Journal of
open research software

Mukha, T 2018 Turbulucid: A Python Package for Post-Processing of Fluid Flow Simulations. *Journal of Open Research Software,* 6: 23. DOI: https://doi.org/10.5334/jors.213

SOFTWARE METAPAPER

Turbulucid: A Python Package for Post-Processing of Fluid Flow Simulations

Timofey Mukha
Department of Information Technology, Uppsala University, SE
timofey.mukha@it.uu.se

A Python package for post-processing of plane two-dimensional data from computational fluid dynamics simulations is presented. The package, called `turbulucid`, provides means for scripted, reproducible analysis of large simulation campaigns and includes routines for both data extraction and visualization. For the former, the Visualization Toolkit (VTK) is used, allowing for post-processing of simulations performed on unstructured meshes. For visualization, several `matplotlib`-based functions for creating highly customizable, publication-quality plots are provided. To demonstrate `turbulucid's` functionality it is here applied to post-processing a simulation of a flow over a backward-facing step. The implementation and architecture of the package are also discussed, as well as its reuse potential.

Keywords: visualization; computational fluid dynamics; data analysis; post-processing; VTK; maplotlib; Python
Funding Statement: The work was supported by Grant No 621-2012-3721 from the Swedish Research Council.

(1) Overview

Introduction

Performing a computational fluid dynamics (CFD) simulation can generally be divided into three stages: pre-processing, setting up and running the solver, and post-processing. The pre-processing stage consists of defining the computational domain and discretizing it with a mesh. Next, the solver is configured to reflect the physics of the problem and run in order to obtain a solution, which is, in general, a three-dimensional time-dependent dataset. At the post-processing stage, the produced solution is analysed. This includes both extracting relevant data (e.g. pressure or velocity values at certain locations) and visualizing it with different types of plots.

Focusing on the post-processing stage, there is a large variety of tools providing associated functionality. Indeed, most CFD software packages, while concentrating on solvers, usually also provide means for conducting basic post-processing. For example, routines for extracting data along a cut-plane or a line are commonly present, as well as the possibility for producing different types of plots (scatter plots, vector plots, streamline plots etc). Specialized software for post-processing exists as well, and typically provides a richer functionality, better performance, and higher-quality rendering. However, the existing solutions tend to:

- Focus on working with large unstructured three-dimensional data.
- Provide limited options for customizing the plots (e.g. fonts, sizes and styles of different plot elements, etc).
- Excel in interactivity, but not reproducibility (e.g. quickly producing the same plot for 10 different datasets).

Indeed, the properties above reflect the average post-processing needs of the users. Nevertheless, there are many situations when a different sort of tool is required. While the absolute majority of CFD datasets are, in fact, three-dimensional, and many are also time-dependent, it is hard to analyse such data as is. Commonly, time-averaging is applied along with plane- or line-data extraction, and the actual analysis and visualization are thus performed on data of lower dimension. Fine-grain plot customization may not be needed in most applications but is an absolute must when producing figures for publications. Finally, interactivity is important when looking at a single case for the first time, but being able to quickly reproduce the analysis or apply it to a new dataset becomes increasingly important in larger simulation campaigns.

It would, therefore, be beneficial to have a complementary post-processing tool that excels at performing easily reproducible analyses of two-dimensional datasets as well as producing high-quality customizable visualizations. The focus of this work is presenting such a tool, namely, a Python package called `turbulucid`. By combining data manipulation functionality provided by

the Visualization Toolkit (VTK) [7] and plotting routines available in matplotlib [1], turbulucid allows to produce publication-quality plots of several types and provides functionality for analysing the dataset with a set of data extraction routines. Using the package, the post-processing can be scripted in Python using the provided objects and functions. This makes it easy to analyse a large set of simulations in a structured way or quickly apply an existing analysis to a new case. Additionally, users have the possibility to use any of the numerous packages available in the Python ecosystem in their analysis.

While the package works exclusively with planar two-dimensional datasets, no assumption is made regarding the topology of the computational mesh, meaning that any mesh consisting of polygonal cells can be used. The data itself is assumed to be stored in a format for unstructured meshes defined by VTK, but the package is designed to be easily extendible to read data in other formats. Scalar, vector and tensorial quantities are supported.

The discussion of the features of turbulucid, as well as its design and implementation, is continued in the sections below. Examples of applying turbulucid to post-processing of simulations of various flows can be found in [3–6]. This includes flat-plate turbulent boundary layer flow, flow around a ship hull, and also flow around a submarine-like axisymmetric body. Here, the functionality of turbulucid is demonstrated by applying it to post-processing a simulation of a flow over a backward-facing step.

Implementation and architecture

As it was mentioned in the introduction, turbulucid uses Python bindings for VTK to handle the unstructured mesh and matplotlib's plotting routines for producing plots. It is important to note that the user is never exposed to VTK objects, therefore familiarity with VTK's API is not a prerequisite for using turbulucid. All data is instead returned to the user as numpy [8] arrays.

By contrast, the constructed plots are returned to the user as objects of the appropriate type defined by matplotlib (e.g. a StreamPlotSet object for a streamline plot.) This allows for customizing the created plots.

To open a dataset, the user has to create a Case object. The path to the dataset is passed to the constructor. Once created, the methods and attributes of the Case object provide access to the dataset. For example, the _getitem_ operator is overloaded to return the values of a field present in the dataset, given the field's name.

To make turbulucid easily extendible to work with datasets saved in various formats, a separate hierarchy of classes responsible for reading the data is present. Currently, readers for legacy and XML VTK data files (extensions .vtk and .vtu, respectively) are implemented, the corresponding classes being LegacyReader and XMLReader. Both implemented readers are derived from Reader, which serves as a base abstract class. The Case class determines which Reader-class should be used based on the extension of the file the dataset is stored in.

Internally, the dataset is stored as vtkPolyData. Note that this and other VTK formats support data of two types: cell and point. The former associates a value with each polygonal cell whereas the latter associates a value with each mesh-node. In turbulucid, the fields are assumed to be stored as cell data. If point data is stored instead, the readers perform linear interpolation in order to produce corresponding cell data.

To reduce the amount of boilerplate code that has to be written by the user, object-oriented programming is not used for implementing the plotting and data extraction features. Instead, they are implemented as functions, which commonly require a Case object as an input parameter. The provided functions and their purpose are summarized in **Tables 1** and **2**.

Table 1: Plot functions available in turbulucid.

Name	Produced plot
plot_field	Each cell is colored with the corresponding value of a given scalar field.
plot_vectors	Arrows showing the magnitude and direction of a vector field.
plot_streamlines	Streamlines following a vector field.
plot_boundaries	Lines showing the boundaries of the geometry.
add_colorbar	Adds a colorbar.

Table 2: Data extraction functions available in turbulucid.

Name	Purpose
profile_along_line	Extract data along a line.
sample_by_plane	Re-sample the dataset using a Cartesian grid.
dist	Compute distances from centres of boundary-adjacent cells to the boundary.
normals	Compute unit outward normals to every edge of a given boundary.
tangents	Compute unit tangent vectors to every edge of a given boundary.

The package is documented using `numpy`-style docstrings. The `Sphinx` package is used to compile them into `html`.[1]

Demonstration of functionality

In this section `turbulucid` is used to post-process results from a simulation of a flow over a backward-facing step (BFS).[2] The goal is to demonstrate the quality of some of the plot types `turbulucid` can be used to produce. The analysis of the flow as such is therefore kept at a superficial level. For completeness, it is noted that the results were obtained by conducting a large-eddy simulation of the flow using the open-source CFD software `OpenFOAM` [9]. The unknowns were averaged in time in the course of the simulation and then also across the statistically homogeneous spanwise direction, thus producing a two-dimensional dataset.

To show the computational domain, the function `plot_boundaries` can be used. It is possible to scale the *x* and *y* axis. In the case of the BFS, it is common to use the step-height, *h*, as a scaling parameter. We can also use `matplotlib` to add annotations to the figure to indicate what boundary conditions are used, as well as add axes labels, see **Figure 1**. This example clearly illustrates how `turbulucid` seamlessly integrates with `matplotlib` allowing the user to take full advantage of this library.

To get a good qualitative understanding of the flow, let us plot the distribution of the wall-parallel component of velocity across the geometry. To this end, the function `plot_field` can be used, and `add_colorbar` can be used to add a colorbar to the plot, see **Figure 2**.

The code used to produce **Figure 2** is given in **Listing 1**. First, the data is opened by creating a `Case` object. Then the step-height, `h`, is defined, followed by a call to the `plot_field` function that creates the plot. Several arguments are passed to `plot_field`. The created `Case` object is the first argument. The second argument is a `numpy` array of values to be plotted. The array is retrieved from the `Case` object by passing the name of the desired field (here `UMean`, i.e. the mean velocity) to the `_getitem_` operator. Finally, the keyword arguments `scaleX` and `scaleY` are set to `h` to scale the axes of the plot with the step-height. The created plot object is assigned to a variable, `f`. A colorbar object is then created using `add_colorbar` and `f`. The object is then manipulated to set the correct colorbar label. Finally, *x*- and *y*-labels are defined using standard functions from `matplotlib.pyplot`, here imported as `plt`.

```
case = Case("path/to/data")
h = 0.0094318
f = plot_field(case, case["UMean"][:,0],
    scaleX=h, scaleY=h)
cbar = add_colorbar(f)
cbar.ax.set_ylabel(r"$u/U_0$, m/s")
plt.xlabel(r"$x/h$")
plt.ylabel(r"$y/h$")
```

Listing 1: Code snippet used to produce Figure 2.

It can be seen in **Figure 2** that a boundary layer approaches the step from the left, separates, and reattaches at $x/h \approx 6$. A recirculation region is formed directly downstream of the step. To investigate this region further, a vector plot can be created using the `plot_vectors` function, see **Figure 3**. Two recirculation bubbles can be observed: the main, larger, bubble and a secondary one in the corner directly downstream of the step.

Profile plots are commonly used to compare the solution with reference data, obtained computationally or experimentally. For the BFS in particular, profiles of the *x* component of velocity as a function of *y*, at different streamwise locations, can be considered. To extract data along a line the `profile_along_line` function can be used. It is then possible to combine `plot_boundaries` with `matplotlib`'s `plot` function to embed line-plots into the geometry of the computational domain, see **Figure 4**.

The inflection of the velocity profile is clearly seen in the separation region, at $x/h = 2$, whereas at $x/h \approx 6$ the inflection vanishes, indicating reattachment. A recovery of a canonical turbulent boundary layer profile is observed downstream.

Quality control

The framework `pytest` is used to test the implemented functionality with unit tests. Travis CI is used to automatically test installing the package and running all the tests, using both Python 3 and Python 2. This is performed using the Anaconda Python distribution, with the latest provided versions of the required packages. Instructions for running the tests locally are given in the package documentation. The best way to validate the functionality of the software is to apply it to post-processing a simple dataset. To that end, three datasets are shipped with `turbulucid`, including the BFS simulation results used here to illustrate its functionality.

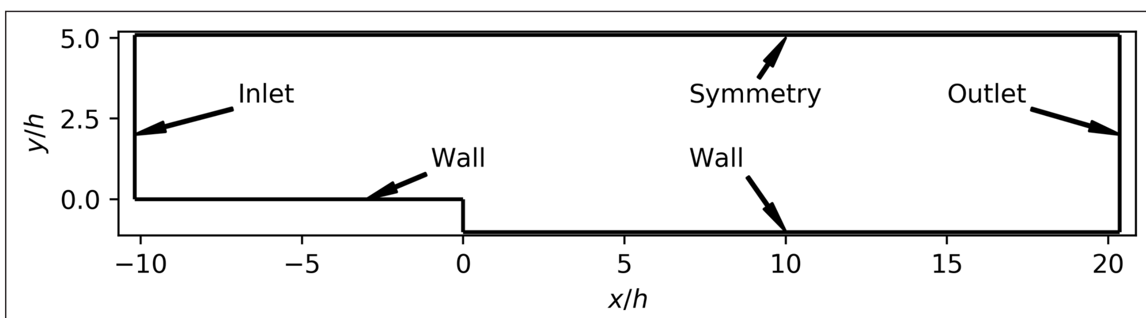

Figure 1: Computational domain of the BFS simulation.

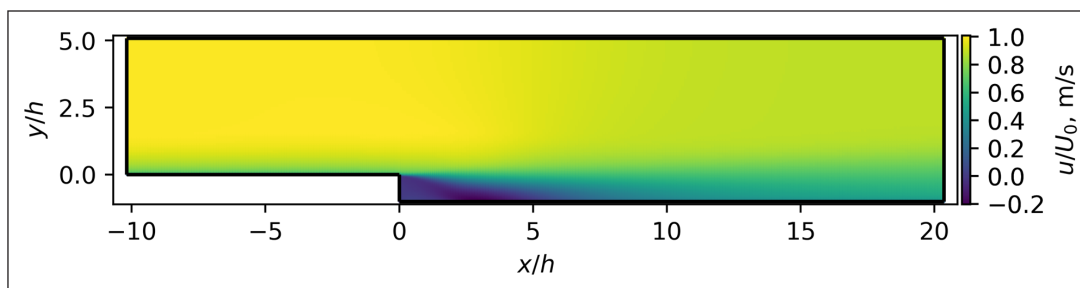

Figure 2: Distribution of the wall-parallel velocity across the domain. Values normalized with a reference velocity, U_0.

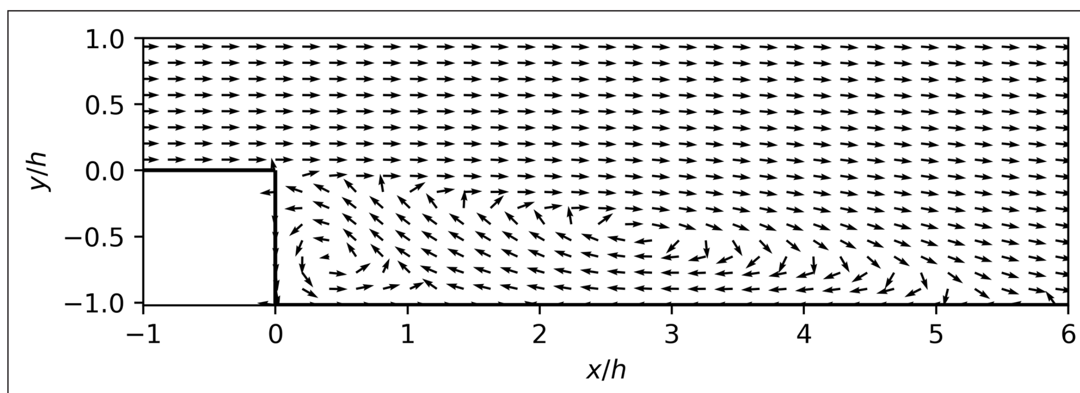

Figure 3: Velocity vectors in and around the recirculation region.

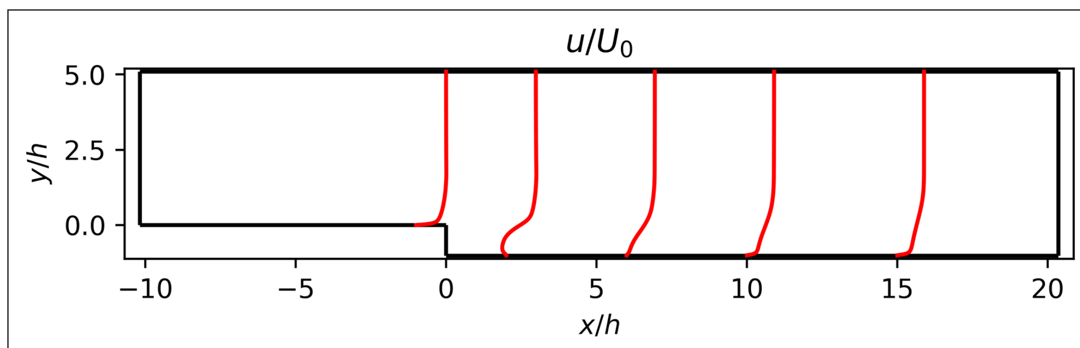

Figure 4: Profiles of wall-parallel velocity at different streamwise locations. Values normalized with a reference velocity, U_0.

(2) Availability

Operating system

The package is expected to run on any operating systems, which are supported by all the dependencies (see below). This includes but is not limited to modern Linux distributions, Windows, and Mac OS.

Programming language

`Turbulucid` is written in Python 3, but is compatible and tested with Python 2 as well.

Additional system requirements

None.

Dependencies

The following Python packages: `numpy`, `matplotlib`, `scipy`, `pytest`. VTK version 7.0.0 or higher, and the associated Python bindings. The `Sphinx` package is needed to build the documentation.

List of contributors

- Timofey Mukha, Uppsala University. Development, testing, writing documentation.
- Saleh Rezaeiravesh, Uppsala University. Validation of functionality.
- Mattias Liefvendahl, Uppsala University and Swedish Defence Research Agency (FOI). Validation of functionality.

Software location

Archive

Name: Github release v0.2
Persistent identifier: https://github.com/timofeymukha/turbulucid/archive/v0.2.zip
Licence: GNU GPL version 3
Publisher: Timofey Mukha
Version published: v0.2
Date published: 16/08/2018

Code repository Github
Name: `turbulucid`
Persistent identifier: https://github.com/timofeymukha/turbulucid
Licence: GNU GPL version 3
Date published: 02/03/2016

Language
English

(3) Reuse potential

`Turbulucid` can be useful to all engineers and researchers working with computational fluid dynamics. In particular, when there is need for producing a publication-quality plot or performing an easily reproducible scripted analysis of a simulation campaign.

The package can be used directly with any CFD solver that supports extracting cut-plane data in VTK format. Otherwise, the data should first be converted into the appropriate format, e.g. using the VTK API. The `turbulucid` package itself can also be extended to read in data stored in a different format and apply appropriate conversion routines on the fly. Such contributions are most welcome, and anyone willing to extend `turbulucid` in this or any other way is encouraged to contact the author or open an issue in the Github repository.

A *readme* file, including installation instructions, is provided with the software. Additionally, a tutorial in form of a Jupyter notebook [2] is provided, demonstrating most of the functionality of the package. While further support cannot be guaranteed, the author will do his best to provide aid to users. Github issues can be used for asking for help.

Notes

[1] See https://timofeymukha.github.io/turbulucid.
[2] The simulation results were provided by Saleh Rezaeiravesh from Uppsala University through personal communication.

Acknowledgements
The incentive to create `turbulucid` came from the need to post-process simulations conducted using computing resources provided by the Swedish National Infrastructure for Computing (SNIC). Therefore SNIC and, in particular, the PDC Center for High Performance Computing (PDC-HPC) are gratefully acknowledged.

Competing Interests
The author has no competing interests to declare.

References

1. **Hunter, J D** 2007 "Matplotlib: A 2D graphics environment." In: *Computing in Science & Engineering*, 9(3): 90–95. DOI: https://doi.org/10.1109/MCSE.2007.55
2. **Kluyver, T,** et al. 2016 "Jupyter Notebooks — a publishing format for reproducible computational workows." In: *ELPUB*, 87–90. DOI: https://doi.org/10.3233/978-1-61499-649-1-87
3. **Liefvendahl, M, Fureby, C** and **Boelens, O J** 2016 "Grid requirements for LES of ship hydrodynamics in model and full scale." In: *31st Symposium on Naval Hydrodynamics*. September. Monterey, California.
4. **Liefvendahl, M, Johansson, M** and **Quas, M** 2017 "Grid generation for wall-modelled LES of ship hydrodynamics in model scale." In: *VII International Conference on Computational Methods in Marine Engineering, MARINE 2017*, 143: 259–268. DOI: https://doi.org/10.1016/j.oceaneng.2017.07.055
5. **Mukha, T, Johansson, M** and **Liefvendahl, M** 2018 "Effect of wall-stress model and mesh-cell topology on the predictive accuracy of LES of turbulent boundary layer flows." In: *7th European Conference on Computational Fluid Dynamics*. Glasgow, UK.
6. **Rezaeiravesh, S, Liefvendahl, M** and **Fureby, C** 2016 "On grid resolution requirements for LES of wall-bounded flows." In: *ECCOMAS Congress 2016*. Crete, Greece.
7. **Schroeder, W, Martin, K** and **Lorensen, B** 2006 *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*. 4th. Kitware.
8. **van Der Walt, S, Colbert, S C** and **Varoquaux, G** 2011 "The NumPy array: A structure for effcient numerical computation." In: *Computing in Science & Engineering*, 13(2): 22–30. DOI: https://doi.org/10.1109/MCSE.2011.37
9. **Weller, H G,** et al. 1998 "A tensorial approach to computational continuum mechanics using object-oriented techniques." In: *Computers in Physics*, 12(6): 620–631. DOI: https://doi.org/10.1063/1.168744

How to cite this article: Mukha, T 2018 Turbulucid: A Python Package for Post-Processing of Fluid Flow Simulations. *Journal of Open Research Software*, 6: 23. DOI: https://doi.org/10.5334/jors.213

Submitted: 12 January 2018 **Accepted:** 12 October 2018 **Published:** 02 November 2018

Copyright: © 2018 The Author(s). This is an open-access article distributed under the terms of the Creative Commons Attribution 4.0 International License (CC-BY 4.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited. See http://creativecommons.org/licenses/by/4.0/.

Journal of Open Research Software is a peer-reviewed open access journal published by Ubiquity Press.

OPEN ACCESS