `uPrime2MeanZZ` — contain the corresponding diagonal component of the Reynolds stress tensor.
- `y` — the locations of the datapoints.

precursorStats

This utility allows to compute mean velocity and the diagonal components of the Reynolds stress tensor of a precursor database. The database should be stored as an HDF5 file, see *The HDF5 format* in *Supported input formats*.

The utility accepts two command line arguments.

- `--database, -d` — the HDF5 file containing the inflow fields.
- `--writepath, -w` — the location where to write the files containing the computed results.

It is possible to run it in parallel using MPI.

The utility will create the following files in the location specified by `writePath`

- `uMeanX`, `uMeanY`, `uMeanZ` — contain the corresponding component of the mean velocity field.
- `uPrime2MeanXX`, `uPrime2MeanYY`, `uPrime2MeanZZ` — contain the corresponding diagonal component of the Reynolds stress tensor.
- `y` — the locations of the datapoints.

convertFoamFileToHDF5

This utility converts a precursor database stored in the foamFile format (see *The foamFile format*) to a database stored as a single HDF5 file, see *The HDF5 file format*.

The utility accepts the following command line arguments.

- `--precursor` — The location of the precusor OpenFOAM case. This path will be used in order to locate the folder with the sampled velocity values.
- `--surface` — The name of the surface that was used for sampling the values. The name is chosen in the `cotrolDict` of the case.
- `--filename` — the name of the HDF5 file that will be produced.
- `--umean` — The file containing the mean velocity profile. The file is assumed to have two or three columns, one with wall-normal coordinate, the second one with the values of mean streamwise velocity, and optionally a third one with the mean wall-normal velocity. If only two columns are present the mean wall-normal velocity is assumed to be zero.

It is possible to run the utility in parallel using MPI.

Workflow guidelines

Here the suggested workflows for using eddylicious in conduction with various solvers are presented. Basically, whatever solver is used, the following steps have to be performed.

- Specifying the geometry of the inlet for eddylicious.
- Choosing an output format that is compatible with the used solver.
- Generating the inflow fields by running the python script associated with the chosen inflow generation method.
- Setting up the solver to read in boundary data from the hard drive.

Using eddylicious with OpenFOAM

Important: This offering is not approved or endorsed by OpenCFD Limited, producer and distributor of the OpenFOAM software and owner of the OPENFOAM® and OpenCFD® trade marks.

Specifying the geometry of the inlet

Specifying the geometry boils down to producing the list of face centres located at the inlet boundary. The coordinates of the face centres can be used using the `sample` utility shipped with OpenFOAM.

In the `system/sampleDict` file, create a sampling surface with the type `patch`, and specify the inlet patch as the basis for the surface. It is better to turn off triangulation to preserve the original geometry. Choose `foamFile` as the write format for surfaces.

Even though we are only interested in the geometry, a field for sampling has to be chosen. Any field can be chosen, besides for the velocity field `U`. This is because the sample utility will attempt to read in the field, and, since we didn't generate it yet, the field-values simply don't exist yet.

A `sampleDict` for a case with the inlet patch named `inlet` might look something like this.

```
*---------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  2.3.1                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      sampleDict;
}
surfaceFormat foamFile;

fields
(
   p
);

surfaces
(
    inlet
    {
        type patch;
        patches (inlet);
        interpolate false;
        triangulate false;
    }

);
```

The produced file can be read using the `foamFile` inflow geometry reader. The path to the `faceCenters` file should also be provided. This is done by adding the following lines to the configuration file for the inflow generation script.

```
inflowGeometryReader    foamFile
inflowGeometryPath      "/path/to/faceCenters/file"
```

Reading the inflow fields from OpenFOAM

OpenFOAM has a special boundary condition that allows reading boundary data from a file, it is called `timeVaryingMappedFixedValue`. A tutorial, which takes advantage of this boundary condition, is shipped with OpenFOAM. It can be found under `tutorials/incompressible/simpleFoam/pitzDailyExptInlet/` .

The boundary condition is quite flexible. If needed, interpolation in space will be used to obtain the vales at the face centres from the values at the provided points. Linear interpolation in time is also supported.

Let `inlet` be the name of the patch for which the inflow fields are generated. Then the following entry should be found in the `U` file.

```
inlet
{
    type            timeVaryingMappedFixedValue;
    offset          (0 0 0);
```

```
    setAverage      off;
    perturb         0;
}
```

Setting `perturb` to 0 is important, since this option perturbs the location of the points.

In order to generate the inflow fields the *OpenFOAM native format* should be used for writing the velocity fields to the hard drive.

Note that, for a large time-span, the amount of files written to disk become extremely large. To rectify this issue, a modified version of `timeVaryingMappedFixedValue` that reads all the data from a single HDF5 file is available. For more information regarding the structure of the file see *The HDF5 file format*.

The modified boundary condition is called `timeVaryingMappedHDF5FixedValue` and can be downloaded at https://bitbucket.org/lesituu/timevaryingmappedhdf5fixedvalue

If this boundary condition is used the entry in the `U` file should look as follows.

```
inlet
{
    type            timeVaryingMappedHDF5FixedValue;
    setAverage      false;
    perturb         0;
    offset          (0 0 0);
    hdf5FileName    nameofthehdf5file.hdf5;
    hdf5PointsDatasetName    points;
    hdf5SampleTimesDatasetName    time;
    hdf5FieldValuesDatasetName    velocity;
}
```

CHAPTER 2

Tutorials

Channel flow in OpenFOAM using Lund's rescaling

Introduction

In this tutorial, channel flow will be simulated. First, a precursor simulation will be setup, which will compute channel flow using periodic boundary conditions in both the streamwise and spanwise directions.

The velocity fields created by the precursor will be used as input for the rescaling procedure developed by Lund et al *[LWS98]*. It will be used to generate the inflow velocity field for the main simulation, which is also channel flow, but with velocity inlet/poressure outlet boundaries in the streamwise direction.

After completing this tutorial you will be able to do the following.

- Set-up a precursor channel flow simulation in OpenFOAM.
- Use the rescaling method developed by Lund et al to generate inflow fields for the main simulation.
- Read in the boundary data generated by Eddylicious in OpenFOAM.

Essentially, the guidelines found in *Using eddylicious with OpenFOAM* are applied here to the concrete case of channel flow and Lund's rescaling procedure to generate the inflow velocity field.

The tutorial cases have been tested with OpenFOAM 2.3.1. It is assumed that the user has some experience in running and setting up simulations in OpenFOAM and is familiar with associated terminology.

Overview of the set-up

As described in the introduction, the goal of this tutorial is to conduct a channel flow simulation using OpenFOAM and eddylicious. Channel flow is a flow between two infinite parallel plates driven by a pressure gradient. The flow is fully defined by the friction velocity-based Reynolds number $\mathrm{Re}_\tau = u_\tau \delta / \nu$, where u_τ is the friction velocity, ν is the kinematic viscosity, and δ is the half-width of the channel.

The computational cost of the simulation grows with Re_τ, therefore in this tutorial we will use the lowest Reynolds number for which DNS data is available, namely $\mathrm{Re}_\tau = 180$. This will allow using a mesh fine enough to resolve a

big part of the turbulent structures present in the fow, yet small enough for the case to be computed in a reasonable time on a single workstation.

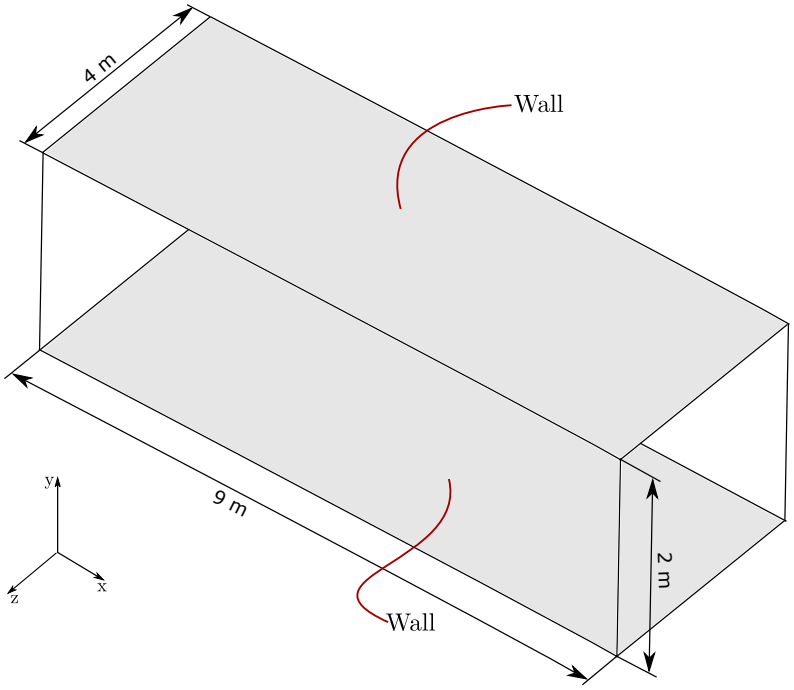

Fig. 2.1: Channel flow domain.

The common way to set-up a channel flow simulation is to create a rectangular domain of a size sufficiently large to contain the largest turbulent structures present in the flow. The domain used in the tutorial is shown in Fig. 2.1. The size of the domain is $L_x \times 2\delta \times L_z = 9 \times 2 \times 4$.

To simulate an infinite domain, periodic boundary conditions are commonly applied in the stream- and spanwise directions. The pressure gradient is then introduced via an extra forcing term in the momentum equations.

However, for the purpose of testing an inflow field generation method, a velocity inlet can be used as the boundary condition at $x = 0$, and a pressure outlet at $x = L_x$.

The peculiarity of this tutorial is that the chosen inflow generation method itself requires us to compute another channel flow (see *Lund's rescaling*), which will serve as a precursor simulation for the "main" channel flow simulation. In the precursor simulation periodic boundaries in both stream- and spanwise direction will be used. The same mesh will be used in both simulations, and in both simulations the Reynolds number Re_τ will be set to 180.

Such a set-up might seem completely meaningless, but in fact it can be used as a reference solution within a simulation campaign that tests various inflow generation methods *[KPBK04]*.

The precursor simulation

The first part of the tutorial will deal with setting up a precursor simulation that will be later used to generate the inflow fields for the main simulation. Please follow the following steps.

1. Unpack `channel_flow_lund_rescaling.zip` found in `eddylicious/tutorials` to a location of your preference (commonly `run`). Two folders will be unpacked, `precursor` and `main`. Go inside of the `precursor` folder.

2. Let us explore the case. Data for time 1000 is available, that is the case has been pre-run to get rid of transients, therefore one can proceed with saving the velocity fields needed for the rescaling procedure directly.

   Opening `1000/U` verifies that cycling boundary conditions are applied in both streamwise and spanwise directions.

```
inlet
{
   type    cyclic;
}

outlet
{
   type    cyclic;
}

left
{
   type    cyclic;
}

right
{
   type    cyclic;
}
```

   In `system/fvOptions` the mean streamwise velocity `Ubar` is prescribed. This is equivalent to prescribing a pressure gradient.

```
momentumSource
{
   type            pressureGradientExplicitSource;
   active          on;
   selectionMode   all;

   pressureGradientExplicitSourceCoeffs
   {
       fieldNames  (U);
       Ubar        (1.0 0 0 );
   }
}
```

3. In the `controlDict` a `surfaces` function object is used to save the velocity field from the inlet patch to a file at every time-step.

```
sampledSurface
{
    type surfaces;
    outputControl timeStep;
    outputInterval 1;
    enabled true;

    surfaceFormat foamFile;
    interpolationScheme none;
    interpolate false;
    triangulate false;

    fields
    (
```

```
        U
    );

    surfaces
    (
        inletSurface
        {
            type patch;
            patches (inlet);
        }
    );
}
```

The appropriate options make sure that the surface is not triangulated, and that no interpolation of the data is performed, we therefore save all the raw values at all the face centres. The `foamFile` format is chosen, since eddylicious can read in data stored in that format.

4. Run `blockMesh` in order to create the mesh. It is recommended that you run the case using 4 processors. You can, however, modify this value in `system/decomposeParDict`. In order to decompose the mesh run `decomposePar`.
5. Now everything is ready to run the case. The solver `pimpleFoam` will be used. Execute the solver. If you are running in parallel, add the `-parallel` flag and execute the solver with your MPI executable and the appropriate number of cores as an argument. Running the case will take a while. After the execution is complete, run `reconstructPar -latestTime` if you've run in parallel.
6. Run `postChannelFlow` (available at https://bitbucket.org/lesituu/postchannelflow) to get the mean velocity and the components of the Reynolds stress tensor averaged along the streamwise and spanwise directions. The setting to the utility are provided in the `constant/postChannelDict` file. One-dimensional profiles are output in the `postProcessing/collapsedFields` directory.
7. If you wish, you can compare the results to the DNS *[LM15]*. The DNS data can be found inside the `postProcessing` directory, in the files `dns_mean.dat` and `dns_fluct.dat`. The original archive can be found at the following address http://turbulence.ices.utexas.edu/channel2015/content/Data_2015_0180.html

   The `post_processing.py` script contains simple code to plot various quanties and compare the to the DNS. The script is found in the `postProcessing` folder as well. But feel free to use your own favorite software to post-process the results.

The main simulation

Now we can proceed with the main simulation that will use the velocity fields sampled in the precursor. The inlet of the main simulation is divided into two patches: `inletBot` and `inletTop`.

In `0/U` the boundary condition for velocity at the inlets is defined as `timeVaryingMappedFixedValue`

```
inletBot
{
   type            timeVaryingMappedFixedValue;
   setAverage      false;
   perturb         0;
   offset          (0 0 0);
}
inletTop
{
   type            timeVaryingMappedFixedValue;
```

```
    setAverage      false;
    perturb         0;
    offset          (0 0 0);
}
```

This allows to read in the velocity values from files located in `constant/boundaryData`, see *OpenFOAM native format*.

1. Go to the case `main`. Run `blockMesh` to create the mesh.
2. In order to provide eddylicious the coordinates of the face centres at the inlet plane we use the `sample` utility. In the `system/sampleDict` file two surfaces coinciding with the inlet patches are defined. Run the utility. This will create a `faceCentres` file for each inlet patch in the `postProcessing/sampledSurfaces/0/*patchname*` directories.
3. Inflow velocity fields are generated for each inlet patch separately. The generation procedure for each patch is controlled by a configuration file. One file for each inlet patch, `rescalingConfigBot` and `rescalingConfigTop` for the `inletBot` and `inletTop` patch respectively. Explore the config files. See *Lund's rescaling* and other relative parts of the User guide to make sure you understand what each option stands for. Note that the chosen values of u_τ, δ_{99} and ν are chosen coincide with the ones in the precursor simulation.
4. Run `runLundRescaling --config=rescalingConfigBot`. The script will write out some integral properties of the precursor, perform the rescaling and then write out similar properties for the generated inflow fields. The properties of the precursor and the main simulation are almost identical, as is intended. Run `runLundRescaling --config=rescalingConfigTop`. Note that the `constant/boundaryData` now contains two directories corresponding to the two inlet pathes. Inside, the generated inflow fields are stored.
6. If possible, decompose the case using `decomposePar`. Run it using `pimpleFoam`. Reconstruct the fields using `reconstructPar` if you've run in parallel.
7. Explore the solution using you favorite post processing software! In particular, see if the soluton convreges to the one obtained in the precursor.

Important: This offering is not approved or endorsed by OpenCFD Limited, producer and distributor of the OpenFOAM software and owner of the OPENFOAM® and OpenCFD® trade marks.

CHAPTER 3

Developer guide

Contributing

Your contributions to the project are most welocome! The best way to precede is open an issue on Github where you can describe the contribution you have in mind. Then it can be discussed with the community.

In order to ensure that the overall quality of the library is maintained, the following requirements have to be fulfilled for the contribution to be accepted:

- The code has to adhere to the existing style guidelines.
- The new functionality has to be covered by unit-tests.
- The new functionality has to be properly documented. This includes docstrings and contributions to the relevant parts of the User guide.
- Ideally, a tutorial showcasing the usage of the new functionality should be provided.

Code and documentation style

Please use the following guidelines when extending eddylicious.

- The suggestions in PEP8 should be followed except for the variable name-style convention.
- Variables should use named using the `mixedCase` style.
- Docstrings should be written using the Numpy docstring standard . Examples are available here.
- In the restructuredText files used for documentation, three spaces are used for indenting.

CHAPTER 4

Source code reference

Readers

Module containing functions for reading in data from various file formats.

eddylicious.readers.**read_structured_points_foamfile**(*readPath*, *addValBot=nan*, *addValTop=nan*, *excludeBot=0*, *excludeTop=0*, *exchangeValBot=nan*, *exchangeValTop=nan*)

Read the coordinates of the points from a foamFile-format file.

Reads in the locations of the face centers, stored in foamFile format by OpenFOAM, and transforms them into 2d numpy arrays.

The points are sorted so that the axes of the arrays correspond to the wall-normal and spanwise directions. The points are first sorted along y, then reshaped into a 2d array and then sorted along z for each value of z.

The function supports manipulating the points in certain ways, see the parameter list below.

Parameters

- **readPath** (*str*) – The path to the file containing the points.
- **addValBot** (*float, optional*) – Append a row of values from below, nothing added by default.
- **addValTop** (*float, optional*) – Append a row of values from above, nothing added by default.
- **excludeBot** (*int, optional*) – How many points to remove from the bottom in the y direction. (default 0).
- **excludeTop** (*int, optional*) – How many points to remove from the top in the y direction. (default 0).
- **exchangeValBot** (*float, optional*) – Exchange the value of y at the bottom.
- **exchangeValTop** (*float, optional*) – Exchange the value of y at the top.

Returns

The list contains 4 items

pointsY : A 2d ndarray containing the y coordinates of the points.

pointsZ : A 2d ndarray containing the z coordinates of the points.

indY : The sorting indices from the sorting performed.

indZ : The sorting indices from the sorting performed.

Return type List of ndarrays

eddylicious.readers.**read_structured_velocity_foamfile**(*baseReadPath*, *surfaceName*, *nPointsZ*, *yInd*, *zInd*, *addValBot=(nan*, *nan*, *nan)*, *addValTop=(nan*, *nan*, *nan)*, *excludeBot=0*, *excludeTop=0*, *interpValBot=False*, *interpValTop=False*)

Read the values of the velocity field from a foamFile-format file.

Reads in the values of the velocity components stored in the foamFile file-format. The velocity field is read and the transformed into a 2d ndarray, where the array's axes correspond to wall-normal and spanwise directions. To achieve this, the sorting indices obtained when reordering the mesh points are used.

Some manipulation with the read-in data is also available via the optional parameters.

Parameters

- **baseReadPath** (*str*) – The path where the time-directories with the velocity values are located.
- **surfaceName** (*str*) – The name of the surface that was used for sampling.
- **nPointsZ** (*int*) – The amount of points in the mesh in the spanwise direction.
- **yInd** (*ndarray*) – The sorting indices for sorting in the wall-normal direction.
- **zInd** (*ndarray*) – The sorting indices for sorting in the spanwise direction.
- **addValBot** (*tuple of three floats, optional*) – Append a row of values from below.
- **addValTop** (*tuple of three floats, optional*) – Append a row of values from above.
- **excludeBot** (*int, optional*) – How many points to remove from the bottom in the y direction. (default 0).
- **excludeTop** (*int, optional*) – How many points to remove from the top in the y direction. (default 0).
- **interpValBot** (*bool, optional*) – Whether to interpolate the first value in the wall-normal direction using two points. (default False)
- **interpValTop** (*bool, optional*) – Whether to interpolate the last value in the wall-normal direction using two points. (default False)

Returns A function of one variable (the time-value) that will actually perform the reading.

Return type function

eddylicious.readers.**read_points_foamfile**(*readPath*)

Read the coordinates of the points from a foamFile-format file.

Reads in the locations of the face centers, stored in foamFile format by OpenFOAM.

Parameters **readPath** (*str*) – The path to the file containing the points.

Returns Two arrays corresponding to y and z components of the points.

Return type List of ndarrays

eddylicious.readers.**read_velocity_foamfile**(*baseReadPath*, *surfaceName*)

Read the values of the velocity field from a foamFile-format file.

Reads in the values of the velocity components stored in the foamFile file-format.

Parameters

- **baseReadPath** (*str*) – The path where the time-directories with the velocity values are located.
- **surfaceName** (*str*) – The name of the surface that was used for sampling.

Returns A function of one variable (the time-value) that will actually perform the reading.

Return type function

eddylicious.readers.**read_structured_points_hdf5**(*readPath*, *addValBot=nan*, *addValTop=nan*, *excludeBot=0*, *excludeTop=0*, *exchangeValBot=nan*, *exchangeValTop=nan*)

Read the coordinates of the points from a hdf5 file.

Reads in the locations of the face centers, stored in a hdf5 file.

The function supports manipulating the points in certain ways, see the parameter list below.

Parameters

- **readPath** (*str*) – The path to the file containing the points
- **addValBot** (*float, optional*) – Whether to append a row of zeros from below.
- **addValTop** (*float, optional*) – Whether to append a row of zeros from above.
- **excludeBot** (*int, optional*) – How many points to remove from the bottom in the y direction. (default 0).
- **excludeTop** (*int, optional*) – How many points to remove from the top in the y direction. (default 0).
- **exchangeValBot** (*float, optional*) – Exchange the value of y at the bottom.
- **exchangeValTop** (*float, optional*) – Exchange the value of y at the top.

Returns

The list contains 2 items

pointsY : A 2d ndarray containing the y coordinates of the points.

pointsZ : A 2d ndarray containing the z coordinates of the points.

Return type List of ndarrays

eddylicious.readers.**read_structured_velocity_hdf5**(*readPath*, *addValBot=(nan, nan, nan)*, *addValTop=(nan, nan, nan)*, *excludeBot=0*, *excludeTop=0*, *interpValBot=False*, *interpValTop=False*)

Read the values of the velocity field from a foamFile-format file.

Reads in the values of the velocity components stored as in hdf5 file format.

Some manipulation with the read-in data is also available via the optional parameters.

Parameters

- **readPath** (*str*) – The path to the file containing the velocity field.
- **addValBot** (*tuple of three floats, optional*) – Append a row of values from below.
- **addValTop** (*tuple of three floats, optional*) – Append a row of values from above.
- **excludeBot** (*int, optional*) – How many points to remove from the bottom in the y direction. (default 0).
- **excludeTop** (*int, optional*) – How many points to remove from the top in the y direction. (default 0).
- **interpValBot** (*bool, optional*) – Whether to interpolate the first value in the wall-normal direction using two points. (default False)
- **interpValTop** (*bool, optional*) – Whether to interpolate the last value in the wall-normal direction using two points. (default False)

Returns A function of one variable (the time-index) that will actually perform the reading.

Return type function

Writers

Module containing functions for writing out the inflow fields in various file formats.

eddylicious.writers.**write_points_to_hdf5**(*hdf5File*, *pointsY*, *pointsZ*, *xVal*)

Write the points into a HDF5 file.

Savs the points into a HDF5 file. The points will be transformed into 1d arrays. The resulting dataset is called points and lies in the root of the file.

Parameters

- **hdf5File** (*h5py.File*) – The path of the HDF5 file.
- **pointsY** (*ndarray*) – A 2d array containing the values of y for the face centres.
- **pointsZ** (*ndarray*) – A 2d array containing the values of z for the face centres.
- **xVal** (*float*) – The x-location of the inflow plane.

eddylicious.writers.**write_velocity_to_hdf5**(*hdf5File*, *t*, *uX*, *uY*, *uZ*, *iteration*)

Write the velocity field into an HDF5 file.

Will also write the corresponding time value.

Parameters

- **hdf5File** (*h5py.File*) – The the HDF5 file.
- **t** (*float*) – The value of time associated with the written velocity field.
- **uX** (*ndarray*) – A 2d ndarray containing the streamwise component of the velocity field.
- **uY** (*ndarray*) – A 2d ndarray containing the wall-normal component of the velocity field.
- **uZ** (*ndarray*) – A 2d ndarray containing the spanwise component of the velocity field.
- **iteration** (*int*) – The position of along the time axis.

eddylicious.writers.**write_points_to_ofnative**(*writePath*, *pointsY*, *pointsZ*, *xVal*)
Write the points in a format used by OpenFOAM's timeVaryingMappedFixedValue boundary condition.

Parameters

- **writePath** (*str*) – The path where to write the points file. Should commonly be constant/boundaryData/nameOfInletPatch.
- **pointsY** (*ndarray*) – A 2d array containing the values of y for the face centres.
- **pointsZ** (*ndarray*) – A 2d array containing the values of z for the face centres.
- **xVal** (*float*) – The x-location of the inflow plane.

eddylicious.writers.**write_velocity_to_ofnative**(*writePath*, *t*, *uX*, *uY*, *uZ*)
Write the velocity field in a format used by OpenFOAM's timeVaryingMappedFixedValue boundary condition.

Parameters

- **writePath** (*str*) – The path where to write the time directories containing the U files. Commonly constant/boundaryData/nameOfInletPatch.
- **t** (*float*) – The value of time associated with the written velocity field.
- **uX** (*ndarray*) – A 2d ndarray containing the streamwise component of the velocity field.
- **uY** (*ndarray*) – A 2d ndarray containing the wall-normal component of the velocity field.
- **uZ** (*ndarray*) – A 2d ndarray containing the spanwise component of the velocity field.

Generators

Module containing functions for the generation of inflow fields.

eddylicious.generators.**blending_function**(*eta*, *alpha=4*, *b=0.2*)
Return the value of the blending function W for Lund's rescaling.

Return the value of the blending function for the inner and and outer profiles produced by Lund's rescaling. For eta>1 the function returns 1.

Parameters

- **eta** (*ndarray*) – The values of the non-dimensionalized wall-normal coordinate.
- **alpha** (*float, optional*) – The value of alpha (default 4.0).
- **b** (*float*) – The value of b (default 0.2).

eddylicious.generators.**delta_99**(*y*, *v*)
Compute δ_{99}.

Parameters

- **y** (*ndarray*) – The independent variable.

- **v** (*ndarray*) – The velocity values.

Returns The value of δ_{99}.

Return type float

eddylicious.generators.**delta_star**(*y*, *v*)
Compute the displacement thickness using Simpson's method.

Parameters

- **y** (*ndarray*) – The independent variable.
- **v** (*ndarray*) – The velocity values.

Returns The value of the displacement thickness.

Return type float

eddylicious.generators.**momentum_thickness**(*y*, *v*)
Compute the momentum thickness using Simpson's method.

Parameters

- **y** (*ndarray*) – The independent variable.
- **v** (*ndarray*) – The velocity values.

Returns The value of the momentum thickness.

Return type float

eddylicious.generators.**chunks_and_offsets**(*nProcs*, *size*)
Given the size of a 1d array and the number of processors, compute chunk-sizes for each processor and the starting indices (offsets) for each processor.

Parameters

- **nProcs** (*int*) – The amount of processors.
- **size** (*int*) – The size of the 1d array to be distributed.

Returns The first array contains the chunk-size for each processor. The second array contains the offset (starting index) for each processor.

Return type List of two ndarrays.

eddylicious.generators.**lund_rescale_mean_velocity**(*etaPrec*, *yPlusPrec*, *uMeanXPrec*, *uMeanYPrec*, *nInfl*, *etaInfl*, *yPlusInfl*, *nPointsZInfl*, *u0Infl*, *u0Prec*, *gamma*, *blending*)
Rescale the mean velocity profile using Lunds rescaling.

This function rescales the mean velocity profile taken from the precursor simulation using Lund et al's rescaling.

Parameters

- **etaPrec** (*ndarray*) – The values of eta for the corresponding values of the mean velocity from the precursor.
- **yPlusPrec** (*ndarray*) – The values of y+ for the corresponding values of the mean velocity from the precursor.
- **uMeanXPrec** (*ndarray*) – The values of the mean streamwise velocity from the precursor.
- **uMeanYPrec** (*ndarray*) – The values of the mean wall-normal velocity from the precursor.

- **nInfl** (*int*) – The amount of points in the wall-normal direction that contain the boundary layer at the inflow boundary.
- **etaInfl** (*ndarray*) – The values of eta for the mesh points at the inflow boundary.
- **yPlusInfl** (*ndarray*) – The values of y+ for the mesh points at the inflow boundary.
- **nPointsZInfl** (*int*) – The amount of points in the spanwise direction for the inflow boundary.
- **u0Infl** (*float*) – The freestream velocity at the inflow.
- **u0Prec** (*float*) – The freestream velocity for the precursor.
- **gamma** (*float*) – The ratio of the friction velocities in the inflow boundary layer and the precursor.
- **blending** (*ndarray*) – The weights for blending the inner and outer profiles.

Returns

- *uX, ndarray* – The values of the mean streamwise velocity.
- *uY, ndarray* – The values of the mean wall-normal velocity.

eddylicious.generators.**lund_rescale_fluctuations**(*etaPrec*, *yPlusPrec*, *pointsZ*, *uPrimeX*, *uPrimeY*, *uPrimeZ*, *gamma*, *etaInfl*, *yPlusInfl*, *pointsZInfl*, *nInfl*, *blending*)

Rescale the fluctuations of velocity using Lund et al's rescaling.

This function rescales the fluctuations of the three components of the velocity field taken from the precursor simulation using Lund et al's rescaling.

Parameters

- **etaPrec** (*ndarray*) – The values of eta for the corresponding values of the mean velocity from the precursor.
- **yPlusPrec** (*ndarray*) – The values of y+ for the corresponding values of the mean velocity from the precursor.
- **pointsZ** (*ndarray*) – A 2d array containing the values of z for the points of the precursor mesh.
- **uPrimeX** (*ndarray*) – A 2d array containing the values of the fluctuations of the x component of velocity.
- **uPrimeY** (*ndarray*) – A 2d array containing the values of the fluctuations of the y component of velocity.
- **uPrimeZ** (*ndarray*) – A 2d array containing the values of the fluctuations of the z component of velocity.
- **gamma** (*float*) – The ratio of the friction velocities in the inflow boundary layer and the precursor.
- **etaInfl** (*ndarray*) – The values of eta for the mesh points at the inflow boundary.
- **yPlusInfl** (*ndarray*) – The values of y+ for the mesh points at the inflow boundary.
- **pointsZInfl** (*ndarray*) – A 2d array containing the values of z for the points of the inflow boundary.
- **nInfl** (*int*) – The amount of points in the wall-normal direction that contain the boundary layer at the inflow boundary.

- **blending** (*ndarray*) – The weights for blending the inner and outer profiles.

Returns The list contains three items, each a 2d ndarray. The first array contains the rescaled fluctuations of the x component of velocity. The second – of the y component of velocity. The third – of the z component of velocity.

Return type List of ndarrays

eddylicious.generators.**lund_generate**(*readerFunction*, *writer*, *writePath*, *dt*, *t0*, *tEnd*, *timePrecision*, *uMeanXPrec*, *uMeanXInfl*, *uMeanYPrec*, *uMeanYInfl*, *etaPrec*, *yPlusPrec*, *pointsZ*, *etaInfl*, *yPlusInfl*, *pointsZInfl*, *nInfl*, *gamma*, *times*, *blending*)

Generate the the inflow velocity using Lund's rescaling.

This function will use Lund et al's rescaling in order to generate velocity fields for the inflow boundary. The rescaling for the mean profile should be done beforehand and is one of the input parameters for this function.

Parameters

- **readerFunction** (*function*) – The function to use for reading in data, generated by the reader. Should contain the reader's name in the attribute "reader".
- **writer** (*str*) – The writer that will be used to save the values of the velocity field.
- **writePath** (*str*) – The path for the writer.
- **dt** (*float*) – The time-step to be used in the simulation. This will be used to associate a time-value with the produced velocity fields.
- **t0** (*float*) – The starting time to be used in the simulation. This will be used to associate a time-value with the produced velocity.
- **timePrecision** (*int*) – Number of points after the decimal to keep for the time value.
- **tEnd** (*float*) – The ending time for the simulation.
- **uMeanXPrec** (*ndarray*) – The values of the mean streamwise velocity from the precursor.
- **uMeanXInfl** (*ndarray*) – The values of the mean streamwise velocity for the inflow boundary layer.
- **uMeanYPrec** (*ndarray*) – The values of the mean wall-normal velocity from the precursor.
- **uMeanYInfl** (*ndarray*) – The values of the mean wall-normal velocity for the inflow boundary layer.
- **etaPrec** (*ndarray*) – The values of eta for the corresponding values of the mean velocity from the precursor.
- **yPlusPrec** (*ndarray*) – The values of y+ for the corresponding values of the mean velocity from the precursor.
- **pointsZ** (*ndarray*) – A 2d array containing the values of z for the points of the precursor mesh.
- **etaInfl** (*ndarray*) – The values of eta for the mesh points at the inflow boundary.
- **yPlusInfl** (*ndarray*) – The values of y+ for the mesh points at the inflow boundary.
- **pointsZInfl** (*int*) – A 2d array containing the values of z for the points of the inflow boundary.
- **nInfl** (*int*) – The amount of points in the wall-normal direction that contain the boundary layer at the inflow boundary.

- **gamma** (*float*) – The ration of the friction velocities in the inflow boundary layer and the precursor.
- **times** (*list of floats or strings*) – The times for which the velocity field was sampled in the precursor simulation.
- **blending** (*ndarray*) – The weights for blending the inner and outer profiles.

eddylicious.generators.**interpolation_generate**(*readerFunction*, *writer*, *writePath*, *dt*, *t0*, *tEnd*, *timePrecision*, *points*, *pointsInfl*, *idxPrec*, *times*)

Generate the the inflow velocity interpolation.

This function will take some precursor data and interpolate it on the grid at the inflow patch.

Parameters

- **readerFunction** (*function*) – The function to use for reading in data, generated by the reader. Should contain the reader's name in the attribute "reader".
- **writer** (*str*) – The writer that will be used to save the values of the velocity field.
- **writePath** (*str*) – The path for the writer.
- **dt** (*float*) – The time-step to be used in the simulation. This will be used to associate a time-value with the produced velocity fields.
- **t0** (*float*) – The starting time to be used in the simulation. This will be used to associate a time-value with the produced velocity.
- **timePrecision** (*int*) – Number of points after the decimal to keep for the time value.
- **tEnd** (*float*) – The ending time for the simulation.
- **points** (*ndarray*) – A 2d array containing the values the points of the source geometry or their Delaunay triangulation.
- **pointsInfl** (*ndarray*) – A 2d array containing the values the points of the inlet geometry.
- **idxPrec** (*ndarray*) – Indices for filtering the read-in velocity
- **times** (*list of floats or strings*) – The times for which the velocity field was sampled in the precursor simulation.

CHAPTER 5

List of references

Bibliography

[Jar08] N. Jarrin. *Syntethic Inflow Boundary Conditions for the Numerical Simulation of Turbulence*. PhD thesis, The University of Manchester, 2008.

[KPBK04] A. Keating, U. Piomelli, E. Balaras, and H.-J. Kaltenbach. A priori and a posteriori tests of inflow conditions for large-eddy simulation. *Physics of Fluids*, 16(12):4696, 2004. doi:10.1063/1.1811672.

[LM15] M. Lee and R. D. Moser. Direct numerical simulation of turbulent channel flow up to re_τ 5200. *Journal of Fluid Mechanics*, 774:395–415, 2015.

[LWS98] T. S. Lund, X. Wu, and K. D. Squires. On the Generation of Turbulent Inflow Conditions for Boundary Layer Simulations. *Journal of Computational Physics*, 140:233–258, 1998. doi:10.1006/jcph.1998.5882.

[ML17] T. Mukha and M. Liefvendahl. The generation of turbulent inflow boundary conditions using precursor channel flow simulations. *Computers and Fluids*, 156:21–33, 2017. doi:10.1016/j.compfluid.2017.06.020.

[TBA10] G.R. Tabor and M. H. Baba-Ahmadi. Inlet conditions for large eddy simulation: A review. *Computers & Fluids*, 39(4):553–567, 2010. doi:10.1016/j.compfluid.2009.10.007.

Python Module Index

e

Index