Aerospace Science and Technology 117 (2021) 106961

Contents lists available at ScienceDirect

Aerospace Science and Technology

www.elsevier.com/locate/aescte

PyVT: A python-based open-source software for visualization and graphic analysis of fluid dynamics datasets

Qing Liu [b], Zheng Qiao [a], Yu Lv [a,c,*]

[a] Department of Aerospace Engineering, Mississippi State University, Mississippi State, MS 39762, USA
[b] Department of Computer Science, Mississippi State University, Mississippi State, MS 39762, USA
[c] The State Key Laboratory of Nonlinear Mechanics, Institute of Mechanics, Chinese Academy of Sciences, Beijing 100190, China

A R T I C L E   I N F O

Article history:
Received 21 February 2021
Received in revised form 26 May 2021
Accepted 9 July 2021
Available online 21 July 2021
Communicated by Qiulin Qu

Dataset link:
https://github.com/lvyu-imech/PyVT.git

Keywords:
Visualization
VTK
PyQt5
Fluid dynamics

A B S T R A C T

We developed a Python-based open-source software for scientific visualization of fluid dynamics datasets. This software is enabled by the VTK graphic rendering library and equipped with an interactive user interface built upon PyQt5. The concept of "scene module" is proposed to construct the rendered scene for graphic analysis. In the context of fluid dynamics, scene module is defined by associating a manifested geometry, such as a cut plane, an iso-surface, a set of streamlines and a vector field, with a set of visual attributes, including rendering scalar, colormap, and opacity. In the proposed framework, each scene module is processed by one single VTK pipeline, and a number of modules can be rendered simultaneously for multivariate graphic analysis. Beyond the VTK built-in functionality, additional implementation efforts are taken to enhance the annotating capability, improve the display of coordinate axes, and load input data of various formats. Two case studies are presented to demonstrate the capability of PyVT, especially the visualization of multivariate fluid dynamic datasets that typically possess complex and interdependent flow physics.

1. Introduction

Visualization is an indispensable means to explore information in scientific data, and it is needed in almost all the research areas. For fluid dynamics research in particular, visualization plays a crucial role in analyzing flow behaviors and extracting fundamental fluid characteristics and principles. For instance, visualization facilitates the identification of vortical structure in the wake of micro-air vehicles [1,2]; visualization analysis helps decode the combustion modality and the key controlling mechanisms in supersonic jet engines [3,4]; visualization analysis helps reveal some abnormal or unexpected flow behaviors in turbomachinery devices [5–7]; furthermore, visualization analysis assists in the discovery of the shock dynamics in different nozzle configurations and the optimization of nozzle geometries to avoid undesired effects [8,9]. Given the numerous applications, it is of great importance to have powerful software tools to efficiently and effectively conduct visualization tasks.

In the scientific community, ParaView and VisIt [10] are two open-source state-of-the-art visualization software that possess powerful capabilities and a large user population. Both ParaView and VisIt employ Visualization Toolkit (VTK) as the data processing and rendering engine, and are equipped with Qt-based intuitive graphic user interface (GUI) to allow users conducting visualization and analysis tasks interactively and efficiently. In addition, both software applications support a variety of computer platforms, including Windows, MacOS, and Linux workstations, and common distributed-memory multicomputers and clusters. To tackle dynamically evolving the scientific datasets, both visualization software provide Python scripting interfaces to allow users to assess the full data-processing and visualization capability under the Python command-line environment. Comparatively speaking, ParaView and VisIt are relatively heavyweight, and VisIt in particular has extensive dependence on third-party packages. The scripting interface introduces another layer of abstraction of VTK API (Application Programming Interface) functions, which can only be utilized in conjunction with the installed specific libraries. Besides the considable success of ParaView and VisIt, there is some latest and notable progress in the development of visualization software and tools. In particular, the popularization of scripting interface gives rise to a few purely VTK-based Python applications in the scientific visualization landscape, of which PyVista [11] and vedo [12] are two notable representatives. Both PyVista and vedo provide direct and streamlined Python API interfaces to the VTK library.

* Corresponding author.
E-mail addresses: ylv@ae.msstate.edu, lvyu@imech.ac.cn (Y. Lv).

https://doi.org/10.1016/j.ast.2021.106961
1270-9638/© 2021 Elsevier Masson SAS. All rights reserved.

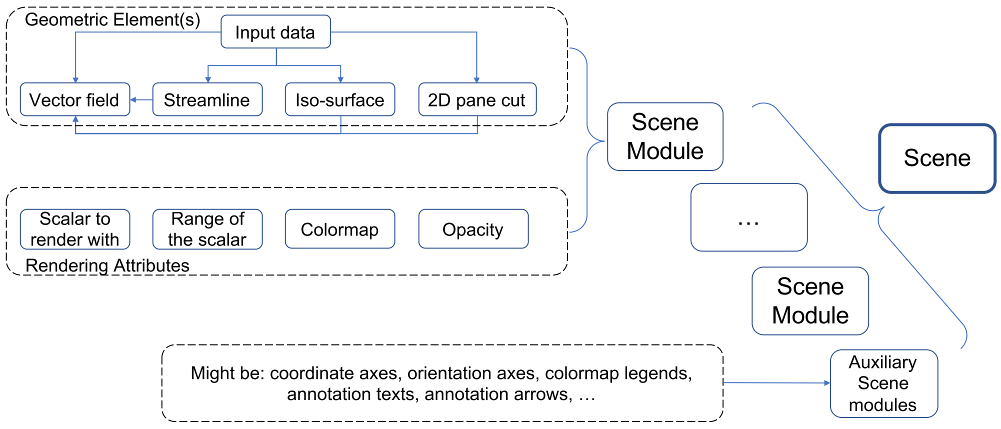


Fig. 1. Illustration of the conceptual design of PyVT algorithm.

Although PyVista and vedo are similar to the python scripting interfaces of ParaView and VisIt in terms of functionality, the PyVista and vedo are much lightweight and exposes VTK functions to users in a more straightforward manner. At the present stage, the major limitation of PyVista and vedo is that they lack GUI support and require users to be sufficiently familiar with their API functions for sophisticated visualization tasks.

The objective of our work is to develop a new visualization tool that overcomes some of the limitations of aforementioned visualization applications. The software we developed is called PyVT (Python-based Visualization Tool), for which VTK is used as the rendering engine and PyQt is employed to provide the interactive GUI. To satisfy the needs of modern visualization requirement, PyVT is able to run in either GUI mode or script mode. Moreover, PyVT features: *i*) a module-based, fine-tuned and highly efficient workflow to visualize and analyze multivariate dataset; *ii*) an easy-to-understand, purely pythonic wrapper of the VTK filters and mappers to utilize VTK visualization pipeline; and *iii*) an intuitive and highly interactive GUI to allow CFD users to define, preview and intermix geometric representations of CFD data. Since PyVT is built purely upon a modern and dynamic programming platform—Python3, thereby it possesses a concise code structure and the high-level language characteristics, and also preserves excellent portability, maintainability and extendibility.

This paper details our software development efforts and demonstrates the capability of PyVT software. The remainder of the paper is organized as follows. Section 2 focuses on the conceptual design of the algorithm, especially a module-based strategy for composing a multicomponent scene. Followed by that, Section 3 provides the implementation details, including the VTK wrapper, the flexibly annotating capability, the improve display of coordinate axes, and the input/output (IO) interface. Section 4 introduces the GUI. Moreover, case studies are carried out to demonstrate the capability of PyVT in Sec. 5. The paper finishes with concluding remarks in Sec. 6.

2. Conceptual design

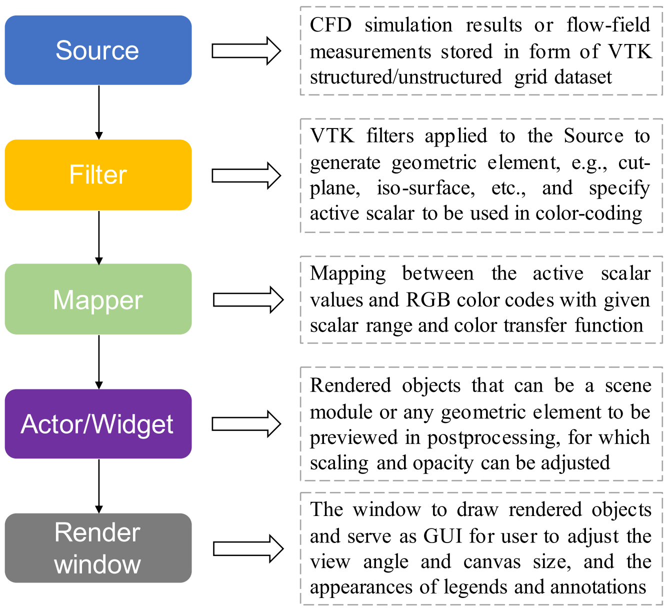
Data visualization is a process of transforming the information stored in scientific dataset to graphic manifestation. To perform this task, in particular for fluid dynamics dataset, it requires the algorithms to be capable of exhibiting multivariate information, and a variety of geometric representations. To elaborate the conceptual design of PyVT algorithm, we first introduce three important definitions:

- Geometric element. A geometric element is a graphic representation of a convenient and meaningful subset of a given dataset, which can be a cut plane, an iso-surface, a set of streamlines, a vector field, or a whole volumetric field. In order to allow the user to prescribe the geometric elements, a dialog process is designated to collect necessary input parameters from the user.
- Scene module. A scene module is defined by associating a set of geometric element (not necessary the same type) with the rendering attributes, such as the scalar field to be used in rendering, color transfer function, and opacity. Scene module is a basic graphic unit in the final scene which is the result of the visualization task.
- Scene. A scene is a rendered view produced for a specific visualization task. A scene might consist of multiple scene modules, including the auxiliary ones such as coordinate axes and annotations.

The conceptual design of our module-based visualization algorithm is illustrated schematically in Fig. 1. The essential unit in the scene creation is the so-called scene module, which is defined by associating a set of rendering attributes with the included geometrical elements. In the context of PyVT, geometric element is considered as the geometric representation of the entire or certain convenient subset of the three-dimensional dataset. For CFD applications, the input dataset is typically the numerical solution on a structured or unstructured grid, and the subset of the input data for visualization purpose can be a two-dimensional plane cut, an iso-surface of certain scalar field, a bundle of streamlines, or a vector field that can be defined from the input data or the aforementioned subset(s) of the input data. In each scene module, after geometric elements are defined, the user can specify a set of attributes, including the scalar to render with, the range of the scalar, the style of colormap, and the opacity, so as to create a rendered module. It is worth mentioning that each scene module is able to include multiple geometric elements, as long as the geometric elements share the same rendering attributes, and the rendering of each module is handled by a single visualization pipeline in the backend. This unique algorithmic design allows us to minimize the use of computer resource, which preserves the optimal efficiency during the human-computer interaction. Overall, a rendered scene module is deemed as a building block of the final scene. To efficiently and effectively manifest the physical information contained in the input dataset, the final scene can include a number of rendered scene modules. In addition, for the clarity in graphic representation and analysis, the final scene, might also contain a set of auxiliary modules that are responsible for displaying coordinate axes, orientation axes, colormap legends, annotation texts, annotation arrows, and so on.

Table 1
Geometric elements, their required input parameters, and the corresponding VTK Filter functions.

Geometric element	Input parameters	VTK filter(s)	Observations
original 3D data	None	`vtkPassThroughFilter`	Only the boundary surfaces rendered by the VTK convention
2D cut-plane(s)	plane origin, plane normal, number of cuts, and signed offset	`vtkCutter`	number of cuts and offset required for defining more than one cut-planes
iso-surface(s)	Specific scalar variable, rescaled range of scalar magnitude, number of iso-values, or a single value	`vtkContourFilter`	Input a single value define a single iso-surface; otherwise, specify number of iso-values for multiple iso-surfaces
streamlines	Source type: (1) Point: coordinate, radius, No. of seeds; (2) Line: coordinates of point A and point B and No. of seeds along the AB line; or (3) Plane: coordinates of origin (point O), points A and B, and No. of seeds along OA and OB dimensions. Tracer integration direction, matching distance, and step size	`vtkStreamTracer`	Only one type of source specified at a time; integration direction can be forward, backward or both.
vector field	parameters of associated geometric element(s), vector variable name, skip ratio and scaling factor	`vtkGlyph3D` `vtkMaskPoints`	defined based on the original dataset, 2D cut-plane(s), iso-surface(s) or streamlines



CFD simulation results or flow-field measurements stored in form of VTK structured/unstructured grid dataset

VTK filters applied to the Source to generate geometric element, e.g., cut-plane, iso-surface, etc., and specify active scalar to be used in color-coding

Mapping between the active scalar values and RGB color codes with given scalar range and color transfer function

Rendered objects that can be a scene module or any geometric element to be previewed in postprocessing, for which scaling and opacity can be adjusted

The window to draw rendered objects and serve as GUI for user to adjust the view angle and canvas size, and the appearances of legends and annotations

Fig. 2. VTK pipeline and description of each component.

```
# scene module class
class Scene_Module():

    scalar_name = ''
    scalar_range = None
    relative_range = None
    opacity = 1
    colormap = ''
    cutplane = None
    cutplane_param = None
    isosurf = None
    isosurf_param = None
    streamline = None
    streamline_param = None
    vectorfield = None
    vectorfield_param = None
    geomcombine = None
    mapper     = None
    actor      = None

    def __init__(self):
        pass

    def write(self, fp):
        ......

    def render(self, input_data):
        ......

    def load(self, fp):
        ......
```

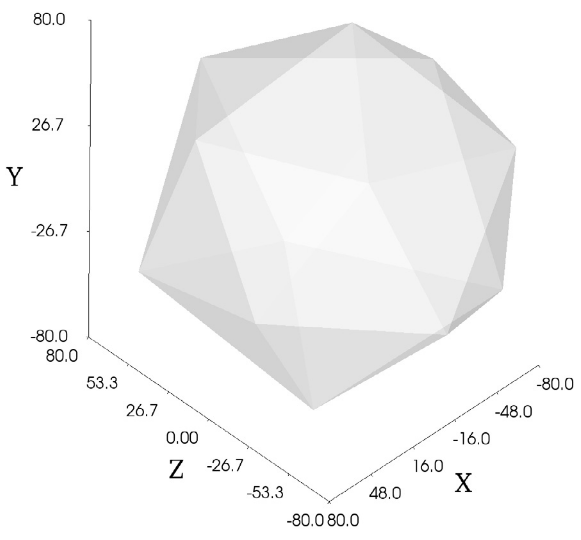
Listing 1: Python pseudo-code of the Scene Module Class.

3. Algorithmic implementations

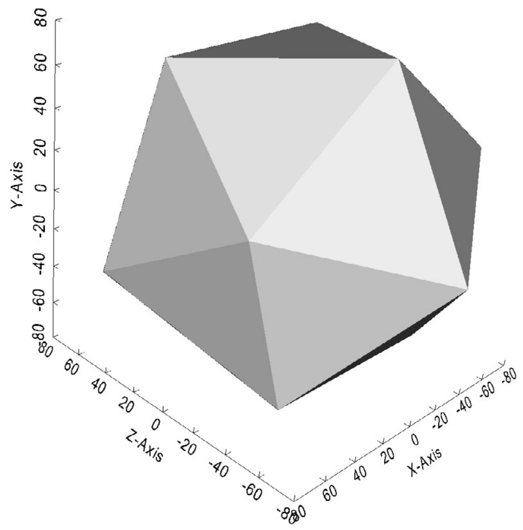
In this section, we will present the implementation details of PyVT, explaining how a rendered scene module is algorithmically generated using the VTK pipeline. In addition, we also take effort to improve the display of coordinate axes, and enable a flexible annotating capability. The programming details will also be given.

3.1. Implementation of scene module

Our implementation is based on Python3 and VTK library of the latest release. To help elaborate the details, the visualization pipeline of VTK is first revisited. As shown in Fig. 2, the VTK Filter plays a central role in the pipeline as it is used to obtain geometric elements of various types. Each type of geometric element requires a specific VTK Filter. In Table 1, we listed all geometric-element types used in PyVT, their required input parameters, and the corresponding VTK Filter functions. The defined geometric elements can be rendered with various attributes to manifest physical information. One of key attributes is the so-called active scalar–a selected physical quantity in the dataset. For CFD applications, the active scalar can be density, pressure, temperature, velocity magnitude, etc. Once the active scalar is chosen, its quantitative information is attached onto the associated geometric element and manifested through colors. The color-coding is accomplished through the VTK Mapper, with which the style of colormap and the mapping relation between scalar value and digitized color code are specified. Besides, the opacity property is another attributed. All rendering attributes are specified by the user. Once the geometric element and all the corresponding attributes are prescribed, a scene module is defined, and can then be rendered by the VTK Actor and revealed in the Render window. It is noted that multiple VTK Actor can be defined in the same Render window, allowing various overlapping modules to be shown simultaneously. As for the algorithmic implementation, we provide a pseudocode of the python class definition of one scene module in Listing 1. It can be seen that the scene module class includes a number of class variables to store the geometric elements and attributes, and three major



(a) Improved manifestation



(b) Built-in manifestation

Fig. 3. Illustration of rendering-quality improvement on coordination axes manifestation with extended implementation (note that all quantities are non-dimensionalized).

class methods: *i*) write–output all module information into a journal file; *ii*) load–input all module information from a journal file; and *iii*) render–execute the VTK visualization pipeline.

3.2. Improvement of coordinate axes manifestation

The *x*-, *y*- and *z*-coordinate axes are an important and necessary component in most fluid dynamic scenes, illustrating the dimensions of a rendered geometric element. In the default VTK implementation, the `vtkCubeAxesActor` class is designated to generate the coordinate axes with a large number of user customization options. However, the title and label texts of `vtkCubeAxesActor` are rendered in the 3D mode, which does not lead to a high-quality manifestation. The application of `vtkCubeAxesActor2D` class may, to some extent, mitigates this issue with the improved antialiasing performance, but does not allow users to fully customize the axes. One notable limitation is that the number of labels in the `vtkCubeAxesActor2D` must set to be the same for all the three different axes.

To address this dilemma and meet the needs of high-quality coordinate axes manifestation, we design and program a new class for coordinate axes, of which the pseudocode is presented in Listing 2. The new axes class employs one `vtkCubeAxesActor2D` instance (although set to invisible) as a baseline entity to retrieve the positions of three axes in the 2D display coordinate since the positions vary with respect to the camera setting and object location on the canvas. With the positions three `vtkAxisActor2D` instances are used to represent the *x*-, *y*- and *z*-coordinates, respectively. This strategy resolves the issues associated with the rendering of label and title texts, and also provides the user as many customization options as possible. As shown in Pseudocode 2, the title name, range, number of labels, font size, label offset, label scaling, and visibility of each individual coordinate axis can be specified by the user. The improved manifestation of coordinate axes is illustrated in Fig. 3.

Although new implementation requires more rendering resources since a few actors are included in one coordinate-axes instance, it is worthwhile to pay the price as our goal is to provide a visualization tool that can generate high-quality figures/images for scientific publications.

```
# scene module class for coordinate axes
class Scene_Module_FrameAxes():

    title_x = 'X'
    title_y = 'Y'
    title_z = 'Z'

    show_x  = True
    show_y  = True
    show_z  = True

    bounds_x = None
    bounds_y = None
    bounds_z = None

    noflabel_x = 4
    noflabel_y = 4
    noflabel_z = 4

    grid_on      = False
    font_factor  = 1.5
    label_factor = 0.6
    offset       = 20

    actor   = vtk.vtkCubeAxesActor()
    actor2D = vtk.vtkCubeAxesActor2D()

    actor2D_x = vtk.vtkAxisActor2D()
    actor2D_y = vtk.vtkAxisActor2D()
    actor2D_z = vtk.vtkAxisActor2D()

    exponent_x = 0
    exponent_y = 0
    exponent_z = 0

    def __init__(self):
        pass

    def write(self, fp):
        ......

    def render(self, input_data):
        ......

    def load(self, fp):
        ......
```

Listing 2: Pseudocode of the scene module of coordinate axes.

3.3. Interactive annotating capability

Annotations are often required in a fluid dynamics scene in order to provide explanatory information. To enable annotating capability, three different annotation types are enabled in the present version of PyVT: (*i*) text; (*ii*) arrow; and (*iii*) colormap

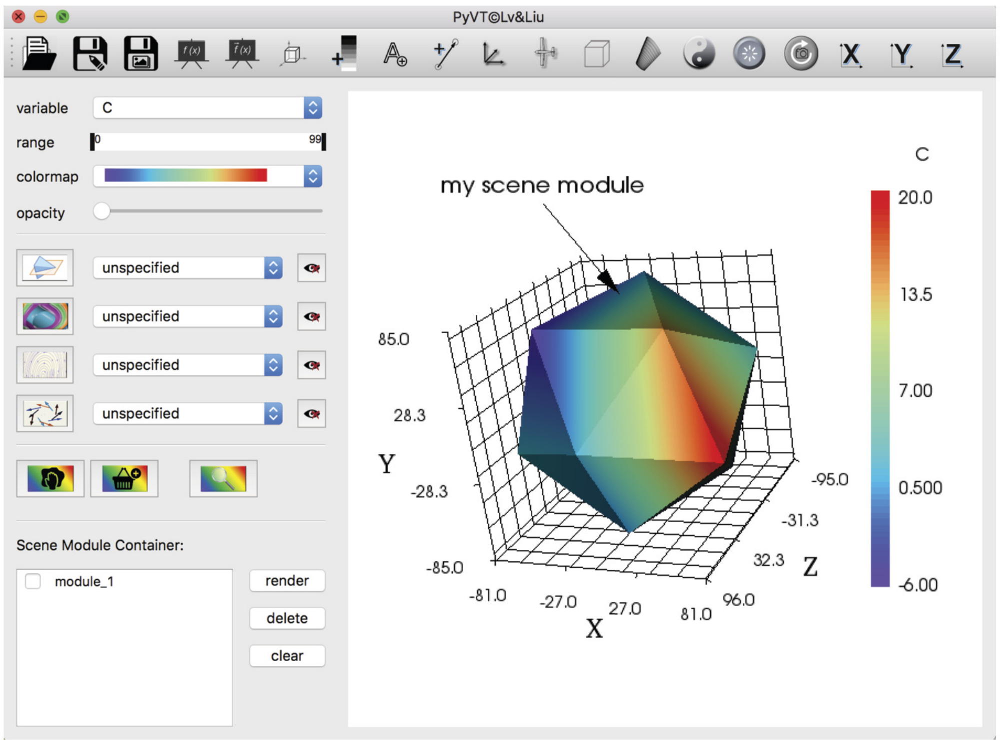


Fig. 4. Snapshot of PyVT graphic user interface (note that all quantities are non-dimensionalized).

legend, which are implemented as instances of the `vtkTextActor`, `vtkLeaderActor2D`, `vtkScalarBarActor` classes, respectively. The main algorithm challenge is to enable convenient user specification of these annotations. We propose a novel two-step approach to achieve this objective. In the first step, the annotation editing mode is activated, and the user is allowed to adjust the size and position of annotation on the canvas. To achieve this functionality algorithmically, the text and colormap legend instances are encapsulated into `vtkTextWidget` and `vtkScalarBarWidget` instances, respectively, which are rendered onto the canvas, so that the user can move and adjust the annotations by selecting and dragging via the mouse. Meanwhile, the arrow instance can be defined directly by prescribing two points on the canvas via the mouse. In the second step, PyVT exits the annotation editing mode once the appearances of annotations are approved by the user and unable to be modified. In the algorithmic level, the widget instances for text and legend annotations turn disabled and the corresponding actors are shallow-copied and left on the canvas. Push buttons are added in the GUI to control the on/off status of the annotation editing mode. This two-step approach allows the user to interactively and efficiently define multiple annotations of various types while not leaving excessive amount of draggable and movable but overlapped entities on the canvas, so as to ensure the quality of the user's interaction with VTK render window.

3.4. IO interface

In the present release of PyVT, a number of data loaders are available to load datasets of several commonly used file formats, and meanwhile VTK built-in image writers are employed to export images of the created scene in a variety of file format. The support input and output file formats are summarized in Table 2. Tecplot data loader is developed based on the Tecplot data format [13], and CGNS (CFD General Notation System) data loading is enabled by the export tool of CGNS library [14]. Except the Tecplot and CGNS data loaders, other data formats are handled using the VTK built-in data loaders. It is worth noting that there are some other VTK built-in data loaders to handle data format of popular CFD software, such as ANSYS FLUENT and OpenFOAM. However, as the software further develops and the version evolves, those data loader might be up to date. If needed, the user might simply extend our implementation with those data loaders and use them cautiously. For the output, the size of the scene canvas can be enlarged by rescaling for better visual quality. Furthermore, for checkpointing and recovery, all the parameters of a scene can be stored into a journal file (.jou), which can be used to restore the scene after reopening in the GUI mode or directly generate the scene image in the script mode.

Table 2
Input and output file formats supported by PyVT.

Input data file	Output image file
VTK legacy format (.vtk)	Portable Network Graphics format (.png)
XML unstructured grid format (.vtu)	Encapsulated PostScript format (.eps)
XML structured grid format (.vts)	Joint Photographic Group format (.jpg)
stereolithography format (.stl)	Tagged Image File format (.tiff)
Plot3D Meta file format (.p3d)	Bitmap Image format (.bmp)
Tecplot ASCII data format (.dat)	
Tecplot binary data format (.plt)	
CGNS data format (.cgns)	

4. Graphic user interface

The GUI is designed under the principle of maximizing the clarity and simplicity and minimizing users' interaction burdens. It is implemented based on PyQt5 [15], one of the state-of-the-art and most stable GUI-development frameworks. Fig. 4 presents a snapshot of the graphic interface of PyVT, illustrating the three major regions of the GUI, namely the top toolbar, the left sidebar, and the right VTK render window. In the following we will provide their implementation details individually.

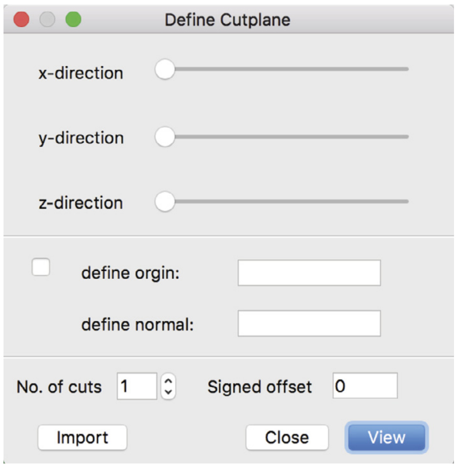
The top toolbar consists of a list of push buttons implemented using the QAction class, and the associated actions are to provide necessary capabilities, such as input-output (IO) operations, data manipulations, prescriptions of frame axes, colormap legends, text and arrow annotations, additions of orientation axes, feature outline, bounding box, and grid mesh, and some view adjustment functionalities. The colormap legend, arrow, and text are considered as annotation entities in PyVT, and multiple and diverse entities of this type can coexist to meet the needs of sophisticated visualization tasks. Their inputs are collected through a two-step process as detailed in Sec. 3.3, for which their corresponding buttons serve as controllers to activate and deactivate the editing mode.

In the left sidebar, the first ribbon from the top is used to specify the attributes of scene modules, including, respectively, a drop-down list (QComboBox) for choosing scalar variables to render geometric element, a double-handle ranger slider (the QRangeSlider adapted from [16]): to adjust the range of scalar magnitude, a drop-down list (QComboBox) for selecting colormap, and a single-handle slider (QSlider) to specify the value of opacity. Below that is the ribbon where the user can specify the geometric elements. For each type of geometric element, a pop-up dialog (QWidget) is activated with the left push-button (QToolButton); once proper parameters are input by the user, the geometric element can then be previewed and imported into the corresponding drop-down list (QComboBox) in the main window for further processing. Below the ribbon of defining geometric elements are three push buttons (QToolButton) that allow the user to preview the defined scene module, add the defined module into the container, and reset view and GUI options for module definition. The module is defined individually with the aforementioned GUI items, and all defined modules are shown in the container (QListWidget) located at the bottom part of the left-split of the main window. There is no limit on the number of modules in the final scene as long as they can be rendered properly by VTK visualization engine. Three addition operations through push buttons (QToolButton) allow the user to render, delete and conceal module(s) selectively, which offers extra forgiveness for the user. The right window is the window to show rendered VTK entities, which is implemented with the QVTKRenderWindowInteractor of VTK library. Examples of PyVT dialogs are demonstrated in Fig. 5.

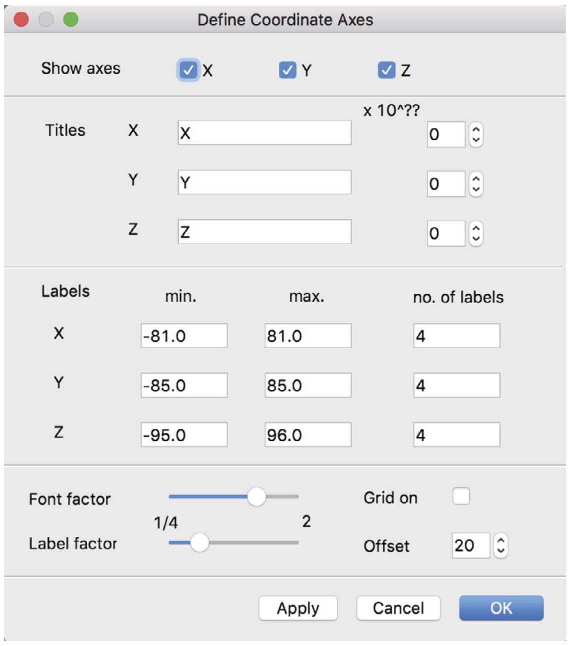
In comparison to the ParaView GUI, the main advantage of PyVT GUI is that it does not explicitly expose the concept of VTK Filters to the user, which reduces some cognitive burdens. The user is able to work directly with the PyVT geometric elements (cut-plane, iso-surface, streamline and vector field), which are conceptually more familiar to the fluid dynamists.

5. Case study

First of all, for the simple visualization tasks (e.g., that shown in Fig. 4), one module is sufficient, and the module container functionality might not be employed in PyVT. Fluid dynamic datasets are typically multivariate and contain complex physical information. PyVT's module-based visualization approach is particularly suited for visualizing and analyzing the datasets of this kind. To illustrate the capability of PyVT, we will consider two examples in this case study, one related to aerodynamic application and the other one related to jet propulsion application. It is worth noting that the focus of this study is placed on demonstrating the capability of PyVT, and therefore the specific visualizations are not meant to be exclusive.



(a)



(b)

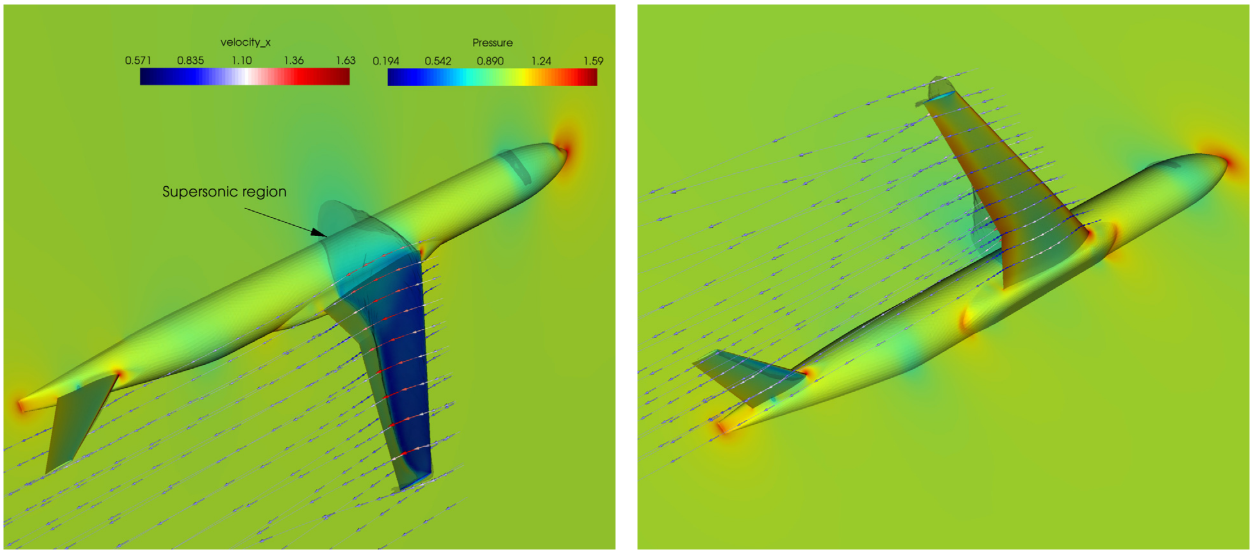
Fig. 5. Design of pop-up dialogs for prescribing cut-planes (a) and coordinate axes (b) in PyVT.

5.1. Graphic analysis of CFD results of a NASA common-research aircraft model

PyVT is used to perform graphic analysis of CFD results of a NASA common-research aircraft model. In this case, the aerodynamic calculation was performed using the SU2 software [17] and the results were stored as a 3D volumetric dataset in a VTK file. This particular analysis includes four major scene modules defined in Table 3, which best exemplifies the capability of PyVT. As shown in Fig. 6, the pressure field on the aircraft surface is visualized through the filled contour plot (Module 1), in which we can identify regions experiencing high structural stress. The flow field around the aero-wing is illustrated using the streamlines (Module 3) and the associated vector field (Module 4). As visualized by these scene modules, the flight condition operates with a minimal angle of attack so that the flow is nearly parallel to the aircraft

Table 3
Specification of major scene modules for Case 1.

Module No.	Geometric element	Parameters	Scalar variable for color-coding
1	domain boundary	none	pressure
2	iso-surface	variable: Mach number number of iso-value: 1 value: 1	Mach number
3	streamlines	seed type: plane origin: (900 100 150) point A: (1700 1200 150) point B: (900 100 270) direction: forward max. distance: 4000 initial step size: 0.1	streamwise velocity
4	vector field (linked to Module 3)	vector variable: velocity skip factor: 20 scaling factor: 40	streamwise velocity



(a) top-side view (b) bottom-side view

Fig. 6. Visualization of CFD results of a NASA common-research model with PyVT. Contour surface and streamline are color-coded by pressure and streamwise velocity; and the translucent surface corresponds to iso-surface of unity Mach number; note that all quantities are non-dimensionalized.

body; furthermore, flow accelerates on the upper surface of the aero-wing leading to the presence of a supersonic region which is illustrated via translucent iso-surfaces (Module 2). Disturbance on the flow, which is introduced by the wing tip, is also manifested at the same time.

In this example, pressure and velocity information are displayed simultaneously with different scene modules. The use of different colormaps makes two kinds of flow information distinguished visually, and the corresponding legends can be placed and arranged in an orderly and tidy manner with PyVT's flexibly annotating capability. The iso-surface of unity Mach number is laid on top of the pressure contour, and made translucent by adjusting the opacity of scene module. In addition, the scene is complemented with the annotations of arrow and text to provide necessary interpretative information.

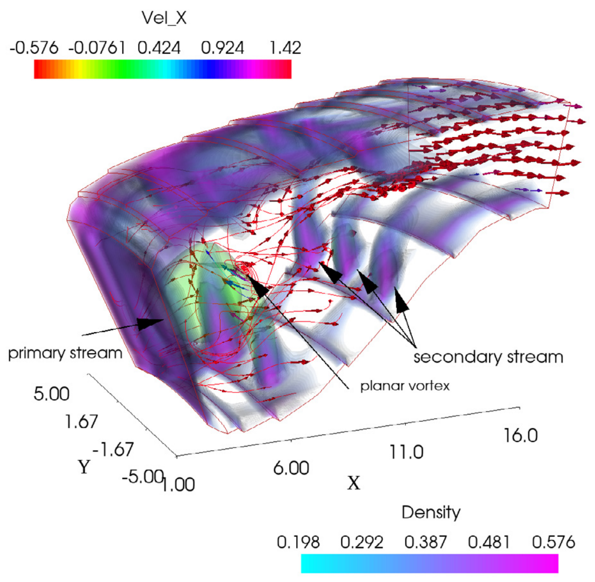
5.2. Graphic analysis of simulation results of a jet-engine combustor

In this case study, the graphic analysis is performed by using PyVT to gain insights to the combustion field in a model gas-turbine combustor. The dataset of this case is taken from a VTK tutorial and stored in a plot3D (.p3d) file. To render the 3D data, several major scene modules are defined in Table 4 to illustrate the complex flow and thermal fields. Since vector can show both direction and magnitude information, the vector field is employed as primary means of visualizing the flow field. To this end, a cut plane at the combustor exit (Module 1) and a set of streamlines initiated inside the combustor (Module 3) are specified at first, and then the arrows are attached to them to display the vector-represented flow fields (Module 2 & 4). As shown in Fig. 7, the rotating flow patten near the primary burner injector is clearly manifested; moreover, the rotation leads to the presence of a recirculation zone (reversed flow) which is illustrated through the translucent iso-surfaces of zero streamwise velocity (Module 5).

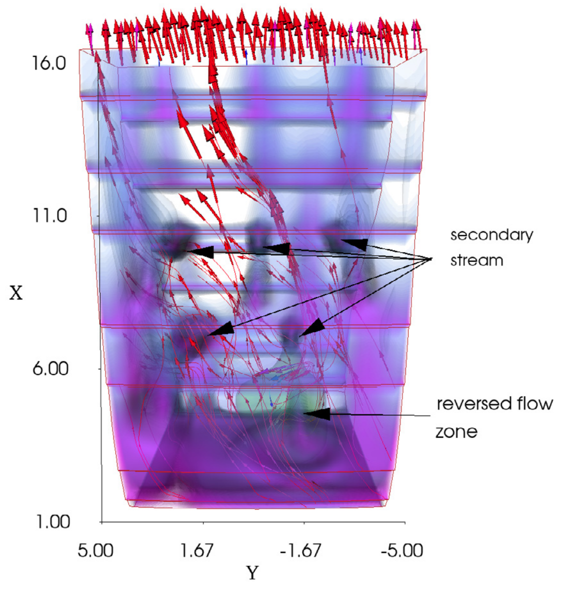
In Fig. 7(b), the top view of the scene manifests important information about the overall flow direction. It can be seen from the streamlines that the overall flow field shows a left-turning pattern, and experiences a significant acceleration along the streamwise direction due to the geometric contraction, which can be clearly observed from the variation in the length of arrows. It can also be seen that the left-turning behavior is diminishing along the

Table 4
Specification of scene modules for Case 2.

Module No.	Geometric element	Parameters	Scalar variable for color-coding
1	cutplane (invisible)	origin (16, 0, 0) normal (1, 0, 0)	none
2	vector field (linked to Module 1)	vector variable: velocity skip factor: 10 scaling factor: 0.1	streamwise velocity
3	streamlines	seed type: plane origin: (5 -5 22) point A: (5 5 22) point B: (5 -5 32) direction: both max. distance: 400 initial step size: 0.1	streamwise velocity
4	vector field (linked to Module 3)	vector variable: velocity skip factor: 100 scaling factor: 0.1	streamwise velocity
5	iso-surface	variable: streamwise velocity number of iso-value: 1 value: -0.1	streamwise velocity
6	iso-surface	variable: density number of iso-value: 40 value: [0.336, 0.71]	density



(a) 3D view



(b) Top view

Fig. 7. Visualization of CFD results of a model jet-engine combustor with PyVT. Volume rendering is provided using density; the single translucent iso-surface corresponds to the zero streamwise velocity; and the vector-field is color-coded by streamwise-velocity. The burner is equipped with a single primary stream inlet and five pairs of secondary stream holes (ten in total); as labeled, two pairs of secondary stream are issued upstream near the primary inlet and the other three pairs are issued at downstream. Note that all quantities are non-dimensionalized.

streamline and the flow at the combustor exit becomes almost streamwise and rather homogeneous.

As for the thermal field, the density field is shown using pseudo volume rendering technique based on translucent iso-surfaces. In this case, a number of iso-surfaces are rendered simultaneously with a small opacity and a distinct colormap so that the density information is manifested in a volumetric sense. From Fig. 7, we can see that the combustor chamber is occupied with burnt low-density hot gas, cold gas is issued through the secondary air injectors and the penetration length of the secondary air streams is about two diameters of the injector hole. The rendered scene is also complemented with the text/arrow annotations to highlight the injector positions, important physical phenomena, and the coordinates axes that illustrate the size of the combustor.

6. Concluding remarks

A new open-source visualization software–PyVT is developed for the visualization and graphic analysis of fluid dynamics dataset. In the conceptual design, the concept of scene module is proposed as the building block to construct the scene which is the result of visualization. Detailed python algorithms are provided to enable rendering of scene module by making use of the VTK visualization pipeline. Special efforts are taken to implement the annotating capability, improve the display quality of coordinate axes, and provide input-output interface for data formats of various types.

A intuitive GUI is developed based on PyQt5 to allow the user to create scene modules in an efficient and interactive manner. The GUI offers the user the full control over the selection of physical variables, colormap style, scalar display range, and opacity. Moreover, a dialog-based inquiry approach is employed to collect parameters from the user so that graphic elements, such as cut plane, iso-surface, streamline, and vector field, can be defined efficiently.

VTK and PyQt5 are two major building blocks of PyVT. Apart from those, PyVT does not have other significant dependences, which makes PyVT lightweight and very portable. PyVT is efficiently implemented in Python3; with the Python3 toolchain, all functional components are efficiently integrated and the capabilities can be easily extended with the future development efforts. Our case-study shows that PyVT is able to serve as a general visualization tool for the analysis of fluid dynamic datasets. The capability of PyVT in displaying multivariate information is well demonstrated through case studies.

PyVT of the current release has not been able to exploit the parallel and decomposed visualization strategy, and handle multi-block datasets (currently it relies on the user to merge the multi-block data to a single-block one). In the next release, we will implement the parallel rendering capability so that PyVT can more efficiently visualize large datasets on multi-threaded computers. Meanwhile, we will also extend the PyVT to load multi-block datasets and allow the user to display specific block(s) selectively. In addition, the supplementary capability to visualize certain special data, such as multiphase simulation data, will be enabled in PyVT in the near future.

Finally, we would like to mention that although the main purpose of developing PyVT is to enrich the landscape of scientific visualization, the PyVT software may also aid in delivering basic knowledges about VTK and PyQt for instructional or educational purposes.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Software Availability

PyVT is available at https://github.com/lvyu-imech/PyVT.git. The data of the case study is also included.

References

[1] S. Deng, B. van Oudheusden, Wake structure visualization of a flapping-wing micro-air-vehicle in forward flight, Aerosp. Sci. Technol. 50 (2016) 204–211, https://doi.org/10.1016/j.ast.2016.01.003.
[2] S. Deng, J. Wang, H. Liu, Experimental study of a bio-inspired flapping wing MAV by means of force and PIV measurements, Aerosp. Sci. Technol. 94 (2019) 105382, https://doi.org/10.1016/j.ast.2019.105382.
[3] Z. Cai, J. Zhu, M. Sun, Z. Wang, Effect of cavity fueling schemes on the laser-induced plasma ignition process in a scramjet combustor, Aerosp. Sci. Technol. 78 (2018) 197–204, https://doi.org/10.1016/j.ast.2018.04.016.
[4] W. Ao, Z. Chen, P. Liu, S. Shang, K. Ma, B. Fu, Mixing enhancement in a subsonic-supersonic shear layer with a cavity splitter plate, Aerosp. Sci. Technol. 102 (2020) 105847, https://doi.org/10.1016/j.ast.2020.105847.
[5] B. Zhao, M. Qi, H. Sun, X. Shi, C. Ma, A comprehensive analysis on the structure of groove-induced shock waves in a linear turbine, Aerosp. Sci. Technol. 87 (2019) 331–339, https://doi.org/10.1016/j.ast.2019.02.036.
[6] Y. Wu, G. An, B. Wang, Numerical investigation into the underlying mechanism connecting the vortex breakdown to the flow unsteadiness in a transonic compressor rotor, Aerosp. Sci. Technol. 86 (2019) 106–118, https://doi.org/10.1016/j.ast.2018.12.040.
[7] R. Li, L. Gao, C. Ma, S. Lin, L. Zhao, Corner separation dynamics in a high-speed compressor cascade based on detached-eddy simulation, Aerosp. Sci. Technol. 99 (2020) 105730, https://doi.org/10.1016/j.ast.2020.105730.
[8] T. Mizukaki, S. Watabe, Visualization of stagnation point inside the closed wake of a 20%-truncated plug nozzle at starting process, Aerosp. Sci. Technol. 50 (2016) 25–30, https://doi.org/10.1016/j.ast.2015.12.013.
[9] P.P. Nair, A. Suryan, H.D. Kim, Computational study on reducing flow asymmetry in over-expanded planar nozzle by incorporating double divergence, Aerosp. Sci. Technol. 100 (2020) 105790, https://doi.org/10.1016/j.ast.2020.105790.
[10] H. Childs, E. Brugger, B. Whitlock, J. Meredith, S. Ahern, D. Pugmire, K. Biagas, M. Miller, C. Harrison, G.H. Weber, H. Krishnan, T. Fogal, A. Sanderson, C. Garth, E.W. Bethel, D. Camp, O. Rübel, M. Durant, J.M. Favre, P. Navrátil, VisIt: an end-user tool for visualizing and analyzing very large data, in: High Performance Visualization–Enabling Extreme-Scale Scientific Insight, 2012, pp. 357–372.
[11] C. Sullivan, A. Kaszynski, PyVista: 3D plotting and mesh analysis through a streamlined interface for the visualization toolkit (VTK), J. Open Sour. Softw. 4 (37) (2019) 1450, https://doi.org/10.21105/joss.01450.
[12] M. Musy, G. Dalmasso, Bane Sullivan, marcomusy/vtkplotter: vtkplotter, https://doi.org/10.5281/ZENODO.2561402, 2019.
[13] Tecplot data format, online, http://paulbourke.net/dataformats/tp/.
[14] S. Legensky, D. Edwards, R. Bush, D. Poirier, C. Rumsey, R. Cosner, C. Towne, CFD general notation system (CGNS) - status and future directions, in: 40th AIAA Aerospace Sciences Meeting & Exhibit, American Institute of Aeronautics and Astronautics, 2002.
[15] Pyqt5 reference guide, https://www.riverbankcomputing.com/static/Docs/PyQt5.
[16] R. Galloway, QrangeSlider, rsgalloway.github.com/qrangeslider/, 2011.
[17] T.D. Economon, F. Palacios, S.R. Copeland, T.W. Lukaczyk, J.J. Alonso, SU2: an open-source suite for multiphysics simulation and design, AIAA J. 54 (3) (2016) 828–846.