Scientific Data Analysis and Visualization with Python, VTK, and ParaView

Cory Quammen[‡][*]

https://www.youtube.com/watch?v=8ugmkKaYKxM

Abstract—VTK and ParaView are leading software packages for data analysis and visualization. Since their early years, Python has played an important role in each package. In many use cases, VTK and ParaView serve as modules used by Python applications. In other use cases, Python modules are used to generate visualization components within VTK. In this paper, we provide an overview of Python integration in VTK and ParaView and give some concrete examples of usage. We also provide a roadmap for additional Python integration in VTK and ParaView in the future.

Index Terms—data analysis, scientific visualization, VTK, ParaView

Introduction

The Visualization Toolkit (VTK) is an open-source, freely available software system for 3D visualization. It consists of a set of C++ class libraries and bindings for Python and several other languages. VTK supports a wide variety of visualization algorithms for 2D and 3D scalar, vector, tensor, and volumetric data, as well as advanced algorithms such as implicit modeling, polygon reduction, mesh smoothing, cutting, contouring, and Delaunay triangulation. VTK has an extensive information visualization framework and a suite of 3D interaction widgets. The toolkit supports parallel processing and integrates with various GUI toolkits such as Qt. Python bindings expose nearly all VTK classes and functions, making it possible to write full VTK-based applications exclusively in Python. VTK also includes interfaces to popular Python packages such as NumPy and matplotlib. Support for writing custom VTK algorithms in Python is also available.

ParaView is a scalable visualization tool based on VTK that runs on a variety of platforms ranging from PCs to some of the largest supercomputers in the world. The ParaView package consists of a suite of executables for generating data visualizations using the techniques available in VTK. ParaView executables interface with Python in a number of ways: data sources, filters, and plots can be defined via Python code, data can be queried with Python expressions, and several executables can be controlled interactively with Python commands. Batch processing via Python scripts that are written either by hand or generated as a trace of events during an interactive visualization session is available for offline visualization generation.

* Corresponding author: cory.quammen@kitware.com
‡ Kitware, Inc.

This paper is organized into two main sections. In the first section, I introduce basic VTK usage, describe the relationship between VTK and Python, and describe interfaces between the two. In the second section, I detail the relationship between ParaView and Python. Examples of Python usage in VTK 6.2 and ParaView 4.3 are provided throughout. I also provide a roadmap for additional Python support in VTK and ParaView.

Python and VTK

VTK Data Model

To understand Python usage in VTK, it is important to understand the VTK data and processing models. At the most basic level, data in VTK is stored in a data object. Different types of data objects are available including graphs, trees, and data sets representing spatially embedded data from sensors or simulations such as uniform rectilinear grids, structured/unstructured grids, and Adaptive Mesh Refinement (AMR) data sets. This paper focuses on spatially embedded data sets.

Each spatially embedded data set consists of *cells*, each of which defines a geometric entity that defines a volume of space, and *points* that are used to define the vertices of the cells. Data values that represent a quantity, e.g. pressure, temperature, velocity, may be associated with both cells and points. Each quantity might be a scalar, vector, tensor, or string value. Vectors and tensors typically have more than one numerical *component*, and the quantity as a whole is known as a *tuple*.

The full collection of a quantity associated with points or cells is known by a number of names including "attribute", "field", "variable", and "array". VTK stores each attribute in a separate data array. For a point-associated array (point array), the number of tuples is expected to match the number of points. Likewise, for cell-associated arrays (cell array) the number of tuples is expected to match the number of cells.

VTK Pipeline

Data processing in VTK follows the data-flow paradigm. In this paradigm, data flows through a sequence of processing algorithms. These algorithms are chained together in a *pipeline*.

At the beginning of a pipeline, a *source* generates a VTK data set. For example, an STL file reader source reads an STL file and produces a polygonal VTK data set as an output. A *filter* can be connected to the file reader to process the raw data from the file. For example, a smoothing filter may be used to smooth the polygonal data read by the STL reader. The output of the smoothing filter can be further processed with a clipping filter to cut away part of the smoothed data set. Results from this operation can then be saved to a file with a file writer.

An algorithm in a pipeline produces one or more VTK data sets that are passed to the next algorithm in the pipeline. Algorithms need only update when one of their properties changes (e.g., smoothing amount) or when the algorithm upstream of it has produced a new data set. These updates are handled automatically by an internal VTK pipeline executive whenever an algorithm is updated.

Because VTK is intended to produce 3D interactive visualizations, output from the final algorithm in a pipeline is typically connected to a *mapper* object. A mapper is responsible for converting a data set into a set of rendering instructions. An *actor* represents the mapper in a scene, and has some properties that can modify the appearance of a rendered data set. One or more actors can be added to a *renderer* which executes the rendering instructions to generate an image.

Python Language Bindings for VTK

Since 1997, VTK has provided language bindings for Python. Over the years, Python has become increasingly important to VTK, both as a route to using VTK, as well as to the development of VTK itself.

The Python binding support in VTK has evolved so that today nearly every semantic feature of C++ used by VTK has a direct semantic analog in Python. C++ classes from VTK are wrapped into Python equivalents. The few classes that are not wrapped are typically limited to classes that are meant for internal use in VTK.

Python Wrapping Infrastructure

Python classes for VTK classes and special types are generated using a shared lex/yacc-based parser tailored for VTK programming conventions and custom code generation utilities for Python wrapping. VTK is organized into a number of C++ modules. When built with shared libraries enabled, a library containing C++ classes is generated at build time for each C++ module. Each Python-wrapped source file is likewise compiled into a shared library corresponding to the C++ module. All wrapped VTK C++ modules are provided in a single `vtk` Python package.

VTK Usage in Python

For convenience, an executable named `vtkpython` is provided in VTK binaries. This is the standard Python executable with environment variables set to make it simple to import the `vtk` package. It is also possible to use VTK in the same `python` executable from the Python installation against which VTK was built by prepending the location of VTK's shared libraries and the location of the parent directory of the file `vtk/__init__.py` to the `PYTHONPATH` environment variable, but using `vtkpython` avoids the need to do this.

To access VTK classes, you simply import `vtk`:

```
import vtk
```

VTK is somewhat unusual for a Python package in that all modules are loaded by this import statement.

Creation of VTK objects is straightforward:

```
contourFilter = vtk.vtkContourFilter()
```

Each Python object references an underlying VTK object. Objects in VTK are reference counted and automatically deleted when no longer used. The wrapping interface updates the underlying VTK object's reference count and alleviates the need for explicit memory management within Python.

One particularly nice semantic equivalence between VTK's C++ and Python interfaces involves member functions that accept a pointer to a C++ array representing a small tuple of elements. Such functions are common in VTK to do things like set a 3D Cartesian coordinate as a property of a class. In Python, the corresponding function accepts a tuple or list object. This works well as long as the list or tuple has the expected number of elements.

```
sphere = vtk.vtkSphereSource()

# Express point as list
sphere.SetCenter([0, 1, 0])

# Express point as tuple
sphere.SetCenter((0, 1, 0))
```

Member functions that return pointers to arrays with a fixed number of elements are also supported. Such functions require a hint to the wrapping infrastructure indicating how many elements are in the tuple that is returned.

```
>>> center = sphere.GetCenter()
>>> print center
(0, 1, 0)
```

For VTK classes that have operators <, <=, ==, >=, > defined, equivalent Python operators are provided.

Some functions in VTK return information via parameters passed by reference. For example, in the following code block, the parameter `t` is a return parameter from the member function `IntersectWithLine`.

```
double t, x[3]
plane->IntersectWithLine(point1, point2, t, x);
```

In Python, the equivalent is

```
t = vtk.mutable(0.0)
plane.IntersectWithLine(point1, point2, t, x)
```

Class and function documentation is processed by the wrapping infrastructure to make it available via Python's built-in help system.

```
>>> help(vtk.vtkSphereSource)
```

The above shows the full documentation of the `vtkSphereSource` class (too extensive to list here), while the code below produces help for only the `SetCenter` member function.

```
>>> help(vtk.vtkSphereSource.SetCenter)

Help on built-in function SetCenter:
```

```
SetCenter(...)
    V.SetCenter(float, float, float)
    C++: void SetCenter(double, double, double)
    V.SetCenter((float, float, float))
    C++: void SetCenter(double a[3])
```

Some less often used mappings between C++ and Python semantics, as well as limitations, are described in the file `VTK/Wrapping/Python/README_WRAP.txt` in the VTK source code repository in versions 4.2 and above.

A full example below shows how to create a VTK pipeline in Python that loads an STL file, smooths it, and displays the smoothed result in a 3D render window.

```
import vtk

reader = vtk.vtkSTLReader()
reader.SetFileName('somefile.stl')

smoother = vtk.vtkLoopSubdivisionFilter()
smoother.SetInputConnection(reader.GetOutputPort())

mapper = vtk.vtkPolyDataMapper()
mapper.SetInputConnection(smoother.GetOutputPort())

actor = vtk.vtkActor()
actor.SetMapper(mapper)

renderer = vtk.vtkRenderer()
renderer.AddActor(actor)

renWin = vtk.vtkRenderWindow
renWin.AddRenderer(renderer)

interactor = vtk.vtkRenderWindowInteractor()
interactor.SetRenderWindow(renWin)
interactor.Initialize()
renWin.Render()
iren.Start()
```

Many additional examples of VTK usage in Python are available in the VTK/Examples/Python wiki page [Wik15].

Integration with NumPy

There are limited functions within VTK itself to process or analyze point and cell arrays. Since 2008, a low-level interface layer between VTK arrays and NumPy arrays has been available in VTK. This interface layer can be used to map VTK arrays to NumPy arrays and vice versa, enabling the full power of NumPy operations to be used on VTK data. For example, suppose that we have a data set from a computational fluid dynamics simulation that we can load with a VTK reader class, and suppose further that the data set has a point array representing pressure. We can find several properties of this array using NumPy, e.g.,

```
import numpy as np
import vtk.util.numpy_support as nps

# Load data with a VTK reader instantiated earlier
reader.Update()

ds = reader.GetOutput()
pd = ds.GetPointData()
pressure = pd.GetArray('pressure')
np_pressure = nps.vtk_to_numpy(pressure)

min_p = np.min(np_pressure)
max_p = np.max(np_pressure)
```

This interface can also be used to add data arrays to loaded data sets that can be handed off to VTK for visualization:

```
norm_pressure = (np_pressure - min_pressure) / \
    (max_pressure - min_pressure)
vtk_norm_pressure = np.numpy_to_vtk(norm_pressure, 1)
vtk_norm_pressure.SetName('normalized pressure')
pd.AddArray(vtk_norm_pressure)
```

The second argument to `np.numpy_to_vtk` indicates that the NumPy array should be deep copied to the VTK array. This is necessary if no reference to the NumPy array will otherwise be kept. If a reference to the numpy array will be kept, then the second argument can be omitted and the NumPy array will be shallow copied instead, saving memory and time because the array data does not need to be copied. Note that the Python interpretter might crash if a NumPy array reference is not held and the data is shallow copied.

More recently, a higher-level NumPy-like interface layer has been added to VTK. This `numpy_interface` was designed to combine the ease of use of NumPy with the distributed memory parallel computing capabilities and broad data set type support of VTK. The straightforward interface between VTK data arrays and NumPy described above works only when the entire data set is available on one node. However, data sets in VTK may be distributed across different computational nodes in a parallel computer using the Message Passing Interface [Sni99]. In this scenario, global reduction operations using NumPy are not possible. For this reason, a NumPy-like interface has been added to VTK that properly handles distributed data sets [Aya14].

A key building block in VTK's `numpy_interface` is a set of classes that wrap VTK data set objects to have a more Pythonic interface.

```
import vtk
from vtk.numpy_interface import dataset_adapter as dsa

reader = vtk.vtkXMLPolyDataReader()
reader.SetFileName(filename)
reader.Update()
ds = dsa.WrapDataObject(reader.GetOutput())
```

In this code, `ds` is an instance of a `dataset_adapter.PolyData` that wraps the `vtkPolyData` output of the `vtkXMLPolyDataReader`. Point and cell arrays are available in member variables `PointData` and `CellData`, respectively, that provide the dictionary interface.

```
>>> ds.PointData.keys()
['pressure']

>>> pressure = ds.PointData['pressure']
```

Note that the `pressure` array here is an instance of `VTKArray` rather than a wrapped VTK data array. `VTKArray` is a wrapper around the VTK array object that inherits from `numpy.ndarray`. Hence, all the standard `ndarray` operations are available on this wrapped array, e.g.,

```
>>> pressure[0]
0.112

>>> pressure[1:4]
VTKArray([34.2432, 47.2342, 38.1211], dtype=float32)

>>> pressure[1:4] + 1
```

```
VTKArray([35.2432, 48.2342, 39.1211], dtype=float32)

>>> pressure[pressure > 40]
VTKArray([47.2342], dtype=float32)
```

The `numpy_interface.algorithms` module also provides NumPy-like functionality:

```
import vtk.numpy_interface.algorithms as algs

>>> algs.min(pressure)
VTKArray(0.1213)

>>> algs.where(pressure > 38)
(array([2, 3], dtype=int64),)
```

In addition to providing most of the ufuncs provided by NumPy, the `algorithms` interface provides some functions to access quantities that VTK can compute in the wide variety of data set types available in VTK. This can be used to compute, for instance, the total volume of cells in an unstructured grid:

```
>>> cell_volumes = algs.volume(ds)
>>> algs.sum(cell_volumes)
VTKArray(847.02)
```

This example illustrates nicely the power of combining a NumPy-like interface with VTK's uniform API for computing various quantities on different types of data sets.

Another distinct advantage of the `numpy_interface.algorithms` module is that all operations are supported in parallel when data sets are distributed across computational nodes. [Aya14] describes this functionality in more detail.

Integration with matplotlib

While VTK excels at interactive 3D rendering of scientific data, matplotlib excels at producing publication-quality 2D plots. VTK leverages each toolkit's strengths in two ways.

First, as described earlier, convenience functions for exposing VTK data arrays as NumPy arrays are provided in the `vtk.util.numpy_support` and `numpy_interface.algorithms` modules. These arrays can be passed to matplotlib plotting functions to produce publication-quality plots.

Second, VTK itself incorporates some of matplotlib's rendering capabilities directly when possible. When VTK Python wrapping is enabled and matplotlib is available, VTK uses the `matplotlib.mathtext` module to render LaTeX math expressions to either `vtkImageData` objects that can be displayed as images or to paths that may be rendered to a `vtkContextView` object, VTK's version of a canvas. The `vtkTextActor`, a class for adding text to visualizations, uses this module to support rendering complex LaTeX math expressions.

Qt applications with Python

Python support in VTK is robust enough to create full-featured applications without writing a single line of C++ code. PyQt [PyQt15] (or PySide [PyS15]) provide Python bindings for Qt. A simple PyQt example adapted from an example by Michka Popoff is provided below:

```
import sys
import vtk
from PyQt4 import QtCore, QtGui
from vtk.qt4.QVTKRenderWindowInteractor \
    import QVTKRenderWindowInteractor

class MainWindow(QtGui.QMainWindow):

    def __init__(self, parent = None):
        QtGui.QMainWindow.__init__(self, parent)

        self.frame = QtGui.QFrame()

        layout = QtGui.QVBoxLayout()
        self.vtkWidget = \
            QVTKRenderWindowInteractor(self.frame)
        layout.addWidget(self.vtkWidget)

        self.renderer = vtk.vtkRenderer()
        rw = self.vtkWidget.GetRenderWindow()
        rw.AddRenderer(self.renderer)
        self.interactor = rw.GetInteractor()

        cylinder = vtk.vtkCylinderSource()
        mapper = vtk.vtkPolyDataMapper()
        mapper.SetInputConnection( \
            cylinder.GetOutputPort())
        actor = vtk.vtkActor()
        actor.SetMapper(mapper)

        self.renderer.AddActor(actor)
        self.renderer.ResetCamera()

        self.frame.setLayout(layout)
        self.setCentralWidget(self.frame)

        self.show()
        self.interactor.Initialize()

if __name__ == "__main__":
    app = QtGui.QApplication(sys.argv)
    window = MainWindow()
    sys.exit(app.exec_())
```

This simple application does little besides what is possible with pure VTK code alone. However, this example can easily be expanded to provide interaction through UI elements such as a menu bar, buttons, text entries, sliders, etc.

VTK filters defined in Python

While VTK sources and filters are available in Python, they cannot be subclassed to create new sources or filters because the virtual function table defined in C++ cannot dispatch to member functions defined in Python. Instead, one can subclass from a special `VTKAlgorithm` class defined in `vtk.util.vtkAlgorithm`. This class specifies the interface for classes that interact with `vtkPythonAlgorithm`, a C++ class that delegates the primary VTK pipeline update functions to equivalent pipeline update functions in the Python `VTKAlgorithm` class. Subclasses of `VTKAlgorithm` can (and usually should) override these functions. By doing this, it is possible to implement complex new sources and filters using Python alone. For more details on the `VTKAlgorithm` class, see [Gev2014].

Python integration in VTK tests

As a project that follows a quality software process, VTK has many regression tests. At present, 26% of tests (544 out of 2046) are written in Python. This integration of Python in

VTK's testing infrastructure shows how important Python is in VTK's development.

Obtaining VTK

VTK and its Python bindings are available on many Linux distributions including Ubuntu, Debian, OpenSUSE. It is also available in Anaconda and Enthought Canopy. Binary installers and source code for the most recent versions are available on the VTK web site [VTK15] for Windows, Mac, and Linux.

Python and ParaView

ParaView is a suite of scalable parallel visualization executables that use VTK to read data, process it, and create visualizations. One of the executables includes a graphical user interface (GUI) to make it possible to create visualizations without programming (when ParaView is mentioned in this section, it is the executable with a GUI unless otherwise specified). Data processing in ParaView follows the same dataflow paradigm that VTK follows. In ParaView, sources and filters are chained together in a Pipeline Browser as shown in Figure 1. Visualization controls are modified with user interaction widgets provided by Qt.

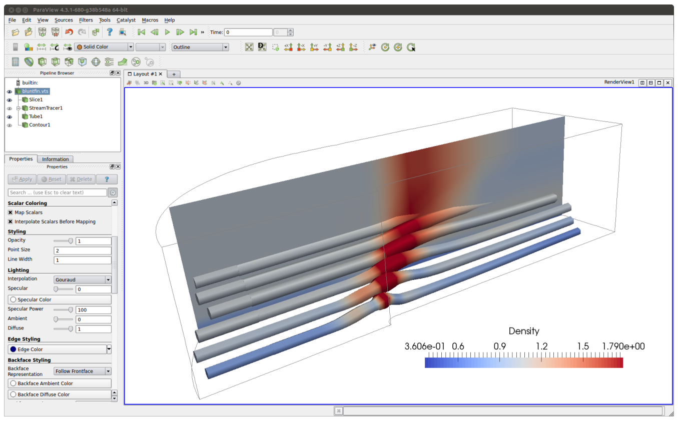


Fig. 1: The ParaView GUI with an example visualization of a data set from a simulation of airflow past a blunt fin. The Pipeline Browser (upper left) shows the sources and filters used to create the visualization. Filter and visualization parameters are shown in the Property window (lower left).

While ParaView can be used to make visualizations without programming, it is also possible to use Python scripting to automate certain operations or even create entire visualizations. In this section, I describe how Python scripting is integrated into ParaView at several different levels. At a high level, Python commands are issued via a console to change properties of a visualization. At a lower level, Python commands are used to set up entire visualizaion pipelines. At an even lower level, Python is used to create custom sources and filters to provide additional data analysis and visualization functionality.

Python Console

ParaView includes a Python console available under the `Tools -> Python Console` menu item. This console is a fully-featured Python console with the environment set up so that the `vtk` package and a `paraview` package are available. When first started, the command

```
from paraview.simple import *
```

is automatically executed to import the `paraview.simple` module. This layer is described in more detail later.

Running commands in ParaView's Python console is identical to running commands in other Python consoles. The key difference is that commands can be used to change the state of the ParaView application. This provides a similar experience to using a Python console to change matplotlib plots.

The Python console also provides a button to load and execute a Python script with ParaView commands from a file. This feature is ideal for iterative Python script development.

pvpython and pvbatch

The ParaView suite of tools includes two Python-based utilities for both interactive and batch generation of visualizations. `pvpython` is an interactive Python shell that provides the same access to the `vtk` and `paraview` packages as provided by the Python console in ParaView. The key difference between ParaView and `pvpython` is that no GUI controls are available to modify pipeline or visualization state. `pvbatch` is a non-interactive executable that runs a Python script and is intended to perform offline data processing and visualization generation.

Python Tracing and State Files

While documentation is available to learn how to write Python scripts for ParaView, it can take some time to find the function calls needed to replicate a sequence of actions performed through the GUI. To reduce script development time, ParaView supports tracing of user interactions where the generated trace is in the form of a Python script. Running the resulting trace script through the ParaView Python console, `pvpython` or `pvbatch` reproduces the effects of the user interactions with the GUI.

Python tracing is implemented by instrumenting the ParaView application with Python generation code at various user event handlers. The tracing mechanism can record either the entire state of ParaView objects or just modifications of state to non-default values to reduce the trace size. Traces can be started and stopped at any time - they do not need to record the full user interaction history.

An application where tracing is useful is the batch conversion of data files. If ParaView can read the source file format and write the destination file format, it is easy to perform the conversion manually one time with the ParaView GUI. For a large list of files, though, a more automated approach is useful. Creating a trace of the actions needed to perform the conversion of a single file produces most of the script that would be needed to convert a list of files. The trace script can then be changed to apply to a list of files.

In addition to saving a trace of user interaction sequences, a Python *state file* may also be produced. Like a Python trace, the state file contains Python commands that set up the pipeline and visualization settings, but unlike a trace, it does not record interaction events as they happen but rather the final state of ParaView.

Simple Python Interface

Much of ParaView is implemented in C++ as VTK classes. These classes are wrapped in Python with the same mechanism that wraps VTK classes. As such, they are accessible within the Python console, `pvpython`, and `pvbatch`. However using these classes directly is often unwieldy. The example below illustrates how to use the direct ParaView API to create a sphere source with radius 2.

```
from paraview import servermanager as sm

pm = sm.vtkSMProxyManager.GetProxyManager()
controller = \
    sm.vtkSMParaViewPipelineControllerWithRendering()

ss = pm.NewProxy('sources', 'SphereSource')
ss.GetProperty('Radius').SetElement(0, 2.0)
controller.RegisterPipelineProxy(ss)

view = pm.GetProxy('views', 'RenderView1')
rep = view.CreateDefaultRepresentation(ss, 0)
controller.RegisterRepresentationProxy(rep)
rep.GetProperty('Input').SetInputConnection(0, ss, 0)
rep.GetProperty('Visibility').SetElement(0, 1)

controller.Show(ss, 0, view)
view.ResetCamera()
view.StillRender()
```

Note in this example the various references to proxies. A *proxy* here refers to the proxy programming design pattern where one object provides an interface to another object. Proxies are central to ParaView's design. In a number of the various client/server configuration in which ParaView can be run, the client software running on a local workstation connects to a remote server running one or more processes on different nodes of a high-performance computing resource. Proxies for each pipeline object exist on the ParaView client, and they provide the interface for communicating state to all the VTK objects in each client and server process.

In the example above, a new proxy for a `vtkSphereSource` object is created. This proxy has a property named 'Radius' that is modified to the value 2.0. Changes to the 'Radius' property are forwarded to the 'Radius' property of the underlying `vtkSphereSource`.

As this example demonstrates, creating a new data source, a representation for it (how it is rendered), and adding the representation to the view (where it is rendered), is an involved process when using the `paraview.servermanager` module directly. Fortunately, ParaView provides a simplified Python interface that hides most of these details, making Python scripting much more accessible.

The `paraview.simple` layer provides simpler Python functions to create pipelines and modify filter and visualization properties. The same example above expressed with `paraview.simple` functions is reduced to

```
from paraview import simple

Sphere(Radius=2.0)
Show()
Render()
```

ParaView traces and Python state files are expressed in terms of `paraview.simple` module functions. For more information on how to use this module, see [Kit15].

Python Programmable Filter

ParaView provides many data filters for transforming data and performing analysis tasks. There are, however, an infinite number of operations one may want to perform on a data set. To address the need for custom filters, ParaView supports a rich plugin architecture that makes it possible to create additional filters in C++. Unfortunately, creating a plugin this way is a relatively involved process.

Aside from the C++ plugin architecture, ParaView provides a Programmable Filter that enables a potentially faster development path. The Programmable Filter has a text property that stores a Python script to execute when the filter is updated. Inputs to the Programmable Filter are available within this script. Complete specification of the output data set is possible within the script, including setting the output data type, the data set toplogy (i.e., type and number of cells), as well as point and cell arrays.

At its core, the Programmable Filter is defined by the VTK-derived C++ class named `vtkPythonProgrammableFilter`. Using the Python C API, the `vtkPythonProgrammableFilter` passes a reference to itself to the Python environment in which the script executes so that it is available within the script itself. This makes it possible to access the inputs and outputs to the filter via:

```
input = self.GetInput()
output = self.GetOutput()
```

Arbitrarily complex Python scripts can be executed to generate the filter's output. The following example moves points in an input `vtkPointSet` along normals associated with the points if available.

```
ipd = self.GetInput()
opd = self.GetOutput()

# Output is shallow-copied by default
# Deep copy the points so that we are not modifying
# the input points.
opd.DeepCopy(ipd)

na = ipd.GetPointData().GetArray('Normals')
if na != None:
    for i in xrange(ipd.GetNumberOfPoints()):
        pt = ipd.GetPoint(i)
        n = na.GetTuple(i)
        newPt = (pt[0]+n[0], pt[1]+n[1], pt[2]+n[2])
        opd.GetPoints().SetPoint(i, newPt)
```

The Programmable Filter also uses the `vtk.numpy_interface.dataset_adapter` module to wrap the inputs to the filter. All of the wrapped inputs are added to a list named `inputs`, and the single output is wrapped in an object named `output`. By using the wrapped inputs and outputs, the filter above becomes simply

```
ipts = inputs[0].Points
normals = inputs[0].PointData['Normals']

output.Points = ipts + normals
```

It is important to note that Python scripts in the Programmable Filter may use only VTK classes and other Python modules, but not any of the modules in the `paraview` package. If those modules are imported, the behavior is undefined.

Python Programmable Source

Within ParaView it is also possible to define Python script that defines data sources using the Python Programmable Source. This source functions much like the Python Programmable Filter, but does not require any input data sets.

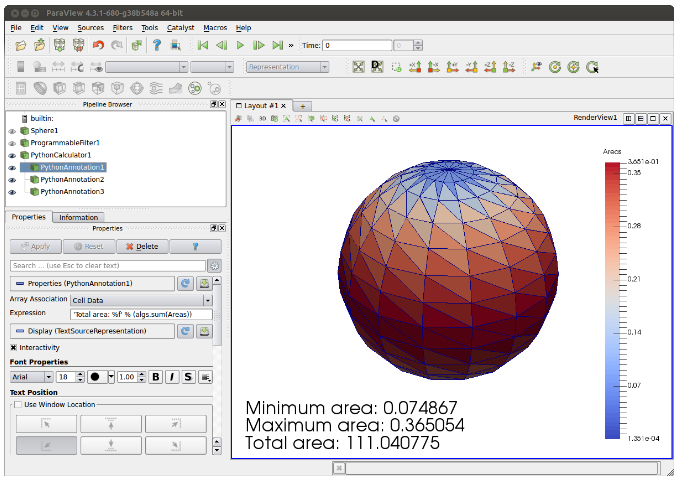
Python Calculator

ParaView's Python Calculator filter is a light-weight alternative to the Programmable Filter used to compute additional point or cell arrays using NumPy or the `numpy_interface.algorithms` module. The following expression computes the areas of polygons in a surface mesh:

```
algs.area(inputs[0])
```

Note that the `numpy_interface.algorithms` is imported with the name `algs` in the Python environment in which the expression is evaluated. In the Python Calculator, the property 'Array Association', which indicates whether the output array should be a point or cell array, must be set to 'Cell Data' because one area value is produced per cell. Note that like the Programmable Filter, the inputs are wrapped with the `vtk.numpy_interface.dataset_adapter` module functions and stored in an `inputs` list.

Python Annotation


ParaView 4.3.1-680-g38b548a 64-bit

Pipeline Browser: builtin:, Sphere1, ProgrammableFilter1, PythonCalculator1, PythonAnnotation1, PythonAnnotation2, PythonAnnotation3

Properties (PythonAnnotation1)
Array Association: Cell Data
Expression: 'Total area: %f' % (algs.sum(Areas))
Display (TextSourceRepresentation)
Interactivity: ✓
Font Properties: Arial, 18, 1.00
Text Position: Use Window Location ☐

Minimum area: 0.074867
Maximum area: 0.365054
Total area: 111.040775

Fig. 2: Three annotations filters in the scene show the minimum, maximum, and total areas of polygons in the sphere source.

It is often desirable to annotate visualizations with numerical values taken either directly from the data set or computed from the data. The Python Annotation filter in ParaView provides this capability in a convenient way. The filter takes a Python expression that is evaluated when the filter is executed and the value returned by the expression is displayed in the render view. Importantly, these annotations can come from data analysis results from NumPy or `numpy_interface.algorithms`. Figure 2 shows an example using the Python Annotation filter.

Python View

While ParaView's roots are in the loading and display of traditional 3D scientific visualizations, it has grown over the years to support more data set types and different displays of those data set types. These different displays, or "Views" in ParaView parlance, include a 3D interactive rendering view, a histogram view, a parallel coordinates view, and a large number of others.

One of these other view types is the Python View. This view is similar to the programmable filter in that the user supplies a Python script that generates some data. In the case of the Python View, the data that is generated is an image to display in the ParaView window. This makes it possible to use Python plotting packages, such as matplotlib, to generate plots to be displayed directly in ParaView.

Scripts used in the Python view are required to define two functions, a `setup_data` function and a `render` function. Rendering in the Python view is done on the local client, so data that resides on remote server processes must first be brought over to the client. Because data sets may be larger than the client's RAM, only a subset of the data arrays in a data set are copied to the client. By default, no arrays are copied. Arrays can be requested using functions available in the `vtkPythonView` class instance that is passed in as an argument to the `setup_data` function, e.g.,

```python
def setup_data(view):
    view.SetAttributeArrayStatus(0, \
        vtkDataObject.POINT, "Density", 1)
```

The actual generation of the plot image is expected to be done in the `render` function. This function is expected to take the same `view` object as is passed to the `setup_data` function. It also takes a width and height parameter that tells how large the plotted image should be in terms of pixels. This function is expected to return an instance of `vtkImageData` containing the plot image. A few utilities are included in the `paraview.python_view` module to convert Python arrays and images to `vtkImageData`. An example that creates a histogram of an array named "Density" is provided here:

```python
def render(view, width, height):
    from paraview \
        import python_view.matplotlib_figure
    figure = matplotlib_figure(width, height)

    ax = figure.add_subplot(1,1,1)
    ax.minorticks_on()
    ax.set_title('Plot title')
    ax.set_xlabel('X label')
    ax.set_ylabel('Y label')

    # Process only the first visible object in the
    # pipeline browser
    do = view.GetVisibleDataObjectForRendering(0)

    dens = do.GetPointData().GetArray('Density')

    # Convert VTK data array to numpy array
    from paraview.numpy_support import vtk_to_numpy

    ax.hist(vtk_to_numpy(dens), bins=10)

    return python_view.figure_to_image(figure)
```

For more information on the Python View, see Section 4.11 in [Aya15] or [Qua13].

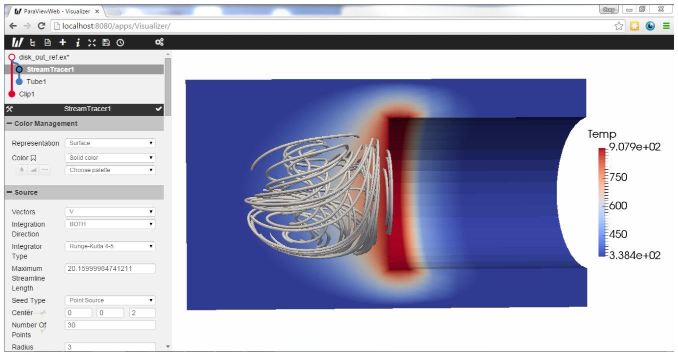
ParaViewWeb

ParaViewWeb is a framework for remote VTK and ParaView processing and visualization via a web browser. The framework on the server side is based on the Autobahn, Twisted, Six, and ZopeInterface Python libraries. On the client side, ParaViewWeb provides a set of JavaScript libraries that use WebGL, JQuery, and Autobahn.js. Images are typically generated on the server and sent to the client for display, but if the visualized geometry is small enough, geometry can be sent to the client and rendered with WebGL.

A nice feature of ParaViewWeb is that the server component can be launched with `pvpython`. No separate web server is needed. For example, on Linux, the following command launches the ParaViewWeb server from the ParaView installation directory

```
./bin/pvpython                                \
    lib/paraview-4.1/site-packages/paraview/\
    web/pv_web_visualizer.py --port 8080      \
        --content ./share/paraview-4.1/www \
        --data-dir /path-to-share/ &          \
```

Once the server is running, it can be accessed through a web browser at the URL http://localhost:8080/apps/Visualizer. This is one example application that comes with the framework. It has much of the same functionality as the ParaView desktop application. ParaViewWeb can also be used to display images within an iPython notebook. For additional information about using and extending the ParaViewWeb framework, see [Pvw15].



disk_out_ref.ex*
StreamTracer1
Tube1
Clip1

StreamTracer1

Color Management
Representation: Surface
Color: Solid color
Choose palette

Source
Vectors: V
Integration Direction: BOTH
Integrator Type: Runge-Kutta 4-5
Maximum Streamline Length: 20.15999984741211
Seed Type: Point Source
Center: 0 0 2
Number Of Points: 30
Radius: 3

Fig. 3: *The ParaViewWeb Visualizer application web interface.*

Unified Server Bindings

As previously discussed, ParaView uses proxies to manage state among VTK class instances associated with pipeline objects on distributed process. For example, when the proxy for a cross-section filter has its cutting plane property changed, the underlying VTK filter on each process is updated so that is has the same cutting plane. These instances are updated via a client/server communication layer that is generated automatically using a wrapping mechanism. The client/server layer consists of one communication class per VTK class that serializes and deserializes state in the VTK class.

As discussed, a similar wrapping process is also performed to generate Python bindings for VTK classes and ParaView classes. Each of these wrappings adds to the size of the executable files and shared libraries. On very large scale parallel computing resources, the amount of RAM available per node can be relatively limited. As a result, when running ParaView on such a resource, it is important to reduce the size of the executables as much as possible to leave room for the data. One way to do this is to use the Python wrapping to communicate among processes instead of using the client/server communication class. Indeed, when this option is enabled, the process of creating the special communication classes is skipped. Instead, communication is performed by sending strings with Python expressions to destination processes. These expressions are then evaluated on each process to change the state of local VTK classes. In this approach, we get the same functionality as the custom client/server communication layer wrapping, but with smaller executables.

Conclusions

Python has been integrated into VTK and ParaView for many years. The integration continues to mature and expand as Python is used in an increasing number of ways in both software packages. As Python continues to grow in popularity among the scientific community, so too does the need for providing easy-to-use Pythonic interfaces to scientific visualization tools. As demonstrated in this paper, VTK and ParaView are well-positioned to continue adapting to the future needs of scientific Python programmers.

Future Work

VTK and ParaView currently support Python 2.6 and 2.7. Support for Python 3 is targeted for sometime in 2016.

Acknowledgements

Contributions to Python support in VTK and ParaView have come from many VTK community members. Deserving special recognition are key contributors David Gobbi, Prabhu Ramachandran, Ken Martin, Berk Geveci, Utkarsh Ayachit, Ben Boeckel, Andy Cedilnik, Brad King, David Partyka, George Zagaris, Marcus Hanwell, and Mathieu Malaterre.

References

[Aya14] U. Ayachit, B. Geveci, *Scientific data analysis and visualization at scale in VTK/ParaView with NumPy*, 4th Workshop on Python for High Performance and Scientific Computing PyHPC 2014, November, 2014.

[Aya15] U. Ayachit, *The ParaView Guide: A Parallel Visualization Application*, Kitware, Inc. 2015, ISBN 978-1930934306.

[Gev14] B. Geveci, *vtkPythonAlgorithm is great*, Kitware Blog, September 10, 2014. http://www.kitware.com/blog/home/post/737

[Kit15] *simple Module*, http://www.paraview.org/ParaView/Doc/Nightly/www/py-doc/paraview.simple.html

[Pvw15] *ParaViewWeb*, http://paraviewweb.kitware.com/#!/guide

[PyQt15] *PyQt4 Reference Guide*, http://pyqt.sourceforge.net/Docs/PyQt4/

[PyS15] *PySide 1.2.2*, https://pypi.python.org/pypi/PySide

[Qua13] C. Quammen. *ParaView: Python View is now more versatile*, http://www.kitware.com/blog/home/post/704

[Sch04] W. Schroeder, K. Martin, and B. Lorensen, *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*, 4th ed. Kitware, Inc., 2004, ISBN 1-930934-19-X.

[Sni99] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI - The Complete Reference: Volume 1, The MPI Core*, 2nd ed., MIT Press, 1999, ISBN 0-262-69215-5.

[VTK15] *VTK - The Visualization Toolkit*, http://www.vtk.org/

[Wik15] *VTK/Examples/Python*, http://www.vtk.org/Wiki/VTK/Examples/Python